ODE-Tutorial

Marcel Kretschmann, 2005
E-Mail: Marcel_Kretschmann@web.de

Anmerkungen

Dieses Tutorial basiert auf einem speziellen Beispielquelltext. Dieser demonstriert den grundlegenden Aufbau eines ODE-basierenden Simulationsprogrammes. Der betreffende Quelltext ist in der Datei `ode_template.cpp` zu finden.

Überblick

Grundsätzlich kann man ODE-Simulationsprogramm sowohl in C als auch in C++ schreiben, da die Funktionalität ausschließlich auf C-Funktionen basiert. Hierzu werden einige Standartfunktionen benötigt, welche verwendet werden müssen. Diese sind im Template die Funktionen:

```
nearCallback (void *data, dGeomID o1, dGeomID o2)
start ()
simLoop ( int pause )
command ( int cmd )
```
und natürlich `main (int argc, char **argv)`

In `main` werden alle Körper und Oberflächen angelegt, possitioniert und passend verbunden. Dann erfolgt erst die eigentliche Simulation. Es wird die Funktion `start` aufgerufen, wo man initialisierende Funktionsaufrufe, und Variablenbelegungen durchführen kann. Die Funktion `start` wird nur einmal aufgerufen. Nun wird die Funktion `simLoop` ausgeführt. Dies erfolgt bis zum Beenden des Programmes. Innerhalb der Funktion `simLoop` erfolgt die Abfrage ob es in der Simulation zu Kollisionen einzelner Körper kam. Im Fall einer Kollision wir die Funktion `nearCallback` aufgerufen, wo die eigentliche Kollisionsbehandlung erfolgt. Falls während der Simulation eine Tasteneingabe durch den Benutzer erfolgt ist wird die Funktion `command` aufgerufen, in welcher eine genauere Auswertung des eingegebenen Kommandos erfolgt.

Datentypen

Die drei wichtigsten Datentypen, welche man für die Benutzung der ODE-Bibliothek kennen muss, sind:

`dBodyID :`
Variablen diesen Typs dienen als Referenzen auf die einzelnen Körper, welche in einer physischen Simulation beteiligt sind.

`dGeomID :`
Variablen dieses Typs dienen als Referenz auf geometrische Hüllen, oder Geometrieobjekte. Diese werden für Körper definiert, da Körper an sich in ODE keine Oberfläche haben. Geometrieobjekte dienen der Visualisierung und vor allem der Kollisionserkennung.

`dJointID :`
Variablen dieses Typs sind Referenzen auf so genannte Joints, also auf Verbindungen die zwischen einzelnen Objekten angelegt werden. Dies sind meist verschiedene Arten von Gelenken.

Desweiteren gibt es noch eine Reihe zusätzlicher Datentypen, wie zum Beispiel `dReal`, welcher als Ersatz für den Datentyp `float` fungiert, oder der dreidimensionale Vektor `dVector3.`

main

Zunächst müssen die einzelnen Funktionsbezeichner festgelegt werden, welche für die oben genannten Funktionen wie simLoop verwendet werden sollen. Es ist also durchaus möglich die Funktionen `start`, `simLoop` und `command` anders zu benennen, darauf wurde aber im vorliegenden Template verzichtet. Die Zuweisung der einzelnen Funktionen werden in einer Variablen des Typs `dsFunctions` gespeichert. Am Namen des Typs ist zu erkennen, dass es sich hierbei nicht direkt um einen Typ der ODE-Bibliothek selbst, sondern der Bibliothek drawstuff handelt (ds = drawstuff).
`path_to_texture = 0` bedeutet, dass die zu verwendeten Texturen im Standartpfad zu suchen sind: /textures/

Als ersten Schritt gilt es die Welt, in der die Simulation laufen soll genauer zu spezifizieren.

- `welt = dWorldCreate ();`
- `raum = dHashSpaceCreate (0);`
- `contactgroup = dJointGroupCreate ( 1000000 );`
  Dies ist eine Sammlung von Objekten, welche für die Kollisionserkennung benötigt wird.
- `dWorldSetGravity ( welt , 0 , 0 , -9.81 );`
  Dies definiert einen globalen, dauerhaft wirkenden Kraftvektor. Im Template ist die Standarterdanziehungskraft defniert.
- `dWorldSetERP ( welt , 1 );`
  Dies gibt den globalen Fehlerkorrekturwert an, anhand welchem Berechnungsfehler an Gelenken korrigiert werden.
- `untergrund = dCreatePlane ( raum , 0 , 0 , 1 , 0 );`
  Hier wird in der Variable Untergrund, vom Typ `dGeomID`, eine Referenz auf die Oberfläche des Bodens gesetzt, was für Kollisionserkennung mit diesem erforderlich ist.

Es folgt nun die Definition der einzelnen Körper, welche an der Simulation beteiligt sind.

Anmerkung:

Im Template wurden die beiden wichtigsten Referenzvariablen für eine Körperdefinition, Variablen von den Typen `dBodyID` und `dGeomID` in einer Struktur zusammengefasst, und bilden den neuen Typ `objekt`. Dies ändert jedoch nichts an der Benutzung, sondern dient nur der Übersichtlichkeit.

- Anlegen eines Körpers:
  1. `km[0].body = dBodyCreate ( welt );` , wobei body1 eine Variable vom Typ dBodyID ist
     Hier wird eine Referenz auf einen Körper, der mit diesem Befehl angelegt wird, in der Variable body1 gespeichert.
  2. `dBodySetPosition ( km[0].body , 0.0 , 0.0 , 1.0 );`
     Hier wird die Position des neu angelegten Körpers km[0].body festgelegt. Die Positionsangabe erfolgt in absoluten Weltkoordinaten.
- Festlegung der Masse der angelegten Körper:
  Für eine korrekte Simulation muss nun jedem Körper eine entsprechende Masseverteilung und eine Gesamtmasse über dieser Verteilung zugewießen werden:

1. `dMassSetBox ( &masse , 1 , 1 , 0.5 , 0.5 );`
   Mit dieser Anweisung wird eine neue Massenverteilung festgelegt, welche in die Variable `masse` gespeichert wird. Wichtig ist hier, dass es weitere Massenverteilungsfunktionen gibt. Diese hängen davon ab, welche geometrische Grundform der Körper haben soll. In dem Beispiel wurde die Funktion für einen Quader gewählt. Eine weitere Funktion wäre zum Beispiel `dMassSetSphere` für Kugeln. Die Variable muss hierbei vom Typ `dMass` sein. Der erste Parameter gibt die Gesamtmasse der Masseverteilung an (also hier zunächst 1). Die folgenden Parameter geben die räumliche Verteilung der Masse an. Sie sind allerdings abhängig von der gewählten Funktion, also von der geometrischen Grundform des Körpers (in diesem Fall x-, y- und z-Ausdehnung). Wenn diese die gleichen Zahlenwerte wie die Größe der Hülle des entsprechenden Körpers (siehe weiter unten im Tutorial) besitzen, so hat jeder Punkt des Körpers die gleiche Dichte, und die Masse ist gleichmäßig verteilt.
2. `dMassAdjust ( &masse , 2 );`
   Hiermit kann nachträglich die Gesamtmasse angepasst werden. In dem Fall wird sie auf 2 gesetzt. Mann muss somit nicht ständig die bereits bestehende Masseverteilung mit angeben.
3. `dBodySetMass ( km[0].body , &masse );`
   Damit muss nun die Massenverteilungsmatrix, auf welche in `masse` referenziert wird, noch dem entsprechenden Körper zugewiesen werden, für den sie angelegt wurde.

- Definieren und Zuweisen der Hülle der Körper:
  Bisher haben Körper keine räumliche Ausdehnung. Um dies zu ändern, und somit Kollisionserkennung zu ermöglichen, werden nun Geometrieobjekte, oder auch Hüllen genannt, angelegt. Diese müssen dann noch den entsprechenden Körpern zugewiesen werden.
  1. `km[0].geom = dCreateBox ( raum , 1 , 0.5 , 0.5 );`
     Für Geometrieobjekte muss man wie bei der Festlegung der Masse, eine entsprechende Funktion für jede Art von geometrischen Grundobjekten wählen, die man für seinen Körper verwenden will. Hier wird ein Quader-Geometrieobjekt angelegt. Der erste Parameter ist der weiter oben definierte Kollisionsraum. Alle weiteren Parameter beziehen sich auf die räumliche Ausdehnung des Geometrieobjektes, und sind von der jeweils gewählten Funktion abhängig (hier die räumliche Ausdehnung des Quaders in x-, y- und z-Richtung).
  2. `dGeomSetBody ( km[0].geom , km[0].body );`
     Hiermit wird das Geometrieobjekt einem Körper zugewiesen. Für diesen wird nun bei der Kollisionserkennung das angelegte Geometrieobjekt als räumliche Ausdehnung und Oberfläche verwendet.

- Beim Anlegen der Geometrieobjekte ist sicherlich aufgefallen, dass man auf diese Art nicht beliebig positionierte Grundobjekte im Raum anlegen kann. Zum Beispiel können Quader nur parallel zu den Achsen x, y und z angelegt werden. Aus diesem Grunde stehen einige Transformationsfunktionen, wie etwa für die Rotation zur Verfügung. Auf diese wird hier allerdings nicht näher eingegangen. Es wird auf den weit ausführlicheren ODE-user-guide verwiesen.
- Definieren und zuweisen von Joints:
  Nun können die angelegten Objekte noch untereinander verbunden werden. Auf diese Art lassen sich recht komlexe Konstrukte erzeugen. Hierfür werden sogenannte Joints angelegt. Es werden, zum Referenzieren auf diese Joints, Variablen vom Typ `dJointID` verwendet. Diese Variablen sind in der Template-Datei in einem Array zusammengefasst, um leichter auf sie zu referenzieren:

```
dJointID j[1];
j[0] = dJointCreateHinge ( welt , 0 );
```

  Hier wird ein Joint angelegt, welcher der Simmulationswelt zugeordnet werden muss, in der er für die Berechnung relevant ist. Ebenso erfolgt mit dem zweiten Parameter eine Zuordnung des Joints zu einer Jointgruppe. Joint-Gruppen können von Vorteil sein, wenn es darum geht mit größeren Mengen von Joints zu arbeiten, welche in einem ähnlichen Kontext verwendet werden. Zum Beispiel wenn eine große Gruppe Joints auf einmal gelöscht werden soll. Hierzu wird hier eine Variable des Typs `dJointGroupID` angelegt werden, welche hier als zweiter Parameter eingetragen wird. Wenn, wie im Template zu sehen, eine `0` eingetragen wird, erfolgt keine Zuordnung des Joints zu einer bestimmten Jointgruppe. Dies hat wiederum den Vorteil das der Joint ohne Probleme einzeln gelöscht werden kann. Im Template ist es nicht nötig an dieser Stelle eine Joint-Gruppe anzulegen.
  Es gibt verschiedene Arten von Joints. Die verschiedenen Joints lassen sich durch jeweils andere Funktionen, analog zum aufgeführten Beispiel anlegen. Je nach Joint variieren die zulässigen Bewegungen die der Joint ermöglicht. Hier die verschiedenen Joint-Typen:
    - Fixed
      Zwei Körper sind fest miteinander verbunden, ohne das sie ihre relative Positionen zueinander ändern können.
    - Slider
      Zwei Körper können sich entlang eines Vektors, der genau zwischen ihnen aufgespannt wird, bewegen.
    - Ball
      Zwei Körper sind mit einem Kugelgelenk verbunden. Es gibt einen Rotationspunkt.
    - Hinge
      Zwei Körper sind mit einem Scharnier verbunden. Es gibt eine Drehachse in einem Punkt.

- Hinge2
  Zwei Körper sind mit einem Gelenk verbunden, das eine Bewegung um zwei Achsen zuläßt. Diese beiden Achsen stehen senkrecht aufeinander. Die beiden Drehachsen können durch verschiedene Punkte laufen.
- Universal
  Wie Hinge2, mit dem Unterschied, dass hier beide Drehachsen durch den gleichen Punkt laufen.

Die beiden folgenden Joints sind Speziallfälle, die man für gewöhnlich nicht direkt verwendet um zwei Körper zu verbinden.

- Contact
  Diese Joints werden verwendet um für einen Zeitschritt einen Kollisions-Kontakt zwischen zwei Körpern herzustellen, wobei dann abstoßende Kräfte wirken.
- AMotor
  Diese Joints werden zusätzlich zu normalen Joints zwischen zwei Körpern angelegt. Sie können eine permanente Drehbewegung um eine Achse bekommen, und dienen somit als Motoren.

1.
```
dJointAttach ( j[0] , km[0].body , km[1].body );
```
Nun werden die zuvor angelegten Joints an die Körper geheftet, zwischen denen sie die Verbindung darstellen sollen. Der erste Parameter ist die Referenz auf den Joint, die beiden Letzten sind Referenzen auf die beiden Körper die der Joint verbinden soll.

2.
```
dJointSetHingeAnchor ( j[0] , (dBodyGetPositionAll ( km[1].
body , 1 ) - dBodyGetPositionAll ( km[0].body , 1 ))/2 ,
(dBodyGetPositionAll ( km[1].body , 2 ) -
dBodyGetPositionAll ( km[0].body , 2 ))/2,
dBodyGetPositionAll ( km[0].body , 3 ) );
```
Nun gilt es noch den Drehpunkt/Ankerpunkt des Joints festzulegen. Hierzu gibt es je nach Jointart ebenfalls wieder verschiedene Funktionen, bei denen einfach die Jointbezeichnung entsprechend angepasst werden muss. Der erste Parameter ist die Referenz auf den Joint, die anderen drei geben den Ankerpunkt in Weltkoordinaten an.
In diesem speziellen Fall wird hier eine weitere Funktion genutzt, die im Template angelegt wurde. Sie gibt die einzelnen Weltkoordinaten eines Körpers zurück. Hier wird also ein Ankerpunkt angelegt, der immer genau in der Mitte zwischen zwei Körpern liegt. Nur die z-Koordinate entpricht ausschließlich der des ersten Körpers.

```
3.dJointSetHingeAxis ( j[0] , 0 , 1 , 0 );
```
In Abhängigkeit des Jointtypen müssen nun eventuell weitere Parameter gesetzt werden. Im Fall das Beispiels im Template ist dies der Vektor der Drehachse. Der Funktionsname ist wiederum abhöngig vom jeweiligen Joint-Typ. Der erste Parameter ist die Referenz auf den Joint, die anderen drei geben den Vektor der Drehachse an.

- Definieren und Zuweisen der Winkelmotoren:
  Im Folgenden wird näher auf Jointss vom Typ AMotor eingegangen.

```
1.jm[0] = dJointCreateAMotor ( welt , 0 );
  dJointAttach ( jm[0] , km[0].body , km[1].body );
```
Diese beiden ersten Anweisungen sind Analog zu den Anweisungen im Joint-Abschnitt.
```
2.dJointSetAMotorMode ( jm[0] , dAMotorEuler );
```
Jeder Amotor kann in einem von zwei Modi sein, welche mit diesem Befehl gesetzt werde.
```
dAMotorEuler
dAMotorUser
```

Hier wird die automatische Winkelberechnung mittels `dAMotorEuler` verwendet. Der Motor berechnet den aktuellen Winkel, den er einnimmt automatisch. Man kann eine Drehgeschwindigkeit, sowie die Drehrichtung, welche gleichzeitig durch ein Vorzeichen angegeben wird, angeben. Dann kann man jederzeit den aktuellen Winkelwert des Motors abfragen. Siehe hierzu 5 und 6.
```
3.dJointSetAMotorAxis ( jm[0] , 0 , 1 , 0 , 1 , 0 );
  dJointSetAMotorAxis ( jm[0] , 2 , 2 , 0 , 0 , 1 );
```
Damit der Winkel auch berechnet werden kann, müssen zwei der drei aufeinanderstehenden Achsen gesetzt werden. Hierzu werden zunächst angegeben, welche der Achsen gesetzt werden sollen. Hier sind es die Achsen 0 und 2. Der dritte Parameter gibt an, dass eine Achse an Köper 1 bzw. an Körper 2 verankert ist. Die letzten drei Parameter geben den Raumvektor der jeweiligen Achse an. Es hat sich gezeigt, dass die erste angegebene Achse die Drehachse selbst sein muss, und die zweite eine Achse die zu dieser Drehachse im rechten Winkel steht. Welche Achse das ist, müsste je nach Beispiel probiert werden, um das gewünschte Ergebniss zu erhalten. Der ODE-userguide gibt hierzu nur etwas unklare Auskunft.

Wenn jemand die genaue Funktionsweise verstehen sollte, so kann er sich gern bei mir melden, und mir dies mitteilen.

4. `dJointSetAMotorParam ( jm[0] , dParamFMax,MAX_MOTOR_KRAFT);`
   Diese Anweisung kann verschiedne Parameter des Motor-Joints festlegen, in diesem Fall wird der Parameter `dParamFMax` festgelegt. Dieser gibt die maximale Kraft an, die ein AMotor-Joint aufwenden kann, um seine Bewegung durchzuführen.
   Es gibt noch eine Reihe weitere Parameter, welche im ODE-userguide nachzulesen sind.
5. `dJointSetAMotorParam ( joint , dParamVel , winkelgeschwindigkeit*MOTOR_WINKELGESCHWINDIGKEITSFAKTOR );`
   Diese Funktion wird im Template nicht in der Funktion main aufgerufen sondern in der Funktion `setMotorWinkel`. Diese Funktion setzt verschiedene Paramter für den Motor-Joint, näheres ist im ODE-userguide nachzulesen sind. In diesem Beispiel wird die Motor-Drehgeschwindigkeit gesetzt (zweiter Parameter). Der dritte Parameter setzt den Wert des zu verändernden Joint-Parameters.
6. `dJointGetAMotorAngle ( jm[n] , 0 );`
   Diese Funktion wird nicht in der Funktion main aufgerufen, sondern in `simLoop`. Hier wird der aktuelle Winkel des angegebenen Joints, auf der angegebenen Achse ausgelesen.

start ()

Alle Anweisungen, welche in dieser Funktion stehen, werden einmal zu Beginn der Simulation, also vor dem Start der simLoop-Schleife aufgerufen.
Hier wird die Startposition der Kamera als Weltpunkt angegeben, als auch der Punkt auf den die Kamera blicken soll. Desweiteren können verschiedene Variableninitialisierungen erfolgen.

simLoop (int pause)

Diese Funktion ist die eigentliche Simulationsschleife. Alle enthaltenen Anweisungen werden in jedem Simulationsschritt ausgeführt.
Im Template gibt es hier drei wichtige Hauptabschnitte:

- Simulationsberechnung
  Hierbei handelt es sich um die eigentliche Berechnung eines ODE Schrittes. Also um die Berechnung aller physikalischen Geschehnisse in der simulierten Welt. Hierzu gehört auch die Kollisionserkennung, die hier stattfindet. `dSpaceCollide ( raum , 0 , &nearCallback );` Ruft die Funktion `nearCallback` auf, sobald eine Kollision von zwei Geometrieobjekten festgestellt wurde. Näheres hierzu in der Beschreibung der Funktion `nearCallback`.

- Sensoraktualisierung
  Wie schon zuvor erwähnt, werden die Motor-Joints mit dem Befehl `dJointGetAMotorAngle` abgefragt. Hier erfolgt eine Speicherung in ein Array von Sensoren.
- Zeichenabschnitt
  Hier befinden sich die Grafikanweisungen aus der Bibliothek `drawstuff`, welche die einzelnen Körper zeichnen. Die einzelnen Zeichenfunktionen sind in der Lage die verschiedenen geometrischen Grundobjekte, die man als Hüllen wählen kann, darzustellen.

  - `dsSetColor (1,0,0);`
    Mit dieser Anweisung wird die aktuelle Zeichenfarbe durch Angabe der Rot- , Grün- und Blauwerte festgelegt.
  - `box[0] = 1; box[1] = 1; box[2] = 1;`
    Dieses Feld ist nötig, da die dsDrawBox Anweisung die daraufhin folgt, einen Zeiger auf ein solches Feld benötigt, um daraus die drei Seitenlängen des zu zeichnenden Quaders zu lesen.
  - `dsDrawBox ( dBodyGetPosition ( km[0].body ) , dBodyGetRotation ( km[0].body ) , box );`
    Es wird ein Quader gezeichnet, der erste Parameter ist ein Zeiger auf ein dreidimensionales Feld, in dem die Position der Grafik in Weltkoordinaten angegeben werden muss. Es wird hier gleich die Position des Körpers benutzt, für welchen die Grafik ohnehin gezeichnet werden soll, da die Funktion `dBodygetPosition` einen Zeiger auf ein dreidimensionales Feld zurückgibt. Ebenso wird als zweiter Parameter die zu verwendende Rotationsmatrix benötigt. Hierzu wird ebenfalls gleich die Rotationsmatrix des entsprechenden Körpers verwendet.

nearCallback (void *data, dGeomID o1, dGeomID o2).

Die Funktion `nearCallback` wird aufgerufen, wenn es zu einer Kollision zwischen zwei Geometrieobjekten kommt. Diese beiden Objekte werden als o1 und o2 der Funktion übergeben.
Hier werden nun Contact-Joints verwendet. Zwischen den beiden kollidierten Körpern wird dieser Kontaktpunkt angelegt. Es werden verschiedene Parameter für diesen Kontaktpunkt festgellegt, wie etwa die Reibung. Im Normalfall muss man an dieser Funktion, so wie sie im Template definiert ist, keine Änderung vornehmen um eine normale Simulation durchzuführen. Für genauere Details sollte deshalb an dieser Stelle auf den ODE-userguide verwiesen werden

command (int cmd)

Die Funktion wird immer aufgerufen, wenn innerhalb der Simulation eine Taste gedrückt wird. Alle Tasteneingaben werden als integer-Wert in die Variable cmd übergeben. Das Problem ist leider, das ODE nicht mit den normalen Eingabefunktionen wie `scanf` kombinierbar ist, da dies zu einem Programmfehler führt. Aus diesem Grunde wurde im Template von Hand eine Eingabefunktion geschrieben, die nach Eingabe eines Buchstaben in einen speziellen Eingabemodus wechselt, in dem einzelnen eingelesenen Zeichen zu einem String hinzugefügt werden. Die Eingabetaste ist die Leertaste, da einige Spezialtasten wie zum Beispiel Enter nicht von der Funktion command erkannt bzw. zugelassen werden.