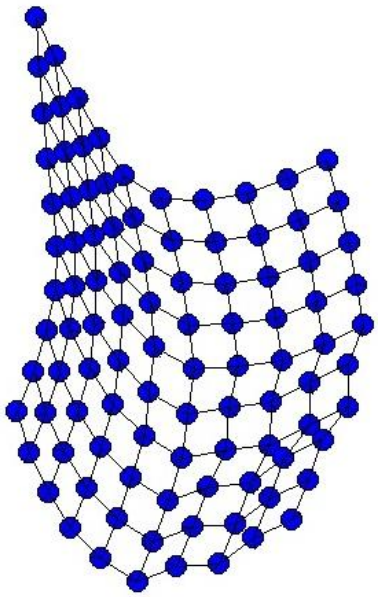
Gymnase de Burier

SiMS

Ein einfacher Physiksimulator



St-Légier, November 2009

Jakob Odersky
3M4

Ich möchte mich bei allen Personen, die mir Hilfe und/oder Unterstützung bei dieser Maturaarbeit geleistet haben, herzlichst bedanken. Insbesondere bei folgenden Personen, ohne deren Hilfe diese Arbeit, wie sie ist, nicht möglich gewesen wäre:

Ulrich Fuchs – für seine Betreuung meiner Maturaarbeit in Deutschland.

Alain Salanon – für seine Betreuung meiner Maturaabeit in der Schweiz.

Ursula Schallenberger – für ihre hilfreichen Tips, was die Rechtschreibung betrifft und das Korrekturlesen.

Sabine Odersky – für ihre Unterstützung und Hilfe in Deutsch.

Martin Odersky – für seine wertvollen Ratschläge und Erklärungen.

Familie Broers – für ihre moralische Unterstützung und das wundervolle Jahr.

Inhalt

Anmerkungen

Schreibweise

Vektoren und Matrizen werden fett geschrieben.

Dreidimensionale Vektoren in einer zweidimensionalen Simulation

In der Mechanik werden manche Größen wie z. B. Winkelgeschwindigkeit oder Drehmoment eines Körpers mit 3-dimensionalen Pseudovektoren dargestellt. Ihre Richtung entspricht der Drehachse und ihre Norm der Stärke. In 2-dimensionalen Simulationen kann sich ein Körper nur auf der Ebene drehen, d. h. seine Rotationsachse ist immer orthogonal zu den x1- und x2-Achsen. Solche pseudovektorialen Größen können auch in 2D mit Skalaren dargestellt werden. Diese Skalare entsprechen der 3. Komponente des Pseudovektors.

In diesem Dokument werden 3-dimensionale Pseudovektoren symbolisch in ihrer Natur dargestellt (als Vektor fett gedruckt) um Rechenmöglichkeiten wie das Vektorprodukt auszunutzen. Numerisch werden sie jedoch, wie oben beschrieben, als Skalar dargestellt. In einer Gleichung zwischen der korrekten symbolischen Darstellung und der numerischen befindet sich ein Äquivalenzzeichen (≡).

Wenn $\boldsymbol{\omega}$ ein 3D-Pseudovektor ist, und $\boldsymbol{r}$ ein 2D-Vektor ist, dann ist das Kreuzprodukt $\boldsymbol{\omega} \times \boldsymbol{r} = \omega \boldsymbol{r}_{\perp}$ wobei ω die 3. Komponente des Pseudovektors ist und $\boldsymbol{r}_{\perp}$ der linke Normalenvektor zu $\boldsymbol{r}$ ist.

Einheiten

Alle verwendeten Größen entsprechen den SI-Einheiten.

UML Diagramme

Diese Dokumentation ist oft mit UML Diagrammen illustriert. Ein gesamtes Klassendiagramm des Simulators befindet sich im Anhang.

1. Einleitung

Die Physiksimulation ist eine Domäne, die mit der Entwicklung und Steigerung der Rechenleistung von Computern immer verbreiteter wurde. Heutzutage findet sie Anwendung in diversen Branchen, von der Unterhaltungstechnologie bis zur Industrie. Um einige Beispiele zu nennen: Sie ist es, die es Computerspielen ermöglicht, realistisch zu wirken oder Automobilherstellern hilft, die Entwicklung neuer Modelle dank virtuellen Simulationen um ein Vielfaches zu beschleunigen. Physiksimulation, wie der Name schon andeutet, heißt, Systeme nach den Gesetzen der Physik möglichst realistisch darzustellen. Um dies zu ermöglichen, muss jedes Gesetz, jeder Zusammenhang von Größen in einem Simulationsprogramm implementiert werden. Wegen der großen Anzahl an Unterdomänen der Physik sind solche Simulationsprogramme meistens auf eine Domäne spezialisiert. So kommt es, dass Computerspiele sich meistens auf die Mechanik der Festkörper konzentrieren, das Verhalten von Molekülen aber vernachlässigen. Letzteres ist aber in der Materialforschung von großer Bedeutung, wobei Festkörper wiederum keine Rolle spielen.

Das Ziel dieser Arbeit war es, eine zweidimensionale Simulationsbibliothek für Festkörper zu erstellen. Somit sollten Interaktionen zwischen Festkörpern und physikalischen Größen veranschaulicht werden.

Während den ersten Nachforschungen ergab sich jedoch, dass viele Physiksimulatoren sehr leistungsstark, deshalb aber für den Laien auch ziemlich kompliziert gestaltet sind. Dadurch verlieren die physikalischen Interaktionen meistens an Anschaulichkeit. Deshalb sollte die Simulationsbibliothek möglichst überschaubar und einfach gestaltet werden. Dazu sollte sie so strukturiert werden, dass man sie ohne großen Aufwand erweitern oder bereits existierende Funktionen verbessern könnte. Der Name dieser Bibliothek ist „Simple Mechanics Simulator“ kurz „SiMS“.

Dieses Dokument ist nur ein Teil der Maturaarbeit. Es handelt sich um eine Dokumentation, die die verschiedenen Konzepte des Simulators und ihre Implementierung erläutert. Für Details über den anderen Teil der Arbeit, die Bibliothek und die gesamte Implementierung, geben der Quellcode und die API Dokumentation Auskunft[1]. Die erläuterten Konzepte fangen mit der Modellierung von Festkörpern an. Anschließend wird das Funktionieren einer virtuellen Welt und die Darstellung der Zeit erklärt. Danach werden Randbedingungen, die die Bewegungen der Körper einschränken, beschrieben. Zum Schluss wird noch eine einfache Methode zur Kollisionserkennung und -reaktion erläutert. Diese Dokumentation gibt keine Schritt-für-Schritt Anleitung und ist nicht dafür geschrieben, dass ein Leser danach ohne Weiteres einen Simulator programmieren kann. Sie ist vielmehr ein Überblick über die gesamte Funktionsweise von SiMS.

Um diese Arbeit besser zu verstehen, sind Grundkenntnisse der Mechanik erforderlich, vor allem Dynamik und Kinematik. Sie können in dem Buch *PHYSIQUE – 1. Mécanique*[2] nachgelesen werden. Die in SiMS verwendete Programmiersprache ist Scala. Diese Sprache kann mit dem Buch *Programming in Scala*[3] gelernt werden. Zwar ist Scala keine zwingende Voraussetzung, dennoch erleichtern gute Kenntnisse von objektorientierten und funktionalen Sprachen das Verstehen der Implementierung. Schließlich sind mathematisch-geometrische Kenntnisse eine Voraussetzung, vor allem Vektor- und Matrizenrechnung.

[1] Siehe Anhang II, Zusammensetzung der Arbeit.
[2] (Halliday, Resnick und Walker 2004)
[3] (Odersky, Spoon und Venners 2008)

2. Körper und Formen

Ein wesentlicher Bestandteil eines Festkörpermechaniksimulators wie SiMS sind Festkörper. Dieses Kapitel erklärt ihren Aufbau und ihr Funktionieren. Aspekte wie die Eigenschaften der Körper und ihre Zerlegung in Formen wurden von dem bereits vorhandenen Simulator jBox2D[4] inspiriert. Der Quellcode für Körper und Formen befindet sich in dem Package `sims.dynamics`.

2.1. Eigenschaften von Körpern und Formen

Körper sind die Grundkomponenten und bilden die Basis des Simulators. Der Zustand eines Systems aus Körpern ergibt sich aus den momentanen Werten der Eigenschaften aller Körper. Diese Eigenschaften lassen sich in zwei verschiedene Arten unterteilen: dynamische und statische. Dynamische Eigenschaften können sich mit jedem Augenblick der Zeit ändern. Statische hingegen bleiben während der Existenz eines Körpers immer Konstant. Erstere wären z. B. Geschwindigkeit, Position und resultierende Kraft und letztere z. B. Masse, Dichte und Trägheitsmoment.
Dynamische Eigenschaften hängen vom Zustand des Körpers ab, statische Eigenschaften dagegen von der räumlichen Struktur des Körpers. Diese Struktur ergibt sich aus einer Zusammensetzung mehrerer 2-dimensionalen Formen.
Ein Körper kann also als eine Sammlung von Formen und dynamischen Informationen betrachtet werden. Manche seiner Eigenschaften sind dann nur Werte und andere werden durch seine Formen ermittelt.

2.2. Körper

In SiMS werden Körper mit der Klasse `Body` definiert. Körper enthalten fast alle dynamischen Eigenschaften als Variablen. Statische Eigenschaften werden als Methoden die auf ihre Formen zurück greifen, modelliert. Abbildung 1 zeigt das Klassendiagramm von Körpern und Formen, wobei `Shape` die Klasse von Formen ist.

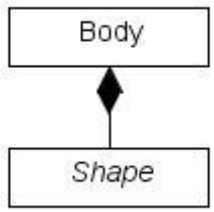


Abbildung 1: Klassendiagramm von Körpern und Formen

Körper enthalten die in folgenden Abschnitten erläuterten Eigenschaften.

2.2.1. Kraft und Drehmoment

Resultierende Kraft und Drehmoment sind dem Körper eigen. Zwar sind ihre Auswirkungen auf den Körper je nach räumlicher Struktur verschieden, doch hängen ihre Werte nicht davon ab. Sie können also als Variablen definiert werden:

```
/**Resultierende Kraft auf den Schwerpunkt dieses Körpers.*/
var force: Vector2D = Vector2D.Null

/**Resultierendes Drehmoment auf den Schwerpunkt dieses Körpers.*/
var torque: Double = 0
```

[4] jBox2D ist eine Javaversion von Erin Cattos Box2D. (jBox2D Demos kein Datum)

Bemerkung: Körper haben keine Variablen für Beschleunigung, denn diese ist mit der Masse proportional zu der resultierenden Kraft. Gleiches gilt für Winkelbeschleunigung.

2.2.2. Geschwindigkeit

Lineargeschwindigkeit und Winkelgeschwindigkeit sind komplett unabhängig von der Formenkomposition des Körpers. Daher können sie ebenfalls als Variablen deklariert werden:

```
/**Lineargeschwindigkeit dieses Körpers.*/
var linearVelocity: Vector2D = Vector2D.Null

/**Winkelgeschwindigkeit dieses Körpers.*/
var angularVelocity: Double = 0
```

2.2.3. Position

Position und Rotation eines Körpers sind eigentlich auch unabhängig von seiner räumlichen Struktur. Doch in SiMS wird die Position eines Körpers als sein Schwerpunkt festgelegt. Dieser ist wiederum abhängig von der räumlichen Struktur. Außerdem verschiebt eine Positionsänderung auch die Formen des Körpers. Somit werden Position und Rotation als Methoden definiert:
Ermittelt wird die Position (=Schwerpunkt) nach den Massen und Schwerpunkten der Formen.

```
  def pos: Vector2D = // Shwerpunkt = Summe(Pos*Masse)/Summe(Masse)
    (Vector2D.Null /: shapes)((v: Vector2D, s: Shape) => v + s.pos *
s.mass) / (0.0 /: shapes)((i: Double, s: Shape) => i + s.mass)
```

Im obrigen code bezeichnet `/:` den Operator für „Faltung“. Gegeben ein Anfangswert z, eine Folge xs, und eine Operation op. So bezeichnet der Ausdruck $(z\ /{:}\ xs)(op)$ die Anwendung $z\ op\ xs_1\ op\ xs_2\ op\ \ldots\ op\ xs_n$.
Um die Position zu setzen, werden alle Formen entsprechend der neuen Position verschoben.

```
def pos_=(newPos: Vector2D) = {
  val stepPos = pos
  shapes.foreach((s: Shape) => s.pos = s.pos - stepPos + newPos)
}
```

Wenn sich ein Körper zum Beispiel auf der Position (1,2) befindet und auf eine neue Position (2,5) gesetzt wird, werden alle Formen um (2,5)-(1,2)=(1,3) verschoben.

Analog zu der Position dreht eine Rotationsänderung alle Formen entsprechend der neuen Rotation.

2.2.4. Masse und Trägheitsmoment

Die letzten beiden mechanischen Eigenschaften in SiMS, die sich auf den ganzen Körper beziehen, sind Masse und Trägheitsmoment. Diese beiden Eigenschaften sind statisch und hängen somit von den Formen des Körpers ab.
Die Masse ist lediglich die Summe der Massen der Formen.

```
def mass: Double = if (fixed) Double.PositiveInfinity else
    (0.0 /:  shapes)((i: Double, s: Shape) => i + s.mass)
```

Das Trägheitsmoment wird nach dem Steinerschen Satz errechnet. Hierfür werden die Masse und das Trägheitsmoment der Formen gebraucht.

```
  def I: Double = if (fixed) Double.PositiveInfinity else
    (0.0 /: (for (s <- shapes) yield (s.I + s.mass * ((s.pos - pos) dot
(s.pos - pos))))))(_+_)
```

Bemerkung: `mass` und `I` prüfen ob `fixed` wahr ist und geben gegebenenfalls einen unendlich großen Wert (`Double.PositiveInfinity`) zurück. Das Feld `fixed` gibt an, ob der Körper fixiert (=unbeweglich) ist. In diesem Fall sind Masse und Trägheitsmoment unendlich groß, um Beschleunigungen durch einwirkende Kräfte nichtig zu machen.

2.3. Formen

Wie beschrieben, enthalten Körper eine Sammlung von Formen. Formen besitzen auch Eigenschaften, aus denen ein Körper Teile seiner eigenen Eigenschaften errechnet. Eine Form ist jedoch in manchen Aspekten abstrakt. Sie könnte ein Kreis sein, ein Rechteck oder irgendein anderes Polygon. Je nach ihrer räumlichen Struktur sind ihre statischen Eigenschaften auch unterschiedlich. In SiMS ist die Klasse, die Formen beschreibt (`Shape`), deshalb auch abstrakt. In ihr werden manche Eigenschaften nur deklariert und nicht implementiert. Somit können neue Formen ohne großen Aufwand definiert werden. Eine neue Form muss nur die Klasse `Shape` erben und die statischen Eigenschaften der Form konkret implementieren. In SiMS sind bereits drei Typen von konkreten Formen implementiert: Kreise (`Circle`), Rechtecke (`Rectangle`) und regelmäßige Polygone (`RegularPolygon`).

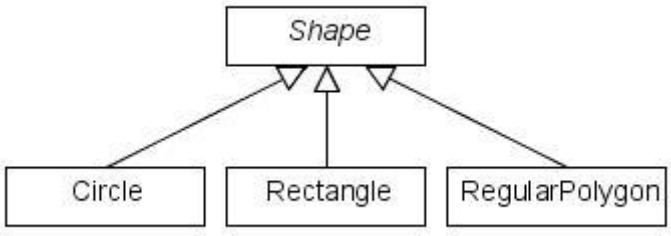


Abbildung 2: Klassenhierarchie von Formen

2.3.1. Statische Eigenschaften

Jede der konkreten Formen implementiert die Eigenschaften: Dichte, Volumen, Masse und Trägheitsmoment.

Dichte, Volumen und Masse hängen alle miteinander zusammen. Kennt man zwei dieser Werte, kann man den dritten errechnen. Um eine Form aus einem bekannten Material zu erstellen, kennt man in der Praxis meistens die Dichte des Materials. Deswegen werden in `Shape` Dichte und Volumen abstrakt gelassen. Die Masse wird dann aus Dichte mal Volumen errechnet.

Bemerkung: Formen besitzen kein Volumen, sie haben nur eine Fläche. In diesem Fall wird aber die Bezeichnung Volumen beibehalten, um dem Satz „Masse gleich Volumen mal Dichte“ konform zu bleiben.

Das Trägheitsmoment hängt direkt von der räumlichen Struktur ab und ist deswegen auch abstrakt.

```
val density: Double //Dichte
val volume: Double //Volumen
def mass = volume * density //Masse
val I: Double //Trägheitsmoment
```

In einem Kreis z. B. sind diese Werte und Methoden konkret implementiert. Die Dichte wird bei der Initialisierung festgelegt und das Volumen wird durch πr^2 (r = Radius des Kreises) gegeben. Das Trägheitsmoment entspricht der Masse mal dem Radius im Quadrat durch zwei.

2.3.2. Formenspezifische Eigenschaften

Die vorher genannten Eigenschaften werden immer auch verwendet um Eigenschaften eines Körpers zu ermitteln. Es gibt aber auch Eigenschaften, die sich nur auf eine Form beziehen und keine Auswirkung auf den gesamten Körper haben.
Die Stoßzahl bei einer Kollision hängt immer von den Materialen der beiden kollidierenden Formen ab. Genauso ist es mit dem Reibungskoeffizienten. Diese beiden Werte sind also nur in `Shape` implementiert und haben keinen Einfluss auf `Body`.

2.4. Zusammenfassung der Eigenschaften

Tabelle 1 fasst die mechanischen Eigenschaften von Körpern zusammen. Kursiv gedruckte Einträge bedeuten, dass diese Eigenschaft aus den Formen der Körper ermittelt wird.
Tabelle 2 fasst die mechanischen Eigenschaften von Formen zusammen.

Eigenschaften von Körpern	Symbol	Name der Variable/Methode
Resultierende Kraft	$\boldsymbol{F}$	`force`
Resultierendes Drehmoment	$\boldsymbol{\tau}$	`torque`
Lineargeschwindigkeit	$\boldsymbol{v_s}$	`linearVelocity`
Winkelgeschwindigkeit	$\boldsymbol{\omega}$	`angularVelocity`
Position (Schwerpunkt)	$\boldsymbol{x}$	`pos`
Rotation	$\boldsymbol{\theta}$	`rotation`
Masse	m	`mass`
Trägheitsmoment	I	`I`

Tabelle 1: mechanische Eigenschaften von Körpern

Eigenschaften von Formen	Name der Variable/Methode
Position	`pos`
Rotation	`rotation`
Dichte	`density`
Volumen	`volume`
Masse	`mass`
Trägheitsmoment	`I`
Stoßzahl	`restitution`
Reibungskoeffizient	`friction`

Tabelle 2: mechanische Eigenschaften von Formen

2.5. Funktionen

Mit ihrem Aufbau und ihren Eigenschaften definiert, könnten nun bereits Körper erstellt werden. Es bleibt aber ziemlich mühsam, wenn man gewisse dynamische Eigenschaften ändern möchte. Wenn man z. B. eine Kraft simulieren möchte, die auf einen Punkt einwirkt, muss man die resultierende Kraft ebenso wie das resultierende Drehmoment ändern. Ähnlich ist es, wenn man die Geschwindigkeit eines Punktes auf dem Körper kennen möchte: erst müssen Lineargeschwindigkeit und Winkelgeschwindigkeit bekannt sein.
Deswegen gibt es in Körpern mehrere Hilfsfunktionen, die die dynamischen Eigenschaften verändern oder andere Werte aus ihnen berechnen. So muss man z. B. nicht Linear- und Winkelgeschwindigkeit getrennt ändern, sondern lässt nur einen Impuls auf einen Punkt mit der Methode `applyImpulse` wirken. Körper besitzen folgende Hilfsfunktionen:

Funktion	Wirkung
`applyForce(force: Vector2D)`	Die Kraft `force` wirkt auf den Schwerpunkt. Die resultierende Kraft wird um `force` erhöht.
`applyForce(force: Vector2D, point: Vector2D)`	Die Kraft `force` wirkt auf den gegebenen Punkt `point`. Resultierende Kraft und resultierendes Drehmoment werden entsprechend verändert.
`applyImpulse(impulse: Vector2D)`	Der Impuls `impulse` wirkt auf den Schwerpunkt. Die Lineargeschwindigkeit ändert sich.
`applyImpulse(impulse: Vector2D, point: Vector2D)`	Der Impuls `impulse` wirkt auf den gegebenen Punkt `point`. Die Linear- und Winkelgeschwindigkeiten werden entsprechend geändert.
`velocityOfPoint(point: Vector2D)`	Die Geschwindigkeit des Punktes `point` wird durch Linear- und Winkelgeschwindigkeit des Körpers ermittelt.

Tabelle 3: Hilfsfunktionen von Körpern

Punkte werden in Weltkoordinaten gegeben.

2.6. Schluss

In diesem Kapitel wurde erklärt, was Körper sind. Ihre Eigenschaften wurden erläutert und es wurde gezeigt, dass sie sich aus Formen zusammensetzen. Das erweiterbare Typsystem in SiMS ermöglicht es, neue Formen ganz leicht zu erstellen und daher Körper mit fast unbegrenzten räumlichen Strukturen zu bauen. Hinzu wurden noch nützliche Funktionen vorgestellt, die das Simulieren erleichtern.

3. Simulation einer Welt

Nachdem Körper erstellt werden können, ist der erste Schritt zu einem Simulator getan. Nun müssen mehrere Körper aber automatisch simuliert werden können. Das heißt, es muss eine Methode geben, die die Eigenschaften der Körper nach der Zeit und nach Bedingungen einer virtuellen Welt ändern kann.

3.1. Die Welt

Ein intuitiver Weg besteht darin eine Welt zu erstellen. Wie eine echte Welt enthält sie Körper und Faktoren, die die Bewegungen der Körper beeinflussen, wie z. B. die Schwerkraft. Körper, die zusammen, unter gleichen Bedingungen simuliert werden sollen, werden ihr hinzugefügt. In SiMS entspricht eine Welt der Klasse `dynamics.World`. Bei der Initialisierung einer neuen Welt ist diese zunächst leer, d. h. sie enthält keine Körper. Körper können ihr durch das Feld `bodies` hinzugefügt werden.

```
val bodies = new ArrayBuffer[Body]
```

3.2. Zeitintegration

Ist eine Welt einmal erstellt und Körper ihr hinzugefügt, soll sie es ermöglichen, alle Eigenschaften der Körper nach der Zeit und entsprechend den Bedingungen, die in ihr herrschen, zu ändern. Die Methode, die dies mit der höchsten Präzision erfüllen würde, wäre die Bewegungen der Körper als eine Funktion nach der Zeit (t) darzustellen: Die Geschwindigkeit ($\boldsymbol{v}$) ist die Integration der Beschleunigung ($\boldsymbol{a}$) über die Zeit und die Position ($\boldsymbol{x}$) ist die Integration der Geschwindigkeit über die Zeit. Diese bekannten Funktionen wären dann:

$$\boldsymbol{v}(t) = \int \boldsymbol{a}\, dt = \boldsymbol{a}t + \boldsymbol{v}_0$$

$$\boldsymbol{x}(t) = \int \boldsymbol{v}\, dt = \frac{1}{2}\boldsymbol{a}t^2 + \boldsymbol{v}_0 t + \boldsymbol{x}_0$$

Doch nicht immer sind diese Gleichungen einfach zu lösen. Was ist, wenn z. B. eine Feder eine Kraft proportional zu der Position eines Körpers ausübt? Oder wenn ein Körper wie bei Luftwiderstand proportional zu seiner Geschwindigkeit gebremst wird? Dann kommen schnell schwer zu lösende Differentialgleichungen ins Spiel. Für den Luftwiderstand wäre diese z. B.: (c ist eine Konstante)

$$\boldsymbol{a} = -c\boldsymbol{v} \text{ und } \boldsymbol{a} = \frac{d\boldsymbol{v}}{dt}$$

$$\Leftrightarrow \frac{d\boldsymbol{v}}{dt} = -c\boldsymbol{v}$$

Solche Differentialgleichungen können in einem System aus vielen Körpern so kompliziert werden, dass es unmöglich wird, sie schnell und symbolisch zu lösen.

Was nun? Nachdem er oben genanntes Problem mit Differentialgleichungen erörtert hat, stellt Chris Hecker in einem Artikel[5] einen anderen Weg vor, wie Geschwindigkeiten und Positionen integriert werden können, ohne Differentialgleichungen lösen zu müssen: die Eulersche Integration. Sie ist eine einfache Variante der numerischen Integration. Sie besteht grob darin, das Integral einer Funktion numerisch zu approximieren.

Greift man zurück auf die Definition einer Ableitung:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

[5] (Hecker 1996)

So kann man daraus herleiten:

$$\lim_{h\to 0} f(x+h) = \lim_{h\to 0}\left(f(x) + h\frac{df(x)}{dx}\right)$$

Grenzt h nicht an null, ist aber dennoch relativ klein, kann man behaupten:

$$f(x+h) \approx f(x) + h\frac{df(x)}{dx}$$

Dies ist die Eulersche Integration: kennt man einen Wert der Funktion, kann man folgende Werte herausfinden, in dem man der Ableitung der Funktion über einen kleinen Wert (h) folgt. Man „tastet" sich sozusagen an der Funktion entlang. Natürlich entstehen dabei kleine Fehler, doch ist h klein, bleiben die Fehler auch klein.

Bezieht man diese Generalität nun auf die Bewegungsgleichungen der Körper, so erhält man

$$\boldsymbol{v}(\boldsymbol{v}_0 + h) \approx \boldsymbol{v}_0 + h\frac{d\boldsymbol{v}}{dt} = \boldsymbol{v}_0 + h\boldsymbol{a}$$

$$\boldsymbol{x}(\boldsymbol{x}_0 + h) \approx \boldsymbol{x}_0 + h\frac{d\boldsymbol{x}}{dt} = \boldsymbol{x}_0 + h\boldsymbol{v}$$

Man kann also die Bewegungen simulieren, ohne Differentialgleichungen zu lösen. Alles was dafür benötigt wird, ist eine bekannte initiale Situation (für die Werte $\boldsymbol{x}_0$ und $\boldsymbol{v}_0$).

3.3. Zeitschritte

Solche initialen Situationen können durch das Einführen von Zeitschritten erschaffen werden. Ein Zeitschritt ist eine sehr kleine Zeitspanne, in der die Welt simuliert wird. Zu Beginn jedes Zeitschrittes werden die Geschwindigkeiten und Positionen der Körper als initiale Werte betrachtet. Dann werden neue Geschwindigkeiten und Positionen mit Hilfe der Eulerschen Integration errechnet. Der Wert h der Eulerischen Integration ist die Dauer des Zeitschrittes. Eine ablaufende Zeit wird durch konsekutives Ausführen von Zeitschritten simuliert. Je kleiner der Zeitschritt, desto größer bleibt die Präzision zu den Bewegungen in einer realen Welt.

In der Welt wird ein Zeitschritt (n) durch die Methode `step()` ausgeführt. Diese Methode simuliert einen Zeitschritt, dessen Dauer durch die Variable `timeStep` definiert ist.

Der Zeitschritt wird in verschiedenen Phasen[6] durchgeführt:

1. Um die Beschleunigungen der Körper zu ermitteln, werden alle Kräfte auf die Körper angewandt (z. B. Schwerkraft, Federkräfte). Durch die resultierenden Kräfte der Körper können ihre Beschleunigungen errechnet werden.
2. Die Beschleunigungen der Körper werden über die Dauer des Zeitschrittes integriert, um so die Endgeschwindigkeiten der Körper zu errechnen.

   $$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h\frac{\boldsymbol{F}_n}{m}$$

3. Die Geschwindigkeiten der Körper werden über die Dauer des Zeitschrittes integriert, um so die Position der Körper zu errechnen.

   $$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\boldsymbol{v}_{n+1}$$

4. Die resultierenden Kräfte der Körper werden auf null zurückgesetzt, für den nächsten Zeitschritt.

[6] Die hier genannten Phasen sind nicht vollständig. Im nächsten Kapitel werden noch zwei weitere hinzugefügt.

In der Welt ist `step()` folgend implementiert:

```
def step() = {
  //...

  //Phase 1, angewandte Kräfte
  for (j <- joints) j match {case f: ForceJoint => f.applyForce;
    case _ => ()}//Kräfte von Objekten werden angewandt (z.B. Federkraft)


  for (b <- bodies) {

    //Schwerkraft wird angewandt
    val m = b.mass
    b.applyForce(gravity * b.mass)

    val a = b.force / b.mass
    val alpha = b.torque / b.I

    //Phase 2, Integration der Beschleunigungen
    b.linearVelocity = b.linearVelocity + a * timeStep
    b.angularVelocity = b.angularVelocity + alpha * timeStep
  }

  //Weitere Phase (siehe Abschnitt 4.10)
  //...

  for (b <- bodies) {
    //...

    //Phase 3, Integration der Geschwindigkeiten
    b.pos = b.pos + b.linearVelocity * timeStep
    b.rotation = b.rotation + b.angularVelocity * timeStep

    //Phase 4, Nullsetzung der resultierenden Kräfte
    b.force = Vector2D.Null
    b.torque = 0.0
  }

  //Weitere Phase (siehe Abschnitt 4.10)
  //...


  //...
}
```

3.4. Schluss

In diesem Kapitel wurde gezeigt, wie Körper zusammen unter realistischen Bedingungen simuliert werden können. Es wurde die Klasse einer Welt eingeführt. Dazu wurde eine Methode beschrieben, wie Körper nach der Zeit simuliert werden können.

4. Constraints[7]

Bis zu dieser Stelle wurde gesehen, wie Körper erstellt und nach der Zeit simuliert werden können. Es wurde immer angenommen, sie könnten sich frei nach angewandten Kräften und Impulsen bewegen. Nun ist dies aber in einer Physiksimulation oft nicht der Fall und Körper unterliegen gewissen Randbedingungen, die die Freiheit ihrer Bewegungen einschränken, wie z. B. die Randbedingung eines Seiles, dass die beiden angebundenen Körper daran hindert, einen größeren Abstand zu halten als die Länge des Seiles. Weitere Randbedingungen wären Kollisionen, die Körper daran hindern sich zu durchdringen, oder Gelenke, die Körper an einem definierten Punkt fixieren. Diese Randbedingungen, auch *Constraints* genannt, müssen in einer realistischen Simulation mit einbezogen werden, ohne die bestehenden Gesetze der Mechanik zu brechen. Um sie aber in einen Physiksimulator zu implementieren, müssen sie erst modelliert werden. SiMS hat alle Constraints, wie sie nach dem Konzept von Erin Catto beschrieben werden, implementiert. In diesem Kapitel wird dieses Konzept erläutert.

4.1. Beispiel – Einheitskreis

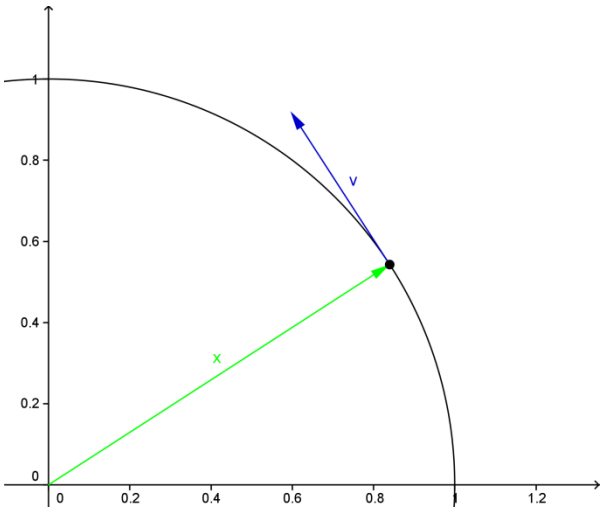


Abbildung 3: Teilchen auf Einheitskreis

Als Einstieg betrachte man folgendes Beispiel:
Ein Teilchen mit der Position $\boldsymbol{x}$ und der Geschwindigkeit $\boldsymbol{v}$ ist mit einer masselosen Stange der Länge 1m mit dem Ursprung verbunden. Das heißt, es kann sich frei bewegen, solange es einen Abstand von einem Meter mit dem Ursprung bewahrt. Oder anders gesagt, das Teilchen unterliegt der Randbedingung, sich immer auf dem Einheitskreis zu befinden.

4.2. Constraints mit Teilchen

Bevor Constraints in einer Simulation mit einbezogen werden können, müssen sie modelliert werden. Hierzu werden als erster Schritt alle erlaubten Positionen des Teilchens definiert. Eine Möglichkeit wäre, eine Menge aller legalen Positionen zu erstellen. In dem Einstiegsbeispiel wären dies alle Punkte auf dem Einheitskreis. Eine Menge mit unendlich vielen Elementen kann jedoch nicht ohne Weiteres modelliert werden.
Viel einfacher ist es, die erlaubten Positionen mittels einer Gleichung zu beschreiben. Diese Gleichung hat eine Lösung, wenn sich das Teilchen auf einer legalen Position befindet. In dem Einstiegsbeispiel wäre diese Gleichung: $\|\boldsymbol{x}\| = 1$ oder $\|\boldsymbol{x}\| - 1 = 0$. Die Randbedingung des Teilchens wäre also, dass diese Gleichung immer wahr ist; sprich, das Teilchen befindet sich immer

[7] Das Konzept der Constraints wurde von (Catto, Modelling and Solving Constraints 2008) übernommen

auf einer erlaubten Position. Man spricht von einer Positionsrandbedingung, auch *Position-Constraint* genannt. Einfachheitshalber wird die zweite Gleichung als Position-Constraint verwendet, denn sie hat den Vorteil, dass sich alle Randbedingungen in die Struktur $C(\boldsymbol{x}) = 0$ umwandeln lassen. Somit muss später nur ein einheitlicher Weg gefunden werden, um alle Randbedingungen in der Simulation zu berücksichtigen.

Nun müssen als nächstes alle erlaubten Geschwindigkeiten definiert werden, damit die Position-Constraint eingehalten bleibt. Da die Geschwindigkeit die erste Ableitung nach der Zeit der Position ist, können alle erlaubten Geschwindigkeiten mit der Gleichung $\dot{C}(\boldsymbol{x}) = 0$ beschrieben werden. Analog zu der Positionsrandbedingung ist diese Gleichung die Geschwindigkeitsrandbedingung, auch *Velocity-Constraint* genannt.

Die linke Seite der Position-Constraint, $C(\boldsymbol{x})$, die *Position-Constraint-Funktion*, ist von der Position des Teilchens abhängig. Die *Velocity-Constraint-Funktion* $\dot{C}(\boldsymbol{x})$ ist also nach der Kettenregel

$$\dot{C}(\boldsymbol{x}) = \frac{dC}{dt} = \frac{dC}{d\boldsymbol{x}} \cdot \frac{d\boldsymbol{x}}{dt} = \frac{dC}{d\boldsymbol{x}} \cdot \boldsymbol{v}$$

Eine differenzierbare Funktion $f: \mathbb{R}^n \to \mathbb{R}$ abgeleitet ergibt eine 1xn-Matrix, wobei die Komponenten der Matrix gleich den partiellen Ableitungen von f sind. Man spricht von einer Jacobi-Matrix[8]. In der Velocity-Constraint-Funktion ergibt $\frac{dC}{d\boldsymbol{x}}$ eine 1x2-Jacobi-Matrix.

$$\boldsymbol{J} = \begin{bmatrix} \frac{dC}{dx_1} & \frac{dC}{dx_2} \end{bmatrix}$$

Nun kann die Velocity-Constraint also folgend dargestellt werden

$$\dot{C}(\boldsymbol{x}) = \frac{dC}{d\boldsymbol{x}} \cdot \boldsymbol{v} = \boldsymbol{J}\boldsymbol{v} = 0$$

Die Jacobi-Matrix $\boldsymbol{J}$ ist ein Reihenvektor, abhängig von der Position $\boldsymbol{x}$. Wenn nun die Velocity-Constraint eingehalten ist, also das Teilchen eine erlaubte Geschwindigkeit hat, beträgt die Velocity-Constraint-Funktion null und $\boldsymbol{J}$ ist orthogonal zu $\boldsymbol{v}$.

$$\boxed{\dot{C}(\boldsymbol{x}) = 0 \Leftrightarrow \boldsymbol{J}^T \cdot \boldsymbol{v} = 0 \Leftrightarrow \boldsymbol{J} \perp \boldsymbol{v}}$$

4.3. Constraint-Kräfte

Soweit wären die Positions- und Geschwindigkeitsrandbedingungen modelliert. Nun braucht man Methoden, diese Randbedingungen in der Simulation zu berücksichtigen. Wenn die Position- und Velocity-Constraints eingehalten sind, muss keine besondere Maßnahme getroffen werden, denn schließlich befindet sich das Teilchen an einem erlaubten Ort. Wenn die Constraints jedoch nicht eingehalten sind, müssen Geschwindigkeit und Position korrigiert werden. Meistens muss nur die Geschwindigkeit korrigiert werden, denn wenn sie legal ist, sollte sich das Teilchen theoretisch nur auf erlaubten Positionen fortbewegen[9].

Dazu werden sogenannte *Constraint-Kräfte* oder *Constraint-Impulse* verwendet. Constraint-Kräfte ($\boldsymbol{F_c}$) sind Kräfte, die auf das Teilchen so einwirken, dass dieses immer legale Geschwindigkeiten beibehält. Da Randbedingungen von physikalischen Objekten (wie die Stange aus 4.1) erzeugt werden, entsprechen Constraint-Kräfte letzteren Reaktionskräften. Diese fügen einer Welt keine zusätzliche Energie hinzu.

[8] (Jacobi-Matrix 2009)

[9] Durch die Ungenauigkeit der Simulation bewegen sich Teilchen mit der Zeit auf unerlaubte Positionen. Siehe Kapitel 4.8

Also dürfen Constraint-Kräfte keine Arbeit leisten[10]:

$$Arbeit: W = Pt = 0$$
$$Leistung: P = \boldsymbol{F_c} \cdot \boldsymbol{v} = 0$$
$$\Leftrightarrow \boldsymbol{F_c} \perp \boldsymbol{v}$$

Es folgt daraus, dass die Constraint-Kraft $\boldsymbol{F_c}$ auch orthogonal zu $\boldsymbol{v}$ ist und somit linear abhängig von $\boldsymbol{J}^T$.

Ein Problem ergibt sich jedoch bei Constraint-Kräften. Wie schon gesagt, soll das Teilchen *immer* eine legale Geschwindigkeit haben. Kräfte beeinflussen nur die Beschleunigung. Um Geschwindigkeit zu ändern, benötigen sie Zeit. Das heißt, die Geschwindigkeit wird erst mit Verspätung korrigiert und das Teilchen könnte sich auf unerlaubten Positionen befinden. Um dieses Problem zu beheben, müsste eine nahezu unendliche Kraft in einer nahezu unendlich kleinen Zeitspanne wirken. Aus praktischen Gründen ist dies in einer Simulation nicht möglich. Jedoch gibt es eine andere physikalische Größe, die direkt[11] Geschwindigkeit beeinflussen kann, den Impuls. Der Impuls ($\boldsymbol{p}$) entspricht dem Produkt der Masse mit der Geschwindigkeit eines Teilchens und ist zeitlich proportional zu einer konstanten Kraft. Durch diese Proportionalität kann man anstelle einer Constraint-Kraft einen Constraint-Impuls ($\boldsymbol{p_c}$) ebenfalls in oben genannte Formel einsetzen. Der Constraint-Impuls ist somit auch orthogonal zu $\boldsymbol{v}$ und linear abhängig von $\boldsymbol{J}^T$

$$\boxed{\boldsymbol{p_c} = \boldsymbol{J}^T \lambda \mid \lambda \in \mathbb{R}}$$

Die Geschwindigkeit des Teilchens wird also mit einem Impuls, der orthogonal zu dieser ist, korrigiert. Nun ist aber die Stärke des Impulses noch unbekannt.

Die korrigierte Geschwindigkeit nach dem Impuls, die Endgeschwindigkeit, entspricht Newtons Gesetz einer gleichförmig beschleunigten Bewegung

$$\boldsymbol{v}_1 = \boldsymbol{v}_0 + \frac{\boldsymbol{F}_c}{m} t$$
$$\boldsymbol{v}_1 = \boldsymbol{v}_0 + \frac{\boldsymbol{p}_c}{m}$$

wobei $\boldsymbol{v_1}$ die Endgeschwindigkeit und $\boldsymbol{v_0}$ Anfangsgeschwindigkeit ist. Die Velocity-Constraint soll nach der Korrektur wieder gelten, also soll nach der Impulseinwirkung wieder folgendes stimmen:

$$\dot{C}(\boldsymbol{x}) = 0$$
$$\boldsymbol{J}\boldsymbol{v}_1 = 0$$
$$\boldsymbol{J}\left(\boldsymbol{v}_0 + \frac{\boldsymbol{p}_c}{m}\right) = 0$$
$$\boldsymbol{J}\left(\boldsymbol{v}_0 + \frac{\boldsymbol{J}^T \lambda}{m}\right) = 0$$

Der Constraint-Impuls kann nach Auflösen der Gleichung nach λ ermittelt werden.

$$\frac{\boldsymbol{J}(\boldsymbol{J}^T \lambda)}{m} = -\boldsymbol{J}\boldsymbol{v}_0$$
$$\lambda = -m\frac{\boldsymbol{J}\boldsymbol{v}_0}{\boldsymbol{J}\boldsymbol{J}^T}$$
$$\boxed{\boldsymbol{p}_c = -\boldsymbol{J}^T m\frac{\boldsymbol{J}\boldsymbol{v}_0}{\boldsymbol{J}\boldsymbol{J}^T}}$$

[10] Die Idee, Constraint-Kräfte dürfen keine Arbeit leisten wird erklärt in (Witkin 1997). Constraints dürfen in SiMS per Definition keine Arbeit leisten. Eine Ausnahme gibt es: wenn ein Energie umwandelndes Objekt einen Impuls parallel zu der Jacobi-Matrix ausübt, kann dieser Impuls als Constraint-Impuls angesehen werden und eine Randbedingung aufgestellt werden. (Reibung stellt somit auch eine Constraint dar.)

[11] In der Praxis benötigen Impulse auch Zeit zu wirken, doch lässt sich die Geschwindigkeit zeitlich unabhängig aus der Impulsformel ($\boldsymbol{p} = m\boldsymbol{v}$) substituieren. Ihre Wirkungsdauer spielt daher keine Rolle.

Nun da der Impuls bekannt ist, kann er auf das Teilchen wirken und dieses bleibt auf seiner erlaubten Laufbahn.

4.4. Bezug auf Beispiel

Die vorherigen Abschnitte haben die Theorie der Randbedingungsmodellierung und -Korrektur erläutert. Nun sollte der Bezug auf ein konkretes Beispiel alles viel verständlicher machen. Dazu betrachte man nochmals das Einstiegsbeispiel.
Wie schon gesagt, ist die Position-Constraint-Funktion in diesem Fall

$$C(\boldsymbol{x}) = \|\boldsymbol{x}\| - 1$$

Die Velocity-Constraint wäre bei diesem Beispiel trivial. Die Geschwindigkeit des Teilchens muss tangential zu dem Einheitskreis oder orthogonal zu dem Ortsvektor des Teilchens verlaufen. Dasselbe Ergebnis bringt auch die Velocity-Constraint-Funktion, wenn man $C(\boldsymbol{x})$ zeitlich ableitet.

$$\begin{aligned}
\dot{C}(\boldsymbol{x}) &= \frac{d}{dt}\left(\sqrt{x_1^2 + x_2^2} - 1\right) \\
&= \frac{d}{dt}\left((x_1^2 + x_2^2)^{\frac{1}{2}} - 1\right) \\
&= \frac{1}{2\sqrt{{x_1}^2 + {x_2}^2}} \frac{d}{dt}(x_1^2 + x_2^2) - \frac{d}{dt}1 \\
&= \frac{2x_1 v_1 + 2x_2 v_2}{2\sqrt{x_1^2 + x_2^2}} - 0 \\
&= \frac{\boldsymbol{x} \cdot \boldsymbol{v}}{\|\boldsymbol{x}\|} = \frac{\boldsymbol{x}^T}{\|\boldsymbol{x}\|}\boldsymbol{v}
\end{aligned}$$

Nach dieser Gleichung wäre die Jacobi-Matrix $\boldsymbol{J} = \frac{\boldsymbol{x}^T}{\|\boldsymbol{x}\|}$.

Nun nehmen wir an, dass sich das Teilchen mit der Masse $m = 1\text{kg}$ auf der Position $\boldsymbol{x} = \binom{1}{0}\text{m}$ befindet und es eine Geschwindigkeit von $\boldsymbol{v} = \binom{0.3}{2}\text{m/s}$ hat. Die Position ist erlaubt, doch ist die Geschwindigkeit um 0.3m/s zu viel in Richtung der x-Achse. Der Constraint-Impuls müsste in diesem Fall 0.3kgm/s gegen die x-Achse betragen.
Wie in 4.3 beschrieben ist $\boldsymbol{p_c}$

$$\begin{aligned}
\boldsymbol{p}_c &= -\boldsymbol{J}^T m \frac{\boldsymbol{J}\boldsymbol{v}_0}{\boldsymbol{J}\boldsymbol{J}^T} \\
&= -\binom{1}{0} \cdot 1 \cdot \frac{(1 \quad 0)\binom{0.3}{2}}{(1 \quad 0)\binom{1}{0}} \\
&= -\binom{1}{0}\frac{0.3}{1} \\
&= \binom{-0.3}{0}
\end{aligned}$$

Der errechnete Constraint-Impuls entspricht dem oben vermuteten Wert.
Das System der Constraint-Impulse mag vielleicht etwas abstrakt erscheinen, vor allem in Fällen wie diesem, wo die Antwort trivial ist. Doch lassen sich somit alle Constraints auf die gleiche Art lösen, auch Constraints, bei denen der Korrekturimpuls nicht trivial erscheint.

4.5. Constraints mit Festkörpern

Vorherige Abschnitte haben sich auf Constraints mit Teilchen bezogen. Für Festkörper gelten diese Constraints nur in gewissen Maßen. Grund dafür ist, dass sich eine Randbedingung immer auf die

Position ($\boldsymbol{x}$) und auf die Geschwindigkeit eines Punktes ($\boldsymbol{v}$) des Körpers bezieht und nicht auf die Geschwindigkeit des Körpers selbst.
Für die Position-Constraint ändert dies nichts, denn die Position des Teilchens wird mit dem Punkt ersetzt, doch gibt es Probleme mit der Velocity-Constraint. Anders als bei einem Teilchen, wird die Geschwindigkeit des Punktes durch mehrere Parameter bestimmt. Sie entspricht der linearen Geschwindigkeit des Schwerpunktes ($\boldsymbol{v_s}$) plus der Winkelgeschwindigkeit ($\boldsymbol{\omega}$) des Körpers kreuz der relativen Distanz des Punktes zum Schwerpunkt ($\boldsymbol{r}$). Anders ausgedrückt:

$$\boldsymbol{v} = \boldsymbol{v_s} + \boldsymbol{\omega} \times \boldsymbol{r}$$

Die Velocity-Constraint bezieht sich weiterhin auf die Geschwindigkeit des Punktes, doch stellt die Korrektur ein Problem dar: der Constraint-Impuls muss gleichzeitig die Lineargeschwindigkeit und die Winkelgeschwindigkeit beeinflussen. Man könnte nun meinen, dass dies gar kein Problem darstellen sollte, denn ein Impuls beeinflusst nun mal diese beiden Größen, wenn er auf den gegeben Punkt wirkt. Wo ist also das Problem?
Wenn ein Impuls ($\boldsymbol{p}$) auf einen Punkt des Körpers einwirkt, ändert er Lineargeschwindigkeit und Winkelgeschwindigkeit wie folgt:

$$\boldsymbol{v_s} = \frac{\boldsymbol{p}}{m}$$
$$\boldsymbol{\omega} = \frac{\boldsymbol{r} \times \boldsymbol{p}}{I}$$

Wobei m die Masse des Körpers ist und I sein Trägheitsmoment. Der Constraint-Impuls ist nach 4.3 aber

$$\boldsymbol{v}_1 = \boldsymbol{v}_0 + \frac{\boldsymbol{p}_c}{m}$$
$$\boldsymbol{p_c} = m(\boldsymbol{v}_1 - \boldsymbol{v}_0)$$
$$\boldsymbol{p_c} = m\Delta\boldsymbol{v}$$

Wenn sich die Randbedingung auf den Schwerpunkt des Körpers bezieht, gibt es kein Problem, doch sobald sie sich auf einen anderen Punkt bezieht, wird der Trägheitsmoment vernachlässigt.
Man müsste also einen anderen Wert als die Masse des Körpers für m einsetzten. Da es aber nur eine Gleichung gibt mit mehreren Unbekannten (m und $\boldsymbol{v}_1$), ist sie unlösbar.
Eine Alternative trotzdem einen Constraint-Impuls zu finden, ist den Körper zwei Randbedingungen zu unterwerfen:

1. Eine, die die Lineargeschwindigkeit des Schwerpunktes einschränkt.
2. Eine, die die Winkelgeschwindigkeit des Körpers einschränkt.

Diese geben dann zusammen die gewünschte Velocity-Constraint auf den Punkt des Körpers. Die erste Schwierigkeit besteht darin, eine Lineargeschwindigkeitsrandbedingung und Winkelgeschwindigkeitsrandbedingung aus einer bekannten Velocity-Constraint zu isolieren. Dazu stellt man als erstes, wie vorhin beschrieben, die Velocity-Constraint-Funktion (mit der Punktgeschwindigkeit) auf:

$$\dot{C}(\boldsymbol{x}) = \boldsymbol{J}\boldsymbol{v}$$

Nachdem $\boldsymbol{J}$ ermittelt wurde, kommt der Knackpunkt. Anstatt mit $\boldsymbol{v}$ weiter zu arbeiten, wird diese mit einem anderen Vektor $\boldsymbol{V}$ ersetzt. Die Komponenten dieses Vektors entsprechen jeweils der Lineargeschwindigkeit und der Winkelgeschwindigkeit des Körpers.

$$\dot{C}(\boldsymbol{x}) = \boldsymbol{J}\boldsymbol{V} = \boldsymbol{J}\begin{pmatrix}\boldsymbol{v_s}\\ \boldsymbol{\omega}\end{pmatrix}$$

Die Jacobi-Matrix $\boldsymbol{J}$ hat nach dieser Transformation auch einen anderen Wert, sie besteht nun aus zwei Reihenvektoren, so dass

$$\begin{pmatrix}\boldsymbol{J}_{v_s} & \boldsymbol{J}_{\omega}\end{pmatrix}\begin{pmatrix}\boldsymbol{v_s}\\ \boldsymbol{\omega}\end{pmatrix} = 0$$

wenn die Velocity-Constraint eingehalten ist. Im Ganzen wurde also die Punktgeschwindigkeit in die Lineargeschwindigkeit und Winkelgeschwindigkeit des Körpers zerlegt.
Der Constraint-Impuls wird auch mit einer Matrix ($\boldsymbol{P_c}$) aus Impuls ($\boldsymbol{p}$) und Drehimpuls ($\boldsymbol{L}$), ersetzt denn dieser beeinflusst dann $\boldsymbol{v_s}$ und $\boldsymbol{\omega}$ getrennt:

$$\begin{bmatrix} \boldsymbol{p} = m\boldsymbol{v} \\ \boldsymbol{L} = I\boldsymbol{\omega} \end{bmatrix}$$
$$\begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{L} \end{pmatrix} = \begin{pmatrix} m & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \boldsymbol{v_s} \\ \boldsymbol{\omega} \end{pmatrix}$$
$$\boldsymbol{P} = \boldsymbol{MV}$$

Hier ist auch zu erkennen, dass die Masse aus der gewöhnlichen $\boldsymbol{p} = m\boldsymbol{v}$ Formel mit einer Matrix ersetzt wird.
Erstaunlicherweise waren dies die einzigen Schwierigkeiten. Der Rest der Geschwindigkeitskorrektur kann mit kleinen Anpassungen fortgeführt werden. Alle Größen müssen nur als Vektor oder Matrix betrachtet werden, damit am Ende $\boldsymbol{v_s}$ und $\boldsymbol{\omega}$ getrennt beeinflusst werden.
Die Constraint-Impuls-Formeln aus 4.3 sind also folgende:

1. Die gleichförmig beschleunigte Bewegung wird gegeben durch

$$\boldsymbol{V}_1 = \boldsymbol{V}_0 + \boldsymbol{M^{-1}P}$$
$$\begin{pmatrix} \boldsymbol{v_{s,1}} \\ \boldsymbol{\omega_1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{v_{s,0}} \\ \boldsymbol{\omega_0} \end{pmatrix} + \begin{pmatrix} \frac{1}{m} & 0 \\ 0 & \frac{1}{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{L} \end{pmatrix}$$

2. Nach der Impulseinwirkung soll die Velocity-Constraint wieder stimmen

$$\dot{C}(\boldsymbol{x}) = 0$$
$$\boldsymbol{JV}_1 = 0$$
$$\boldsymbol{J}\left(\boldsymbol{V}_0 + \boldsymbol{M^{-1}P}\right) = 0$$
$$\boldsymbol{J}\left(\boldsymbol{V}_0 + \boldsymbol{M^{-1}J}^T\lambda\right) = 0$$

3. Durch Isolierung von λ kann der Constraint-Impuls ermittelt werden

$$\boldsymbol{JM^{-1}}(\boldsymbol{J}^T\lambda) = -\boldsymbol{JV}_0$$
$$\lambda = -\frac{\boldsymbol{JV}_0}{\boldsymbol{JM^{-1}J}^T} = -\frac{\dot{C}(\boldsymbol{x})}{\boldsymbol{JM^{-1}J}^T}$$
$$\boxed{\boldsymbol{P}_c = -\boldsymbol{J}^T\frac{\boldsymbol{JV}_0}{\boldsymbol{JM^{-1}J}^T}}$$

Der Constraint-Impuls beeinflusst also Lineargeschwindigkeit und Winkelgeschwindigkeit getrennt, jedoch in einer Weise, dass die Punktgeschwindigkeit $\boldsymbol{v_s} + \boldsymbol{\omega} \times \boldsymbol{r}$ der Velocity-Constraint konform bleibt.

4.6. Bezug auf Beispiel

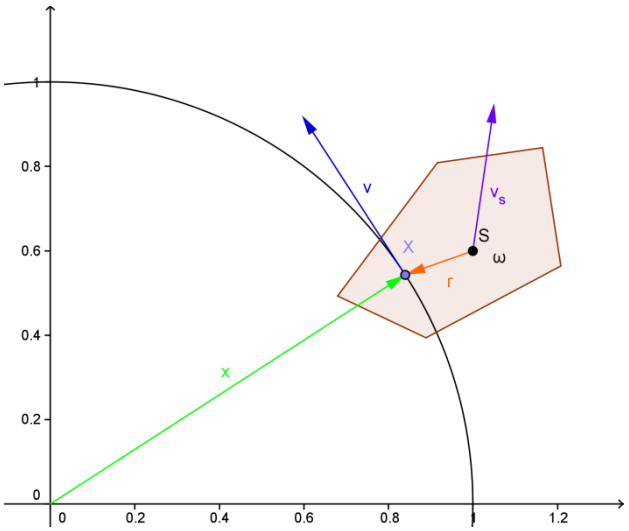


Abbildung 4: Körper auf Einheitskreis

Nehmen wir als konkreten Fall nochmals das Einstiegsbeispiel mit einigen Änderungen.

1. Die Position des Teilchens wird mit der Position eines Punktes ($\boldsymbol{x}$) auf dem Körper ersetzt.
2. Dieser hat die Geschwindigkeit $\boldsymbol{v}$.
3. Er befindet sich $\boldsymbol{r}$ relativ zu dem Schwerpunkt ($\boldsymbol{S}$) des Körpers.

Die Randbedingung besagt nun, dass der gegebene Punkt sich immer auf dem Einheitskreis befinden soll. Die Position- und Velocity-Constraints sind wie folgt:

$$C(\boldsymbol{x}) = \|\boldsymbol{x}\| - 1$$

$$\dot{C}(\boldsymbol{x}) = \frac{\boldsymbol{x}^T}{\|\boldsymbol{x}\|}\boldsymbol{v} = \boldsymbol{u}\cdot\boldsymbol{v} = \boldsymbol{u}\cdot\boldsymbol{v_s} + \boldsymbol{u}\cdot(\boldsymbol{\omega}\times\boldsymbol{r})$$

Nach der Isolierung von $\boldsymbol{v_s}$ und $\boldsymbol{\omega}$ ergibt sich:

$$\dot{C}(\boldsymbol{x}) = \boldsymbol{JV}$$

$$\dot{C}(\boldsymbol{x}) = \begin{pmatrix}\boldsymbol{u} & \boldsymbol{r}\times\boldsymbol{u}\end{pmatrix}\begin{pmatrix}\boldsymbol{v_s}\\ \boldsymbol{\omega}\end{pmatrix}$$

Der Constraint-Impuls ist in diesem Fall:

$$\boldsymbol{P} = \boldsymbol{J}^T\lambda = -\begin{pmatrix}\boldsymbol{u}\\ \boldsymbol{r}\times\boldsymbol{u}\end{pmatrix}\frac{\begin{pmatrix}\boldsymbol{u} & \boldsymbol{r}\times\boldsymbol{u}\end{pmatrix}\begin{pmatrix}\boldsymbol{v_{s,0}}\\ \boldsymbol{\omega_0}\end{pmatrix}}{\begin{pmatrix}\boldsymbol{u} & \boldsymbol{r}\times\boldsymbol{u}\end{pmatrix}\begin{pmatrix}m^{-1} & 0\\ 0 & I^{-1}\end{pmatrix}\begin{pmatrix}\boldsymbol{u}\\ \boldsymbol{r}\times\boldsymbol{u}\end{pmatrix}}$$

$$\boldsymbol{P} = -\begin{pmatrix}\boldsymbol{u}\\ \boldsymbol{r}\times\boldsymbol{u}\end{pmatrix}\frac{\boldsymbol{v_{s,0}}\cdot\boldsymbol{u} + \boldsymbol{\omega_0}\cdot\boldsymbol{r}\times\boldsymbol{u}}{\frac{1}{m}\boldsymbol{u^2} + \frac{1}{I}(\boldsymbol{r}\times\boldsymbol{u})^2}$$

mit

$$\lambda = -\frac{\boldsymbol{v_{s,0}}\cdot\boldsymbol{u} + \boldsymbol{\omega_0}\cdot\boldsymbol{r}\times\boldsymbol{u}}{\frac{1}{m}\boldsymbol{u^2} + \frac{1}{I}(\boldsymbol{r}\times\boldsymbol{u})^2}$$

Also werden die Geschwindigkeiten folgend korrigiert:

$$\begin{pmatrix}\boldsymbol{v_{s,1}}\\ \boldsymbol{\omega_1}\end{pmatrix} = \begin{pmatrix}\boldsymbol{v_{s,0}}\\ \boldsymbol{\omega_0}\end{pmatrix} + \begin{pmatrix}\lambda\frac{\boldsymbol{u}}{m}\\ \lambda\frac{\boldsymbol{r}\times\boldsymbol{u}}{I}\end{pmatrix}$$

Als Zahlenbeispiel nehmen wir einmal an, der Punkt befindet sich auf der Position $\boldsymbol{x} = \begin{pmatrix}1\\0\end{pmatrix}$m. Der Vektor vom Schwerpunkt des Körpers zu diesem Punkt $\boldsymbol{r} = \begin{pmatrix}0.1\\0.2\end{pmatrix}$m. Der Körper hat eine Lineargeschwindigkeit von $\boldsymbol{v_s} = \begin{pmatrix}0.3\\2\end{pmatrix}$m/s und eine Winkelgeschwindigkeit $\omega = 2$rad/s. Die

Geschwindigkeit des Punktes beträgt also $\boldsymbol{v_0} = \begin{pmatrix} -0.1 \\ 2.2 \end{pmatrix}$m/s. Der Körper hat dazu eine Masse von 1kg und ein Trägheitsmoment von 1kgm^2.
Nach etwas rechnen ergibt sich:

$$\lambda = -\frac{\boldsymbol{JV}}{\boldsymbol{JM^{-1}J^T}} = -\frac{-0.1}{\frac{1}{1}+\frac{1}{1}\cdot 0.04} = 0.096$$

Die Geschwindigkeiten nach der Korrektur sind:

$$\boldsymbol{v_{s,1}} = \boldsymbol{v}_{s,0} + \frac{\lambda}{m}\boldsymbol{u} = \begin{pmatrix} 0.3 \\ 2 \end{pmatrix} + \frac{0.096}{1}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.396 \\ 2 \end{pmatrix}$$

$$\boldsymbol{\omega}_1 = \boldsymbol{\omega}_0 + \frac{\lambda}{I}\boldsymbol{r} \times \boldsymbol{u} \equiv 2 + \frac{0.096}{1}\cdot(-0.2) = 1.98$$

Die Geschwindigkeit des Punktes beträgt nach der Korrektur $\boldsymbol{v_{s,1}} + \boldsymbol{\omega_1} \times \boldsymbol{r} = \begin{pmatrix} 0 \\ 2.198 \end{pmatrix}$. Die erste Komponente ist null, d. h. der Punkt hat wieder eine legale Geschwindigkeit.

4.7. Constraints mit mehreren Körpern

Bis jetzt wurden nur Constraints mit einem Körper betrachtet. Meistens sind Körper jedoch nicht an statische Constraints gebunden, wie z.B. der unbewegliche Einheitskreis, sondern an Randbedingungen zwischen zwei Körpern. Das Einstiegsbeispiel auf zwei Körper bezogen wäre z. B.: Körper 1 darf nicht näher als 1m an Körper 2 heran kommen.
Dies bedeutet dann auch, dass die Velocity-Constraint nicht mehr mit absoluten Geschwindigkeiten arbeiten kann, sondern nur mit relativen. Diese werden aus dem Unterschied der betrachteten Geschwindigkeiten der Punkte errechnet. Wie aber in vorherigem Abschnitt beschrieben, muss diese Geschwindigkeit in Lineargeschwindigkeit und Winkelgeschwindigkeit der Körper zerlegt werden. Das Gleichungssystem von 4.5 muss also geändert werden: (Indizes geben den Körper an)

- $\boldsymbol{V}$ besteht nun aus den Lineargeschwindigkeiten und Winkelgeschwindigkeiten von Körper eins und zwei.

$$\boldsymbol{V} = \begin{pmatrix} \boldsymbol{v_{s,1}} \\ \boldsymbol{\omega_1} \\ \boldsymbol{v_{s,2}} \\ \boldsymbol{\omega_2} \end{pmatrix}$$

- Dem entsprechend, erhält $\boldsymbol{J}$ zwei Komponenten dazu

$$\boldsymbol{J} = (\boldsymbol{J_{v_{s,1}}} \quad \boldsymbol{J_{\omega_1}} \quad \boldsymbol{J_{v_{s,2}}} \quad \boldsymbol{J_{\omega_2}})$$

- $\boldsymbol{M}$ ist eine Diagonalmatrix bestehend aus den Geschwindigkeiten und Trägheitsmomenten der Körper

$$\boldsymbol{M} = \begin{pmatrix} m_1 & 0 & 0 & 0 \\ 0 & I_1 & 0 & 0 \\ 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & I_2 \end{pmatrix}$$

- $\boldsymbol{P}$ enthält zwei Dimensionen dazu. Diese entsprechen dem Impuls und Drehimpuls auf den zweiten Körper

$$\boldsymbol{P} = \begin{pmatrix} \boldsymbol{p}_1 \\ \boldsymbol{L}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{L}_2 \end{pmatrix}$$

Danach kann die Geschwindigkeitskorrektur der Körper wie gewohnt folgen.

4.8. Positionsfehler und Positionskorrektur

Theoretisch dürften sich Körper und Teilchen, nachdem ihre Geschwindigkeit korrigiert wurde, nur auf erlaubten Positionen fortbewegen. In einer Simulation läuft die Zeit jedoch nicht kontinuierlich ab, sondern wird in Zeitschritten berechnet. Zwischen diesen Schritten ändert sich die Position der Körper und Teilchen linear und proportional zu ihrer Geschwindigkeit. Hinzu kommen numerische Fehler bei der Geschwindigkeitskorrektur. All dies führt dazu, dass Körper und Teilchen mit der Zeit auf Positionen geraten, die der Position-Constraint nicht entsprechen.
Ähnlich wie bei der Velocity-Constraint muss nun auch die Position korrigiert werden, um eine realistische Simulation zu bewahren. Die Positionskorrektur verläuft fast so wie die Geschwindigkeitskorrektur.
Zunächst wird die erlaubte Geschwindigkeit mit der Positionsabweichung ersetzt. Der Korrekturimpuls wird dann mit derselben Jacobi-Matrix aus der Velocity-Constraint errechnet. Der Impuls leistet in diesem Fall aber Arbeit, denn er ist parallel zu der neuen erlaubten Geschwindigkeit. Dies ist jedoch nur gerechtfertigt, schließlich wird der Körper auf eine andere Laufbahn gebracht. Um die Position zu korrigieren, wirkt sich der Impuls aber nicht auf die Geschwindigkeit aus, sondern direkt auf die Position des Körpers. Es gilt nämlich, dass in einer Zeit h der Positionsunterschied $\Delta \boldsymbol{x}$ durch folgende Formel gegeben wird

$$\Delta \boldsymbol{x} = h\boldsymbol{v}$$

Auf den Körper wird der Korrekturimpuls während einer Zeit wirken. Dieser Impuls hat zur Folge, dass der Körper eine Geschwindigkeit entsprechend der Positionsabweichung bekommt. Durch oben genannte Formel erreicht diese Geschwindigkeit über die Zeit die gewünschte Positionsverschiebung des Körpers. Seine neue Position ist dann wieder mit der Positionsrandbedingung konform. Da der Körper nach dem Impuls aber nicht sofort wieder auf unerlaubte Positionen geraten soll, müsste seine, zu dem Impuls parallele, Geschwindigkeit wieder auf null gesetzt werden. Deswegen, um dies zu umgehen, beeinflusst der Impuls bei der Positionskorrektur direkt die Position des Körpers.

Es gibt noch zwei weitere Methoden, numerische Fehler so klein wie möglich zu machen und somit größere Korrekturen zu vermeiden.
Die erste besteht darin, die Dauer der Zeitschritte zu reduzieren. Dadurch ändert sich die Position der Körper nicht so viel, während sie sich nach der Geschwindigkeitskorrektur linear fortbewegen. Diese Methode ist sehr effektiv, doch hat sie eine große Nebenwirkung: wenn Zeitschritte klein werden, braucht die Simulation während einer realen Zeit viel länger als mit größeren Zeitschritten. Das heißt, wenn die Zeitschritte zu klein werden, verläuft die ganze Simulation je nach Rechenleistung des Computers auch viel langsamer.
Die zweite Methode kostet zwar auch Rechenzeit, doch nicht so viel wie die Zeitschrittreduktion. Sie besteht lediglich darin, alle Constraints mehrmals pro Zeitschritt zu korrigieren. So werden numerische Fehler klein und die Simulation verläuft realistischer. Die Anzahl dieser Korrekturen wird in folgenden Abschnitten auch Anzahl an Iterationen pro Zeitschritt oder nur Iterationen genannt.

4.9. Implementierung in SiMS

Das Trait[12] `dynamics.Constraint` enthält zwei abstrakte Methoden `correctVelocity(h: Double): Unit` und `correctPosition(h: Double): Unit` die zur Korrektur der Geschwindigkeit bzw. der Position dienen. Jede Art von Objekt, das eine Randbedingung darstellt, erbt dieses Trait. In dem Objekt findet dann die konkrete Implementierung der Methoden statt. Die

[12] *Traits* in Scala sind ähnlich wie *Interfaces* in Java. Im Gegensatz zu letzteren können Traits abstrakte als auch konkrete Felder und Methoden enthalten.

Methoden nehmen als einzigen Parameter `h`, die Zeitschrittsdauer, durch die sich Impulse in Kräfte umwandeln lassen und umgekehrt. Die Objekte müssen also andere Möglichkeiten haben Körper zu referenzieren. Typischerweise sind diese Methoden in Klassen vorhanden, die selbst Variablen mit Referenzen zu diesen Körpern besitzen.
In SiMS werden Constraints von zwei Arten von Objekten implementiert, von Verbindungen (`Joint`) und Kollisionen (`Collision`)[13]. Diese sind abstrakt und werden von konkreten Klassen geerbt.

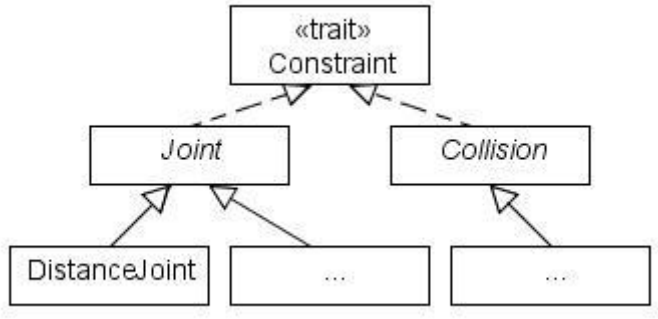


Abbildung 5: Constraints Klassenhierarchie

Ein Beispiel wäre die Klasse `joints.DistanceJoint`. Sie ist eine Verbindung die zwei Punkten zweier Körper der Randbedingung unterwirft, immer einen gegebenen Abstand zu halten. Die Punkte sind in den Feldern `connection1` und `connection2` enthalten. Diese Punkte befinden sich auf den Körpern `node1` und `node2`.

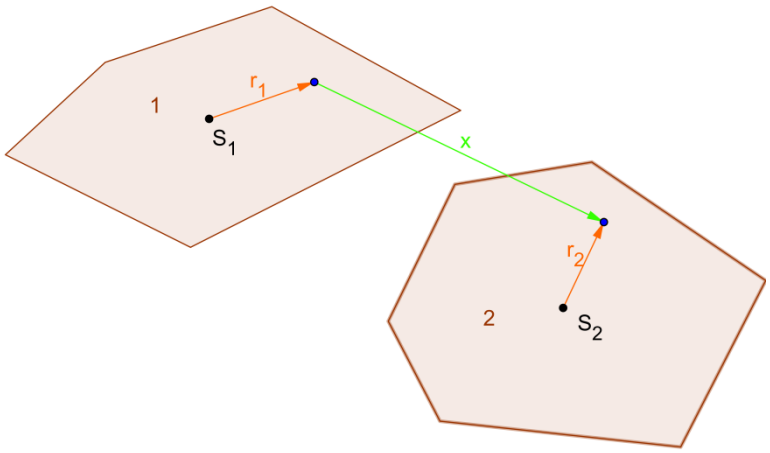


Abbildung 6: DistanceJoint

Wie in vorherigen Beispielen gesehen, muss die relative Geschwindigkeit immer orthogonal zu der relativen Position der Punkte sein. Es ergibt sich:

$$\dot{C}(\boldsymbol{x}) = \frac{\boldsymbol{x}^T}{\|\boldsymbol{x}\|}\boldsymbol{v} = \boldsymbol{u}\cdot\left(\boldsymbol{v}_{s,2} + \boldsymbol{\omega}_2 \times \boldsymbol{r}_2 - \boldsymbol{v}_{s,1} - \boldsymbol{\omega}_1 \times \boldsymbol{r}_1\right) \Big| \boldsymbol{u} = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}$$
$$= (\boldsymbol{u} \quad \boldsymbol{r}_1 \times \boldsymbol{u} \quad -\boldsymbol{u} \quad -\boldsymbol{r}_2 \times \boldsymbol{u})\boldsymbol{V}$$
$$\Leftrightarrow \boldsymbol{J} = (\boldsymbol{u} \quad \boldsymbol{r}_1 \times \boldsymbol{u} \quad -\boldsymbol{u} \quad -\boldsymbol{r}_2 \times \boldsymbol{u})$$

Nun könnte man `correctVelocity` implementieren, sodass sie, wie beschrieben, den Korrekturimpuls ausrechnet und ihn auf die Geschwindigkeiten der Körper wirken lässt. Dies ist ein genereller Weg für den komplizierte Matrizenrechnungen nötig sind, der aber für jeden Constraint funktioniert.

[13] Kollisionen werden in Kapitel 5 beschrieben.

Doch bei dieser Randbedingung lassen sich auf den zweiten Blick nützliche Eigenschaften aus der Velocity-Constraint ziehen:

1. Die Korrekturimpulse auf den jeweiligen Körper werden den gleichen Betrag nur mit umgekehrtem Zeichen haben.
2. Die Korrekturimpulse beeinflussen Linear- und Winkelgeschwindigkeiten der Körper genauso wie ein Impulsvektor auf die den Randbedingungen unterliegenden Punkte.

Das heißt also, es muss nur *ein* Impulsvektor auf *einen* der Körper ausgerechnet werden.
In SiMS ist die Implementierung wie folgt:

```
  override def correctVelocity(h: Double) = {
    val x = this.x      //relativer Abstand
    val v = this.v      //relative Geschwindigkeit
    val r1 = (connection1 - node1.pos)//Abstand Punkt-Schwerpunkt,
Körper 1
    val r2 = (connection2 - node2.pos)    //Abstand Punkt-Schwerpunkt,
Körper 2
    val cr1 = r1 cross x.unit //Kreuzprodukt
    val cr2 = r2 cross x.unit //Kreuzprodukt

    val Cdot = x.unit dot v   //Velocity-Constraint
    val invMass = 1/node1.mass + 1/node1.I * cr1 * cr1 + 1/node2.mass +
1/node2.I * cr2 * cr2   //=J M^-1 JT
    val m = if (invMass == 0.0) 0.0 else 1/invMass   //Test um
Nulldivision zu vermeiden
    val lambda = -m * Cdot    //=-JV/(JM^(-1)JT)
    val impulse = x.unit * lambda   //P=J lambda
    node1.applyImpulse(-impulse, connection1)
    node2.applyImpulse(impulse, connection2)
  }
```

Schauen wir uns diese Methode genauer an. Mit dem Code

```
    val x = this.x
    val v = this.v
    val r1 = (connection1 - node1.pos)
    val r2 = (connection2 - node2.pos)
    val cr1 = r1 cross x.unit
    val cr2 = r2 cross x.unit
```

werden $\boldsymbol{x}$, $\boldsymbol{v}$, $\boldsymbol{r_1}$, $\boldsymbol{r_2}$, $\boldsymbol{r_1} \times \boldsymbol{u}$ und $\boldsymbol{r_2} \times \boldsymbol{u}$ definiert. Die relative Position und Geschwindigkeit sind in der umgebenden Klasse `DistanceJoint` definiert.

```
  def x = connection2 - connection1
  def v = node2.velocityOfPoint(connection2) -
          node1.velocityOfPoint(connection1)
```

Die Velocity-Constraint-Funktion $\dot{C}(\boldsymbol{x}) = \boldsymbol{JV} = \boldsymbol{u} \cdot \boldsymbol{v}$ wird gegeben durch

```
    val Cdot = x.unit dot v
```

Der Korrekturimpuls ist in der Implementierung etwas anders strukturiert als in der Theorie. Anstatt λ gleich auszurechnen, wird erst der Wert unter dem Bruchstrich $\boldsymbol{JM^{-1}J^T}$ ausgerechnet. Dieser ist gleich

$$\frac{1}{m_1}+\frac{1}{I_1}(\boldsymbol{r_1}\times\boldsymbol{u})^2+\frac{1}{m_2}+\frac{1}{I_2}(\boldsymbol{r_2}\times\boldsymbol{u})^2$$

```
val invMass = 1/node1.mass + 1/node1.I * cr1 * cr1 + 1/node2.mass +
              1/node2.I * cr2 * cr2
```

Erst danach wird $1/\left(\boldsymbol{JM^{-1}J^T}\right)$ ausgerechnet

```
val m = if (invMass == 0.0) 0.0 else 1/invMass
```

Diese Vorsichtsmaßnahme, erst den Teiler auszurechnen, dient lediglich dazu, eine Nulldivision zu vermeiden wenn beide Körper fixiert sind und daher unendlich große Massen haben. Sollte dies tatsächlich der Fall sein, ist eine Korrektur der Körper überflüssig und λ wird auf null gesetzt.
Lambda wird folgendermaßen errechnet:

```
val lambda = -m * Cdot
```

Dies entspricht

$$\lambda=-m\boldsymbol{JV}\;\Big|\;m=\frac{1}{\boldsymbol{JM^{-1}J^T}}$$

Nun wird der Korrekturimpuls in Form eines Vektors ausgerechnet und auf die Körper angewandt.

```
val impulse = x.unit * lambda
node1.applyImpulse(-impulse, connection1)
node2.applyImpulse(impulse, connection2)
```

Der Impulsvektor entspricht:

$$\boldsymbol{p_c}=\boldsymbol{u}\lambda$$

Dadurch, dass er auf je einen Punkt pro Körper einwirkt, entspricht er den Korrekturen, die die Impulsmatrix auf die verschiedenen Geschwindigkeiten hätte. (Erste Zahlenindizes entsprechen hier den Körpern. Die Endgeschwindigkeit wird mit einem Apostroph $\boldsymbol{v'}$ gekennzeichnet.)

Impulswirkung: $\boldsymbol{v'_{s,1}}=\boldsymbol{v_{s,1,0}}-\frac{\boldsymbol{u}}{m}\boldsymbol{\lambda}$ und $\boldsymbol{\omega'_1}=\boldsymbol{\omega_{1,0}}-\frac{\boldsymbol{r_1}\times\boldsymbol{u}}{I}\lambda$

$\boldsymbol{v'_{s,2}}=\boldsymbol{v_{s,2,0}}+\frac{\boldsymbol{u}}{m}\boldsymbol{\lambda}$ und $\boldsymbol{\omega'_2}=\boldsymbol{\omega_{1,0}}+\frac{\boldsymbol{r_2}\times\boldsymbol{u}}{I}\lambda$

Impulsmatrix: $\begin{pmatrix}\boldsymbol{v'_{s,1}}\\ \boldsymbol{\omega'_1}\\ \boldsymbol{v'_{s,2}}\\ \boldsymbol{\omega'_2}\end{pmatrix}=\boldsymbol{V_0}+\boldsymbol{J^T}\lambda=\begin{pmatrix}\boldsymbol{v_{s,1,0}}\\ \boldsymbol{\omega_{1,0}}\\ \boldsymbol{v_{s,2,0}}\\ \boldsymbol{\omega_{2,0}}\end{pmatrix}+\begin{pmatrix}-\lambda\frac{\boldsymbol{u}}{m}\\ -\lambda\frac{\boldsymbol{r_1}\times\boldsymbol{u}}{I}\\ \lambda\frac{\boldsymbol{u}}{m}\\ \lambda\frac{\boldsymbol{r_2}\times\boldsymbol{u}}{I}\end{pmatrix}$

Somit erfolgt die Implementierung bei diesem Beispiel ohne Matrizenrechnungen. Dennoch können die Geschwindigkeiten im Allgemeinen auch separat mit der Impulsmatrix korrigiert werden.

4.10. Simulieren der Constraints

Um Constraints in der Simulation zu berücksichtigen, müssen sie auch in der Welt vorhanden sein. Um Verbindungen mit einzubeziehen, enthält `World` ein Feld namens `joints`, das eine Sammlung von Verbindungen beschreibt.

```
val joints = new ArrayBuffer[Joint]
```

Die andere Art von Constraints, Kollisionen, werden durch die Klasse `Detector` ermittelt. Jede Welt enthält ein Feld namens `detector`, dass den Zugriff auf Kollisionen in der Welt ermöglicht. Dies wird genauer im nächsten Kapitel beschrieben. Abbildung 7 zeigt die Herkunft von Constraints in einer Welt.

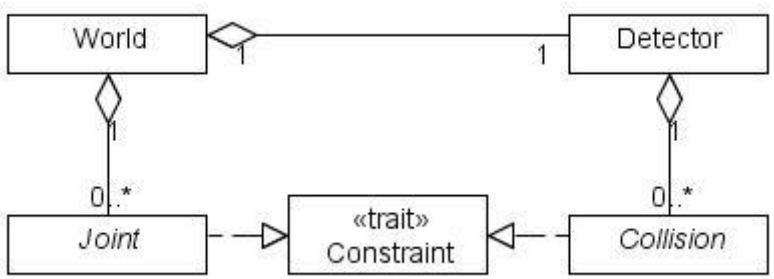


Abbildung 7: Constraints in SiMS

Der Zeitschritt der Welt wird nun auch noch um zwei zusätzliche Phasen ergänzt:
Die erste Ergänzung, die Geschwindigkeitskorrektur nach Phase 2, besteht darin, die Geschwindigkeiten der Körper zu korrigieren. Dafür wird die Methode `correctVelocity` von jedem Constraint aufgerufen.
Die zweite Ergänzung, die Positionskorrektur nach Phase 3, besteht darin, die Positionen der Körper zu korrigieren. Dafür wird die Methode `correctPosition` von jedem Constraint aufgerufen.

```
def step() = {
//Kräfte {...}

//Integration der Beschleunigung {...}

// Geschwindigkeitskorrektur
   for (i <- 0 until iterations){
       for(c <- joints) c.correctVelocity(timeStep)
       if (enableCollisionDetection)
          for (c <- detector.collisions) c.correctVelocity(timeStep)
   }

//Integration der Geschwindigkeit {...}

//Positionskorrektur
   if (enablePositionCorrection) for (i <- 0 until iterations){
      for (c <- joints) c.correctPosition(timeStep)
      if (enableCollisionDetection)
         for (c <- detector.collisions) c.correctPosition(timeStep)
   }
}
```

Alle Korrekturen werden um `iterations` mal wiederholt. Dies sind die Iterationen aus Abschnitt 4.8, die die Präzision der Korrekturen erhöhen. Der Parameter, mit dem die Korrekturmethoden aufgerufen werden, entspricht der aktuellen Zeitschrittsdauer der Welt. Die Felder `enableCollisionDetection` und `enablePositionCorrection` geben an, ob Kollisionen berücksichtigt werden sollen, bzw. ob Positionen korrigiert werden sollen. Sie können vom Benutzer geändert werden und dienen hauptsächlich Versuchszwecken.

4.11. Schluss

Constraints sind eine wichtige Komponente eines realistischen Physiksimulators. Sie ermöglichen das Simulieren von Körpern, deren Bewegungen gewissen Randbedingungen unterliegen. In diesem Kapitel wurde erklärt, wie sie modelliert, in der Simulation berücksichtigt und in SiMS implementiert werden. Ohne sie könnten sich Körper nur frei nach den Newtonschen Gesetzen bewegen, nicht einmal Kollisionen wären möglich. Mehr dazu im nächsten Kapitel.

5. Kollisionen

Kollisionen entstehen, wenn Körper auf einander treffen. In einem Mechaniksimulator müssen solche Ereignisse erkannt und in der Simulation berücksichtigt werden. Dazu werden Algorithmen benutzt, die es ermöglichen, Kollisionen zu erkennen und entsprechend zu handeln. Kollisionserkennung ist ein sehr spezialisierter Teil der Physiksimulation, man könnte fast von einer Wissenschaft für sich reden. Daher wird in diesem Kapitel nur eine einfache Variante von Kollisionserkennung beschrieben, die auch in SiMS implementiert ist. Viele geometrische Konzepte und ihre Implementierung werden auch nicht erläutert. Sie können genauer in dem Quellcode nachgelesen werden.

5.1. Kollisionen als Constraints

Entsteht eine Kollision in der Wirklichkeit, prallen die kollidierenden Körper, je nach ihren materiellen Eigenschaften, Geschwindigkeiten und Massen, mit niedriger oder höherer Geschwindigkeit ab oder bleiben aufeinander liegen. Auf keinen Fall aber durchdringen sie sich wie ein Geist, der durch eine Wand schwebt. Ihre Bewegung ist also bei einem Aufprall eingeschränkt. Eine eingeschränkte Bewegung entspricht auch einer Bewegung, die gewissen Randbedingungen unterworfen ist. Kollisionen können also als Constraints betrachtet werden, die Körpern verbieten, sich zu durchdringen.

Wie aber stellt man so eine Randbedingung dar? Dazu muss man als erstes eine Kollision zwischen Körpern genauer betrachten[14].

- Um Constraints so einfach wie möglich zu halten, werden Kollisionen als binär betrachtet. Sie entstehen immer zwischen zwei Körpern.
- Einer der beiden Körper wird als Referenzkörper betrachtet, der andere als eindringender Körper.
- Bei einer Kollision entstehen ein oder mehrere Kontaktpunkte zwischen den zwei Körpern. Diese Kontaktpunkte befinden sich immer auf der Oberfläche des Referenzkörpers. In zwei Dimensionen entspricht diese seinen Konturen.

Die Randbedingung besagt nun, dass die beiden Körper sich nicht durchdringen dürfen. Die Geschwindigkeit der Kontaktpunkte muss also korrigiert werden, so dass die Punkte ebenfalls nicht in die Körper eindringen.

[14] Diese Art Kollisionen darzustellen, wurde nach den Konzepten von (Collision Detection kein Datum) und (Catto, Contact Manifolds 2007) übernommen.

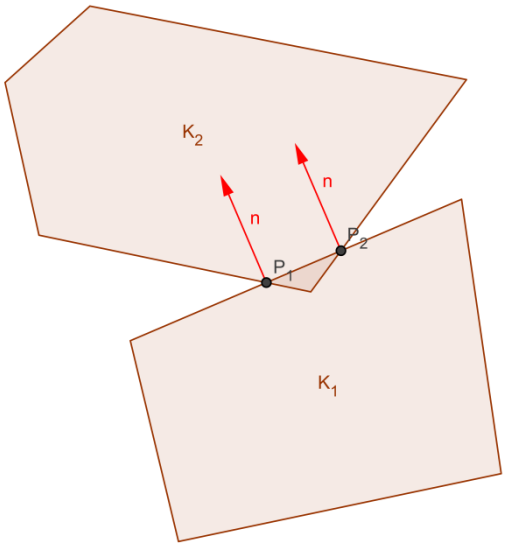


Abbildung 8: Kollision zwischen zwei Polygonen

Damit ein Kontaktpunkt (P_i) auf der Oberfläche des Referenzkörpers (K_1) bleibt, muss dessen Geschwindigkeit parallel zu letzterer sein. Anders gesagt muss dessen Geschwindigkeit ($\boldsymbol{v}$) orthogonal zu dem Normalenvektor[15] ($\boldsymbol{n}$) der Oberfläche bei seiner Position ($\boldsymbol{x}$) sein. Seine Velocity-Constraint wäre also:

$$\dot{C}(\boldsymbol{x}) = \boldsymbol{n} \cdot \boldsymbol{v} = 0$$

Die Geschwindigkeit eines Kontaktpunktes muss jedoch nicht immer zwingend auf der Oberfläche verlaufen, sondern soll gemäß den Gesetzen bei einer Kollision abprallen. Wenn c eine durch die materiellen Eigenschaften des Körpers ermittelte Stoßzahl ist[16], dann ist die Velocity-Constraint:

$$\dot{C}(\boldsymbol{x}) = \boldsymbol{n} \cdot \boldsymbol{v}\,(1 + c) = 0$$

Die Kontaktpunkte dürfen sich jedoch auch schneller entfernen als die erhaltene Geschwindigkeit durch den Rückstoß. Dies passiert z. B. Wenn eine Feder noch zusätzlich an einem Körper zieht. Berücksichtigt man dies in der Velocity-Constraint, so erhält man:

$$\dot{C}(\boldsymbol{x}) = \boldsymbol{n} \cdot \boldsymbol{v}\,(1 + c) \geq 0$$

In der Praxis erfolgt die Korrektur eines solchen „ungleichen“ Constraints wie die eines normalen Constraints, d. h. die Geschwindigkeit wird so korrigiert, dass die Constraint-Funktion gleich null ist. Der einzige Unterschied ist, dass die Korrektur nur erfolgt, wenn die Constraint-Funktion kleiner als null ist.

Um die Velocity-Constraint nach den Geschwindigkeiten der Körper zu berechnen, wird die Geschwindigkeit eines Kontaktpunktes gegeben durch:

$$\boldsymbol{v} = \boldsymbol{v}_{s,2} + \boldsymbol{\omega}_2 \times \boldsymbol{r}_2 - \boldsymbol{v}_{s,1} - \boldsymbol{\omega}_1 \times \boldsymbol{r}_1$$

Die endgültige Velocity-Constraint einer Kollision ist demnach:

$$\dot{C}(\boldsymbol{x}) = \boldsymbol{n} \cdot \left[\left(\boldsymbol{v}_{s,2} + \boldsymbol{\omega}_2 \times \boldsymbol{r}_2 - \boldsymbol{v}_{s,1} - \boldsymbol{\omega}_1 \times \boldsymbol{r}_1\right)(1 + c)\right] \geq 0$$

Diese Randbedingung muss für jeden Kontaktpunkt aufgestellt werden und die Geschwindigkeiten der Körper müssen korrigiert werden, um Kollisionen in der Simulation zu berücksichtigen.

[15] Die Richtung des Normalenvektors ist gemäß Definition immer vom Referenzkörper abgewandt.
[16] $c = 0$ bei einer inelastischen Kollision, und $c = 1$ bei einer elastischen Kollision

5.2. Kollisionen in SiMS[17]

In SiMS werden Kollisionen mit der abstrakten Klasse `Collision` dargestellt. Sie erbt von dem Trait `Constraint` und enthält somit, wie in 4.9 erwähnt, Methoden zu den Randbedingungen einer Kollision. Um diese Randbedingungen darstellen zu können, braucht eine Kollision die im letzten Abschnitt genannten Informationen. Darum enthält sie folgende Felder:

- Da eine Kollision aufgrund der räumlichen Struktur der kollidierenden Körper entsteht, benötigt sie Referenzen zu den kollidierenden Formen

```
val shape1: Shape
val shape2: Shape
```

  Hierbei wird `shape1` als Referenzform betrachtet und `shape2` als eindringende.

- Die Kollisionspunkte werden gegeben durch

```
val points: Iterable[Vector2D]
```

- Schließlich wird einfachheitshalber nur eine Normale für alle Kollisionspunkte gegeben

```
val normal: Vector2D
```

Da Kollisionspunkte und Normale je nach Art der kollidierenden Formen ermittelt werden, ist `Collision` abstrakt deklariert. Nichtsdestotrotz, reichen ihre abstrakten Felder aus, um die Methoden `correctVelocity` und `correctPosition` konkret zu implementieren und somit Geschwindigkeiten und Positionen der kollidierenden Körper gemäß den Randbedingungen einer Kollision korrigieren zu können. Um konkrete Kollisionen zwischen Formen zu ermöglichen, genügt es, von `Collision` zu erben und die Kollisionspunkte und Normale konkret zu implementieren. In SiMS sind bereits drei Arten von Kollisionen implementiert:

1. Kollisionen zwischen zwei Kreisen: `CircleCollision`
2. Kollisionen zwischen einem Kreis und einem konvexen Polygon: `PolyCircleCollision`
3. Kollisionen zwischen zwei konvexen Polygonen: `PolyCollision`

Die Kollisionsklassenhierarchie ist in Abbildung 9 dargestellt.

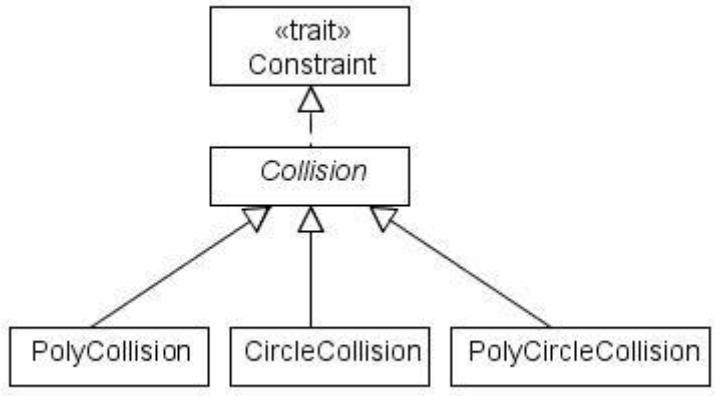


Abbildung 9: Klassenhierarchie von Kollisionen

Soweit wären nun Kollisionen definiert und implementiert. Wie in Kapitel 4.10 erwähnt, werden Kollisionen bei dem Ausführen eines Zeitschrittes in der Simulation berücksichtigt. Die Frage, die nun aber bleibt, ist, wie Kollisionen erkannt werden. Wie schon gesagt, ist Kollisionserkennung ein sehr spezialisierter Teil der Physiksimulation. Daher wird in den folgenden Abschnitten nur eine einzige, nicht sehr effiziente Methode zur Kollisionserkennung beschrieben.

[17] Objekte die Kollisionserkennung und -reaktion ermöglichen, befinden sich in `sims.collision`

5.3. Kollisionen ermitteln

In SiMS erhält die Welt während der Simulation eines Zeitschrittes alle ihre Kollisionen über die Variable

```
val detector: Detector = new GridDetector(this).
```

Die abstrakte Klasse `Detector` enthält eine Referenz zu der Welt und eine Methode `collisions`, die alle Kollisionen angibt. Um es zu ermöglichen, fortgeschrittene Kollisionserkennungsalgorithmen in SiMS zu implementieren, kann anstelle von `GridDetector` eine andere Klasse, die von `Detector` erbt, bei der Initialisierung einer Welt verwendet werden:

```
new World {override val detector = new MyAdvancedDetector(this)}
```

Der Standardalgorithmus in SiMS ist in der Klasse `GridDetector` implementiert. Wenn die Welt `collisions` in einem Zeitschritt aufruft, ermittelt `GridDetector` alle Kollisionen. Dies geschieht in mehreren Phasen. Als erstes werden Formenpaare erstellt, bei denen eine Kollision wahrscheinlich ist. Dies passiert in der sogenannten „Broadphase“. Danach werden die Paare auf genauere Kollisionen geprüft, dies entspricht der sogenannte „Narrowphase“. Dieses Unterteilen in Phasen ermöglicht es, nicht jede Form mit jeder anderen der Welt auf Kollisionen zu überprüfen und somit Kollisionserkennung zu beschleunigen.
Abbildung 10 zeigt den Verlauf einer Kollisionserkennung, die von einem Zeitschritt der Welt aufgerufen wird. In den folgenden Abschnitten wird dieses Verfahren genauer erklärt.

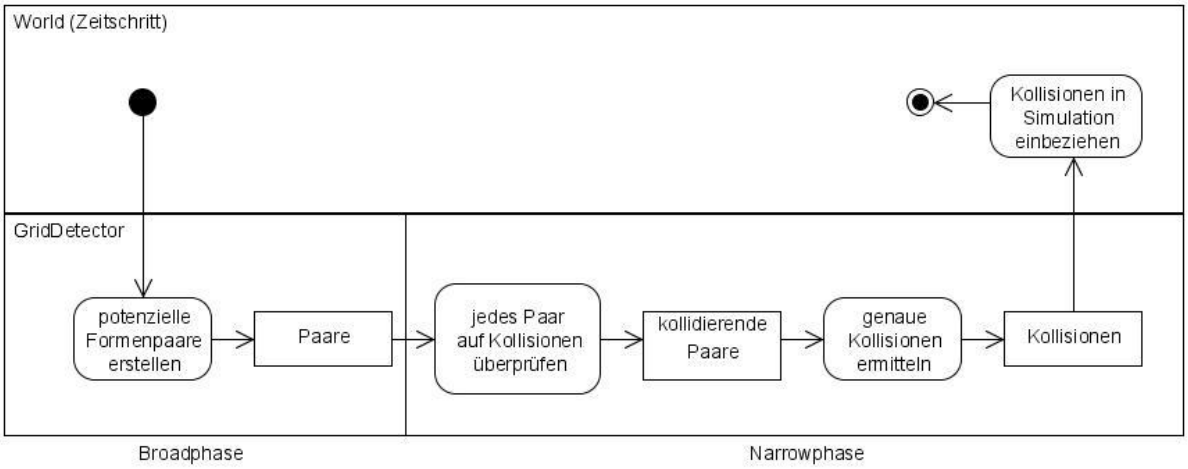


Abbildung 10: Verlauf einer Kollisionserkennung

5.3.1. Broadphase

Alle Kollisionen können nur zwischen Formenpaaren entstehen, die in einem geringen Abstand voneinander stehen. Um diese potenziellen Kollisionen zu ermitteln, teilt `GridDetector` die Welt in ein Gitter mit konstanten Abständen. Danach wird für jede Zelle ermittelt, ob und welche Formen sich in ihr befinden.

Anschließend werden Hüllkörper[18] um die Formen jeder Zelle gebildet und auf Überlappung geprüft. Nur wenn dies der Fall ist, kommt das Formenpaar für eine mögliche Kollision in Frage.

5.3.2. Narrowphase

Nachdem potenzielle Kollisionspaare ermittelt wurden, werden die Formen dieser genauer auf Kollisionen geprüft. Dazu muss zwischen den Arten der Formen unterschieden werden. `GridDetector` enthält Methoden zur Kollisionsprüfung in einer Sammlung von partiellen Funktionen.

```
val detectionMethods = new ArrayBuffer[PartialFunction[(Shape, Shape), Boolean]]
```

Somit können Kollisionserkennungsmethoden zwischen selbst erstellten Formentypen ohne großen Aufwand hinzugefügt werden. Es wird nur eine Funktion benötigt, die zwei Formen als Parameter nimmt und einen Booleschen[19] Wert zurück gibt. Während der Narrowphase, wird für jedes Formenpaar die passende Kollisionsmethode gesucht und angewandt. Gibt sie `true` zurück, so gibt es eine Kollision. In `GridDetector` sind bereits Methoden für Kreise und konvexe Polygone definiert.

- Wenn zwei Kreise kollidieren, ist ihr Abstand kleiner als die Summe ihrer Radien

```
detectionMethods += {
      case (c1: Circle, c2: Circle) => {
            val d = (c1.pos - c2.pos).length
            val rSum = c1.radius + c2.radius
            d - rSum <= 0
      }
}
```

- Um Kollisionen zwischen konvexen Polygonen zu überprüfen, wird das „Separating Axis Theorem“ (SAT)[20] verwendet. SAT ist ein geometrisches Theorem, welches besagt, dass sich zwei konvexe Polygone nicht überlappen, wenn und nur wenn man eine Gerade zwischen ihnen erstellen kann, die sie nicht schneidet. Diese Gerade ist parallel zu einer der Seiten der in Frage kommenden Polygone oder anders gesagt orthogonal zu eine der Seitennormalen.

```
detectionMethods += {
      case (p1: ConvexPolygon, p2: ConvexPolygon) => {
            val sides = p1.sides ++ p2.sides
            val axes = sides map (_.n0)
            axes.forall((a: Vector2D) =>
                  p1.project(a) overlaps p2.project(a))
      }
}
```

  Die Methode projiziert beide Polygone auf jede der Seitennormalen und überprüft, ob es bei einer Projektion Überlappungen gibt. Dies entspricht dem Versuch, eine Gerade zwischen beiden Polygonen zu ziehen.

[18] Hüllkörper sind in SiMS Rechtecke, die eine Form umhüllen und nach den X- und Y-Achsen orientiert sind. Sie ermöglichen eine schnelle Überprüfung auf Überlappung.
[19] logischer Wert (wahr/falsch)
[20] (Burns und Sheppard kein Datum)

- Um Kollisionen zwischen einem Polygon und einem Kreis zu ermitteln, wird überprüft, ob eine der Seiten oder der Ecken des Polygons sich in dem Kreis befindet oder ob dieser sich in dem Polygon befindet.

```
detectionMethods += {
      case (p: ConvexPolygon, c: Circle) => {
            val distances = for (s <- p.sides) yield
                  (s distance c.pos)
            distances.exists(_ - c.radius <= 0) || (p contains c.pos)
      }
}
```

Nachdem alle kollidierenden Paare ermittelt wurden, kommt der letzte Teil der Kollisionserkennung. Alle kollidierenden Paare werden analysiert und je eine konkrete Kollision, wie in 5.2 beschrieben, zurückgegeben. Dies geschieht ähnlich wie bei der Kollisionsüberprüfung mit einer Sammlung von partiellen Funktionen, die es ebenfalls ermöglichen, Kollisionen zwischen neuen Formen zu erstellen. Das Erstellen eines Kollisionsobjekts beinhaltet das Ermitteln der Kontaktpunkte und der Kontaktnormalen und ist relativ kompliziert. Um die Physiksimulation in dieser Arbeit im Vordergrund zu lassen, werde ich an dieser Stelle nicht weiterhin über das Erstellen konkreter Kollisionen schreiben. Dies kann im Quellcode genauer nachgelesen werden.

5.4. Schluss

In diesem Kapitel wurde erklärt, wie Kollisionen in einem Physiksimulator berücksichtigt werden können. Die in SiMS implementierte, relativ einfache Methode braucht bei der Simulation eines Zeitschrittes verhältnismäßig am meisten Rechenzeit und ist nicht 100% realitätsgetreu. Dank dem erweiterbaren Kollisionserkennungssystem von SiMS, können jedoch schnellere Systeme erstellt und ohne großen Aufwand implementiert werden.

6. Schluss

Diese Dokumentation gibt einen Überblick über die gesamte Funktionsweise der Physiksimulationsbibliothek „Simple Mechanics Simulator“. Sie zeigt wie die Grundbausteine des Simulators - die Körper - modelliert werden und wie sie in einem System simuliert werden. Dazu werden noch detailliertere Informationen über Körper mit eingeschränkten Bewegungen und über Kollisionserkennung gegeben.

SiMS veranschaulicht physikalische Interaktionen zwischen Festkörpern. Das besondere ist die einfache Struktur der Bibliothek. Sie ermöglicht zwar nicht die beste Optimierung und Rechengeschwindigkeit, dennoch macht sie den Simulator überschaubar und leicht verständlich. Hinzu kommt, dass SiMS dank ihr sehr ausbaufähig ist. Ein Entwickler kann dem Simulator völlig neue Objekte hinzufügen, existierende Funktionen erweitern oder sie mit anderen, effizienteren ersetzen; all dies mit minimalem Aufwand. Das Ziel dieser Maturaarbeit ist somit erfüllt.

Durch diese Arbeit habe ich viel Neues gelernt. Zum einen konnte ich meine Kenntnisse in der Dynamik und Kinematik vertiefen. Zum anderen habe ich noch viel Neues in der Mathematik gelernt, wie z. B. Matrizenrechnung. Das wirklich Interessante dieser Arbeit war aber, physikalische Gesetze und sonst so abstrakt klingende mathematische Konzepte zusammen in einem angewandten Beispiel zu vereinen. Das Implementieren dieser stellt nochmals eine ganz andere Schwierigkeit dar, es braucht viel Planung und vor allem Geduld, denn nicht immer funktioniert alles wie gewollt auf den ersten Versuch! Ganz zu schweigen von dem Lernen der eigentlichen Programmiersprache hat mir die Arbeit auch geholfen, meine Fähigkeiten in Softwarearchitektur zu verbessern. Zusätzlich hat mir so ein großes Projekt der Maturaarbeit ganz neue Erfahrungen gebracht. Das Arbeiten während eines ganzen Jahres an einem selbstgewähltem Projekt kann sehr spannend aber auch sehr frustrierend sein. Es braucht viel Recherchen und Zeitinvestition, dennoch bietet es am Ende einen echten Gewinn.

Die Maturaarbeit ist nun fertig, doch der Simulator ist noch lange nicht perfekt. Man könnte seine Kollisionserkennung verbessern oder seine ziemlich limitierten Arten von Verbindungen erweitern. In der Zukunft könnte SiMS vielleicht sogar auf drei Dimensionen erweitert werden. Eines sollte bei allen Erweiterungen aber immer im Vordergrund stehen: das Hauptziel von SiMS ist es, so leicht verständlich und erweiterbar wie möglich zu sein.

Quellen

(Burns und Sheppard kein Datum) Burns, Raigan, und Mare Sheppard. *N Tutorials - Collision Detection and Response.* Metanet Software Inc. http://www.harveycartel.org/metanet/tutorials/tutorialA.html (Zugriff am 22. 7 2008).

(Catto, Contact Manifolds 2007) Catto, Erin. „Contact Manifolds." *Game Developers Conference.* San Francisco, 2007.

(Catto, Modelling and Solving Constraints 2008) —. „Modelling and Solving Constraints." *Game Developers Conference.* 2008.

(Collision Detection kein Datum) „Collision Detection." *Essential Math for Game Programmers.* http://www.essentialmath.com/CollisionDetection.pps (Zugriff am 11. 10 2009).

(Halliday, Resnick und Walker 2004) Halliday, David, Robert Resnick, und Jearl Walker. *PHYSIQUE - 1. Mécaniqe.* Dunod, 2004.

(Hecker 1996) Hecker, Chris. „Physics, The Next Frontier." *Game Developer*, Oktober/November 1996: 18-20.

(Jacobi-Matrix 2009) *Jacobi-Matrix.* Wikipedia. 3. 7 2009. http://de.wikipedia.org/wiki/Jacobi-Matrix (Zugriff am 8. 9 2009).

(jBox2D Demos kein Datum) *jBox2D Demos.* http://www.jbox2d.org/ (Zugriff am 20. 9 2009).

(Odersky, Spoon und Venners 2008) Odersky, Martin, Lex Spoon, und Bill Venners. *Programming in Scala.* artima, 2008.

(Witkin 1997) Witkin, Andrew. *Physically Based Modelling: Principles and Practice - Constrained Dynamics.* Robotics Institute Carnegie Mellon University, 1997.

Abbildungen

Tabellen

Glossar

Constraint	Eine Randbedingung, die die Bewegung eines oder mehrerer Körper einschränkt.
Constraint-Impuls	Ein Impuls, der auf einen oder mehrere Körper so einwirkt, dass danach deren Geschwindigkeiten bzw. Positionen den Randbedingungen der Körper wieder entsprechen.
Constraint-Kraft	Eine Kraft, die auf einen oder mehrere Körper so einwirkt, dass danach deren Geschwindigkeiten bzw. Positionen den Randbedingungen der Körper wieder entsprechen.
Position-Constraint	Eine Randbedingung, die die Position eines oder mehrerer Körper einschränkt.
Position-Constraint-Funktion	Eine zu einer Randbedingung zugeordnete Funktion. Sie nimmt als Parameter einen Punkt und gibt null zurück, wenn dieser Punkt sich auf einer der Randbedingung erlaubten Position befindet.
Velocity-Constraint	Eine Randbedingung, die die Geschwindigkeit eines oder mehrerer Körper einschränkt.
Velocity-Constraint-Funktion	Eine zu einer Randbedingung zugeordnete Funktion. Sie nimmt als Parameter einen Punkt und gibt null zurück, wenn dieser Punkt eine der Randbedingung erlaubten Geschwindigkeiten hat.

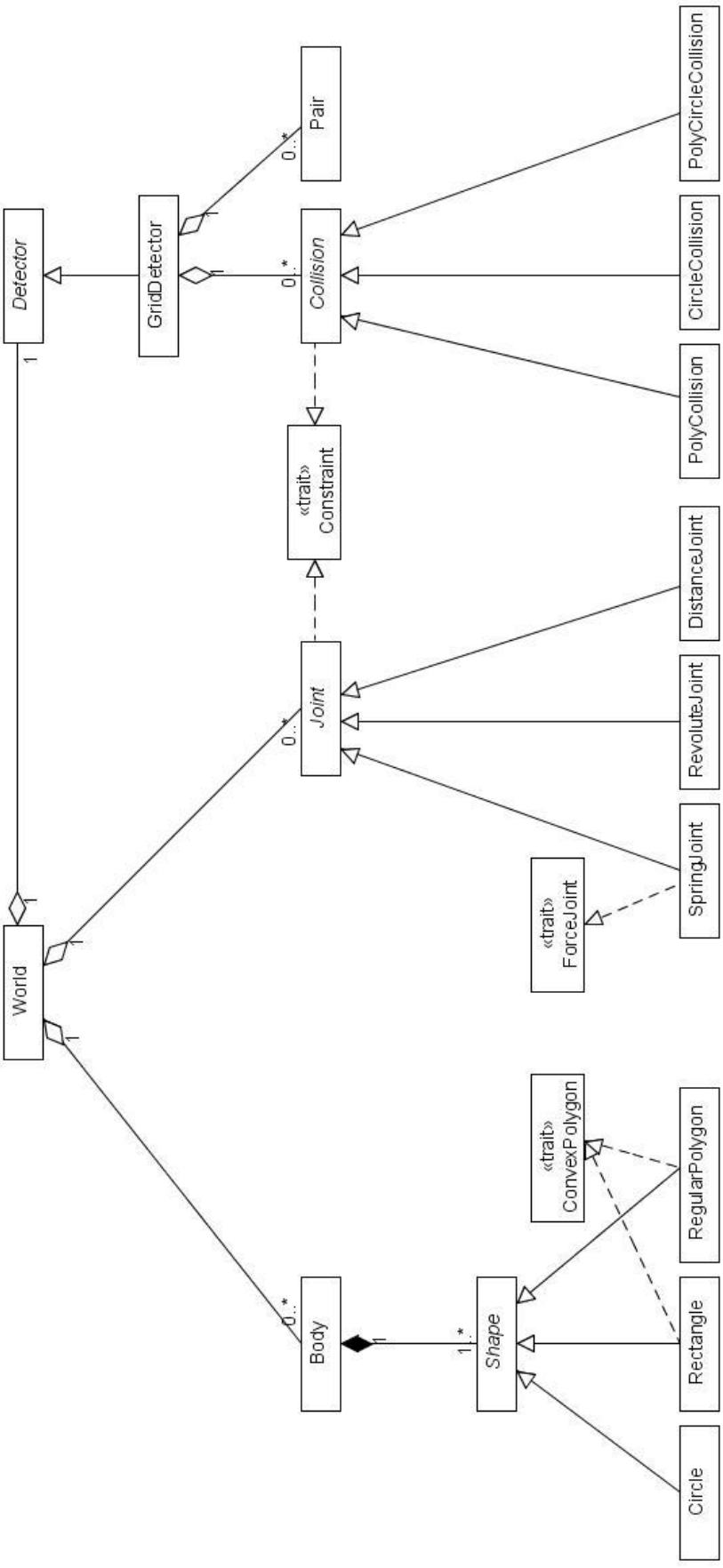
Verwendete Symbole

Abkürzung	*Bedeutung*
$\boldsymbol{a}$	Beschleunigung.
$C(\boldsymbol{x})$	Position-Constraint-Funktion, ergibt 0, wenn $\boldsymbol{x}$ eine legale Position ist.
$\dot{C}(\boldsymbol{x})$	Velocity-Constraint-Funktion, ergibt 0, wenn der Punkt $\boldsymbol{x}$ eine legale Geschwindigkeit hat.
$\boldsymbol{F}$	Kraft.
$\boldsymbol{F_c}$	Constraint-Kraft.
h	Dauer (meistens Dauer eines Zeitschrittes).
I	Trägheitsmoment.
$\boldsymbol{J}$	Jacobi-Matrix.
$\boldsymbol{L}$	Drehimpuls.
m	Masse.
$\boldsymbol{M}$	Massenmatrix.
$\boldsymbol{n}$	Normalenvektor.
$\boldsymbol{p}$	Impuls.
$\boldsymbol{p_c}$	Constraint-Impuls.
$\boldsymbol{P}$	Impulsmatrix.
$\boldsymbol{P_c}$	Constraint-Impulsmatrix.
$\boldsymbol{r}$	Relative Distanz zwischen Schwerpunkt und einem Punkt.
$\boldsymbol{S}$	Schwerpunkt.
t	Zeit.
$\boldsymbol{u}$	Einheitsvektor der Position.
$\boldsymbol{v}$	Geschwindigkeit. Meistens bezogen auf einen Punkt.
$\boldsymbol{v_s}$	Lineargeschwindigkeit des Schwerpunktes eines Körpers.
$\boldsymbol{V}$	Geschwindigkeitsmatrix.
$\boldsymbol{x}$	Position.
Δx	Unterschied von x.
$\boldsymbol{\theta}$	Rotation.
λ	Constraint-Impulsstärke.
$\boldsymbol{\tau}$	Drehmoment.
$\boldsymbol{\omega}$	Winkelgeschwindigkeit des Schwerpunktes eines Körpers.
$\boldsymbol{x} \perp \boldsymbol{y}$	$\boldsymbol{x}$ ist orthogonal zu $\boldsymbol{y}$.

Indizes können verschiedene Bedeutungen haben. Diese ergeben sich aus dem Kontext bei Verwendung der Abkürzung.

Anhang I

Klassendiagramm von SiMS



Anhang II

Zusammensetzung der Maturaarbeit

1. Dokumentation

Erläuterung der verwendeten Konzepte in SiMS.
Pfad auf CD : /SiMS.pdf

2. Bibliothek und Quellcode

Der Quellcode von SiMS.
Es besteht die Möglichkeit den Quellcode in Eclipse als Projekt zu importieren.
Pfad auf CD : /SiMS/src/

3. API Dokumentation

Dokumentation, aus den Kommentaren des Quellcodes generiert.
Die Bibliothek befindet sich in dem Package `sims`.
Pfad auf CD : /SiMS/doc/index.html

Die Dokumentation, der Quellcode und die API Dokumentation sind ebenfalls erhältlich im Internet unter:

http://sourceforge.net/projects/simplemechanics/