1 Vorwort

1.1 Motivation

Von Computern war ich schon sehr früh fasziniert. Ich interessierte mich für alles, was mit Computern zusammenhing und kam zum ersten Mal mit programmieren in Kontakt, als ich bemerkte, dass man in einem Strategiespiel eine eigene „Mission" programmieren konnte. Das machte mir Spass und der Erfolg liess sich schnell sehen, was mich sehr motivierte. Natürlich waren die Programmiermöglichkeiten eines sogenannten „Worldeditors" sehr beschränkt und auf das allgemeine Spielprinzip ausgerichtet. Damit konnte man nicht gleich ein eigenes Spiel programmieren.
Deshalb hatte ich mir vorgenommen, für die Matura eine echte Programmiersprache zu lernen, mit der man seine eigenen Programme schreiben konnte. In den Physikstunden waren mir schon öfter die Java-Applets aufgefallen. Diese fand ich sehr praktisch, zeigten sie doch auf, wie man sich die graue Theorie in der Realität vorstellen konnte.
So kam ich auf die Idee, für diese Maturaarbeit die Programmiersprache Java zu erlernen und selbst eine Simulation zu programmieren.

1.2 Das Ziel

In dieser Simulation behandle ich einen Fall aus der granulären Physik[2], und zwar mit der Materie Sand. Der Fall beschäftigt sich mit Sand, der auf den Boden geschüttet wird und einen Hügel mit einem gewissen Winkel bildet. Das Ziel hierbei war herauszufinden, wie sich der Winkel verhält. Ich fand dieses Thema interessant, denn der Winkel war ja aufgrund physikalischer Gesetze nicht auszurechnen, und die Programmierung erschien mir anspruchsvoll.

1.3 Programmierung

Der Grossteil meiner Arbeit befasst sich mit der Alghorithmik[3] und deren Umsetzung im Programm. Zuerst erlernte ich die Grundlagen der Programmiersprache Java durch eine Einführung aus dem Internet. Für die Programmierung verwendete ich die Programmierumgebung Eclipse[4]. Java ist eine relativ einfach zu erlernende Programmiersprache. Ich hatte jedenfalls nicht viele Schwierigkeiten einzusteigen, weil ich aus früherer Erfahrung fundamentale Befehle bereits kannte, mit ihnen umgehen konnte und ich mich eigentlich nur an die Prinzipien der Sprache gewöhnen musste.

Ich hoffe, ihnen einen interessanten Einblick in die Welt der Programmierung verschaffen zu können und wünsche ihnen noch viel Spass beim Lesen.

Inhaltsverzeichnis

2 Theorie

2.1 Grundfrage der Maturaarbeit

Diese Maturaarbeit beschäftigt sich mit einem Thema der granulären Physik[2]. Die Frage ist: welchen Winkel nimmt Sand an, wenn er zu einem Haufen aufgeschüttet wird? Und wie verhält sich dieser Sand bei Veränderung der Feuchtigkeit und der Grösse der Sandkörner?

Hierzu habe ich Experimente durchgeführt und diese in einer grafischen Simulation am Computer umgesetzt.

2.2 Wie entstehen verschiedene Winkel?

Nasser Sand ist wesentlich stabiler als trockener.

In dieses Programm habe ich Regler für zwei Faktoren eingebaut, die den Winkel beeinflussen:

- **Feuchtigkeit**
- **Grösse der Sandkörner**

Feuchtigkeit verändert die Stabilität des Sandes und hat so einen direkten Einfluss auf den Winkel. Bis zur Sättigung des Sandes, d.h. solange der Sand nicht zu viel Wasser aufgenommen hat und dadurch eher flüssig wird als stabil, wirkt sich Wasser stabilisierend auf den Sand aus und erhöht den Winkel.

Meine Überlegung zur Grösse der Sandkörner war, das bei grösseren Sandkörnern mehr Oberfläche für die Bindungen des Wassers da ist.

2.3 Berechnung des zu erwartenden Winkels

2.3.1 Ansatz

Bei granulärer Physik ist es nicht möglich, rein mit physikalischen Gesetzen zu rechnen. Ich möchte auch in dieser Maturaarbeit keine allgemeine Formel zur Berechnung eines Winkels aufstellen, sondern viel eher eine Art Annäherung an die Wirklichkeit erreichen, um mein Programm möglichst realistisch wirken zu lassen.

Für mein Programm gehe ich von meinen Experimenten aus, also von dem Winkel, den der Sand trocken einnimmt. Ich beschränke mich bei meiner Theorie bezüglich des Winkels auf die Verhältnismässigkeit: Mit dieser werde ich am Schluss, mithilfe der Experimente, die Winkel berechnen können:

$$\alpha_0 = 36.83\,°$$

$$\alpha_n = \alpha_0 * \frac{\alpha_n}{\alpha_0}$$

$$\frac{\alpha_n}{\alpha_0} = ?$$

Hier ist α_0 der gemessene Winkel von trockenem Sand. Mit α_n ist der Winkel bei einer bestimmten Feuchtigkeit und Grösse gemeint.

Als Ansatz für die Winkelberechnung habe ich mir überlegt, den Haufen als schiefe Ebene

anzusehen, und das einzelne Korn als Objekt, dass an dieser herunterrollt.

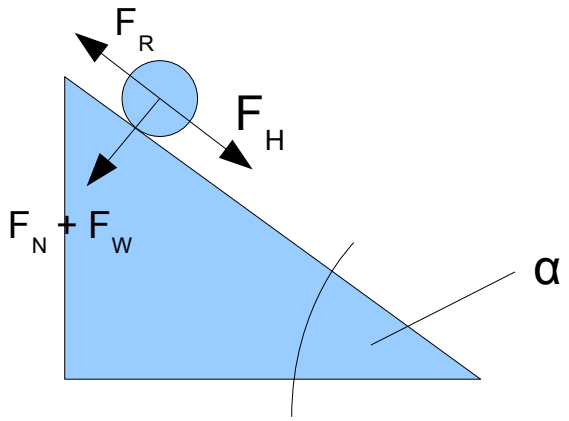

Abbildung 1: Darstellung des Haufens als schiefe Ebene

Dabei gehe ich nun von physikalischen Gesetzen aus: ich betrachte die Hangabtriebskraft F_H und bezeichne die Kraft, die das Korn vom Herunterrollen abhält, als Reibungskraft F_R. Diese ist abhängig von der Feuchtigkeit, dessen Effekt wiederum von der Grösse der Körner abhängig ist.

Die Grundgleichung meiner Überlegungen ist folgende: Wenn die „schiefe Ebene" den maximalen Winkel einnimmt, bei dem das Korn noch gerade nicht herunterfällt, dann sind F_H und F_R gleichwertig:

$$F_H = F_R$$

2.3.2 Einfluss der Feuchtigkeit

Die Hangabtriebskraft eines Körpers ist definiert als:

$$F_H = F_G * \sin(\alpha)$$

während die Reibungskraft wie folgt definiert ist:

$$F_R = \mu * F_N$$

Die Feuchtigkeit hat bis zur Sättigung des Sandes eine stabilisierende Wirkung und mit ihr sind höhere Winkel möglich. Das bedeutet in diesem Fall, dass die Reibungskraft vergrössert wird und somit etwas zu F_N addiert wird. Wenn man diesen Gedanken in die Formel einsetzt, heisst es dann:

$$F_R = \mu * (F_N + F_W)$$

F_W ist hierbei die vom Wasser verursachte Kraft die sich mit der Normalkraft addiert.
Somit lautet nun die ursprüngliche Formel wie folgt:

$$F_G * \sin(\alpha) = \mu * (F_N + F_W)$$

Dabei berechne ich die durch das Wasser ausgelöste Kraft als F_W, also als Erhöhung der Normalkraft, die eine Erhöhung der Reibungskraft verursacht, ein.

Die Gleichung kann umgeformt werden:

$$F_G * \sin(\alpha) = \mu * (F_N + F_W)$$

$$F_G * \sin(\alpha) = \mu * (\cos(\alpha) * F_G + F_W)$$

$$\sin(\alpha)=\mu*(\cos(\alpha)+\frac{F_W}{F_G})$$

$$\sin(\alpha)-\mu*\cos(\alpha)=\mu*\frac{F_W}{F_G}$$

$$\sin(\alpha)-\mu*\sin(90-\alpha)=\mu*\frac{F_W}{F_G}$$

Mithilfe dieser Gleichung lässt sich anhand der Experimentdaten ein Verlauf des Winkels in Relation zur Feuchtigkeit ausrechnen.

2.3.3 Einfluss der Grösse

Je kleiner die Körner sind, umso mehr Freiraum zwischen ihnen kann mit dem Wasser verbunden werden. Je kleiner also das einzelne Korn, umso höher die Kraft, die das Wasser zur Normalkraft addiert. Diesen Sachverhalt habe ich mir so vorgestellt:

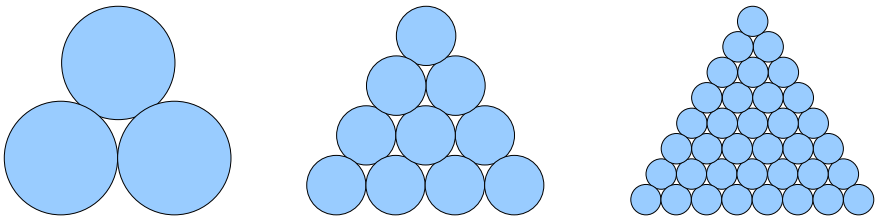

Abbildung 2: Das Volumen ist zwar gleich, aber je kleiner die Körner umso mehr Bindungen entstehen durch das Wasser zwischen den Körnern

Je mehr Flächeninhalt der Sand hat, umso mehr Einfluss hat das Wasser. Daraus folgere ich folgende Gleichung:

$$F_W=\frac{F_{WG}}{S}$$

$$F_{WG}=F_W*S$$

F_W ist hier die gesamte Kraft des Wassers im Verhältnis zum Oberflächeninhalt. Mit F_{WG} ist die gesamte Kraft, die das Wasser bewirkt, gemeint und mit S der Oberflächeninhalt.

Für das Verhältnis der gesamten Kraft des Wassers einer Probe F_{W1} mit Oberflächeninhalt S_1 zur gesamten Kraft des Wassers der anderen Probe F_{W2} mit Oberflächeninhalt S_2 gilt:

$$\frac{F_{WG1}}{F_{WG2}}=\frac{F_W*S_1}{F_W*S_2}$$

$$\frac{F_{W1}}{F_{W2}}=\frac{S_1}{S_2}$$

Für die Berechnung des Einflusses der Grösse der Körner gehe ich davon aus, dass sie Kugeln wären, da es den Rahmen sprengt, die Form jedes winzigen Sandkornes einzeln zu bestimmen. Durch diese Annahme lassen sich die Formeln für die Oberfläche sowie das Volumen eines Sandkornes ableiten:

$$S = 4\pi r^2$$
$$V = \frac{4}{3}\pi r^3$$

Für die Berechnung des Unterschieds des Oberflächeninhalts von kleineren Sandkörnern in Relation zu grösseren benutze ich die Formel des Volumens. Unter der Annahme, dass der Zwischenraum bei grösseren und kleineren Kugeln im selben Verhältnis zum Gesamtvolumen ist, kann ich das Verhältnis der Stückzahl von den grösseren zu den kleineren in demselben Volumen berechnen. Daraus kann schliesslich das Gesamtvolumen berechnet werden.

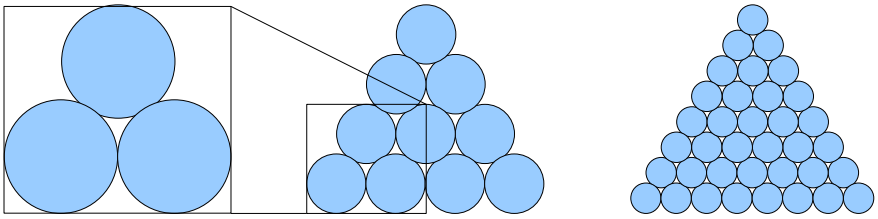

Abbildung 3: Jede Figur ist aus kleineren Stücken des Vorherigen aufgebaut, das Verhältnis von Volumen und Zwischenraum ist also gleich

Das Verhältnis der Stückzahl der Körner n_1 zu grösseren Körnern n_2 ist demnach:

$$\frac{n_1}{n_2} = \frac{\frac{V_{Gesamt}}{V_1}}{\frac{V_{Gesamt}}{V_2}}$$
$$\frac{n_1}{n_2} = \frac{V_2 V_{Gesamt}}{V_1 V_{Gesamt}}$$
$$\frac{n_1}{n_2} = \frac{V_2}{V_1}$$
$$\frac{n_1}{n_2} = \frac{\frac{4}{3}\pi r_2^3}{\frac{4}{3}\pi r_1^3}$$
$$\frac{n_1}{n_2} = \frac{r_2^3}{r_1^3}$$

Daraus lässt sich das Verhältnis des Oberflächeninhalts ausrechnen:

$$S_{Gesamt} = n S_{Einzeln}$$
$$\frac{S_{Gesamt\,1}}{S_{Gesamt\,2}} = \frac{n_1}{n_2} * \frac{S_1}{S_2}$$
$$\frac{S_{Gesamt\,1}}{S_{Gesamt\,2}} = \frac{r_2^3}{r_1^3} * \frac{S_1}{S_2}$$

$$\frac{S_{Gesamt\,1}}{S_{Gesamt\,2}} = \frac{r_2^3}{r_1^3} * \frac{4\pi r_1^2}{4\pi r_2^2}$$

$$\frac{S_{Gesamt\,1}}{S_{Gesamt\,2}} = \frac{r_2^3 r_1^2}{r_1^3 r_2^2}$$

$$\frac{S_{Gesamt\,1}}{S_{Gesamt\,2}} = \frac{r_2}{r_1}$$

Wenn ich die letzte Formel nun in die Ursprungsformel einbaue, lautet sie so:

$$\frac{F_{W1}}{F_{W2}} = \frac{S_1}{S_2}$$

$$\frac{F_{W1}}{F_{W2}} = \frac{r_2}{r_1}$$

2.3.4 Umsetzung im Programm

Im Programm werden die beiden Formeln in der Methode setAngle( ) umgesetzt.

Anhand des Experiments kann man mit den Messdaten des trockenen Sandes μ berechnen und mit den weiteren Daten F_W/F_G und dessen Veränderung berechnen.

Im Programm wird dann bei der Berechnung zuerst anhand der Gleichung aus Kapitel 2.3.3 der Einfluss der bestimmten Grösse berechnet.

Anschliessend wird F_W/F_G entsprechend angepasst. Für die Berechnung des Winkels wird dann die umgeformten Gleichung aus Kapitel 2.3.2 verwendet:

$$\alpha = mod\left(\left(\left(-\arcsin\left(\frac{\mu \frac{F_W}{F_G}}{\sqrt{\mu+1}}\right) - sign(\mu)*\pi + \arctan(\mu) + 2*C*\pi\right)*\frac{\pi}{180}\right), 90\right)$$

Dabei ist C eine beliebig definierbare Variable. Diese kommt daher, dass sich beispielsweise die Sinuskurve alle 360° wiederholt. Deshalb habe ich die Funktion mod() benutzt, die in diesem Fall den Winkel auf die relevanten 0-90° reduziert.

3 Experiment

Für diese Maturaarbeit untersuchte ich die Winkel, die sich bei der Anhäufung bilden, wenn der Sand trocken bis so feucht ist, dass er am stabilsten ist. Die Winkel hinter dieser Grenze habe ich mir nicht angeschaut.

Um das Experiment vorzubereiten nahm ich zwei Sandproben von einem Strand mit. Davon enthielt eine Probe feineren Sand als die andere. Ich wollte den Sand erst völlig trocken messen und ihn dann schrittweise anfeuchten. Zum Trocknen der Proben liess ich sie mehrere Tage in der Sonne stehen.

Das Durchführen des Experiments gestaltete sich schwieriger als erwartet. Zwar bin ich bereits davon ausgegangen, dass der Sand bereits bei einer sehr kleinen Menge Wasser am stabilsten ist, doch hätte ich nicht gedacht, dass dies bereits bei weniger als 5% des eigenen Volumens der Fall ist. Schon als ich es erst mit dieser Menge Wasser versuchte, war der Sand bereits so klumpig geworden, dass es unmöglich war, ihn zum „Rieseln" zu bringen, was eine Messung des Winkels unmöglich machte.

Abbildung 4: Ein Bild des Experiments

Schon bei kleinsten Berührungen bewegte sich der Sand nach dem Aufschütten wieder, deshalb bewertete ich einen Massstab oder ähnliche Messgeräte als ungeeignet. Stattdessen machte ich ein Foto von der Seite, und berechnete daraus den Winkel.

Nachdem ich die Daten des Experiments ausgewertet hatte ergab sich, dass die Probe, die kleinere Körner hatte, mit wesentlich weniger Wasser sofort stabil wurde. Konnte ich bei der anderen Probe den Sand bis 3.2% neu anfeuchten, war bei dieser bereits nach 1.6% Schluss. Der Sand war zu feucht geworden, als dass ich ihn hätte messen können.

H_2O in % des Sand-Volumens	Probe 1 (gröberer Sand)	Probe 2 (feiner Sand)
Trocken	36.83°	35.83°
1.6%	41°	51°
3.2%	44.33°	Zu klumpiger Sand

Tabelle 1: Durchschnittswerte des Experiments

4 Programmierung

Vor dem Beginn dieses Abschnittes möchte ich darauf aufmerksam machen, dass in diesem Abschnitt programmiertechnische Begriffe benutzt werden, welche mit einer Beschriftung in Form von einer Zahl (Beispiel: Klasse[6]) versehen sind, die ihren Nummern bei der Begriffserklärung auf Seite 32 entsprechen.

4.1 Programmiersprache

Die erste Frage, die beim Erstellen eines Programmes beantwortet werden muss, ist die der ausgewählten Programmiersprache. Je nach Sprache können zwei Programme, die absolut dasselbe tun, einen völlig verschiedenen Sourcecode[5] besitzen.

4.1.1 Auswahl

Ich nahm mir für meine Maturaarbeit vor, eine „echte" Programmiersprache - im Sinne ihrer Programmierfreiheit, sodass man also alles mit ihr machen kann, zu erlernen. Ich hatte die Auswahl zwischen Programmiersprachen wie beispielsweise C++ oder Java. Hierbei entschied ich mich nach Abwägung der Kriterien für Java.

4.1.2 Weshalb Java?

Java bietet einige Vorteile, die für mein Programm von Bedeutung sind:

- Java funktioniert auf fast allen Betriebssystemen, vom Desktop Computer bis zum Mobiltelefon.
- Java ist eine einfach zu erlernende Programmiersprache.
- Es ist eine objektorientierte Programmiersprache.
- Java hat eine automatische Speicherverwaltung wie z.b. Den sogenannten Garbage-Collector.
- Java ist eine moderne Programmiersprache.
- Java ist nicht so überladen wie C++, anders gesagt gibt es nicht so viele unterschiedliche Wege ein Problem zu lösen, was den Code übersichtlicher macht.

Für meine Maturaarbeit habe ich mir ein Tutorial in der Programmiersprache Java gesucht und durchgearbeitet (der Link zu diesem Tutorial steht im Anhang).

In dieser Maturaarbeit werde ich nicht konkret erklären, wie mit Java programmiert wird, sondern meine Algorithmik[3] zum Lösen des Problems mit Java beschreiben. Wer aufgrund dieser Arbeit nun auch ein Programm mit Java schreiben möchte oder den Quellcode dieses Programmes nachvollziehen will, dem rate ich, erst das von mir benutzte Tutorial oder ein selbst gesuchtes Tutorial durchzuarbeiten.

4.2 Konzept

Damit mein erstes Programm in Java nicht allzu kompliziert würde, entschied ich mich, die Grafik meines Programms in 2-D zu programmieren.

Das Programm sollte folgendes beinhalten:

- Beim Start sollte ein Fenster aufgemacht werden
- Auf diesem Fenster sollte eine grafische Darstellung des Sandes, der sich aufschüttet zu sehen sein

- Es sollte jeweils einen verstellbaren Regler für Feuchtigkeit und Grösse der Sandkörner geben
- Es sollte gewisse Schalter geben, mit denen sich die Simulation z.b. mit neuen Einstellungen neu starten lässt.

Angefangen hatte ich das Programm mit einem Fenster, auf dem man mit zwei Reglern Feuchtigkeit und Grösse der Körner ändern und mit einem einzelnen Button – dem „restart"-Button - die Simulation neu starten konnte.

Später kam ich auf die Idee, weitere Buttons einzufügen, mit denen sich weitere Einstellungen vornehmen liessen.

Ich baute beispielsweise den „Settingsbutton" ein. Sobald nun dieser gedrückt wird, kann man erweiterte Einstellungen für die Simulation vornehmen.

Ein weiterer Button, den ich einbaute, war der „Add Simulation"-Button. Mit ihm kann man mehrere Fenster öffnen, was zur Folge hat, dass ich eine neue Klasse[6] einbaute, „SimulationsFenster" (näheres dazu im nächsten Kapitel), da ich nun mehrere Simulationen gleichzeitig koordinieren musste.

Aus diesem Button resultierte auch mein letzter Button „Close all Simulations". Seine Funktion besteht darin, beim Betätigen das gesamte Programm zu schliessen, was das Schliessen aller Fenster vereinfacht.

Für den Fall, das nach dem Abschliessen der Programmierung noch etwas Zeit bleibt, überlegte ich mir, ich könnte noch die Grafik von dem anfänglich schwarz-weissen Stil etwas erweitern, indem ich z.b. Farbtöne für den Sand einbaute. So erweiterte ich auch noch das Menü für den Settingsbutton mit der Farbeinstellung, mit Reglern für die Anzahl an Sandkörnern, für die Breite des „Bodens", den Startpunkt der Körner sowie dessen Breite, und einem „Change Color"-Button.

4.3 Strukturierung und Design

Java ist eine objektorientierte Sprache[6], das bedeutet, sie arbeitet mit Klassen[6], Objekten[6] und Instanzen[6]. Nach diesem Prinzip muss auch die Planung des Programms erfolgen.

4.3.1 Klassenstruktur

Das Programm habe ich in 5 Klassen unterteilt:

- SimulationsFenster
- SandSimulation
- Regler
- Grafik
- SandKorn

Die 5 Klassen sind voneinander unabhängig programmierte Klassen, es gibt unter ihnen also keine Super- oder Subklassen untereinander. Allerdings erstellen sie untereinander Instanzen der jeweilig anderen Klasse, wie in Abbildung 6 dargestellt.

In den folgenden Abbildungen ist die Klassenaufteilung veranschaulicht:

In Abbildung 5 sieht man die Funktionen der Klasse SimulationFenster, von der es schlussendlich nur ein Objekt gibt: dieses Objekt wird anschliessend das Öffnen und Schliessen von einer oder mehreren Simulationen übernehmen.

In Abbildung 6 ist die Aufteilung innerhalb einer Simulation dargestellt: links das Aussehen im fertigen Programm, sieht man rechts eine grafische Aufteilung des Fensters in die Klassen einer Simulation, um so dessen Aufgaben symbolisch darzustellen.

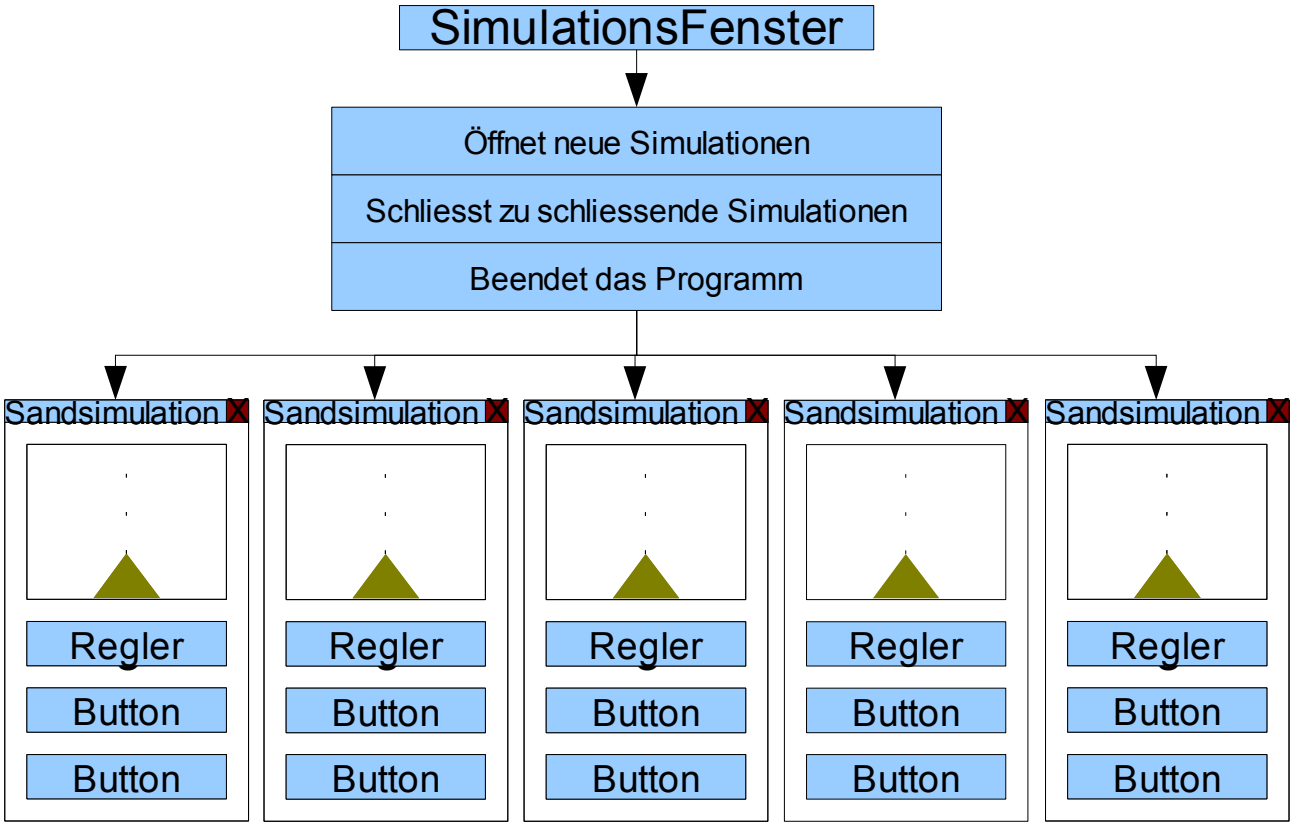

Abbildung 5: Veranschaulichung der Klasse "SimulationsFenster"

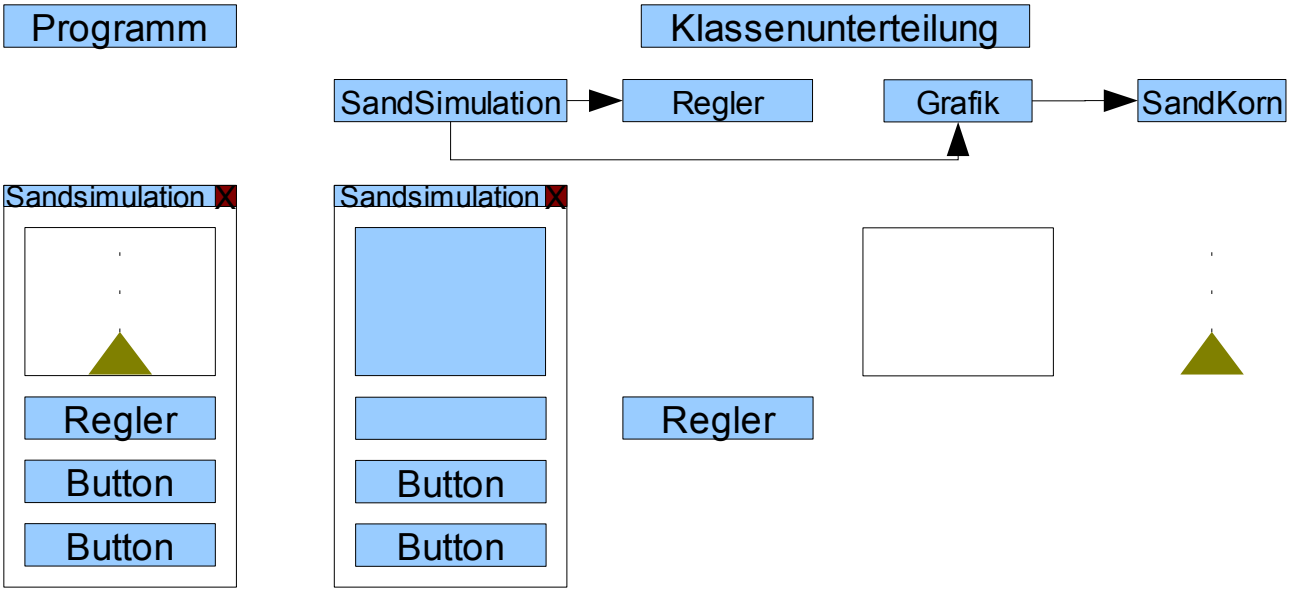

Abbildung 6: Veranschaulichung der Klassenaufteilung innerhalb einer Simulation

4.3.2 Programmablauf

4.3.2.1 Programmstart

Beim Start des Programms wird eine Instanz der Klasse „SimulationsFenster" erstellt. Diese startet zu Beginn eine Instanz der Klasse „SandSimulation".

Die gestartete Instanz von „SandSimulation" startet ihrerseits eine Instanz der Klasse „Grafik", 7 Instanzen der Klasse „Regler" und Alle Buttons und Checkboxen.

Die Instanz der Klasse „Grafik" startet zuletzt alle Körner als Instanzen der Klasse „SandKorn". Im Zusammenspiel dieser 5 Klassen entsteht so das erste Fenster des Programmes.

4.3.2.2 Programmablauf der Simulation

Bei Erstellung einer Simulation wird erst das Fenster, dann dessen Inhalt erstellt. Dann werden die Körner „gestartet". Anschliessend läuft die Simulation ab, bis das letzte Sandkorn an seinem Platz ist. Während des Ablaufs der Simulation lassen sich Einstellungen ändern und die Werte der Regler anpassen. Die Klasse „SandSimulation" reagiert darauf beim Anklicken des Buttons „restart". Wird dieser geklickt, so fragt die Klasse ihre Instanz der Klasse „Grafik" nach den Werten der relevanten Regler, passt entsprechend den Winkel an, und startet anschliessend die Simulation neu.

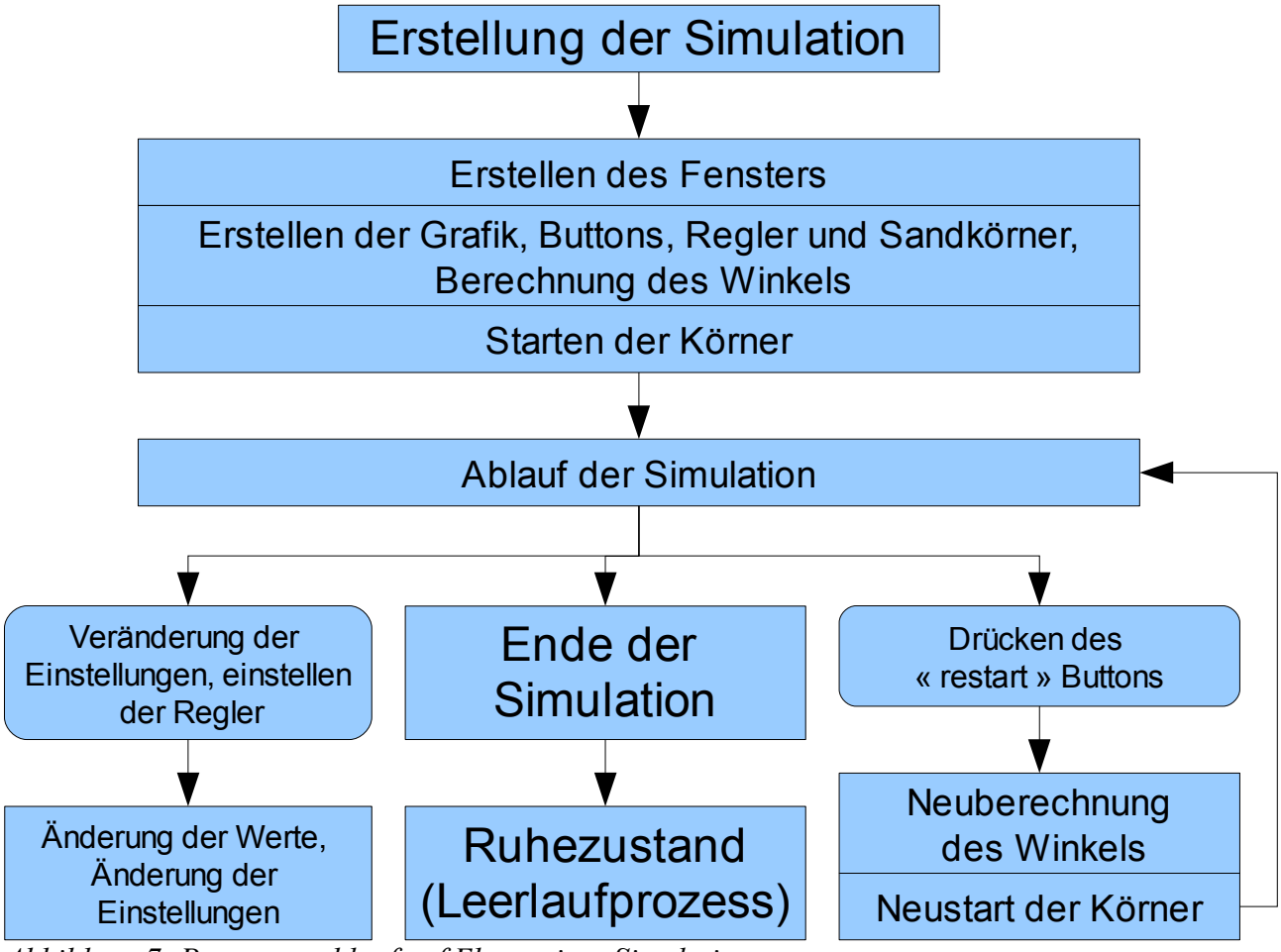

Abbildung 7: Programmablauf auf Ebene einer Simulation

4.3.2.3 Programmablauf des gesamten Programms

Beim Programmstart wird eine Instanz der Klasse „SimulationsFenster" erstellt, also der Klasse, die Öffnung und Schliessung der Simulationen sowie das Programmende regelt.

Nach dem Erstellen der Simulationen regelt diese Instanz die Simulationen insofern, dass sie in regelmässigen Abständen Aktualisierungen in die Wege leitet.

Wird ein Fenster geschlossen, oder auf den „Add Simulation"- oder „Close all Simulations"-Button gedrückt, so reagiert die Instanz der Klasse „SimulationsFenster", indem sie die jeweilige Sandsimulation schliesst, eine neue öffnet oder das gesamte Programm beendet.

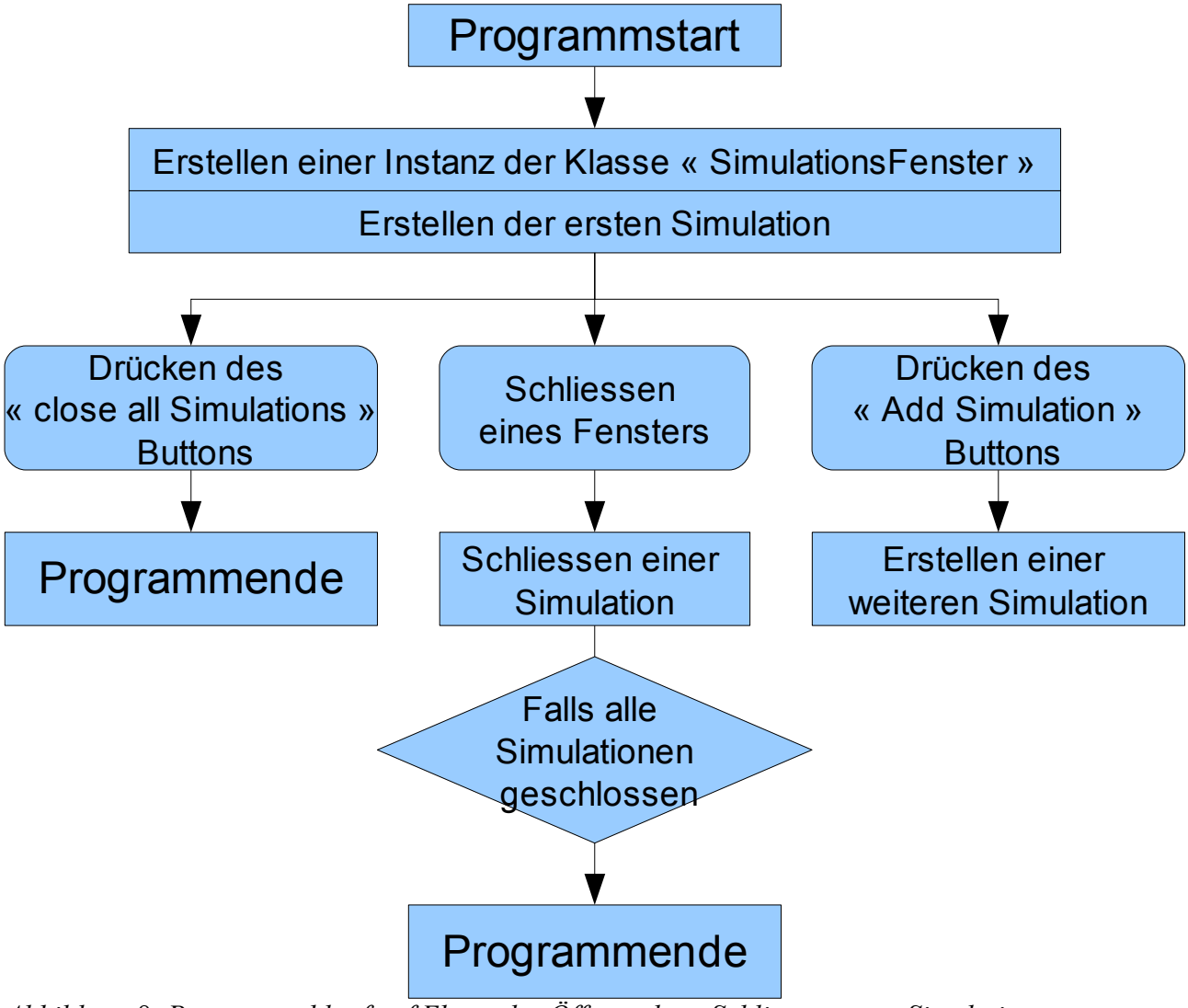

Abbildung 8: Programmablauf auf Ebene der Öffnung bzw. Schliessung von Simulationen

4.3.3 Klassenaufgaben

4.3.3.1 SimulationsFenster

Die Klasse „SimulationsFenster" ist die Klasse, die für das Öffnen und Schliessen der einzelnen Simulationen sowie für die Koordination der Updates, also der Aktualisierung der Simulation in gewissen Zeitabständen, zuständig ist. Von ihr aus wird das Programm

gestartet, werden durch sie Fenster geöffnet, Simulationen erstellt, Fenster wieder geschlossen und schlussendlich wird das gesamte Programm wieder geschlossen.

Die Klasse enthält vier Variablen[7]: Eine Arraylist[8] der Simulationen (also eine Arraylist der Klasse „SandSimulation", eine weitere für deren Fenster, die boolean[9]-Variable „closeSimulation", die, sobald sie „true" beinhaltet, die Schliessung des gesamten Programmes auslöst, sowie die Variable t, die die bisher verstrichene Zeit angibt.

Des weiteren enthält sie die Variablen, welche die standardmässigen erweiterten Einstellungen neuer Simulationen bestimmen. Diese sind Einstellungen, die beim Start eingestellt werden. Sie können per Einstellung bei einer Simulation verändert werden.

4.3.3.2 SandSimulation

„SandSimulation" definiert die Klasse, die in einem einzelnen Fenster die Koordination einer einzelnen Simulation übernimmt. Sie erstellt eine Instanz der Klasse „Grafik", zwei Regler, die Buttons und übernimmt deren Funktion. Sie startet die Simulation beispielsweise neu, wenn auf den „restart"-Button gedrückt wird, oder teilt der Klasse Simulationsfenster im Falle des Drückens des „Add Simulation"-Button mit, das eine weitere Simulation geöffnet werden soll.

Ansonsten verfügt sie noch über zusätzliche Buttons und Regler, die ich noch für die erweiterten Einstellungen programmiert habe.

4.3.3.3 Regler

Die nächste Klasse, genannt „Regler" ist dazu da, einen selbst gemachten Regler zu erstellen, mit dem man gewisse variable Werte beliebig einstellen kann.

Diese Klasse beinhaltet die Variable für ihren minimalen, maximalen sowie momentanen verstellbaren Wert und die X-Position des „Cursors". Ebenso enthält sie Funktionen die zum Beispiel den aktuellen Wert bei einer gewissen X-Position ausrechnen.

4.3.3.4 Grafik

Die Koordination des Bildes in der Simulation übernimmt die Klasse „Grafik". Sie koordiniert sozusagen die einzelnen Körner und übernimmt ihr Update. Das beinhaltet zum Beispiel, dass sie den Körnern mitteilt, wann sie auf ein anderes Korn gestossen sind oder wann sie auf dem Boden gelandet sind. Näheres zu ihrer Funktion finden Sie im nächsten Kapitel „Algorithmus".

4.3.3.5 SandKorn

Die letzte Klasse, „SandKorn" beinhaltet alle Eigenschaften eines einzelnen Sandkornes und gewisse Funktionen. Beispielsweise ist in einzelnen Instanzen die X- oder Y-Position des jeweiligen Kornes eingetragen, sowie die Funktion, das Sandkorn zu starten oder zu stoppen.

4.4 Algorithmus

Bevor man anfängt zu programmieren, sollte man sich erst einmal Gedanken zum zu verwendenden Algorithmus machen.

Mein Programm umfasst gleich mehrere Algorithmen: vom Starten des Programms über die Regelung des eigens erstellten Reglers bis hin zur Strukturierung der Koordination der Simulation; in dieser schriftlichen Arbeit möchte ich mich jedoch auf den spannendsten Teil konzentrieren: die Koordination der Sandkörner.

Dieser Algorithmus beschäftigt sich mit der Frage, wie man die einzelnen Sandkörner dazu bringt, einen Haufen mit einem bestimmten Winkel zu formen, oder anders gesagt: Woher soll ein Sandkorn wissen, ob es bei einer Kollision mit einem bestehenden Korn nach rechts oder nach links ausweichen oder aber stehen bleiben soll?

Der eben beschriebene Algorithmus soll schlussendlich von der Klasse „Grafik" angewandt werden, die die Koordination der Sandkörner übernimmt.

Im Laufe meiner Programmierung entstanden zwei Algorithmen. Beide nutzen den im Theorie-teil angesprochenen vorher berechneten Winkel, der mit der Methode setAngle() unter der Variable „angle" gespeichert wird.

Die Grösse eines Sandkornes entspricht in meinem Programm der eines Pixels. Entsprechend gestalten sich die Algorithmen. Es sind Vergrösserungen bis in die Dimension von Pixeln gezeichnet, und das auf den Haufen fallende Korn ist mit roter Farbe hervorgehoben.

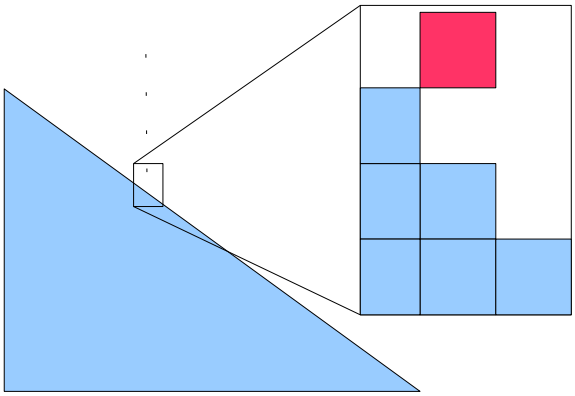

Abbildung 9: Situation vor dem Algorithmus: Ein Sandkorn fällt auf den Sandhaufen (hier eine Seite davon)

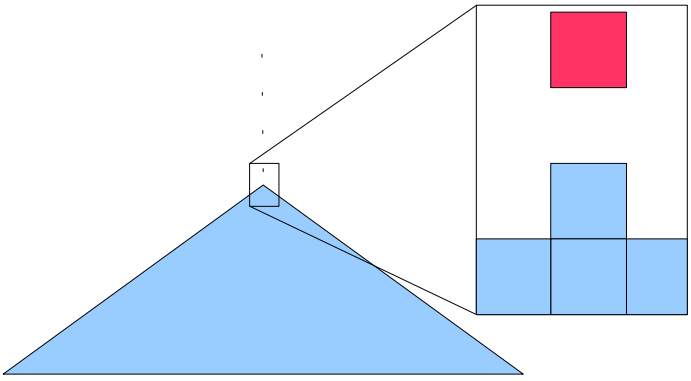

Abbildung 10: Ein Spezialfall: Ein Korn fällt genau auf die Spitze des Haufens

4.4.1 1. Algorithmus

Mein erster Algorithmus bestand in der Grundidee, die Ähnlichkeit[10] des gesamten entstehenden Dreiecks mit dem Dreieck von einem Pixel zum nächsten auszunutzen. Fiel also ein Korn auf einen „Sandturm" der Breite eines Pixels, so wurde anhand der nebenstehenden Türme durch die gegebene Steigung ermittelt, ob der Turm mit einem weiteren Korn zu hoch sei oder nicht. War er es nicht, so blieb das Korn an der Aufprallstelle stehen. War er es, so wich das Korn auf die entsprechende Seite aus. Konnte es auf beide Seiten ausweichen, so wurde per Zufallsgenerator ausgewählt, auf welche Seite das Sandkorn auswich.

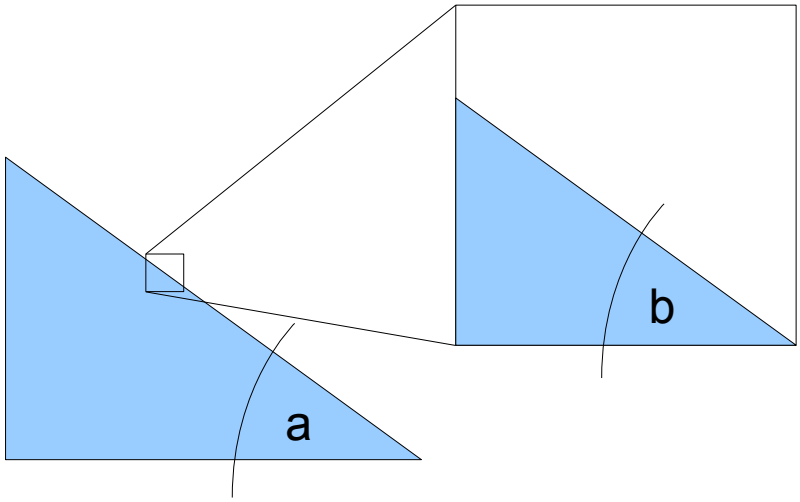

Abbildung 11: geometrische Ähnlichkeit: Winkel a ist gleich dem Winkel b

Die folgenden Abbildungen sind zur Veranschaulichung des Algorithmus:

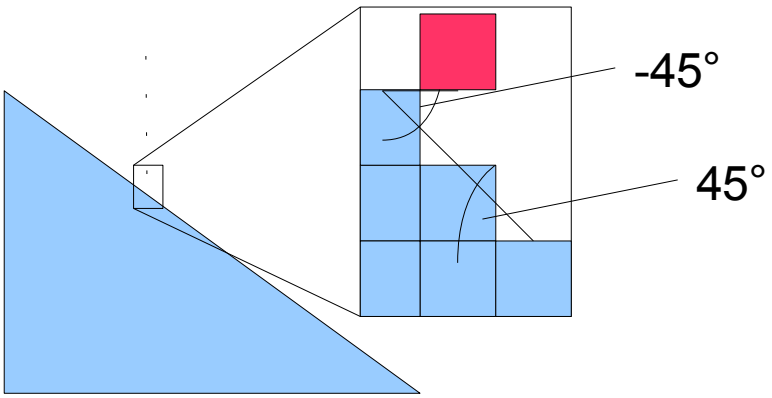

Abbildung 12: Schritt 1: Der Winkel auf der linken und rechten Seite wird bestimmt.

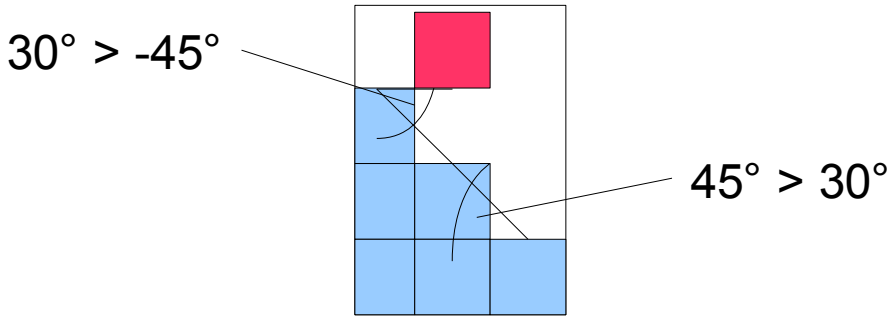

Abbildung 13: Schritt 2: Die Winkel werden mit dem berechneten Wert verglichen

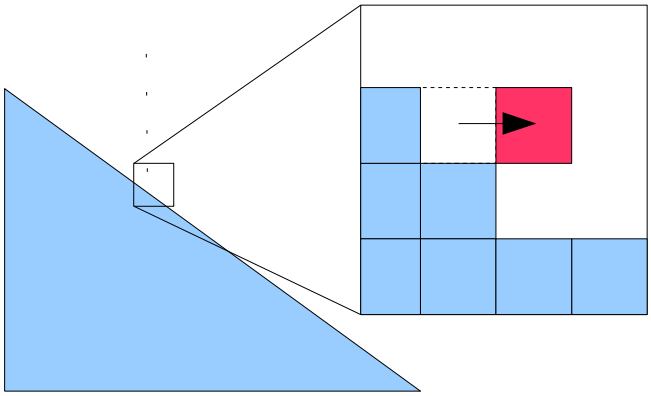

Abbildung 14: Schritt 3: Das Korn weicht nach dem Aufprall auf die gegebene Seite aus oder bleibt stehen.

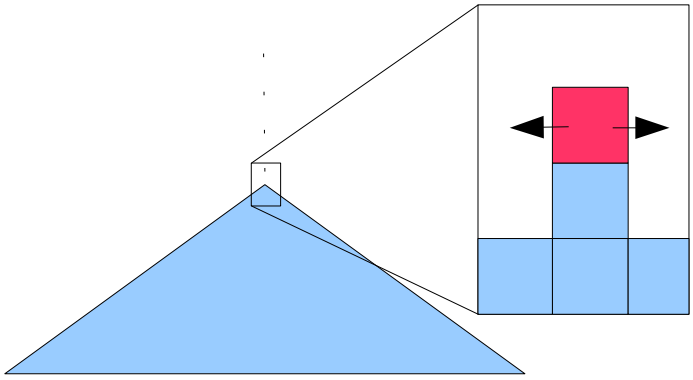

Abbildung 15: Ergebnis im Spezialfall: Das Korn weicht zufällig auf eine Seite aus

4.4.1.1 Pseudocode

Die Methode, die den Algorithmus ausführt, heisst collision(); sie wird dann ausgeführt, wenn sich eine Kollisionsabfrage eines Kornes als positiv erweist, anders gesagt also dann, wenn ein Korn auf den Haufen gefallen ist.

Der Pseudocode meines ersten Algorithmus sieht in etwa so aus:

```
collision( Nummer des Korns, Y-Position nach der Kollision )
{
If(Korn liegt am linken oder rechten Rand)
        Stoppe das Korn
        Setze Y-Position des Korns = Y-Position nach der Kollision
Else If( Winkel des Sandturms beim Pixel des Sandkorns zum nächsten Turm rechts
        >
        angle )
                {
                If (    Winkel des Sandturms beim Pixel des Sandkorns zum nächsten Turm
                        links
                        >
                        angle )
                                Weiche zufällig auf links oder rechts aus.
                                Setze Y-Position des Korns = Y-Position nach der Kollision
                Else
                                Weiche nach rechts aus.
                                Setze Y-Position des Korns = Y-Position nach der Kollision
                End
                }
Else if (       Winkel des Sandturms beim Pixel des Sandkorns zum nächsten Turm links
                >
                angle )
        Weiche nach links aus.
        Setze Y-Position des Korns = Y-Position nach der Kollision
Else
        Stoppe das Korn
        Setze Y-Position des Korns = Y-Position nach der Kollision
End
}
```

Die Funktion wird mit den Variablen Nummer des Korns und Y-Position nach der Kollision aufgerufen, sie müssen also beim Aufruf der Methode angegeben werden. Die Nummer benötigt es für die Identifizierung des Kornes. Die Y-Position nach der Kollision ist gegeben durch die vorher ausgeführte Methode checkForCollision(). Sie rechnet aus, auf welcher Y-Position das Korn nach der Kollision liegen würde, also direkt über dem Korn, mit dem es kollidiert.

Zunächst wird im Code abgefragt, ob das Korn ganz rechts vom Rand liegt, also ob es sich genau am Anfang oder am Ende der Grafik befindet.

Ist dies der Fall, so wird es gestoppt. Es fällt also nicht weiter. Zu guter Letzt wird seine Y-Position auf die Position nach der Kollision angepasst.

Falls es nicht der Fall ist, wird erst der Winkel zur rechten Seite berechnet (in unserem Beispiel in Abbildung 12 also 45°) und mit dem berechneten Wert „angle" verglichen.

Falls der Winkel zu gross ist, wird dasselbe noch einmal für den Winkel nach links ausgeführt: Der Winkel wird gemessen und mit dem berechneten Wert verglichen.

Falls dies ebenfalls zutrifft, so kann das Korn auf beide Seiten ausweichen: es wird also zufällig bestimmt, auf welche Seite es ausweichen wird.

Falls der Winkel nach links nicht zu gross ist, weicht das Korn nach rechts aus.

Auf der nächsten Zeile befinden wir uns bei der Abfrage, die dann ausgeführt wird, falls der Winkel nach rechts nicht zu gross ist. Hier wird nun der Winkel nach links abgefragt und mit dem berechneten Wert verglichen.

Ist der Winkel zu gross, kann das Korn nur nach links ausweichen.

Die nächste Zeile wird dann ausgeführt, wenn sowohl der Winkel nach rechts als auch der Winkel nach links nicht zu gross ist. In diesem Fall kann das Korn auf keine Seite ausweichen und wird deshalb gestoppt.

4.4.2 2. Algorithmus

Mein zweiter Algorithmus entstand aufgrund der Nachteile des ersten, auf die ich noch eingehen werde. Zuerst widmet er sich der Bestimmung, wo ein „Sandhaufen" zu Ende ist, und rechnet anschliessend von dort aus, ob der Sandturm zu gross ist. Bei der Möglichkeit des Ausweichens auf beide Seiten wird hier festgestellt, in welcher Richtung der Winkel stärker abweicht, und in diese Richtung weicht schliesslich das Korn aus.

In den Abbildungen zur Veranschaulichung ist rechts der Spezialfall abgebildet.

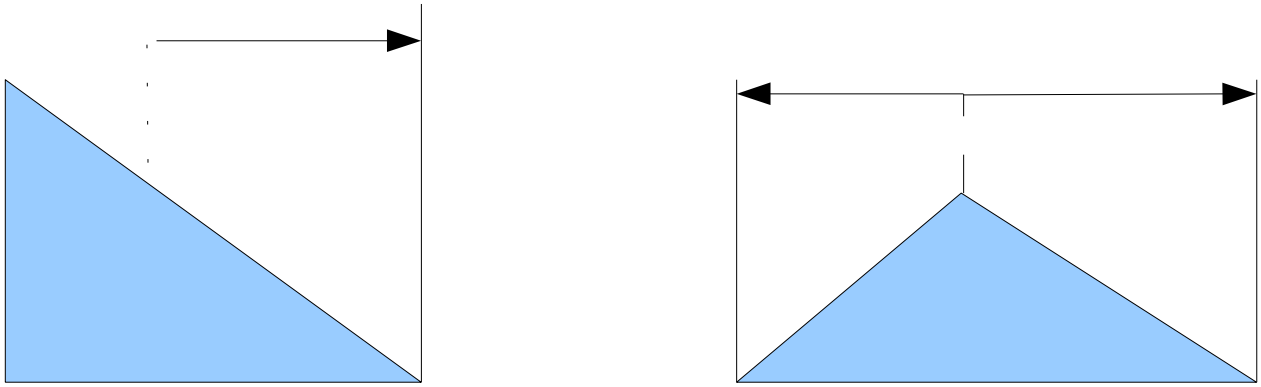

Abbildung 16: Schritt 1: Das Ende des Sandhaufens wird ermittelt

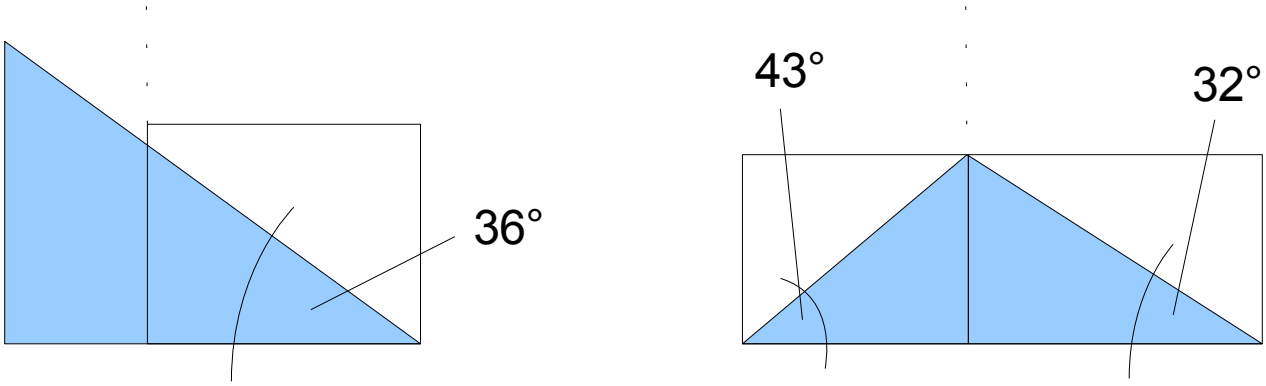

Abbildung 17: Schritt 2: Ausgehend von der X-Position des fallenden Kornes wird der Winkel des Dreiecks vom Aufprall zum Ende des Haufens ermittelt

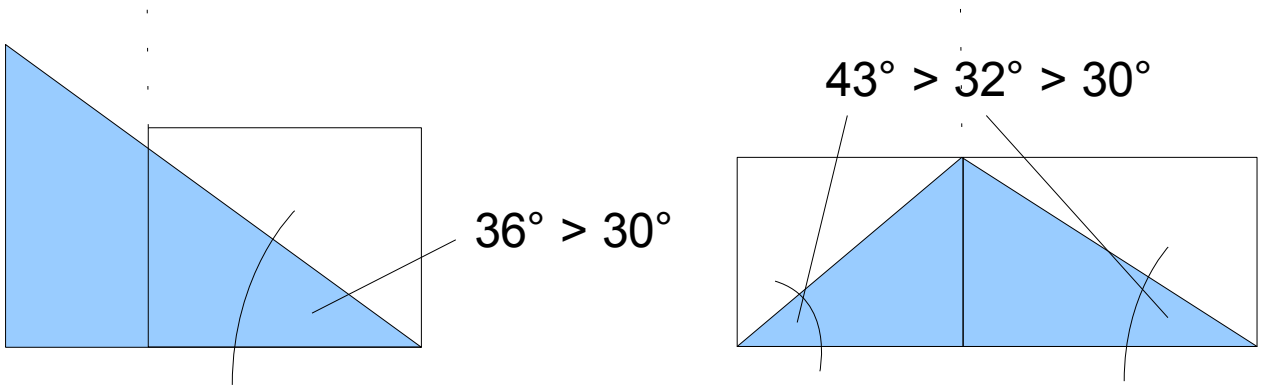

Abbildung 18: Schritt 3: Der Winkel wird mit dem berechneten Wert verglichen

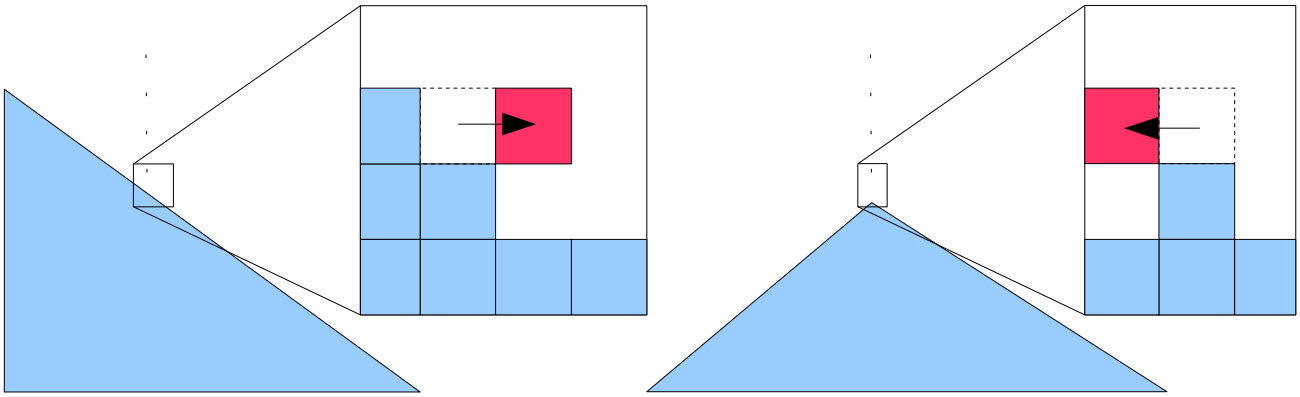

Abbildung 19: Schritt 4: Das Korn weicht auf die entsprechende Seite aus oder bleibt stehen. Im Spezialfall weicht es auf den Winkel aus, der eine grössere Abweichung vom berechneten Wert aufweist

4.4.2.1 Pseudocode

Im zweiten Algorithmus besitzt jeder „Sandturm", also jede natürliche X-Position von Anfang bis Ende der Grafik eine Variable „direction". Diese besagt, ob ein Korn, falls es auf dieser X-Position mit dem Sandhaufen kollidiert, stillstehen, nach rechts ausweichen oder nach links ausweichen soll.

Es wird am Anfang und jedesmal, wenn ein Korn zum Stillstand kommt, für jeden „Sandturm" ausgerechnet, auf welche Seite ein Korn ausweichen soll, falls es auf diesem Turm landet. Das macht die Funktion collision() wie im Pseudocode des ersten Algorithmus für den zweiten Algorithmus etwas unspektakulär:

```
collision( Nummer des Korns, Y-Position nach der Kollision )
{
If (Variable „direction" des Turmes = 0)
        Stoppe das Korn
        Setze Y-Position des Korns = Y-Position nach der Kollision
Else if (Variable „direction" des Turmes = 2)
        Weiche zufällig nach links oder rechts aus.
        Setze Y-Position des Korns = Y-Position nach der Kollision
Else if (Variable „direction" des Turmes = 1)
        Weiche nach rechts aus.
        Setze Y-Position des Korns = Y-Position nach der Kollision
Else
        Weiche nach links aus.
        Setze Y-Position des Korns = Y-Position nach der Kollision
End
}
```

Falls die Variable „direction" für die X-Position des Kornes = 0 ist, wird das Korn gestoppt. Falls sie = 2 ist, weicht das Korn nach rechts oder links aus. Für „direction" = 1 weicht es nach rechts aus. Trifft keine dieser Bedingungen zu, ist „direction" = -1, was bedeutet, dass das Korn nach links ausweichen muss.

Interessanter ist die Funktion setDirection( ), die die anfangs auf 0 gestellten „direction"-Variablen aller X-Positionen nach jedem Stillstand eines Kornes neu berechnet:

Die Methode benutzt vier eigene Variablen: „nachLinksAusweichen", „nachRechtsAusweichen", „linkerWinkel" und „rechterWinkel". Diese sind zu Beginn auf false bzw. 0 gestellt.

```
setDirection( )
{
Führe für jeden „Sandturm" aus:
(
        Suche das rechte Ende des „Hügels" vom Sandturm aus
        Berechne den Winkel vom Ende aus zur Höhe des Sandturmes
        → speichern unter der Variable „rechterWinkel"
        If (rechterWinkel > angle)
                Variable nachRechtsAusweichen = true
        End

        Suche das linke Ende des „Hügels" vom Sandturm aus
        Berechne den Winkel vom Ende aus zur Höhe des Sandturmes
        → speichern unter der Variable „linkerWinkel"
        If (linkerWinkel > angle)
                Variable nachLinksAusweichen = true
        End

        If (nachLinksAusweichen = true)
                If (nachRechtsAusweichen = true)
                        If (rechterWinkel = linkerWinkel)
                                Variable „direction" = 2
                        Else if (rechterWinkel > linkerWinkel)
                                Variable „direction" = 1
                        Else
                                Variable „direction" = -1
                        End
                Else
                        Variable „direction" = -1
                End
        Else if(nachRechtsAusweichen = true)
                Variable „direction" = 1
        Else
                Variable „direction" = 0
        End
)
}
```

Das beschriebene Verfahren wird für jeden „Sandturm" ausgeführt, also für jede X-Position von Anfang bis Ende der Grafik.

Der Code des Algorithmus lässt sich in zwei Teile unterteilen: Im ersten Teil wird untersucht, auf welcher Seite der Winkel zu gross ist und das Korn ausweichen könnte, und anschliessend wird die Variable „direction" im zweiten Teil entsprechend bestimmt.

Im ersten Teil wird wie in Schritt 1 in Abbildung 16 zunächst das Ende des Hügels auf der rechten Seite bestimmt.

Dann wird der Winkel ausgehend von diesem Ende bis zur X-Position, an der das Korn mit dem Haufen kollidiert, ausgerechnet.

Dieser wird nun der Variable „rechterWinkel" zugeordnet. Ist diese Variable grösser als der berechnete Winkel, so kann das Korn auf die rechte Seite ausweichen. Die Variable „nachRechtsAusweichen" wird auf true gesetzt, was bedeutet, dass das Korn nach rechts ausweichen kann.

Dasselbe Verfahren wird nun auch auf die linke Seite angewendet, wobei „rechterWinkel" in diesem Fall durch „linkerWinkel" und „nachRechtsAusweichen" durch „nachLinksAusweichen" ersetzt wird.

Nun kommen wir zum zweiten Teil: Dem bestimmen der Variable „direction".

Aus dem ersten Teil haben wir nun die Variablen „nachLinksAusweichen" und „nachRechtsAusweichen", die besagen, ob das Korn nach links bzw. rechts ausweichen kann.

Erst wird abgefragt, ob „nachLinksAusweichen" true ist, das Korn also nach links ausweichen kann.

Ist dies der Fall, wird zusätzlich abgefragt, ob das Korn auch nach Rechts ausweichen kann.

Stimmt dies ebenfalls, so könnte das Korn also auf beide Seiten ausweichen. In diesem Fall werden die Einkel „rechterWinkel" und „linkerWinkel" verglichen. Je nachdem welcher grösser ist, wird die Variable „direction" auf -1 für links oder auf 1 für rechts gesetzt. Im Fall, dass beide Winkel gleichgross sind, wird „direction" auf 2 gesetzt. Körner, die dann auf den „Sandturm" fallen, weichen zufällig auf links oder rechts aus.

Falls „nachLinksAusweichen" true ist, „nachRechtsAusweichen" aber nicht, so kann das Korn nur nach links ausweichen. Die Variable „direction" wird auf -1 gestellt.

Wenn „nachLinksAusweichen" false ist, so wird der Code mit der Abfrage von „nachRechtsAusweichen" fortgesetzt. Ist diese nun true, so kann das Korn nur nach rechts ausweichen. „direction" wird auf 1 gestellt.

Ist weder „nachLinksAusweichen" noch „nachRechtsAusweichen" true, so ist keiner der Winkel zu gross und das Korn kann nicht ausweichen. „direction" wird in diesem Fall auf 0 gesetzt, sodass Körner bei diesem „Sandturm" gestoppt werden.

4.4.3 Vorteile und Nachteile

Es gibt einen Vorteil des ersten Algorithmus gegenüber dem zweiten. Der 2. Algorithmus hat den geringen Nachteil, dass es bei einem Winkel von weniger als 5° (der bestimmt wirklich kaum anzutreffen ist) der Unterschied in der Abweichung von links und rechts so klein sein kann, dass man klar erkennt, welche Seite die Körner zuerst auffüllen.

Der grosse Nachteil des ersten Algorithmus ist, dass der Vergleich zwischen zwei Pixeln über die Ähnlichkeit keinen genauen Winkel zulässt.

Deshalb entwickelte ich den zweiten Algorithmus, mit dessen Hilfe sich genauere Winkel darstellen lassen.

(Im Anhang finden Interessierte den Quellcode der zwei Algorithmen.)

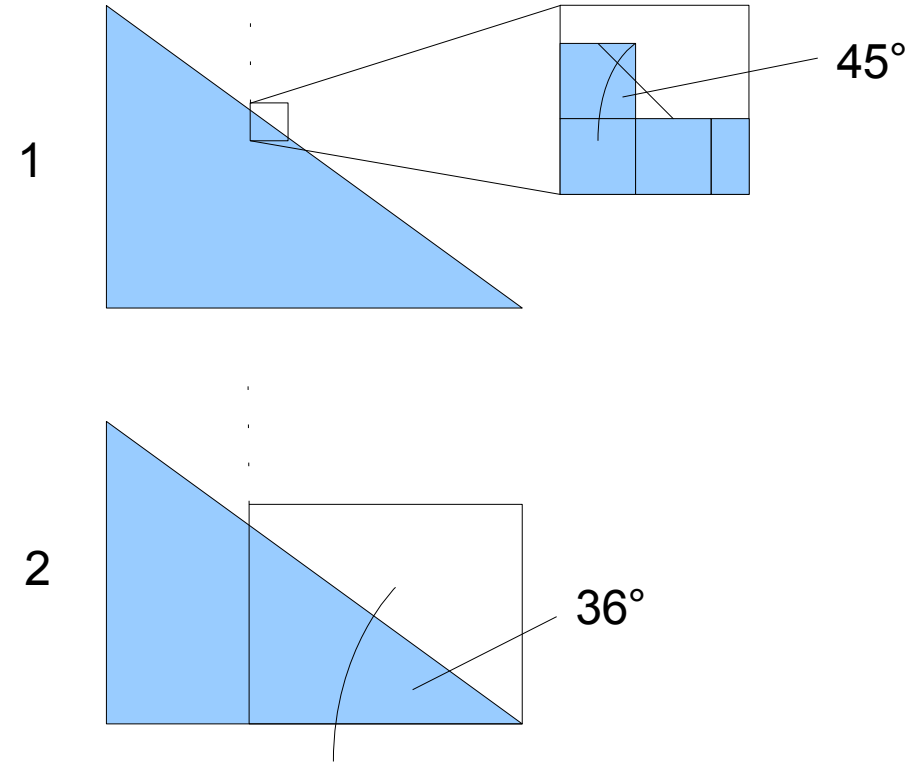

Abbildung 20: Winkelberechnung des ersten und zweiten Algorithmus bei derselben Situation

4.5 Fertiges Programm

4.5.1 Screenshot

Nach dem Fertigstellen meines Programmes sah es in etwa so aus:

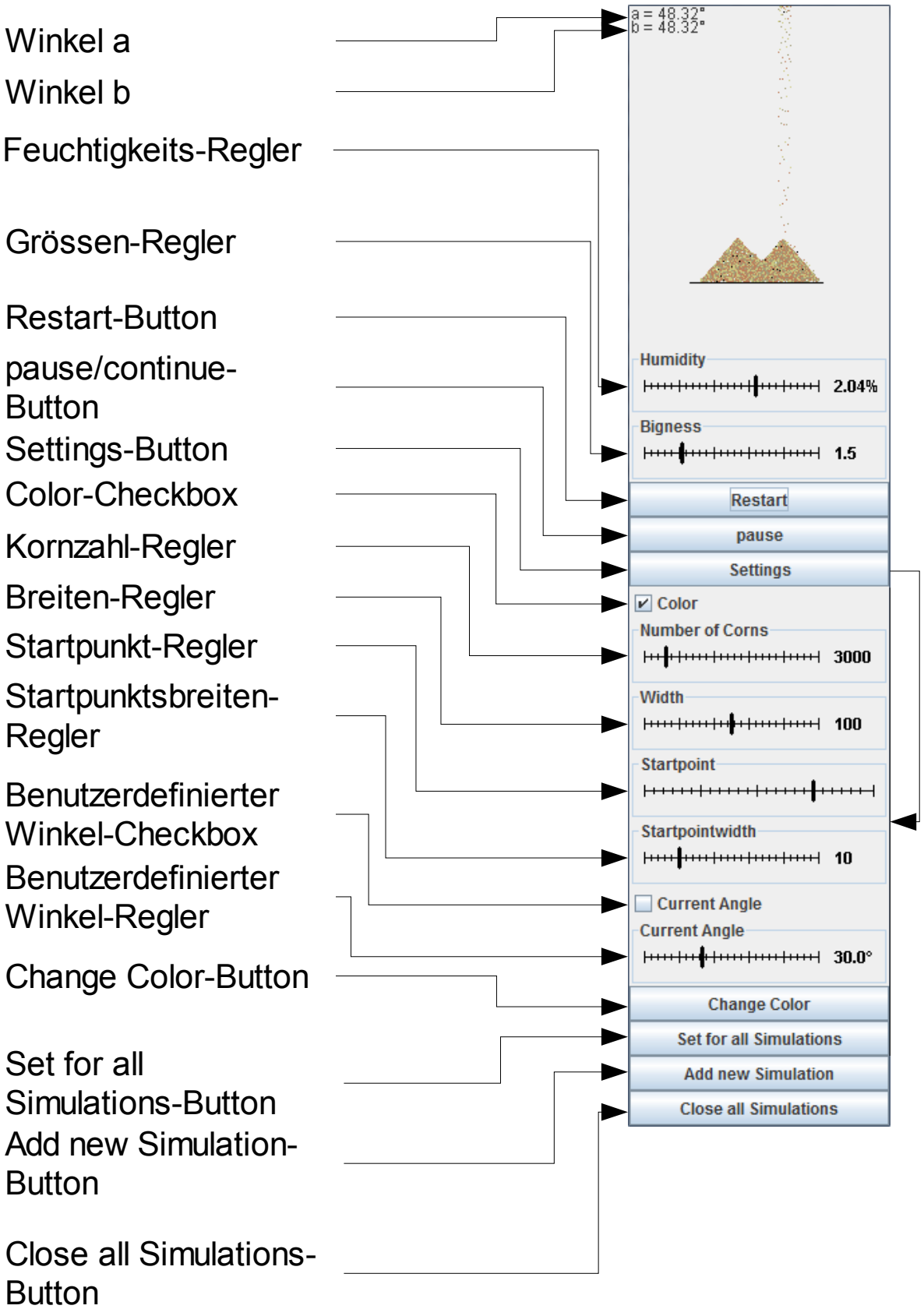

Abbildung 21: Screenshot des fertigen Programmes mit Beschriftungen

4.5.2 Bedienung

4.5.2.1 Anzeigen

Der Winkel **a** bezeichnet den Winkel, der durch die Werte der Regler berechnet wird. Er zeigt an, welcher Winkel benutzt wird, falls nun der „restart"-Button gedrückt wird.

B ist der momentan in der Simulation verwendete Winkel.

4.5.2.2 Buttons

Die Buttons in meiner Simulation geben dem Programm das „Startzeichen" für einen gewissen Vorgang.

Der elementarste Button ist der **Restart**-Button. Nach dem Betätigen dieses Buttons startet die Simulation neu. Zu beachten ist auch dass vorher bestimmte Einstellungen wie die des Feuchtigkeits-, Grössen-, Kornzahl-, Breiten- sowie eigenen Winkel-Reglers erst jetzt in die Rechnung miteinbezogen werden.

Der nächste Button ist relativ selbsterklärend, der **pause**-Button. Er hält die Simulation an und wird dann zum **continue**-Button, der die Simulation beim Drücken wieder weiterlaufen lässt.

Der **Settings**-Button hat die Funktion, die erweiterten Einstellungen ein- oder auszublenden. Er setzt allerdings die dort einstellbaren Werte nicht ausser Kraft.

Der in den durch den Settings-Button geöffneten erweiterten Einstellungen verfügbaren **Change Color**-Button hat die Funktion, das Farbset des Sandes augenblicklich nach dem Drücken zu ändern.

Ein zweiter in den erweiterten Einstellungen verfügbare Button ist der **Set for all Simulations**-Button. Einerseits setzt er die in den erweiterten Einstellungen veränderbaren Werte aller gerade laufenden Simulationen auf die der Simulation (mit Ausnahme des Startpunktes und dessen Breite), bei der er gedrückt wurde (diese werden aber gegebenenfalls erst bei einem Neustart aktiv), andererseits setzt er die Werte der Simulation, bei der er gedrückt wurde als Norm für zukünftig geöffnete Simulationen.

Der zweitletzte Button ist der **Add new Simulation**-Button. Er öffnet eine neue Simulation, also ein neues Fenster, in dem dann eine neue Simulation abläuft.

Der letzte Button, der **Close all Simulations**-Button, schliesst das gesamte Programm, also alle Simulationen und anschliessend das Programm selbst. Zwar wird das Programm auch geschlossen, wenn alle Simulationen normal geschlossen werden, doch dieser Button ist im Fall mehrerer offener Simulationen auf alle Fälle schneller.

4.5.2.3 Regler

Mithilfe der Regler, damit eingeschlossen die selbst erstellten Regler und die Checkboxen, lassen sich gewisse Werte verändern, die dann den Winkel oder die grafische Darstellung der Simulation beeinflussen.

Die zwei elementaren Regler meiner Simulationen sind der **Feuchtigkeits**- und **Grössen**-Regler. Diese geben an, wie „feucht" oder wie „gross" der Sand der Simulation ist, und je nachdem welchen Wert sie beinhalten wird beim Drücken des Restart-Buttons ein anderer zu erwartender Winkel berechnet. Beim Feuchtigkeits-Regler ist die Einheit der Prozentsatz

des Volumens des Sandes in Wasser. Der Grössen-Regler geht von dem Radius des feinsten gemessenen Sand aus.

In den erweiterten Einstellungen, die mithilfe des Settings-Buttons aufgerufen werden, sind die restlichen Regler zu finden.

Die **Color**-Checkbox zeigt an, ob eine Simulation farbig sein soll. Falls diese Checkbox nicht markiert ist, ist die Farbe des Sandes schwarz, was meist einen höheren Kontrast zum Hintergrund verursacht. Falls sie markiert ist, ist der Sand in einem zufällig bestimmten Farbset (also einer Farbpalette, die alle einzelnen Farben beinhaltet) farbig.

Der **Kornzahl**-Regler beeinflusst einfach gesagt, wieviele Körner in der Simulation vorhanden sind. Der anfängliche Normwert beträgt 3000. Nachdem alle Körner gefallen sind, werden keine neuen Körner gestartet. Die Veränderungen dieses Reglers werden erst nach dem Drücken des Restart-Buttons aktiv.

Der **Breiten**-Regler beeinflusst die Breite des Bodens. Je nachdem wie er eingestellt ist hat der Sand mehr Boden, auf dem er sich verteilen kann. Die Veränderungen dieses Reglers werden erst nach dem Drücken des Restart-Buttons aktiv.

Ein weiterer Regler der erweiterten Einstellungen ist der **Startpunkt**-Regler. Er besitzt keinen sichtbaren Wert, denn er bestimmt, wo genau der Sand gestartet werden soll. Dabei berücksichtigt er die Breite des Startpunktes. Wenn das verschiebbare Element dieses Reglers also ganz links ist, fällt auch der Sand an der Stelle, die ganz links am Bodenrand ist. Die Veränderung dieses Reglers wirken sich sofort auf die Simulation aus.

In Relation zum Startpunkt-Regler steht der **Startpunktsbreiten**-Regler. Dieser setzt, wie „breit" der Startpunkt ist. Falls der Wert also 1 entspricht, wird immer nur genau 1 Korn an einer Stelle herabfallen, falls er 11 beinhaltet, werden am genauen Startpunkt, aber auch 10 Stellen links bzw. Rechts davon Körner herabfallen. Die Veränderung dieses Reglers wirken sich sofort auf die Simulation aus.

Falls die **Benutzerdefinierter Winkel**-Checkbox markiert ist, wirken sich nicht mehr der Feuchtigkeits- und Grössen-Regler auf den berechneten Winkel aus, sondern der genaue Wert des **Benutzerdefinierter Winkel**-Reglers. Dieser kann einen Winkel von 0° bis 90° beinhalten. Der berechnete Winkel wird dann der exakte Wert des Reglers sein. Die Veränderungen dieses Reglers werden erst nach dem Drücken des Restart-Buttons aktiv.

5 Vergleich Simulation – Experiment

5.1.1 Grafischer Vergleich

Da meine Simulation zweidimensional programmiert ist, kann ich den grafischen Vergleich nur von der Seitenansicht anstellen. Von daher kann man sagen, dass es beim experimentell aufgeschütteten Sand weniger zu klar gerade verlaufenden Steigungen gekommen ist als es nun bei der Simulation der Fall ist, sondern viel eher zu kurvenartigen Steigungen.

Die Grösse eines Sandkornes ist in meinem Programm gleich der eines Pixels.

Die Körner fallen realistisch nach dem Prinzip des freien Falles, aber die Aufschüttung dauert länger als in der Realität.

5.1.2 Vergleich der Steigung

Da die Steigung des Sandhaufens von mehr Faktoren als nur der Feuchtigkeit und Grösse des Sandes abhängt, ist die Steigung der Simulation der Formel im Theorieteil auf die Messdaten der zwei Sandproben angepasst.

Abbildung 22: Bild zum Vergleich von Experiment - Simulation

6 Nachwort

6.1 Ausblick

Für mein Programm gäbe es noch Möglichkeiten zur Erweiterung, beispielsweise habe ich bei meinem Programm die Höhe, aus der die Körner fallen, nicht berücksichtigt. Würde ich sie berücksichtigen, dürfte der Winkel bei höherem Startpunkt der Körner abnehmen, da diese dann auf die Spitze des Sandhaufens anschlagen und so Körner herunterschubsen oder mitreissen könnten.

Ebenfalls ist mein Programm eine zweidimensionale Simulation. Diese in eine dreidimensionale Simulation umzusetzen wäre eine weitere Erweiterungsmöglichkeit, um die Darstellung realitätsnaher zu gestalten.

Ein weiterer Punkt ist, dass der Code mein allererstes Programm in Java ist und deswegen ist er auch nicht allzu „schön" im Sinne von gut überschaubar. Seine Effizienz dürfte in den Augen eines erfahreneren Java-Programmierers zu wünschen übrig lassen.

Mein Programm geht im Moment nur von den Messdaten zweier Sandproben aus. Man könnte es also erweitern, indem man beispielsweise eine richtige Gleichung für den Winkel eines Sandhaufens aufstellt, die alle relevanten Eigenschaften des Sandes beinhaltet und die man nun im Programm einstellen könnte.

6.2 Danksagung

An dieser Stelle möchte ich mich bei Herrn Steiger bedanken, der mich bei meiner Maturaarbeit unterstützt hat.

6.3 Abschliessender Kommentar

Das Programmieren hat mir sehr viel Spass gemacht. Nach anfänglichen Schwierigkeiten, die mein programmiertechnisches Verständnis herausforderten, arbeitete ich mich nach und nach durch die Funktionen, die ich für mein Programm vorgesehen hatte. Es benötigt Durchhaltevermögen, den Einstieg in eine komplizierte Programmiersprache wie Java zu schaffen. Doch hat man erst einmal die grundlegenden Prinzipien verstanden und auch schon genügend Praxis hinter sich, macht einem das weitere Programmieren richtig Spass und geht auch schnell von der Hand.

7 Anhang

7.1 Quellenverzeichnis

- Link zum benutzten Java-Tutorial (stand 24.11.2010)

http://www.programmersbase.net/Content/Java/Content/Tutorial/Java.htm

7.2 Begriffserklärung

• 1 *Java-Applet: Simulationen eines physikalischen Gesetztes oder Begebenheit, die öfters im Unterricht verwendet werden.*

• 2 *Granuläre Physik: Physik granulärer Materie, also kleiner fester Materie wie Sandkörner oder kleiner Kugeln.*

• 3 *Algorithmik: Im Allgemeinen ist hiermit das „Rezept" gemeint, welches dem Computer in einzelnen Schritten sagt, wie ein bestimmtes Problem zu lösen ist. Näheres dazu findet man unter folgendem Link:*

http://www.algo.informatik.tu-darmstadt.de/algorithmik/was-ist-algorithmik/

• 4 *Eclipse: Eclipse nennt sich die frei erhältliche Programmierumgebung, die ich zum Programmieren benutzt habe. Zu finden ist sie unter folgender offizieller Webseite:*

http://www.eclipse.org/

• *5 Sourcecode: Ein Programmiercode, wie er in der jeweiligen Programmiersprache geschrieben für einen Programmierer lesbar ist.*

• *6 Objektorientierte Programmiersprache: In einer objektorientierten Programmiersprache wird mit dem Prinzip der Klassen und Objekte gearbeitet. Bei diesem werden die Funktionen und Variablen eines Programmes in sogenannten Objekten möglichst zusammengefasst und abgekapselt.*

Ein Klasse kann man am besten wie einen Bauplan beschreiben: in der Klasse steht also, wie ein bestimmtes Objekt auszusehen hat, welche Variablen es hat, wie die Anfangswerte bestimmt werden, wie dessen Funktionen definiert sind.

Java benutzt ebenfalls das Prinzip der Vererbung unter Klassen. Das bedeutet, dass die Superklasse (also die „Elternklasse") der Subklasse (der „Kindklasse") alle Variablen und Funktionen weitervererbt, die dafür freigegeben sind. Hat also zum Beispiel eine Klasse „Fisch" die Funktion genannt getSpecies(), bei der sie „Fisch" ausgibt, so wird diese auch an ihre Subklasse, „Barsch", weitervererbt.

Eine Instanz einer Klasse (= Objekt) könnte man als ein Exemplar des Bauplans der Klasse bezeichnen, also ein Objekt erstellt nach dem Bauplan der Klasse. Die Instanz verfügt dann über alle Variablen und Funktionen, die in ihrer Klasse verzeichnet sind.

• *7 Variablen: Eine Variable kann, je nach ihrem Typ, einen bestimmten Wert speichern. Zum Beispiel kann eine Boolean-Variable entweder true oder false beinhalten, eine Integer-Variable eine natürliche Zahl oder eine String-Variable eine gewisse Zeichenfolge, beispielsweise „Variable".*

Die standardmässigen Variablentypen kann man unter folgendem Link nachlesen:

http://www.programmersbase.net/Content/Java/Content/Tutorial/Java/Datatype.htm

- *8 Arraylist: Eine Liste von Variablen, also quasi ein unendlich langes Array. Ein Arraylist kann unendlich viele Werte beinhalten (solange diese den verfügbaren Speicher nicht überschreiten), die mithilfe einer Nummer aufgerufen werden können.*
- *9 boolean: Eine Art von Variablen, die entweder „true" oder „false" beinhaltet, zu deutsch also „Wahr" oder „Falsch".*
- 10 *Ähnlichkeit: Hierbei ist die geometrische Ähnlichkeit gemeint: Zitat: „In der Geometrie sind zwei Figuren genau dann zu einander ähnlich, wenn sie durch eine Ähnlichkeitsabbildung (auch diese Abbildung wird häufig als Ähnlichkeit bezeichnet), das heißt eine geometrische Abbildung, die sich aus zentrischen Streckungen und Kongruenzabbildungen - Verschiebung, Drehung, Spiegelung - zusammensetzen lässt, ineinander überführt werden können." Nachzulesen unter folgendem Link:*

http://de.wikipedia.org/wiki/Ähnlichkeit_(Geometrie)

7.3 Abbildungsverzeichnis

Alle Abbildungen dieser Maturaarbeit sind vom Autor selbst erstellt.

7.4 Sourcecode

Anders als im Pseudocode wird im Sourcecode nicht der Winkel, sondern die Steigung verglichen. Es dient der einfacheren Berechnung, da man in diesem Fall nicht jedes mal die momentane Steigung des Haufens in einen Winkel umrechnen muss und dafür nur einmal beim Start den Winkel in eine Steigung umrechnen muss.

7.4.1 1. Algorithmus

Hier möchte ich den Sourcecode des ersten Algorithmus zeigen, also die Methode collision(), wie sie in Java geschrieben verwendet wurde. Mein jetziges Programm benutzt diese Methode nicht mehr, stattdessen wird der 2. Algorithmus verwendet.

7.4.1.1 collision()

```
  /**
   * Berechnet Die neue Position eines Sandkornes nach der Kollision.
   * @param number      Die Array-Nummer des Sandkornes
   * @param newy  Die Y-Koordinate, die das Sandkorn nach der Kollision hat.
   */
  public void Collision(int number, double newy)
  {
    if (((int)this.Korn[number].x - this.x0Korn + this.width / 2 == 0) ||
((int)this.Korn[number].x - this.x0Korn + this.width / 2 == this.width - 1))
    {
      this.Korn[number].stop();
      this.Korn[number].y = newy;
      this.height[((int)this.Korn[number].x - this.x0Korn + this.width / 2)] +=
1;
    }
    else if (this.height[((int)this.Korn[number].x - this.x0Korn + this.width /
2)] - this.height[((int)this.Korn[number].x - this.x0Korn + this.width / 2 + 1)]
> this.increase)
    {
      if (this.height[((int)this.Korn[number].x - this.x0Korn + this.width / 2)]
- this.height[((int)this.Korn[number].x - this.x0Korn + this.width / 2 - 1)] >
this.increase)
      {
        if ((int)(Math.random() * 2.0D) == 1)
        {
          this.Korn[number].x += 1.0D;
          this.Korn[number].y = newy;
        }
        else
        {
          this.Korn[number].x -= 1.0D;
          this.Korn[number].y = newy;
        }
      }
      else
      {
        this.Korn[number].x += 1.0D;
        this.Korn[number].y = newy;
      }

    }
    else if (this.height[((int)this.Korn[number].x - this.x0Korn + this.width /
2)] - this.height[((int)this.Korn[number].x - this.x0Korn + this.width / 2 - 1)]
> this.increase)
```

```
    {
      this.Korn[number].x -= 1.0D;
      this.Korn[number].y = newy;
    }
    else
    {
      this.Korn[number].stop();
      this.height[((int)this.Korn[number].x - this.x0Korn + this.width / 2)] +=
1;
      this.Korn[number].y = (this.ymax - this.height[((int)this.Korn[number].x -
this.x0Korn + this.width / 2)]);
    }
  }
```

7.4.2 2. Algorithmus

7.4.2.1 collision()

```
  /**
   * Berechnet Die neue Position eines Sandkornes nach der Kollision.
   * @param number      Die Array-Nummer des Sandkornes
   * @param newy        Die Y-Koordinate, die das Sandkorn nach der Kollision
hat.
   */
  private void collision(int number, double newy)
  {
        if (direction[((int)this.Korn[number].x - this.x0Korn + this.width /
2)]==0)
        {
            this.Korn[number].stop();
            this.height[((int)this.Korn[number].x - this.x0Korn + this.width /
2)] += 1;
            this.Korn[number].y = (this.ymax -
this.height[((int)this.Korn[number].x - this.x0Korn + this.width / 2)]);
            setDirection();
        }
        else if (direction[((int)this.Korn[number].x - this.x0Korn +
this.width / 2)]==2)
        {
            if ((int)(Math.random() * 2.0D) == 1)
            {
              this.Korn[number].x += 1.0D;
              this.Korn[number].y = newy;
            }
            else
            {
              this.Korn[number].x -= 1.0D;
              this.Korn[number].y = newy;
            }
        }
        else
        {
            this.Korn[number].x += direction[((int)this.Korn[number].x -
this.x0Korn + this.width / 2)];
            this.Korn[number].y = newy;
        }
  }
```

7.4.2.2 setDirection()

```
  /**
   * Berechnet die Richtung, in die die Sandkörner ausweichen sollen, neu
   */
  private void setDirection()
  {
        int i;
        boolean leftIsPossible;
        double leftIncrease;
        boolean rightIsPossible;
        double rightIncrease;
        for (int n = 0; n < this.width; ++n)
        {
              i = n;
              leftIsPossible = false;
              while (i-1 > -1 && !(height[i] == 0) && !(height[i] < height[i-
1]))
              {
                    i -= 1;
              }
              if ( i == n )
              {
                    leftIncrease = 0;
              }
              else
              {
                    leftIncrease = 1.0 *( (height[n] - height[i]) )/( (n - i) );
                    if(leftIncrease > increase)
                          leftIsPossible = true;
              }

              i = n;
              rightIsPossible = false;
              while (i+1 < width && !(height[i] == 0) && !(height[i] <
height[i+1]))
              {
                    i += 1;
              }
              if ( i == n )
              {
                    rightIncrease = 0;
              }
              else
              {
                    rightIncrease = 1.0 * ( height[n] - height[i] )/( -(n -
i) );
                    if(rightIncrease > increase)
                          rightIsPossible = true;
              }

              if(leftIsPossible)
              {
                    if(rightIsPossible)
                    {
                          if(rightIncrease > leftIncrease)
                          {
                                direction[n] = 1;
                          }
                          else if(leftIncrease > rightIncrease)
                          {
```

```
                        direction[n] = -1;
                    }
                    else
                    {
                        direction[n] = 2;
                    }

                }
                else
                {
                    direction[n] = -1;
                }
            }
            else if(rightIsPossible)
            {
                direction[n] = 1;
            }
            else
            {
                direction[n] = 0;
            }
        }
    }
```