Preliminary version: March 1, 2002

ECoDS: An Enhanced Collision Detection System for Crystal Space

Flavius Gruian and Magdalene Grantson
<firstname.lastname@cs.lth.se>

Abstract
The present report describes an implementation of an N-body collision detection system, similar to V-COLLIDE built on top of a 2-body collision detection algorithm. Our system uses the Crystal Space graphic engine, and its RAPID collision detection plug-in. The system makes use of frame coherence and a collision cache to speed up the collision detection procedure. Moreover, it uses a fast method to detect and exclude from further processing pairs of objects that can not collide. The experimental results, based on a demo application, show that our Enhanced Collision Detection System (ECoDS) can significantly speed up the collision detection procedure in Crystal Space. The reduced complexity of the implementation makes it a good choice for developers interested in fast collision detection.

1. Introduction

Collision detection deals with geometric contact between distinct objects. It is an important concept in many fields in computer graphics and computational geometry like simulation, and animation [3]. In this paper we present an efficient, real-time algorithm to handle collision detection of several thousands of objects made up of collection of polygons with no topological information. Our system makes extensive use of the Crystal Space 3D graphic engine [2] written in C++. Our basic aim is to speed up the collision detection procedure in Crystal Space. We handle collision detection in two phases.

In the first phase, we find potentially colliding pairs of objects in the scene graph, making use of axis-aligned bounding boxes(AABB). We sort the end points of the AABBs into three lists L_x, L_y, L_z (one for each coordinate axis). The lists L_x, L_y, L_z are re-sorted at each frame using insertion sort. The choice of insertion sort, is due to the fact that L_x, L_y, L_z are partially sorted and insertion sort gives a linear time complexity if lists are partially sorted. Only pairs of objects whose AABBs overlap in all three dimensions are sent to the second phase to check for real collisions.

The second phase deals with detecting whether potential colliding pairs really collide. We make use of RAPID collision detection plug-in from Crystal Space. The bounding volume used here is the oriented bounding boxes(OBBs). To check for collisions between pairs of objects sent from phase one, RAPID traverses the OBB hierarchies to find boxes which overlap.

The report is organized as follows: We provide an overview of related work and advantages/drawbacks/usability of other systems and our system in section 2. An overview of ECoDS is given in section 3. Section 4 presents our implementation and describes specific issues such as algorithms, limitations, and how to use ECoDS in CS. We compare the performance of ECoDS with a pair wise RAPID based method in section 5. Finally, in section 6 we summarize our contributions and results.

2. Collision Detection Basics

Most collision detection systems deal with scene graphs made up of hierarchies of bounding volumes. Some of the hierarchical structures used here include cone trees, k-d trees, octrees, sphere trees, etc. The bounding volumes used can be axis-aligned bounding boxes (AABB), spheres, and oriented bounding boxes (OBB). The choice of bounding volumes depends on the application requirements.

In recent years there has been several implementations of CD algorithms, such as DEEP, SWIFT++, PIVOT, H-COLLIDE, RAPID, PQP, V-COLLIDE, I-COLLIDE, IMMPACT [1, 4]. Each of these packages are suitable for specific tasks (See Table 1 for advantages/drawbacks/usability). ECoDS is similar to V-COLLIDE, having only the difference that it builds on top of RAPID in Crystal Space. We implemented something similar to V-COLLIDE because of its performance and advantages/usability stated below and our objectives.

Table 1: A Few Collision Detection Algorithms

RAPID	Works with polygon soups. Requires that models are composes of triangles. Suitable when one has a small or moderate number of complex polygonal models and wants to make pair-processing queries explicitly.
I-COLLIDE	Works only for convex polyhedra models. Collision query times are extremely fast when models move only a relatively small amount between frames. It uses an n-body processing algorithm, and returns the distance between intersecting pairs of objects.
DEEP	It is an incremental algorithm, which estimates the penetrating depth between convex polytopes along and the associated penetration direction.
SWIFT++	Provides proximity queries such as intersection detection, tolerance verification, exact and approximate distance computation and contact determination of general 3-D polyhedral objects undergoing rigid motion.
PIVOT	This is a 2D proximity engine. The proximity queries in PIVOT include detecting collisions, computing intersections, separate distances, penetration depths and contact points with normals.
H-COLLIDE	This is a framework for fast and accurate collision detection for haptic (touch-enabled) interaction.
PQP	Provides support for distance computation and tolerance verification queries. Its API is similar to that of RAPID and gives the client program the flexibility of using more than one bounding volumes for a given query.
IMPACT	Provides interactive collision detection and proximity computations on massive models composed of millions of millions of geometric primitives.
V-COLLIDE	An n-body processor built on top of RAPID system. It decides which pairs of models are potentially in contact and for each potential contact pair it uses RAPID to determine true contact pairs. It remembers where all models are. It works with polygon soups. The client program can add or delete models from the collection managed by V-COLLIDE.
ECoDS	An 2-body processor built on top of RAPID system in Crystal Space. It decides which pairs of models are potentially in contact and for each potential contact pair it uses RAPID to determine true contact pairs. It remembers where all models are. It works with polygon soups. The client program can add or delete models from the collection managed by ECoDS.

3. Overview of ECoDS

Starting from an existing 2-body collision detection algorithm our intent was to build a higher level manager that makes calls to the 2-body collision detection function only when necessary. The reasonable assumption was that a flat collision detection system (Figure 1.a) would take more time since it has to check all object pairs for collision. A higher level management strategy (Figure 1.b) would likely filter the unwanted/redundant checks, requiring thus less processor time. Our system, ECoDS implements the N-Body level management strategy. It uses a fast pre-detection phase to identify objects that are unlikely to collide. It employs frame coherency to speed up this pre-detection phase. It also makes use of a collision cache for frame-wise stationary objects.

We wanted our N-body collision detection system to be as flexible and modular as possible. In principle it can use any two body collision detection available. Moreover, the developer is able to use different collision detection algorithms for different bodies without affecting ECoDS. In the demo described in section 5, we used the RAPID plug-in already available in the Crystal Space distribution. An schematic overview of ECoDS is depicted in Figure 2.

The main data structures in ECoDS are the lists of AABB Projections on all three axis, the Collision Cache and the Potentially Colliding Objects (PCO) List. For each collider registered with ECoDS (**Register Collider**) we extract the AABB of its object and use its projections on the Ox, Oy, and Oz. These three pairs of values are stored in the corresponding lists of AABB Projections. These lists have to be ordered to get the overlaps on all three axis. Whenever two AABBs overlap on all three axis, they are marked as potentially colliding and passed on to the PCO lists. If a collider moves (or there is a general change in the mesh that may affect the AABB) ECoDS must be informed (**Report Moves**). If an object is marked as moved, the information stored in the Collision Cache will be considered invalid, its AABB is recomputed but no reordering is performed yet. Whenever the applications wants to find out about new collisions, it has to call **CheckCollisions**. At this moment, the AABB Projection lists are reordered while tracking the AABB overlap and updating the PCO list. After reordering, for any pair of colliders that did not move, the cached collision information is used. Otherwise, if at least one object moved (the cache is invalid) we have to call the 2-Body collision detection system for that pair. This 2-Body collision detection algorithm can be for example the "RAPID" plug-in. It is up to the developer to decide (by calling

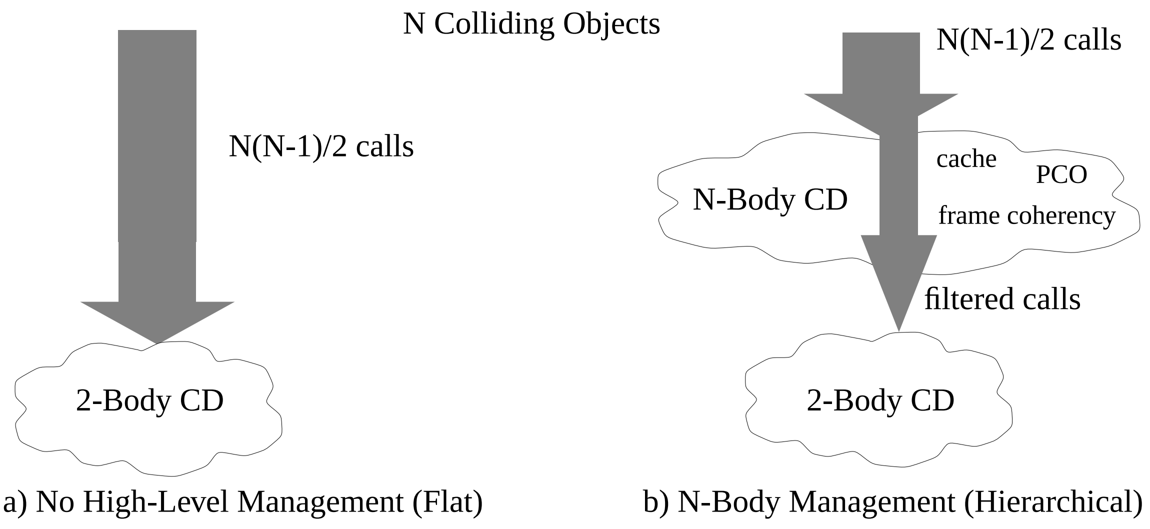


Figure 1: Flat and Hierarchical Collision Detection Systems

RAPID or otherwise) whether the two bodies collide. The result of this call is then stored in the collision cache for later use.

4. Implementation and Usage

The current implementation of ECoDS consists on several classes which have to be compiled with the application using the collision detection system, but we believe it can be easily transformed into a plug-in.
As depicted in the class diagram (Figure 3) ECoDS classes are rather disconnected from Crystal Space making the implementation independent of the 2-body collision detection algorithm. The n-body collision detection manager is implemented in **xCoDS**. All colliders must be registered with this class using **setCollider()**. Besides the list of colliders, **xCoDS** also harbors the lists of AABB projections on the three axis and a collision cache.

Collisions are detected and handled via calls to **updateCollisions()**. Once called, this function re-sorts all three projection lists using **resort()**. Sorting is implemented using insertion sort since this seems to be the fastest method for partially ordered lists. This is where frame coherency comes in: since the objects move rather little between two frames, the projection lists remain rather ordered with probably only a few out of order values. Although insertion sort behaves badly on random sets of values, it is much faster on partially ordered lists compared to more fancy algorithms [1]. Furthermore, while resorting of the list the potential collisions are also tracked. When-

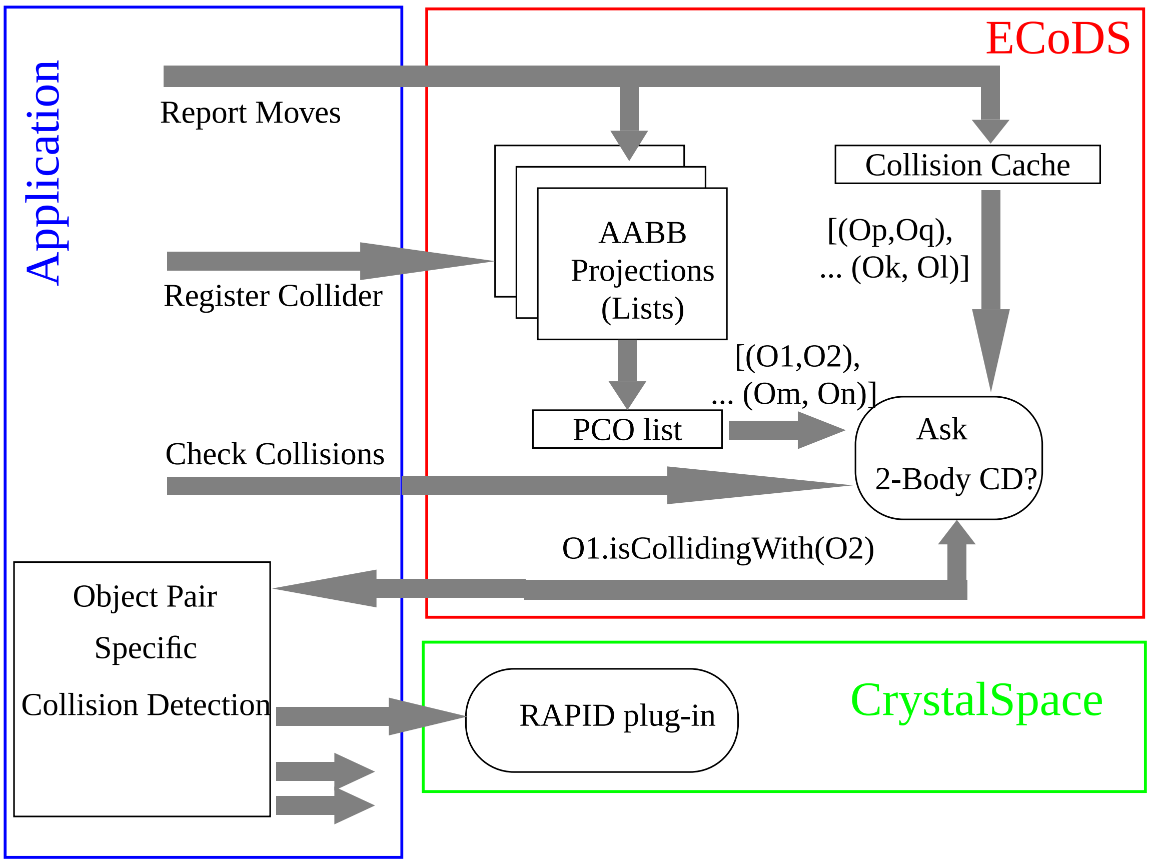


Figure 2: An overview of ECoDS and its place in a CrystalSpace application.

ever, during swapping values in the projection lists, two intervals (projections of two AABBs) change their overlap, the **PCO** array in each **xCollider** reflects these changes. Once the reordering is complete, we have to handle all the potential collisions. For pairs of immobile colliders we reuse the values from the **CollisionCache**. For pairs containing at least one object that moved, we let the implementation of **xObject.isCollidingWith()** to check and handle the possible collision. Its result is then stored in the CollisionCache. Note that the developer is able to implement different algorithms for the 2-body CD, depending on object shape, distance, properties, etc. This makes our approach very flexible. One limitation of the current implementation is that for projecting AABBs the objects needs to implement an **iMeshWrapper** interface, but this can be easily modified to require directly the AABB.

The colliders in ECoDS are implemented by the **xCollider** class. Whenever an object moves or otherwise changes its AABB, its corresponding xCollider must be informed via **setMovedFlag()** which updates the AABB projections. The **PCO** array contains information about the relation between the owner collider and all other colliders. It only states on which axis the AABBs overlap. This array is updated whenever a call to **xCoDS.updateCollisions()** occurs. All the information about the object (AABB, 2-body collision handling) is obtained via a pointer to an **xObject**. In principle the abstract class **xObject** is an interface that has to be implemented by all the objects involved in the ECoDS collision detection. All objects have to be able to return an **iMeshWrap-**

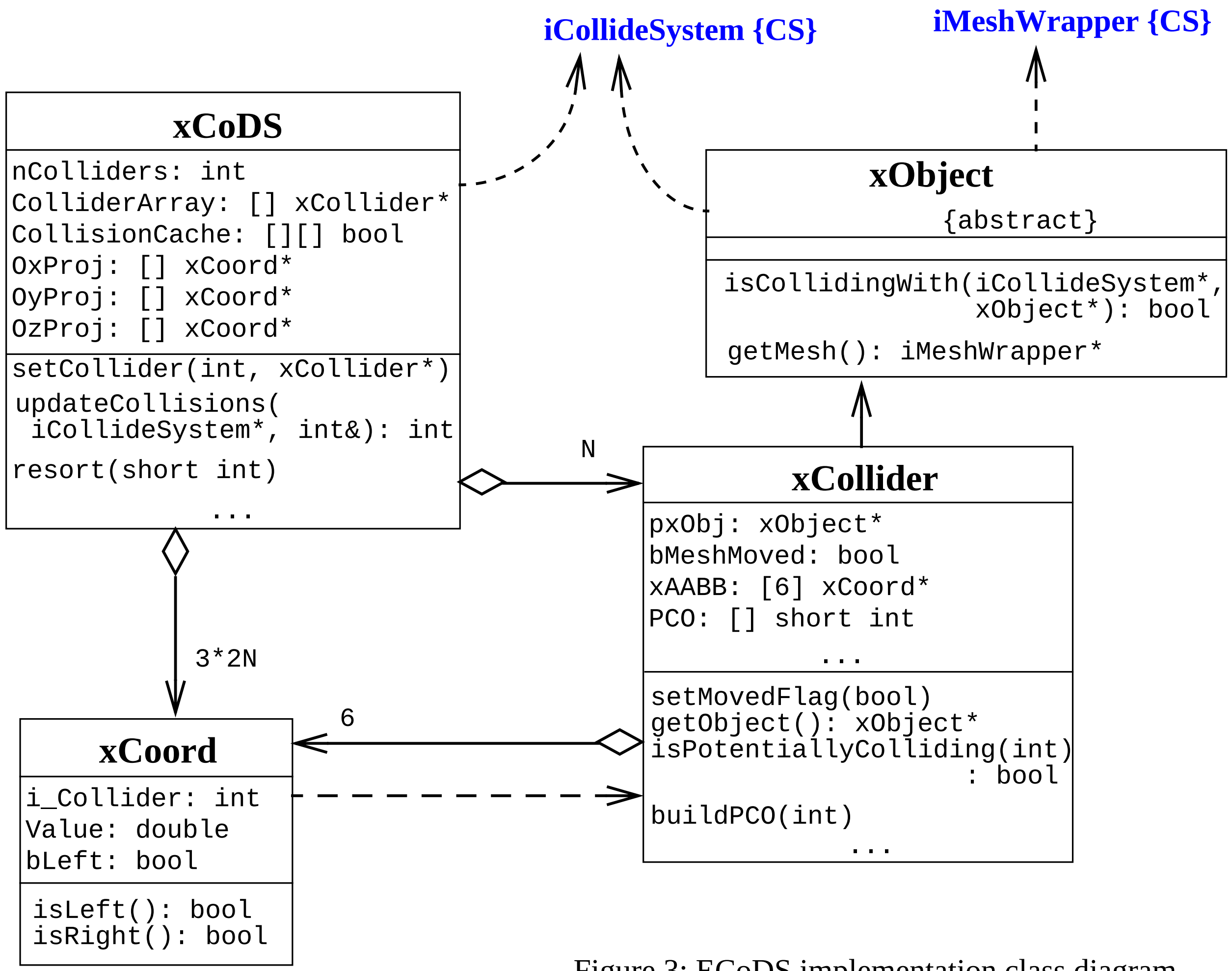


Figure 3: ECoDS implementation class diagram

per (from which ECoDS just uses the AABB). The objects also have to implement a handler for 2-body collision checking, **isCollidingWith()**. Note that this handler both detects and deals with the collision it necessary. So the handler should contain a call to a 2-body collision detection algorithm (“RAPID” for example, as we do in our demo described later). It should also modify/update whatever necessary attributes (speed, position,...) in case a collision occurred.

In brief, the developer has to take the following steps in order to use ECoDS:

- use **xObject** as a base class for all objects involved
- create an **xCollider** for each object and register it with **xCoDS**
- if the object changed its AABB (movement, scaling, shape change,...) inform the associated **xCollider**
- use the **updateCollisions()** from **xCoDS** whenever the collisions have to be handled

Note that ECoDS has no problem with objects that change shape (mesh).

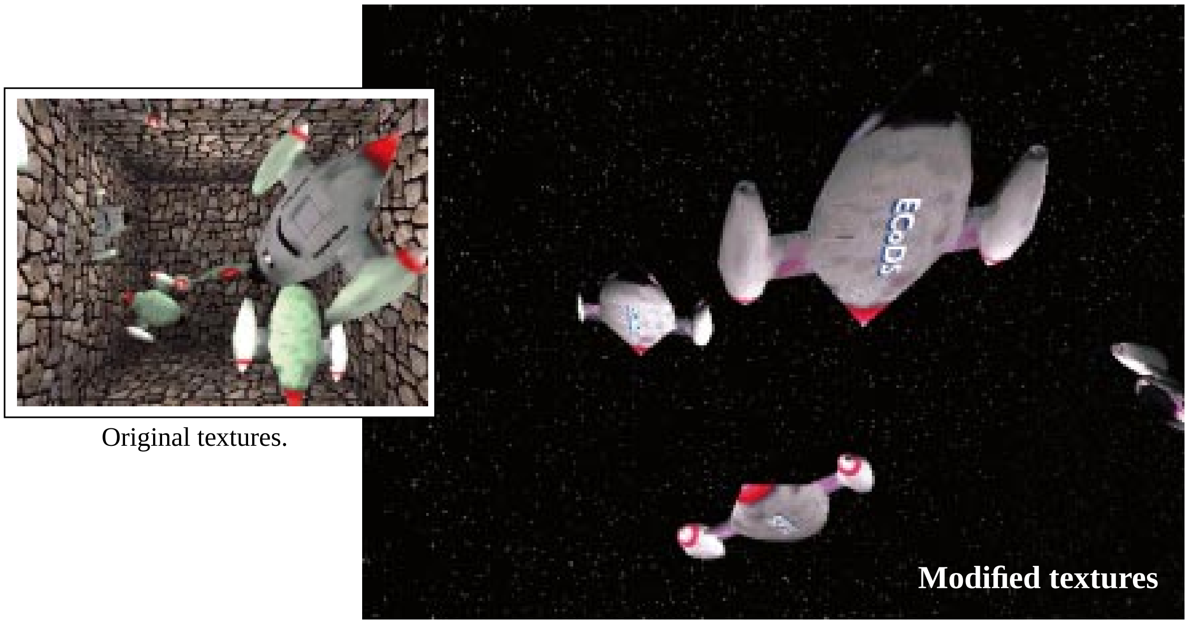


Figure 4: Demo Snapshots. Bouncing ships in a small room.

5. Experiments

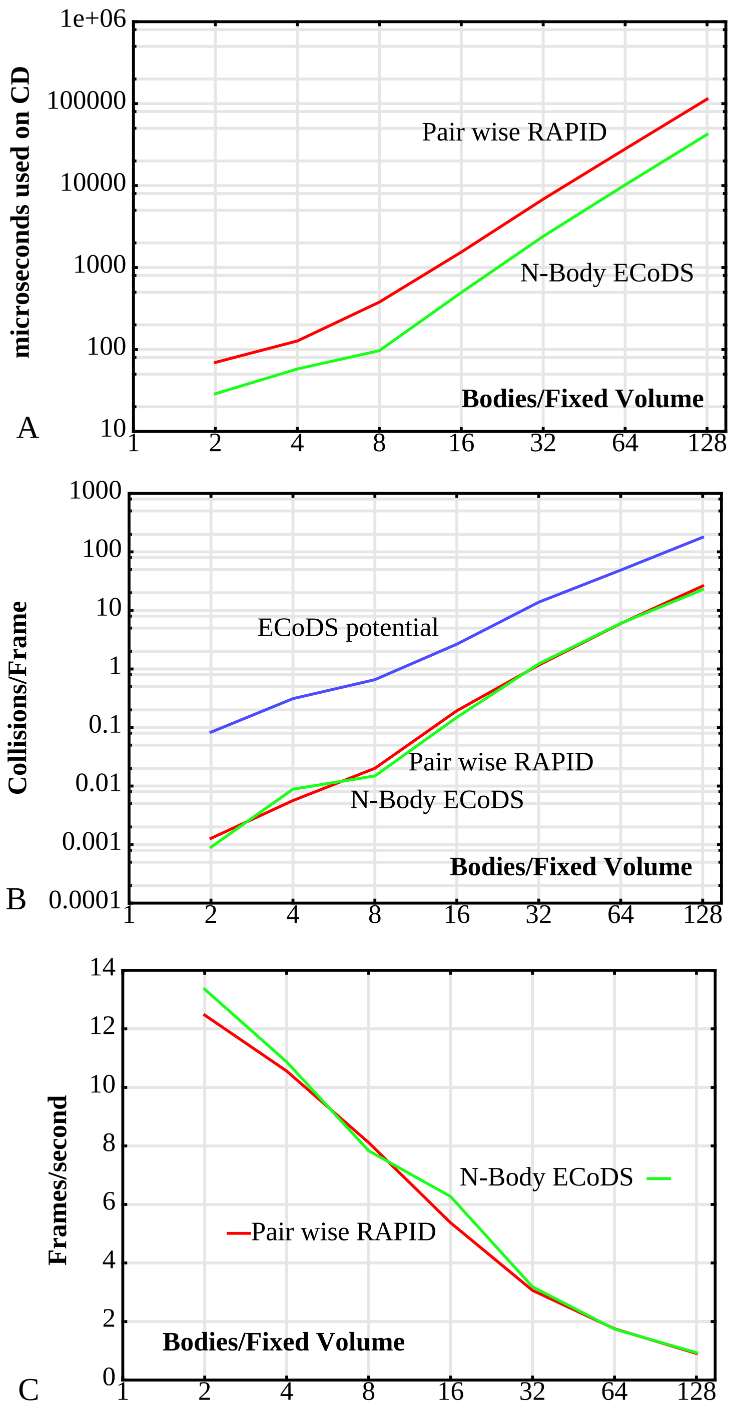
The experiments described here are based on a simple demo, consisting of a rectangular room containing a set of bouncing objects (Figure 4). As objects we used a spaceship sprite already available in the Crystal Space 0.92 distribution (from “data/demodata.zip” the mesh “objects/th_ship” and texture “textures/shiptex.jpg”). This sprite contains 300 vertices and 430 triangles. The objects in the room are randomly set initially to have certain position, speed, roll, yaw, and pitch. Whenever a collision occurs between objects, they reverse and randomly scale their speed, roll, yaw and pitch. This helps the viewer to notice the collisions. All experiments were run on a SunBlade100[1], 502.0 MHz UltraSPARC4, 256MB using SunOS 5.8 (Solaris 8).

To evaluate the performance of our ECoDS implementation, we compared it to a pure flat collision detection system using RAPID (see Figure 1.a), referred to as pair wise RAPID. We looked

1. http://www.sun.com/desktop/sunblade100/

at the actual time spent on collision detection, frame rate and the number of potential collisions and detected collisions. We wanted to compare the two methods in two different situations. First, we examined the case in which the volume remains constant and the number of objects increases (Figure 5 A, B, C).

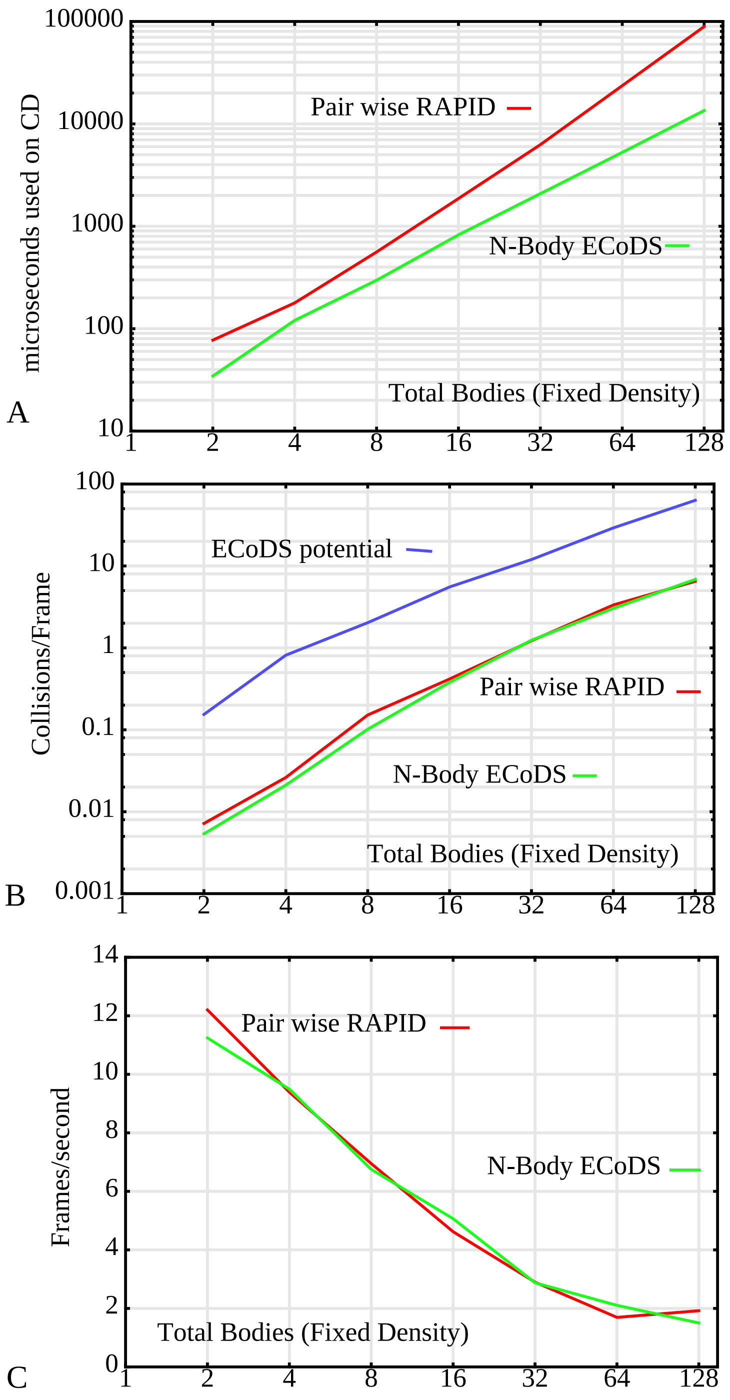
Figure 5: Case1: Fixed volume / variable body density



First, our system (N-Body ECoDS) is always faster than pair wise RAPID (Figure 5.A). Secondly, as expected the number of potential collisions detected by ECoDS increase and therefore the number of calls to RAPID increases abruptly towards the number of calls used in pair wise

RAPID (Figure 5.B). Finally, there is a slight variation in the number of total collisions per frame detected by the two methods because the number of frames per second varies (Figure 5.C). Since for one method the objects move a longer distance between frames, it can happen that there will be more collisions between two frames. But the variation, as resulting from the graph is rather small. This is mainly because the collision detection takes anyway a small percentage of the total computation for one frame. On a faster machine the difference would probably be more significant.

Figure 6: Case2: Fixed density / variable volume



The second case we examined, was when the density of bodies remains constant although the number of colliding bodies increases. We considered a density that gave one collision per frame in the previous case (32 bodies in the room) and adjust the room size depending on the number of bodies. The results are depicted in Figure 6 A, B, C.It is clear that our N-Body ECoDS system performs better and better than the pair wise RAPID when the number of colliders increase but the density is kept at a constant value (Figure 6.A). This is because the number of potential collisions detected by ECoDS increase slower (proportional with the number of objects added) than the number of calls to RAPID in the pair wise RAPID (proportional with the square of the number of objects added). This is clear from Figure 6.B, where whenever the number of objects doubles, the number of potential collisions grows by approx. 2.5 times. Note that the number of potential collisions keeps at a constant ten times the number of actual collisions (meaning the object density is indeed constant). Finally, as in the first case, there is a small variation in the number of collisions per frame resulted from the frames per second variation.

6. Conclusions

Collision detection is required in most dynamic 3D graphic applications. Moreover, when timing is an issue (games, real-time simulations, etc.) fast collision detection is a must. In the current report we presented ECoDS, an enhanced collision detection system that is a high level manager for n-body systems. ECoDS can use any 2-body collision detection algorithm (e.g. RAPID) to identify real collisions, but tries to avoid unnecessary calls by various methods. It uses AABB overlaps to detect and handle only those objects that may collide. It also uses a collision cache for stationary objects. Finally it employs frame coherency to perform faster detection.

The implementation presented here uses the Crystal Space graphic engine and its RAPID collision detection plug-in. Its modularity and small number of classes makes it easy to use and extend. The experimental results show impressive results compared to a flat, pair wise collision detection approach. For large number of sparse objects ECoDS gets much faster (8 times) than the flat pair wise approach. Therefore, whenever one needs collision detection in an n-body system, it makes sense to put a small effort into implementing a high level manager for collision detection.

7. References

1. Gamma Research Group at University of North Carolina
   Collision Page: http://www.cs.unc.edu/~geom/collide/index.shtml
2. CrystalSpace project page: http://crystal.sourceforge.net
3. T. Möller, E.Haines. *Real-Time Rendering,* A.K. Peters Ltd., 1999
4. S. Gottschalk, M. C. Lin and D. Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In *Proc. of ACM Siggraph'96.*