AS3-Tutorium:Physics:Vertiefung

Wechseln zu:Navigation, Suche

Inhaltsverzeichnis

1 Überblick

Dieser Artikel erklärt den Aufbau und die Funktionsweise einer vollwertigen Physics-Engine und stellt außerdem die wichtigsten Physics-Engines vor. Im Detail werden die Vorgehensweisen und Algorithmen erklärt, welche in der lyneth Physics API verwendet werden. Diese Sammlung stellt nur eine begrenzte Zahl der Möglichkeiten dar, welche im Bereich der Physics zum Einsatz kommen. Für den 3D-Raum werden einige Sachverhalte komplizierter, deshalb beschränkt sich die Beschreibung

auf den 2D-Raum. Informationen zur Verwendung von vorhanden 3D-Physics-Engines in Actionscript3 finden sie im Artikel AS3-Tutorium:Physics:3D-Physics-Engines.

2 Komponenten einer Physics-Engine

2.1 Mathematische Objekte

Die wichtigste Grundlage für alle physikalischen Simulationen sind mathematische Hilfsklassen. Diese werden nicht zwingend für die Realisierung eine Physics-Engine benötigt, sind aber aufgrund der Vereinfachung der Sachverhalte unverzichtbar.

2.1.1 Vektor

Ein zweidimensionaler Vektor besteht aus einer x- und einer y-Komponente. Dieser hat immer eine Länge und eine Richtung. Benutzt man einen Vektor für eine Position, so versteht man den Vektor als Abstand und Richtung der Position vom Ursprung aus. Vektoren werden für folgende Sachverhalte verwendet:

Position
Geschwindigkeit
Beschleunigung
Kollisionsnormale
Kollisionstangente
Koordinatensysteme (ein Paar von Vektoren)

Wichtig ist also alle relevanten mathematischen Funktionalitäten für Vektoren zu implementieren. Dazu gehören:

Addition, Subtraktion, Multiplikation, Division
Längenbestimmung
Normierung (den Vektor auf Länge 1 skalierung)
Normalenberechnung (einen Vektor berechnen, welcher senkrecht auf dem Ursprungsvektor steht)
Skalarprodukt von zwei Vektoren
Kreuzprodukt von zwei Vektoren (in 2D-Raum ein Skalar)

2.1.2 Projektion

Für 2D-Physics-Engines sind lediglich 1D-Projektionen relevant. Hierzu wird ein Projektionsvektor (Vektor mit der Länge 1) und mindestens ein zu projizierender Vektor benötigt. Die Projektion wird berechnet, indem man das Skalarprodukt für alle gewünschten Vektoren mit dem Projektionsvektor berechnet. Es entsteht ein Intervall, welches in Kombination mit dem Projektsvektor die Projektion innerhalb des ursprünglichen Koordinatensystems beschreibt. Projektionen werden benötigt für:

Berechnung der Tiefe einer Kollision (Minimum Translation Distance)
Berechnung der Normalen- und Tangenanteile von Geschwindigkeiten
Bestimmung von Überschneidung von polygonalen Formen

2.2 Geometrische Objekte

Geometrische Objekte sind relevant für die Beschreibung der Formen von physikalischen Objekten. Die Geometrie wird strikt von der Dynamik getrennt, um beide Bereiche unabhängig voneinander zu halten.

2.2.1 Abstrakte Form

Eine Form besitzt immer eine globale Position und Rotation (unter Umständen auch eine lokale Position und Rotation).

2.2.2 Kreis

Der Kreis ist die einfachste geometrische Form, er wird durch die Attribute einer Form und einen Radius beschrieben.

2.2.3 (Konvexes) Polygon

Ein Polygon erweitert eine abstrakte Form um eine feste Anzahl von Eckpunkten. Für die meisten Algorithmen der Kollisionserkennung werden konvexe Polygone benötigt. Konkave Polygone setzen sich aus der Kombination von mehreren konvexen Polygonen zusammen. Um Berechnungen sparen zu können werden die Normalen der Seiten eines Polygons und die Projektionen entlang der eigenen Normalen zwischengespeichert.

2.3 Dynamik

> Die Dynamik (gr. dynamis Kraft) ist das Teilgebiet der Mechanik, das sich mit der Wirkung von Kräften befasst. [..] Hier wird unter Dynamik die Beschreibung der Bewegung von Körpern unter Einfluss von Kräften verstanden, im allgemeineren Sinn auch das Zeitverhalten eines Systems und die zu seiner Beschreibung verwendeten Bewegungsgleichungen.*(Quelle: Wikipedia)*

2.3.1 Zustand

Ein Zustand zu einem bestimmten Zeitpunkt t setzt sich zusammen aus Position, Geschwindigkeit und Beschleunigung. Jedes dieser Attribute wird durch einen 2D-Vektor kombiniert mit einem Wert für die Rotation realisiert (für Rotation, Winkelgeschwindigkeit und Winkelbeschleunigung).

2.3.2 Körper

Jeder Sammlung von geometrischenen Form wird ein Zustand zugeordnet. Die Kombination von geometrischen Formen und Zuständen ergibt einen Körper.

2.3.3 Kräfte

Kräfte sind im Normalfall Dynamik-Objekte ohne geometrische Form. Für den Zustand ist in den meisten Fällen allein eine Position ausreichend. Zusätzlich besitzt eine Kraft einen Kraftvektor und

einen Wirkungsbereich (oft durch einen Radius realisiert). Wie die Kraft letzendlich auf Körper angewendet wird, hängt von der Art der Kraft ab. Die Hook´sche Feder (siehe letzter Vortrag) kann ebenfalls als Kraft realisiert werden, welche allerdings nur auf zwei Körper wirkt und allein diese beeinflusst.

Beispiele:

http://glossar.hs-augsburg.de/beispiel/tutorium/physics/Lawrence_Physics/swf/TutoriumPhysics-2-01-Formen.swf
http://glossar.hs-augsburg.de/beispiel/tutorium/physics/Lawrence_Physics/swf/TutoriumPhysics-2-02-Kraefte.swf
http://glossar.hs-augsburg.de/beispiel/tutorium/physics/Lawrence_Physics/swf/TutoriumPhysics-2-03-Kraefte.swf
http://glossar.hs-augsburg.de/beispiel/tutorium/physics/Lawrence_Physics/swf/TutoriumPhysics-2-04-Kraefte.swf

2.4 Integration

Die Integration beschreibt numerische Lösungsverfahren für Differentialgleichungen. Die Zustandsänderungen der Dynamik eines Körpers werden mithilfe dieser Algorithmen berechnet (Kräfte, Beschleunigungen, Geschwindigkeiten, Orte). Es gibt viele verschiedene Algorithmen, welche sich in Effizienz und Exaktheit unterscheiden. Wichtige Links zu diesem Thema sind:

http://de.wikipedia.org/w/index.php?title=Datei:RK_Verfahren.png&filetimestamp=20090121000101
http://gafferongames.com/game-physics/integration-basics/

2.4.1 Euler

Das Verfahren nach Euler ist die einfachste Form der Integration. Hierbei wird abhängig von der vergangenen Zeit (im Normalfall ein festes Zeitintervall) die Beschleunigung auf die Geschwindigkeit und die Geschwindigkeit auf die Position addiert. Die Beschleunigung setzt sich aus der Summe der wirkenden Kräfte zusammen. In einfachen Fällen wird die Beschleunigung einfach manuell gesetzt.

```
a1 = f / m
v1 = v0 + a1 * t
x1 = x0 + v1 * t
```

Liegt etwas anderes als eine konstante Geschwindigkeit und keine Beschleunigung vor, so erzeugt dieses Verfahren sehr große Berechnungsfehler. Das Euler-Verfahren kommt jedoch sehr häufig zum Einsatz, vor allem auch in AS3-Physics-Engines, da es einfachste und performanteste Verfahren darstellt. Die Berechnungsfehler sind zu vernachlässigen, da die Ergebnisse meistens trotzdem realistisch wirken.

2.4.2 Runge-Kutta

Das Runge-Kutta-Verfahren ist dem Taylor-Polynom sehr ähnlich. In Physics wird meistens das Verfahren in der vierten Ordnung verwendet (man spricht daher von RK4). Dieses Verfahren zur Integration ist das mit dem geringsten Fehler aber auch dem höchsten Rechenaufwand. Hierbei wird pro Zeitintervall nicht nur der neue Wert berechnet, sondern auch zwei weitere Zwischenwerte, außerdem wird der alte Wert mit in Betracht gezogen. Diese werden dann gewichtet miteinander verrechnet.

> Die dabei bei nichtlinearen Funktionen notwendigerweise auftretenden Fehler (es werden sämtliche höheren Glieder der Taylor-Entwicklung vernachlässigt) können durch geeignete Kombinationen verschiedener Differenzquotienten teilweise kompensiert werden. Das Runge-Kutta-Verfahren ist nun eine solche Kombination, die Diskretisierungsfehler bis zur dritten Ableitung kompensiert.*(Quelle:Wikipedia)*

Formel bei Wikipedia

2.4.3 Midpoint

Das sogenannte Midpoint-Verfahren wird in den meisten als RK2, also als Runge-Kutta-Verfahren der 2. Ordnung implementiert. Dementsprechend ist der Rechenaufwand etwas höher als bei Euler, das Ergebnis allerdings nicht so genau wie RK4. Der Algorithmus wird nicht häufig in Physics-Engines eingesetzt.

2.4.4 Verlet

Der Verlet-Algorithmus verspricht eine größere Stabilität und einen geringeren Fehler als bei Euler. Dazu wird nicht mit Variablen für Geschwindigkeiten gerechnet, sondern werden diese Werte aus der aktuellen Position und der vorgehenden berechnet. Das Verfahren ist allerdings nicht sehr weit verbreitet, da die Implementierung oftmals Änderungen an der Architektur von Engines voraussetzt.

2.5 Kollisionen

Im Folgenden werden die beiden gängigsten Verfahren für die Kollisionserkennung und -auflösung erklärt.

2.5.1 Kollisionserkennung

Wie bereits in der Einführung vorgestellt reicht der Satz von Pythagoras und ein simpler Wertevergleich aus um die Kollision zwischen zwei Kreisen zu bestimmen. Für Rechtecke mit Seiten parallel zu den Achsen des Koordinationssystems auf Kollision zu testen reicht ebenfalls ein Wertevergleich aus. Diese Verfahren funktionieren nicht bei Polygonen, jedoch benutzt man sie gerne um erstmalig festzustellen ob Polygone potentiell überhaupt kollidieren können. Dazu wird jedem Polygon ein Bounding Circle (Umkreis) oder eine Bounding Box (umschließendes Rechteck) zugewiesen. Somit muß nicht für jedes Polygonenpaar eine aufwändige Kollisionerkennung durchgeführt werden.

Das einfachste Verfahren ist die Anwendung des Trennungssatzes (englisch: Seperating Axis Theorem oder SAT). Dabei wird jede Normale (Vektor senkrecht zur Seite eines Polygons) als Projektionsachse für beide Polygone benutzt. Sobald eine dieser Projektionen zu keiner Überlappung der beiden Intervalle führt, liegt keine Kollision vor. Gibt es allerdings auf jeder Achse eine Überlappung der Intervalle, so liegt eine Kollision vor. Die Kollisionnormale ist der Vektor bei welchen die Überlappung der Projektionsintervalle minimal ist. Der Betrag dieser Überlappung beschreibt die Tiefe der Kollision. Der Teil mit dem größten Aufwand ist die Berechung der Kollisionspunkte. Hierbei muß nach den Fällen Punkt-Flächen- und Flächen-Flächen-Kollisionen unterschieden werden (Punkt-Punkt-Kollision wird ignoriert).

Das SAT-Verfahren arbeitet nach dem Ausschlussprinzip, d.h. es arbeitet sehr schnell wenn wenig Kollisionen vorliegen. Bei vielen Kollisionen arbeitet das Verfahren aber umso langsamer, da es immer alle Normalen aller Polygone testen muß. Abgesehen von der Performance, erzeugt das Verfahren gerine Fehler und funktioniert stabil.

2.5.2 Kollisionsauflösung

Ebenso wie bei der Kollisionserkennung gibt es verschiedene Algorithmen um die Kollisionen zu lösen. Bei den meisten Systemen und den angeführten Verfahren werden Kollisionen erst erkannt und aufgelöst wenn sie bereits geschehen sind. Deshalb ist es wichtig die miteinander kollidierenden Objekte erst wieder so weit voneinander zu entfernen, bis sie sich nicht mehr überschneiden. Kollisionsauflösungen dieser Art werden als "penalty based" bezeichnet. Wie bereits im vorherigen Vortrag erwähnt, wird dazu die Minimum Translation Distance (MTD) kombiniert mit der Kollisionsnormale benötigt. Die Objekte werden abhängig von ihrer Masse voneinander entfernt.

Nun kommt es zu eigentlichen Reaktion auf die Kollision. Wie bei einer Kollision von Kreisen kann man das Prinzip des zentralen, unelastischen Stoßes implementieren, allerdings wird man bei diesem Verfahren keine Winkelgeschwindigkeiten und entsprechende Rotationsänderung berechnen können. Um dies möglichst einfach berechnen zu können verwendet man den Gesamtimpuls der Kollision am Kollisionspunkt. Dazu werden Linear- und Winkelgeschwindigkeiten an diesem Punkt, die Trägheiten der Körper (massenabhängig), die Reibung und die Elastizität (nicht sichtbar) der Körper berechnet. Die Geschwindigkeiten werden in Normalen- und Tangenanteil aufgeteilt und zusammen mit den neuen Winkelgeschwindigkeiten den Körpern zugewiesen.

Für exakte Beschreibungen gibt es hier eine Sammlung Dokumenten. Diese wurde auch als Basis für die Implementierung des lyneth Physics API benutzt.

3 Verfügbare Physics-Engines

3.1 Box2D

Die Box2D-Engine basiert auf der gleichnamigen C++-Engine von Erin Catto. Die Engine ist plattformunabhängig, wurde bereits für den Nintendo DS und das iPhone eingesetzt und wird in vielen aktuellen PC-Spielen verwendet. Box2D ist mit Abstand die performanteste und umfangreichste Physics-Engine für AS3. Die einzige Kritik gilt dem Aufbau der API, diese wird manchmal als unhandlich und/oder unverständlich bezeichnet, hauptsächlich weil die AS3-Version ein direkter Port der C++-Version ist und somit für viele Actionscript-Entwickler nicht intuitiv genug ist.

3.2 Motor Physics

Motor Physics ist im Inneren Box2D sehr ähnlich, da der Entwickler sich stark an den Algorithmen und Vorgehensweise von Erin Catto orientiert hat. Die Engine soll in gewissen Bereichen eine bessere Performance bieten als Box2D, jedoch gibt es bisher keine direkten Vergleiche oder Benchmarks. Außerdem bietet Motor Physics nicht den vollen Funktionsumfang von Box2D. Der Aufbau der API unterscheidet sich jedoch etwas, da die Engine von Anfang auf AS3 ausgelegt war.

3.3 Fisix Engine

Die Fisix Engine ist eine der älteren Physics-Engines. Anfangs wirkte sie sehr vielsprechend, liegt jedoch inzwischen in Sachen Performance und Möglichkeiten hinter Box2D und Motor. Ein nennenswertes Feature ist Gravitation von Körpern. Das größte Problem ist allerdings, dass die Engine nur frei ist für nichtkommerzielle Zwecke. Alle anderen Engines sind komplett Open-Source.

3.4 APE - Another Physics Engine

Die APE ist eine Weiterentwicklung der AS2-Physics-Engine Flade. Ein zentrales Argument für diese Engine, ist die oft genannte einfache und intuitive Struktur der API, was dadurch begründet sein kann, dass die Engine bereits seit 2005 entwickelt wurde und von Beginn an für Actionscript ausgelegt war. Im Gegensatz zu den oben genannten Engines besitzt sie jedoch einen sehr eingeschränkten Funktionsumfang.

3.5 aM Physics Engine

André Michelle, bekannter Actionscript-Entwickler und Sprecher auf vielen Flash-Konferenzen, stellt eine einfache Physics-Engine zur Verfügung. Diese ist jedoch nicht für die Produktion von Applikationen sondern für das Ausprobieren von Physics geeignet.

3.6 FOAM

> It is meant as an architectural and mathematical reference for developers interested in physics simulation in the area of game development or otherwise. It trades efficiency for modularity and extensibility.*(Quelle:[1])*

Die FOAM-Engine ist ebenfalls nicht für den Produktionsbetrieb geeignet, zeigt jedoch eindrucksvoll und verständlich wie Algorithmen für eine Physics-Engine implementiert werden müssen.

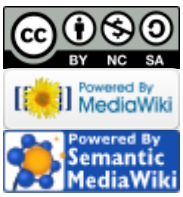
3.7 lyneth

lyneth ist eine Physics-Engine von Alexander Lawrence. Der Fokus dieser Engine liegt auf der Verständlichkeit der API und der Anbindung an Flash-Applikationen. Die Performance ist jedoch in keiner Weise vergleichbar mit der von Box2D oder Motor Physics. Die lyneth-Engine ist nicht für den Produktionsbetrieb geeignet und momentan nicht frei verfügbar. Der Kern der Engine wird unter einer Open-Source Lizenz veröffentlicht.

Kategorien:
Spielephysik
AS3-Tutorium: Physics

Diese Seite wurde zuletzt am 6. November 2016 um 11:56 Uhr bearbeitet.

DE 3.0.


BY NC SA
Powered By MediaWiki
Powered By Semantic MediaWiki