pyode - Rigid Body Dynamics for Pure Data

Frank Barknecht
Neusser Wall 2
50670 Cologne
Germany
fbar@footils.org

ABSTRACT

For several years simulations of particle systems according to Newtonian laws are possible in Pd thanks to the PMPD and MSD externals. Both systems however only deal with point masses. The pyode Python external for Pd is a realisation of rigid body dynamics and animates bodies with volume, orientation and various surface characteristics. pyode for Pd is based on the Open Dynamics Engine (ODE), a C/C++ library that is well documented and easy to integrate in various programs. For the Pd object, ODE's Python wrapper was used, because it allows fast prototyping and easy customization. Porting the external to C or C++ could be an interesting project for the future.

Keywords

Pure Data, Python, Physical Simulation, Rigid Bodies, Particle Systems, Open Dynamics Engine

1. FROM POINTS TO BODIES

When referring to particle systems in this text, I follow the simple definition, that Andrew Witkin used to start his "An Introduction to Physically Based Modeling"[4]:

> "Particles are objects that have mass, position, and velocity, and respond to forces, but that have no spatial extent."

PMPD[1] and MSD[2] are two externals for Pd that help with simulating such particle or point mass systems. In addition to simple particles they provide "links", virtual springs with rigidity and damping, that can be used to connect two particles. Both externals have shown their usefulness in several areas of Pd ranging from graphics generation, helper for synthesis algorithms (e.g. scanned synthesis), parameter generators for scores or creating flexible and natural controller mappings.

[1]http://drpichon.free.fr/pmpd/
[2]http://drpichon.free.fr/msd/

Pd Convention 2007. Montréal, Québec, Canada.

While PMPD and MSD allow to create structures with a spatial extent by connecting various point masses with springs, these structures only poorly approximate a real rigid body. Either the strings aren't stiff enough to avoid deformations of the (then not really) rigid body, or the strings may be too stiff, which lets numerical errors grow and the whole system quickly becomes unstable.

Unfortunately realistically simulating the behaviour of rigid bodies requires much more complicated maths (cf. [1] and [2]) than the simple point masses physics of PMPD/MSD. Thankfully physicists and engineers were in need of computer-simulated rigid body dynamics long before us audio-visual artists. And thankfully++, in computer games physics play an important roles as well: The kids grew out of playing the particle-based "Pong".

A result of this is the availability of several high-performance libraries for rigid body simulations. One of the popular, open source tools for this is the Open Dynamics Engine ODE[3], which I chose to use in the pyode-Python-external for Pd

2. PYODE

According to its own description, "ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools."

What's good for computer games, should be good for audio-visual art as well. The current implementation of ODE inside Pd uses the Python-wrapper around ODE called PyODE[4]. The decision to not write a C or C++ external was mainly driven by the fast turnaround cycle of developing an external in Python compared to development in C, where the "write, compile, reload" cycle takes more time. It wasn't clear from the start what kind of "API" the Pd object should present to its user, that is, which messages it would accept and generate. Through using Thomas Grill's pyext[5] a rapid prototyping approach was possible during development of the external.

Additionally I assumed, that most of the computational burden would lie in the physics simulation, so that the over-

[3]http://www.ode.org
[4]http://pyode.sf.net/
[5]http://grrrr.org/ext/py/

head of running a Pd object through Python would be negligible compared to that. The Python wrapper for ODE in the end will compute the physics simulation through the C/C++ library anyways, so it should be of comparable speed. But I admit, that I didn't run any comparison benchmarks yet.

3. ODE IN PD

Typically simulations of particle systems have to deal with four tasks:[3]

- Creation
- Termination
- Animation
- Visualisation

It seems sensible to start with a similar approach when structuring the implementation of rigid body systems.

3.1 Creation

Creation is at the beginning: A world with various parameters like gravity or air drag has to be defined. The world then needs to be populated with bodies, that have various characteristics as well. In MSD these bodies are all point masses. Their only difference is the initial position and their mass. Rigid bodies however additionally have a shape and an orientation and possibly a surface friction, that have to be defined as well.

Creation in pyode for Pd is handled in a similar fashion as in MSD: A single Pd object holds a complete world of physical bodies. Bodies are created through messages to one of the object's inlets. A message to create a body may look like this: "body box MyBox 0 10 0 90 0 1 0 1000 1 0.1 2". Help for the parameters of a message can be accessed by reading the "docstring" of this method in pyode for Pd. For the "body"-message this gives:

> shape: box, sphere, etc.
>
> id: a name for the body, used as a key in self.bodies. Has to be unique.
>
> Position of center: x, y, z
>
> Orientation: angle (deg), rx, ry, rz
>
> args: further properties of the body, that will get passed to a method '"self.create_%s" % shape'.
>
> Use this to set special properties like density, size of a box, radius of a sphere etc.

As ODE provides several possible body-types like box, sphere, cylinder or even a triangle mesh, the implementation will automatically search for a special "create_X"-method, if it receives a "body X ..." message, and then calls "create_X with the supplied parameters (using Python's "getattr" function). A message "body box ..." will look for "create_box", a message "body cylinder ..." will look for "create_cylinder". If the method doesn't exist, an exception will be raised and an error will be printed to Pd's console.

Similar message-methods are available to create what ODE calls "geoms" (geometry objects similar to bodies but without a mass) or to influence parameters of the contained world.

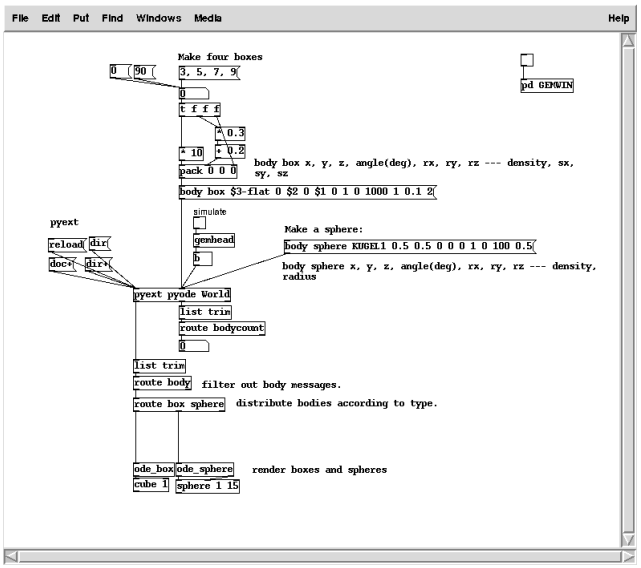

Figure 1: Example patch illustrating how to populate the ODE-world with some boxes and a sphere.

3.2 Termination

Termination has to deal with removing objects from the world again after they've reached a certain age. This is not currently implemented in pyode, but a "delete_body" method is planned.

3.3 Animation

Animation then is the phase, where the behaviour of the world and the contained actors is simulated according to the rules and differential equations of a dynamics system. The simulation looks at all the bodies' positions, velocities (both angular and global), accelerations, forces and collisions, and then steps the whole world one time-step further, while calculating the effects of speeds, forces, collisions etc. on the bodies and geometries.

Advancing the world simulation in Pd is done through a "bang"-message. The corresponding Python method basically just asks ODE's "quickStep" method to do all necessary calculations.

3.4 Visualisation

In Pure Data, Visualisation may also mean sonification or something more exotic. From a pure-data-centric point of view visualisation just are the messages, that leave the pyode object's outlets. In my implementation the messages are lists: After every simulation step is complete, the positions, speeds etc. of all existing bodies is sent to pyode's outlet. Drawing an ODE body for example will return a list composed of:

- Position (3): x,y,z
- Orientation (4): angle, rx, ry, rz
- Shape-specific data: e.g. size for a box as a list of 3 lengths.

Several ready-made Pd abstractions that are part of the pyode for Pd distribution simplify parsing these lists (which

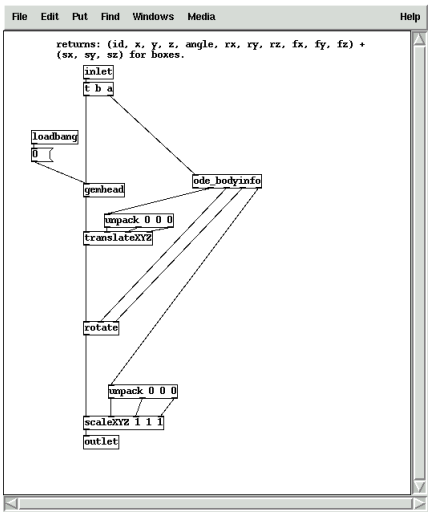

Figure 2: Helper abstraction ode_box.pd to render boxes.

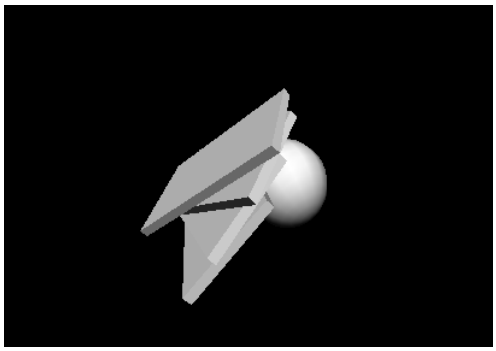

Figure 3: Gem rendering the example patch.

can get quite verbose) and can automatically map them to commands for Gem-objects like [translateXYZ].

The abstraction ode_bodyinfo.pd contains a chain of [list split] operations that distributes data common to all bodies (like position vector, angle and vector of rotation) to various outlets, and it passes the specific data like the radius of a sphere along to the last outlet. It is used in body-specific rendering abstractions, for example ode_box.pd, to finally make rendering of various bodies in GEM easy. The rendering abstractions like ode_box.pd only modify the current GEM pointer and they don't contain final primitives for rendering. This way it still is possible for example to use an OpenGL cube for displaying what in ODE is modelled as a sphere. The rendering objects automatically render as many copies of a body as necessary using forced rendering in GEM with a "double gemhead".

4. JOINTS INSTEAD OF LINKS

ODE holds a surprise for users of the PMPD/MSD externals when it comes to creating links between masses. ODE doesn't just have one catch-all link type, instead it provides various kinds of Joints. Joints restrict the possible degrees of freedom when moving a body that is joined with another body. Just like a finger on a human hand can only move in a certain direction and a certain range - without breaking - and just like a human head can do different movements, different joints in ODE have different characteristics. Because of this Joints need to be chosen carefully. In the pyode-Pd object joints are created with special messages similar to the creation of bodies. One of the possible types has to be selected as well as the bodies to connect with a joint. For example a "ball" joint always has a constant length and allows twisting movements in all direction, a "hinge" joint restricts torque to one direction only, while a "slider" joint doesn't allow for torque, but it can change its elongation.

Contact joins are a group of special joints that are automatically created and deleted, when two objects touch each other. These are used internally for the collision detection, which is active as default. Through various messages to the pyode object, parameters for the collision, like the bounciness of a collision, can be set while the simulation is running.

Similar to the abstractions for bodies, a set of abstractions to make rendering joints with GEM easier is included.

5. FUTURE

The prototype of pyode for Pd so far is working fairly well. Performance-wise it doesn't seem to be necessary to go with a C/C++ external for now, as the Python version can easily calculate about 100 bodies on my test machine, a AMD Athlon64 Processor 3000+. Still a respectable part of the ODE-API doesn't have an equivalent in pyode for Pd, especially Motors are completely missing. Filling the holes is planned for the near future.

6. REFERENCES

[1] D. Baraff. An introduction to physically based modeling: Rigid body simulation 1: Unconstrained rigid body dynamics. `http://www.cs.cmu.edu/~baraff/pbm/rigid1.pdf`, 1997.

[2] D. Baraff. An introduction to physically based modeling: Rigid body simulation 2: Nonpenetration constraines. `http://www.cs.cmu.edu/~baraff/pbm/rigid2.pdf`, 1997.

[3] G. Trogemann and J. Viehoff. *CodeArt*. Springer, Wien [u.a.], 2005.

[4] A. Witkin. An introduction to physically based modeling: Particle system dynamics. `http://www.cs.cmu.edu/~baraff/pbm/particles.pdf`, 1997.