Parallelizing Broad Phase Collision Detection for Animation in Games: A Performance Comparison of CPU and GPU Algorithms

Ygor R. Serpa[1] Maria Andréia F. Rodrigues[2]

[1]Universidade de Fortaleza (UNIFOR)
Centro de Ciências Tecnológicas (CCT)
Av. Washington Soares 1321, Bloco J
60811-905 Fortaleza-CE Brasil

[2]Universidade de Fortaleza (UNIFOR)
Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Av. Washington Soares 1321, J(30)
60811-905 Fortaleza-CE Brasil

Abstract

Games, computer animations and three-dimensional interactive simulations have required the use of realistic and faster than ever before broad phase collision detection algorithms. In this work, we compare the performance of four broad phase algorithms implemented on CPU and GPU, using four different test scenarios. More specifically, one of them is a new GPU-based algorithm that we have developed in the *Bullet* library using CUDA, and the other three remaining implementations are CPU-based algorithms available in the same library. The experimental results show that the heterogeneous algorithm is competitive when compared to some robust methods available in *Bullet*, particularly in scenes with a large number of objects whose movements are complex and unpredictable. We believe that initiatives like this, which explore solutions for new implementations of collision algorithms running on GPU and operating asynchronously with the CPU, are extremely important and useful for game *designers*, especially in the area of digital games based on Physics, considering there are other elements of the animation, *e.g.*, sound and artificial intelligence, which can thus be executed during the broad phase calculation.

Keywords: performance analysis, broad phase collision detection, CPU, GPU, *Bullet*

Authors' contact:
`{ygor.reboucas,andreia.formico}@gmail.com`

1. Introduction

Increasingly realistic interactive graphics applications, either visually or physically, have been developed. Among the factors that have enabled this increasing level of realism are the algorithms for 3D collision detection [Coutinho 2001]. Basically, collision detection systems ensure that animations generated at runtime are consistent with the major laws of Physics, by generating plausible responses to contacts and avoiding object interpenetration.

In digital games, particularly those based on Physics, the collision detection algorithms are essential to the gameplay. Some examples of popular games in this category are *AngryBirds* [Rovio 2009], *World of Goo* [2D Boy 2008], *Crayon Physics* [Kloonigames 2009], *Armadillo Run* [Stock 2006], etc. Generally, the collision detection system is triggered after the execution of the dynamic system, which is responsible for moving objects based on the values of their velocities. The collision detection is then performed in two steps: (1) finding the intersections between objects in the scene; and (2) responding to the detected collisions [Moore and Wilhelms 1988]. In the first stage, by applying a discrete method, the objects can be evaluated from time to time to find any collisions, or using a continuous method, through parameterization techniques of their positions over time. In the second, physical models are used to simulate the action-reaction forces that occur between colliding bodies.

Usually, different levels of detail of 3D environments are used in such a way as to ensure the operations of collision detection occur as planned. For example, medical applications or 3D realistic simulations require more precise information about the collisions, particularly on the geometry of the intersection and applied forces. On the other hand, 3D real-time gaming and interactive graphics applications require that the relevant information about the collisions are calculated and extracted quickly, which often simplifies the whole process because it considers a minimum number of contact points between objects, instead of using the full geometry of the intersection [Bergen 2004].

One of the most frequent ways to implement a system of collision is dividing the process into two or even three phases [Hubbard 1995] and [Mirtich 1997]. The first one is called *broad phase* and aims to analyze all objects in the scene, and then select which pairs are possibly colliding (the others are discarded). This phase uses simplification techniques to improve the overall performance of the application that is essential to speed up the process as a whole, especially when there are a large number of objects in the scene [Rocha *et al.* 2006]. The second phase, known as *narrow phase*, aims to test pairs of objects found in the previous phase, by applying more robust and accurate methods. The third (which is optional), is the *exact phase*, in which the calculations are performed at the level of vertices, in order to obtain greater accuracy.

There are many algorithms devoted to each of these phases. Many of them are recommended only for specific scenes, hindering fair comparisons of performance between them. Thus, it is crucial to choose efficient and specific algorithms for scenarios where they will be executed. This type of problem is accentuated in multimedia applications such as digital games and interactive simulations, because more than one execution scenario may exist, besides the management of CPU threads and memory resources.

At the same time, only recently there was a high diffusion of graphics cards in personal computers (although these resources exist for several years), as well as in the number of real-world applications implemented using computing platforms on GPU, such as Nvidia CUDA [Nvidia 2014a]. Modern GPUs are extremelly efficient at manipulating computer graphics and their highly parallel structure makes them more effective than general-purpose CPUs.

Currently, there are three physics libraries that stand out due to their complete system of collision detection: *Bullet* [Coumans 2014a], *PhysX* [NVIDIA 2014b], and *Havok* [Havok 2014]. Definitely, these highly optimized collision detection libraries allow for more realistic virtual worlds in games. *Bullet* is open source (whereas *Havok* and *PhysX* are not) and fully implemented at the CPU level. Only *PhysX* has an implementation at the GPU level, however, limited to the video cards from Nvidia.

In this work, using four different test scenarios, we present a comparative performance analysis of four broad phase collision detection algorithms. One of them is a new GPU-based algorithm that we have developed in the *Bullet* library using CUDA, and the other three remaining implementations are CPU-based algorithms available in the same library. We believe that initiatives like this are extremely important and useful for game *designers*, especially in the area of digital games based on Physics.

2. Related Work

Currently, there are few works on broad phase collision detection using GPU devices [Grand 2007; Liu, *et al.* 2010; Lo, *et al*. 2013]. Additionally, even fewer studies simultaneously address this issue and use the *Bullet* physics library. Many of them, which combine parallelism with broad phase collision detection algorithms, basically focus on adapting some existing methods to multi-core architectures [Avril 2010].

For example, the work of Grand [Grand 2007] pioneered the use of GPUs for processing broad phase collision detection. The author presents a GPU-based implementation of a regular grid to exploit the spatial coherence of the scene [Bergen 2004], which states that objects positioned far apart do not intersect. The whole process of construction and use of the grid is done at the level of the graphics card.

A hybrid algorithm using *Sweep and Prune* & regular grids is described in [Liu *et al.* 2010]. It is implemented entirely on the GPU. Although the authors report that their algorithm achieves excellent results, such as simulating up to one million of objects in a scene, there is a lack of using a direct equivalent method implemented on CPU, for conducting a fair performance comparison.

Other authors have been working on adapting existing methods to multi-core architectures [Avril 2010], which are currently available in the modern processors and have a robust model of parallelization. However, few threads are actually executed in parallel. Algorithms developed for these architectures can be easier adapted to many-core architectures than algorithms designed to single-threaded systems. Besides, this approach is undoubtedly important to the state-of-the-art in general, but is not ideal for games and other media intensive applications that have highly complex threading models to handle a multithreaded game-loop structure.

Recently, Bullet's developers announced that the next version of the library will contain a fully GPU-based pipeline for collision detection and physics [Coumans 2014b]. However, so far the library has made available only one experimental implementation of a regular grid, very similar to the approach proposed by Grand [Grand, 2007].

3. Bullet Broad Phase Collision Detection

The *Bullet* physics library contains an interface for defining broad phase collision algorithms that enables integration of new algorithms together with other modules that comprise the collision detection pipeline. This interface receives from the detection pipeline all the colliding objects in the form of Axis-Aligned Bounding Boxes (AABBs), and returns to the pipeline the broad phase results, as a structure called of colliding pairs cache.

Two considerations are relevant on the input and output data of the broad phase algorithms in the *Bullet* library: (1) every frame, all the objects have their new AABBs calculated and passed to the algorithm in order to update them; and (2) the colliding pairs cache is a persistent structure, *i.e*., the objects that remained in collision during multiple frames should remain in the same position in the cache. Therefore, we cannot empty the cache every frame in order to facilitate the management of colliding pairs of objects. More specifically, there are three algorithms that implement this interface in the *Bullet* library: *Brute Force* (BF), *Sweep & Prune* (SAP) and *Bounding Volume Hierarchy* (BVH). All those algorithms run on CPU and are single-threaded.

3.1 Brute Force (BF)

In the BF algorithm, every object in the scene is tested against every other object. For each test, if the objects are in collision, it is checked if this colliding pair is already in the cache. If it is not, it will be added; otherwise, it will be removed. Although it is generally slower than the other algorithms that exist in the *Bullet*, for scenes containing few objects it can be more efficient, since it has no updating overhead of its data structures. However, it is too inefficient to be used when the number of objects is large. On the other hand, its performance depends solely on the number of objects, thus being independent of the movements and physical characteristics of the objects. This is due to the fact that it always performs the same number of operations, regardless of these factors. However, it also presents some limitations, *e.g.,* if the pair of objects is (or is not) in the colliding pairs cache, the algorithm does not interfere in the process, being necessary to consult the structure for obtaining such information.

3.2 *Sweep & Prune* (SAP)

The traditional SAP algorithm reduces the 3D dimension of calculating the broad phase to three 1D related problems. Basically, the strategy consists of designing all objects in the scene in the coordinate axes x, y and z and inserting these projections into three lists, one for each axis. These lists are sorted according to the minimum point of the projection. Finally, each list is traversed to find intersections between the projections of objects on each axis. Pairs that intersect in the three lists are then sent to the next stage of the collision process, *i.e*., the narrow phase.

The big difference between this method and the BF is that there is no need to traverse the entire list for each projection. Since this list is ordered, if the minimum point of a projection is higher than the maximum one of the current projection, this means that no other projection from this one will intercept the current projection.

In particular, in the *Bullet*, a variation of the traditional SAP algorithm is implemented. More specifically, the ordered lists are maintained between frames of the simulation. During the sorting process, each object whose AABB was updated has its projections updated, moving the projection through the list, until its new position is found. When a projection is moved, it fails to intercept some projections but intercepts others. However, it is possible to infer, using the direction of the projection motion if it is ceasing to intercept (or intercepting) an object, in such a way as to make the necessary changes in the cache.

At the end of the sorting step, the cache will contain the new colliding pairs (all colliding pairs which are not anymore in collision have already been removed). That is, all the processing is performed during the updating of the objects and not in the calculation method of the possibly colliding pairs.

3.3 Bouding Volume Hierarchy (BVH)

The BVH algorithm performs spatial partitioning of objects using bounding volumes. At each level of the hierarchy, two bounding volumes are used to separate the objects in a left and right group. In the last level, there will be only one object per bounding.

The insertion of objects is done seeking the nearest leaf node of the inserted object and turning it into an internal node with a bounding volume enough to contain the old and new objects. Then, if necessary, the volume of the nodes located above this one can be changed to contain the new volume. The removal is done analogously, by removing the node and its parent node, leaving only its brother node. The volume of the nodes located above is reduced, if necessary.

The update of the objects is done very simply: if an object moves beyond a certain threshold, it is removed and re-inserted into the structure. Collision detection is performed during the updating of objects. If an object has been updated, its volume is tested against the tree to find all leaf nodes that intersect it. Finally, the pair of objects is added to the cache. Prior to the collision detection, a scan is made through all colliding pairs currently in the cache to verify whether they no longer collide. If they do not, these pairs are removed.

4. GPU Brute Force (GPU-BF)

The algorithm we have proposed and implemented for broad phase collision detection in the *Bullet* library is described in this section. Named as GPU-BF, this algorithm is a variant of the BF algorithm presented in Section 3.1. Just as the three algorithms previously presented in Section 3, the GPU-BF also implements the interface provided by the *Bullet* library. Despite its relative simplicity, it explores the huge processing power of modern graphics cards (GPUs) to perform broad phase collision detection among a massive number of objects.

Traditionally, GPUs have been used only for graphics processing, remaining idle during other processing steps of the simulation. Therefore, it is important to explore approaches that enjoy this latent computational resource. The relevance of the GPU-BF algorithm is in verifying the feasibility of performing the broad phase processing on GPU, with the premise that if a simpler algorithm is feasible for collision detection, more robust methods probably will be too.

However, GPUs have a different execution model that is different from the traditional CPUs. For example, these graphics cards work in a massively parallel manner by threading models that are simpler to manage than the model used in processors. Thus, they are able to launch hundreds of processing lines simultaneously, but all these lines should run the same instruction sequence, changing only the data that is provided for each processing line. This high degree of parallelism at the hardware level is obtained through the methodology SIMT (Single Instruction, Multiple-Thread) [Nvidia 2014c], where the same instruction is executed on different threads. The graphics card is then composed of several arithmetic logic units (ALUs) which are capable of executing an instruction in parallel on multiple entries.

One of the most important and fragile points is when there are instructions for flow control, in which a portion of the threads should follow through a stream of instructions and partly by another. When this occurs, the flow is serialized. Initially, the threads that run along a path are executed until the point at which the streams are merged, then the other thread group is executed, and finally, the execution continues normally. Another important point of weakness is the memory access pattern. The main memory of the device is unique to all threads, so memory readings must also be serialized. However, if all threads of a ALU order sequential data in memory, these readings are coalesced into a single operation.

Another way to minimize this problem is to use another type of memory present in the device, the shared memory. This memory is optimized for individual accesses at the cost of low capacity, staging and scope. The accesses are optimized using two techniques: (1) the shared memory is located on the same chip as the ALUs; and (2) this memory is partitioned between groups of threads, so the serialization of accesses is partially parallelized. On the other hand, it offers low capacity. Due to the limited physical space, data have low staging (whenever a group of threads has finished running, the data stored in this memory will be lost) and, finally, the scope of the data is limited to the group of threads that manipulate them. Apart from these two types of memory, there are two other regions dedicated to textures and constant data.
The development of algorithms that efficiently utilize the memory resources of graphics cards is complex due to the non-triviality in exploring these diverse memory areas. For example, to use shared memory, initially the data that are in global memory must be loaded and then transferred to the shared memory. If they are not contiguous in memory, the transfer will not be coalesced and, therefore, will spend part of the cycles saved during the execution on shared memory.

Therefore, to implement a GPU algorithm we have to adapt it from the SISD (Single Instruction, Single Data) methodology to the SIMT methodology used by the graphics cards, as well as to structure it to minimize flow control structures and random accesses to memory. A final consideration that we should emphasize in relation to algorithms on the GPU is the cost of data transfer, from the computer's main memory to the graphics card. This cost plus the running time of the algorithm results in the total time spent for its execution, making impossible to execute transactions that involve little processing on graphics cards. One example of this is sound processing, which has high input/output bandwidth requirements to process multiple input/output audio channels.

Our algorithm, illustrates how these requirements can be met. Since no test between objects is made (only the intersection test), we identified a low number of flow control structures. The tests of intersection between AABBs, in turn, can be entirely made in the bitwise form, further reducing the number of flow controls. Since all objects are tested against all others, many threads read the AABB of the same object and the AABBs whose objects are located side by side in memory, which shows an efficient use of global memory and no need to use the shared one.

The execution of the GPU-BF algorithm is performed as follows: each AABB is located in a list of locations, with n being the number of AABBs. Each thread has an index i and should test an object A against B. The object A is the element i / n and B is the element $i \% n$ (rest) of the list. It is important to note that not all combinations of elements must be tested. If the indices of A and B are equal, the same object is being tested. On the other hand, when the content of B is less than A, this pair will have been tested (in our implementation, we test the elements of A against the elements of B, and not B against A). For each pair of objects tested, a boolean result is generated indicating whether there was an intersection between the objects. These results are stored in a pre-allocated array and then sent to the CPU.

More specifically, for n objects, there are $n(n-1) / 2$ pairs to be tested. Each AABB is represented by a minimum and a maximum point, having in total six variables of type float, or 24 bytes. Each test consists of one byte, since the manipulation of bits on GPU is hampered by the parallelism. Consequently, for each frame generated we have $\{24n + n(n-1) / 2\}$ bytes transferred between CPU and GPU. Let's consider the indices A and B of two objects. If we know that A is different and smaller than B, we can find the index i of the byte that contains the test result of this pair of objects by calculating $\{i = A \bullet n + (B-1) - A(A-1) / 2\}$.

Thus, this equation returns the index i of the pair A and B, in the linear arrangement of the results.

From the point of view of analyzing the time complexity of the algorithm, where p is the number of pairs and k is the number of threads, the time complexity of the GPU-BF algorithm is $O(p/k)$. For a number of pairs smaller than the maximum threads in parallel on the graphics card, k is equal to p, so the time complexity is constant or $O(1)$. However, for a number of pairs bigger than the maximum of concurrent threads of the graphics card, k becomes constant and therefore no longer appears in the calculation, with the time complexity being $O(p)$. Note that we can represent p as a function of the number of n objects, as follows: $p = n\ (n\text{-}1)\ /\ 2$. Finally, for n tending to infinity, the time complexity of the GPU-BF algorithm is $O(n^2)$. In recent midrange graphics card, as the one we used in this work, it can be thrown up to 1,024 threads in parallel. This implies that, for example, from 33 objects, the complexity of the algorithm changes from $O(1)$ to $O(n^2)$.

Because all objects are arranged sequentially in memory and all threads are launched to sequential pairs of objects, we have groups of threads that are always running on sequential cells. This allows the graphics board to optimize the readings from the global memory, replacing the various accesses with a single access burst that will supply several threads. Each thread reads two AABBs, one for each object of the tested pair. Adjacent threads share the first object in the pair and the second object is adjacent in memory. This results in two coalesced readings. The only shared data between the threads is the first object in the pair. Thus, there is no need to use shared memory. However, when launching threads to sequential pairs, the GPU-BF algorithm prevents the results are saved as bits (rather than bytes), otherwise, 8 threads would try to write into the same memory cell simultaneously. This policy implementation has advantages and disadvantages: optimized memory accesses are produced, but a greater amount of data is sent from the graphics card to the processor board.

Finally, when the results come in the main memory, they are iterated. For each result, the cache is updated. However, this update is more optimized when compared to the same algorithm implemented on CPU. The GPU-BF keeps in memory the results of both the current and previous frames. Thus, instead of consulting the cache to test whether or not the pair is in the structure, we use a simple XOR operation. If the result of this operation is positive, it means that the pair status has changed, *i.e.*, the pair has not collided previously and currently collides, so it should be added to the cache. Otherwise, the pair has collided and no longer collides, and should be removed from the structure. If the operation result is zero, there is no need to perform any further action.

5. Performance Analysis

In order to perform the comparative performance analysis between the broad phase algorithms and verify the scalability of the algorithms, tests were made using four different scenarios, each of them with different objects distribution's patterns and specific movement controls, as detailed in the following sections.

5.1 Methodology

All tests were made using an Intel Core i7 2600 (3.4 GHz) machine with 8GB of RAM and Nvidia GTX630 video card, capable of launching up to 1,024 threads in parallel. For each of the four scenarios, we conducted tests with 500, 1,250, 2,000, 2,750 and 3,500 objects. Each test configuration (scenario– algorithm–number of objects) was executed five times. The averages were calculated and used for the results consolidation and generation of the diagrams. All tests have 2,048 frames and each frame progresses 1/30s in the animation time, with a total time of approximately 68s.

Several processing times were calculated for the comparative performance analysis: (1) the update time of the objects in the structures of the algorithms; (2) the time spent during the search of possibly colliding pairs of objects; and (3) the total processing time consumed during the simulation, including the time spent in collision response. For example, with the information about the time spent updating AABBs, we could analyze the difficulty of the algorithm in dealing with complex or unpredictable objects' movements.

5.2 Test Scenarios

The simulations in the four scenarios (Figure 1) start with all instantiated objects with their positions bounded by the inner boundary of a geometric cube and with random speeds.

The simulations are governed by gravity force in the first scene (first row of Figure 1). At the end of the simulation, it can be observed that all objects stop fully on the cube's bottom face.

In the second scenario (second row of Figure 1), every frame, a new random velocity is attributed to 25 objects, which are also chosen randomly. All frames generated are similar, as there is a continuous time-changing data streams, keeping the objects moving.

In the third scene (third row of Figure 1), the modeled environment is somewhat different, because rather than alter the speed of some objects, we modify

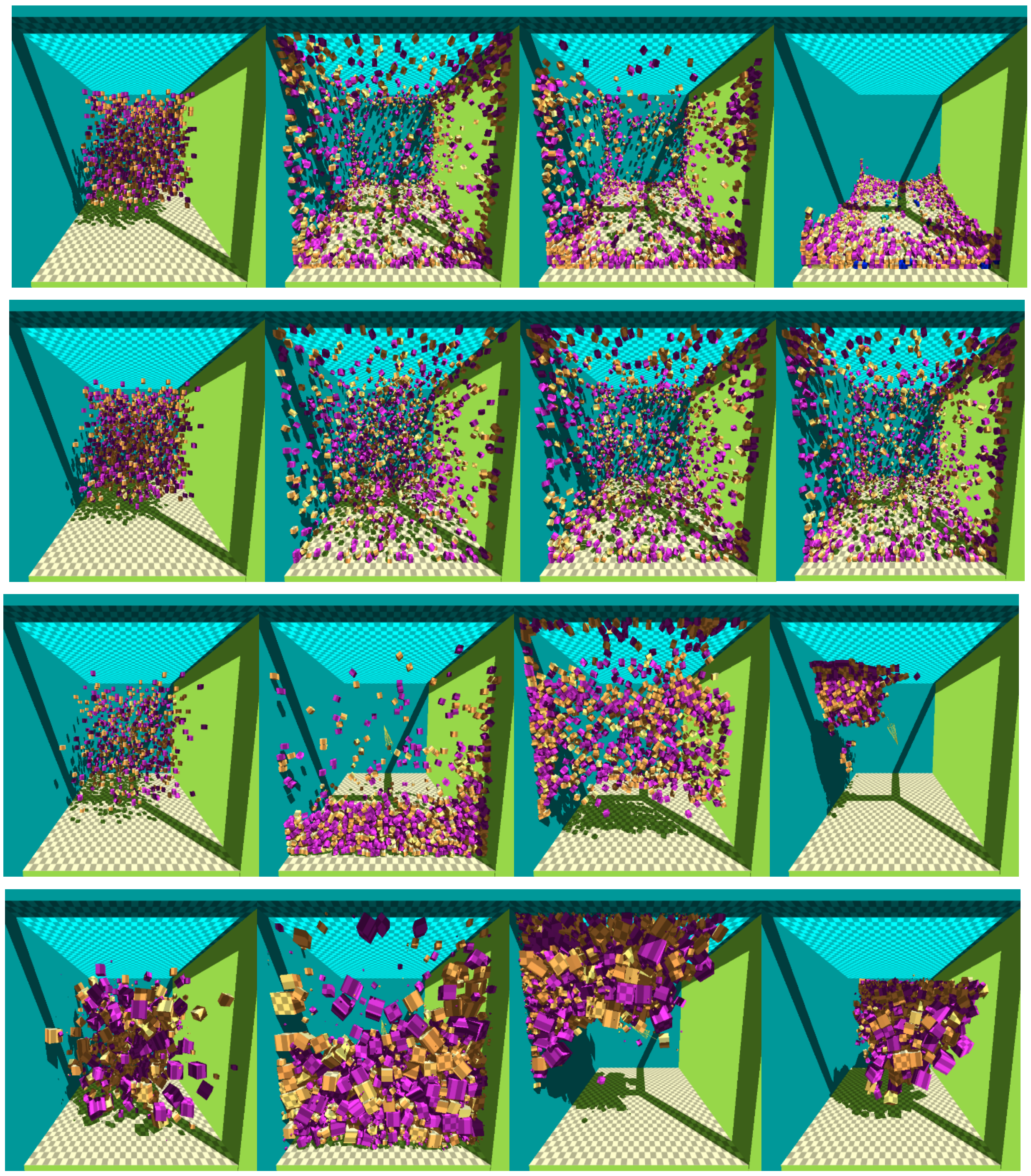

Figure 1. From the top to the bottom, four keyframes of the animations generated with scenarios 1, 2, 3 and 4, respectively.

the gravity vector (for each frame, the gravity vector is rotated around an axis e_1, which in turn is rotated around e_2 and around e_3). These three axes are initialized with the unit vectors *i*, *j*, *k* and these rotations are of the 1 / 120 radians. All objects are thrown against the cube's bottom face, and then against its sides and the top of it. Over the generated frames, the objects collide against the six faces of the cube. This movement forces that all objects are always located close to each other and constantly mixed.

Finally, the fourth test modeled (fourth row of Figure 1) corresponds to a variation of the third one. The objects in the environment have different sizes, ranging between 10% and 400% of their original size. The size distribution was divided into three classes: small (between 10% and 40%); average (between 80% and 120%); and large (between 160% and 400%). Each third of the objects is contained in one of these three classifications. The tests carried out in environments 1 and 2 use the same gravitational acceleration value of the Earth (9,798 m/s²), whereas the environments 3 and 4 use this same value multiplied by 50.

5.3 Results and Discussion

The graphical results with 1,250 objects in the scene for the environments 1, 2, 3 and 4 were quite distinct and are shown, respectively, in Figures 2, 3, 4 and 5.

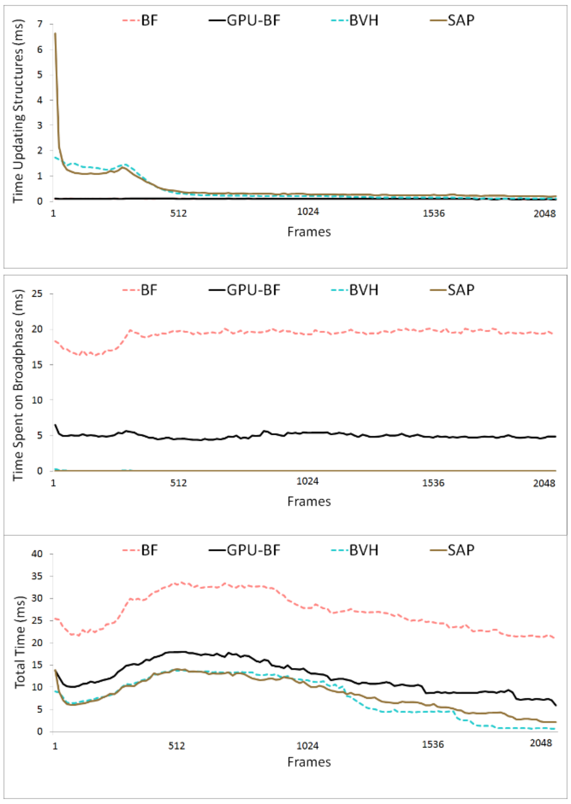

Figure 2. Diagrams generated for scenario 1 with 1,250 objects. Top, middle and bottom: the time to update the structures, the time spent on the broad phase, and the total time to process the frame, respectively.

We can observe that the first scenario is the simplest to be treated by the most robust algorithms (middle and bottom of Figure 2), since after a few seconds many objects are inert in the data structures (top of Figure 2, from the frame 512).

In the second scene, all the algorithms show a similar level of complexity during the animation, generating more stable curves with constant slopes (top, middle and bottom of Figure 3).

However, in the third scenario, due to the constant shuffling and grouping of objects during the simulation, the data structures used by the SAP and BVH algorithms had to be the most required, demonstrating a clear difficulty in the upgrade process (top of Figure 4). On the other hand, the BF and GPU-BF algorithms did not show this behavior, since they only updated values of the AABBs in their lists of objects. This is an important feature of brute force methods, *i.e.*, each object takes the same time to be updated, regardless of whether in motion or not (middle and bottom of Figure 4).

In the fourth case, there is a marked difficulty of the robust algorithms to handle more complex scenarios. Additionally, we observe that the SAP algorithm has greater difficulty in dealing with objects of varying sizes (Figure 5). This results from the fact that small objects, when displaced in the lists of projections tend to cross many other objects and thus perform various operations to keep the list organized. The BVH algorithm, in turn, shows a better performance in the treatment of small and large objects, since the small objects, as they move, hardly change their bounding volumes, resulting in no major performance impact of upgrades.

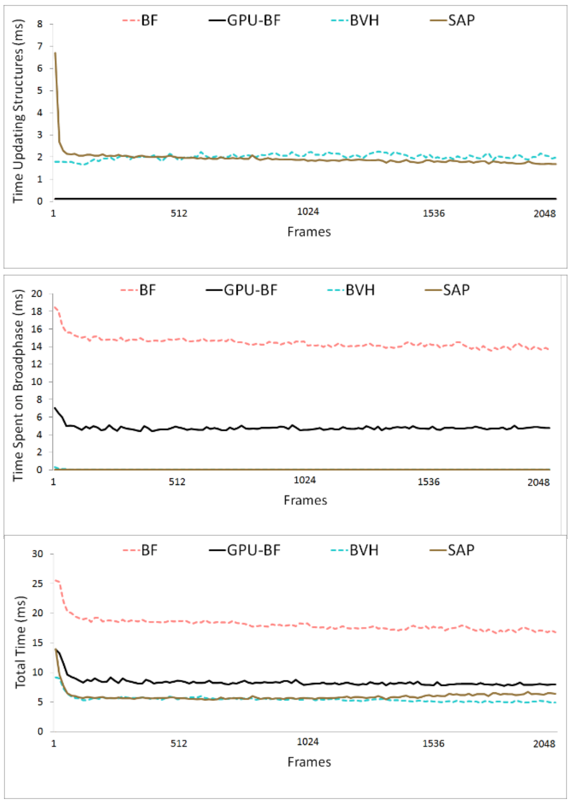

Figure 3. Diagrams generated for scenario 2 with 1,250 objects. Top, middle and bottom: the time to update the structures, the time spent on the broad phase, and the total time to process the frame, respectively.

In terms of performance, in all tests the GPU-BF algorithm shows superior results to the BF algorithm and practically similar to the most robust CPU methods tested (bottom of Figures 2, 3, 4 and 5). In addition, in scenarios 3 and 4 (Figures 4 and 5, respectively) the GPU-BF algorithm is very competitive compared to the others. We can clearly note, by considering the total running time for the fourth scenario (bottom of Figure 5), that the GPU-BF algorithm has the best performance, almost during the entire animation. We can also verify a greater difficulty of the SAP algorithm in treating the objects in such an environment, when compared to the performance demonstrated by the BVH algorithm.

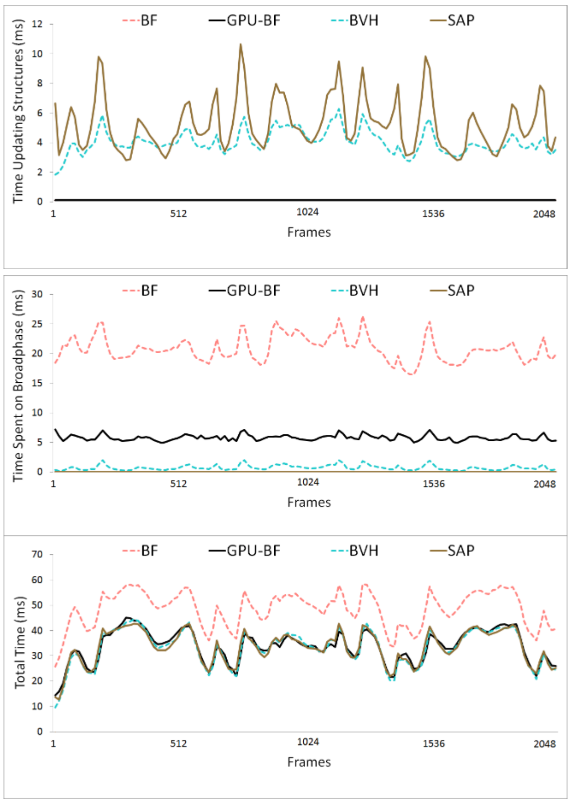

Figure 4. Diagrams generated for scenario 3 with 1,250 objects. Top, middle and bottom: the time to update the structures, the time spent on the broad phase, and the total time to process the frame, respectively.

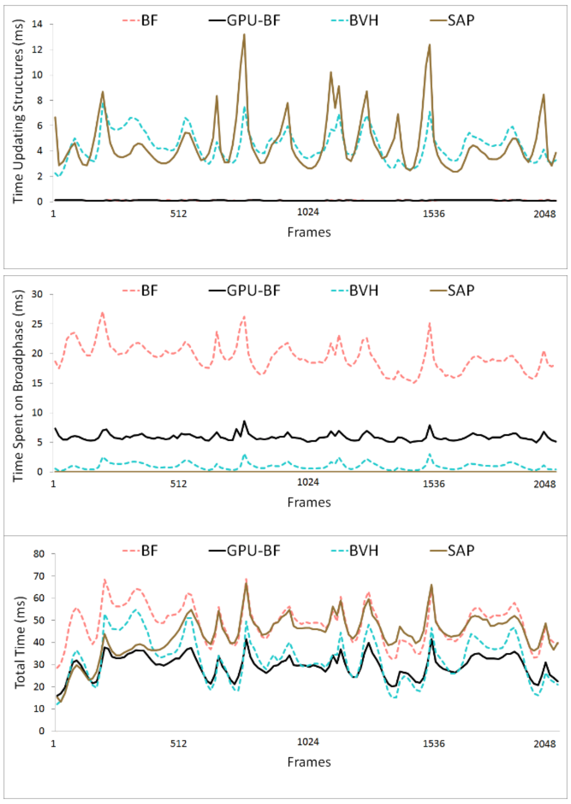

Figure 5. Diagrams generated for scenario 4 with 1,250 objects. Top, middle and bottom: the time to update the structures, the time spent on the broad phase, and the total time to process the frame, respectively.

Moreover, in the first frames of the animation we perceive a considerably lower performance of the algorithms than in subsequent frames. This probably occurs for the following reasons: (1) in the first frame all objects are added to the scene and, therefore, to the structures; (2) algorithms such as SAP and BVH have to perform sorting and building the tree for all objects, and particularly in the case of BVH, all objects are initialized in the dynamic tree; and (3) in the first frames, few collisions occur, however, in the following frames, various objects have an intersection, due to their initial speeds.

The moments in which all algorithms suffer some negative impact on performance are the times when new collisions occur. In those moments, all algorithms are forced to make changes in the colliding pairs cache. These moments are easily observed in the total time of scenarios 3 and 4 (bottom of Figures 4 and 5, respectively), at peak points in the graphs. For example, changes in the gravity vector generated a movement in which the objects were thrown against the cube's faces, resulting in a large amount of collisions. In the following frames, the objects that were located behind the first objects collided with the objects that were already on the cube's wall.

Finally, the graphs in Figure 6 show the total processing times for all test scenarios, as the number of objects increases. Note the superiority of the GPU-BF algorithm compared to the BF, the similar performance of the algorithms in scenario 3 (Figure 6.c) and the advantage and disadvantage of the BVH and SAP algorithms, respectively, in scenario 4 (Figure 6.d).

6. Conclusion and Future Work

We presented a detailed performance analysis conducted using four test scenarios, running a new GPU-based algorithm for broad phase collision detection (which we have implemented in the *Bullet* library, using CUDA) and three CPU-based algorithms available in the same Physics library.

The results show that the heterogeneous algorithm implemented is competitive when compared to some robust methods available in the *Bullet*, particularly in scenes with a large number of objects whose movements are complex and unpredictable.

We believe that initiatives like this, which explore solutions for new implementations of collision algorithms running on GPU and operating asynchronously with the CPU, are extremely important and useful for game *designers*, especially in the area of digital games based on Physics, considering there are other elements of the animation which are not good candidates for GPU paralellization, *e.g.*, sound and artificial intelligence, that can thus be executed during the broad phase calculation.

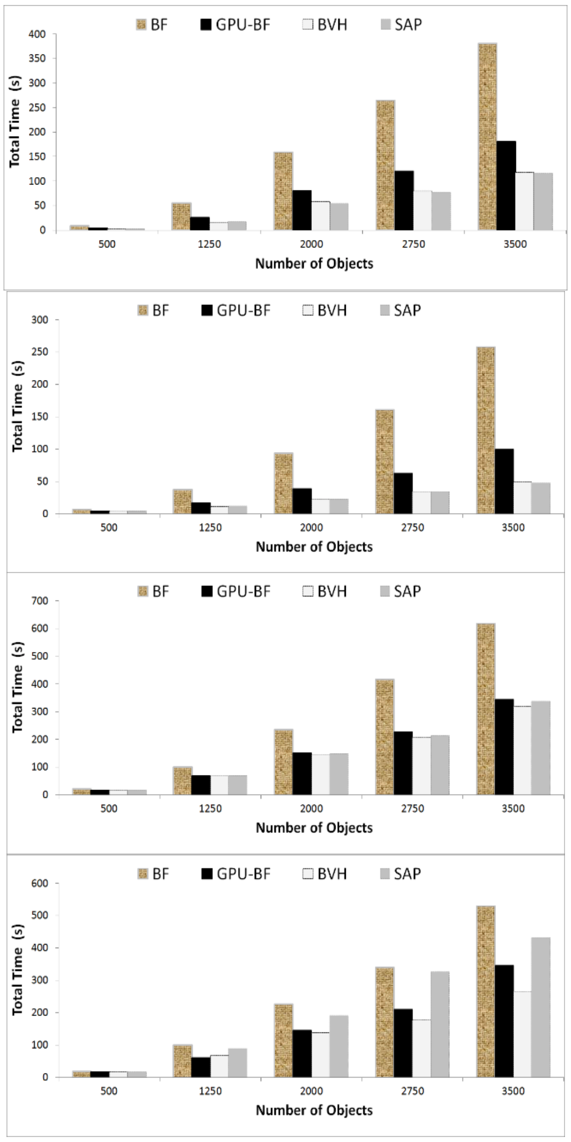

Figure 6. In (a), (b), (c) and (d) the total processing time as the number of objects in the scene increases (from 500 to 3,500), respectively, for scenarios 1, 2, 3 and 4.

As future work, we anticipate the interest in exploring other approaches for performing broad phase collision detection on GPUs and to test then against other GPU based solutions, such as [Lo *et al.,* 2013]. Moreover, we plan to compare the performance of algorithms taking into account the operations of insertion and removal of objects, as well as to test asynchronous approaches in which the calculation of broad phase collision detection is performed in parallel to the CPU, while other tasks are performed by the CPU processing. Additionally, we plan to explore in detail and compare the behavior of *btCudaBroadPhase* algorithm, which is available in the *Bullet* library, with our GPU-BF algorithm, since the time this work was submitted the former produced inconclusive results. Although most parameters of this algorithm have an intuitive meaning (3D grid dimensions, number of small and large proxies, maximum number of colliding pairs per proxy, number of objects per cell, world's AABB, etc.), they vary considerably from scenario to scenario, thus, tuning its parameters accordingly is not a trivial task. In most cases, these parameters had an unpredictable and unstable effect on the simulated motions. Therefore, this issue deserves indeed further attention. Finally, another possibility for future work might be the implementation of the remaining stages of the collision detection on GPU, with the aim of reducing the amount of data sent and received from the graphics card and performing, for bytes sent, a large number of operations.

Acknowledgements

This project was partially funded by FUNCAP-CE (Fundação Cearense de Apoio ao Desenvolvimento Científico e Tecnológico), Open Call application 36/2012.

References

2D BOY GAMES, 2008. World of Goo. Available at www.2dboy.com/games.php. Accessed 30/06/14.

AVRIL, Q., GOURANTON, V., ARNALDI, B., 2010. Broad Phase Collision Detection Algorithm Adapted to Multi-cores Architectures. In Proc. of Virtual Reality International Conference. Laval, France.

BERGEN, G. VAN DEN., 2004. Collision Detection in Interactive 3D Environments. Morgan and Kaufmann Publishers, 2004.

COUMANS, E., 2014. Bullet Physics Library. Available from www.bulletphysics.org. Accessed 30/06/14.

COUMANS, E., 2014. Bullet 3.x teaser. Available at www.bulletphysics.org/wordpress/?p=381. Accessed 30/06/14.

COUTINHO, M. G., 2001. Dynamic Simulations of Multibody Systems. LA, CA: Springer-Verlag.

LIU, F., HARADA, T., LEE, Y., KIM, Y. J., 2010. Real-time Collision Culling of a Million Bodies on Graphics Processing Units. ACM TOG, vol. 29(6), 154.

GRAND, S. L., 2007. GPU Gems 3. Boston, MA: Addison-Wesley Professional, p. 697–721.

HAVOK, 2014. Havok Physics. Available at www.havok.com/products/physics. Accessed 30/06/14.

HUBBARD, P. M., 1995. Collision Detection for Interactive Graphics Applications. IEEE TVCG, vol.1(3), p. 218–230.

LO, S. H., LEE, C. R., CHUNG, I. H., AND CHUNG, Y. C., 2013. Optimizing Pairwise Box Intersection Checking on GPUs for Large-scale Simulations. ACM TOMACS, vol. 23(3), Article 19, p. 1-22.

MIRTICH, B., 1997. Efficient Algorithms for Two-Phase Collision Detection. Practical Motion Planning in Robotics: Current Approaches and Future Directions, p. 203-223.

MOORE, M., WILHELMS, J., 1988. Collision Detection and Response for Computer Animation. In Proc. of the 15th SIGGRAPH. ACM Press, p. 289–298.

NVIDIA, 2014. Nvidia CUDA Parallel Programming and Computing Platform. Available at www.nvidia.com/object/cuda_home_new.html. Accessed 30/06/14.

NVIDIA, 2014. CUDA C Programming Guide. Available at www.docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 30/06/14.

NVIDIA, 2014. Physix Library. Available at www.geforce.com/hardware/technology/physx. Accessed 30/06/2014.

ROCHA, R.S., RODRIGUES, M.A.F., TADDEO, L.S. 2006. Performance Evaluation of a Hybrid Algorithm for Collision Detection in Crowded Interactive Environments. In Proc. of the SIBGRAPI'06. Manaus-AM, Brazil: IEEE CS Press. p. 86–93.

ROVIO ENTERTAINMENT, 2009. Angry Birds. Available at www.rovio.com/en/our-work/games/view/1/angry-birds. Accessed 30/06/14.

KLOONIGAMES, 2009. Crayon Physics. Available at www.crayonphysics.com. Accessed 05/07/14.

STOCK, P., 2006. Armadillo Run. Available at www.armadillorun.com. Accessed 05/07/14.