Simulation des Dreikörperproblems mit Hilfe der Programmiersprache JAVA

Johannes Buchner

15. August 2000

Zusammenfassung

Thema dieser Arbeit ist das Dreikörperproblem, ein Spezialfall des allgemeinen n-Körperproblems. Dieses beschäftigt sich mit der Frage, wie sich massebehaftete Körper aufgrund ihrer Gravitationskräfte bewegen. Hierbei ist natürlich besonders die Bewegung von Himmelskörpern interessant, z.B. das Umkreisen der Sonne durch die neun Planeten oder die Frage, ob ein Komet die Erde trifft oder sie nur passiert etc. Astronomen beschäftigen sich seit langem mit solchen Fragestellungen, aber erst durch die Rechenleistung heutiger PC's wurde eine Computersimulation derartiger Probleme möglich. Die rasante technische Entwicklung im Bereich der Computerhardware eröffnet hier also zahlreiche neue Möglichkeiten, diese in der Praxis auszuprobieren war das Ziel dieser Facharbeit. Das vorliegende Computerprogramm simuliert die Bewegung dreier Himmelskörper unter Einfluss ihrer Anziehungskräfte, es soll nun der Weg von Newtons Gravitationsgesetzen bis hin zur fertigen Computersimulation aufgezeigt werden.

Inhaltsverzeichnis

Teil I

Theoretischer Teil

1 Das Dreikörperproblem als Spezialfall des n-Körperproblems

1.1 Mathematische und physikalische Grundlagen

Unter der Vorraussetzung, dass man die zu untersuchenden (Himmels-)Körper in guter Näherung als Massepunkte auffassen kann, sollen nun n Massepunkte mit den Massen m_i $(i = 1, 2, \ldots, n)$ betrachtet werden, die sich nach dem NEWTONschen Gravitationsgesetz anziehen. Ihre Positionen im Raum seien durch die Ortsvektoren $\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n$ gegeben, der Abstand zweier Massen i und j sei $r_{ij} = r_{ji} = |\vec{p}_j - \vec{p}_i|$. Dann ergibt sich nach dem NEWTONschen Gravitationsgesetz für den Betrag der Kraft, mit der die Masse m_i von der Masse m_j angezogen wird:

$$\left|\vec{F}_{ij}\right| = \mu \frac{m_i m_j}{r_{ij}^2} \tag{1}$$

Die Richtung dieser Kraft ist durch $(\vec{p}_j - \vec{p}_i)$ festgelegt , sodass sich der Kraftvektor als Produkt des Einheitsvektors in diese Richtung $\frac{\vec{p}_j - \vec{p}_i}{|\vec{p}_j - \vec{p}_i|}$ und seinem Betrag $\left|\vec{F}_{ij}\right|$ergibt:

$$\vec{F}_{ij} = \mu \frac{m_i m_j}{r_{ij}^2} \frac{\vec{p}_j - \vec{p}_i}{|\vec{p}_j - \vec{p}_i|} = \mu \frac{m_i m_j}{r_{ij}^3} (\vec{p}_j - \vec{p}_i)$$

Betrachtet man nun die Kräfte, die auf einen Körper m_i von allen anderen $n - 1$ Massen ausgeübt werden, so ergibt sich folgende resultierende Kraft auf m_i :

$$\vec{F}_i = \mu m_i \sum_{i \neq j}^{n} \frac{m_j}{r_{ij}^3} (\vec{p}_j - \vec{p}_i) \tag{2}$$

Gleichzeitig muss für die resultierende Kraft nach der Grundgleichung der

Mechanik ebenfalls gelten: $\vec{F}_i = m\vec{a}_i = m\ddot{\vec{p}}_i$, weil die Beschleunigung ja die 2. Ableitung des Ortes nach der Zeit ist. Es gilt folglich:

$$\vec{F}_i = m\ddot{\vec{p}}_i = \mu m_i \sum_{i \neq j}^{n} \frac{m_j}{r_{ij}^3} (\vec{p}_j - \vec{p}_i)$$

Dies stellt ein System von n vektoriellen Differentialgleichungen zweiter Ordnung dar, für das bisher bis auf den Spezialfall $n = 2$, dem sogenannten *Zweikörperproblem*, keine geschlossene Lösung gefunden wurde. Das bedeutet, dass man bei einem Mehrkörperproblem mit $n > 2$ keine exakte mathematische Funktion angeben kann, die den Ort eines Körpers in Abhängigkeit von der Zeit angibt. In der Praxis ist dies allerdings oft auch gar nicht nötig, denn es ist gelungen, diese Schwierigkeiten zumindest teilweise zu umgehen und das Dreiköperproblem - speziell auch seine Anwendung auf das Planetensystem - in sehr guter Näherung zu behandeln.

1.2 Analytische Methoden

Die Grundidee hierbei ist die Einschränkung des sehr allgemeinen Dreikörperproblems in einer Weise, so dass es mathematisch besser handhabbar wird. Man versucht also, die Problemstellung zu vereinfachen, in man gewisse einschränkende Vorraussetzungen über Art und Bewegung des Systems macht. Geht man beispielsweise davon aus, dass sich die drei Körper nur in einer Ebene statt im Raum bewegen, so spricht man vom *ebenen Dreikörperproblem.* Man erhält zwar eine größere Einfachheit und Übersichtlichkeit der Formeln, die eigentlichen Schwierigkeiten der Integration werden allerdings nicht wesentlich geringer.

In der astronomischen Praxis ist auch dass sog. *eingeschränkte Dreikörperproblem* von Bedeutung, welches auf folgenden, die Allgemeinheit des Dreikörperproblems schon recht stark einschränkenden Annahmen beruht, die aber in unserem Sonnensystem vielfach erfüllt sind:

- Das System besteht aus zwei endlichen Massen m_1 und m_2, und einer viel kleineren, zur Vereinfachung unendlich klein gesetzten Masse $m_3 =$

0, die sich in der gleichen Ebene wie die beiden anderen Massen bewegt und deren Anziehungskräfte auf diese vernachlässigt werden darf.

- Die beiden endlichen Massen führen eine ungestörte KEPLERsche Bewegung um den gemeinsamen Schwerpunkt aus, der gleichzeitig der Massenschwerpunkt des gesamten Systems ist

Man berechnet also quasi die ungestörte Zweikörperbewegung von m_1 und m_2 und anschließend die sog. *Störungen*, also die Abweichungen davon. So kann man z.B. die Bewegung eines Planetoiden auf einer durch den Einfluss des Jupiters gestörten Bahn um die Sonne untersuchen.

Eine detaillierte mathemathische Betrachtung dieser analytischen Methoden ist im Rahmen dieser Arbeit allerdings nicht möglich, sie gehen auch weit über den mathematischen Horizont des Autors hinaus. Ein prinzipielles Verständnis der Vorgehensweise soll jedoch den Unterschied zur numerischen Integration verdeutlichen.

1.3 Numerische Integration

1.3.1 Allgemeine Beschreibung und Erläuterung der verschiedenen Verfahren

Allgemein ist die numerische Integration ein Verfahren zur Berechnung von Näherungslösungen der Differentialgleichungen von dynamische Systemen. Von einem bekannten Ausgangszustand eines Systems Z_0 bei $t = t_0$ wird der Zustand des Systems zum Zeitpunkt $t = t_0 + \Delta t$ iterativ wie folgt berechnet:

$$Z(t_0 + \Delta t) = Z(t_0) + m \star \Delta t$$

Hierbei ist m eine Änderungsrate des Systems im Zeitintervall $[t_0; t_0 + \Delta t]$ darstellt. Zur Berechnung von m existieren mehrere Verfahren:

Das *Euler-Cauchy-Verfahren* verwendet die Momentanänderung zum Zeitpunkt t_0 als Änderungsrate für das gesamte Zeitintervall Δt. Zur Veranschaulichung: Das Verfahren versucht also, die exakte, gekrümmte Kurve

des Systems durch kleine Geradenstücke anzunähern. Im Intervall Δt wird die Kurve durch die Tangente am Kurvenpunkt bei t_0 ersetzt.

Das *verbesserte Euler-Cauchy-Verfahren* und das *Range-Kutta-Verfahren* versuchen, bei der Bestimmung der Änderungsrate m die Kurvenkrümmung stärker zu berücksichtigen. Dies geschieht dadurch, dass *Euler-Cauchy*-Schritte für verschiedene Intervalle wie z.B. $[t_0; t_0 + \Delta t]$, $[t_0; t_0 + \frac{\Delta t}{2}]$, $[t_0 + \Delta t; t_0 + 2\Delta t]$ berechnet werden und aus diesen dann ein gewichteter Mittelwert für m berechnet wird.

Wenn Δt ausreichend klein gewählt werden kann, kann es allerdings sein, dass das einfache *Euler-Cauchy-Verfahren* bereits befriedigende Ergebnisse liefert und die aufwendigeren Verfahren gar nicht notwendig sind. Auch im Simulationsprogramm war dies der Fall (eine genaue Beschreibung des konkret benutzten Algorithmus folgt weiter unten !).

1.3.2 Anwendung auf das Dreikörperproblem

Der Ansatz dieses Verfahrens zur Berechnung von $s(t)$, also des Ortes eines Planeten zu einem bestimmten Zeitpunkt, ist also der folgende: Ausgehend von einem Ausgangszustand zum Zeitpunkt $t = t_0$, bei dem der Ort und die Geschwindigkeit eines Planeten bekannt sind, berechnet man $s(t + \Delta t)$ iterativ:

$$s(t_0 + \Delta t) = s(t_0) + v_{\Delta t} \star \Delta t$$

Es wird also versucht, die Bahnen der einzelnen Massepunkte näherungsweise in kleinen Schritten zu berechnen. Dabei wird angenommen, dass $v_{\Lambda t}$ im Zeitintervall Δt konstant ist, was natürlich in der Realität nicht der Fall ist, man macht also einen Fehler. Dieser hängt natürlich von der Schrittweite Δt ab (je kleiner Δt, desto kleiner der Fehler !), allerdings auch vom Verfahren, dass zur Festlegung von $v_{\Delta t}$ benutzt wird. Die konkrete Umsetzung in einem Simulationsprogramm ist im praktischen Teil dieser Arbeit beschreiben.

2 Die Programmiersprache JAVA

2.1 Einleitung - Geschichte und Konzeption

JAVA ist noch eine recht junge Programmiersprache, die Version 1.0 wurde 1994 von Sun Microsystems der Öffentlichkeit vorgestellt. Seit dem hat sich auch noch einiges getan, so wurde der Quellcode von JAVA 1998 offengelegt und es wurden noch zahlreiche Änderungen und Erweiterungen getätigt. Mittlerweile ist man bei Version 1.3 angelangt und die Sprache erfreut sich hoher Beliebtheit unter Entwicklern. Dies hängt auch damit zusammen, dass JAVA einen etwas anderen Anstatz als traditionelle Programmiersprachen vertritt: Der Sourcecode wird vom JAVA-Compiler nicht gleich in Maschinensprache übersetzt, sondern in einen Bytecode, der dann von der *JVM*[1] zur Laufzeit interpretiert wird. Dieser Zwischenschritt hat natürlich Performanceeinbußen zur Folge, er hat aber auch zwei entscheidende Vorteile: Sicherheit (vor Programmabstürzen durch ein Fehlermanagement durch *Exceptions* und Sicherheit im Internet z.B. vor potentiellen Angriffen auf den ausführenden Rechner) und Plattformunabhängigkeit (es existieren Implementierungen der JVM für fast alle Plattformen !). So war es auch möglich, dass JAVA in verschiedene Browser integriert werden konnte und die sog. *JAVA-Applets* auf den vielen verschiedenen im Internet vertretenen Rechenerarchitekturen laufen. Dort ermöglichen sie aktive Web-Inhalte wie z.B. Spiele, komplexe Menüs etc. und dynamische Webseiten z.B. mit Datenbankabfragen. JAVA steht in gewisser Hinsicht also zwischen compilierenden und interpretierenden Programmiersystemen und versucht, die Vorteile der beiden Ansätze zu verbinden.

2.2 Objektorientierte Programmierung

Die Syntax von JAVA lehnt sich stark an C/C++ an, außerdem ist JAVA konsequent objektorientiert. Im Gegensatz zu den strukturellen Programmiersprachen (wie z.B. PASCAL, C, BASIC) existiert hier keine Trennung von Daten (in globalen/lokalen Variablen) und Funktionen, sondern diese

[1] Java Virtual Machine

sind zu Objekten zusammengefasst. Dies ist in JAVA (wie in C++) durch Klassen realisiert. Eine Klasse besitzt Variablen und Methoden, sie verwaltet ihre Daten selbst. Dies wird auch als *Datenkapselung* bezeichnet. Die Daten innerhalb einer Klasse können vollständig vor anderen Klassen verborgen werden, was die Fehleranfälligkeit senkt. Außerdem sind Klassen erweiterbar, und abstrakte Datenstrukturen (z.B. Bäume, Listen etc.) können aufgebaut und mittels *Vererbung* für bestimmte Anwendungen spezialisiert werden. Es existiert eine große Klassenbliothek, die z.B. Multithreading, Grafik - und Netzwerkprogrammierung, Datenbankanbindung etc. umfasst.

2.3 Entscheidungsgründe für JAVA

Die Summe der obigen aufgezählten Eigenschaften ließen JAVA für das Simulationsprogramm geeignet erscheinen. Die Nähe der Sprache zu C++ reduzierte den Einarbeitungsaufwand, obgleich die Bereiche Multithreading, Grafikbibliotheken und GUI-Programmierung dem Autor auch relativ unbekannt waren. Doch insbesondere bei letzterem leisteten Tools wie der *Borland JBuilder* wertvolle Unterstützung, denn getreu dem JAVA-Motto "write once, run anywhere" konnte unter LINUX entwickelt werden (wofür der *JBuilder* kostenlos erhältlich ist), auch wenn das Programm später unter Windows laufen sollte. Des weiteren ist z.B. die GUI-Programmierung nicht auf eine Plattform beschränkt, wie dies z.B. bei einer Einarbeitung in die MFC (Microsoft Foundation Classes für Windows-GUIs) oder in Qt (für KDE-GUIs) der Fall gewesen wäre. Das erworbene Wissen hat so (hoffentlich) eine höhere "Halbwertszeit" und ist weniger an einen Hersteller oder ein Betriebssystem gebunden. Dies rechtfertigt auch den Nachteil der geringeren Ausführungsgeschwindigkeit von JAVA-Programmen im Vergleich zu "nativen" C++ Anwendungen, bei den Rechenleistungen heutiger PC's fällt dies sowieso eher weniger ins Gewicht.

Teil II

Praktischer Teil

3 Erläuterungen zum Simulationsprogramm in JAVA

3.1 Einführung - generelle Vorgehensweise des Programms

Zunächst ist festzustellen, dass sich die auf einen Planeten zu einem Zeitpunkt t_1 wirkende Kraft nach Gleichung 2 berechnen lässt. Kern des Programms ist die Idee, für jeden Planeten einen sog. Thread, also einen eigenen Prozess, zu starten, der diese Kraft jeweils im Abstand von Δt berechnet und die Bewegung des Planeten entsprechend ausführt. Beim Dreikörperproblem laufen also drei Threads gleichzeitig ab, und immer nach der Zeitspanne Δt wird in einem Thread die auf einen Planeten wirkende Kraft aufgrund der momentanen Position der anderen Planeten neu berechnet und seine Position aktualisiert.

Beim Programmstart erscheint zuächst das *EingabeFrame*. Wenn die Werteeingabe im *EingabeFrame* abgeschlossen ist, wird die Simulation durch Drücken des Buttons gestartet. Dabei wird ein neues Fenster geöffnet, für dessen Inhalt sich die Klasse *SimulationArea* verantwortlich zeigt. Sie erzeugt zunächst die gewünschte Anzahl von Planeten (*class planet*), um anschließend die Threads (*class PlanetThread*) zu starten, die für deren Bewegung verantwortlich sind. Diese simulieren die Planetenbewegung solange, bis das Simulationsfenster geschlossen wird.

3.2 Konkrete Umsetzung - Beschreibung der wichtigen Klassen und Funktionen

3.2.1 *class EingabeFrame*

Im *EingabeFrame* kann der User alle für das Simulationsprogramm wichtigen Daten für die Planeten eingeben: jeweils die Masse, die Position und die Ausgangsgeschwindigkeit. Außerdem kann durch *pix_scale* der örtliche Maßstab (ein Pixel entspricht *pix_scale* Kilometern) und durch *time_scale* der zeitliche Maßstab (eine Sekunde entspricht *time_scale* Jahren in der Simulationsrealität) angegeben werden. Die Variable *delta_t* legt die "Auflösung" der Simulation fest (in ms), je kleiner dieser Wert, umso genauer das Ergebnis. Außerdem beinhaltet die Klasse *EingabeFrame* eine Reihe von "Textfeldern" (*JTextFields*) als Variablen, die die Benutzereingaben aufnehmen. Außerdem gibt es eine Variable *simArea*, eine Instanz der Klasse *SimulationArea*, welche sich um die Simulationsdarstellung kümmert. Wenn der Buttton zum Starten der Simulation gedrücken wurde, wird jedoch zunächst die Funktion *EingabeFrame.jButton1_mouseClicked()* aufgerufen, bevor die Kontrolle dann an *simArea* übergeben wird.

3.2.1.1 *EingabeFrame*.jButton1_mouseClicked(MouseEvent e)

```
void jButton1_mouseClicked(MouseEvent e) {

    simArea.mass[1] = (Float.parseFloat(mass_base1.getText())) * ten-
exp(Float.parseFloat(mass_exp1.getText()));
    simArea.mass[2] = (Float.parseFloat(mass_base2.getText())) * ten-
exp(Float.parseFloat(mass_exp2.getText()));
    simArea.mass[3] = (Float.parseFloat(mass_base3.getText())) * ten-
exp(Float.parseFloat(mass_exp3.getText()));

    simArea.pos_x[1] = (Float.parseFloat(pos_x1.getText()));
    simArea.pos_x[2] = (Float.parseFloat(pos_x2.getText()));
    simArea.pos_x[3] = (Float.parseFloat(pos_x3.getText()));
```

```
        simArea.pos_y[1] = (Float.parseFloat(pos_y1.getText()));
        simArea.pos_y[2] = (Float.parseFloat(pos_y2.getText()));
        simArea.pos_y[3] = (Float.parseFloat(pos_y3.getText()));

        simArea.v_x[1] = (Float.parseFloat(v_x1.getText()));
        simArea.v_x[2] = (Float.parseFloat(v_x2.getText()));
        simArea.v_x[3] = (Float.parseFloat(v_x3.getText()));

        simArea.v_y[1] = (Float.parseFloat(v_y1.getText()));
        simArea.v_y[2] = (Float.parseFloat(v_y2.getText()));
        simArea.v_y[3] = (Float.parseFloat(v_y3.getText()));

         simArea.pix_scale = (Float.parseFloat(pix_base.getText())) * ten-
exp(Float.parseFloat(pix_exp.getText()));
        simArea.delta_t = (Float.parseFloat(delta_t.getText()));

        simArea.start_simulation();
      }
    }
```

Diese Funktion wandelt die Zeichenketten aus den Eingabefeldern (*String JTextField.getText()*) mittels *Float.parseFloat(String s)* von *Strings* in *Floats* um und speichert sie in Variablen von *EingabeFrame.simArea* (eine Instanz der Klasse *SimulationArea* !). Dieses sind "öffentliche" Variablen, d.h. aufgrund des Schlüsselwortes *public* bei der Definition dieser Variablen in der Klassendefinition von *SimulationArea* ist es anderen Klassen möglich, auf sie zuzugreifen. Beispielsweise wird die Masse des Planeten 1 aus dem Inhalt von *mass_base1* und *mass_exp1* berechnet und nach *simArea.mass[1]* geschrieben, die X-Position des Planeten steht in *pos_x1* und wird nach *simArea.pos_x[1]* geschrieben. Entsprechendes gilt für die anderen Eingabefelder. Anschließend startet die Simulation mit Aufruf von *simArea.start_simulation()*.

3.2.2 *class SimulationArea*

Die Klasse *SimulationArea* ist von *Canvas* aus dem AWT[2] abgeleitet ist, was übersetzt so viel wie "Leinwand" bedeutet. In der Tat stellt die *SimulationArea* eine Art Leinwand für die Simulation dar, auf der die Planeten ihre Bahnen durchlaufen. Sie besitzt folgende Variablen:

```
public class SimulationArea extends Canvas
{
        volatile Graphics2D g2d;

        public volatile planet planet[];
        public volatile PlanetThread pThread[];
        public int nr_planets;

        public double mass[];
        public double pos_x[];
        public double pos_y[];
        public double v_x[];
        public double v_y[];
        public double pix_scale;
        public double delta_t;

        Frame f;
```

Die Klasse enthälet eine Reihe von *double* Variablen, denen das Schlüsselwort *public* vorangeht. In diesen öffentlichen Variablen werden die aus den Benutzereingaben im *EingabeFrame* gewonnenen Angaben zu den Planeten und z.B. *pix_scale* etc. zwischengespeichert, die zum Erzeugen der Planeten notwendig sind (siehe Konstruktor planet !). Außerdem gibt es zwei Arrays von Objekten, die die an der Simulation beteiligten Planeten (*SimulationArea.planet[]*) und für deren Bewegung verantwortlichen Threads (*pThread[]*

[2]Abstract Window Toolkit, eine GUI-Bibliothek für JAVA

) beinhalteten. Schließlich gibt die (ganzzahlige) Variable *nr_planets* die Anzahl der an der Simulation beteiligten Planeten an, währen *Graphics2D g2d* den sogenannten "Grafikkontext" darstellt. Er übernimmt die Darstellung von Objekten, die Teil des *JAVA2D*[3]-Paketes sind (z.B. die der deswegen von *Ellipse2D* abgeleiteten Planeten !).

3.2.2.1 *SimualtionArea.start_simulation()*

In der Methode *SimulationArea.start_simulation()* wird ein neues Fenster (das *Frame f*) geöffnet, in dem die *SimulationArea* angezeigt wird:

```
public void start_simulation()
{
  f = new Frame( "DreiKörper" );
  f.setLayout( new FlowLayout() );
  Panel p = new Panel();
  f.add( p );
  this.setBackground( Color.lightGray );
  this.setSize( 600, 600 );
  this.setLocation(350,10);
  f.add( this );

// Registrierung der Listener
  f.addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    {
        for (int i = 1; i < nr_planets; i++)
        {
            pThread[i].end = true;
            pThread[i] = null;

        };
        f.dispose();
```

[3]JAVA2D ist ein Toolkit zur platformunabhängigen Erstellung von 2D-Grafiken

```
      }
    } );

// Fenstergroesse einstellen und Fenster anzeigen

    f.pack();
    f.show();
  }
}
```

Zunächst wird das Simulationsfenster erzeugt (*f = new Frame("Dreikörper")*). Der Bezeichner *this* steht für die Klasse *SimulationArea* selbst; mit ***this.setSize()*** bzw. *this.setLocation()* wird ihre Größe und ihr Ort festgelegt. Dann wird sie dem Frame hinzugefügt wird (*f.add(this)*), anschließend wird nur noch sichergestellt, dass beim Schließen des Simulationsfensters *f* auch alle Threads beendet werden, und schließlich kommt es zur Anzeige (*f.show()*).

Der Inhalt des Simulationsfensters besteht allein aus der *SimulationArea*, weshalb nun die Funktion SimulationArea.paint(Graphics g) betrachtet werden soll (bei jeder Komponente des AWT (Abstract Window Toolkit) wird ***paint()*** aufgerufen, sobald sie gezeichnet wird, und SimulationArea ist Erbe von *Canvas* aus dem AWT !).

3.2.2.2 *SimulationArea.paint(Graphics g)*

Der Inhalt der *SimulationArea* wird also in der Methode *SimualtionArea.paint(Graphics g)* festlegt. Zu Beginn wird die in *SimulationArea.nr_planets* festgelegte Anzahl von Planeten erzeugt:

```
for (int i = 1; i <= nr_planets; i++)
{
  planet[i] = new planet(i, pos_x[i], pos_y[i],v_x[i],v_y[i], mass[i], this);
};
```

Dafür werden dem Planeten-Konstruktor die benötigten Werte für den jeweiligen Planeten übergeben (siehe *class planet* !). Allerdings erhält der Konstrukor auch eine Referenz auf die aktuelle *SimulationArea* (*this*), da er sich später ja in diese zeichnen soll und dafür auf den Grafikkontext *SimulationArea.g2d* zugreifen muss. Anschließend werden die *PlanetThreads* erzeugt und gestartet:

```
for (int i = 1; i <= nr_planets; i++)
{
  pThread[i] = new PlanetThread( String.valueOf(i), planet[i], this);
};

for (int i = 1; i <= nr_planets; i++)
{
  pThread[i].start();
};
```

Die Simulation an sich wird nun von den *PlanetThreads* verwaltet, die Funktion *paint()* stellt nur noch sicher, dass sie erst beendet wird, wenn das Simulationsfenster geschlossen wurde und die Funktion folglicherweise ihre Aufgabe erfüllt hat. Dies wird durch einen Thread *waiter* realisiert:

```
    Thread waiter = new Thread("waiter");
    waiter.start();
    while (pThread[1].end == false) {

    try {
                    waiter.wait();
                } catch (InterruptedException e) {}

}
```

3.2.3 *class PlanetThread*

Die Klasse *PlanetThread* ist von der Klasse *Thread* abgeleitet und für die Bewegung eines Planeten verantwortlich:

```
public class PlanetThread extends Thread
{
        private SimulationArea myArea;
        private volatile planet p = null;

        public boolean end;
```

Neben der Referenz auf "ihren" Planeten (*PlanetThread.p*) und auf die aktuelle *SimulationArea* besitzt sie noch das Flag *end*, dass vom Konstruktor auf *false* gesetzt wird. Es wird erst *true*, wenn der Benutzer die Simulation durch das Schließen des Fensters beendet (siehe Registrieung des entsprechenden Window-Listeners in *SimulationArea.start_simulation()*).

3.2.3.1 *PlanetThread.run()*

Der Programmcode eines *Threads* befindet sich in der Methode *Thread.run()*, die deshalb die zentrale Stelle des Simulationsprozesses darstellt:

```
        public void run() {
           do
           {
                   p.compute();
                   draw_planet();

                   try {
                     sleep(p.delta_t);
                   } catch ( InterruptedException e) {};

                   delete_planet();
```

```
        } while (end != true);
    }
```

Wie im Teil 3.1 beschrieben, basiert die Simulation darauf, die Kräfte auf einen Planeten zu einem bestimmten Zeitpunkt zu berechnen und daraus die Bewegung im nächsten Zeitabschnitt Δt zu folgern. Im Programm sieht das dann so aus:

Zuerst wird die neue Position des Planeten, also seine Position bei $t = t_0 + \Delta t$, berechnet (siehe 3.2.4.1), dann wird er neu gezeichnet (-> *draw_planet()*), und anschließend "schläft" der Thread für die Zeit Δt (-> *sleep(p.delta_t)*). Nachdem die Zeitspanne Δt vergangen ist wird der Planet gelöscht; die Schleife beginnt von vorne, falls das Flag *PlanetTread.end*$\neq$*true* ist.

3.2.4 *class planet*

Die Klasse *planet* beinhaltet alle physikalischen Eigenschaften und Methoden, die für einen Planeten im Simulationsprogramm notwendig sind. Dies sind im Einzelnen:

```
public class planet extends Ellipse2D.Float
{

    public int nr;
    public long pix_scale;
    public long delta_t;
    public long sim_delta_t;
    public double mass;
    public SimulationArea myArea;

    private Vector f = null;
```

Sie ist von *Ellipse2D.Float* abgeleitet und erbt dadurch die Möglichkeit, mittels der Funktion *Graphics2D.draw(Shape s)* gezeichnet zu werden (dies geschieht in *PlanetThread.drawplanet()*). Die Variable *pix_scale* ist (wie

oben schon genannt) ein örtlicher Maßstab (ein Pixel entspricht *pix_ scale* Kilometern), wohingegen *time_ scale* einen zeitlichen Maßstab darstellt (eine Sekunde entspricht *time_ scale* Sekunden in der Simulationsrealität). Für die Simulation wird daher im Konstruktor von *planet* die Variable *sim_ delta_ t* berechnet, die ensprechend dem gewählten *time_ scale* gesetzt wird:

```
sim_delta_t = (long) ( delta_t * 31536 * myArea.time_scale)
```

Wenn *time_ scale* gleich 1 ist, dann soll definitionsgemäß eine Sekunde in der Simulation einem Jahr in der Realität entsprechen. Wenn nun *delta_ t* = 1000 (entspricht 1 Sekunde, weil *delta_ t* in Millisekunden angegeben werden muss [dies verlangt die Methode *Thread.sleep()* !]), dann ist *sim_ delta_ t* = 31536000, was der Zahl der Sekunden in einem Jahr entspricht. Die Variable *sim_ delta_ t* enthält also die Zeit in Sekunden, die in der Simulationsrealität währen der Wartezeit *PlanetThread.sleep(delta_ t)* vergehen.

Der Vektor f speichert die momentan auf den Planeten wirkende Kraft, der Vektor v die momentane Geschwindigkeit. Sie werden in der Methode *compute()* berechnet und verarbeitet, weshalb diese Funktion den physikalischen Kern der Simulation darstellt.

3.2.4.1 *planet.compute()*

planet.compute() wird von *PlanetThread.run()* aufgerufen, um die Position eines Planeten zum aktuellen Zeitpunkt zu berechnen. In der ersten Schleife werden mittels der Funktion *compute_ planet_ force(planet p)* die Kräfte der einzelnen Planeten berechnet und zum eigenen Kraftvektor f hinzuaddiert:

```
public void compute()
{
f.setx(0);
f.sety(0);

for (int i = 1; i <= myArea.nr_planets; i++)
```

```
{
    if (myArea.planet[i].nr != nr)
    f.v_add( compute_planet_force(myArea.planet[i]) );
};
```

Anschließend sind die gerade auf den Planeten wirkende Kräfte in f gespeichert. Daraus wird nun die Geschwindigkeit für dass nächste Zeitintervall Δt berechnet, und zwar nach der Formel $v = v_0 + a \cdot t = v_0 + \frac{f}{m} \cdot t$:

```
double new_ax = (f.getx() / mass);
double new_ay = (f.gety() / mass);

v.setx( v.getx() + ( new_ax * sim_delta_t ) );
v.sety( v.gety() + ( new_ay * sim_delta_t ) );
```

Diese Geschwindigkeit wird nun als konstant in Δt angesehen und die zurückgelegte Strecke nach der Formel $s = v \cdot t$ berechnet. Dies geschieht sowohl für die X-Koordinate als auch für die Y-Koordinate. Die neuen Koordinaten des Planeten berechnen sich nun als Summe der Ausgangskoordinaten und der in Δt zurückgelegten Strecke. Anschließend werden sie für den nächsten Zeichenvorgang durch *PlanetThread.draw_planet()* neu gesetzt:

```
double newx = x + ( v.getx() * sim_delta_t);
double newy = y + ( v.gety() * sim_delta_t);
this.setFrame(newx , newy, width, height);
}
```

3.2.4.2 *compute_planet_force(planet p)*

Die Funktion *planet.compute_planet_force(planet p)* berechnet die Gravitationskraft von p auf den Planeten, der die Funktion aufruft. Dabei wird wie folgt vorgegangen: Zunächst wird die Entfernung r der beiden Planeten berechnet. Dies geschieht mit Hilfe des "Satz des Pythagoras":

$$r_{ij} = \sqrt{\Delta x_{ij}^2 + \Delta y_{ij}^2} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Der Abstand kann also aus den Differenzen der X- und Y-Koordinaten der Planeten berechnet werden:

```
double distanceX = (double) p.getCenterX() - (double) this.getCenterX();
double distanceY = (double) p.getCenterY() - (double) this.getCenterY();

double distance = Math.sqrt( (distanceX * distanceX) + (distanceY * distanceY));
```

Der Betrag lässt sich dann nach Newtons Gravitationsgesetz (siehe Gleichung 1, Seite 4) wie folgt berechnen, wobei zu beachten ist, dass der Abstand in Pixeln (*distance*) erst noch mit *pix_scale* multipliziert werden muss, um den benötigten Abstand in Kilometern zu erhalten:

```
double distance_km = distance * pix_scale;

double betrag = (6.67259e-11f) * (this.mass * p.mass) / (distance_km * distance_km);
```

Nun wird ein Einheitsvektor in Richtung der Verbindungslinie der zwei Planeten erzeugt:

```
Vector ret = new Vector(distanceX,distanceY);
ret.s_mult( ( 1 / distance) );
```

Anschließend wird dieser noch mit dem zuvor berechneten *betrag* multipliziert und dann zurückgegeben:

```
ret.s_mult(betrag / pix_scale);
return ret;
```

Teil III

Anhang

A Literaturverzeichnis

- Grehn, Joachim: Metzler Physik, 2. Auflage, Schroedel Verlag, Hannover 1992
- Koller, Dieter: Simulation dynamische Vorgänge. Ein Arbeitsbuch. 1.Auflage, Ernst Klett Schulbuchverlag, Stuttgart 1995
- Kopp, Markus; Wilhelms, Gerhard: Java Professionell. 1.Auflage, MITP-Verlag, 1999
- Montenbruck, Oliver: Taschenbuch 10. Grundlagen der Ephemeridenrechnung. 4. Auflage, Verlag Sterne und Weltraum, München 1989
- Stumpf, Karl: Himmelsmechanik, Band II. 1.Auflage, Deutscher Verlag der Wissenschaften, Berlin 1965

B Beispielsimulation zur Überprüfung des Programms

Es soll nun anhand eines Beispiels getestet werden, ob das Programm sinnvolle Ergebnisse liefert. Die Theorie besagt, dass ein Körper mit der Geschwindigkeit v auf einer Kreisbahn gehalten werden kann, wenn eine zum Drehzentrum gerichtete Zentripetalkraft F_Z wirkt, die sich nach der Formel $F_Z = -m\frac{v^2}{r}$berechnen lässt. Diese entspricht der Gravitationskraft, die die Körper aufeinander ausüben, die Beträge der Kräfte müssen also gleich sein ($|F_Z| = |F_G|$). Für die Bahngeschwindigkeit v folgt daraus bei gegebener Masse des Zentralkörpers m_z, Masse des umlaufenden Körpers $m_k$und einem Radius von r :

$$v = \sqrt{\frac{|F_Z| \cdot r}{m_k}} = \sqrt{\frac{|F_G| \cdot r}{m_k}} = \sqrt{\frac{\gamma \frac{m_z \cdot m_k}{r^2} \cdot r}{m_k}} = \sqrt{\frac{\gamma m_z}{r}}$$

Die Bahngeschwindigkeit, die notwendig ist, damit sich die Erde auf einer stabilen Bahn um die Sonne dreht, läßt sich also mit der Formel $v = \sqrt{\frac{\gamma m_z}{r}}$berechnen, es ergibt sich ein Wert von $v \approx 29790\frac{km}{s}$. Um zu sehen, ob dieser theoretisch erhaltene Wert im Simulationsprogramm zum gewünschten Ergebnis führt, wurden also die Angaben von Sonne und Erde ins *Eingabeframe* eingetragen, sie sind auch als default-Werte eingestellt und erscheinen bei jedem Programmstart. Wird die Simulation also ohne Veränderung dieser Werte gestartet, so kann man tatsächlich die erwartete stabile Bahn der Erde um die Sonne beobachten, was auf die physikalische Richtigkeit des Programms hindeutet. Eine ausgiebige Testsimulation wurde jedoch nicht durchgeführt, eine abschließende Garantie für die Gültigkeit der Simulationsergebnisse kann also nicht gegeben werden.

C Bemerkungen zum Programm

Das vorliegende Programm ist in der Lage, die geforderte Aufgabe, nämlich die Simulation des Dreikörperproblems, zu bewältigen. Allerdings ist es bei weitem kein ausgereiftes Stück Software, was an der kurzen Debug- und Testphase liegt. Folgende Punkte beeinträchtigen zwar die Funktionalität des Programms nicht, sie sollten allerdings trozdem erwähnt werden:

- Die Methode *SimulationArea.paint()* produziert folgende Exception: *java.lang.IllegalMonitorStateException: current thread not owner* . Dies hängt damit zusammen, dass die Funktion *paint()* nicht beendet werden darf, solange die Simulation noch läuft. Um dies zu verhindern, wird ein Thread *waiter* gestartet und solange die Methode *waiter.wait()* aufgerufen, bis das Flag *pThread[1].end* true ist. JAVA beschwert sich über diesen Funktionsaufruf, da er in der Funktion *SimulationArea.paint()* stattfindet und sie eigentlich nur vom Thread *waiter* selbst aufgerufen werden sollte. Ich habe allerdings keine andere Lösung dieses Problems gefunden, in C++ könnte man dank der Mehrfachvererbung die Klasse *SimulationArea* auch von *Thread* ableiten und dann die Funktion *SimulationArea.wait()* aufrufen, in JAVA ist dies nicht möglich. Allerdings hat diese Exception außer einer unschönen Meldung auf der Konsole keine negativen Effekte.

- Ab und zu treten Synchronisationsprobleme der Threads auf, was zu Grafikfehlern führen kann. Diese sind jedoch zum Glück relativ selten. In einer Multitasking-Umgebung ist es generell ein Problem, wenn mehrere parallel laufende Prozesse auf die gleichen Ressourcen (in diesem Fall der Grafikkontext in der *SimulationArea* !) zugreifen wollen, in JAVA wurde deshalb das Schlüsselwort *synchronized* eingefürt, welches problematischen Funktionen als Attribut zugewiesen werden kann. Allerdings konnte auch hierdurch das Auftreten der geschilderten Probleme nicht vollständig gelöst werden.

- Das Programm erwartet korrekte Benutzereingaben, es sind keine Routinen zur Behandlung von Fehleingaben vorhanden. Dies ist allerdings

dank des Sicherheitskonzeptes von JAVA kein größeres Problem, denn eine Fehleingabe löst schlicht eine *Exception* aus, deren Meldung auf dem Terminal ausgegeben wird. Die Eingabe kann anschließend wiederholt werden, ein Programmabsturz wie bei manchen Dos-Programmen ist nicht zu befürchten.

Zum Berechnungsalgorithmus ist folgendes zu sagen:

Das benutzte Verfahren berechnet aufgrund der gerade wirkenden Kräfte die Geschwindigkeit v für dass Intervall Δt und nimmt diese als konstant an. Ein anderer Ansatz bestünde darin anzunehmen, dass die Kraft f im Zeitintervall Δt konstant ist. Dann ließe sich nach den Gesetzen der gleichförmig beschleunigten Bewegung die zurückgelegte Strecke in diesem Zeitintervall wie folgt berechnen:

$$s = \frac{1}{2}at^2 = \frac{1}{2}\frac{f}{m}t^2$$

Allerdings führte dieser Ansatz in der Simulation zu falschen Ergebnissen, so dass er verworfen wurde.

D Sourcecode [Anlage]