Implementierung und Vergleich verschiedener Methoden zur Berechnung der gravitativen Dynamik von n Körpern

David Berger

07. November 2016

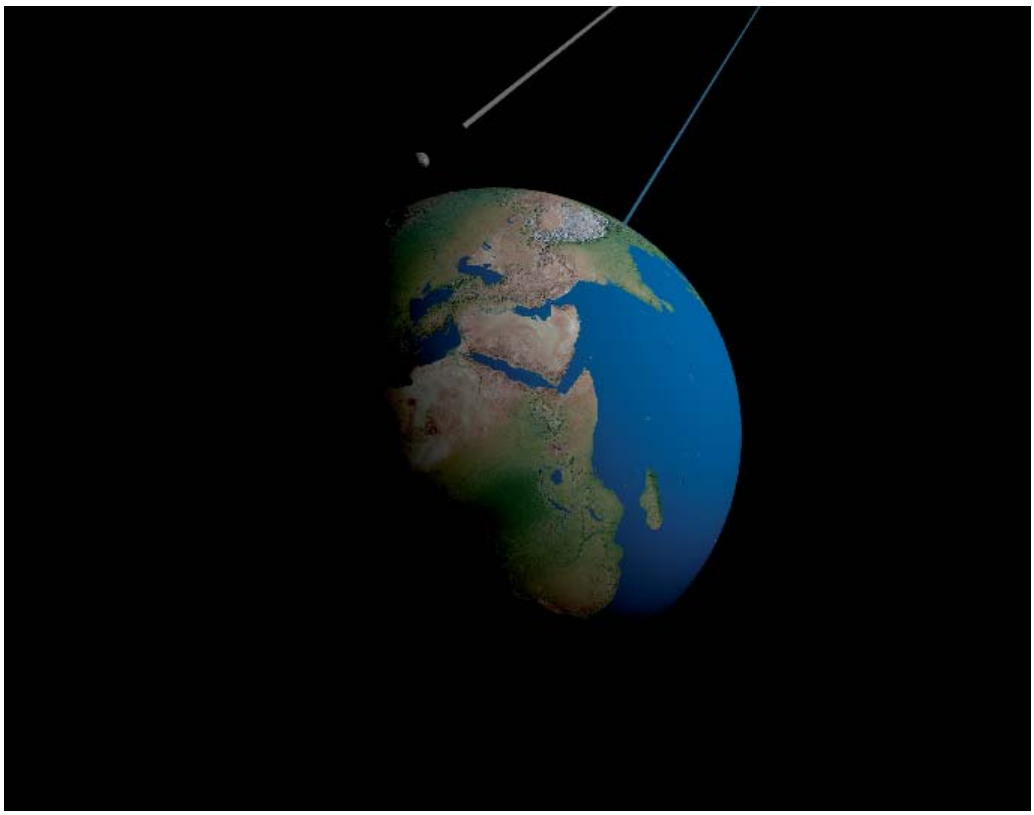

Abbildung 1: Simulation des Sonnensystems

Inhaltsverzeichnis

1 Bestimmung von Planetenbahnen

Der Mechanismus von Antikythera ist eine mindestens 2000 Jahre alte Vorrichtung, die unter Anderem zur Vorhersage von Sonnen- und Mondfinsternissen entworfen wurde. Die Apparatur funktioniert aufgrund bloßer Beobachtungen und nutzt dabei nur das periodische Auftreten der Ereignisse. Dennoch spiegelt es das immer noch aktuellen Bestreben der Menschen wider, astronomische Geschehnisse vorauszusagen. Mehr als 1600 Jahre danach machte Johannes Kepler die Entdeckung, dass die Planeten des Sonnensystems auf elliptischen Bahnen die Sonne umkreisen. Nur wenig später, 1687, gelang es Isaac Newton, auch den Grund hinter den Bewegungen zu verstehen: Er verband erstmals die Schwerkraft, die den Menschen am Boden hält, mit den Bahnen der Objekte am Himmel und den zwischen ihnen wirkenden Kräften. Damit war der Sprung von der Kinematik zur Dynamik getan, der es uns heute erlaubt, durch das Verständnis der wirkenden Kräfte und mit Hilfe von Computern auch völlig unbekannte Systeme zu simulieren und präzise vorherzusagen. Doch auch moderne Computer gelangen bei der Berechnung der Dynamik von vielen Körpern an ihre Grenzen.

2 Mathematische Erklärung des n-Körper Problems

Das n-Körperproblem ist eine Problemstellung der Astronomie. Es besteht aus dem Versuch, die Positionen von n Punktmassen, die sich gegenseitig ohne relativistische Effekte durch Gravitationskräfte beeinflussen, zu einem Zeitpunkt t zu bestimmen. Die gegebenen Anfangswerte sind die Masse m, die aktuelle Position $\vec{x}$ und die Geschwindigkeit $\vec{v}$ der Objekte. Daraus lässt sich über das Newtonsche Gravitationsgesetz der Betrag der wirkenden Kraft zwischen zwei Körpern der Masse m_1 und m_2 wie in (1) berechnen.

$$|\vec{F}| = G \cdot \frac{m_1 \cdot m_2}{r^2} \tag{1}$$

Dabei ist die Entfernung der Objekte voneinander bzw. der Radius r gegeben ist als

$$r = |\vec{x}_1 - \vec{x}_2| \tag{2}$$

Die Richtung der Kraft zwischen den Objekten ergibt sich aus

$$\vec{F}_0 = \frac{\vec{x}_1 - \vec{x}_2}{|\vec{x}_1 - \vec{x}_2|} \tag{3}$$

Weil gilt

$$\vec{F} = m \cdot \vec{a} \tag{4}$$

und daraus folgend

$$\vec{a} = \frac{\vec{F}}{m} \tag{5}$$

kann man (2) in (1) und (1) in (5) einsetzen und erhält als Formel für die ungerichtete Beschleunigung

$$|\vec{a}_1| = G \cdot \frac{\frac{m_1 \cdot m_2}{r^2}}{m_1} = G \cdot \frac{m_1 \cdot m_2}{r^2 \cdot m_1} = G \cdot \frac{m_2}{|\vec{x}_1 - \vec{x}_2|^2} \tag{6}$$

Um die Richtung der Beschleunigung zu erhalten multipliziert man mit dem Einheitsvektor der Richtung aus (3) und erhält

$$\vec{a}_1 = G \cdot \frac{m_2}{|\vec{x}_1 - \vec{x}_2|^2} \cdot \frac{\vec{x}_1 - \vec{x}_2}{|\vec{x}_1 - \vec{x}_2|} = G \cdot \frac{m_2 \cdot (\vec{x}_1 - \vec{x}_2)}{|\vec{x}_1 - \vec{x}_2|^3} \tag{7}$$

Allgemein formuliert für eine beliebige Anzahl an Körpern ist die Beschleunigung die Summe aller von anderen Körpern verursachten Einzelbeschleunigungen. Hierbei sind i und j Zählvariablen, die bei 0 beginnen und n Körper durchlaufen.

$$\vec{a}_i = G \cdot \sum_{j \neq i}^{i,j<n} \frac{m_j \cdot (\vec{x}_i - \vec{x}_j)}{|\vec{x}_i - \vec{x}_j|^3} \tag{8}$$

Die gesuchte Position und die Beschleunigung hängen über die zweite Ableitung zusammen.

$$\ddot{\vec{x}}_i = G \cdot \sum_{j \neq i} \frac{m_j \cdot (\vec{x}_i - \vec{x}_j)}{|\vec{x}_i - \vec{x}_j|^3} \tag{9}$$

Da diese Differentialgleichung aber wie im Satz von Bruns und Poincaré bewiesen[9] analytisch nicht lösbar ist, muss die Lösung der Gleichung mit Hilfe numerischer Integratoren angenähert und die Methode der kleinen Schritte angewendet werden.

3 Funktionsweise der Integratoren

Die in den folgenden Abschnitten verwendeten Variablen h, y und t sind die üblichen Konventionen für numerische Lösungsverfahren: h steht für einen (Zeit)Schritt, der optimal gegen 0 geht, was aber in der Realität nicht umsetzbar ist. Daher wird h in den meisten finalen Formeln durch Δt ersetzt. y stellt den unbekannten Funktionswert dar, der bei dieser Problemstellung mit dem gesuchten $\vec{x}$ gleichzusetzen ist. Die Zeit wird gewöhnlich durch t angegeben. Die ebenfalls Häufig verwendete Funktion f ist die erste Ableitung der Funktion von y, wobei f immer mindestens vom Funktionswert y abhängt, also $f(y)$. Die Abhängigkeit des y von $f(y)$ ist eine Eigenschaft der zu lösenden Differentialgleichung. Gesucht sind jeweils die schrittweisen Formulierungen für $\vec{x}$ und $\vec{v}$ in Abhängigkeit von $\vec{a}$, um die Verfahren anschließend einfach in die Implementierung übernehmen zu können.

3.1 Euler-Verfahren

Das Euler-Verfahren verfolgt den Ansatz, dass $\vec{v}$ sowie $\vec{a}$ über einen Zeitschritt konstant bleiben. Durch einsetzen der physikalischen Definitionen von Position, Geschwindigkeit und Beschleunigung ($\vec{a} = \dot{\vec{v}} = \ddot{\vec{x}}$) in die Euler-Methode ($y_{i+1} = y_n + hf(t_n, y_n)$) wird die Position nach einem Zeitschritt bei gegebener Beschleunigung berechnet durch:[4,6]

$$\vec{x}_{i+1} = \vec{x}_i + \vec{v}_i \cdot \Delta t \tag{10}$$

Danach wird für den nächsten Schritt die Geschwindigkeit berechnet, wobei diese Reihenfolge eingehalten werden muss.

$$\vec{v}_{i+1} = \vec{v}_i + \vec{a}_i \cdot \Delta t \tag{11}$$

Das Problem bei einer angenommenen konstanten Geschwindigkeit während eines Schrittes ist, dass entweder mit der Geschwindigkeit *nach* (explizit, Abb. 2 links) oder *vor* (implizit, Abb. 2 links) dem Zeitschritt (anschaulich) an die Fläche unter der Funktion der Geschwindigkeit (also die Position) angenähert wird. Gesucht ist eigentlich das genaue Integral der Geschwindigkeit wie in Abbildung 2 rechts.

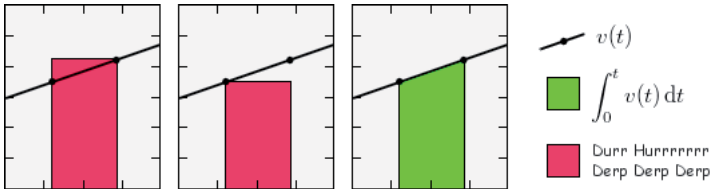

Abbildung 2: Darstellung des Integrals der Geschwindigkeit
Quelle: http://lolengine.net/raw-attachment/blog/2011/12/14/understanding-motion-in-games/accurate-integral.png

Mit der Beschriftung der roten Fläche, die die Euler-Methode veranschaulicht, will der Ersteller der Grafik auf die Ausbaufähigkeit eben jenes Verfahrens hinweisen.

3.2 Bewegungsgleichung

Eine weitere Annäherung kann durch das Anwenden der klassischen Bewegunsgleichungen ((12),(14)) erreicht werden. Diese Methode ist bereits in Abbildung 2 in grün dargestellt. Wenn man annimmt, dass $\vec{a}$ konstant ist, lässt sich die Position nach der Zeitspanne Δt durch (12) berechnen, wobei x_0 die Position und v_0 die Geschwindigkeit vor dem Schritt darstellt.

$$\vec{x} = \vec{x}_0 + \vec{v}_0 \Delta t + \frac{1}{2}\vec{a}\Delta t^2 \tag{12}$$

Dies lässt sich auch iterativ formulieren:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{v}_i \Delta t + \frac{1}{2}\vec{a}_i \Delta t^2 \tag{13}$$

Danach muss noch die Geschwindigkeit nach dem Zeitschritt berechnet werden, damit im nächsten Schritt v_0 wieder die derzeitige Geschwindigkeit angibt. Dazu wird erneut eine Bewegungsgleichung angewendet.

$$\vec{v}_{0,neu} = \vec{v}_0 + \vec{a} \cdot \Delta t \tag{14}$$

3.3 Geschwindigkeits-Verlet

Die in 3.2 beschriebene Methode kann nun im Sinne des Geschwindigkeits-Verlet Algorithmus noch verfeinert werden. Dieser nutzt zwar ebenfalls Formel (13) zur Positionsbestimmung, verwendet aber zur Bestimmung der neuen Geschwindigkeit eine Gleichung, die eine (lineare) Änderung der Beschleunigung berücksichtigt. Hierfür wird der Mittelwert zwischen vorheriger Beschleunigung und der Beschleunigung nach dem Schritt verwendet.[2] Zur Umsetzung wird zusätzlich eine Funktion $\vec{a}(\vec{x})$ benötigt die aber in (8) schon gegeben ist. Dabei ist $\vec{x}$ die Position vor und $\vec{x}_{i+1}$ die Position nach dem Schritt.

$$\vec{v}_{0,neu} = \vec{v}_0 + \frac{1}{2}(\vec{a}(\vec{x}_i) + \vec{a}(\vec{x}_{i+1})) \cdot \Delta t \tag{15}$$

3.4 Verlet-Integration

Anders als in 3.3 verwendet die Verlet-Integration nicht die Geschwindigkeit, sondern zusätzlich zur aktuellen Position auch die Position vor dem letzten Berechnungsschritt. Aus diesen Daten wird dann mit Hilfe der aktuellen Beschleunigung $\vec{a}$ die nächste Position nach einem Zeitschritt Δt berechnet. Aufgrund der Notwendigkeit der vorherigen Position stellt sich ein Startwertproblem: Beim ersten Schritt ist nur die aktuelle Position bekannt. Dies wird gelöst indem der erste Schritt mit der Bewegungsgleichung aus 3.2 berechnet wird. Erst danach kommt die klassische Verlet-Methode (17) zum Einsatz.[3]
Das bedeutet, dass für $i = 0$, also beim ersten Schritt, gilt:

$$\vec{x}_1 = \vec{x}_0 + \vec{v}_0 \Delta t + \frac{1}{2}\vec{a}_0 \Delta t^2 \tag{16}$$

und für $i > 0$

$$\vec{x}_{i+1} = 2\vec{x}_i - \vec{x}_{i-1} + \vec{a}\Delta t^2 \tag{17}$$

Weil diese Methode ohne Geschwindigkeit arbeitet, muss die Geschwindigkeit, die für den Vergleich mit den anderen Methoden notwendig ist, anderweitig berechnet werden. Da

mit dieser Geschwindigkeit nicht weiter gerechnet wird und sich dadurch auch kein Fehler fortpflanzen kann, sind die leicht ungenauen Werte, die hierbei entstehen, zur Auswertung völlig ausreichend. So kann einfach aus den vorherigen Positionen die aktuelle Bewegung ermittelt werden.

$$\vec{v}_n = \frac{\vec{x}_n - \vec{x}_{n-1}}{\Delta t} \tag{18}$$

3.5 Zeitkorrigierte Verlet-Integration

Dass der in 3.4 beschriebene Algorithmus ein einziges, allgemeines Δt verwendet und trotzdem zwei vorhergehende Werte mit einbezieht, lässt darauf schließen, dass $\Delta t_i = \Delta t_{i-1}$ sein muss. Um nicht mehr von einem konstanten Δt abhängig zu sein, kann Formel (19) verwenden:[3]

$$x_{i+1} = x_i + (x_i - x_{i-1}) \cdot \frac{\Delta t_i}{\Delta t_{i-1}} + a \cdot \frac{\Delta t_i + \Delta t_{i-1}}{2} \cdot \Delta t_i \tag{19}$$

Eine Unabhängigkeit von einem konstanten Δt ist besonders bei Echtzeitsimulationen wünschenswert, da Δt dann meistens von der leicht schwankenden Bildfrequenz abhängt, um immer ein gleichmäßiges und flüssiges Bild zu bieten. Bei konstantem Δt besteht kein Unterschied mehr zum ursprünglichen Verlet-Algorithmus, weil sich die veränderten Teile kürzen lassen: $\frac{\Delta t_i}{\Delta t_{i-1}} = 1$ und $\frac{\Delta t_i + \Delta t_{i-1}}{2} = \frac{2\Delta t_i}{2} = \Delta t_i$

3.6 Symplektisches Euler-Verfahren

Das symplektische oder auch semi-implizite Euler-Verfahren sieht dem normalen Euler-Verfahren aus 3.1 sehr ähnlich. Trotzdem gehört es zur Gruppe der strukturerhaltenden Zeitintegratoren[7], was bedeutet, dass es bei einem Hamilton-System, also einem realen, physikalischen System, die Energieerhaltung zumindest teilweise berücksichtigt. Die Rechenvorschrift lautet in dieser Reihenfolge:[7]

$$\vec{v}_{i+1} = \vec{v}_i + \vec{a} \cdot \Delta t \tag{20}$$

$$\vec{x}_{i+1} = \vec{x}_i + \vec{v}_{i+1} \cdot \Delta t \tag{21}$$

3.7 Klassisches Runge-Kutta-Verfahren

Nachdem nun die Notwendigkeiten einer konstanten Geschwindigkeit und eines konstanten Δt überwunden sind, entfernt das Runge-Kutta-Verfahren Vierter Ordnung jetzt auch teilweise die Notwedigkeit einer konstanten oder linearen Beschleunigung. Die allgemeine

Formel für den Runge-Kutta-Algorithmus vierter Ordnung lautet[8]

$$\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\
k_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{1}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4) + O(h^5)
\end{aligned} \tag{22}$$

Die Methode schließt alle Fehler, die Proportional zu h^4 (h ist die Integrationsschrittweite) oder weniger sind aus (nur noch Fehler der Ordnung $+O(h^5)$ bleiben bestehen). Das bedeutet, dass zum Beispiel bei doppelt so vielen Schritten in der selben Zeit die Genauigkeit $2^5 = 32$ mal höher sein sollte. Dieser Integrator ist aber eigentlich nur für die erste Ableitung vorgesehen.[8] Weil $\vec{x}$ aber die zweite zeitliche Ableitung der gegebenen Beschleunigung ist, muss ein leicht verändertes Verfahren angewendet werden.
Konkret nach den im n-Körperproblem bekannten Großen aufgelöst und unter Berücksichtigung der Notwendigkeit einer zweiten Ableitung lautet die Rechenvorschrift dann wie folgt: (auf Basis von[1])

$$\begin{aligned}
\vec{k_0} &= \Delta t \cdot \vec{v} \\
\vec{l_0} &= \Delta t \cdot \vec{a}(\vec{x}) \\
\vec{k_1} &= \Delta t \cdot (\vec{v_0} + \frac{1}{2}\vec{l_0}) \\
\vec{l_1} &= \Delta t \cdot \vec{a}(\vec{x_0} + \frac{1}{2}\vec{k_0}) \\
\vec{k_2} &= \Delta t \cdot (\vec{v_0} + \frac{1}{2}\vec{l_1}) \\
\vec{l_2} &= \Delta t \cdot \vec{a}(\vec{x_0} + \frac{1}{2}\vec{k_1}) \\
\vec{k_3} &= \Delta t \cdot (\vec{v_0} + \vec{l_2}) \\
\vec{l_3} &= \Delta t \cdot \vec{a}(\vec{x_0} + \vec{k_2}) \\
\vec{x}_{i+1} &= \vec{x_i} + \frac{1}{6}(\vec{k_0} + 2\vec{k_1} + 2\vec{k_2} + \vec{k_3}) \\
\vec{v}_{i+1} &= \vec{v_i} + \frac{1}{6}(\vec{l_0} + 2\vec{l_1} + 2\vec{l_2} + \vec{l_3})
\end{aligned} \tag{23}$$

Dazu wird erneut eine Funktion $\vec{a}(\vec{x})$ benötigt, die aber schon mit Formel (8) gegeben ist.

4 Implementierung

Um alle Methoden aus 3 auf Genauigkeit und Rechenzeit zu untersuchen, werden diese in der jeweils angegebenen iterativen Form ($\vec{x}_{i+1} = ...$) implementiert. Da die Dynamik von Himmelskörpern simuliert wird bietet es sich an, diese Bewegung auch zu visualisieren, um mögliche Fehler leichter zu finden und die Übersicht zu bewahren. Für die dreidimensionale Darstellung wurde die Unity Engine Version 5.3.4f1 gewählt, der Programmcode ist im Editor von Visual Studio 2015 Community Edition in C# geschrieben. Weil C# auf das .Net-Framework aufbaut sind bereits mehrere nützliche Klassen gegeben, wie zum Beispiel die Stopwatch Klasse, mit der sehr genau die Laufzeit eines bestimmten Codeabschnitts bestimmt werden kann. Ein weiterer Vorteil ist die Möglichkeit für einfaches Debugging, weil C# nur in die Intermediate Language übersetzt und dann Just-in-time kompiliert wird. Da das Programm aber von der .Net Runtime verwaltet wird, können bei kleinen Zeitmessungen für den selben Code leicht unterschiedliche Ergebnisse auftreten. Deshalb sind alle Messungen eher im Minutenbereich, damit die prozentuale Messungenauigkeit möglichst gering bleibt. Der Code ist so weit wie möglich auf Laufzeit optimiert, weshalb kleine Abweichungen zu den erläuterten Formeln auftreten. Die errechneten Daten werden dann im csv Format gespeichert, was die Analyse in Tabellenkalkulationsprogrammen stark vereinfacht.
Um den Code übersichtlicher zu gestalten wurden die Klassen Objekt und Vector3dbl eingeführt. Objekt enthält Positionsvektor, Geschwindigkeit, Kraft, Beschleunigung und Masse eines Objekts, während Vector3dbl einige Vektoroperationen zur Verfügung stellt sowie die x, y, und z Koordinaten als double beinhaltet. Alle Objekte werden in einem zentralen Array verwaltet, wobei nur jeweils die Objekte eines „layers“ miteinander Interagieren und die gleiche Berechnungsmethode teilen. Die gesamte Implementierung kann am Besten anhand des beigelegten, vollständigen Quellcodes nachvollzogen werden. Zusätzlich befindet sich eine kurze Gebrauchsanleitung für das fertige Programm in dessen Verzeichnis.

4.1 Berechnung der Beschleunigungen

Als Beispiel für die Optimierungen ist hier die Berechnung der Beschleunigungen näher erläutert. Da alle Verfahren die auf ein Objekt wirkende Beschleunigung benötigen, wird diese in einer zentralen Funktion berechnet. Anstatt die Beschleunigung auf jeden Körper wie in Formel (8) einzeln zu berechnen, wird hier Newtons drittes Gesetz angewendet, wonach $\vec{F}_1 = -\vec{F}_2$ ist. Das vermindert die Rechenschritte von $n \cdot (n-1)$ auf $\frac{n \cdot (n-1)}{2}$, also um 50%, wodurch sich die auch die Rechenzeit ohne Informationsverlust halbiert. Der zugehörige Code ist in Beispiel 1 enthalten.

```
1  for (int i = 0; i < anzahl; i++)
2    objekte[layer, i].kraft = Vector3dbl.zero;
3  for (int x = 0; x < anzahl; x++)
4  {
5    for (int y = x + 1; y < anzahl; y++)
6    {
7      Objekt a = objekte[layer, x], b = objekte[layer, y];
8      double entfernung = Vector3dbl.Entfernung(a.position,
           b.position);
9      double F = G * (a.masse * b.masse) / (entfernung *
           entfernung);
10     a.kraft += (b.position - a.position) * F / entfernung;
11     b.kraft -= a.kraft;
12   }
13 }
14 for(int i = 0;i < anzahl; i++)
15   objekte[layer,i].beschleunigung = objekte[layer,i].kraft/
       objekte[layer,i].masse;
```

Beispiel 1: Berechnung der Kraft

4.2 Umgang mit Fließkommaungenauigkeiten

Um eine möglichst hohe Genauigkeit bei den Berechnungen zu erhalten, wird nicht mit dem vordefinierten Vector3-Typ gerechnet. Dieser nutzt nämlich im Gegensatz zum neu erstellten Vector3dbl nur Fließkommazahlen mit einfacher Genauigkeit, also keine double. Da die verwendete Unity-Engine aber für die Grafikberechnungen nur float-Genauigkeit unterstützt, müssen die genauen Vektoren (nur zur Anzeige) umgewandelt werden. Daraus folgt die übliche Aufteilung in Modell und Beobachter: gerechnet wird mit genauen Werten, die aber für den Beobachter in halbe Genauigkeit konvertiert werden, damit sie angezeigt werden können.

4.3 Implementierung der einzelnen Verfahren

Die Implementierung jedes Verfahrens einzeln zu erklären würde bei Weitem den Rahmen der Arbeit sprengen. Deshalb sind alle in der Klasse „Berechnungen“ unter ihrem jeweiligen Namen enthalten. Diese Klasse befindet sich vollständig im Anhang. Die wichtigen Funktionen sind zur besseren Verständlichkeit kommentiert und die Variablen besitzen meist sinnvolle Namen.

5 Vergleich

Der relative Vergleich stellt kein Problem dar, die absolute Genauigkeit und Abweichung von den echten Werten festzustellen ist hingegen wegen der numerischen Natur des n-Körperproblems fast unmöglich. Leider sind auch keine schon auf Großrechnern berechneten Systeme als Vergleichsgröße frei erhältlich. Daher bieten sich nun zwei Möglichkeiten: Einerseits könnte man ein bereits bekanntes System, nämlich unser Sonnensystem, verwenden. Dazu müsste man alle massereichen Himmelskörper des Sonnensystems mit in die Berechnung einbeziehen, um die berechneten Positionen und Geschwindigkeiten mit denen aus der Datenbank des Jet-Propulsion-Laboratorys der NASA zu vergleichen. Diese Methode scheidet aber aus, weil Größen wie die Gravitationskonstante und die Massen der Objekte nur auf ca. fünf Nachkommastellen bekannt sind. Außerdem können weitere Abweichungen auf Grund von relativistischen Effekten oder der nicht punktförmigen Masseverteilung auftreten. Die Lösung *dieser* Probleme hätte aber nichts mehr mit der numerischen Stabilität und Genauigkeit zu tun, sondern mit den physikalischen Gegebenheiten und Messungenauigkeiten.
Die andere Möglichkeit ist, ein einfaches System mit dem genauesten Algorithmus und mehreren Größenordnungen kleineren Zeitschritten im Voraus zu berechnen. Da die angenäherten Werte beim Runge-Kutta-Verfahren Vierter Ordnung für $\Delta t \to 0$ am schnellsten gegen die exakten Ergebnisse konvergieren (siehe (22), Fehlerordnug h^5), können diese dann als Vergleichsgrößen verwendet werden. Um weitere Fehlerquellen auszuschließen wird ein Zweikörpersystem ohne chaotische Elemente simuliert.

5.1 Anfangsbedingungen

In allen Vergleichen wird das selbe System mit den gleichen Startbedingungen verwendet.
Der Wert der verwendeten Gravitationskonstante beträgt $6.6740831 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$ [5]. Die Startwerte der Körper 1 und 2 sind nebenstehend in Tabelle 1 in Meter, Kilogramm und Meter pro Sekunde enthalten. Durch den Unterschied in der Masse um zehn Größenordnungen bleibt Objekt 1 relativ stabil ohne Bewegung. Dieser Umstand lässt außerdem eine Beurteilung der Verfahren aufgrund der Bewegung des einen, leichten Objektes zu. Daher wird im Folgenden auch nur die Abweichung der Bahn von Objekt 2 verglichen, das sich auf einem stabilen elliptischen Orbit mit maximalen Bahnradius von 10^7 Kilometern in der $x_1 - x_2$-Ebene befindet.

Tabelle 1: Startwerte

	Objekt 1	Objekt 2
$\vec{x}$	$\begin{pmatrix} 0 \\ 10^{10} \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$
$\vec{v}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 10^5 \\ 0 \\ 0 \end{pmatrix}$
m	$2 \cdot 10^{30}$	$2 \cdot 10^{20}$

5.2 Abweichung vom Ergebnis bei gleichem Zeitschritt

Für den ersten Vergleich wurde ein Δt von ca. $1666\frac{2}{3}s$ (in double Präzision) verwendet. Dabei wurden 200 Tage mit allen sieben Methoden aus 3 simuliert, was ungefähr 44 Umläufen entspricht, weshalb deutliche Abweichungen schon erkennbar sein sollten. Die optimalen Positionswerte wurden mit Runge-Kutta Vierter Ordnung und einem 10^3 mal kleineren Zeitschritt berechnet, was bei der in (22) angegebenen Fehlerordnung ca. $(10^3)^5 = 10^{15}$ mal bessere Ergebnisse liefern sollte. Der nächste Absatz bezieht sich voll-

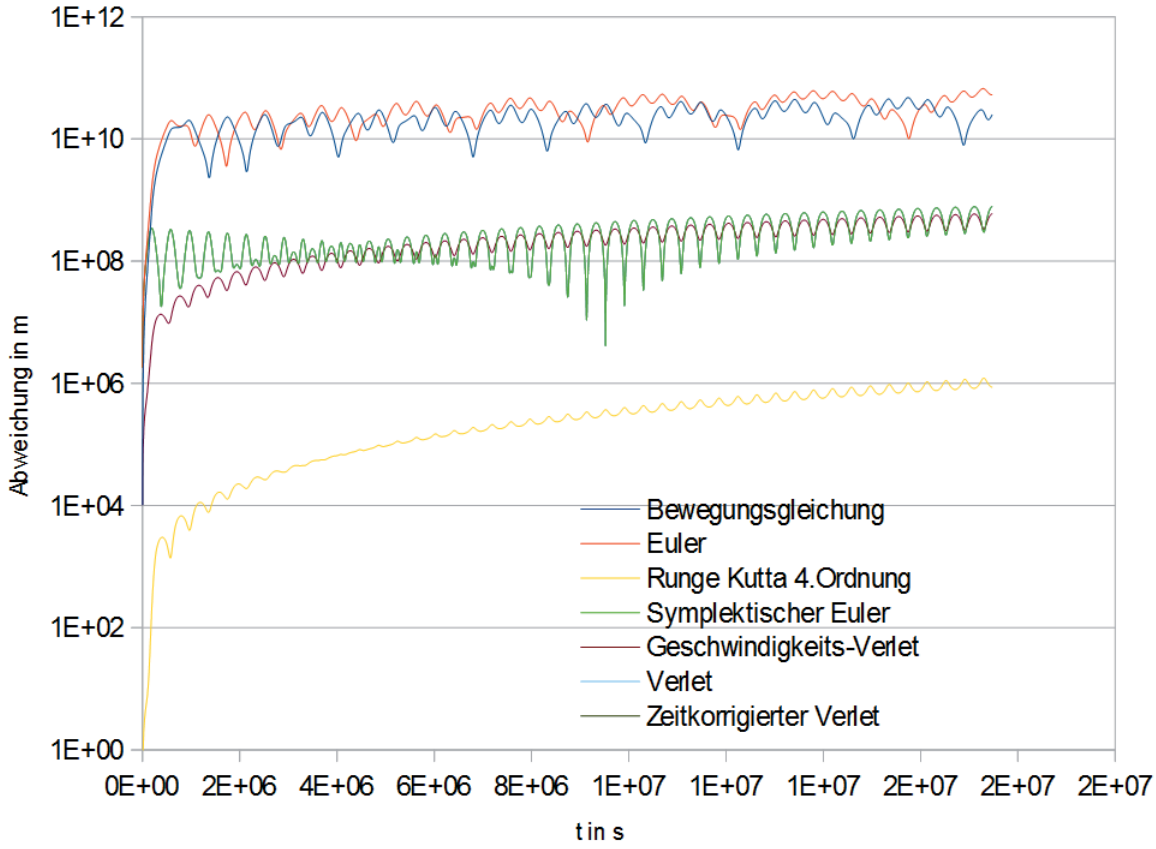

Abbildung 3: Abweichung vom Optimalwert

ständig auf Abbildung 3.

Um die unterschiedlichen Verfahren anschaulich und aussagekräftig in einem Diagramm darzustellen, wurde für die Abweichung eine logarithmische Skala verwendet. Daraus folgt, dass Graphen, die Logarithmusfunkionen ähnlich sehen, eigentlich eine mit der Zeit linear steigende Abweichung anzeigen.

Zunächst fällt auf, dass scheinbar der Verlet-Algorithmus und der Zeitkorrigierte-Verlet nicht abgebildet sind. Dass sie aufgrund ihrer ähnlichen Beschaffenheit bei gleichbleibendem Δt keine relative Abweichung haben und deshalb nicht voneinander zu unterscheiden sind, war zu erwarten. Erstaunlich aber ist, dass diese beiden identischen Verfahren vom

symplektischen Euler-Verfahren überdeckt werden, einen völlig anderen Aufbau vorweist. Die nächste Auffälligkeit ist die Aufteilung in drei Gruppen, die sich jeweils in der Genauigkeit mindestens 100-Fach unterscheiden. Der Runge-Kutta Algorithmus ist mit hundert- bis tausendfach besseren Ergebnissen bei gleicher Schrittzahl das präziseste Verfahren und bildet alleine die Gruppe mit der kleinsten Abweichung.
Die zweite Gruppe setzt sich aus den verschiedenen Verlet-Verfahren und dem semi-impliziten Euler-Verfahren zusammen. Trotz der kleineren Amplitude des Geschwindigkeits-Verlet-Verfahrens sind diese vier Verfahren als ungefähr gleichwertig anzusehen, da sie besonders gegen Ende der Messung um einen gemeinsamen Wert der Abweichung oszillieren.
Die größte Abweichung von den Vergleichswerten haben die Lösungsansätze des Euler-

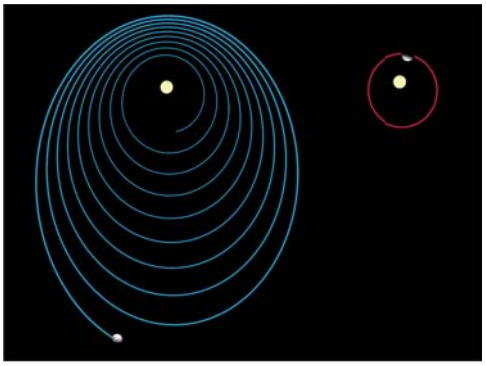

Abbildung 4: Euler-Verfahren(links) gegenüber Runge-Kutta-Verfahren(rechts) nach gleicher Zeit (200 Tage)

Verfahrens und der Lösung mit Hilfe der Bewegungsgleichungen. Ihre Ungenauigkeit bewegt sich nicht wie bei den anderen Verfahren im Bereich des gleichen Orbits, sondern beträgt vielfache des maximalen Bahnradius (10^{10} Meter). Das liegt daran, dass der Körper im Orbit bei diesen Methoden nicht an die Energieerhaltung gebunden ist. Dies zeigt sich auch deutlich in Abbildung 4. Bis auf die unpräziseste Gruppe dieses Absatzes ähneln alle Verfahren eher dem Runge-Kutta-Verfahren aus Abb. 4. Beim Euler-Verfahren higegen entfernt sich der Planet immer weiter von der Sonne statt auf einem Orbit zu bleiben. Selbst wenn dies in der logarithmischen Skalierung nur schwer zu erkennen ist, kann eine leichte Tendenz der Bewegungsgleichung zu genaueren Werten festgestellt werden.
Die periodischen Schwingungen in Abbildung 3 lassen sich ebenfalls anhand Abbildung 4 erklären: Durch den leicht elliptischen Orbit ist das Objekt beim Anflug Richtung Sonne etwas schneller als das potentiell vor ihm liegende, richtig berechnete Vergleichsobjekt, das den tiefsten Punkt bereits wieder verlässt und somit abgebremst wird. Deshalb holt das zurückliegende Objekt etwas auf und nähert sich mehr den richtigen Werten an, bevor es selbst abgebremst wird. Diese regelmäßige Änderung der Abweichung kann bei den mit Euler und Bewegungsgleichung berechneten Objekten nicht beobachtet werden, weil diese

bereits nach kurzer Zeit einen völlig anderen, spiralförmigen Orbit mit anderer Umlaufzeit beschreiben.

5.3 Veränderung zu 5.2 bei halbem Zeitschritt

Wenn man nun die Anzahl der Schritte pro Zeit verdoppelt, also Δt auf ca. $833\frac{1}{3}s$ halbiert, erhält man ein ähnliches Bild (Abb. 5).

Hierbei können erneut die drei Auffälligkeiten aus 5.2 nachvollzogen werden: Die ge-

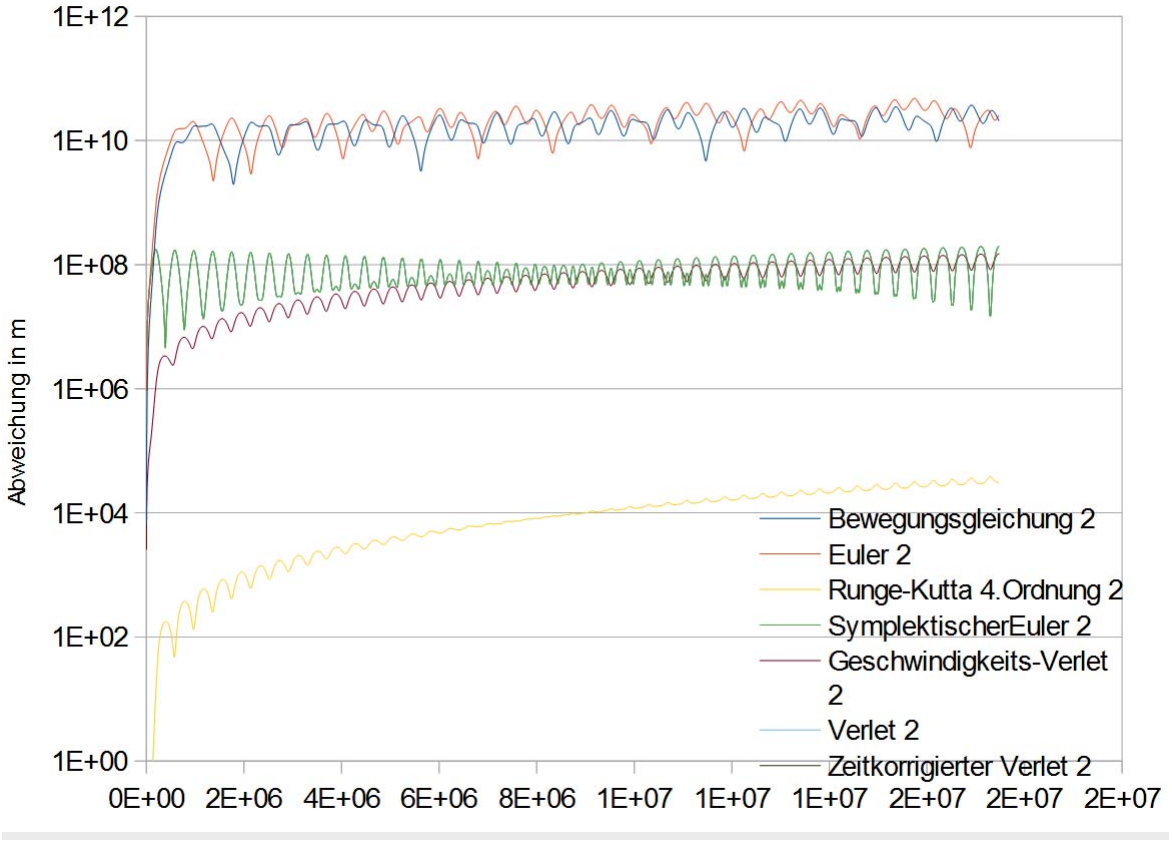

Abbildung 5: Abweichung vom Optimalwert

genseitige Überdeckung der Verlet-Verfahren, die Aufteilung in drei Gruppen und das periodische „Schwingen“ der unteren zwei Gruppen. Zum besseren Verständnis sollte die Aufmerksamkeit diesmal aber auch auf die Zahlenwerte gerichtet werden, die sich in fast allen Fällen stark unterscheiden. Und sich merklich verbessern: Die Verlet-Verfahren und der symplektische Euler sind bei Ende der Messung im Bereich der Abweichung von 10^8 im Gegensatz zu 5.2 mit einer fast zehn mal größeren Abweichung (10^9). Der Runge-Kutta-Algorithmus hingegen hat sich bei halbiertem Zeitschritt um fast ungefähr das 30-fache verbessert($10^6 \rightarrow 3 \cdot 10^4$). Um die Verbesserung bei Euler und der Bewegungsgleichung zu veranschaulichen müssen die Daten anders präsentiert werden, weil diese Verfahren schon

völlig andere Systeme beschreiben. In Abbildung 6 kann man erkennen, dass die mit besagten Methoden berechneten Bahnen langsamer divergieren. Trotzdem hat die Halbierung

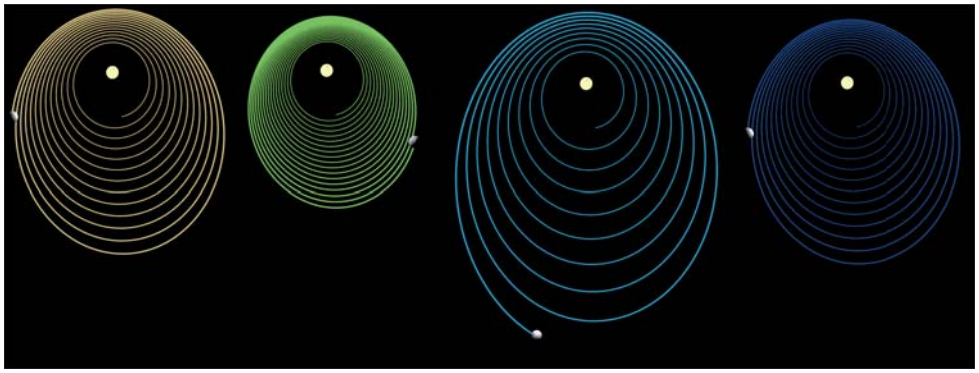

Abbildung 6: Von links nach rechts: Bewegungsgleichung(Δt), Bewegungsgleichung($\frac{\Delta t}{2}$), Euler(Δt), Euler($\frac{\Delta t}{2}$) nach gleicher Zeit (200 Tage)

des Zeitschritts einen unterschiedlich starken Effekt auf die verschiedenen Gruppen von Berechnungsmethoden, was in 5.4 genauer untersucht wird.

5.4 Abhängigkeit der Genauigkeit von der Größe des Zeitschritts

Um genaue Ergebnisse zu erhalten, ist es notwendig, mehrere Tests durchzuführen, um zufällige Ergebnisse ausschließen zu können. Deshalb wird das bisher verwendete System jetzt mit einfacher, doppelter, vierfacher und achtfacher Schrittanzahl berechnet und die Ergebnisse in Abbildung 7 auf ungewöhnliche Art aufgetragen:
eingezeichnet wird jeweils, wie viel mal näher das Verfahren dem exakten Wert gekommen ist bei jeweils halbiertem Δt. Dazu wird die erneut die jeweilige Abweichung der Position nach simulierten 200 Tagen verwendet. Dann wird die Abweichung bei ursprünglichem Δt durch die Abweichung bei $\frac{1}{2}/\frac{1}{4}/\frac{1}{8}$ Δt geteilt, um die relative Verbesserung zum ursprünglichen Δt erhalten.
Wie in 5.2 und 5.3 zeigen sich wieder die drei Gruppen. Hierbei ist auffällig, dass sich erneut die Verlet-Verfahren und der symplektischer Euler (beinahe) gleiche Werte teilen. Dank der logarithmischen Skalierung beider Achsen wird eine Funktion der Form $f(x) = x^n$ als Gerade mit der Steigung n abgebildet, weshalb man die Abhängigkeit direkt von der Steigung der Geraden ablesen kann. In diesem Fall verbessert sich die Genauigkeit bei jeder Verdopplung der Schritte ungefähr den Faktor vier, was auf eine quadratische Abhängigkeit (Steigung 2, also x^2) der Genauigkeit pro Zeitschritt schließen lässt. Das trifft auch auf den Geschwindigkeits-Verlet zu.
Die schon vorher eher chaotischen Integratoren Euler und Bewegunsgleichung zeigen erneut scheinbar zufälliges Verhalten: es ist nicht ersichtlich wie Zeitschrittgröße und Genauigkeit voneinander abhängen.

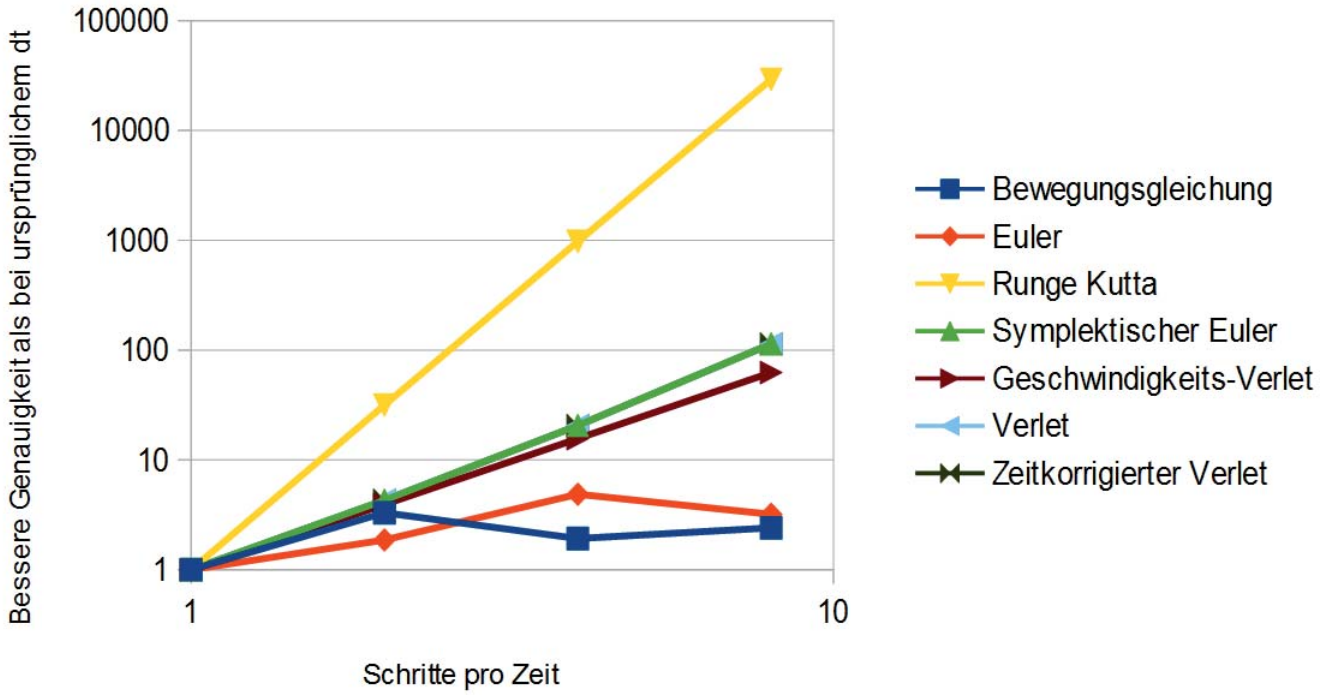

Abbildung 7: Verbesserung der Genauigkeit

Der Runge-Kutta-Algorithmus hingegen behält seine Genauigkeit bei und steigt trotz der logarithmischen Skalierung stark (mit Steigung 5) an. Dieses Verfahren erhöht also jeder Halbierung des Δt seine Präzision ungefähr um das 32-fache. Das bedeutet, seine Abweichung ist, wie in Theorie bereits in Formel (22) festgestellt, über Δt^5 zur Zeitschrittgröße proportional.

5.5 Genauigkeit pro Rechenzeit

Für die Berechnung und für Echtzeitsimulationen von Mehrkörpersystemen ist wahrscheinlich die wichtigste Frage, mit welcher Methode man in der kürzesten Zeit die besten Ergebnisse erreicht. Um diese Fragestellung zu lösen, kann man das Problem weiter vereinfachen.
Die Rechenzeit für die Beschleunigung steigt quadratisch mit der Anzahl der Körper(siehe 4.1), die Zeit für die eigentliche Umsetzung der Formel nur linear, weil dieser Teil nur einmal pro Objekt ausgeführt werden muss. Da die beschriebenen Methoden hauptsächlich Anwendung in der Lösung eines Vielkörpersystem finden, kann der linear ansteigende Anteil der Rechenzeit gegenüber dem quadratischen bei einer großen Anzahl von Körpern vernachlässigt werden. Also beeinflussen nur die Aufrufe der Methode *Beschleunigung-Berechnen* die Rechenzeit wesentlich. Außer der Runge-Kutta-Methode, die vier dieser Aufrufe benötigt, verwenden alle Verfahren diese Methode nur ein mal und sind daher

von der Laufzeit nahezu identisch. Deshalb sind bereits die Verfahren, die bei identischem Zeitschritt eine niedrigere Genauigkeit vorweisen(also Euler und Bewegungsgleichung), zu vermeiden.
Wenn allerdings eine sehr große Genauigkeit gefordert ist, stellt das Runge-Kutta-Verfahren doch die beste Leistung der behandelten Verfahren bereit. Wenn man dessen Schrittweite um Faktor vier erhöht, negiert das die vier Funktionsaufrufe von *BeschleunigungBerechnen* und macht es wieder den anderen Verfahren gleichwertig. Weil die Genauigkeit dieser Methode aber die der andere bei Weitem übertrifft (bei gleichem Zeitschritt ca. 100-fach, siehe 5.2) und mit Δt^5 steigt können damit dann trotzdem noch sehr genaue Werte berechnet werden.

6 Fazit

Die Wahl der Berechnungsmethode bleibt letztendlich abhängig vom jeweiligen Anspruch: Wenn nur mäßige Genauigkeit gefordert und die Rechenzeit entscheidend ist, sollte auf das symplektische Euler-Verfahren zurückgegriffen werden. Es überbietet das Verlet-Verfahren sogar noch, obwohl es sehr ähnliche Eigenschaften im Bezug auf Genauigkeit pro Δt hat, indem es durch seine Unabhängigkeit von vorherigen Schritten sehr viel leichter zu implementieren ist. Wenn allerdings mehr Genauigkeit notwendig ist oder ein System auch für längere Zeit stabil sein muss, ist das Runge-Kutta-Verfahren Vierter Ordnung fast unentbehrlich. Trotz der vergleichsweise schweren Umsetzung als Programmcode bietet es bei geringem Rechenaufwand bereits die genauesten Ergebnisse aller hier erwähnten Algorithmen.
Beide hier empfohlenen Methoden sind auch gut für Echtzeitsimulationen geeignet, weil sie anders als die klassische Verlet-Integration kein konstantes Δt voraussetzen. Das ermöglicht eine Anpassung an die Framerate und „vorspulbare“ Simulationen.

Literatur

[1] Amzoti. Help with using the Runge-Kutta 4th order method on a system of 2 first order ODE's, aufgerufen am 02.11.2016. URL http://math.stackexchange.com/questions/721076/help-with-using-the-runge-kutta-4th-order-method-on-a-system-of-2-first-order-od.

[2] Patric Büchele. Klassische Molekulardynamik Simulationen, Seite 10, 11.11.2009. URL https://www.icp.uni-stuttgart.de/~icp/mediawiki/images/5/50/Hs0910_buechele_ausarbeitung.pdf.

[3] Jonathan "lonesock"Dummer. A Simple Time-Corrected Verlet Integration Method, aufgerufen am 02.11.2016. URL http://lonesock.net/article/verlet.html.

[4] Rüdiger Mitdank. Numerische Integration der Bewegungsgleichung, aufgerufen am 02.11.2016. URL http://people.physik.hu-berlin.de/~mitdank/dist/scriptenm/Eulerintegration.htm.

[5] Barry N. Taylor Peter J. Mohr, David B. Newell. CODATA Recommended Values of the Fundamental Physical Constants: 2014 (arXiv:1507.07956 [physics.atom-ph]), 2015.

[6] sam. Understanding basic motion calculations in games: Euler vs. Verlet , aufgerufen am 02.11.2016. URL http://lolengine.net/blog/2011/12/14/understanding-motion-in-games.

[7] Leonard Schlag. Einführung in die Numerik strukturerhaltender Zeitintegratoren (Kapitel 1.4.2, S.6), 2010. URL https://www-m2.ma.tum.de/foswiki/pub/M2/Allgemeines/HauptseminarWohlmuthWS10/Ausarbeitung_Einfuehrung.pdf.

[8] Eric W. Weisstein. Runge-Kutta Method From MathWorld–A Wolfram Web Resource, aufgerufen am 02.11.2016. URL http://mathworld.wolfram.com/Runge-KuttaMethod.html.

[9] Edmund Taylor Whittaker. *Analytische Dynamik der Punkte und starren Körper*, chapter 14. Die Sätze von Bruns und Poincaré. 1924.

Ich habe diese Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt.

Ort, Datum	Unterschrift

Anhang

...minar Numerik\Seminararbeit\Anhang\Code\Berechnungen.cs 1

```
1  using UnityEngine;
2  using System.Diagnostics;
3  using System.Collections;
4  using System.Collections.Generic;
5  using UnityEngine.UI;
6  using System.IO;
7  using System;
8   //Hier laufen die Berechnungen ab
9  public class Berechnungen : MonoBehaviour
10 {
11     public GameObject[,] Objekte; //Die Engine Objekte der Planeten
12     public double zeitbeschleunigung = 1; //...
13     public float vergrößerung; //verhältnis berechneter Werte zu Positionen
         für die Engine
14     public string GeladenName = "Error"; //nur für UI
15     const double G = 6.6740831E-11; //Gravitationskonstante
16     public Objekt[,] objekte; //alle Objekte in allen layern, von außerhalb
         vor dem Start gesetzt
17     public Layer[] layers; //informationen zu den einzelnen Layern(z.B
         Iterationstiefe, Methode etc.)
18     GlobalGlobalVariables GlobVar; //Enthält Globale Variablen wie die
         Zeitbeschleunigung
19     const double fps = 60;//target framerate
20     public double[] zeit; //Rechenzeit jedes layers
21     const double invfps = 1 / fps; //Konstante um Rechenzeit zu sparen
22     public double[] prevdt; //vorheriges dt, wichtig für Zeitkorrigierten
         Verlet
23     public int anzahl; //anzahl der Objekte von außerhalb vor dem Start
         gesetzt
24     public string pfad = @"D:\Unity\Planeten\Output";
25     public bool schreiben = true; //ob ouputfiles generiert werden
26     public bool pause = true; //ob pausiert ist
27     public double schreibintervall = 1000; //Daten werden alle
         schreibintervall Sekunden gespeichert (csv)
28     double schreibtimer = 0; //Hilfsvariable zum herunterzählen der Zeit bis
         zum nächsten Speichern der Daten
29     StreamWriter[] Schreiber; //Ein Streamwriter für jedes Objekts
30     Stopwatch stp; //Die Stoppuhr
31     int untilflush = 60; //alle 60 frames den Filestream leeren
32     public double VergangeneZeit = 0;//in Sekunden
33
34     public enum Methode //Enthält alle Berechnungsmethoden
35     {
36         Bewegungsgleichung,
37         Euler,
38         RungeKutta4,
39         SymplektischerEuler,
40         VelocityVerlet,
41         Verlet,
42         ZeitKorrigiertVerlet
43     };
44
45
46     void Awake() //Diese Funktion wird beim Start des Programms vor Start()
         aufgerufen
47     {
```

```
48          Application.targetFrameRate = (int)fps;
49          pfad = Application.dataPath + @"\Output";
50      }
51
52      void Start() //Diese Funktion wird beim Start des Programms aufgerufen
53      {
54          stp = new Stopwatch();
55          GlobVar = GameObject.Find
              ("Scripts").GetComponent<GlobalGlobalVariables>();
56          zeitbeschleunigung = GlobVar.timescale;
57          vergrößerung = GlobVar.scale;
58      }
59      public void startdt()//Diese Funktion wird beim Start der Simulation
          aufgerufen
60      {
61          VergangeneZeit = 0;
62          prevdt = new double[layers.Length];
63          for (int i = 0; i < layers.Length; i++)
64          {
65              prevdt[i] = zeitbeschleunigung * invfps / layers
                  [i].Iterationstiefe;
66          }
67          try//hiermit werden die output files geöffnet
68          {
69              if (schreiben)
70              {
71                  untilflush = 60;
72                  Schreiber = new StreamWriter[anzahl];
73                  for (int y = 0; y < anzahl; y++)
74                  {
75                      string k = pfad + @"\" + GeladenName;
76                      if (!Directory.Exists(k))
77                      {
78                          Directory.CreateDirectory(k);
79                      }
80                      k += @"\" + Objekte[0, y].name.Remove(Objekte[0,
                        y].name.IndexOf('(')).TrimEnd(' ') + ".csv";
81                      Schreiber[y] = new StreamWriter(k, false);
82                      string s = "t;dt";
83                      for (int i = 0; i < layers.Length; i++)
84                      {
85                          s += ";Layer " + i + ":" + layers[i].Methode.ToString
                        ("F") +
                        ";x;y;z;vx;vy;vz;v;ax;ay;az;Iterationen;Rechenzeit";
86                      }
87                      Schreiber[y].WriteLine(s);
88                      Schreiber[y].Flush();
89                  }
90              }
91          }catch(Exception ex)
92          {
93              UnityEngine.Debug.Log(ex.Message);
94          }
95      }
96
97      void OnApplicationQuit() //Dokumente Schließen, damit sie eider bearbeitet
```

```
        werden können
98      {
99          CloseWriters();
100     }
101
102     void Update()
103     {
104         if (Input.GetKeyDown(KeyCode.P)) //Der Pause key...
105             pause = !pause;
106         if (layers.Length < 1 || pause)//wenn keine Objekte oder pause, mache
              nichts
107             return;
108         zeitbeschleunigung = GlobVar.timescale;
109         if (Schreiber != null && schreibtimer <= 0) //ausgabe
110         {
111             for (int x = 0; x < anzahl; x++)
112             {
113                 string s = VergangeneZeit.ToString() + ";" +
                      zeitbeschleunigung * invfps ;
114                 for (int y = 0; !schreiben || y < layers.Length; y++)
115                 {
116                     s += "; ;" + objekte[y, x].position + ";" + objekte[y,
                        x].geschwindigkeit + ";" + objekte[y,
                        x].geschwindigkeit.Betrag() + ";"+ objekte
                        [y,x].beschleunigung + ";" + layers[y].Iterationstiefe +
                        ";" + zeit[y];
117                 }
118                 if (Schreiber[x] != null)
119                     Schreiber[x].WriteLine(s);
120             }
121             schreibtimer += schreibintervall;
122         }
123         untilflush--;
124         if (untilflush <= 0)
125         {
126             untilflush = 60;
127             if (Schreiber != null)
128                 foreach (StreamWriter s in Schreiber)
129                 {
130                     if (s != null)
131                         s.Flush();
132                 }
133         }
134         for (int i = 0; i < layers.Length; i++)//Berechnungen für alle Layer
135         {
136             switch (layers[i].Methode)
137             {
138                 case Methode.Bewegungsgleichung:
139                     BerechneBewegungsgleichung(i, layers[i].Iterationstiefe);
140                     break;
141                 case Methode.VelocityVerlet:
142                     BerechneVelocityVerlet(i, layers[i].Iterationstiefe);
143                     break;
144                 case Methode.RungeKutta4:
145                     BerechneRK4(i, layers[i].Iterationstiefe);
146                     break;
```

```
147                 case Methode.SymplektischerEuler:
148                     BerechneSymplektischEuler(i, layers[i].Iterationstiefe);
149                     break;
150                 case Methode.Euler:
151                     BerechneEuler(i, layers[i].Iterationstiefe);
152                     break;
153                 case Methode.ZeitKorrigiertVerlet:
154                     BerechneZeitKorrigiertVerlet(i, layers
                        [i].Iterationstiefe);
155                     break;
156                 case Methode.Verlet:
157                     BerechneVerlet(i, layers[i].Iterationstiefe);
158                     break;
159                 default:
160                     BerechneEuler(i, layers[i].Iterationstiefe);
161                     break;
162             }
163         }
164         PositionenSetzen();//Positionen vom Modell auf die Objekte in der
              Engine übertragen
165         VergangeneZeit += zeitbeschleunigung * invfps;
166         schreibtimer -= zeitbeschleunigung * invfps;
167     }
168 
169     void PositionenSetzen()//steht oben...
170     {
171         for(int x = 0; x<layers.Length;x++)
172         {
173             for(int y = 0;y<anzahl;y++)
174             {
175                 Objekte[x, y].transform.position = objekte[x, y].position /
                      vergrößerung;
176             }
177         }
178     }
179 
180     public void CloseWriters()//Streams leeren und schließen
181     {
182         if(Schreiber != null)
183         foreach (StreamWriter s in Schreiber)
184         {
185                 if (s != null)
186                 {
187                     s.Flush();
188                     s.Close();
189                 }
190         }
191     }
192 
193     void BerechneBeschleunigung(int layer)//Erklärt in der Arbeit bei 4.1
194     {
195         for (int i = 0; i < anzahl; i++)
196             objekte[layer, i].kraft = Vector3dbl.zero;
197         for (int x = 0; x < anzahl; x++)
198         {
199             for (int y = x + 1; y < anzahl; y++)
```

```
200             {
201                 Objekt a = objekte[layer, x], b = objekte[layer, y];
202                 double entfernung = Vector3dbl.Entfernung(a.position,
                      b.position);
203                 double F = G * (a.masse * b.masse) / (entfernung *
                      entfernung);
204                 a.kraft += (b.position - a.position) * F / entfernung;
205                 b.kraft -= a.kraft;
206             }
207         }
208         for (int i = 0; i < anzahl; i++)
209         {
210             objekte[layer, i].beschleunigung = objekte[layer, i].kraft /
                  objekte[layer, i].masse;
211         }
212     }
213     //Die Verschiedenen Integratoren, setzen eigentlich nur die Formeln aus 2
          in der Arbeit um
214
215     void BerechneEuler(int layer, int Iterationen)
216     {
217         stp.Reset();
218         stp.Start();
219         if (objekte != null)
220         {
221             double dt = invfps / Iterationen * zeitbeschleunigung;
222             for (int k = 0; k < Iterationen; k++)
223             {
224                 if (anzahl > 1)
225                 {
226                     BerechneBeschleunigung(layer);
227                     for (int i = 0; i < anzahl; i++)
228                     {
229                         objekte[layer, i].position += objekte[layer,
                          i].geschwindigkeit * dt;
230                         objekte[layer, i].geschwindigkeit += objekte[layer,
                          i].beschleunigung * dt;
231                     }
232                 }
233             }
234         }
235         zeit[layer] = stp.Elapsed.TotalMilliseconds;
236     }
237
238     void BerechneSymplektischEuler(int layer, int Iterationen)
239     {
240         stp.Reset();
241         stp.Start();
242         if (objekte != null)
243         {
244             double dt = invfps / Iterationen * zeitbeschleunigung;
245             for (int k = 0; k < Iterationen; k++)
246             {
247                 if (anzahl > 1)
248                 {
249                     BerechneBeschleunigung(layer);
```

```
250                     for (int i = 0; i < anzahl; i++)
251                     {
252                         objekte[layer, i].geschwindigkeit += objekte[layer,
                        i].beschleunigung * dt;
253                         objekte[layer, i].position += objekte[layer,
                        i].geschwindigkeit * dt;
254                     }
255                 }
256             }
257         }
258         zeit[layer] = stp.Elapsed.TotalMilliseconds;
259     }
260
261     void BerechneVerlet(int layer, int Iterationen)
262     {
263         stp.Reset();
264         stp.Start();
265         if (objekte == null)
266             return;
267         if (anzahl < 2)
268             return;
269         double dt = zeitbeschleunigung * invfps / Iterationen;
270         for (int k = 0; k < Iterationen; k++)
271         {
272             BerechneBeschleunigung(layer);
273             for (int i = 0; i < anzahl; i++)
274             {
275                 if (VergangeneZeit == 0)
276                 {
277                     objekte[layer, i].geschwindigkeit += objekte
                    [layer,i].beschleunigung * dt;
278                     objekte[layer, i].prevpos = objekte[layer, i].position;
279                     objekte[layer, i].position += objekte[layer,
                    i].geschwindigkeit * dt;
280                 }
281                 else
282                 {
283                     Vector3dbl pp = objekte[layer, i].position;
284                     objekte[layer, i].position = 2 * objekte[layer,
                    i].position - objekte[layer, i].prevpos  + objekte
                    [layer,i].beschleunigung * dt * dt ;
285                     objekte[layer, i].prevpos = pp;
286                     objekte[layer, i].geschwindigkeit = (objekte[layer,
                    i].position - objekte[layer, i].prevpos) / dt;
287                 }
288             }
289             prevdt[layer] = dt;
290         }
291         zeit[layer] = stp.Elapsed.TotalMilliseconds;
292     }
293
294     void BerechneZeitKorrigiertVerlet(int layer, int Iterationen)
295     {
296         stp.Reset();
297         stp.Start();
298         if (objekte == null)
```

```
299             return;
300         if (anzahl < 2)
301             return;
302         double dt = zeitbeschleunigung * invfps / Iterationen;
303         for (int k = 0; k < Iterationen; k++)
304         {
305             BerechneBeschleunigung(layer);
306             for (int i = 0; i < anzahl; i++)
307             {
308                 if (VergangeneZeit == 0)
309                 {
310                     objekte[layer, i].geschwindigkeit += objekte[layer,
                    i].beschleunigung * dt;
311                     objekte[layer, i].prevpos = objekte[layer, i].position;
312                     objekte[layer, i].position += objekte[layer,
                    i].geschwindigkeit * dt;
313                 }
314                 else
315                 {
316                     Vector3dbl pp = objekte[layer, i].position;
317                     objekte[layer, i].position = objekte[layer, i].position +
                    (objekte[layer, i].position - objekte[layer, i].prevpos) *
                     (dt / prevdt[layer]) + objekte[layer, i].beschleunigung *
                     dt * (dt+prevdt[layer])/2;
318                     objekte[layer, i].prevpos = pp;
319                     objekte[layer, i].geschwindigkeit = (objekte[layer,
                    i].position - objekte[layer, i].prevpos) / dt;//nur zum
                    Spaß, Performance?
320                 }
321             }
322             prevdt[layer] = dt;
323         }
324 
325         zeit[layer] = stp.Elapsed.TotalMilliseconds;
326     }
327 
328     void BerechneBewegungsgleichung(int layer, int Iterationen)
329     {
330         if (anzahl < 2)
331             return;
332         stp.Reset();
333         stp.Start();
334         if (objekte != null)
335         {
336             double dt = invfps / Iterationen * zeitbeschleunigung;
337             for (int k = 0; k < Iterationen; k++)
338             {
339                 if (anzahl > 1)
340                 {
341                     BerechneBeschleunigung(layer);
342                     for (int i = 0; i < anzahl; i++)
343                     {
344                         objekte[layer, i].position += objekte[layer,
                        i].geschwindigkeit * dt + 0.5* objekte
                        [layer,i].beschleunigung * dt*dt;
345                         objekte[layer, i].geschwindigkeit += objekte[layer,
```

```
                    i].beschleunigung * dt;
346                  }
347              }
348          }
349      }
350      zeit[layer] = stp.Elapsed.TotalMilliseconds;
351  }
352
353  void BerechneVelocityVerlet(int layer, int Iterationen)
354  {
355      if (anzahl < 2)
356          return;
357      stp.Reset();
358      stp.Start();
359      if (objekte != null)
360      {
361          double dt = invfps / Iterationen * zeitbeschleunigung;
362          Vector3dbl[] prevA = new Vector3dbl[anzahl];
363          for (int k = 0; k < Iterationen; k++)
364          {
365              if (anzahl > 1)
366              {
367                  if(VergangeneZeit == 0)
368                  BerechneBeschleunigung(layer);
369                  for (int i = 0; i < anzahl; i++)
370                  {
371                      prevA[i] = objekte[layer, i].beschleunigung;
372                      objekte[layer, i].position += objekte[layer,
                    i].geschwindigkeit * dt + 0.5 * objekte[layer,
                    i].beschleunigung * dt * dt;
373                  }
374                  BerechneBeschleunigung(layer);
375                  for(int i = 0;i< anzahl;i++)
376                  {
377                      objekte[layer, i].geschwindigkeit += 0.5 * (objekte
                    [layer,i].beschleunigung + prevA[i]) * dt;
378                  }
379              }
380          }
381      }
382      zeit[layer] = stp.Elapsed.TotalMilliseconds;
383  }
384
385  void BerechneBeschleunigungAnPos(Vector3dbl[] pos,int layer)
386  {
387      for (int i = 0; i < anzahl; i++)
388          objekte[layer, i].kraft = Vector3dbl.zero;
389      for (int x = 0; x < anzahl; x++)
390      {
391          for (int y = x + 1; y < anzahl; y++)
392          {
393              Objekt a = objekte[layer, x], b = objekte[layer, y];
394              double entfernung = Vector3dbl.Entfernung(pos[x], pos[y]);
395              double F = G * (a.masse * b.masse) / (entfernung *
                  entfernung);
396              a.kraft += (pos[y] - pos[x]) * F / entfernung;
```

```
397                 b.kraft -= a.kraft;
398             }
399         }
400         for (int i = 0; i < anzahl; i++)
401         {
402             objekte[layer, i].beschleunigung = objekte[layer, i].kraft /
                  objekte[layer, i].masse;
403         }
404     }
405
406     void BerechneRK4(int layer, int Iterationen)
407     {
408         stp.Reset();
409         stp.Start();
410         if (objekte != null)
411         {
412             double dt = invfps / Iterationen * zeitbeschleunigung;
413             for (int k = 0; k < Iterationen; k++)
414             {
415                 #region einSchritt
416                 if (anzahl > 1)
417                 {
418                     Vector3dbl[] k0 = new Vector3dbl[anzahl],
419                         k1 = new Vector3dbl[anzahl],
420                         k2 = new Vector3dbl[anzahl],
421                         k3 = new Vector3dbl[anzahl],
422                         l0 = new Vector3dbl[anzahl],
423                         l1 = new Vector3dbl[anzahl],
424                         l2 = new Vector3dbl[anzahl],
425                         l3 = new Vector3dbl[anzahl];
426                     Vector3dbl[] tmppos = new Vector3dbl[anzahl];
427                     BerechneBeschleunigung(layer);
428                     for (int i = 0; i < anzahl; i++)
429                     {
430                         Objekt o = objekte[layer, i];
431                         k0[i] = dt * o.geschwindigkeit;
432                         l0[i] = dt * o.beschleunigung;
433                     }
434                     for (int i = 0; i < anzahl; i++)
435                     {
436                         Objekt o = objekte[layer, i];
437                         k1[i] = dt * (o.geschwindigkeit + 0.5 * l0[i]);
438                     }
439                     for (int i = 0; i< anzahl;i++)
440                     {
441                         Objekt o = objekte[layer, i];
442                         tmppos[i] = o.position + 0.5 * k0[i];
443                     }
444                     BerechneBeschleunigungAnPos(tmppos, layer);
445                     for (int i = 0; i < anzahl; i++)
446                     {
447                         Objekt o = objekte[layer, i];
448                         l1[i] = dt * o.beschleunigung;
449                     }
450                     for (int i = 0; i < anzahl; i++)
451                     {
```

```
452                         Objekt o = objekte[layer, i];
453                         k2[i] = dt * (o.geschwindigkeit + 0.5 * l1[i]);
454                     }
455                     for (int i = 0; i < anzahl; i++)
456                     {
457                         Objekt o = objekte[layer, i];
458                         tmppos[i] = o.position + 0.5 * k1[i];
459                     }
460                     BerechneBeschleunigungAnPos(tmppos, layer);
461                     for (int i = 0; i < anzahl; i++)
462                     {
463                         Objekt o = objekte[layer, i];
464                         l2[i] = dt * o.beschleunigung;
465                     }
466                     for (int i = 0; i < anzahl; i++)
467                     {
468                         Objekt o = objekte[layer, i];
469                         k3[i] = dt * (o.geschwindigkeit + l2[i]);
470                     }
471                     for (int i = 0; i < anzahl; i++)
472                     {
473                         Objekt o = objekte[layer, i];
474                         tmppos[i] = o.position + k2[i];
475                     }
476                     BerechneBeschleunigungAnPos(tmppos, layer);
477                     for (int i = 0; i < anzahl; i++)
478                     {
479                         Objekt o = objekte[layer, i];
480                         l3[i] = dt * o.beschleunigung;
481                     }
482                     for (int i = 0; i < anzahl; i++)
483                     {
484                         Objekt o = objekte[layer, i];
485                         o.position = o.position + (k0[i] + 2*k1[i] + 2*k2[i] +↵
                          k3[i]) /6.0;
486                         o.geschwindigkeit = o.geschwindigkeit + (l0[i] + 2*l1 ↵
                        [i] + 2*l2[i] + l3[i]) /6.0;
487                     }
488                 }
489                 #endregion
490             }
491         }
492         zeit[layer] = stp.Elapsed.TotalMilliseconds;
493     }
494
495     public void SetLayerRender(int layer, bool render) //Unwichtig, nur UI
496     {
497         layers[layer].render = render;
498         for (int i = 0; i < anzahl; i++)
499         {
500             Objekte[layer, i].SetActive(render);
501         }
502     }
503 }
504
505 [System.Serializable]
```

```
506 public struct Layer //Struct mit den Informationen über ein layer
507 {
508     public Berechnungen.Methode Methode;
509     public int Iterationstiefe;
510     public bool render;
511     public Layer(Berechnungen.Methode m, int i)
512     {
513         Methode = m;
514         Iterationstiefe = i;
515         render = true;
516     }
517 }
518
```