Open Dynamics Engine Tutorial version 1
USM High Performance Visualization Center
Department of Computer Science

Matthew Bennett 11-27-03

*for more information on ODE, please refer to the user's guide at http://opende.sourceforge.net
This tutorial is not a definitive reference, and you will need to review the user's guide.
*also, if you speak Python, check out http://i31www.ira.uka.de/~baas/pyode/ for some great tutorials from Matthias Baas

What is ODE?

Open Dynamics Engine (ODE) is a cross-platform open-source library that is useful in creating simulations of simple rigid-body physical systems. A rigid body system is any physical system in which the individual basic units do not change shape or deform. The ODE engine has many application areas, including games, VR, and elementary physics simulation for research in areas like genetic algorithms for real world devices (see the links page on the ODE website for some cool applications).

ODE is distributed under the lesser GNU public license and the Berkeley license.

What are the benefits of using ODE?

The main benefit of using ODE in your project is that you don't have to write the physics code yourself. Instead, you can write code at a higher level ob abstraction, saving time, work, and headaches. With ODE, you can do things that would take you hundreds of lines of physics and math intensive code with just five or six lines of library calls. Here are some features, excerpted from the user's guide:

* Rigid bodies with arbitrary mass distribution.
* Several joint types: ball-and-socket, hinge, slider (prismatic), hinge-2, fixed, angular motor, universal.
* Collision detection with several collision primitives: sphere, box, capped cylinder, plane, ray, and triangular mesh.
* Collision spaces: Quad tree, hash space, and simple.
* A first order integrator. Very fast but not too accurate.
* The library works well on the IRIX (SGI), windows, and Linux environments, and claims portability to many other Unix based environments.
* interfaces for C, C++, Delphi, and Python (using pyode) are available

and several other features. Don't let the terminology used here scare you away, as I'll get to explaining some of this shortly.

Aside from saving you from having to write your own physics code, ODE also provides a collision detection system that is entirely internal, so that you can even use the collision detection it provides. Alternatively, you can write your own routines. ODE is a great way to go for a collision detection and response system.

What are the limits of ODE?

From my own work with ODE, I've noticed that ODE doesn't scale well above 40 or so separate bodies on our machines here in the visualization lab. Additionally, the website warns that ODE is not accurate enough for engineering tasks, though it is fast enough for games and graphical simulation demos. Also, keep in mind that ODE is a rigid body physics library, so it is not very adaptable to simulations in which bodies become deformed or change shape during the simulation.

Prerequisites for this tutorial

A basic college-level understanding of physics is vital to the use of this library or any physical simulation. Familiarity with concepts of conservation of linear and angular momentum, mass, force, torque, linear and angular velocity, inertia, and displacement are vital before the library's usage can be understood.

A general understanding of the basics of the OpenGL graphics environment is useful, but not totally necessary. ODE provides a library called drawstuff that lets the user play with ODE concepts without knowing much about computer graphics. For reference to OpenGL topics, visit http://www.opengl.org Also, familiarity with basic concepts of collision detection is handy.

The version of ODE used to build this tutorial was 0.039 stable. Any other version may work, but be cautioned in case of inconsistencies. The operating system used to develop this tutorial was RedHat 9.0 stable. The version of ODE currently working on the SGI is 0.035.

ODE must of course be installed before undertaking this tutorial. You can download and install ODE from http://opende.sourceforge.net Follow the installation instructions provided in the user's guide for your operating system.

The purpose of this tutorial is only to help you get your first steps working with ODE, and not to serve as a definitive guide on the subject. You must visit http://opende.sourceforge.net/docs and obtain a copy of the user's guide. A copy of the 0.039 stable user's guide is mirrored at borz under the account mbennett, but will be quickly outdated.

Basic parts of an ODE program

Most ODE programs will include the following items:

1. Global or static variable declarations to keep track of the "state machine" of ODE, including things like a list of geometry objects, a list of bodies, and a list of collision-bounding geometries such as walls.
2. Initialization area for the physical world, which is where joints, bodies, geometries, spaces, and worlds are created.
3. An event loop for the physical world,
4. If ODE's collision detection is used, a function to be passed as a callback to the collision detection routine whenever there is a possible collision
5. Other stuff not pertaining to the physical simulation portion (I/O, graphics, etc).

A jump into our tutorial code

The tutorial is a simulation of several balls bouncing around inside of a cube. The balls start out lined up in a row, each with a random angular velocity (spin), identical mass and diameter, and a random directional force applied to each, so that they will start colliding with each other. The initial spin is provided so that the collisions will be more interesting and realistic. The walls, floor, and ceiling of the cube act as bounding geometry planes so that the balls cannot roll or bounce out. A routine, pull(), is provided to keep the balls moving, can be called anytime by pressing the Enter key, or will be called automatically every 500 frames. When called, it places a centripetal force on all the balls so that they rush towards one another at the center and collide. If the force in continuously applied, circular motion occurs. Take a brief look at the source file, main.cpp , to familiarize yourself with the structure of the program, and then run the compile script provided to compile and run the program (on linux only).

Global declarations

```
#include <ode/ode.h>

//---- global physics stuff ----
const int numBodies = 6;
const int explosionTime=500;
//minimum of 2, max of 16

void explosion();
void pull();

int explosionTimer = explosionTime; //we use this to keep the balls from
stagnating
dReal balRadius = 48.0/numBodies;
dWorldID theWorld;            //ode world
dBodyID  bodies[numBodies];   //ode bodies
dSpaceID theSpace;            // we'll use only one space for collision
detection
dGeomID      bodiesG[numBodies];    //we'll need a geometry describing each
body
dGeomID  theFloor[2];         //we'll also need to detect a collision with the
floor/ceiling
dGeomID  walls[4];            //and some walls

dJointGroupID contacts;
dContactGeom contactAry[255]; //well need to store a list of contacts
temporarily
//we'll need to keep a group listing of all contacting points for collision
handling
```

We have to have a dWorldID for the demo as a whole, and a dSpaceID for the collision detection. Floor and walls of the bounding cube are stored as geometries only, since they will be static units, whereas we need an array of dBodyID and an array of dGeomID for the balls, since they will be hysically dynamic. Basically, Each object in our scene needs a dBodyID for the physics part and a dGeomID for the collision detection part. We can associate these two later in the initialization phase of our program. We also need to keep a list of contact joints for the collision detection later, so that ODE can keep track of which bodies had a collision and respond accordingly in a collision response phase (nearCallback) separate from our collision detection phase.

ODE Initialization phase

We use a function for this, called InitPhysics(), but you can call it whatever you like.

```
void InitPhysics()
{
 theWorld=dWorldCreate(); //in the beginning...
 dWorldSetGravity(theWorld, 0.0, -9.8, 0.0); //and there was gravity

 dWorldSetERP(theWorld, 0.8);  //set the error correction to 80% NECCESSARY
 dWorldSetCFM(theWorld, 1.0E-5); //set the constraint force mixing

 //collision-detection stuff
 theSpace = dSimpleSpaceCreate(theSpace); // Create a space object
 //create geometries within the space
 //the eq for describing a plane is Ax+By+Cz=D
 //the syntax for dCreatePlane is dCreatePlane(dSpaceid ___, dReal A, dReal B,
dReal C,dReal D)
 theFloor[0] = dCreatePlane(theSpace, 0.0, 1.0, 0, -50.0);
 theFloor[1] = dCreatePlane(theSpace, 0.0, -1.0, 0, -50.0);
 walls[0] = dCreatePlane(theSpace,1.0,0.0,0.0,-50.0);
 walls[1] = dCreatePlane(theSpace,-1.0,0.0,0.0,-50.0);
 walls[2] = dCreatePlane(theSpace,0.0,0.0,1.0,-50.0);
 walls[3] = dCreatePlane(theSpace,0.0,0.0,-1.0,-50.0);

 contacts=dJointGroupCreate(0); //create an empty group of contacts for
sotring contact joints

for (int i=0; i<numBodies;i++)
{
 bodies[i]=dBodyCreate(theWorld); //create a body in this world
 dBodySetPosition(bodies[i], (i+1)*balRadius*2.0-numBodies*balRadius-
.5*balRadius,
  stepBack/3.0, 0.0);
 //put it at the origin, and located in some place along X based upon index
//set the position to some place on a line that bisects the center of the
bounding cube
 dBodySetAngularVel (bodies[i], rand()%5/8.0, rand()%5/8.0, rand()%5/8.0);
//randomize its spin so that collisions will be more interesting

/* //the following are not used, but have been provided to demonstrate syntax
 dBodySetLinearVel(bodies[i], 0.0, 0.0, 0.0 );
 dBodyAddForce(bodies[i], 0.0, 0.0, 0.0);
*/
 bodiesG[i] = dCreateSphere(theSpace, balRadius);
 //we want each body's associated geom to be that of a sphere of radius 1
 dGeomSetBody(bodiesG[i], bodies[i]);
}
```

```
}
```

Here, we must first create a world, assigning its value to the global variable we set up earlier. We'll need to keep track of this as several function calls later require it. Then we set gravity in our world to 9.8 (units of length)/(time units squared) downward along the Y axis. It should be noted that ODE has no default modeling units, so here we use mock metric units, but you can use whatever you like in your code. Mock units generally help keep everything in perspective.
The error correction and constraint force mixing control how accurate the simulation is. They should not be set too high or too low, or the simulation may become unstable or too slow. Play with the values passed in if you want to see what sort of effect they have.
We then set up some collision boundaries as geometries within the space we created as a global variable. Here, the bounds we want form the inside of a cube, so infinite planes can be used to model the sides. Most of the time, infinite planes are not suitable for modeling collision objects (except for perhaps a flat desert). Other collision geometries provided include sphere (we'll use this one in a bit), rectangular prism, capped cylinder, and any composite of these basic 3d shapes. The standard equation for a plane is Ax+By+Cz+D=0, but these functions want the form Ax+By+Cz=-D, so the D passed in as the fifth argument to `dCreatePlane` should be negative.
We then set up a body for each ball in the simulation, give it a position and spin, create a geometry for that body, and associate the geometry we've just created with the body we've just created using `dGeomSetBody`. Easy, huh? The for loop allows processing of every body we want in the scene.

What to do each step

```
void phyTurn() //gets called each frame
{
for (int i = 1; i<numBodies; i++)
{ dSpaceCollide(theSpace, contacts, nearCallback); }
//do collision detection for all bodies using the function nearCallback when
there is a likely collision
dWorldStep(theWorld, dt); // step the world
dJointGroupEmpty(contacts); //empty the list of joint contacts
```

This takes care of our collision detection. First we call ODE function `dSpaceCollide` to see if there are any likely colliding objects. If there are, the callback nearCallback we have defined elsewhere gets called (I'll get to this soon). Then, the world is "stepped" using `dWorldStep` with a "time step" dt. When this happens, the forces on each body are applied by ODE and the positions are updated. Finally, the group of contact joints (which bodies are touching and therefore need to respond to some collision) is emptied ou so that it can be used again on the next iteration.

Then, the scene is drawn, which isn't technically part of ODE, but some sample code is provided here that shows how to extract position and rotation information for all bodies in the scene, for drawing or output purposes.

```
for (int i=0; i< numBodies; i++)
{
 glPushMatrix();
  const dReal * pos = dBodyGetPosition(bodies[i]);
  const dReal * rot = dBodyGetRotation(bodies[i]);
  GLfloat matrix[16];
  matrix[0]=rot[0];
  matrix[1]=rot[4];
  matrix[2]=rot[8];
  matrix[3]=0;
  matrix[4]=rot[1];
  matrix[5]=rot[5];
  matrix[6]=rot[9];
  matrix[7]=0;
```

```
  matrix[8]=rot[2];
  matrix[9]=rot[6];
  matrix[10]=rot[10];
  matrix[11]=0;
  matrix[12]=pos[0];
  matrix[13]=pos[1];
  matrix[14]=pos[2];
  matrix[15]=1;
  glMultMatrixf (matrix);

}
```

Since the dBodyGetRotation routine returns a 4x3 rotation matrix, it is not directly what we want for drawing in OpenGL. The above code, however, translates it for us, and is roughly based upon the python ode tutorial code of Matthias Baas, available at http://i31www.ira.uka.de/~baas/pyode/ .

The nearCallback function for collision response

```
void nearCallback (void * data, dGeomID o1, dGeomID o2)
{
 int i;
 const int MAX_CONTACTS = 1;
 dBodyID b1 = dGeomGetBody(o1);
 dBodyID b2 = dGeomGetBody(o2);
 if (b1 && b2 && dAreConnectedExcluding (b1,b2,dJointTypeContact)) return;
```

If the bodies are connected by a joint, then we don't need to do collision response, as they are supposed to touch.

```
 dContact contact[MAX_CONTACTS];   // up to MAX_CONTACTS contacts per box-box
 for (i=0; i<MAX_CONTACTS; i++)
 {
  contact[i].surface.mode = dContactBounce;
  contact[i].surface.mu = 5000.0;
  contact[i].surface.bounce = 0.8;
  contact[i].surface.bounce_vel = 0.8;
 }

if (int numc = dCollide (o1,o2,MAX_CONTACTS,&contact[0].geom,
sizeof(dContact)))
 {
  dMatrix3 RI;
  dRSetIdentity (RI);
  const dReal ss[3] = {0.02,0.02,0.02};
  for (i=0; i<numc; i++)
  {
   dJointID c = dJointCreateContact (theWorld,contacts, contact+i);
   dJointAttach (c,b1,b2);
  }
 }
```

Here is the code for reacting to a possible collision. Parameters for a joint are set up (surface friction and "bounciness"), and then for each collision between two bodies, a contact joint is created so that ODE can handle applying appropriate forces before the next world step.

Sources

Some of my code structure, the nearCallback and pull routines came from Matthias Baas, http://i31www.ira.uka.de/~baas/pyode/

The ODE user's guide, version 0.039, at http://opende.sourceforge.net/ode-0.039-userguide.html