Enhanced Collision Detection in Crystal Space (Reviewed project proposal)

Flavius Gruian & Magdalene Grantson
December 13, 2001
Reviewed January 8, 2002 (track changes via the red sidebars)

1. Introduction

1.1 Motivation

Detecting colliding objects is a must in any graphic engine or simulation environment. Real-time 3D simulations, consisting of complex scenes with many moving objects, require fast collision detection algorithms in order to keep a high frame rate. More complexity is added if one intends to handle variable size bodies, i.e. deforming bodies.

Moreover, there has to be a tight connection between collision detection and the part implementing the kinematics/physical behavior of the simulated universe. From this point of view, the collision detection part has to be able to provide enough information about each collision, information used to recompute the accelerations, speed, or/and forces that change the behavior of the object. A good separation between the collision detection part and the timely behavior of the universe (kinematics, physics) is also required if one expects to employ various approaches for any of these two parts. Having an efficient interface (API) between these plug-ins is, thus, a must. These are just some of the problems to be overcome in writing a good collision detection plug-in.

1.2 Idea

The focus of this project is on enhancing the already existent Crystal Space [3] collision detection plug-in. The current plug-in is based on RAPID [1] and can only handle a small to moderate number of rigid objects. In an initial phase, it constructs a object oriented bounding box (OBB) hierarchy for each collider[1]. At runtime, the user is expected to perform collision queries between pairs of colliders. In a bad implementation, every pair of dynamic objects in the universe have to be checked for collisions. In practice, only “close” objects can collide, so only these should be tested for an accelerated query. There are several approaches that for detecting “close” object pairs, such as:

- use a grid decomposition of the universe and only test objects located in adjacent cells
- use a cache of previously “close” objects and start testing these. Use these results to reason about the closeness of other objects. For example, if A is close to B but rather far away from C, it is very likely that B is far away from C so there would be no collision between B and C.

Combinations of these and other approaches can also be imagined. One example is V-COLLIDE [2].

1. A collider is a polyhedra that can collide with other colliders. It does not necessarily have to have the same geometry as the object it represents. Note also that not all visible objects must be reported as colliders.

Another drawback with the current collision detection plug-in, is the lack of any information about the surface (or point) of impact between the two colliders. We just know if the objects collide or not. This is not enough to derive a reasonable physical model. We plan to extend the current collision detection mechanism with some feedback about the impact surface (volume, point). This improvement should be usable with the current **csPhyziks** library, but general enough to be used with any other physics plug-in.

Finally, the existent **iCollideSystem** plug-in can not handle dynamic meshes. That is, objects with growing size or deforming features. This would imply recomputing the OBB hierarchy each time the object deforms, which can be quite time consuming. We aim at including a fast way of recomputing an OBB hierarchy after a deformation. This should be the first step towards adding soft bodies in the csPhyziks library.

2. Prototype

2.1 Crystal Space Extensions

We will modify the **iCollideSystem** plug-in to include accelerated queries, support for deforming bodies and return the point/surface/volume of contact. If the first enhancement may be transparent to the user of the collision system, the next two require a well defined API. More precisely the interface would be something as follows:

Accelerated Query. (also referred to as “frame coherence”) Functions for setting and updating the location of the objects and possibly their orientation too:

void iCollisionSystem::SetPostion(Collider object, csVector3 position)
void iCollisionSystem::UpdatePosition(Collider object, csVector3 position)

These are required for accurate tracking of “closeness” between objects.

Deforming Bodies. We will assume that only the position of certain vertices can change (i.e. no vertices or edges are inserted or deleted). Every time a deformation occurs, meaning some vertices changed their position, relative to the other vertices of the same object, there should be a way to report this back to the collider for that object. A list of pairs containing old position and the new position of a vertex have to be reported in this case:

void Collider::Difform(list_of_vertex_pairs)

Our implementation will have to identify the OBBs that were affected and adjust the hierarchy correspondingly. Initially we will simply recompute the whole bounding volume hierarchy, since we build on the RAPID algorithm that uses OBBs. A better choice, which can use spatial locality, would be to use AABBs. In that case only a certain branch of the hierarchy will change, meaning we can rebuild the hierarchy faster, as in SOLID [4]. Unfortunately that would mean using an AABB based algorithm, (not RAPID) that we would have to build from scratch instead of using RAPID. Depending on the available time left to finish the project we may consider implementing such an algorithm, thus accelerating the hierarchy recomputation.

Surface of Contact. Any collision check between two objects should return the point/surface or volume of contact in some way. In principle, only impact forces and their origin points are interesting. This is the least clear part of the project, and we have to do more literature survey before we can define the right approach. This part may prove to be less complex for convex objects and

depending on the time remaining after the demo milestone, we might decide to implement it partially or not at all.

2.2 Demo Application

The demo application(s) should show the power of our enhancements. For the first enhancement we need a large number of colliders - i.e. a room full of bouncing balls, possibly different in size, that can collide with the walls and among them. We chose balls (spheres) since the point of contact in this case is very easy to compute, in case we can not finish the last enhancement in time. As an extension to the Pong3D application that we already have, we could include a large number of bouncing ping-pong balls or other objects (i.e. cubes) in the room besides the existing objects.

For the last two enhancements, the best demo would be a deforming L shaped object falling down some stairs. With each collision, the shape would change and finally at the end of the staircase, the L shape will be all crumbled together.

If we implement only the first two enhancements, deforming we could make a demo with a lot of bouncing objects (spheres) changing sizes, say a Pong3D with ping-pong balls with variable x, y, z dimensions.
Finally, the demo will depend on how much we manage to implement before the demo deadline.

3. Time Schedule

3.1 Work Steps

1. Define a data structure that will keep track of where objects are, such that at a particular query, objects which are outside the vicinity of the query will be ignored for collision detection.
2. Define a (fast) way of recomputing an OBB hierarchy after a deformation. If an object's geometry changes (deforms) one will have to recompute its bounding volume. If it decomposes several time it is natural that we think of a fast way of recomputing their bounding volumes.

The following steps depend on the progress on the previous two. We will only implement them if we consider that we have enough time. Alternatively we could implement them after the demo deadline, meaning the demo will not be using them.

3. Extend the **RAPID** collision detection plug-in to output the surface or point of impact.
4. Instead of RAPID, implement and use another collision detection plug-in based on AABB in order to be able to recompute the bounding volume hierarchy faster.

Our aim will be to extend the **RAPID** or a novel collision detection plug-in which will also be useful in the **csPhyziks** library.

3.2 Milestone Demo

Using **csPhyziks**, define Pong3D room full of bouncing balls of different sizes, possibly size changing too. The balls collide with the floor, walls, rackets, table, net, and among them. Compare the speed between using only **RAPID** (check each pair of balls) and our own enhancement for different number of balls ranging from 10 to 1000.

4. Report Outline

The report will take the following format:

- Introduction and abstract giving one a basic idea of the problem being addressed.
- Elaborate on existing work in collision detection
- Problem at hand and motivation for our work
- Present the data structure and algorithms we plan to use
- Results. We will compare the time complexity of our proposed algorithms with that of **RAPID** and do some time complexity analysis on each algorithm
- Attach code, class diagrams and specific comments for our implementation

5. References

[1] OBB-Tree: A Hierarchical Structure for Rapid Interference Detection, S. Gottschalk, M.C. Lin and D. Manocha Technical report TR96-013, Department of Computer Science, University of N. Carolina, Chapel Hill. Proc. of ACM, Siggraph'96.

[2] V-COLLIDE Website: http://www.cs.unc.edu/~geom/V_COLLIDE/index.html

[3] Crystal Space Documentation at http://crystal.linuxgames.com/

[4] SOLID http://www.win.tue.nl/~gino/solid/