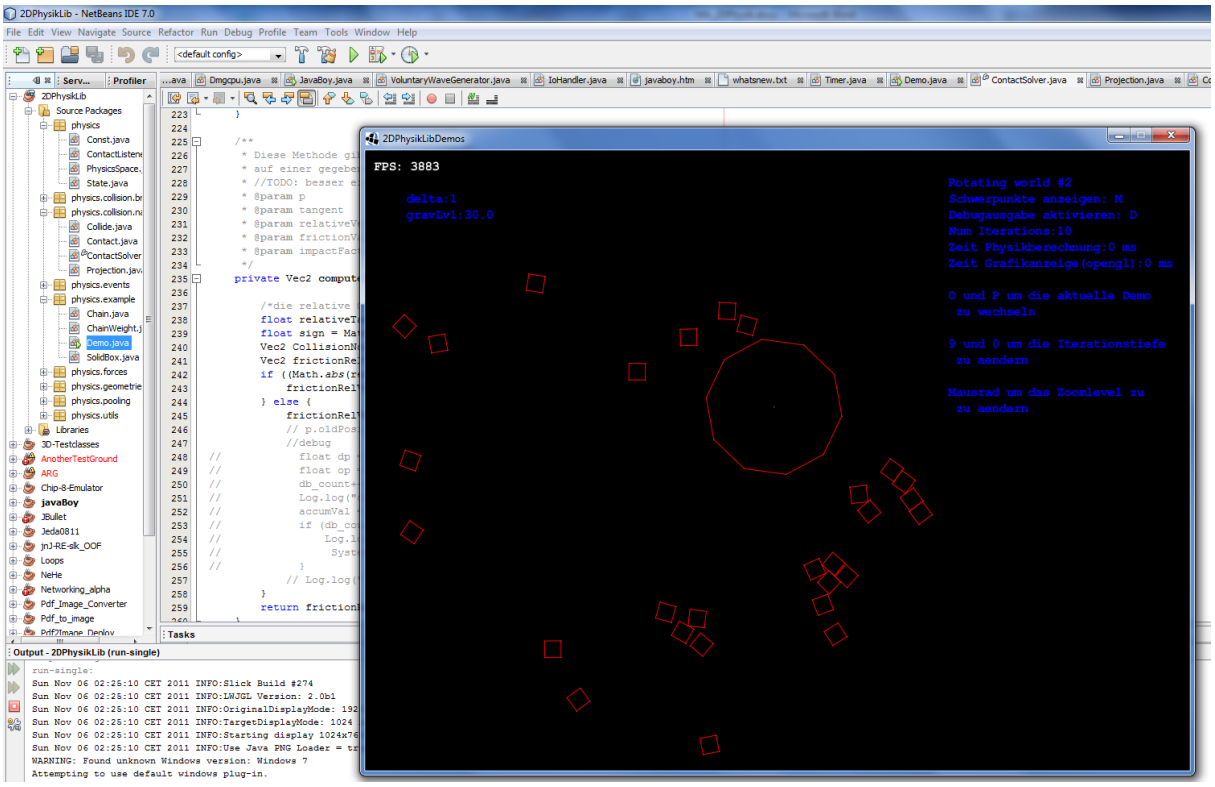
2D **Physik-Engine**

programmiert in Java

Robert Balas



Gymnasium Kirchenfeld, Abteilung MN
Betreuer Stefan Rothe, Bern 2011

1 Inhaltsverzeichnis

2 Einleitung

Eine Physik-Engine ist ein Teil einer Computersoftware, die versucht die physikalischen Gesetze der Natur möglichst genau nachzuahmen. Die Idee von Physik-Engines ist eigentlich schon mehr als 30 Jahre alt. Ursprünglich wurden in Computerspielen nur diejenigen Physikeffekte integriert, die nötig waren. So wurden etwa für Pfeile, die eine Geschossbahn verfolgen, eine entsprechende Gleichung verwendet. Der Nachteil ist, dass nur Pfeile oder ähnliche Geschosse mit dieser Gleichung simuliert werden konnten [1, p. 2]. Für ein einfaches Spiel war dies ausreichend. Doch wenn die Komplexität zunimmt, dann wird es immer schwieriger für alle auftretenden Fälle und Möglichkeiten der Bewegung entsprechende Simulationen zu konstruieren. Eine Physik-Engine stellt eine Lösung dar, die alle möglichen physikalischen Situationen simulieren kann und die immer wieder verwendbar ist. In vielen Computerspielen höchster Qualität, sogenannten AAA-Titeln, wie etwa Crysis, Battlefield 3, Cryostatis und Deus Ex: Human Revolution, gilt es heutzutage als Pflicht aufwändige Physikeffekte in das Spielerlebnis miteinzubeziehen.

Generell unterscheidet man zwischen zwei Arten von Physik-Engines, so genannte high-precision und real-time engines. Beide haben ihre unterschiedlichen Anwendungsgebiete. So werden high-precision engines hauptsächlich in der Forschung und in Animationsfilmen verwendet. Man kann zum Beispiel das Fahrverhalten von Autos auf nassen Strassen simulieren und daraus wertvolle Informationen gewinnen um etwa bessere Reifen zu entwickeln. Diese Programme sind auf hohe Genauigkeit ausgelegt und dementsprechend langsam. Real-time engines werden hingegen in Computerspielen benutzt aufgrund ihrer Echtzeitfähigkeit. Damit in Computerspielen eine konstante Bildrate erzielt werden kann ist es nicht möglich allzu komplexe Berechnungen durchzuführen. Es müssen also einfachere Formeln gefunden werden unter denen jedoch die Genauigkeit bzw. der Realismus der Simulation leidet. Das Ziel der vorliegenden Arbeit ist das Programmieren einer echtzeitfähigen 2D Physik-Engine basierend auf der Verlet-Integration.

Die Motivation dieser Arbeit entsprang dem Computerspiel „World of Goo“. Ziel des Spieles ist es mit möglichst wenigen Bällen eine Konstruktion zu errichten um ein Rohr zu erreichen. Sobald man das Rohr erreicht, werden die überzähligen Bälle vom Rohr eingesogen und geben Punkte. Die Konstruktion muss den physikalischen Gesetzen entsprechend errichtet werden. Ich dachte es wäre ein leichtes ein solches Spiel zu programmieren doch ich stiess schnell an meine Grenzen bei der Simulation der Physik. Mein Interesse war daraufhin geweckt und somit entstand die Begeisterung für diese Arbeit.

3 Physikmodell

Es wird zwischen zwei physikalischen Modellen unterschieden. Es gibt Engines, die vollkommen starre Körper (rigid bodies) und solche, die Anhäufung von Massenteilchen simulieren (mass-aggregate) [1, p. 5]. Beim rigid-body Ansatz werden vollkommen starre Körper simuliert die unter keinen Umständen deformiert werden können. Bei rigid-body Engines ist es nötig zwischen linearen und rotationalen Fällen zu unterscheiden. Also beim Aufprall von zwei Objekten muss aus dem Impuls oder der Kraft, die zwischen den Körpern wirkt, eine neue rotationale und lineare Geschwindigkeit für beide Objekte berechnet werden.

Bei mass-aggregate Engines hingegen ist die rotationale Geschwindigkeit implizit. Das liegt daran, dass man Objekte nicht als starres Ganzes betrachtet sondern aus einem Verbund von vielen Partikeln. Bei einem Zusammenstoss von Objekten werden alle Partikel einzeln betrachtet. Sie unterliegen nur einer Bedingung: Sie dürfen untereinander einen Mindestabstand nicht über- oder unterschreiten. Durch diese Abhängigkeit entsteht bei richtiger Handhabung die rotationale Geschwindigkeit automatisch. Dieses Modell entspricht mehr der Wirklichkeit, weil Gegenstände in der Realität aus einer riesigen Anzahl von Atomen bestehen, die alle alleine agieren. Nur durch zwischenmolekulare Kräfte werden sie zusammengehalten und schränken sich so die Bewegungsfreiheit untereinander ein.

In dieser Arbeit wurde der zweite Ansatz von Thomas Jakobsen [2, p. 8] verfolgt. Simulierte Objekte werden in Particles und Constraints unterteilt, wobei man sich letzteres als eine Art Gestänge oder als unendlich starre Federn vorstellen kann. Von nun an wird der Begriff Feder als Synonym für Constraint verwendet. In den Eckpunkten der Objekte befindet sich jeweils eins dieser besagten Partikel und alle diese Partikel werden durch die Federn verbunden. Zudem wird jedem Partikel eine Masse zugewiesen.

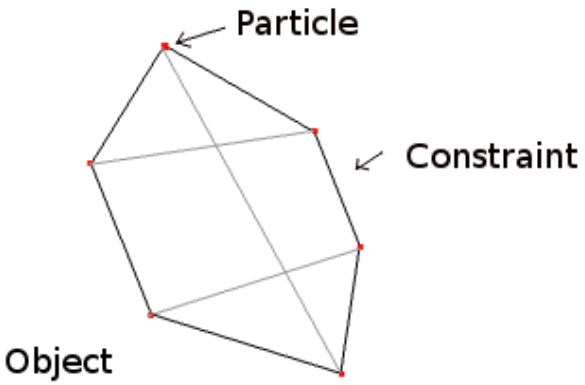


Abb. 1: Aufbau eines Objektes. Die grauen Linien stellen auch Constraints dar. Sie sind nötig, damit das Objekt nicht zusammenbricht.

4 Übersicht über die Physik-Engine

Zuerst wird ein Raum definiert indem man Objekte erstellen kann, die es zu simulieren gilt. Diese Objekte kann man sich als sehr unterschiedliche zweidimensionale geometrische Figuren vorstellen. Die jeweilige Form wird durch den Benutzer festgelegt. Dem Raum kann man jetzt noch eine Gravitationskraft zuweisen. Schliesslich muss der Benutzer nur noch sagen, dass der Raum mit der Simulation beginnen soll. Das ist die grundlegende Bedienung der Physik-Engine.

Der Simulationsteil macht den grössten Teil der Arbeit aus. Er ist in vier Teile gegliedert:

- Der Integrator
- Der Constraint Algorithmus
- Die Kollisionserkennung
- Die Kollisionsauflösung

Diese vier Abschnitte werden der Reihe nach immer wieder durchlaufen. Zuerst werden die neuen Positionen der Objekte aufgrund der einwirkenden Kräfte (z.B. Gravitationskraft) mit dem Integrator berechnet. Der Constraint Algorithmus schaut nun, dass die Objekte ihre Form behalten. Im nächsten Schritt sucht die Kollisionserkennung ob Objekte kollidiert sind. Falls ja, werden gewisse Informationen festgehalten und der Kollisionsauflösung übermittelt. Diese separiert die kollidierten Objekte und verändert ihre Geschwindigkeit, so dass Reibung und Elastizität entsteht. Durch eine schnelle Wiederholung dieser vier Teile, gewöhnlich zwischen 20 bis 100-mal pro Sekunde, entsteht der Eindruck einer fliessenden Simulation.

Die Definition die zu Beginn erwähnt wurde und das Physikmodell im vorherigen Abschnitt ergibt folgende Aufstellung der Klassen:

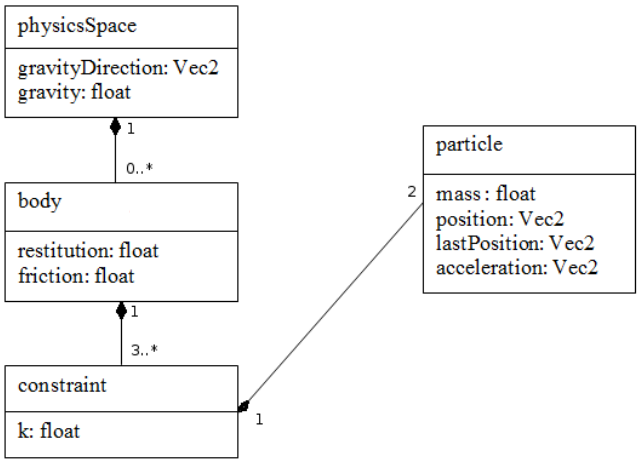


Abb. 2: Klassendiagramm der Physik-Engine

Die Objekte (Klasse body) können jede beliebige Form vom Benutzer zugewiesen bekommen. Einer Bedingung unterliegen jedoch diese zweidimensionalen Figuren: Sie müssen konvex sein. Diese Annahme wurde getroffen um den Kollisionsüberprüfungsteil zu vereinfachen. Diese Klasse hat jeweils zwei willkürlich festlegbare Variablen, den Restitutions- und der Reibungskoeffizient. Ersteres legt fest, wie viel der kinetischen Energie bei jeder Kollision unter den Objekten in Wärmeenergie umgewandelt wird. Der Reibungskoeffizient legt fest wie stark ein Objekt senkrecht zur Kollisionsfläche abgebremst wird.

Die Partikel bilden, wie im Kapitel „Physikmodell“ beschrieben, die Eckpunkte eines jeden Objektes. Jede auf ein Partikel einwirkende Kraft kann mit Newtons zweitem Gesetz in eine Beschleunigung umgewandelt werden. Der Integrator berechnet nicht die neue Position der Klasse body, sondern die neue Position jeder seiner Partikel.

Die Partikel alleine stellen nur ein loses Gebilde dar. Wenn sie von diversen Kräften in alle Richtung gezogen werden, dann verliert das Objekt die Form. Hier kommen die Federn (Klasse constraint) ins Spiel. Sie sorgen dafür, dass die einzelnen Partikel an den richtigen

Stellen stehen. Die Konstante k gibt an, wie stark eine Feder dessen Partikel an die optimalen Stellen ziehen oder stossen soll. Im Kapitel „Constraint Algorithmus“ wird erläutert, wie diese Federn genau funktionieren.

5 Der Integrator

Die Simulation der Objekte bedient sich der klassischen Mechanik. Es gilt also die Differentialgleichungen, die das System beschreiben (Newtonsche Axiome), zu lösen. Wenn die Start- und Randbedingungen bekannt sind ist dies auch möglich. Dabei werden oftmals keine exakten Lösungen ermittelt, da diese aufwändig sind, sondern nur Annäherungen berechnet [3]. Der Integrator führt diese Berechnungen durch. Im Endeffekt errechnet der Integrator aus den gegebenen Kräften die neuen Positionen der Partikel und damit der zu simulierenden Körper.

5.1 Herleitung und Erklärung

Aus der Mechanik ist bekannt, dass die Geschwindigkeit v die Ableitung der Strecke s und das die Beschleunigung a die Ableitung der Geschwindigkeit bzw. die zweite Ableitung der Strecke nach der Zeit ist:

$$v = s'$$
$$a = v' = s''$$

Leitet man nun die Geschwindigkeit einmal auf bzw. die Beschleunigung zweimal auf erhält man folgende bekannten Gleichung aus der klassischen Physik:

$$v = v_0 + a \cdot t$$
$$s = s_0 + v \cdot t + \frac{a}{2} t^2$$

Wenn nun also ein Simulationsobjekt mit der Beschleunigung a beschleunigt wird, dann kann man für jeden Zeitpunkt t dessen Position s berechnen. Jedoch gilt dies nur für eine konstante Beschleunigung a über den Zeitraum t. Bei Kollisionen oder sonstigen Krafteinwirkungen wird die Beschleunigung (und die Geschwindigkeit) verändert und somit kann mit der obigen Formel die Position nicht mehr ermittelt werden. Eine Differenzialgleichung zu finden, die diese Beschleunigungsänderungen beinhält und lösbar ist, ist sehr schwer zu finden wenn nicht gar unmöglich. Dieses Problem wird umgangen indem man sich der numerischen Integration bedient.

5.2 Euler-Integration

Man versucht die neue Position mit Hilfe der alten zu ermitteln. Man zerstückelt dafür einen langen Zeitabschnitt und versucht dann schrittweise die Lösung zu berechnen. Es wird ein Zeitschritt Δt gewählt und in jedem Zeitschritt wird folgendes getan:

$$s_{n+1} = s_n + v \cdot \Delta t + \frac{a}{2} \Delta t^2$$
$$v_{n+1} = v_n + a \cdot \Delta t$$

Wenn man nun die Position des Simulationskörpers bei t = 50 s erfahren will und der willkürlich gewählte Zeitschritt $\Delta t = 1$ ist, müsste man die obige Vorschrift 50-mal durchführen. Geschieht nun eine Geschwindigkeits- oder Beschleunigungsänderung in diesen 50 Sekunden, wird sie in nun im nächsten Berechnungsschritt „bemerkt“ und fliesst somit in das Resultat ein. Diese numerische Integrationsmethode heisst Euler-Integration.

Grafisch dargestellt passiert folgendes:

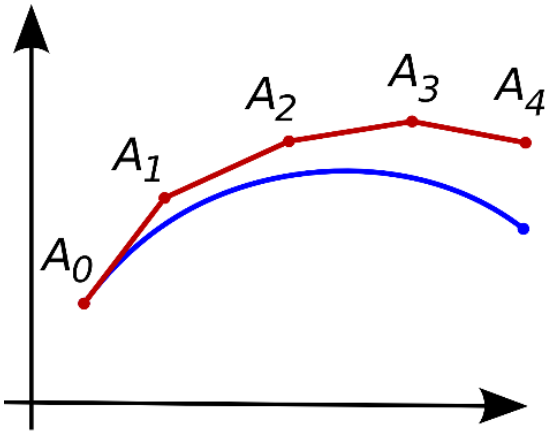


Abb. 3: Approximation (obere Kurve) der exakten Lösung (untere Kurve) durch numerische Integration

Die blaue Kurve stellt die exakte Lösung dar, während dessen die rote Kurve die Approximation mit dem oben beschriebenen Verfahren ist. A_0 ist die Ausgangslage und A_1-A_4 sind die nächsten berechneten Positionen. Es lässt sich gut erkennen, dass A_n jeweils aus A_{n-1}berechnet wird.

Die obige Abbildung lässt erahnen, dass die numerische Lösung nicht ganz exakt ist. Da die Berechnung des nächsten Wertes vom vorigen Wert abhängt, machen sich kleine Fehler später immer stärker bemerkbar. In der Mathematik der numerischen Integration spricht man vom Fehlerterm. Der Fehlerterm hängt von diversen Faktoren ab. Einer davon ist der Zeitschritt Δt. Dieser sollte möglichst klein sein um eine exakteres Resultat zu erzielen, weil dann mehr Änderungen „registriert“ werden. Weiterhin dürfen nicht zu grosse und kurze Beschleunigungen auftreten, da sonst die Position und Geschwindigkeit „explodieren“. Als Beispiel sei folgendes genannt: Man stelle sich vor das der Zeitschritt Δt eine Sekunde beträgt. Das Testobjekt wird gleichmässig beschleunigt durch die Gravitation. Plötzlich verdreifacht sich dieser Wert für eine Zehntelsekunde. Doch da der Zeitschritt eine Sekunde beträgt, wird diese starke Beschleunigung für eine ganze Sekunde angenommen. Das Objekt schiesst über das Ziel bzw. die Lösung hinaus.

5.3 Verlet Integration

In dieser Arbeit wurde jedoch nicht die Euler-Integration verwendet. Stattdessen wurde die so genannte Position Verlet-Integration gewählt. Dieses Integrationsverfahren ist wie die Euler-Version eine Methode um die Bewegungsgleichungen von Newton zu integrieren. Die Formel lautet folgendermassen nach [4]:

$$x_{n+1} = x_n + (x_n - x_{n-1}) \cdot a \cdot \Delta t^2$$

Wie der Name dieser Integrationsmethode andeutet wurde eine Veränderung an der Positionen vorgenommen. Statt eines Geschwindigkeitswerts wird nun eine Variable x_{n-1} eingeführt, welches die letzte Position des Objektes darstellt. Die Geschwindigkeit wird nun implizit angedeutet durch die Differenz zwischen der momentanen Position x_n und der letzten Position x_{n-1}. Dies hat besondere Vorteile bei der Modellierung von starren Federn auf die ich im Kapitel „Constraint Algorithmus“ und „Kollisionsauflösung“ eingehen werde. Die Verlet-Integration besitzt noch zwei weitere Eigenschaften, die sie gegenüber der Euler-Integration überlegen macht. Die Integrationsmethode ist sogenannt symplektisch, das heisst sie konserviert die Energie im System besser. Zudem ist der Fehlerterm ist nicht wie bei der Euler-Integration $O(\Delta t)$ sondern $O(\Delta t^2)$. Diese beiden Eigenschaften führen dazu, dass die Verlet-Methode exaktere Resultate liefert.

5.4 Zeitschritt

Eine wichtige Frage stellt sich noch: Wie gross soll der Zeitschritt Δt sein? In einem ersten Versuch wurde Δt abhängig von der Bildwiederholungsrate gemacht. Normalerweise liegt diese um die 60 Hz, somit ist $\Delta t = 1/60$. Auf den ersten Blick scheint dies durchaus vernünftig. Wenn der Computer unter Last steht (insbesondere durch Computerspiele), dann kommt es oftmals zu Einbrüchen bei der Bildwiederholungsrate. In diesem Fall wird nun auch der Zeitschritt Δt angepasst. In anderen Worten gefasst: Wenn der Computer mit den Berechnungen nicht mithalten kann, dann macht die Simulation grössere Schritte so dass sie immer noch gleich schnell zu laufen scheint. Doch hier liegt das Problem vom diesem Ansatz. Bei drastischen Einbrüchen der Bildwiederholungsrate wird Δt unkontrollierbar gross. Dadurch integriert der Integrator Beschleunigungen über einen zu grossen Zeitraum. Es tritt das Problem der „explodierenden" Simulation auf wie schon im vorherigen Abschnitt erklärt wurde.

Im zweiten Versuch wurde $\Delta t = const$ festgelegt. $const$ ist dabei ein willkürlich gewählter konstanter Wert. Somit ist Δt kein Fehlerfaktor mehr. Um auch noch den Effekt der gleichmässig laufenden Simulation des ersten Versuches zu erhalten werden mehr Iterationen durchgeführt wenn die Framerate sinkt, wie es Glenn Fiedler vorgeschlagen hat [4].

5.5 Pseudocode

Die Integrationsmethode für ein Partikel könnte folgendermassen aussehen:

```
integriere(float deltaT)
{
 tmp = position; //Variablen sind Vektoren!
 position = position +(position-letztePosition)*beschleunigung*(deltaT*deltaT);
 letztePosition = tmp;
}
```

6 Constraint Algorithmus

Die Eckpunkte der Objekte in der Physik-Engine werden mit Partikeln beschrieben. In jedem Iterationsschritt der Physik-Engine können sich die Distanzen zwischen den einzelnen Partikeln ändern, wie etwa wenn ein Partikel bereits den Boden berührt währenddessen das andere sich immer noch im freien Fall befindet. Diese Distanzänderung zwischen den Partikeln hat eine Formänderung des Physikobjektes zur Folge (Abb. 4).

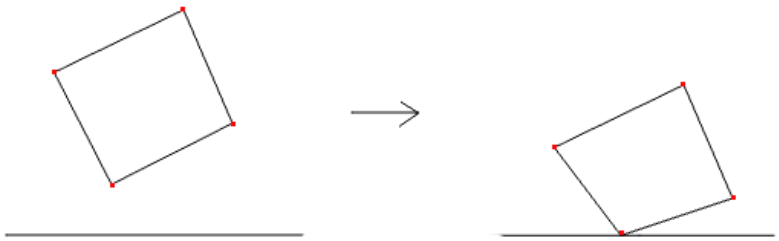


Abb. 4: Die fallende Kiste deformiert sich

In einem ersten Anlauf wurde zwischen den Partikeln jeweils eine Feder eingespannt, welche die Partikel auseinanderstossen bzw. zusammenziehen sollte. Das Hookesche Gesetzt besagt, dass die ausgeübte Kraft der Feder proportional zu der Belastung ist:

$$F = -k \cdot s$$

Man könnte nun die Kraft F durch die jeweilige Masse m der Partikel teilen um eine Beschleunigung a zu erhalten. Die Partikel müssten jetzt noch mit dieser Beschleunigung korrigiert werden. Doch dieser Ansatz hat ein fundamentaler Fehler.
Um eine Stangen zu simulieren benötigt es eine starre Feder. Damit eine die Feder so starr wie möglich ist, muss k, die Federkonstante, möglichst gross sein. Lässt man nun k gegen unendlich streben wird auch die ausgeübte Kraft unendlich. Die Simulation bzw. der Computer kann jedoch nicht mit solchen Werten umgehen, folglich muss also eine andere Lösung gesucht werden.

Hier kommen Constraint Algorithmen ins Spiel. Mit denen ist es möglich solche starre Federn zu simulieren. Allgemein hat ein Constraint Alogrithmus hat die Aufgabe die Bewegung zweier gegebener Punkten in einer bestimmten Weise einzuschränken z.B. sie dürfen sich nur bis n Meter nähren. Für die Physik-Engine wird eine Funktion benötigt welche die Partikel genau auf eine bestimmte Distanz hält damit der gewünschte Effekt einer Stange auftritt. Mathematisch ausgedrückt:

$$\left|\overrightarrow{x1} - \overrightarrow{x2}\right| - d = 0$$

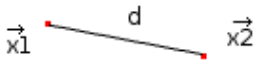


Abb. 5: x1 und x2 sind die Positionen der jeweiligen Endpunkte, d ist die Distanz zwischen ihnen

Doch bevor der Lösungsansatz präsentiert wird, gilt es noch ein Problem zu lösen. Wenn man nämlich die Länge einer Feder korrigiert, dann ist es durchaus möglich, dass sich ein benachbarte Feder, welcher mit dem gleichen Partikel verbunden ist, sich wiederum verformt.

Hier gibt es zwei Lösungsansätze: eine so genannte iterative und eine globale Lösung. Bei der globalen Lösung werden alle Federn in die Berechnung einbezogen und es wird eine Lösung ermittelt welche die Partikel so positioniert, dass alle Federlängen als erfüllt gelten. Die iterative Lösung hingegen betrachtet jede Feder einzeln und setzt sie auf die richtige Federlänge. Dieser Lösungsvorgang wird mehrmals wiederholt wobei sich alle Constraints schrittweise der tatsächlichen Lösung annähern. Wenn die Genauigkeit genug hoch ist, dann kann der Vorgang abgebrochen werden[1].

In dieser Arbeit wurde der iterative Lösungsansatz von Thomas Jakobsen gewählt [2]. Der vorgeschlagene Pseudo-Code sieht folgendermassen aus:

```
delta = x2-x1
deltalength = sqrt(delta*delta)
diff = (deltalength-d)/deltalength
x1 += delta * 0.5 * diff;
x2 -= delta * 0.5 * diff;
```

Grundsätzlich berechnet diese Rechenvorschrift die Distanz zwischen den beiden Partikeln und Vergleicht sie mit der gewünschten Grösse (In diesem Falle d genannt). Die Partikel werden dann jeweils um die Hälfte der Differenz zwischen der gewünschten und der tatsächlichen Grösse in die entsprechende Richtung verschoben. Es wird jedoch nicht auf einen Schritt um die ganze Differenz verschoben sondern je näher man sich das Partikel an der gewünschten Position befindet, desto geringer fällt die Korrektur aus. Damit wird verhindert, dass die Korrektur nicht das Partikel über das Ziel hinausschiebt. Hier zeigt sich ein Vorteil der Verlet-Integration. Bei der Korrektur werden die Positionen der Partikel jeweils geändert. Dadurch wird implizit die Geschwindigkeit verändert, da diese die Differenz zwischen der aktuellen und der letzten Position ist. So nimmt die Geschwindigkeit der Partikel ab und folglich verhält sich das System ruhiger. Nichtsdestotrotz bleiben alle

[1] In der Mathematik ist dieser Vorgang unter dem Namen „Splitting-Verfahren“ bekannt

physikalisch relevanten Faktoren unverändert wie das lineare und angulare Drehmoment [5, p. 4].

Doch dieser Algorithmus genügt den Anforderungen dieser Physik-Engine noch nicht. Es wurde noch einige kleinere Modifikationen an diesem Algorithmus unternommen. In den beiden letzten Zeilen werden die Partikeln an die neue Position verschoben mit jeweils dem Faktor 0,5. Dies gilt jedoch nur wenn die Massen der beiden Partikeln gleich gross sind. Abbildung sechs zeigt ein extremes Beispiel.

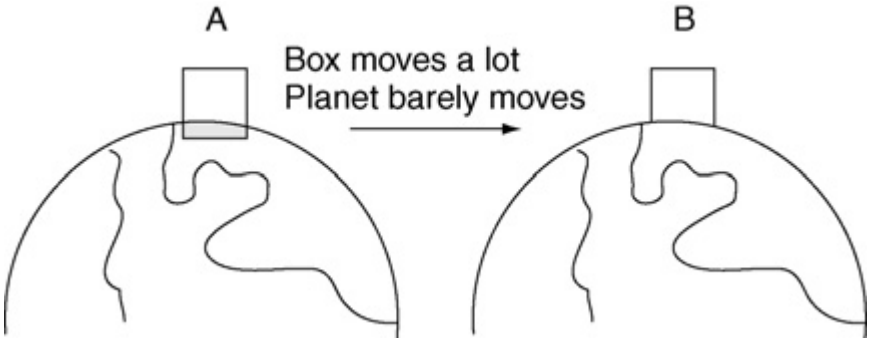


Abb. 6: Planet und Kiste

Analog zu diesem Bild sollten die beiden Objekte umgekehrt proportional zu ihren Massen verschoben werden. Ein Objekt mit grosser Masse verschiebt sich beinahe nicht, während dessen ein Objekt mit kleiner Masse sich viel bewegt [1, p. 113]. Hergeleitet wird der neue Verschiebungsfaktor wie folgt:

$$s_1 + s_2 = d$$

s_1 und s_2 sind die Verschiebungen, d die Gesammtverschiebung

Die Verschiebung von der jeweiligen Masse abhängig machen:

$$m_1 \cdot s_1 = m_2 \cdot s_2$$

Einsetzen:

$$m_1 \cdot (d - s_2) = m_2 \cdot s_2$$
$$m_1 \cdot d - m_1 \cdot s_2 = m_2 \cdot s_2$$
$$s_2 \cdot (m_1 - m_2) = m_1 \cdot d$$

Das ergibt dann für s_1 und s_2:

$$s_1 = \frac{m_2}{m_1 + m_2} d$$
$$s_2 = \frac{m_1}{m_1 + m_2} d$$

Die 0.5 stellt sich dabei als den Spezialfall für $m_1 = m_2$heraus:

$$s_1 = \frac{1}{1 + 1} d$$

Weiterhin wurde noch der Steiffheitsparameter k eingeführt, welcher beeinflusst, wie stark bzw. wie schnell die Partikel ihre original Distanz einnehmen. Hierzu werden einfach die Korrektionswerte mit dem Faktor k, der zwischen null und eins liegt, multipliziert. Doch es stellt sich heraus, dass die Auswirkung von k bei mehreren Berechnungsdurchgängen nicht linear ist. In anderen Worten: Für jede weitere Iteration wird der Effekt von k schwächer. In Formeln gefasst ist der verbleibende Fehler nach n Iterationen:

$$\Delta s \cdot (1 - k)^n$$

M. Müller et al. [5, p. 5] schlagen nun vor die Korrektionswerte nicht direkt mit k zu multiplizieren. Stattdessen sollte man folgende Formel verwenden, um eine lineare Abhängigkeit zu erhalten:

$$k' = 1 - (1 - k)^{1/n}$$

Zur Überprüfung:

$$\begin{aligned} & \Delta s \cdot (1 - k')^n \\ = \Delta s \cdot & (1 - \left(1 - (1 - k)^{1/n}\right))^n \\ = \ \Delta s \cdot & ((1 - k)^{1/n})^n \\ = \Delta s \cdot & (1 - k) \end{aligned}$$

Die Variable n steht nicht mehr im Exponent bzw. ist verschwunden somit ist die Korrektion jetzt linear zu k.

Der neue Pseudocode mit allen Veränderungen sieht so aus:

```
korrigiereFedern(float k)
{
 delta = x2-x1;//x1 und x2 sind die Positionen der Partikel bzw. Endpunkte der Federn
 momentaneLaenge = sqrt(delta*delta);
 diff = (momentaneLaenge-originaleLaenge)/momentaneLaenge;

 kLinear = 1-(1-k)^1/n
 x1 += delta * m2/(m1+m2) * diff * kLinear;
 x2 -= delta * m1/(m1+m2) * diff * kLinear;
}
```

7 Kollisionserkennung

Die Aufgabe der Kollisionserkennung ist wie der Name schon sagt das Erkennen von Kollisionen und das Herausfiltern von relevanten Informationen für die Kollisionsauflösung. Die Kollisionserkennung wird oftmals in zwei Teile unterteil, der Broad- und Narrowphase.

7.1 Broadphase

Dieser Codeteil schätzt ab, ob sich gewisse Objekte berühren oder nicht. Es wird differenziert zwischen zwei Möglichkeiten:

- Die Objekte berühren sich sicher nicht
- Die Objekte könnten sich berühren

Wenn der zweite Fall eintrifft, dann wird wie zuvor bereits erwähnt, der Narrowphasealgorithmus zu Rate gezogen um endgültige Aussagen zu machen.

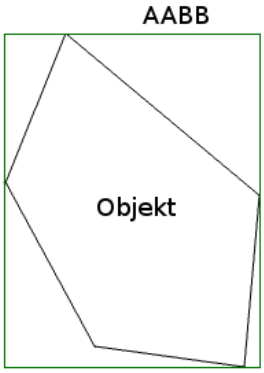


Abb. 7: ein AABB an ein Objekt angeglichen

Es gibt viele Broadphasealgorithmen, die diverse Stärken und Schwächen aufweisen. Bekannte Algorithmen sind sweep and prune, Octree, Dynamic Tree etc.

Einer der einfachsten Broadphasealgorithmen stellen die simplen axis aligned bounding boxes (AABB) dar. Das Grundprinzip ist, dass man über alle Physikobjekte einen rechteckigen Rahmen zieht, der parallel zur x-Achse ausgerichtet ist (Abb. 7). Durch die Parallelität aller Boxen mit der gleichen Achse, kann man mit einfachen Koordinatenvergleichen feststellen ob sich die Boxen berühren. In dieser Arbeit wurde auf komplizierte Broadphasealgorithmen verzichtet und nur die Idee der AABBs implementiert.

7.2 Narrowphase

Der Algorithmus der in diesem Bereich werkelt, macht die eigentliche Kollisionüberprüfung. Narrowphasealgorithmen können besonders im 3D Bereich sehr komplex sein. Dieser

Themenbereich kann bereits mehrere Bücher füllen, da er vor allem wegen performancetechnischen Gründen sehr interessant ist. So sind viele gute und schnelle Algorithmen wie etwa V-Clip und deren Varianten patentiert [1, p. 293].

Grundsätzlich muss eine Implementation genau vier Informationen liefern können:Ob überhaupt eine Kollision stattfindet und wenn ja dann muss der genaue Ort, die Eindringtiefe und die Kollisionsnormale ausgelesen werden können.

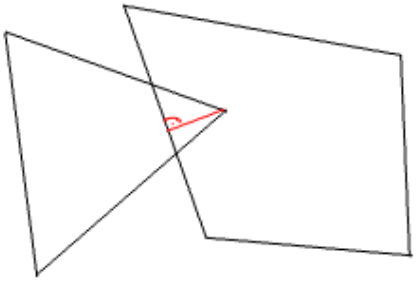


Abb. 8: Zwei Objekte kollidieren: die Richtung der roten Gerade ist die Kollisionsnormale, dessen Länge die Eindringtiefe

In dieser Arbeit wurde ein Algorithmus auf Basis des separating axis theorem, auch SAT genannt, implementiert. Dieses besagt, dass „bei zwei konvexen Polygonen genau dann eine Achse existiert, auf welche sich deren Projektionen nicht schneiden, wenn sich die Polygone nicht überlappen.“ [6] (Abb. 9).

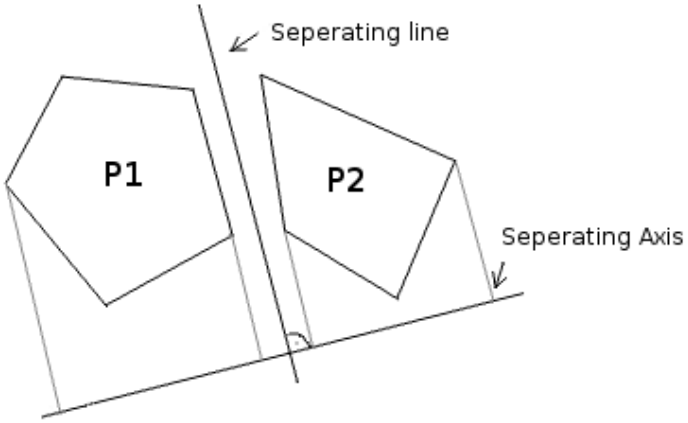


Abb. 9: Die Projektionen (grau) der Polygonen P1 und P2 überschneiden sich nicht auf der Trennungsachse

Auf dem ersten Blick sieht es aus, als ob es unendlich viele Trennungsachsen zu prüfen gibt. Doch glücklicherweise kommen nur genau die Achsen in Frage, die senkrecht zu den Polygonseiten stehen [6].

Um nun zu prüfen ob sich zwei gegebene Polygone P1 und P2 schneiden, muss auf jede Trennungsachse P1 und P2 projiziert werden. In einem nächsten Schritt wird überprüft, ob sich die Projektionen überlappen. Dieser Vorgang wird so lange wiederholt, bis entweder eine Achse gefunden wird deren Projektionen sich nicht überlappen (d.h. die Polygone

schneiden sich nicht) oder es keine Achsen mehr zu überprüfen gibt (die Polygone schneiden sich).

So könnte der Algorithmus aussehen (angelehnt an [6]):

```
testeKollision(Polygon p1, Polygon p2)
{
 for (1 to p1.anzahlKanten)
 {
   trennachse = momentaneKante.berechneSenkrechte();

   projektion1 = p1.projiziereAuf(trennachse);
   projektion2 = p2.projiziereAuf(trennachse);

    If(projektion1.schneidetSichMit(projektion2))
    {
     return true; //Es ist nun klar, dass keine Überlappung stattfinden kann
    }

 }
 for (1 to p2.anzahlKanten)
 {//das selbe für p2

    trennachse = momentaneKante.berechneSenkrechte();

    projektion1 = p1.projiziereAuf(trennachse);
    projektion2 = p2.projiziereAuf(trennachse);

    If(projektion1.schneidetSichMit(projektion2))
    {
     return true; //Es ist nun klar, dass keine Überlappung stattfinden kann
    }

 }
 return false;//Es wurden alle Achsen in Betracht gezogen->die Polygone überlappen sich
}
```

In dem oben abgebildeten Algorithmus sind alle Kanten als zweidimensionale Vektoren betrachtet worden. Damit gestaltet sich die Berechnung einer Trennungsachse aus einer Kante sehr einfach. Da die Trennungsachse genau orthogonal auf die Kante zeigt, lässt sich diese durch einen einfachen Komponententausch erzeugen:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}$$

Eine grössere Hürde ist die Projektionsmethode. Die Abbildung neun lässt erkennen, dass die äussersten Ecken des Polygons entscheidend sind für die Projektion auf eine Trennungsachse. Ein einfacher Weg ist es, alle Ecken auf die Achse zu projizieren und dann nur die geeigneten Punkte auszuwählen (diejenigen die am weitesten auseinander liegen). Die Projektion für sich geschieht durch Vektorprojektion:

$$c = \vec{a} \cdot \frac{\vec{b}}{|b|}$$

Der Vektor $\vec{a}$ ist dabei die Position des Eckpunktes und $\frac{\vec{b}}{|b|}$ der Einheitsvektor der Normale.

```
projiziereAuf(Vektor trennachse){
  int maximum = //auf einen Wert initialisieren, der zwischen dem erwarteten maximalen
  int minimum = //und minimalen Wert liegt
                //denkbar wäre etwa das erste berechnete c

 for(1 to anzahlEckpunkte){
  int c = eckpunkt * (kante /kantenlänge);
  max = max(maximum, c);//die Funktion gibt den grösseren Wert zurück
  min = min(minimum, c);//analog
```

```
    }
}
```

Der letzte Schritt, das Überprüfen der beiden Projektionen mit der Methode schneidetSichMit(), gestaltet sich als ziemlich einfach. Für jede Projektion wurde ein maximaler und ein minimaler Wert berechnet. Durch einfaches Vergleichen dieser Zahlen kann man nun bestimmen ob sich die Projektionen überlappen. Falls sich die Projektionen schneiden kann noch die Eindringtiefe d berechnet werden.

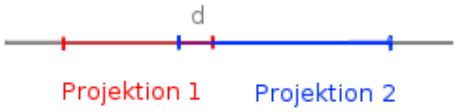


Abb. 10: Zwei Projektionen die sich überlappen

Jetzt müssen nur noch alle Kollisionsinformationen festgehalten werden. Sobald eine Achse gefunden wurde, auf dieser sich die Projektionen nicht überlappen, kann der Berechnungsvorgang abgebrochen werden und folglich müssen keine Kollisionsinformationen festgehalten werden. Geschieht dies jedoch nicht, dann ist die zu festzuhaltende Eindringtiefe die kleinste aller berechneten Eindringtiefen und die Kollisionsnormale die Achse auf der die kleinste Eindringtiefe gefunden wurde. Als Ort wird die Kante des durchdrungenen Objektes und das Partikel, das die besagte Kante durchdringt angegeben.

8 Kollisionsauflösung

8.1 Übersicht

Nachdem alle Kollisionen und deren Informationen gefunden wurden, werden sie vom nächsten Teil, der Kollisionsauflösung, bearbeitet. Deren Aufgabe ist es, die Objekte wieder zu separieren, so dass sie sich nicht mehr berühren. Doch das genügt noch nicht: die Objekte müssen sich drehen, Reibung erfahren und auch abprallen können.

Es gibt diverse Methoden wie man Kollisionen handhabt. Eine Möglichkeit wäre es eine Feder zwischen den beiden Objekten einzuführen um sie auseinander zu halten. Doch es ist oftmals schwierig eine passende Federkonstante zu wählen, so dass die Objekte nicht ineinander versinken oder dass das System unstabil wird und die Objekte zu zittern beginnen. Man könnte auch, nachdem eine Kollision festgestellt wurde, die Zeit „zurückdrehen“ zum Zeitpunkt der Kollision und von hier an weiterfahren. Jedoch ist dieser Ansatz für Echtzeitsimulationen nicht geeignet, da bei vielen Kollisionen die Simulation zu stark abgebremst würde. Bei diesem Problem zeigt sich einer der Stärken der Verlet–Integration. Die Punkte, die Ursache der Kollision sind, werden einfach senkrecht nach aussen projiziert. Durch das Bewegen der aktuellen Position der Partikel, verändert sich auch die Richtung und die Distanz der eben genannten Position zu der letzten Position. Dies hat eine Änderung der Länge und Richtung des Geschwindigkeitsvektors des Partikels zur Folge. Es ist somit nicht nötig die Partikel manuell abzubremsen, da sie es von alleine tun werden, ähnlich wie im Kapitel „Constraint Algorithmus“ [2].

Die Kollisionsinformationen aus dem Kollisionsprüfungsteil beinhalten die Kante (Constraint) und das Partikel, die in der Kollision involviert sind. Nun gilt es sie auseinander zu schieben, so dass sie sich nur noch knapp berühren. Es wird dabei zwischen zwei Kollisionsfällen unterschieden.

8.2 Kollision Typ 1

Im ersten Fall durchdringt die Ecke den Boden(Abb. 11).

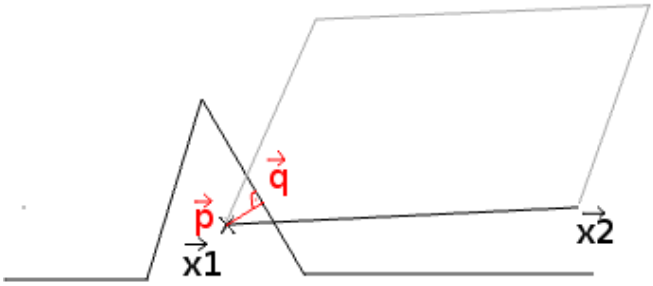


Abb. 11: Ecke durchdringt den Boden

Dieser Kollisionsfall ist einfach zu lösen. Es genügt den Punkt $\overrightarrow{x_1}$ in Richtung $\vec{q} - \vec{p}$ zu verschieben, bis er den vorherstehenden Boden nicht mehr berührt.

8.3 Kollision Typ 2

Im weitaus schwierigeren Fall, durchdringt der Boden die Kante des Objekts (Abb. 12).

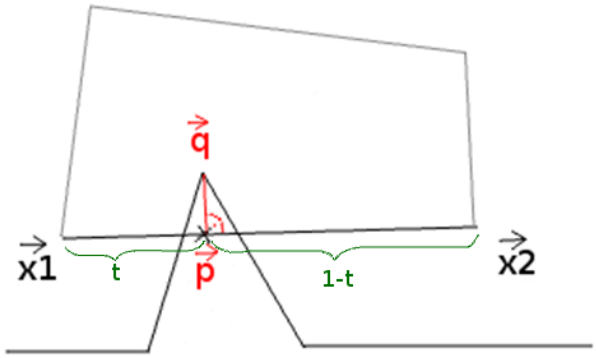


Abb. 12: Eine Kante wird von dem Boden durchdrungen

Das Ziel ist, dass die beiden Enden der Formel so verschoben werden, dass $\vec{p}$ auf $\vec{q}$ zu liegen kommt.Die folgende Herleitung basiert auf der Arbeit *Advanced Character Physics* [2].

Da p sich auf der Kante befindet, kann der Punkt als Linearkombination der beiden Enden $\overrightarrow{x_1}$ und $\overrightarrow{x_2}$ ausgedrückt werden.

$$\vec{p} = c_1 \cdot \overrightarrow{x_1} + c_2 \cdot \overrightarrow{x_2} \; mit \; c_1 + c_2 = 1$$
$$c_1 = 1 - t \; und \; c_2 = t$$

Die beiden Punkte werden nun jeweils mit c_1 bzw. c_2 gewichtet. Das heisst während sich $\overrightarrow{x_1}$ um $1 - t$ bewegt, bewegt sich $\overrightarrow{x_2}$ um t. In anderen Worten: $\overrightarrow{x_1}$, welches sich näher bei $\vec{p}$ befindet wird stärker bewegt als $\overrightarrow{x_2}$, das sich viel weiter entfernt befindet. Die Verschiebung ist ein Vielfaches der Gewichtung in Richtung der Normale. Deshalb wird eine Unbekannte λ eingeführt. In Formeln gefasst:

$$\vec{n} = \vec{q} - \vec{p}$$
$$\overrightarrow{x_1'} = \overrightarrow{x_1} + (1 - t) \cdot \lambda \cdot \vec{n}$$
$$\overrightarrow{x_2'} = \overrightarrow{x_2} + t \cdot \lambda \cdot \vec{n}$$

Die Beziehung von $\overrightarrow{p'}$ zu $\overrightarrow{x_1'}$ und $\overrightarrow{x_2'}$ ist analog zu $\vec{p}$ mit $\overrightarrow{x_1}$, und $\overrightarrow{x_2}$:

$$\overrightarrow{p'} = c_1 \cdot \overrightarrow{x_1'} + c_2 \cdot \overrightarrow{x_2'} \; mit \; c_1 + c_2 = 1$$

λ muss jetzt so gewählt werden, dass $p' = q$ ist. Partikel werden nur in Richtung der Normale verschoben. Folglich gilt es diese Gleichung zu lösen:

$$\overrightarrow{p'} \cdot \vec{n} = \vec{q} \cdot \vec{n}$$

Für $\vec{q}$ wird $\overrightarrow{p'}$ eingesetzt:

$$\overrightarrow{p'} \cdot \vec{n} = \left((1-t) \cdot \overrightarrow{x_1'} + t \cdot \overrightarrow{x_2'}\right) \cdot \vec{n}$$

$\overrightarrow{x_1'}$ und $\overrightarrow{x_2'}$ werden mit $\overrightarrow{x_1}$, und $\overrightarrow{x_2}$ beschrieben:

$$\begin{aligned}\overrightarrow{p'} * \vec{n} &= \left((1-t) \cdot (\overrightarrow{x_1} + (1-t) \cdot \lambda \cdot \vec{n}) + t \cdot (\overrightarrow{x_2} + t \cdot \lambda \cdot \vec{n})\right) \cdot \vec{n} \\ &= \left((1-t) \cdot \overrightarrow{x_1} + t \cdot \overrightarrow{x_2}\right) \cdot \vec{n} + \lambda \cdot ((1-t)^2 + t^2) \cdot \vec{n}^2 \\ &= p \cdot \vec{n} + \lambda \cdot ((1-t)^2 + t^2) \cdot \vec{n}^2\end{aligned}$$

Für $\overrightarrow{p'} \cdot \vec{n}$ nun $\vec{q} \cdot \vec{n}$ einsetzen und nach λ auflösen. $\vec{n}$ kürzt sich raus das ergibt:

$$\lambda = \frac{1}{(1-t)^2 + t^2}$$

λ wird nun mit den zu Beginn aufgestellten Gleichung zur Berechnung von $\overrightarrow{x_1'}$ und $\overrightarrow{x_2'}$ eingesetzt:

$$\overrightarrow{x_1'} = \overrightarrow{x_1} + (1-t) \cdot \frac{1}{(1-t)^2 + t^2} \cdot \vec{n}$$

$$\overrightarrow{x_2'} = \overrightarrow{x_2} + t \cdot \frac{1}{(1-t)^2 + t^2} \cdot \vec{n}$$

Interessant ist, dass die Rotationseffekte automatisch entstehen, wenn die Objekte separiert werden, was ein Effekt der Verlet-Integration ist [4] .

8.4 Reibung

Die Umsetzung von Reibung gestaltet sich als ziemlich schwierig, besonders bei dynamischen Objekten. Aus der klassischen Physik ist die Formel $F_r = F_n \cdot \mu$, die Coulombsche Reibung bekannt. F_r ist die wirkende Reibungskraft parallel zur Reibungsfläche, F_n die Normalkraft und μ der Reibungskoeffizient der von beiden beteiligten Materialien abhängt. Doch dieses Modell funktioniert, wenn eines der Objekte statisch ist, wie etwa einem Holzklotz und dem Fussboden. Es ist jedoch möglich diese Formel auf zwei dynamische Objekte anzuwenden, indem man eines als unbeweglich betrachtet. Das ist möglich indem man die relative Kraft der beiden beteiligten Objekte errechnet:

$$F_{relativ} = F_1 - F_2$$

Analog für die relative Geschwindigkeit:

$$V_{relativ} = V_1 - V_2$$

Die Physik-Engine kann jedoch nicht gut mit Kräften umgehen, da deren Auswirkung nicht linear auf die Geschwindigkeit ist. Eine ein wenig schlecht dosierte Kraft kann das ganze System stören. Dieses Kräftemodell kann man vereinfachen auf ein Modell, das mit Geschwindigkeiten funktioniert. Bekannt ist $m \cdot v = p$, die Gleichung für Impulse. Weiterhin weiss man, dass ein Impuls eine ausgeübte Kraft über einen kurzen Zeitraum ist:

$$\Delta p = F \cdot \Delta t$$

Fügt man nun diese beiden Formeln in die Coulombsche Reibungsformeln ein und kürzt, dann erhält man:

$$V_r = V_n \cdot \mu$$

Nun muss V_r und somit V_n berechnet werden. Zur Veranschaulichung siehe Abbildung 13.

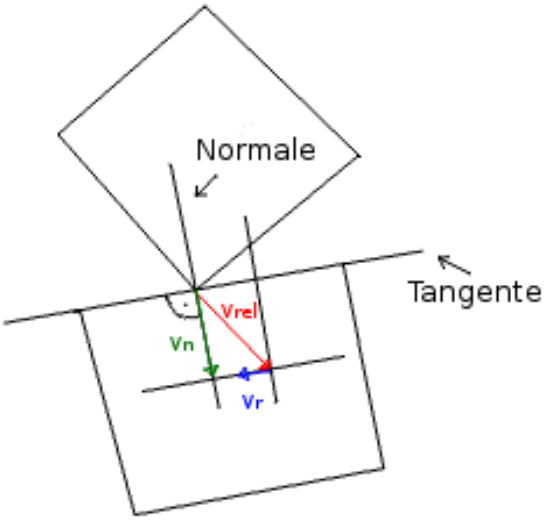


Abb. 13: Zwei Objekte prallen aufeinander. Vn und Vr gilt es zu berechnen.

Es gilt also:

$$\overrightarrow{V_n} = (\overrightarrow{V_{rel}} \cdot \frac{\vec{n}}{|n|}) \cdot \frac{\vec{n}}{|n|}$$

Und mit $V_r = V_n \cdot \mu$ ergibt dies:

$$\overrightarrow{V_r} = (\overrightarrow{V_n} - \overrightarrow{V_{rel}}) \cdot |\overrightarrow{V_n}|$$

Die neue relative Geschwindigkeit sollte zwischen den beiden Objekten sollte nun

$$\overrightarrow{V_{rel\,neu}} = \overrightarrow{V_{rel}} + \overrightarrow{V_r}$$

sein, damit ein Reibungseffekt erzielt wird.
Es wird angenommen, dass beide Geschwindigkeiten V_1und V_2 in verschiedene Richtungen korrigiert werden müssen:

$$\overrightarrow{V_1'} = \overrightarrow{V_1} + \vec{k}$$
$$\overrightarrow{V_2'} = \overrightarrow{V_2} - \vec{k}$$

Nun muss der Korrekturvektor $\vec{k}$ berechnet werden:

$$\overrightarrow{V_1'} - \overrightarrow{V_2'} = \overrightarrow{V_{rel\,neu}}$$
$$\overrightarrow{V_1} + \vec{k} - (\overrightarrow{V_2} - \vec{k}) = (V_1 - V_2) + \overrightarrow{V_r}$$
$$2\vec{k} + V_1 - V_2 = (V_1 - V_2) + \overrightarrow{V_r}$$
$$\vec{k} = \frac{\overrightarrow{V_r}}{2}$$

Somit sind die neuen Geschwindigkeiten:

$$\overrightarrow{V_1'} = \overrightarrow{V_1} + \frac{\overrightarrow{V_r}}{2}$$
$$\overrightarrow{V_2'} = \overrightarrow{V_2} - \frac{\overrightarrow{V_r}}{2}$$

8.5 Impuls

Die Berechnung der resultierenden Impulse nach einer Kollision und somit die Verwendung des Restitutionskoeffizienten hat nicht mehr den Weg in die Engine gefunden. Es wurden aber diverse Ansätze entwickelt. Der erste Ansatz basierte auf dem Prinzip der Energieerhaltung:

$$E_{kin} = \frac{m}{2} \cdot v^2$$
$$E_{rot} = \frac{1}{2} I \omega^2$$
$$E_{tot} = E_{kin} + E_{rot}$$

Jetzt musste die Energie für ein Objekt nach der Kollision

$$E_{tot\,neu} = E_{tot} \cdot r$$

sein, wobei r der Restitutionskoeffizient ist. Das grösste Problem ist jedoch, dass es zu viele unbekannte Variablen hat und somit das Gleichungssystem nicht lösbar ist.

Der zweite Ansatz basiert auf der Reflexion der Geschwindigkeit in Richtung der Kollisionsnormale.

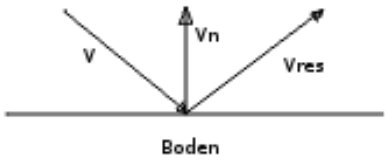


Abb. 14: Reflexion der Geschwindigkeit

Nach Abbildung 14 wäre die resultierende Geschwindigkeit v_{res}:

$$v_{res} = v + (v_n \cdot r)$$

Doch diese Formel funktioniert nur bei einem einzelnen Partikel, weil die geometrischen Begebenheiten eines Objektes nicht miteinbezogen werden. Wenn das Partikel eines Objektes, welches mit dem anderen Objekt kollidiert ist, mit dieser Formel reflektiert wird, dann stimmt die neu zugewiesene Geschwindigkeit nur für das Partikel selber, jedoch nicht für das ganze Objekt. Ziel ist es also, dem Partikel eine solche Geschwindigkeit zuzuweisen, dass das ganze Objekt die korrekte lineare und angulare Geschwindigkeit erhält. Es wurde angenommen, dass das tatsächliche v_{res} für das Partikel ein Vielfaches des Berechneten ist. Doch Trial and Error Tests haben ergeben, dass der Faktor je nach Form und Kollisionswinkel des Objektes unterschiedlich ist.

9 Performance

In zeitkritischen Programmen ist Performance bzw. Geschwindigkeit einer der wichtigsten Faktoren. Die Ausführungsgeschwindigkeit eines Programms lässt sich durch bessere Hardware oder durch geschicktes Programmieren verbessern. Diese hier beschriebene Physik-Engine legt grossen Wert auf Echtzeitfähigkeit, so dass sie in Computerspielen eingesetzt werden kann. Das heisst die Engine muss genug schnell sein und somit die Berechnungen sparsam halten, damit einerseits die Physikberechnungen in Echtzeit erledigt werden können und andererseits genug Ressourcen (Rechenleistung) frei lassen, damit dass Programm, welches die Engine nutzt, genug Rechenleistung zur Verfügung hat für die eigenen Berechnungen. Im Endeffekt muss die Physik-Engine also so sparsam wie möglich arbeiten. Hier kommt die Optimierung ins Spiel, das Nachbearbeiten des meist schon lauffähigen Programmcodes um eine bessere Performance zu erzielen. Einige Beispiele:

9.1 Optimierung der Methodenaufrufe

Die Vektorklasse der Physik-Engine hatte viele Aufrufe zu primitiven Datentypen, obwohl diese direkt zugänglich waren. So wurden die Komponenten x und y meist mit getX() und getY() aufgerufen. Diese Optimierung war insofern wichtig, da die Klasse einer der höchsten Methodenaufrufe verzeichnet. Bei jedem Methodenaufruf wird der momentane

Programmzähler auf den Runtimestack gelegt und der Programmzähler auf die Methodenaddresse gesetzt. Die Funktionen selber waren aber kleiner als dieser Overhead damit sehr ineffizient. Hier könnte man natürlich anmerken, dass die Java VM kleine Methoden automatisch als inline Methoden behandelt, doch darauf kann man sich nicht bei jeder VM Implementation verlassen.

9.2 Optimierung der Schleifen

Kollisionsprüfungen sind extrem rechenintensiv. Entsprechend effizient musste dieser Codeteil gestaltet werden. Bei einer verschachtelten Schleife, die alle Objekte untereinander testen sollte, ist dabei ein kleiner Fehler unterlaufen, der jedoch nur die Ausführgeschwindigkeit beeinflusste.

```
for(y=0; y to anzahlObjekte){
 for(x=0; x to anzahlObjekte){
  //Kollision untereinander testen
 }
}
```

Die Variablenzuweisung $x = 0$ wurde auf $x = y + 1$ geändert um somit der Rechenaufwand von n^2 auf $n(n-1)$ reduziert.

10 Fazit

In dieser Arbeit habe ich gezeigt, wie man mit einem einfachen Physikmodell glaubhafte und vor allem stabile physikalische Simulationen erzeugen kann. Es wurden alle entscheidenden Aspekte, die ein Physik-Engine ausmachen, wie der Integrator, die Kollisionserkennung und die Kollisionsauflösung, beleuchtet und teilweise mit Pseudocode illustriert. Das Produkt ist eine voll funktionsfähige 2D Physik-Engine die man als Library in jedes beliebige Java Projekt integrieren und verwenden kann.

Das einfache Physikmodell erlaubte schnelle und gute Ergebnisse, zeigte jedoch seine Grenzen bei der Simulation von Reibung und Impuls. Die Implementation von Reibung stellte sich als schwierig heraus und das Anbringen von korrekten Separationsimpulsen ist trotz mehreren Versuchen gescheitert. Die hier gezeigte Physik-Engine ist sehr schnell. Eine Begrenzung der Geschwindigkeit erfolgt durch die Kollisionserkennung, die sehr rechenzeitintensiv ist. Dieses Problem weisen anderen Physik-Engines auch auf.

Ein nächster Schritt wäre das Problem mit der Impulserhaltung zu lösen und die Engine in die nächste Dimension zu erweitern. Letzteres würde eine komplette Neuschreibung des Kollisions- und Kollisionsauflösungsteiles bedeuten. Dies wäre mit meinem neu erworbenen Know-how in einem vernünftigen Zeitrahmen möglich.

Meine Erwartung an das Produkt hat sich erfüllt. Die investierte Zeit war für mich sehr lehrreich.

11 Literaturverzeichnis

[1] I. Millington, Game Physics Engine Development, San Francisco: Elsevier, 2007.

[2] T. Jakobsen, «Advanced Character Physics,» 21 Januar 2003. [Online]. Available: http://www.gotoandplay.it/_articles/2005/08/advCharPhysics.php. [Zugriff am 19 Oktober 2011].

[3] Wikipedia, «Physik-Engine,» 2011. [Online]. Available: http://de.wikipedia.org/wiki/Physik-Engine. [Zugriff am 30 Oktober 2011].

[4] G. Fiedler, «Fix Your Timestep!,» 2 September 2006. [Online]. Available: http://gafferongames.com/game-physics/fix-your-timestep/. [Zugriff am 19 Oktober 2011].

[5] M. Müller, B. Heidelberger, M. Hennix und J. Ratcliff, «Position Based Dynamics,» 2006. [Online]. Available: http://www.matthiasmueller.info/publications/posBasedDyn.pdf. [Zugriff am 19 Oktober 2011].

[6] W. Bittle, «SAT (Seperating Axis Theorem),» 1 Januar 2010. [Online]. Available: http://www.codezealot.org/archives/55. [Zugriff am 19 Oktober 2011].

[7] B. Bitterli, «A Verlet based approach for 2D game physics,» 19 November 2009. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/math-and-physics/a-verlet-based-approach-for-2d-game-physics-r2714. [Zugriff am 19 Oktober 2011].

12 Abbildungsverzeichnis

13 Danksagung

Ich danke all jenen, die zur Entstehung dieser Arbeit beigetragen haben. Ein besonderer Dank geht an meinen Betreuer Stefan Rothe, der mir mit Tipps und Ratschlägen sehr geholfen hat. Mein Dank geht auch an meine Familie, insbesondere an meinen Vater, der mir bei der Suche nach Rechtschreibefehler geholfen hat und an meine Schwester die mich stetig motiviert hat. Ein weiteres Dankeschön richtet sich auch an alle Leser, die sich die Zeit genommen haben diese Arbeit zu lesen.

Ostermundigen bei Bern im Oktober 2011
Robert Balas