A Rigid Body Physics Engine for Interactive Applications

Marco Santos Souza
The Cyclops Group

Tiago de H. C. Nobrega
The Cyclops Group

André Ferreira Bem Silva
The Cyclops Group

Diego D. B. Carvalho
The Cyclops Group

Aldo von Wangenheim
The Cyclops Group

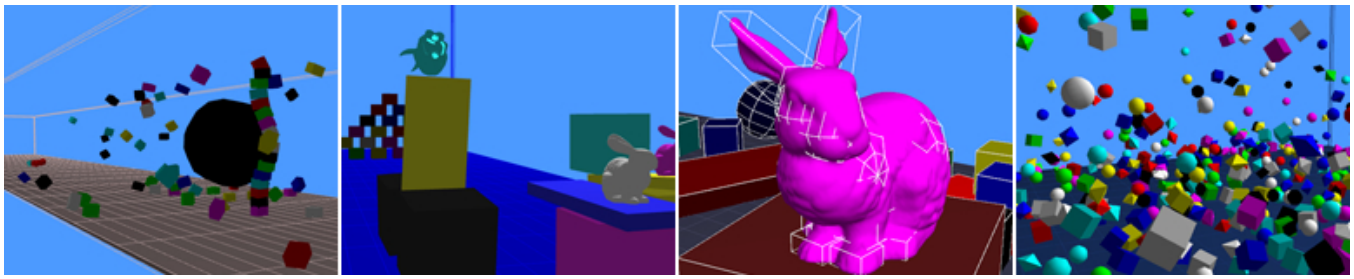

Figure 1: *Simulation of different scenarios using the presented engine*

Abstract

We have conceived and implemented a software library to be employed in speeding up the development of animations or applications that make use of interactive physics simulation. This paper focuses on the discussion of some of the algorithms and techniques that were used on its basic implementation, and also on expansions and optimizations that were applied to specific parts of the developed simulator aiming faster and more stable results. We conclude presenting some results along with a brief discussion of specific topics and possibilities for a future work. Our goal is to end up with a fast, stable and useful engine.

Keywords:: GJK, Rigid Body, Collision Detection, Collision Response

Author's Contact:

sms.cpp@gmail.com
{tigarmo, dedefbs, diegodbc, awangenh}@inf.ufsc.br

1 Introduction

Rigid body simulation has many applications such as in video games, movies and robotics. In games, for example, rigid bodies have become widely used to build interactive and more realistic environments. Therefore, there is a constant desire to have more and more objects being simulated faster and faster. Due to computational limitations, however, fast enough results are achieved only at the cost of a number of techniques and simplifications of mathematical and physical models. The use of these simplifications are justified by the fact that we are interested in *visually plausible* animations, i.e., our results must at least look correct. Our goal is to adjust the trade-off between accuracy and a visually appealing interactive simulation.

By "rigid body simulation" we mean the simulation of multiple rigid bodies, with possibly all in mutual contact [Erleben 2004; Millington 2007; Eberly 2010]. Basically, to resolve these multiple contacts it is necessary to calculate and apply forces to ensure that objects do not interpenetrate. In a realistic simulation, however, this can not be made in an arbitrary manner [Baraff 1992]. The implementation of a technique to accomplish this task properly is far from being trivial, specially when considering articulated rigid bodies (connected by *joints*) [Erleben 2004]. For this reason, most of the recent researches in rigid body simulation have been related to *collision response*.

The task preceding the collision response is called *collision detection*. Its purpose is to obtain the necessary information about a probable collision between two or more objects so that the collision response can be applied. Collision detection by itself is a considerably big topic—much more related to geometry than to physics—having numerous applications beyond physics simulation. It is also an active area of research.

Rigid body simulation is a highly-interdisciplinary field, and the necessary background to build a physics engine relates to a vast subject. Hence, in this paper, which has the purpose of presenting our software, we intend to provide a general overview of the engine operation and its basics components (section 3). We comment on some topics with further details (sections 4, 5 and 6), present some results (section 7) and then we make a brief discussion of topics whose exploration we believe to be of highest priority in future work (section 8).

2 Related Work

The most recent breakthroughs in rigid body simulation—results of which are visible in large commercial productions—are usually not published, since they are kept as "business secrets". Moreover, there are very few works that provide a "big picture" view of a practical implementation [Bourg 2001; Millington 2007]. Most of them are quite theoretical [Erleben 2004; Eberly 2010], or related exclusively to some specific part of the simulation process (e.g., *collision detection* [van den Bergen 2003; Ericson 2005], *collision response* [Catto 2005], or even just more specific algorithms such as *GJK* [van den Bergen 1999], *SAP* [Terdiman 2007], etc.).

Due to the way collisions are treated, our engine can be classified as an impulse-based simulator. Hahn [Hahn 1988] was the first to use a series of collision impulses (sometimes called *micro-collisions*) to prevent penetrations of resting objects. Based on Hahn's results, Mirtich [Mirtich and Canny 1995; Mirtich 1996] has proposed the impulse-based paradigm. Erleben [Erleben 2004] makes a good distinction between the different approaches and also presents a hybrid simulator. In an instructive way, Millington [Millington 2007] presents a rigid body simulator that produces not so stable results, but is still highly functional and suitable for many applications.

In relation to collision detection, the books of Bergen [van den Bergen 2003] and Ericson [Ericson 2005] are the most complete references that we have found. About the *GJK* algorithm, Bergen's paper [van den Bergen 1999] is quite valuable, as well the original paper where this algorithm was first suggested [Gilbert et al. 1988].

3 Engine Overview

Figure 2 shows a high-level view of the simulation pipeline. At this level of abstraction the engine operation seems quite simple. In the stage called *integration*, the movement of each object is simply integrated by a chosen time step. Then, the *collision detection* stage calculates the required information about the way objects possibly

touch or penetrate each other. As an impulse-based simulator, all interaction between different objects are modeled only through contact points and impulses are applied at these points in the *collision response* stage.

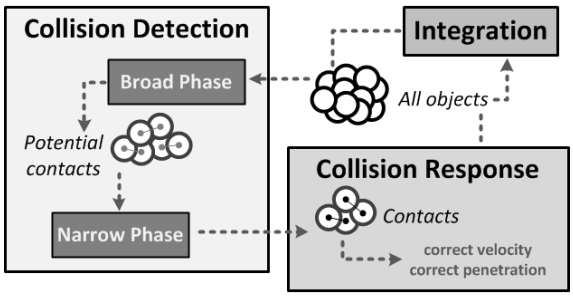

Figure 2: *High-level view of the simulation pipeline.*

3.1 Integration

Since all interactions between different objects are treated only at the collision response stage, the integration process is quite simple: each object is integrated in time without any concern about other objects around. In other words, collisions are totally ignored here and objects are allowed to be interpenetrating after this process.

We provide two options for integration: a simple Euler based method and a more expensive fourth order Runge-Kutta (also known as RK4) based method. The desired method can be chosen offline through configuration parameters. We could observe that the accuracy provided by Euler method is usually sufficient for games as well as for most applications. An in-depth explanation about integration methods can be found in Eberly's book [Eberly 2010].

3.2 Collision Detection

We have adopted the common approach of separating the collision detection process in an initial *broad phase* followed by a *narrow phase*.

3.2.1 Broad phase

In a scene with n objects, a brute-force approach for broad phase would result always in $\frac{n(n-1)}{2} \approx \mathcal{O}(n^2)$ tests. Fortunately, for most frames, we can do a lot better. The best option we have found was a technique first named "sort and sweep" by Baraff [Baraff 1992] and later popularized by Cohen [Cohen et al. 1995] with the name of "sweep and prune" (SAP). Since it can exploit frame coherence, SAP has proved being very suitable for rigid body simulation and similar applications. For comparison purposes, we have also implemented a spatial partition technique using an uniform grid. A known problem with spatial partition approaches is the overhead caused by the necessity to maintain complex data structures updated in very dynamic scenes.

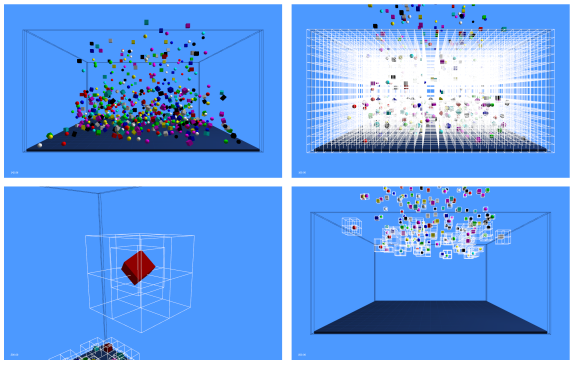

Figure 3: *Screenshots of the application used to perform comparison tests of broad phase techniques.*

All of our tests for the broad phase were performed on an application that consists of a scene with 1024 objects (boxes and spheres) initially suspended in the air. Subject only to gravity, objects may collide with each other and also with the floor and walls of a box-shaped virtual scenario, as shown in the upper left image of figure 3. Also in this figure, the upper right image shows the structure used by the uniform grid approach to subdivide space by many cells. In the bottom images only the cells that contain some object are shown.

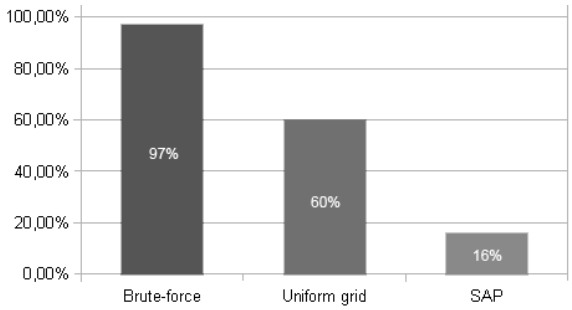

Figure 4: *For each approach, percentage of profiler samples taken only in broad phase routines.*

To compare the performance of each broad phase approach we performed a sampling test provided by a profiling tool of our integrated development environment (*Microsoft Visual Studio 2010*). The results are illustrated by figure 4. When using brute-force, 97 percent of samples were taken in routines related to broad phase. When using SAP, however, this number drops to only 16 percent.

3.2.2 Narrow phase

In order to obtain contact data, we have implemented some specific routines which are correctly called according to the type of the geometry involved. This is easily achieved through a double dispatch technique. We have also provided a more generic solution (section 4), which works for any two convex shapes.

3.3 Collision Response

The greatest difficulty in the collision response stage is to produce an appropriate behavior when, in a certain frame, there are multiple contacts involving the same objects. Unfortunately, such a situation is quite common in most practical scenarios; therefore, a proper technique to handle this case is of crucial importance. The approach that was implemented in our engine is an iterative technique that is simple to understand and implement, being explained in detail by Millington [Millington 2007]. Each contact is separately resolved and the effects of each resolution have to be spread along all other contacts in the same *group* or *island* (see section 6). At each iteration, the most severe contact needs to be found and resolved. An appropriate result is achieved usually at the cost of a few iterations. Although not being suitable for handle high stacks of objects, this approach is quite adequate to simulate very dynamic scenarios, like explosions, for example. It is interesting to note that by using the technique explained in section 5, our engine can produce considerably more stable results than Millington's simulator.

4 Collision Detection with GJK

The Gilbert-Johnson-Keerthi distance algorithm, also known as GJK, is an extremely versatile method since it does not need to know much about the input objects. The only requirement is that the objects can return a most extreme point for a given direction, which is called a "support point". The original purpose of this algorithm is to calculate the shortest distance between two convex objects. However, it can be easily adapted to perform intersection tests and also can be extended to calculate all information necessary at narrow phase collision detection. Another feature that makes GJK interesting is that, like SAP (subsection 3.2.1), it can exploit frame coherence, providing extremely fast results. A good introductory

explanation of GJK algorithm is given by Souza [Souza 2011], and a more in-depth discussion can be found in Bergen's works [van den Bergen 1999; van den Bergen 2003].

To calculate the penetration depth in our engine, another algorithm, called "Expanding Polytope Algorithm" (EPA), is used in conjunction with GJK, as proposed by Bergen. The input for EPA is a simplex that contains the origin, which is exactly the output of GJK when objects are intersecting. Therefore, GJK and EPA are complementary algorithms. Unfortunately, EPA is very susceptible to numerical errors when dealing with small penetrations. Hence, Bergen suggests a hybrid technique that combines the use of GJK and EPA in a clever manner, where EPA is used only for large penetrations. Due to the way such technique is implemented, it is also possible to easily calculate the contact normal vector and the contact point. Additionally, frame coherence can still be exploited by using the separation vector found in one frame as a starting vector for the next frame.

The GJK algorithm by itself is quite simple to understand and implement. However, there is an internal problem that is not so trivial to address properly, and has been receiving some attention in recent works [Ericson 2005]. The problem is as following: given a simplex represented by a finite set of vertices W, we have to calculate the point v such that v is the closest point to the origin in the region of simplex W. Thus, if simplex W contains the origin, v will be the origin itself, otherwise v will be on the surface of simplex W. In addition, we have to determine the smallest $X \subseteq W$ such that v can be described as a convex combination of X.

In the original paper of GJK [Gilbert et al. 1988] as well in Bergen's works [van den Bergen 1999; van den Bergen 2003], the described problem is handled by a routine called Johnson's distance subalgorithm. It is a purely analytical approach that is not so simple to understand and implement properly. Moreover, in practice, due to the finiteness of floating point representation, the technique is very susceptible to numerical problems. Fortunately, today there is a more adequate approach, which addresses the problem geometrically, being easier to implement and not running in the same numerical issues. A good explanation is provided by Ericson [Ericson 2005]. In our engine, we have provided both approaches, and the desired one can be chosen offline.

There is an interesting and very simple optimization of this geometrical approach, which apparently was not yet explicitly explored by any other published work. This optimization was informally suggested by Casey Muratori in a video posted by himself on the Web in 2006[1]. The geometrical approach needs to test the features of the current simplex to find which one has the Voronoi region that contains the origin. Then, the suggested optimization says that, due to the incremental nature of current simplex set, it is not necessary to test all features, but only the ones related to the last vertex added to the set. This way, many tests can be avoided.

5 Contact Region Approximation

Millington [Millington 2007] uses only one contact point to approximate the region of intersection between colliding objects regardless of the geometries involved. This is the most practical solution since collision detection works faster and is easier to implement. Moreover, collision response also runs quite fast. For many cases, however, a single point represents a too poor approximation, resulting in very unstable simulations.

Figure 5 shows a good approximation for the contact region between two colliding boxes, which, in this particular example, is composed by four points. The set of points is composed by the vertices of a polygon that fits properly inside the intersection area. To generate this point set, there are roughly two different ways: at once or incrementally.

In its version 2.78, *Bullet* engine has implemented the "at once" approach using a polyhedral clipping algorithm. This technique has the advantage of being precise, however it has the drawback of requiring more processing time. In our engine we have used the incrementally approach, which has the advantage that we can keep simple our collision algorithms: per frame, they have to find out only one contact point. For most applications, the instability caused by this approach is completely negligible.

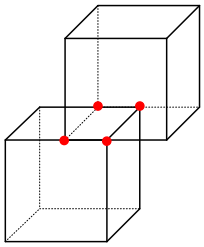

Figure 5: *Four contact points to approximate the contact region between two boxes.*

The idea is to maintain the points calculated in one frame and try to gather new ones in the next frames. Points are saved in a contact cache, and this cache must be updated during the simulation. An algorithm to accomplish this task was informally described by Erwin Coumans more than once in the Bullet discussion forum on the Web[2]. A similar approach is discussed by Mirtich [Mirtich 1998] under the name of "contact tracking".

6 Contacts Groups

In subsection 3.3 we said that the greatest difficult at the collision response stage is to cope with the interdependence between multiple contacts in a certain frame. It is interesting to note that the scope of such interdependence is limited. Some objects break the contact dependency chain because they are completely ignored at the collision response stage: the floor, the walls, or any other object that can be classified as a "static object". They must not have their speeds and positions corrected. Thus, for a certain frame, the contacts found during collision detection can be grouped into independent sets. These sets or groups of contacts are also called "contact islands" in some simulators (e.g., in the *Open Dynamics Engine*). The task of calculating such groups is a typical graphs problem. The algorithms that we have used to accomplish this task are better explained in Souza's work [Souza 2011].

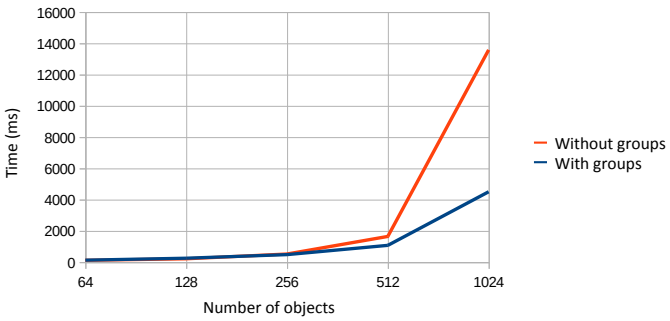

Figure 6: *The total amount of time required to execute the test simulation, with different amounts of objects.*

In order to measure the potential gain of performance achieved by using this technique, we have performed some tests over the same application described in subsection 3.2.1. The results are illustrated by figure 6. A significant performance improvement can be observed in the cases with 512 or more objects when using contacts groups. Although there is no noticeable gain in the cases with 256 objects or less, we note that there was no loss due to the possible overhead caused by the additional task of generating the groups.

[1] https://mollyrocket.com/849

[2] http://www.bulletphysics.org/Bullet/phpBB3/viewtopic.php?p=&f=&t=226

7 Implementation and Results

We have used the C++ language for all implementations. The engine was separated in three distinct modules. The "*Core*" module implements mathematical functions and other routines that are useful to the other modules. It does not have any external dependency, except for the C++ standard template library. The "*ColD*" module is responsible for all tasks related to collision detection, and it depends only on the "*Core*". The "*Dym*" module implements the integration and the collision response stages (see figure 2). In attempts to abstract the engine's internal workings, it also provides classes with a friendly interface which perform higher-level operations. The "*Dym*" module depends both on "*Core*" and "*ColD*".

Scene	# of frames	Min FPS	Max FPS	Avg FPS
Fig. 7 (A)	14867	938	1522	991
Fig. 7 (B)	7521	345	645	501
Fig. 7 (C)	1574	101	107	104
Fig. 7 (D)	2552	141	188	170

Table 1: *Number of rendered frames, minimum, maximum and average FPS for each scene after 15 seconds of simulation.*

We have developed an application to test out our engine using different scenarios. Some screenshots with a brief description are shown in this section. The 3D model that appears in some images is the popular Stanford Bunny, which is composed by 69,451 triangles. Collision detection is performed over a much simpler geometry, though (see the third picture in figure 1). The applications use the OpenGL API to draw the scenes. The test machine is a PC with an Intel Core 2 Duo E6600 (2.40 GHz), 2 GB of RAM memory and a NVIDIA GeForce 9600 GT with 512 MB of DDR3 dedicated memory. The operating system is Microsoft Windows 7 Professional (32 bits). Table 1 presents a benchmark result for the scenes illustrated by the screenshots in this section. The total number of rendered frames, the minimum, the maximum and the average FPS (frames per second) that were achieved for each scene after 15 seconds of simulation are shown.

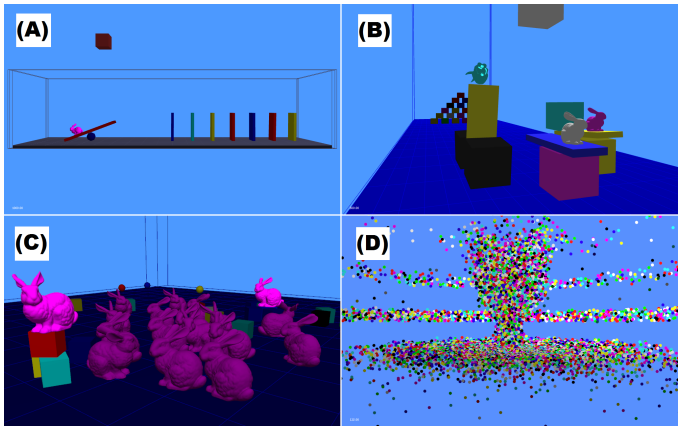

Figure 7: *(A) A Stanford bunny is thrown into the air with a seesaw, hitting a row of dominoes. (B) Three Stanford bunnies are catapulted into a wall of blocks, knocking it down. (C) Plenty of Stanford bunnies interacting with each other and with other objects. (D) 20,000 particles released into the air (no collision detection and response, only integration).*

8 Conclusions and Future Works

We can say that we have reached our goal of producing a fast, stable and useful engine. Different scenarios can be quickly generated and the developer is free to create and configure a new simulation according to his needs. However, there are some points that deserve attention.

The inability of properly simulating stacks with many objects is a meaningful limitation. Another issue is that we have not covered the simulation of objects attached to each other by *joints*. A common approach is to handle contacts and joints in an unified way, where a joint is seen as a permanent movement constraint and a contact as a temporary one. Stacks and joints are two topics that have caused a lot of headache in a recent past, but which nowadays most of modern simulators are able to handle. In this way, the improvement and expansion of our collision response method is seen as a very interesting first future work.

Another promising topic to be explored is GPU parallelization. Internal algorithms or even whole sections of our engine could be redesigned in order to perform tasks in parallel. With the advent of GPGPU languages—such as NVIDIA's CUDA or OpenCL—and their growing support and popularization, optimizations of this kind have became massively explored by recent works in physics simulation.

References

BARAFF, D. 1992. *Dynamic simulation of nonpenetrating rigid bodies*. PhD thesis, Ithaca, NY, USA. UMI Order No. GAX92-36100.

BOURG, D. 2001. *Physics for Game Developers*, 1st ed. O'Reilly Media, Inc., November.

CATTO, E. 2005. Iterative dynamics with temporal coherence.

COHEN, J. D., LIN, M. C., MANOCHA, D., AND PONAMGI, M. K. 1995. I-collide: An interactive and exact collision detection system for large-scale environments. In *Symposium on Interactive 3D Graphics*, 189–196, 218.

EBERLY, D. H. 2010. *Game Physics, Second Edition*, book & cd-rom 2st ed. Morgan Kaufmann.

ERICSON, C. 2005. *Real-Time Collision Detection*. Morgan Kaufmann, January.

ERLEBEN, K. 2004. *Stable, Robust, and Versatile Multibody Dynamics Animation*. PhD thesis, University of Copenhagen, Denmark.

GILBERT, E. G., JOHNSON, D. W., AND KEERTHI, S. S. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *Robotics and Automation, IEEE Journal of 4*, 2, 193–203.

HAHN, J. K. 1988. Realistic animation of rigid bodies. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 299–308.

MILLINGTON, I. 2007. *Game Physics Engine Development*, book & cd-rom 1st ed. Morgan Kaufmann.

MIRTICH, B., AND CANNY, J. F. 1995. Impulse-based simulation of rigid bodies. In *Symposium on Interactive 3D Graphics*, 181–188, 217.

MIRTICH, B. V. 1996. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California at Berkeley.

MIRTICH, B. 1998. Rigid body contact: Collision detection to force computation. Tech. rep., IEEE International Conference on Robotics and Automation.

SOUZA, M. S. 2011. *Animação Baseada em Física: Desenvolvimento de um Simulador de Corpos Rígidos para Aplicações Interativas*.

TERDIMAN, P. 2007. Sweep-and-prune.

VAN DEN BERGEN, G. 1999. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools 4*, 2, 7–25.

VAN DEN BERGEN, G. 2003. *Collision Detection in Interactive 3D Environments (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann, October.