

# **29. Bundeswettbewerb Informatik 2010/2011**

## **1. Runde**

Tim Taubner, Einsendungsnummer 108

13. November 2010

Dies ist die Dokumentation zu den von mir bearbeiteten Aufgaben 1 bis einschließlich 4 des 29. Bundeswettbewerb Informatik 2010/2011 mit der Einsendungsnummer 108. Für alle Aufgabe werden - wie in der Aufgabenstellung verlangt - jeweils die Lösungsidee und eine Programm-Dokumentation angegeben, sowie geeignete Programm-Ablaufprotokolle. Bei den Aufgaben, zu der eine ausführbare Lösung erstellt wurde, wird auf diese in der Dokumentation verwiesen. Zusätzlich ist am Ende eine allgemeine Beschreibung enthalten, wie die erstellten Programme von der mitgelieferten CD aus gestartet werden können. Alle eingereichten Quelldateien, Kunsterzeugnisse (wie z.B. Bilder) und ausführbare Programmdistributionen wurden alleine von mir, Tim Taubner, erstellt.

## Inhaltsverzeichnis

<b>1 Aufgabe 1: "Informatik"</b>	<b>3</b>
1.1 Lösungsidee . . . . .	3
1.2 Programm-Dokumentation . . . . .	3
1.3 Programm-Ablaufprotokoll . . . . .	4
1.3.1 Erzeugung . . . . .	4
1.3.2 Ergebnisse . . . . .	5
1.4 Programm-Text . . . . .	8
1.5 Programm . . . . .	8
<b>2 Aufgabe 2: "Robuttons"</b>	<b>9</b>
2.1 Lösungsidee . . . . .	9
2.2 Programm-Dokumentation . . . . .	10
2.3 Programm-Ablaufprotokoll . . . . .	12
2.4 Programm-Text . . . . .	16
2.5 Programm . . . . .	22
<b>3 Aufgabe 3: "Logistisch"</b>	<b>23</b>
3.1 Lösungsidee . . . . .	23
3.1.1 Teilaufgabe 1 . . . . .	23
3.1.2 Teilaufgabe 2 . . . . .	23
3.2 Programm-Dokumentation . . . . .	23
3.3 Programm-Ablaufprotokoll . . . . .	24
3.4 Programm-Text . . . . .	24
3.5 Programm . . . . .	25
<b>4 Aufgabe 4: "Drehzahl"</b>	<b>26</b>
4.1 Lösungsidee . . . . .	26
4.2 Programm-Dokumentation . . . . .	26
4.3 Programm-Ablaufprotokoll . . . . .	27
4.4 Programm-Text . . . . .	27
4.5 Programm . . . . .	29
<b>5 Allgemeine Nutzungsanleitung</b>	<b>30</b>
5.1 Dateistruktur der CD . . . . .	30
5.2 Ausführvoraussetzungen . . . . .	30
5.3 Starten der Programme . . . . .	30

## 1 Aufgabe 1: “Informatik”

### 1.1 Lösungsidee

**Künstlerische Absicht:** Meine künstlerische Absicht ist es, ein mathematisch/informatisches Prinzip grafisch ansprechend darzustellen. Hierfür wurde das Konstrukt des Pythagorasbaums benutzt. Ein Pythagorasbaum ist - vereinfacht gesagt - ein rechwinkeliges [Pythagoras-] Dreieck mit Hypotenuse auf einem Quadrat und zwei weiteren Pythagorasbäumen auf den beiden anderen Schenkeln des Dreiecks.

**Formale Definition:** Formal lässt sich der Pythagorasbaum folgendermaßen - wie Abbildung 1 veranschaulicht - definieren:

Verfahren nach (\*) mit zwei beliebigen Punkten A,B mit  $x(B) > x(A)$  und  $y(B) = y(A)$ .

(\*) Errichte auf zwei gegebenen Punkten A,B ein Quadrat mit Eckpunkten A,B,C,D und Seitenlänge  $\overline{AB}$  so, dass gilt  $\angle CAB = 90^\circ$  und  $\angle DBA = -90^\circ$ .

Sei Punkt E so, dass gilt  $\overline{CD}^2 = \overline{CE}^2 + \overline{DE}^2$  und  $y(C) < y(E) > y(D)$ . Verfahren weiter nach (\*) mit  $(A, B) = (C, E)$  und  $(A, B) = (D, E)$ .

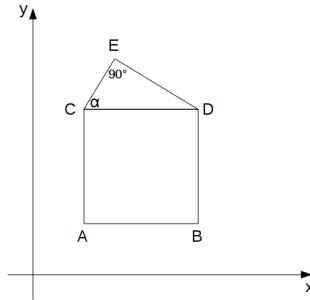


Abbildung 1: Veranschaulichung der Definition

**Besonderheit dieser Inszenierung:** Ein Pythagorasbaum lässt sich mit beliebigen Winkeln zeichnen, ich beschränke mich jedoch auf die drei Winkel  $\alpha = 30^\circ, 60^\circ, 45^\circ$ , wobei  $\alpha = \angle ECD$ . Im Gegensatz zu anderen Pythagorasbaumadaptionen<sup>1</sup>, die versuchen eine möglichst realistische zweidimensionale Darstellung eines realen Baumes zu erlangen, war es hier Ziel, eine informatisch-künstlerische Inszenierung zu erzeugen.

Ich verwende den Begriff Ast für ein Stamm-Quadrat, ein Dreieck und zwei Nachfolgeäste. Es gibt somit drei verschiedene Äste, die ich mit folgenden Ideen implementiere: Äste mit  $\alpha = 30^\circ$  sollen die Sättigung verändern (nach links ungesättigter, nach rechts gesättigter) während Äste mit  $\alpha = 60^\circ$  den Farbton auf der Farbskala verschieben (nach links in Richtung rot, nach rechts in Richtung lila) und Äste mit  $\alpha = 45^\circ$  die Helligkeit manipulieren (nach links heller, rechts dunkler).

### 1.2 Programm-Dokumentation

**Implementation:** Es wurden eine Startregel (*BAUM*), eine Hauptregel (*AST*) und drei gewichtete Teilregeln (*WINKELAST*) definiert. Jede der drei *WINKELAST*-Regeln stellt jeweils einen Winkel dar und zeichnet rekursiv zwei Unteräste, durch zweimaligem Aufruf der *AST*-Regel. Die Berechnung der Koordinaten für die Folgebäume für jeden Winkel folgt aus der Definition.

<sup>1</sup> siehe z.B. [http://de.wikipedia.org/w/index.php?title=Datei:Pythagoras\\_baum\\_color\\_random.png](http://de.wikipedia.org/w/index.php?title=Datei:Pythagoras_baum_color_random.png)

### 1.3 Programm-Ablaufprotokoll

#### 1.3.1 Erzeugung

Das verwendete Programm “Context Free Art” (CFDG) erzeugt Bilder durch interpretieren der Regeln, angefangen bei einer Startregel (hier BAUM). Rekursive Regeln ohne Rekursionsschluss sind erlaubt, das Rendern hört ab einer definierten Größe auf. Regeln lassen sich mehrdeutig definieren, wodurch eine von diesen zufällig gezeichnet wird. Der Zufall lässt sich durch Gewichte an den Regeln beeinflussen. Die Erzeugung des Pythagorasbaums veranschaulicht folgende Abbildung.

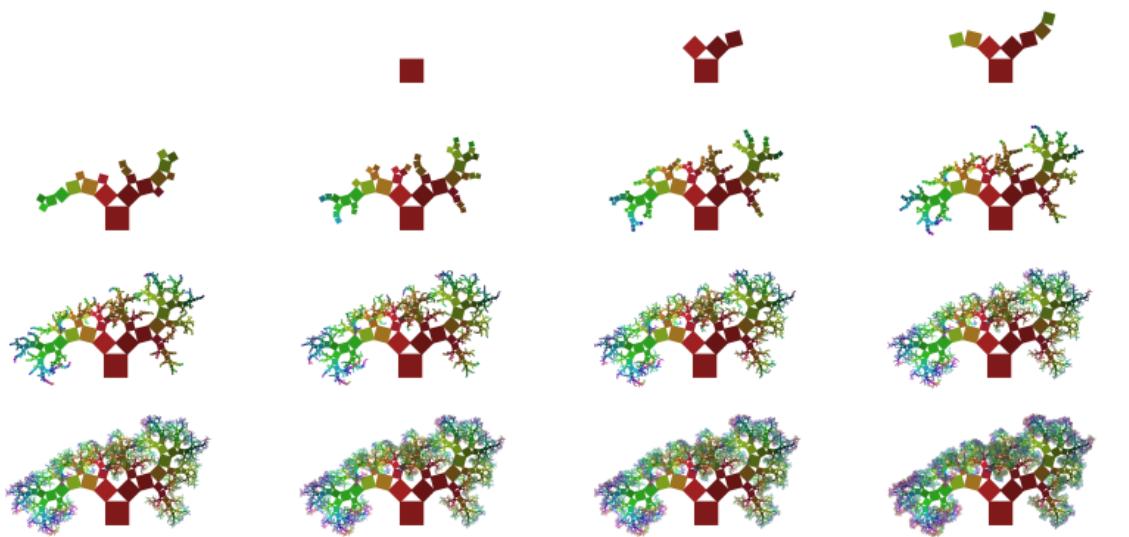
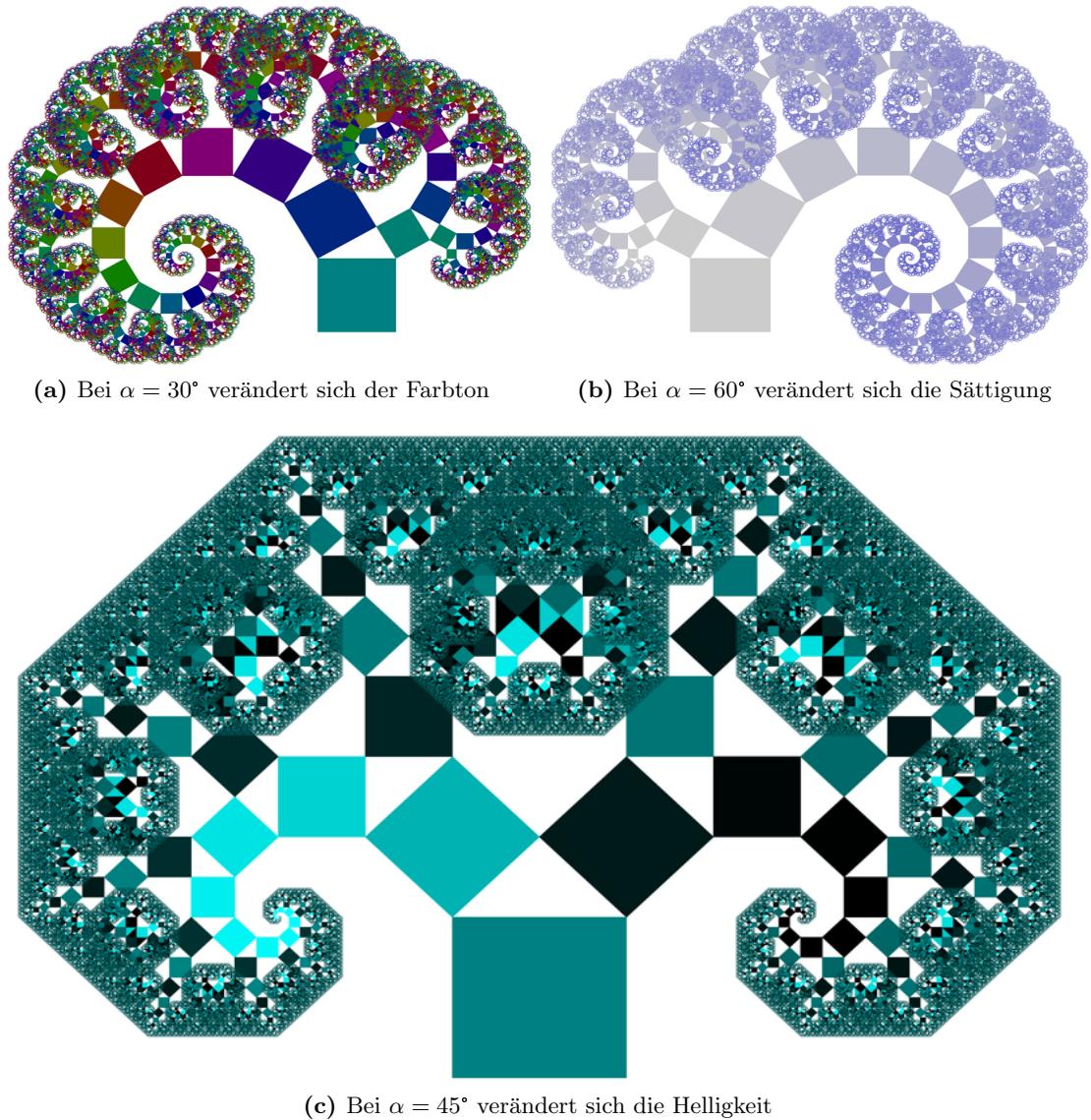


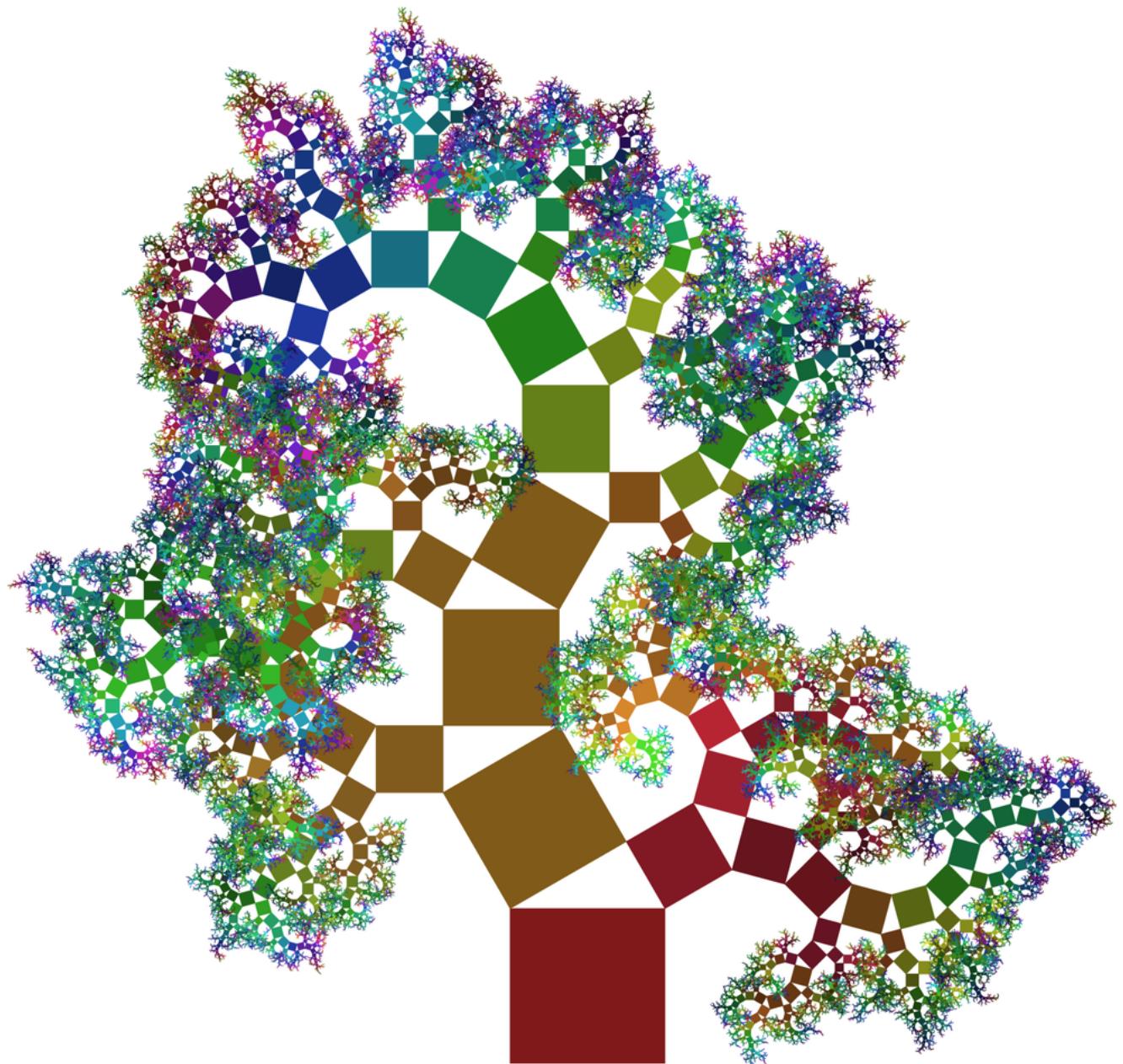
Abbildung 2: Veranschaulichung der rekursiven Erzeugung

### 1.3.2 Ergebnisse

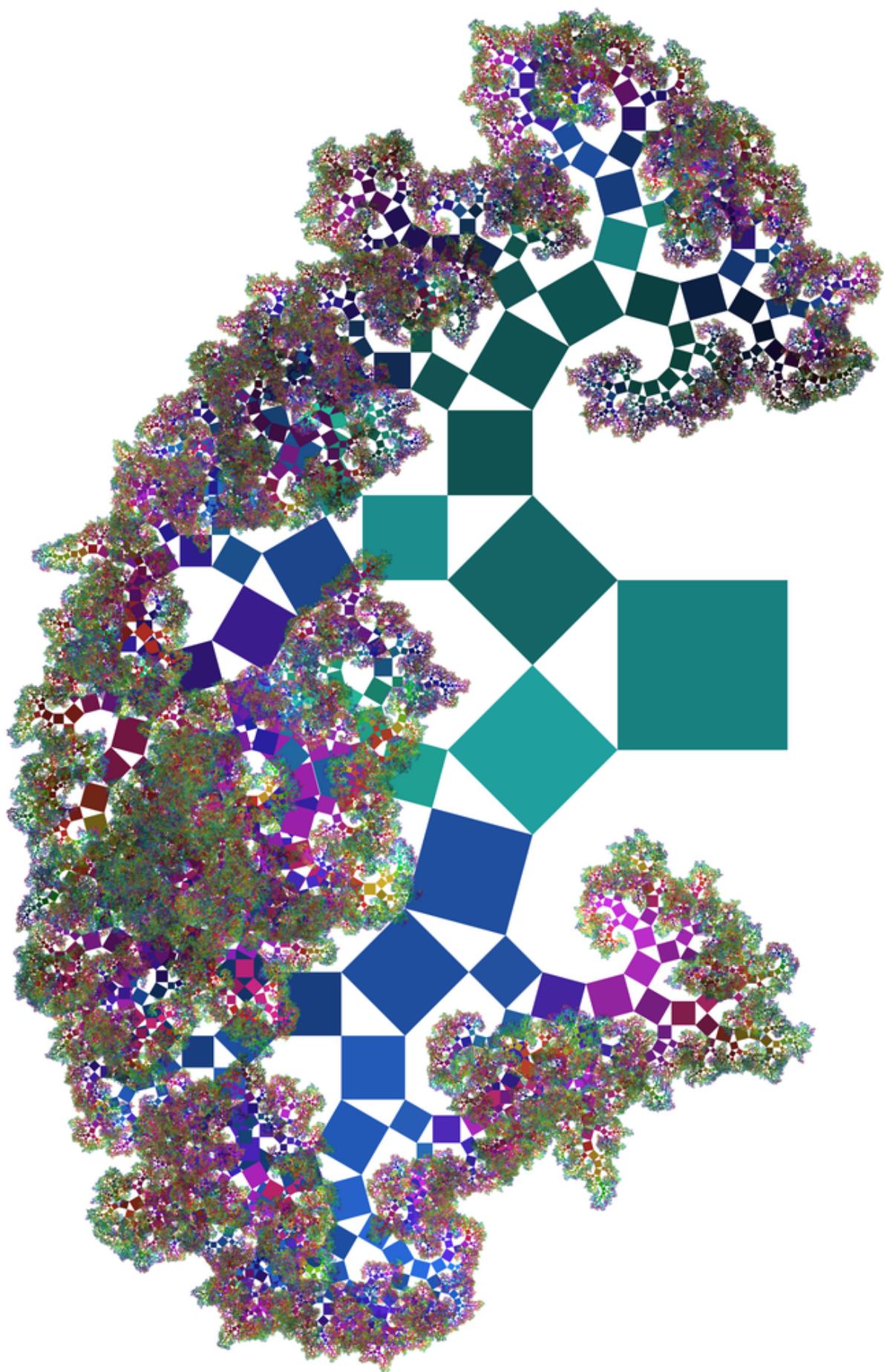
Durch veränderte Gewichte und Startfarben werden stark unterschiedliche Bäume erzeugt. Die verschiedenen Auswirkungen erkennt man am deutlichsten, wenn zunächst nur Bäume mit durchgehend gleichem Winkel betrachtet werden. Dies wird für alle drei Winkel in Abbildung 1 gezeigt: Abb. (a) zeigt das Verändern des Farbtons bei  $\alpha = 30^\circ$ . In Abb. (b) lässt sich die Veränderung der Sättigung bei  $\alpha = 60^\circ$  beobachten, wohingegen sich in Abb. c die Helligkeit bei  $\alpha = 45^\circ$  ändert.



**Abbildung 3:** Auswirkungen der verschiedenen Winkel



**Abbildung 4:** Ein vollständiges Kunstwerk mit Startfarbe rot



**Abbildung 5:** Ein Kunstwerk mit besonders vielen Iterationen, Startfarbe türkis

## 1.4 Programm-Text

Der dokumentierte Quellcode (Aufgabe1/src/baum.cfdg):

```

1 startshape BAUM
2 background {b 1}
3
4 #Rekursionsstart, zeichnet einen Pythagorasbaum
5 rule BAUM {
6   AST {b 0.5 h 0 sat .8}
7 }
8 #Hauptregel, zeichnet Viereck und Hilfsregeln
9 rule AST {
10   SQUARE {}
11   #Unterregel: Winkelast, also ein Ast mit speziellem Winkel, z.B. 45
12   WINKELAST{}
13 }
14
15 #Bemerkung: alpha ist der linke Winkel ueber dem Quadrat.
16
17 #Teilregel, zeichnet einen Ast mit alpha=45
18 rule WINKELAST {
19   #nach (rechts dunkler; links heller)
20   AST{x .5 y 1 s (1/sqrt(2)) r -45 b -.2}
21   AST{x -.5 y 1 s (1/sqrt(2)) r 45 b +.25}
22 }
23
24 #Teilregel, zeichnet einen Ast mit alpha=30
25 rule WINKELAST 1.8 {
26   #nach (rechts "roeter"; links "lilaner")
27   AST{x(.5+sin(15)/sqrt(8))
28     y(sin(75)/sqrt(8) +.5) s (1/2) r -60 h -6}
29   AST{x(-.5+sin(15)/sqrt(8)*sqrt(3))
30     y(sin(75)/sqrt(8)*sqrt(3)+.5) s(sqrt(3)/2) r 30 h +38}
31 }
32
33 #Teilregel, zeichnet einen Ast mit alpha=60
34 rule WINKELAST 1 {
35   #nach (rechts ungesaettigter; links gesaettigter)
36   #aber nie unter sat 0.2, oder ueber sat 1.
37   AST{x(-.5-sin(15)/sqrt(8))
38     y(sin(75)/sqrt(8) +.5) s (1 /2) r 60 sat -.03141|sat 0.1}
39   AST{x(.5-sin(15)/sqrt(8)*sqrt(3))
40     y(sin(75)/sqrt(8)*sqrt(3)+.5) s(sqrt(3)/2) r -30 sat +.03141|sat 1}
41 }
```

## 1.5 Programm

Der Quellcode kann nicht direkt ausgeführt werden, aber er kann mit cfdg interpretiert werden. Beachten Sie jedoch, dass jede Ausführung mit cfdg - aufgrund der durch Zufall ausgewählten Regeln - ein anderes Ergebnis erzeugt. Die Ausführung folgt unter dem Wurzelverzeichnis mit: ‘cfdg Aufgabe1/src/baum.cfdg Pfad/zur/Ausgabe.png’. Da auf die CD keine Dateien geschrieben werden können, geben Sie bitte einen Pfad an, in dem Sie die Ausgabedatei speichern möchten. Es sind außerdem die Erzeugnisse unter ‘Aufgabe1/dist’ enthalten.

## 2 Aufgabe 2: “Robuttons”

### 2.1 Lösungsidee

Ein Robutton und eine Münze werden durch Kreise mit einem Radius  $r$  und einer als Vektor dargestellten Position  $\vec{loc}$  modelliert. Ein Robutton hat zusätzlich eine Geschwindigkeit  $v$  in eine bestimmte ebenfalls als Vektor dargestellte Richtung  $\vec{dir}$  und gegebenfalls eine Münze, die er gerade trägt. Alle Simulationsregeln verändern lediglich die Richtung der Robuttons aber nicht die Geschwindigkeit. Idee ist nun, einen Tisch mit einer bestimmten Form (z.B. Rechteck) und verschiedenen Robuttons und Münzen wie folgt zu simulieren:

In einem Iterationsschritt (mit Zeitdifferenz  $t$  zum vorherigen) muss folgendes erledigt werden:

- Jeder Robutton muss um  $\frac{v}{t}$  in seine Richtung  $\vec{dir}$  bewegt werden.
- Für jeden Robutton muss geprüft werden, ob er mit einer Tischkante kollidiert. Falls er kollidiert, wird er reflektiert.
- Für jeden Robutton muss für jeden anderen Robutton geprüft werden, ob diese kolli-dieren. Falls eine Kollision vorliegt, müssen sich die Robuttons entsprechend der Regeln drehen.
- Für jeden Robutton muss geprüft werden, ob er mit einer Münze kollidiert. Falls zu-treffend, muss er sie aufheben bzw. absetzen und sich ggf. drehen (entsprechend der Regeln).

**Kollisionerkennung:** Zwischen zwei Kreisen (also Münzen oder Robuttons) liegt eine Kol-lision genau dann vor, wenn die Summe der Radien größer ist als der Abstand der beiden Mittelpunkte.

Um die Kollisionserkennung an Tischkanten einfach zu halten, nehme ich an, dass alle Tische eine konvexe polygonale Form besitzen. Durch diese Annahme lässt sich jeder Tisch als eine Menge von Geraden darstellen. Nun liegt genau dann eine Kollision vor, wann an einer der Tischkanten - also einer Gerade - eine Kollision vorliegt. Die Kollisionserkennung an einer Geraden jedoch ist einfach: Ein Robutton kollidiert genau dann, wenn der Radius größer ist als der Abstand zwischen der Geraden und dem Mittelpunkt des Robuttons.

**Reflektion und Drehung:** Um die Reflektion an einer Tischkante zu implementieren, genügt die Betrachtung Einfallswinkel ist gleich Ausfallwinkel.

Um eine Kollision zwischen einem Robutton und einer Münze zu simulieren, muss eine Dre-hung um  $180^\circ$  ermöglicht werden. Diese ist trivial, sie kann z.B. durch Negation des Richtungs-vektors erfolgen. Es sind jedoch auch Drehungen um einen beliebigen Winkel erforderlich, da diese nach einer Kollision zwischen zwei Robuttons erforderlich sind. Alle drei erläuterten Problemstellungen lassen sich zusammenfassend durch eine allgemeine Funktion zur Rotation um einen beliebigen Winkel beschreiben. Durch Trigonometrie kann diese allgemeine Drehung z.B. um einen Winkel  $-78^\circ$  berechnet werden.

## 2.2 Programm-Dokumentation

Die Implementierung erfolgt in mehreren Stufen.

**Kern:** Zunächst wird der Kern entwickelt, er besteht im Prinzip aus drei Klassen, die jeweils einen Teil der Simulationslogik implementieren. Die drei Klassen sind: Robutton, Coin (Münze) und Table (Tisch). Da Robuttons und Münzen beide jeweils einen Radius und einen Ort haben, wurde dieser Teil in eine Superklasse Button ausgelagert. Ein Robutton benötigt zusätzlich eine Geschwindigkeit und eine Richtung, sowie Methoden, welche die Simulationsregeln implementieren. Ein Tisch enthält Münzen, Robuttons und Methoden, um neue Robuttons oder Münzen hinzuzufügen.

Im nächsten Schritt werden Hilfsklassen definiert. Eine der wichtigsten Hilfsklassen ist die Vektorklasse (Vec2), die einen Punkt im zweidimensionalem Raum darstellt. Es existieren zwei Unterklassen, Radial2 und Cartese2, die jeweils eine Kreiskoordinate bzw. eine kartesische Koordinate repräsentieren. Die Vektorklassen stellen Methoden zur Distanzberechnung, Winkelberechnung, Vektoraddition, etc. bereit.

Weiter wird eine Klasse Shape, also Form eingeführt. Diese stellt die Form des Tisches dar. Um alle möglichen konvexen Polygone zu modellieren, genügt es, eine Klasse für Geraden (LineShape) und eine für Polygone - also einer Menge von Geraden - (CompositeShape) zu implementieren.

Die Klasse LineShape stellt Geraden besonders sparsam durch einen Kreisvektor dar. Die Geradendarstellung kann folgendermaßen veranschaulicht werden: Drehe man die x-Achse um den Winkel des Kreisvektors, ist die dargestellte Gerade die Senkrechte auf der gedrehten x-Achse mit Abstand in Höhe der Länge des Kreisvektors. Die Reflektion an dieser Geraden ist dadurch sehr einfach, wenn man folgende Überlegung einbezieht. Erinnere man sich an die Grundüberlegung Einfallswinkel gleich Ausfallswinkel. Aus dieser lässt sich folgern, dass auch auf der Senkrechten Einfallswinkel gleich Ausfallswinkel gilt. Die Senkrechte auf einer Geraden wird auch deren Normale genannt. Diese ist dargestellt als der negierte, normalisierte Gera- denvektor (der Kreisvektor, der die Gerade darstellt). Kollidiert nun ein Robutton mit einer Geraden, wird direkt diese Normale benutzt, um die neue Bewegungsrichtung zu errechnen.

**Simulations-Leistungsfähigkeit:** Es ist Ziel, eine möglichst hohe Anzahl Robuttons und Münzen bei komplexen Tischformen und trotz hoher Bewegungsgeschwindigkeit fehlerfrei simulieren zu können. Dies wird durch möglichst viele Iterationsschritte pro Zeiteinheit erreicht. Auch wenn das Programm effizient geschrieben ist, stößt es schnell an die Grenzen eines Prozessors. Um Kapazitäten moderner Rechner mit Mehrkernprozessoren ausnutzen zu können, implementierte ich ein parallel ablaufendes Framework. Dieses wurde durch die moderne Java Concurrency API erstellt. Das Modell ist folgendermaßen implementiert<sup>2</sup>: Eine Klasse verwaltet einen ExecutorService (Java API), der Routine-Aufgaben ausführt. Jede Klasse, die Updates benötigt (z.B. Table oder Robutton) reiht ihre Aufgaben in dieser ExecutorService-Klasse ein. Die Aufgaben werden dann auf verschiedene Threads verteilt, die von verschiedenen Kernen ausgeführt werden. Eine wichtige Voraussetzung dafür war, dass alle Klassen thread-sicher sind, also alle Informationen zwischen den Threads synchronisiert werden.

Durch das Aufteilen der Simulation in einfache, voneinander unabhängige Teile und ihre parallele Ausführung konnten auf der Entwicklermaschine (4 Kerne, Intel Core 2 Quad q6700 @ 3,2 GHz, 4 GB RAM) dadurch bis zu knapp 300 Robuttons mit etwa 1000 Münzen flüssig und fehlerfrei simuliert werden. Ohne Multithreading (Ausführen aller Aufgaben in einem Thread - daher sequentiell - auf einer Maschine) konnte nur eine Maximalanzahl von etwa 100 Robuttons mit etwa 300 Münzen erreicht werden. Die Werte mit Multithreading sind nicht

---

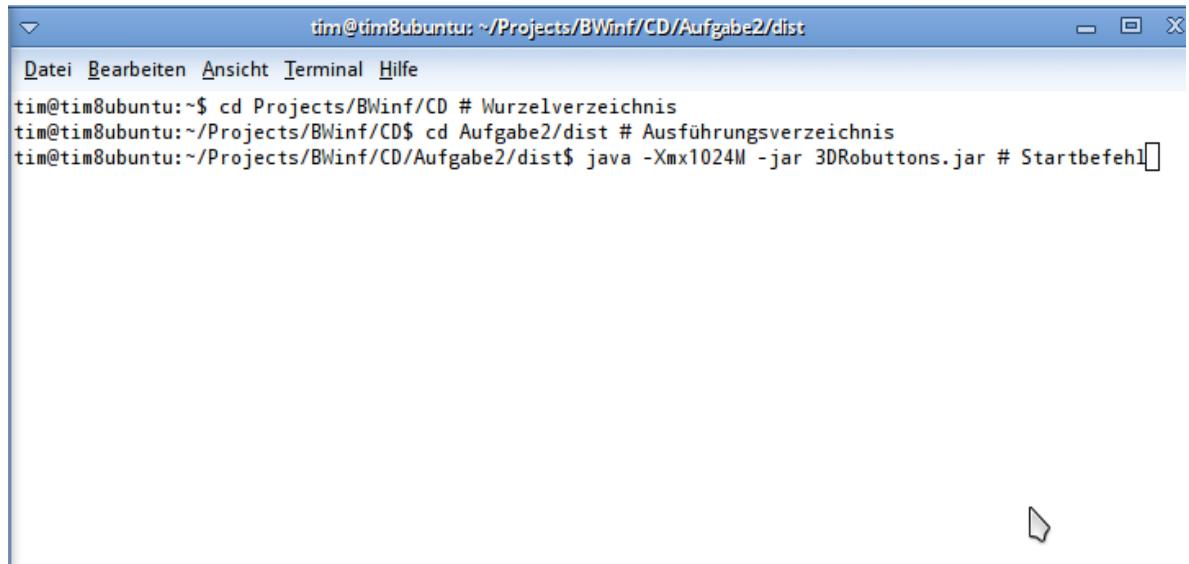
<sup>2</sup>Lediglich vereinfacht erklärt, diese Multithreading-Mechanismen sind für den eigentlichen Algorithmus nicht relevant

ganz das Vierfache von dem ohne Multithreading. Dies liegt daran, dass die der grafische Darstellung auch Rechenzeit beansprucht und dass das Verteilen der Aufgaben auf die Kerne Overhead erzeugt. Der Overhead konnte jedoch relativ niedrig gehalten werden, da sich die voneinander unabhängigen Prozesse nicht blockieren.

**Grafische Darstellung:** Zusätzlich bietet das entkoppelte, stark parallelisierte Modell die Grundlage für eine von der Simulation unabhängig ausgeführte, grafische Darstellung. Diese blockiert während dem Rendern auf der GPU (Grafikkarte) die CPU nicht, welche die Simulation ausführt, was zu einer noch weiter erhöhten Parallelität führt. Die implementierte 3D-Ansicht basiert auf der Grafik-Engine jMonkeyEngine 3.0. Da diese sich zum Entwicklungszeitpunkt noch in der Betaphase befindet, tauchen leider kleinere graphische Fehler auf (z.B. "huschen" Schatten über das Bild). Ziel der 3D-Darstellung ist es, eine schöne, graphisch ansprechende Mensch-Computer-Schnittstelle zu realisieren. Deswegen wurde sie auch durch passende 3D-Modelle und Texturen graphisch aufbereitet. Die eigentlich zweidimensionale Simulation wird durch eine dritte Dimension erweitert, so dass der Benutzer die Simulationswelt ähnlich der wirklichen Welt beobachten kann. Auf eine detailliertere Dokumentation der grafischen Darstellung habe ich hier verzichtet, weil der eigentliche Lösungsalgorithmus für die Robuttons in dem Kern-Programmteil enthalten ist.

### 2.3 Programm-Ablaufprotokoll

Hier ist die Ausführung des Programms protokolliert, wie sie auch bei einer Ausführung zu beobachten ist. Zunächst werden die Befehle, wie in folgender Abbildung dargestellt, ausgeführt.



```
tim@tim8ubuntu:~/Projects/BWinf/CD/Aufgabe2/dist
Datei Bearbeiten Ansicht Terminal Hilfe
tim@tim8ubuntu:~$ cd Projects/BWinf/CD # Wurzelverzeichnis
tim@tim8ubuntu:~/Projects/BWinf/CD$ cd Aufgabe2/dist # Ausführungsverzeichnis
tim@tim8ubuntu:~/Projects/BWinf/CD/Aufgabe2/dist$ java -Xmx1024M -jar 3DRobuttons.jar # Startbefehl
```

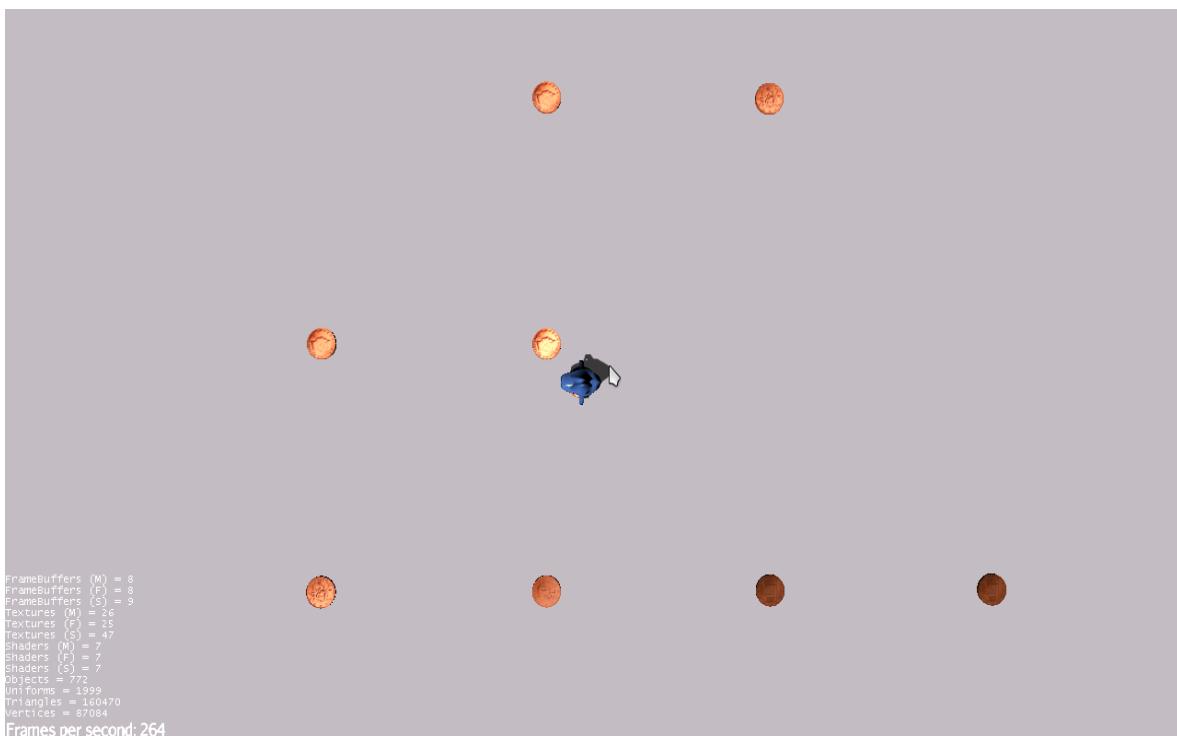
**Abbildung 6:** Die drei Befehle um das Programm zu starten

Daraufhin öffnet sich der Startbildschirm. Hier wird der Fenstermodus ausgewählt mit maximaler Auflösung (1280x720), um besser Screenshots machen zu können. Also ist die Fullscreen-Auswahlbox nicht ausgewählt.



**Abbildung 7:** Auswahl der Auflösung und Bestätigung per OK

Nun öffnet sich das 3D-Fenster. Hier sind Robuttons blau, während Münzen als orange- bis bronzerfarben zu erkennen sind.



**Abbildung 8:** Vor Start der Simulation, Standardansicht

Die Simulation wird nun mit der Leertaste gestartet, dann werden aus verschiedenen Perspektiven und zu verschiedenen Zeitpunkten Screenshots gemacht. Besonders spannend zu beobachten, ist das Abbauen von kleineren Haufen. Nach längerer Zeit bleiben in der Regel nur wenige, größere Haufen übrig, da kleinere Haufen leichter abgebaut werden und Münzen nur an Haufen wieder abgelegt werden.

Hier nicht zu beobachten, aber auch interessant: Wenn es so viele Robuttons gibt, dass alle Münzen aufgenommen werden, dann sind nur noch Robuttons unterwegs und es bilden sich keine Haufen mehr, da Münzen ohne andere, freiliegende Münzen nicht mehr abgelegt werden können.

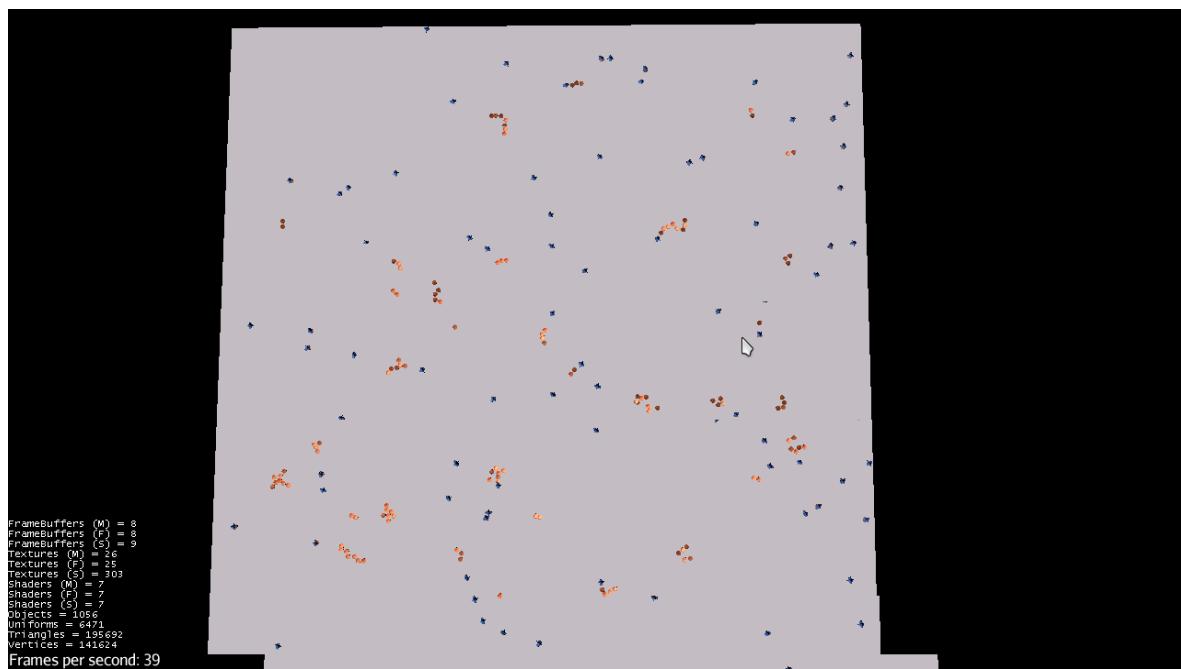


Abbildung 9: Kurz nach Start der Simulation, bereits schwache Haufenbildung

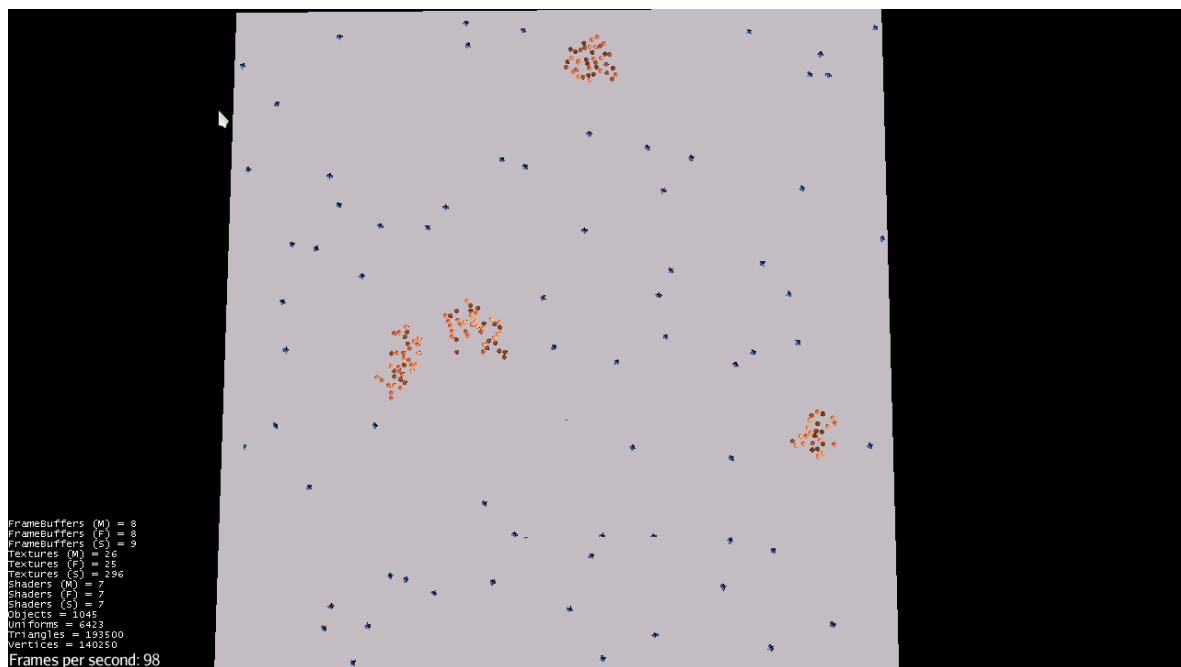


Abbildung 10: Fortgeschrittene Simulation, nur noch vier große Haufen übrig

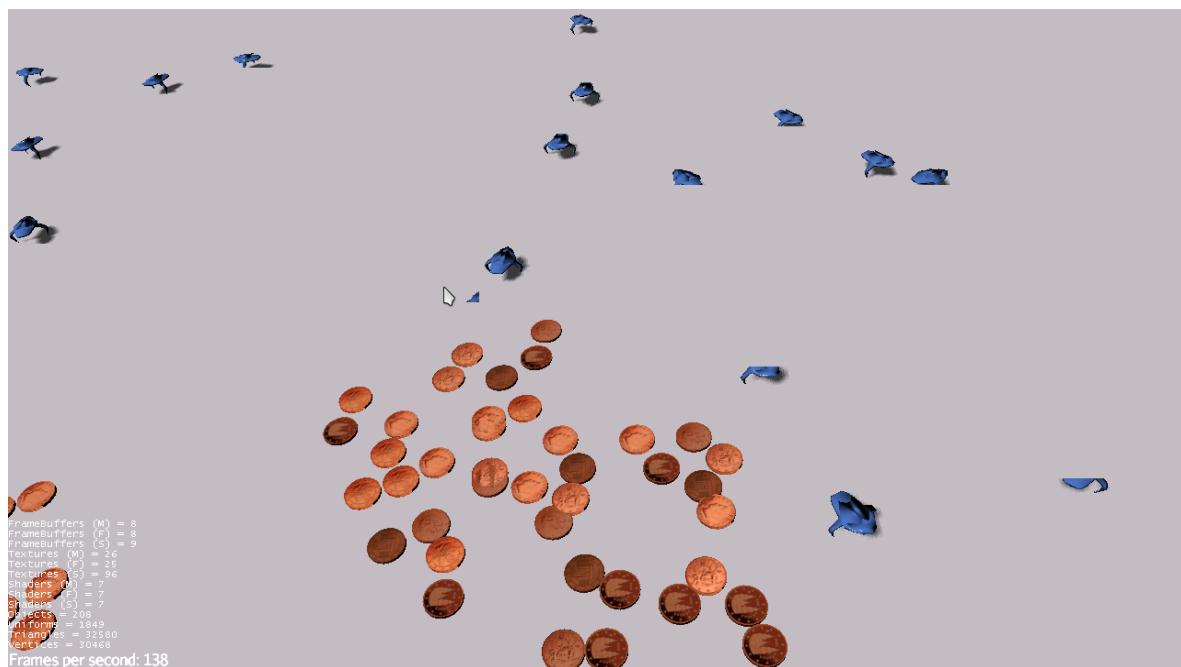


Abbildung 11: Einer der übriggebliebenen Haufen

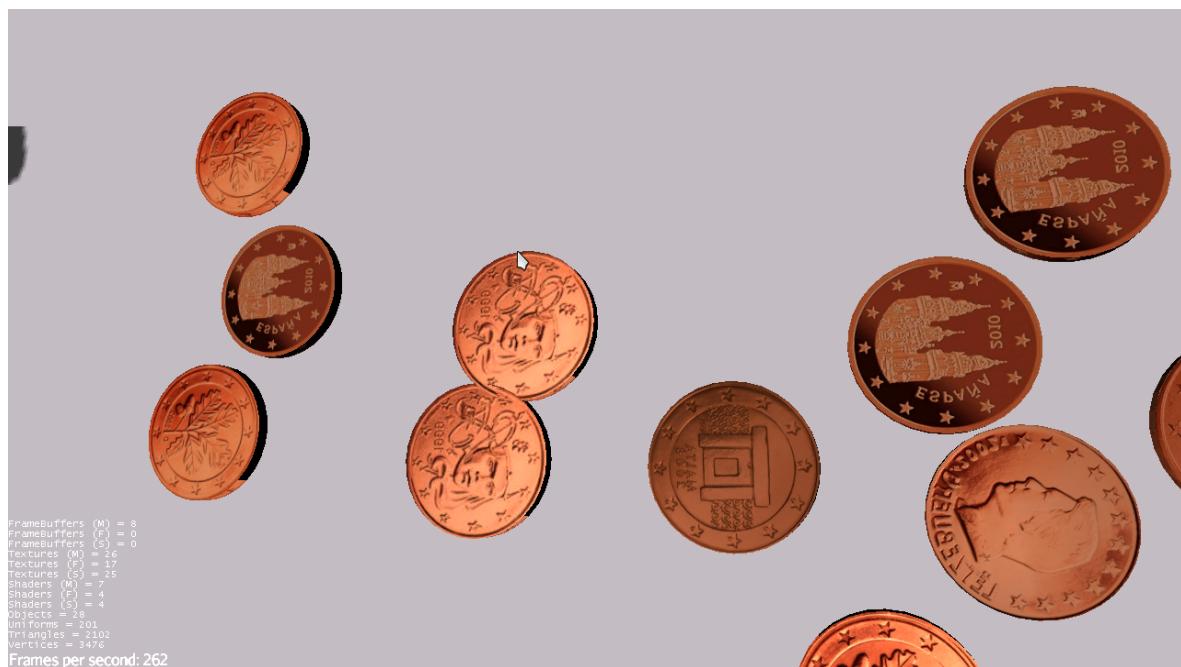


Abbildung 12: 5ct Münzen aus der Nähe

## 2.4 Programm-Text

Der Programm-Text ist wie folgt. Für das Multithreading ist einiger Zusatz-Code erforderlich. Dieser konnte nicht ausgelassen werden, da er teilweise direkt die Logik implementiert.

```

1 package tau.tim.robbuttons.core;
2
3 import java.util.concurrent.atomic.AtomicReference;
4 import net.jcip.annotations.ThreadSafe;
5
6 import tau.tim.robbuttons.math.Vec2;
7 import tau.tim.robbuttons.math.Utils;
8
9 /**
10 * Abstrakter Button mit Position und Radius
11 * Kollidierbar mit anderen Buttons
12 * @author Tim Taubner
13 */
14 @ThreadSafe
15 public abstract class Button implements Collidable<Button> {
16
17     private final double radius;
18     // Die Position in einem thread-sicherem Container:
19     private final AtomicReference<Vec2> loc =
20         new AtomicReference<Vec2>();
21
22     public Button(double radius, Vec2 loc) {
23         this.radius = radius;
24         this.loc.set(loc == null ? Utils.ZERO : loc);
25     }
26
27     public Vec2 getLoc() {
28         return loc.get();
29     }
30
31     protected final void setLoc(Vec2 loc) {
32         this.loc.set(loc);
33     }
34
35     protected final boolean setLoc(Vec2 loc_o, Vec2 loc_n) {
36         return loc.compareAndSet(loc_o, loc_n);
37     }
38
39     public double getRadius() {
40         return radius;
41     }
42
43     public boolean collidesWith(Button o) {
44         if(this == o) return false; // Nicht mit sich selbst kollidierbar.
45         // Ist Abstand < Summe der Radien?
46         double distance = loc.get().sub(o.getLoc()).length();
47         return distance < radius + o.getRadius();
48     }
49 }
```

```

1 package tau.tim.robbuttons.core;
2
3 import java.util.concurrent.atomic.AtomicReference;
4 import tau.tim.robbuttons.core.tasks.UpdateTask;
5 import tau.tim.robbuttons.core.tasks.Updatable;
```

```

6 import java.text.NumberFormat;
7 import tau.tim.robotttons.math.Vec2;
8 import java.util.ArrayList;
9 import java.util.Collection;
10 import java.util.Random;
11 import java.util.concurrent.Callable;
12 import net.jcip.annotations.ThreadSafe;
13
14 import static tau.tim.robotttons.math.Utils.*;
15
16 /**
17 * Robutton mit [fester] Geschwindigkeit und [veränderbarer] Bewegungsrichtung
18 * @author Tim Taubner
19 */
20 @ThreadSafe
21 public class Robutton extends Button implements Updatable {
22 private final double mps; // Geschwindigkeit in m/s
23 private final AtomicReference<Vec2> dir = new AtomicReference<Vec2>(X);
24 private final AtomicReference<Coin> c = new AtomicReference<Coin>();
25
26 public Robutton(double radius, Vec2 loc, double mps) {
27     super(radius, loc);
28     this.mps = mps;
29 }
30
31 public Vec2 getDir() { return dir.get(); }
32
33 public Robutton setDir(Vec2 dir) {
34     this.dir.set(dir);
35     return this;
36 }
37 private boolean setDir(Vec2 dir_o, Vec2 dir_n) {
38     return dir.compareAndSet(dir_o, dir_n);
39 }
40
41 public void onCollision(Button b) {
42     if(b instanceof Robutton) onCollision((Robutton) b);
43     else if(b instanceof Coin) onCollision((Coin) b);
44 }
45
46 // ===== Die Kollisionsregeln:
47 // ===== Mit Tisch:
48 public void onCollision(Table t, Vec2 normal) {
49     if(normal.angleBetween(dir.get().neg()) < HALF_PI)
50         reflect(normal);
51 }
52
53 // ===== Mit anderem Robutton:
54 private void onCollision(Robutton r) {
55     rot(RotAct.ROT180); // um 180 Grad drehen
56     rot(RotAct.ROT RAND); // nochmal um -90..90 Grad drehen
57 }
58
59 // ===== Mit Minze
60 private void onCollision(Coin c_n) {
61     boolean suc = false;
62     do {
63         Coin c_o = c.get();
64         if (c_o == c_n)
65             return;
66         if (c_o != null) {
67             suc = c.compareAndSet(c_o, null);
68             if(suc) {

```

```

69             c_o.drop(this);
70             rot(RotAct.ROT180); // um 180 Grad drehen nach Kollision
71         }
72     } else {
73         suc = c.compareAndSet(c_o, c_n);
74         if(suc)
75             c_n.take(this);
76     }
77 } while (!suc);
78 }

79
80 private void reflect(Vec2 normal) {
81     boolean suc = false;
82     do {
83         // Relektion durch Winkelrechnung am Einheitskreis:
84         Vec2 dir_new = this.dir.get();
85         double rad = normal.neg().angle() - dir_new.angle();
86         suc = setDir(dir_new, normal.rot(rad));
87     } while (!suc);
88 }

89 private void rot(RotAct act) {
90     boolean suc = false;
91     do {
92         Vec2 dir_o = getDir();
93         Vec2 dir_n = dir_o.rot(act.radians());
94         suc = setDir(dir_o, dir_n);
95     } while (!suc);
96 }

97 private void step(double secs) {
98     boolean suc = false;
99     do {
100        Vec2 loc = getLoc();
101        Vec2 loc_n = loc.add(dir.get().mul(mps*secs).cartesian());
102        suc = setLoc(loc, loc_n);
103    } while (!suc);
104 }

105
106 private static enum RotAct {
107     ROT180 { double radians(){ return PI; } },
108     ROTRAND{ double radians(){ return rand.nextDouble()*PI-HALF_PI; } },
109     final Random rand = new Random();
110     abstract double radians();
111 }

112 private static enum CoinAct{ DROP, TAKE };

113
114 public Collection<Callable<Boolean>> updateCalls() {
115     Collection<Callable<Boolean>> calls = new ArrayList<Callable<Boolean>>(5);
116     calls.add(new UpdateTask(240) { // Der Schrittgeber:
117         public boolean update(long millis) {
118             step(millis / 1000.0);
119             return false;
120         }
121     });
122     return calls;
123 }

124
125 @Override
126 public boolean collidesWith(Button o) {
127     // Prüfen ob \vec{V}=o.loc-this.loc o, etwa gleiche Richtung wie \vec{dir} hat.
128     Vec2 diff = o.getLoc().sub(getLoc());
129     if(diff.angleBetween(dir.get()) < HALF_PI) // Wenn Richtungsabweichung nicht < 90
130         if(o instanceof Coin)
131             return o.collidesWith(this);

```

```

132         else
133             return super.collidesWith(o);
134     return false;
135 }
136
137 //== Methoden ohne Logik ausgelassen ==
138 }

1 package tau.tim.robbuttons.core;
2
3 import tau.tim.robbuttons.math.Vec2;
4
5 /**
6 * Diese Klasse repräsentiert eine Minze.
7 * @author Tim Taubner
8 */
9 public class Coin extends Button implements Comparable<Coin> {
10 private final Table t;
11 // Teil des Radius, der nicht zur Kollisionsberechnung benutzt wird.
12 private final double bound;
13
14 public Coin(Table t, double radius, Vec2 loc) {
15     this(t, radius, loc, 0.1*radius);
16 }
17
18 public Coin(Table t, double radius, Vec2 loc, double bound) {
19     super(radius, loc);
20     this.t = t;
21     this.bound = bound;
22 }
23
24 // Nichts zu tun: Wird aufgenommen und fallengelassen von den Robuttons
25 public void onCollision(Button c) {}
26
27 public void drop(Robutton r) {
28     setLoc(r.getLoc());
29     t.addCoin(this, r);
30 }
31 public void take(Robutton r) {
32     t.remCoin(this, r);
33 }
34
35 @Override
36 public boolean collidesWith(Button o) {
37     if(this == o) return false; // Nicht mit sich selber kollidierbar.
38     double distance = getLoc().sub(o.getLoc()).length();
39     return distance < (getRadius()-bound + o.getRadius());
40 }
41
42 //== Methoden ohne Logik ausgelassen ==
43 }


```

```

1 package tau.tim.robbuttons.core;
2
3 import java.util.Collection;
4 import java.util.concurrent.ConcurrentLinkedQueue;
5 import java.util.concurrent.CopyOnWriteArraySet;
6 import java.util.logging.Level;
7 import java.util.logging.Logger;
8 import tau.tim.robbuttons.core.tasks.Updatable;


```

```

9 import tau.tim.robotttons.core.tasks.UpdateTask;
10 import tau.tim.robotttons.core.tasks.Updater;
11 import tau.tim.robotttons.math.Vec2;
12 import tau.tim.robotttons.shape.Shape;
13
14 /**
15 * Abstrakter Tisch mit Robuttons und Münzen(Coins)
16 *
17 * @author Tim Taubner
18 */
19 public abstract class Table implements Collidable<Button>, Updatable, Runnable {
20 private volatile boolean started;
21
22 private static final Logger log = Logger.getLogger(Table.class.getCanonicalName());
23 protected final Collection<Robutton> robs = new CopyOnWriteArraySet<Robutton>();
24 protected final Collection<Coin> coins = new ConcurrentLinkedQueue<Coin>();
25
26 private final Shape shape;
27
28 public Table(Shape shape) {
29     this.shape = shape;
30 }
31
32 public void addCoin(Coin c, Robutton r) {
33     coins.add(c);
34 }
35
36 public void remCoin(Coin c, Robutton r) {
37     coins.remove(c);
38 }
39
40 public void addRobutton(Robutton rob) {
41     robs.add(rob);
42     if(started)
43         createRobCollisionUpdates(rob);
44 }
45
46 public boolean collidesWith(Button r) {
47     Vec2 loc = r.getLoc();
48     double radius = r.getRadius();
49     boolean collided = shape.collidesWith(loc, radius);
50     if(collided && r instanceof Robutton)
51         onCollision(r);
52     return collided;
53 }
54 public void onCollision(Button r) {
55     if(r instanceof Robutton)
56         onCollision((Robutton) r, r.getLoc(), r.getRadius());
57 }
58 private void onCollision(Robutton r, Vec2 loc, double radius) {
59     r.onCollision(this, reflectNormal(loc, radius));
60 }
61 private Vec2 reflectNormal(Vec2 loc, double radius) {
62     return shape.reflectNormal(loc, radius);
63 }
64
65 // Beginne die Simulation
66 public void run() {
67     started = true;
68     Updater.UPDATER.update(this);
69     for (Robutton rob : robs) {
70         Updater.UPDATER.update(rob);
71         createRobCollisionUpdates(rob);

```

```

72     }
73 }
74
75 private void createRobCollisionUpdates(final Robutton rob) {
76     final Table table = this;
77     try {
78         Updater.UPDATER.execute(new UpdateTask(800) {
79             public boolean update() {
80                 table.collidesWith(rob)
81                 return false;
82             }
83         });
84         Updater.UPDATER.execute(new UpdateTask(60) {
85             public boolean update() {
86                 for (Robutton r_1 : robs)
87                     if (rob.collidesWith(r_1)) rob.onCollision(r_1);
88                 return false;
89             }
90         });
91         Updater.UPDATER.execute(new UpdateTask(600) {
92             public boolean update() {
93                 for (Coin coin : coins)
94                     if (rob.collidesWith(coin)) rob.onCollision(coin);
95                 return false;
96             }
97         });
98     } catch (InterruptedException ex) {
99         log.log(Level.SEVERE, null, ex);
100    }
101 }
102 }
```

**LineShape.java:** Dies ist der Programm-Text, welcher die Kollisionserkennung in einem Tisch berechnet, gekürzt und vereinfacht.

```

1 package tau.tim.robuttons.shape;
2
3 import tau.tim.robuttons.math.Radial2;
4 import tau.tim.robuttons.math.Vec2;
5
6 import static java.lang.Math.*;
7
8 /**
9  * Teilweise vereinfachte Klasse, welche eine Linie repräsentiert.
10 * Jedes Polygon kann auf eine Menge Linien abgebildet werden.
11 *
12 * @author Tim Taubner
13 */
14 public class LineShape implements Shape {
15 // Der Vektor, welche die Gerade beschreibt:
16 private final Radial2 line;
17 // Die Normale in Richtung Ursprung
18 private final Vec2 normal;
19
20 public LineShape(Vec2 line) {
21     this.line = line.radial();
22     // Geradenvektor normalisieren und negieren (Richtung Ursprung!)
23     this.normal = line.normal().neg();
24 }
25
26 public boolean collidesWith(Vec2 loc, double radius) {
27     Radial2 ang = loc.radial(); // Radialer Vektor (Kreiskoordinate) der Position
```

```

28     double alpha = line.angleBetween(ang); // Zwischenwinkel
29     // Entfernung vom Ursprung
30     double l = ang.length() + radius;
31     // a = Teil des Vektors auf der senkrechte der Geraden
32     // cos(alpha) = a / l ⇒ a = cos(alpha) * l
33     double a = cos(alpha) * l;
34     // Abstand:
35     double dist = a - line.length();
36     return dist >= 0; // Ist Radius+Abstand > Abstand der Gerade vom Ursprung?
37 }
38
39 /**
40 * Gibt die Normale zur Reflektion zurück.
41 */
42 public Vec2 reflectNormal(Vec2 loc, double radius) {
43     return normal;
44 }
45
46 }
```

## 2.5 Programm

**Systemvoraussetzungen:** (zusätzlich zu den allgemeinen Voraussetzungen, siehe 5.2): Da bei diesem Programm Multithreading benutzt wird, sollte der Prozessor wenigstens 2 Kerne haben, um diese Funktion ausnutzen zu können. Außerdem muss die Grafikkarte OpenGL2 implementieren, um die dreidimensionale Ansicht anzeigen zu können. Eine möglichst hohe Rechenleistung der Grafikkarte, aber auch des Prozessors ermöglicht eine hohe Anzahl gleichzeitig simulierter Robuttons und Münzen.

**Ausführung:** Aufgrund von Beschränkungen durch die benutzte Engine, muss die Anwendung direkt im ‘dist’ Verzeichnis ausgeführt werden. Wechseln Sie daher zunächst aus dem Wurzelverzeichnis in das richtige Verzeichnis mit ‘cd Aufgabe2/dist’. Die Ausführung erfolgt nun durch Eingabe des Befehles ‘java -Djava.library.path=. -Xmx1024M -jar 3DRobuttons.jar’ in der Konsole.

Es öffnet sich ein Fenster jME (jMonkeyEngine), in dem Sie aufgefordert sind, Ihre gewünschten Anzeigeeinstellungen anzugeben. Empfohlen ist die Auswahl von Fullscreen und dann der maximalen Auflösung (z.B. 1280x720). Wenn Sie einen langsameren Rechner haben, wählen Sie bitte eine niedrigere Auflösung. Stellen Sie im 2. Feld ebenfalls den höchsten Wert ein (unter Windows “32 bpp”, Linux/Unix “24 bpp”). Das Feld “disabled” bitte nicht verändern. Nach Bestätigen mit OK öffnet sich ein weiteres Fenster mit 3D-Ansicht, in der Sie einen quadratischen Tisch mit Robuttons und Münzen sehen können.

**Steuerung:** Die Steuerung in der 3D-Ansicht, die sich öffnet, ist folgende:

STEUERUNG	EINGABEMETHODE, z.B. TASTE
Simulationsstart	Leertaste
Kamera Hoch/Runter	Tasten W/S
Kamera Links/Rechts	Tasten A/D
Kamera-Rotation	Maus hoch/runter, links/rechts für Rotation um die Kameraachsen
Kamera-Zoom	Mausrad
Simulationsende	Esc (oder zur Not Alt+F4)

### 3 Aufgabe 3: “Logistisch”

#### 3.1 Lösungsidee

##### 3.1.1 Teilaufgabe 1

Man betrachte für jeden Standort A, B und C die Fahrzeuge, die an jedem der fünf Tage dort stehen sollen. Eine negative Anzahl Fahrzeuge signalisiert dabei einen entsprechenden Bedarf an Fahrzeugen. Nun rechnet man den Bedarf an Fahrzeugen für jeden Tag und Standort wie folgt aus: Seien zu Beginn der Woche an jedem Standort 0 Fahrzeuge. Anhand des Tourenplans, an dem ablesbar ist, wieviele Fahrzeuge von welchem Standort zu welchem fahren, lässt sich die Anzahl der erforderlichen Fahrzeuge des nächsten Tages berechnen. So werden auch alle Folgetage berechnet. Für jeden Standort wird nun der Tag gesucht an dem der Bedarf an Fahrzeugen am höchsten ist (die Zahl obiger Rechnung also am kleinsten, “negativsten” ist). Ist dieser errechnete Wert negativ, so ist der Betrag dieses Wertes die Startanzahl, ist er jedoch positiv, genügt der Startwert 0. Nachdem dies für jeden Standort erfolgt ist, hat man die Startanzahlen für alle Standorte A, B und C.

##### 3.1.2 Teilaufgabe 2

(Zu dieser Teilaufgabe 2 wurden zwar Überlegungen angestellt, für eine Implementierung fehlte jedoch die Zeit.) Um nur eine möglichst niedrige Anzahl Fahrzeuge bereitzustellen zu müssen, können Leerfahrten benutzt werden, um überschüssige Fahrzeuge an einem Standort, zu einem anderen zu bringen. Hier kann die in Teil 1 erzeugte Liste für jeden Standort mit theoretischen - also auch negativen - Werten benutzt werden. Wenn an einem Standort negative Werte auftauchen, wird versucht von den beiden anderen Standorten Fahrzeuge durch Leerfahrten bereitzustellen. Da es Ziel ist, den minimalen negativen Wert auszugleichen, sollten immer an dem Tag mit dem minimalen Wert Optimierungsversuche erfolgen. Nach einem Optimierungsversuch werden die Listen neu berechnet und bei dem neuem Minimum weiteroptimiert. Wichtig ist, dass eine Leerfahrt nur dann erfolgt, wenn dieses Fahrzeug nicht nächsten Tag am derzeitigen Standort gebraucht wird.

#### 3.2 Programm-Dokumentation

Aus den Vorüberlegung folgt: Die Startanzahl der Fahrzeuge für einen Standort ist der Betrag des Minimums aller kalkulierten Fahrzeuganzahlen an einem jedem Tag und dem Startwert 0.

**Implementierung:** Nach dem Einlesen der Daten aus einer Datei, werden zunächst die Differenzen zwischen den Ankünften und Abfahrten von Fahrzeugen berechnen. Diese Werte errechnen sich aus der Summe der Zufahrten von den beiden anderen Standorten minus die Summe der Abfahrten zu den anderen beiden Standorten. Bei mehr Ab- als Zufahrten ist der Differenzwert somit negativ. Da die kalkulierte Fahrzeuganzahl eines Tages sich aus der Addition des Differenzwertes mit dem Vortageswert errechnet, können Listen mit den Fahrzeuganzahlen rekursiv an dem ersten Tag beginnend berechnet werden (durch eine Hilfsfunktion “berechneStandort” und dem theoretischen Startwert 0). Nachdem an jede Liste eine 0 (Startwert) angehängt wurde, wird der Betrag des Minimums dieser Liste als tatsächlich benötigter Startwert für diesen Standort zurückgegeben. (Der Startwert 0 wird angehängt, um ein nicht positives Minimum zu erhalten.)

**Testfälle:** Es wurde eine Datei TestCases.scala erstellt, in der ein “JUnit TestCase” implementiert ist. Dieser enthält verschiedene Eingaben die getestet werden, unter anderem die Mustereingaben aus der Materialsektion von der Webseite des BWINFs.

### 3.3 Programm-Ablaufprotokoll

Ablauf mit einer Mustereingabe (die anderen wurden ebenfalls getestet, dies ist hier jedoch nicht abgebildet):

```

1 # A->B A->C B->A B->C C->A C->B
2 1 2 3 4 5 6
3 2 3 4 5 6 1
4 3 4 5 6 7 8
5 1 2 3 4 5 6
6 1 2 3 4 5 6
7 4 3 5 6 1 2

```

Wird eingelesen und folgt zu diesem Ablaufprotokoll:

```
1 Fahrzeugdifferenzen für Standort A:
```

```

2      5
3      5
4      5
5      5
6      5
7     -1

```

```
8 Fahrzeugdifferenzen für Standort B:
```

```

9      0
10     -6
11      0
12      0
13      0
14     -5

```

```
15 Fahrzeugdifferenzen für Standort C:
```

```

16     -5
17      1
18     -5
19     -5
20     -5
21      6

```

```

22 Standort A: aktueller Zwischenwert: 5
23 Standort A: aktueller Zwischenwert: 10
24 Standort A: aktueller Zwischenwert: 15
25 Standort A: aktueller Zwischenwert: 20
26 Standort A: aktueller Zwischenwert: 25
27 Standort A: aktueller Zwischenwert: 24
28 Standort B: aktueller Zwischenwert: 0
29 Standort B: aktueller Zwischenwert: -6
30 Standort B: aktueller Zwischenwert: -6
31 Standort B: aktueller Zwischenwert: -6
32 Standort B: aktueller Zwischenwert: -6
33 Standort B: aktueller Zwischenwert: -11
34 Standort C: aktueller Zwischenwert: -5
35 Standort C: aktueller Zwischenwert: -4
36 Standort C: aktueller Zwischenwert: -9
37 Standort C: aktueller Zwischenwert: -14
38 Standort C: aktueller Zwischenwert: -19
39 Standort C: aktueller Zwischenwert: -13
40 Anzahl für Standort A: 0
41 Anzahl für Standort B: 11
42 Anzahl für Standort C: 19

```

### 3.4 Programm-Text

Der Code ist entnommen aus Aufgabe3/src/tau/tim/logistik/TourenPlan.scala auf der CD. Zum Ausführen ist zwar noch zusätzlich Code erforderlich, wie z.B. der Dateieinlesecode, jedoch wird hier nur der inhaltlich relevante Teil eingefügt.

```

1  /**
2   * Berechnet aus einer Liste mit je 6 Werten eine mögliche Zahl an Fahrzeugen
3   * für jeden Standort A, B, und C.
4   * Die Wert in der Liste sind wie in den Beispieldateien:
5   * A->B, A->C, B->A, B->C, C->A, C->B
6   *
7   * @param einAus: Die Liste mit den Werten.
8   * @param log: Eine Funktion zur Ausgabe des Protokolls
9   */
10 def berechne(einAus: List[(Int, Int, Int, Int, Int, Int)], log: String ⇒ _) = {
11   // Aus den Spalten entsprechende Werte einlesen
12   // Berechnet gleichzeitig die Differenz zwischen Ein- und Ausfuhren für jeden Tag.
13   val a_diff = for(i ← einAus) yield i._3+i._5 - (i._1+i._2)
14   val b_diff = for(i ← einAus) yield i._1+i._6 - (i._3+i._4)
15   val c_diff = for(i ← einAus) yield i._2+i._4 - (i._5+i._6)
16
17   log { "Fahrzeugdifferenzen für Standort A:" + a_diff.mkString("\n", "\n", "") }
18   log { "Fahrzeugdifferenzen für Standort B:" + b_diff.mkString("\n", "\n", "") }
19   log { "Fahrzeugdifferenzen für Standort C:" + c_diff.mkString("\n", "\n", "") }
20
21   // Die Fahrzeugzahlen für jeden Standort:
22   val a = 0 :: berechneStandort(0, a_diff, (s: String) ⇒ log{ "Standort A:" + s})
23   val b = 0 :: berechneStandort(0, b_diff, (s: String) ⇒ log{ "Standort B:" + s})
24   val c = 0 :: berechneStandort(0, c_diff, (s: String) ⇒ log{ "Standort C:" + s})
25
26   // Die Anzahl der Fahrzeuge für jeden Standort ist das Minimum.
27   val res = (-a.min, -b.min, -c.min) // Minimum <= 0, weil 0 am Anfang angehängt wurde.
28   log { "Anzahl für Standort A:" + res._1 }
29   log { "Anzahl für Standort B:" + res._2 }
30   log { "Anzahl für Standort C:" + res._3 }
31   res
32 }
33
34 private def berechneStandort(vorher: Int, diffs: Seq[Int], log: String ⇒ _): List[Int] =
35   diffs.headOption match {
36     case Some(diff) ⇒
37       // Der jetzige Wert ist der vorherige plus die Differenz aus den Ein- und Ausfuhren
38       val jetzt = vorher + diff
39       log { "aktueller Zwischenwert:" + jetzt }
40       jetzt :::
41       // Rekursiver Aufruf mit Zwischenwert jetzt und Restwerten diffs.tail
42       berechneStandort(jetzt, diffs.tail, log)
43     case _ ⇒ Nil // wenn keine Werte mehr vorhanden, Rekursionsschluss.
44   }

```

### 3.5 Programm

Das Programm kann durch den Befehl ‘java -jar Aufgabe3/dist/Logistik.jar’ ausgeführt werden (Voraussetzungen und Startanleitung siehe 5.2). Die Eingabedaten sind in der Datei Aufgabe3/dist/table.txt. Diese wird zu Programmbeginn eingelesen. Um diese Mustereingabe zu ändern, muss der gesamte Aufgabe3 Ordner in ein beschreibbares Verzeichnis kopiert werden. Geben Sie dann Ihre gewünschte Eingabe in die Aufgabe3/dist/table.txt Datei. Führen Sie den Befehl statt auf dem Wurzelverzeichnis der CD dann auf dem entsprechendem Verzeichnis aus.

## 4 Aufgabe 4: “Drehzahl”

### 4.1 Lösungsidee

Um nach dem Würfeln einer Zahl, die optimale Entscheidung zu treffen, wird für jede Möglichkeit Karten zu streichen, die erwartete Höchstpunktzahl berechnet. Es wird diejenige Streichung ausgewählt, von der die maximale Punktzahl erwartet wird.

**Das Berechnen der Höchstpunktzahl erfolgt rekursiv:**

**begin**

```

for all Zahl z ← Würfelbare Zahlen do
    if Zahl z nicht aus Karten streichbar then
        Punktzahl ist Summe der Kartenbeträge {Rekursionsschluss}
    else
        for all Streichmöglichkeit s ← Streichmöglichkeiten für die Zahl z do
            Kartensatz k ← Kartensatz nach Streichen der Streichmöglichkeit s
            Berechne die Punktzahl des Kartensatzes k {Rekursiver Aufruf}
        end for
        Punktzahl ist das Maximum der zuvor berechneten Punktzahlen
    end if
    Die gewichtete Punktzahl ist die soeben errechnete Punktzahl mal die Wahrscheinlichkeit
    der Zahl z.
end for
Addiere alle gewichteten Punktzahlen
end
```

**Mit diesem Algorithmus lässt sich nun die Entscheidung wie folgt berechnen:**

**begin**

```

Zahl z ist die gewürfelte Zahl
if Zahl z nicht aus Karten streichbar then
    Gewonnen mit Punktzahl p ← Summe der Kartenbeträge
else
    for all Streichmöglichkeit s ← Streichmöglichkeiten für die Zahl z do
        Kartensatz k ← Kartensatz nach Streichen der Streichmöglichkeit s
        Berechne die Punktzahl des Kartensatzes k {mit Hilfe des oben beschriebenen Algo-
        rithmus'}
    end for
    Entscheidung fällt für die Streichmöglichkeit mit der höchsten Punktzahl
end if
end
```

### 4.2 Programm-Dokumentation

**Maximale Punktzahl** Durch den Algorithmus zur Berechnung der Höchstpunktzahl wurde der Wert  $24,34456$  ausgerechnet. Im Experiment (1000-maliges Ausführen von OptimalSpiel, siehe unten) wurde der Wert  $23,646$  empirisch berechnet, dies ergibt eine Differenz von lediglich  $0,69856$ .

### 4.3 Programm-Ablaufprotokoll

**Ablauf eines “OptimalSpiel”** Ein sogenanntes OptimalSpiel ist ein protokolliertes Spiel des Computers. Hier wird der oben beschriebene Algorithmus zur Entscheidungsfindung verwendet.

```

1 OptimalSpiel startet .
2 Karten sind: [1|2|3|4|5|6|7|8|9]
3 Gewürfelt wurde 6
4 => Streichmöglichkeiten: [1|2|3],[1|5],[2|4],[6]
5 Es wird [1|2|3] gestrichen (erwartete Punktzahl: 24,24562492590899)
6
7 Karten sind: [4|5|6|7|8|9]
8 Gewürfelt wurde 6
9 => Streichmöglichkeiten: [6]
10 Es wird [6] gestrichen (erwartete Punktzahl: 24,316367550678248)
11
12 Karten sind: [4|5|7|8|9]
13 Gewürfelt wurde 8
14 => Streichmöglichkeiten: [8]
15 Es wird [8] gestrichen (erwartete Punktzahl: 19,45130315500686)
16
17 Karten sind: [4|5|7|9]
18 Gewürfelt wurde 2
19 Keine Streichmöglichkeit !
20
21 Spielende! – Es wurden 25 Punkte erreicht !

```

### 4.4 Programm-Text

Der inhaltlich interessante Code Aufgabe4/src/tau/tim/drehzahl/Karten.scala:

```

1 package tau.tim.drehzahl
2
3 import Wuerfel.{zahlen, wahrscheinlichkeit}
4
5 /**
6  * Ein Kartensatz, z.B. der Handkartensatz
7  * Definiert Funktionen zur Berechnung
8  * (a) der Durchschnittspunktzahl
9  *      (für eine Zahl oder alle Zahlen gewichtet nach Wahrscheinlichkeit)
10 * (b) der Möglichkeiten für Streichsätze einer gegebenen Zahl,
11 * (c) eines neuen Kartensatz nach Streichung eines Anderen.
12 */
13 class Karten(private val karten : List[Int]) extends Ordered[Karten] {
14   lazy val punkte = karten.sum
15   lazy val gewichtetePunkte: Double = gewichtetePunkteListe.sum
16
17 /**
18  * Berechne für jede mögliche Würfelzahl die erwartete Punktzahl,
19  * gewichtet nach Wahrscheinlichkeit
20 */
21 private lazy val gewichtetePunkteListe: Seq[Double] =
22   for(zahl <- zahlen)
23     yield wahrscheinlichkeit(zahl)*gewichtetePunkte(streichbar(zahl))
24 // Rekursive Hilfsfunktion zur Punkteberechnung:
25 private def gewichtetePunkte(streichbar: List[Karten]): Double =
26   // Rekursionsschluss, wenn keine Karte streichbar
27   if(streichbar.isEmpty) punkte
28   else { // Rekursiver Aufruf:
29     // Errechnen der Punktzahl für alle durch Streichen erhaltene Kartensätze
30     val ges = for(s <- streichbar) yield streiche(s).gewichtetePunkte

```

```

31      // Erwartete Punktzahl ist das Maximum
32      ges.max
33  }
34
35 /**
36 * Berechnet die durchschnittlich zu erwartende Punktzahl bei Würfeln der
37 * angegebenen Zahl.
38 * @param zahl Die Würfelzahl
39 */
40 def gewichtetePunkte(zahl: Int) : Double = gewichtetePunkteListe(zahl)
41
42 /**
43 * Streicht die gegebenen Karten aus diesem Kartensatz und gibt
44 * diesen neuen Kartensatz zurück.
45 * @param that Die rauszustreichenden Karten
46 */
47 def streiche(that: Karten): Karten =
48 if(that.karten.toSet.subsetOf(this.karten.toSet)) {
49     val ret = Karten(karten diff that.karten)
50     ret
51 } else throw new IllegalArgumentException()
52
53 /**
54 * Berechnet für eine gegebene Zahl nach den Spielregeln mögliche Kartensätze,
55 * um diese Zahl aus dem Kartensatz zu streichen.
56 * @param num Die zu streichende Zahl
57 */
58 def streichbar(num: Int) : List[Karten] = streichbar(num, Nil, karten)
59
60 /**
61 * Rekursive Hilfsfunktion zu streichbar
62 * @param nums: Die Zahlen sind abnehmend geordnet
63 */
64 private def streichbar(num: Int, nums: List[Int], karten: List[Int]): List[Karten] =
65 if(num < 0) Nil // Abbruch bei negativen Zahlen.
66 // Rekursionsschluss, drehe Zahlen für steigende Sortierung
67 else if(num == 0) Karten(nums.reverse) :: Nil
68 // Rekursionsschluss wenn keine Zahlen mehr vorhanden
69 else if(karten.isEmpty) Nil
70 else {
71     val i = karten.head // Betrag der betrachteten Karte
72     // Rekursiver Aufruf, einmal nach streichen (abziehen) des Betrages
73     streichbar(num-i, i :: nums, karten.tail) :::
74     streichbar(num,           nums, karten.tail) // ← einmal ohne
75 }
76
77 // Ordnung nach Punkten:
78 def compare(andere : Karten) = punkte - andere.punkte
79 override lazy val toString = karten.mkString("[", "|", "]")
80 }
81
82 object Karten extends Karten(Nil) {
83     // Hilfsfunktionen zur einfachen Erzeugung von Kartensätzen
84     def apply(karten: Int*) : Karten = apply(karten.toList)
85     def apply(karten: List[Int]) =
86         new Karten(karten.toList.sortWith((x,y) ⇒ x < y))
87
88 // Der Standardkartensatz
89 lazy val standard = apply(1,2,3,4,5,6,7,8,9)
90 }

```

#### 4.5 Programm

Der Code ist ausführbar auf der CD mitgeliefert.

Das Programm kann durch folgenden Befehl aus der Kommandozeile ausgeführt werden (Voraussetzungen siehe 5.2): 'java -jar Aufgabe4/dist/Drehzahl.jar'.

## 5 Allgemeine Nutzungsanleitung

### 5.1 Dateistruktur der CD

Die Dateien auf der CD sind folgendermaßen strukturiert. Jede Aufgabe hat einen Ordner AufgabeX mit den beiden folgenden Unterordnern.

**src** Unterordner, in dem die Quelltexte in der Paketstruktur (tau/tim/..) liegen

**dist** Unterordner, in dem ausführbare Dateien oder - wie bei Aufgabe 1 - andere Erzeugnisse sowie benötigte Bibliotheken enthalten sind

Zusätzlich ist die vorliegende Dokumentation digital unter **Doku-Einsendung-108-Tim-Taubner.pdf** im Wurzelverzeichnis zu finden. Auch die L<sup>A</sup>T<sub>E</sub>X-Quelldateien, mit der diese Dokumentation erzeugt wurden, ist im Verzeichnis "TeX-Quelldateien-Doku" zu finden.

### 5.2 Ausführvoraussetzungen

Um die Programmbeispiele ausführen zu können, müssen folgende Voraussetzungen erfüllt werden:

**Aufgabe 1:** cfdg muss vorinstalliert sein

**Aufgabe 2, 3, 4:** Java Version 5, besser 6, muss installiert sein.

Zusätzlich sind folgende Systemvoraussetzungen zu erfüllen (beachten Sie bitte die zusätzlichen Anforderungen bei Aufgabe 2):

**Prozessor:** Mindestens 1 GHz, empfohlen:  $\geq 1.6 \text{ GHz}$

**Arbeitsspeicher:** Mindestens 512 MB, empfohlen:  $\geq 1 \text{ GB}$

**Grafikkarte:** beliebig

**Getestete Betriebssysteme:** Windows 7, Windows XP, Linux (Ubuntu 10.04, Kubuntu 10.10)

### 5.3 Starten der Programme

**Wurzelverzeichnis** Im Wurzelverzeichnis der CD beginnt die Ordnerhierarchie der mitgelieferten Dateien. Stellen Sie bitte sicher, dass Sie im Wurzelverzeichnis sind, bevor Sie die in den jeweiligen Aufgaben beschriebene Startanleitungen ausführen. (z.B. durch neues Starten der Kommandozeile gemäß folgender Anleitung)

**Starten der Kommandozeile** Da die meisten mitgelieferten Programme aus der Kommandozeile gestartet werden müssen, soll hier kurz erläutert werden, wie Sie die Kommandozeile unter den gängigeren Betriebssystemen starten können.

**Unter Windows** Unter Windows starten Sie die Kommandozeile durch: Start → Ausführen → 'cmd' eingeben → Kommandozeile. Nun können Sie durch Angabe des Laufwerkbuchstabs des CD-Laufwerks (z.B. "E:") auf das Wurzelverzeichnis der CD wechseln. Alle Befehle können nun durch Copy&Paste entsprechend der Nutzungsdocumentation der jeweiligen Aufgabe ausgeführt werden. Z.B. für Aufgabe 3 mit:

```
java -jar Aufgabe3/dist/Logistik.jar
```

**Unter GNOME** Unter gängigeren GNOME Distributionen wie z.B. Ubuntu 8 starten sie die Kommandozeile durch: Applikationen → System → Terminal. Die CD wird unter Standard-distributionen unter /media/disk o.ä. eingehängt. Wechseln Sie durch ‘cd /media/disk’ in das Wurzelverzeichnis der CD. Alle Befehle können nun durch Copy&Paste entsprechend der Nutzungsdocumentation der jeweiligen Aufgabe ausgeführt werden. Z.B. für Aufgabe 3 mit:

```
java -jar Aufgabe3/dist/Logistik.jar
```

**Unter KDE** Unter gängigeren KDE-Distributionen wie z.B. Kubuntu 9 starten sie die Kommandozeile durch: Start → Applikationen → System → Terminal. Die CD wird unter Standard-distributionen unter /media/disk o.ä. eingehängt. Wechseln Sie durch ‘cd /media/disk’ in das Wurzelverzeichnis der CD. Alle Befehle können nun durch Copy&Paste entsprechend der Nutzungsdocumentation der jeweiligen Aufgabe ausgeführt werden. Z.B. für Aufgabe 3 mit:

```
java -jar Aufgabe3/dist/Logistik.jar
```

**Unter Mac OS X** Leider steht mir kein Mac zur Benutzung bereit, das Öffnen der Konsole sollte jedoch entweder selbsterklärend oder ähnlich der unter KDE/GNOME sein. (Beachten Sie bitte, dass Aufgabe2 leider nicht unter Mac OS X lauffähig ist)