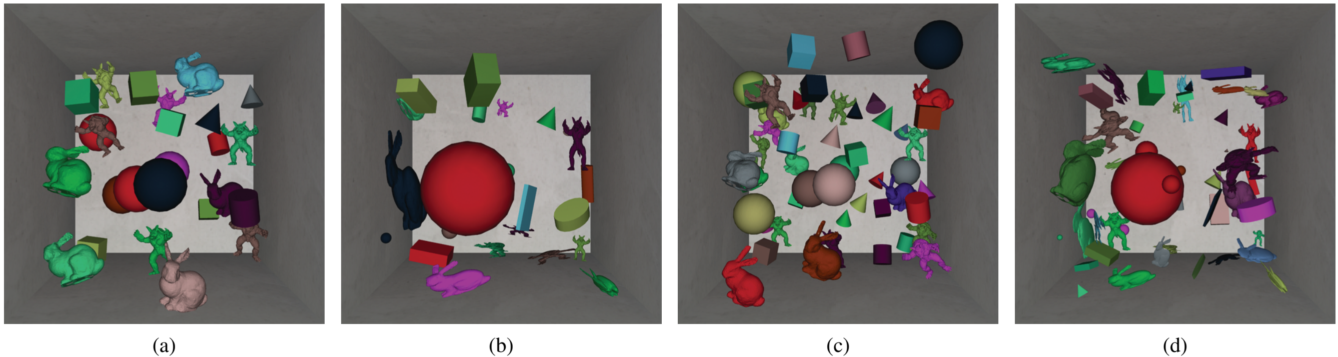
A Framework for Benchmarking Interactive Collision Detection

Muiris Woulfe* Michael Manzke†

Trinity College Dublin



(a) (b) (c) (d)

Figure 1: Screenshots of our framework executing a variety of benchmarks, each with the angular velocity of the simulated objects uniform-distributed between (–0.25, –0.25, –0.25) and (0.25, 0.25, 0.25). (a) 25 objects of size 250. (b) 25 objects with uniformly-distributed sizes between 50 and 550. (c) 50 objects of size 400. (d) 50 objects with uniformly-distributed sizes between 50 and 850.

Abstract

Collision detection is a vital component of applications spanning myriad fields, yet there exists no means for developers to analyse the suitability of their collision detection algorithms across the spectrum of scenarios that could be encountered. To rectify this, we propose a framework for benchmarking interactive collision detection, which consists of a single generic benchmark that can be adapted using a number of parameters to create a large range of practical benchmarks. This framework allows algorithm developers to test the validity of their algorithms across a wide test space and allows developers of interactive applications to recreate their application scenarios and quickly determine the most amenable algorithm. To demonstrate the utility of our framework, we adapted it to work with three collision detection algorithms supplied with the Bullet Physics SDK. Our results demonstrate that those algorithms conventionally believed to offer the best performance are not always the correct choice. This demonstrates that conventional wisdom cannot be relied on for selecting a collision detection algorithm and that our benchmarking framework fulfils a vital need in the collision detection community. The framework has been made open source, so that developers do not have to reprogram the framework to test their own algorithms, allowing for consistent results across different algorithms and reducing development time.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object hierarchies; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

Keywords: collision detection, benchmark, framework

*e-mail: woulfem@tcd.ie

†e-mail: michael.manzke@cs.tcd.ie

1 Introduction

Collision detection refers to the process of determining if two simulated objects are intersecting. It is a vital component of computer-aided design (CAD), robotics, virtual environments and other interactive computer graphics applications, yet it remains a fundamental problem as collision detection forms the computational bottleneck in many of these applications [Lin and Gottschalk 1998]. Interactive applications are particularly challenging as they demand a frame rate of at least 30 Hz to maintain the illusion of visual continuity, and this duration must be sufficient to execute the entire simulation potentially comprising input processing, collision detection, physics simulation, artificial intelligence, audio generation and graphical display. This scenario becomes even more challenging when haptics are added as the collision detection must now update at 1 kHz [Vahora et al. 1999]. Despite numerous developments, it remains a challenge to perform accurate collision detection in this short time.

To address this challenge, new algorithms and adaptations of existing algorithms appear on a regular basis. These algorithms are often tested using a small selection of benchmarks designed by the algorithm developer, but these are unlikely to recreate the full range of scenarios the algorithm will encounter. This is a significant problem, as Caselli et al. [2002] determined that the behaviour of collision detection algorithms is often very sensitive to the problem being solved, since these algorithms have been optimised for specific scenarios. It would be advantageous if developers could rapidly test a spectrum of benchmarks to gain a more complete understanding of how their collision detection algorithms perform. This would

assist them in focusing on optimising those cases in which their algorithms fail to perform efficiently.

Benchmarks designed by algorithm developers also create challenges for developers of applications utilising collision detection. Execution times measured using these benchmarks are often published as results in collision detection papers, making it difficult to determine the algorithm most amenable to the application's scenarios. A developer could select an algorithm that appears to be superior to existing algorithms, but this might not perform efficiently for the scenarios used in the application. Alternatively, they could select the algorithm whose published benchmarks most closely match the application scenarios, but this removes the possibility of selecting a better algorithm whose developer did not publish certain benchmarks. In both cases, there is a significant risk that a developer will select a non-optimal algorithm. Therefore, it would be beneficial if an application developer could quickly determine the behaviour of a wide variety of collision detection algorithms using benchmarks approximating application scenarios.

To mitigate these aforementioned issues, we propose a framework for benchmarking collision detection that currently focuses on the analysis of rigid body collision detection algorithms for interactive applications utilising polygonal objects. It consists of a single generic benchmark that may be modified using a number of parameters to create a large range of practical benchmarks. Using this framework, algorithm developers can test their algorithm against a wide range of potential benchmarks to determine which are problematic for their algorithms. After rectifying any issues highlighted by the framework, they can conclusively prove that their algorithm works well across a broad range of potential scenarios. Alternatively, application developers can use our framework to implement benchmarks approximating the scenarios found in their applications, to quickly check a wide range of potential algorithms in order to determine the most appropriate one for their application. Therefore, our framework should prove invaluable to both algorithm and application developers.

Yet there exist a number of challenges in designing this framework. Firstly, the framework must be sufficiently realistic to recreate scenarios found in interactive applications. Secondly, it must be adequately flexible to emulate all common scenarios, but this flexibility must not result in an overly complex framework. Thirdly, the framework must be extensible to allow for the integration of new collision detection algorithms and not restricted to currently available algorithms. Fourthly, it must record accurate and relevant performance data. Finally, for the framework to be useful, it must gain traction with the collision detection community. We believe that we have solved the first four challenges and that these work towards achieving the final. Details of our solutions are provided in the following sections.

2 Related Work

Researchers in many areas of computer graphics use standardised benchmarks. For example, there exist two such benchmarks in the field of raytracing. The Standard Procedural Databases (SPD) [Haines 1987] consist of simple scenes primarily containing recursively-generated objects, while A Benchmark for Animated Ray Tracing (BART) [Lext et al. 2001] consists of detailed animated environments. For GPUs, 3DMark Vantage [Futuremark 2009] is widely used to benchmark their performance. This application consists of two graphics tests and six feature tests designed to exercise every feature found in modern graphics cards. The standardised benchmarks used in these areas allow developers to accurately test the performance of their designs across every likely scenario, while facilitating comparisons between different implementations.

Despite the existence of these benchmarks, there exist very few standardised benchmarks in the area of collision detection. Zachmann [1998] outlines a benchmark in a paper proposing dynamically aligned discrete-orientation polytope (DOP) trees. However, the benchmark focuses on offline algorithms for unmoving objects with very large numbers of vertices. As a result, the benchmark only provides car doors, car door locks, car exteriors and sets of pipes as models and these are unsuitable for accurately benchmarking algorithms designed for interactive applications. Trenkel et al. [2007] outlines a development of this benchmark, which uses a castle, a helicopter, a laurel wreath, the Apollo 13 capsule, sets of pipes and a chandelier as models. Two parameters, object distance and number of vertices, are used to alter the benchmark. However, as for the previous benchmark, it is limited to unmoving objects with large numbers of vertices. Moreover, the variety of parameters is less wide-ranging than ours. A different approach is taken by Caselli et al. [2002], which tests a range of collision detection algorithms for their suitability in probabilistic motion planners, by attempting to plan the motion of an object through a variety of grid-like structures. However, the described benchmark is not of general utility and is restricted to a fixed set of scenarios. For haptics, Cao [2006] has created a framework for emulating a haptic device to which benchmarks can be attached. However, this is unsuitable for benchmarking non-haptic algorithm behaviour. Due to the various problems with each of these benchmarks, they have failed to gain traction with the collision detection community.

Most collision detection developers design their own benchmarks. For example, I-COLLIDE [Cohen et al. 1995] was tested using a static scene comprising a single object type parameterised by the quantity of objects, their complexity, their linear velocity, their angular velocity and the density of objects in the scene. Van den Bergen [1997] uses three static benchmarks consisting of a single object type (either a torus, teapot or X-wing) initialised at random locations within an enclosing cube. Govindaraju et al. [2005] take a different approach as their algorithm is designed to work with deformable bodies. Their first three benchmarks model cloth of different complexities, the fourth models catheter in liver chemoembolisation and the fifth models folding curtains. Even though each of these papers has been widely cited, it appears that their benchmarks have never been used outside of these papers, as they are insufficient to test the behaviour of algorithms across a wide spectrum of scenarios.

Despite the lack of standardised collision detection benchmarks, there has been significant research into collision detection algorithms. This research can be divided into two distinct phases, referred to as the *broad phase* and the *narrow phase* in the taxonomy proposed by Hubbard [1993]. The broad phase is the first collision detection phase, which uses an approximate and fast test to enumerate appropriate pairs of objects into a potentially colliding set (PCS), while the narrow phase is the second phase, which checks the PCS using an accurate or exact algorithm.

Broad phase algorithms have been traditionally based on spatial partitioning. Examples include sweep and prune [Baraff 1992; Cohen et al. 1995], uniform grids, hierarchical grids, spatial hashing, hierarchical spatial hashing [Mirtich 1998a], octrees and k-d trees [Bentley 1975; Friedman et al. 1977].

Narrow phase algorithms for rigid-body collision detection can been classified in terms of the bounding volume hierarchy (BVH) used to model the objects. A multitude of BVHs have been proposed, including sphere trees [Quinlan 1994], oriented bounding box (OBB) trees [Gottschalk et al. 1996], axis-aligned bound-

ing box (AABB) trees [van den Bergen 1997] and k-DOP trees [Klosowski et al. 1998]. Non-BVH based algorithms include the Lin-Canny [Lin and Canny 1991] and Voronoi-Clip (V-Clip) [Mirtich 1998b] algorithms, which both track the closest features of polyhedra, and the Gilbert-Johnson-Keerthi (GLK) algorithm [Gilbert et al. 1988; Gilbert and Foo 1990], which determines intersection between two polyhedra. All of these algorithms have traditionally been designed for performing collisions at discrete time intervals. However, continuous collision detection methods have also been proposed [Redon et al. 2002]. Detailed surveys of collision detection algorithms can be found in Lin and Gottschalk [1998], Jiménez et al. [2001] and Lin and Manocha [2004].

Over the past few years, there has been significant research interest in the area of deformable bodies. The same broad phase spatial partitioning techniques can be used as for the rigid-body case, although there exist some techniques specialised for use with deformable bodies [Luque et al. 1996]. For the narrow phase, deformations are usually solved by refitting the deformed object's BVH. Efficient BVH refitting algorithms exist for both AABB trees [Larsson and Akenine-Möller 2001] and sphere trees. There also exist narrow phase spatial subdivision techniques such as Teschner et al. [2003], which presents an alternative based on spatial hashing. Deformable collision detection techniques can be adapted, as for the rigid body case, to perform continuous collision detection [Redon et al. 2005]. A detailed survey of deformable collision detection algorithms can be found in Teschner et al. [2005].

3 Design

As previously outlined, the range of applications utilising collision detection is extremely large. It would be difficult to create a tool that effectively emulates scenarios found in every potential application. For this reason, we decided to narrow our focus to scenarios encountered in interactive applications.

An analysis of modern interactive applications found that the majority use some form of physics engine to control the motion of the objects within the simulated environment. To accurately recreate such application scenarios, we determined that it was vital to use a physics engine in our framework. Our analysis also indicated that the majority of objects in these applications are modelled as rigid bodies due to their simplicity and rapid execution times. For this reason, we decided to initially concentrate on rigid body collision detection, although our framework could easily be expanded to support deformable bodies, particle systems and articulated rigid bodies in the future. Similarly, we have focused on collisions between polygonal objects and left parametric surfaces, implicit surfaces and voxel objects as future work.

We also determined that the majority of these applications contain a ground plane. However, many existing benchmarks have no ground plane and we believe that this is a significant oversight, since in simulations involving gravity, the majority of objects will tend to come to rest on the ground. If the collision detection system has been poorly designed, objects on the ground may need lengthy tests before they can be eliminated from consideration, as they will all lie in the same plane and overlap along one axis. Algorithms that are more sophisticated avoid this problem by processing axes in a specific order or by deactivating stationary objects. Therefore, it is vital to implement a ground plane so that the likely performance of a collision detection algorithm can be quantified.

Based on these identified requirements, we constructed a generic benchmark consisting of a cube containing a variable quantity of objects. The size of the cube is determined by multiplying the number of objects by 50.

3.1 Parameters

To translate the generic benchmark into a variety of practical benchmarks, we have defined ten parameters that can be used to specify the initial state of each benchmark. These parameters mimic the standard geometric and physical properties of rigid bodies and were chosen after a preliminary analysis of a variety of interactive applications. We believe these should enable developers to accurately recreate real application scenarios as benchmarks. The dialog boxes used to control the parameters are illustrated in Figure 2. Each of these parameters and their potential impact on the behaviour of the collision detection algorithm are outlined below.

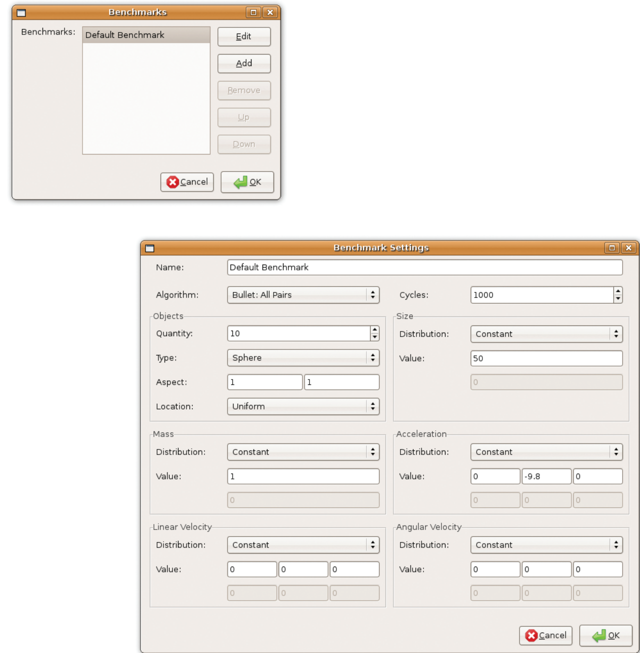


Figure 2: Dialog boxes used to control the parameters.

Cycles This parameter controls the number of collision detection cycles that will be executed, with the default being 1000 cycles.

Different benchmarks require different numbers of cycles before reaching a steady state. For example, an object with a large mass falling under the influence of gravity will reach a steady state quickly, once the object comes in contact with the floor. However, if the mass were reduced substantially, the object would take longer to come into contact with the floor and it would take more cycles for the benchmark to reach a steady state. This parameter provides an opportunity to adapt the number of cycles to the behaviour of the benchmark under consideration.

Objects – Quantity This parameter modifies the quantity of simulated objects, with the default being 10 objects.

Object quantity influences algorithm behaviour, since greater quantities tend to take longer to process than smaller quantities. Poorly designed algorithms may not be able to cope with larger quantities and may, for example, make inefficient memory requests.

Objects – Type This parameter selects the type of the simulated objects, with an option of spheres, cuboids, cylinders, cones, Stanford Bunnies [Turk and Levoy 1994], Stanford Armadillos [Krishnamurthy and Levoy 1996] or a combination of objects determined by the uniform probability distribution. The default is spheres. As the Stanford Bunny and Armadillo are concave objects, we construct them as a compound shape of convex hulls.

Object type influences algorithm execution speed, as some algorithms optimise for specific primitive objects while others consider every object using a generic bounding volume. Additionally, complex, concave objects tend to be processed differently than simpler convex primitives and can execute at very different speeds with different algorithms.

Objects – Aspect This parameter determines the aspect ratio of the simulated objects. This ratio can be considered to be of the form 1:y:z where y and z are the user-modifiable options. The default is 1:1:1.

Aspect ratio influences the shape of the objects. Most collision detection algorithms utilise bounding volumes (BVs) and some types of BV may be difficult to fit around long, thin objects. The use of this parameter can reveal a suboptimal BV selection, as many collisions will occur in the PCS and the collision detection algorithm will execute slowly.

Objects – Location This parameter selects the probability distribution used to populate the cube, from a choice of the uniform or normal distributions. The default is the uniform distribution. Objects are placed inside the cube according to the selected distribution. Once an object location is generated, it is checked to ensure it overlaps no other object or the cube walls. If it overlaps, a new location is generated according to the distribution. The distribution parameters are determined by the size of the cube. For the uniform distribution, the minimum and maximum are the coordinates of the cube walls. For the normal distribution, the mean is 0 and the variance is the coordinates of the cube walls in the positive direction divided by three. This choice of variance ensures 99.7% of objects will be within the walls. The 0.3% of objects outside the walls will be detected and new normal locations within the walls will be generated.

The effect of the uniform distribution is to place objects throughout the cube, while the normal distribution tends to cluster the objects in the centre of the cube. A cluster of objects is likely to result in increased collisions, as the objects within the cluster interact with one another.

Size This parameter determines the size of the simulated objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant 50. When using the probability distributions, each side of each simulated object will be of differing size. Any sizes less than or equal to zero will be discarded and a new size will be generated.

Larger objects will tend to result in more collisions. The uniform distribution will tend to create objects across a full range of sizes, while the normal distribution will tend to create objects whose sizes are centred on the mean.

Mass This parameter controls the mass of the simulated objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant 1.

The mass of the objects determines how quickly the objects will move under the influence of the acceleration. If the mass is determined by a distribution, the varying masses will change the rate at which objects move relative to one another. This will affect the number of collisions that occur. The mass will also determine how the objects move once they reach the ground plane. Objects with a small mass may tend to roll, while those with a larger mass will tend to come to a stop more quickly. Therefore, the mass will also determine the number of collisions that occur between objects and the ground plane.

Acceleration This parameter controls the acceleration that acts on all simulated objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant (0.0, –9.8, 0.0), corresponding to acceleration due to gravity. The acceleration is applied as a single value to the entire simulated environment.

Acceleration determines the motion of the objects within the world, influencing the number of collisions. For example, the acceleration could be used to move all of the objects towards one wall of the cube. Alternatively, if zero acceleration were applied, objects would float, resulting in zero collisions, unless a linear or angular velocity were employed.

Linear Velocity This parameter controls the linear velocity of the simulated objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant (0, 0, 0).

Linear velocity determines the linear motion of objects. This parameter has an effect similar to that of the acceleration, but different values can be given to different objects by using a probability distribution. If objects move towards each other, this will tend to maximise the number of collisions that occur.

Angular Velocity This parameter controls the angular velocity of the simulated objects, using a constant value or a choice of the uniform or normal probability distributions. The default is a constant (0, 0, 0).

Angular velocity determines the rotational motion of objects. This adds a degree of complexity to the simulation due to the difficulty of fitting BVs to rotated objects. Moreover, different BVs fit rotated objects very differently, so the optimality of the collision detection algorithm's choice of BV can be tested using this parameter.

Remarks The random number generators that power the uniform and normal probability distributions are individually seeded with a constant value of hexadecimal AAAAAAAA to make the benchmarks repeatable. Each uniform distribution takes a minimum and a maximum value as parameters, while each normal distribution takes a mean and a variance value as parameters.

3.2 Application Programming Interface (API)

Since the purpose of our framework is to allow algorithm and application developers to analyse a variety of collision detection algorithms, we have designed it to be extensible. To this end, we have created an API for retrieving geometry data from the framework and supplying the framework with a list of collisions.

To add an algorithm to the framework, the developer first inherits from a collision detection class. To initialise the appropriate algorithm data structures, the inherited class' initialisation method can be overridden. Inside, a variety of method calls can be used to retrieve object properties and geometry.

To perform the collision detection routines, the class' collision detection method can be overridden with the appropriate collision detection method from the algorithm under consideration. Inside this method, the algorithm can call a selection of framework methods to retrieve object status data. This method will be called during each physics engine time step. For each collision, a point on both colliding objects, a contact normal and the distance between the two contact points is required. Using this data, the physics engine is able to perform the appropriate physics calculations before processing the next step.

4 Implementation

Our framework is implemented in C++. For the uniform and normal distributions, the relevant classes defined in the Technical Report 1 (TR1) [JTC1/SC22/WG21 2005] additions to the C++ standard library are used. The Boost 1.36.0 library [Boost 2009] provides some additional data structures. Single precision is used for floating-point numbers, but this can be changed to double precision using an appropriate preprocessor directive.

For the physics engine, we selected Bullet Physics SDK 2.74 [Coumans 2009], due to its status as a mature open source cross-platform engine featuring collision detection and physics, optional multithreaded execution and a PlayStation 3 port. All of these were deemed desirable attributes as they expand the flexibility of the framework.

To visualise the benchmark and the effect of modifying the parameters, a graphical display was created using OpenGL [Khronos Group 2009]. A GUI, created using the cross-platform open source wxWidgets 2.8.7 library [wxWidgets 2009], allows the user to visually modify the parameters.

To test our framework and acquire sample results, a collision detection system based on the one provided with Bullet is included. For comparison purposes, the user can change the broad phase algorithm, although the narrow phase algorithm remains constant in this initial implementation. The three broad phase algorithm options are the all pairs brute force algorithm (referred to as "simple" by Bullet), the sweep and prune algorithm [Baraff 1992; Cohen et al. 1995] and the custom dynamic bounding volume tree (DBVT) algorithm, which is based on AAABs. Alternative algorithms can easily be added to our framework using its aforementioned API. Both the broad and narrow phases can be changed using this API.

Screenshots of our framework executing a variety of benchmarks are shown in Figure 1.

5 Results

Our framework was compiled with G++ 4.2.4 and optimised for speed. It was executed on an AMD Opteron 2350 2 GHz quad core CPU with 8 GB RAM running 64-bit Ubuntu Linux 8.04. For each of our experiments, we recorded the broad phase, narrow phase and complete collision detection execution times using high-resolution performance counters.

Our first experiment uses the sweep and prune algorithm to execute a benchmark consisting of 5000 objects whose types are determined by the uniform distribution. The variation in execution times across 1000 collision detection cycles is plotted in Figure 3.

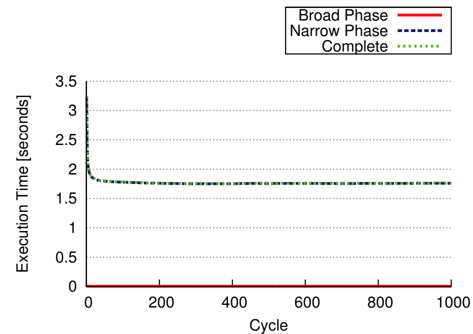


Figure 3: Results from the first experiment.

From our results, it is evident that the algorithm has a significant startup cost as the data structures are initialised during the first cycle. For the subsequent cycles, execution times remain relatively constant. We can also see that an insignificant amount of time is spent in the broad phase, with the narrow phase accounting for most of the complete execution time.

Our second experiment uses Bullet's three collision detection algorithms to execute benchmarks with object types determined by the uniform distribution. The quantity of objects is varied between 1000 and 5000, and the z value of the aspect ratio is varied between 1 and 5, to create twenty-five variations of the benchmark. The mean execution times, averaged over 1000 collision detection cycles, are graphed in Figure 4.

From our results, we can see that for each algorithm, the aspect ratio has a negligible effect while the number of objects has a very significant effect. Examining the performance of the different algorithms reveals that sweep and prune and DBVT are the most efficient in the broad phase. This is to be expected as these algorithms, unlike all pairs, exploit coherence. However, the situation changes for the narrow phase as sweep and prune takes significantly longer to execute than all pairs, suggesting that sweep and prune's efficient broad phase comes at the expense of an inefficient narrow phase. It is likely that Bullet's implementation of sweep and prune creates a larger PCS than all pairs. With regard to the complete execution time, DBVT performs best, followed by all pairs and then sweep and prune. This holds true for each benchmark, suggesting that the behaviour of these algorithms is not significantly affected by changes to either object count or aspect ratio.

Our third and final experiment uses Bullet's three collision detection algorithms to execute a wide range of benchmarks that demonstrate all of the framework's parameters, except the aspect ratio as this was analysed in the second experiment. These twenty-two benchmarks are listed in Table 1 and their mean execution times, averaged over 1000 collision detection cycles, are graphed in Figure 5.

These graphs illustrate how different parameters influence the three algorithms. It is clear that the quantity and complexity of the simulated objects has the greatest influence on the performance of the algorithms. In particular, we can see that cones do not work efficiently with sweep and prune and that distributing objects normally is less efficient than distributing them uniformly. Other parameters have a lesser influence, but a certain degree of variation is evident. As for the second experiment, we can see that sweep and prune

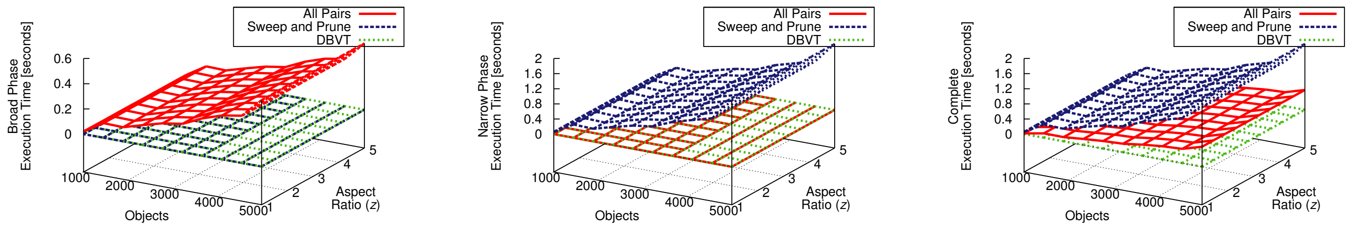


Figure 4: Results from the second experiment.

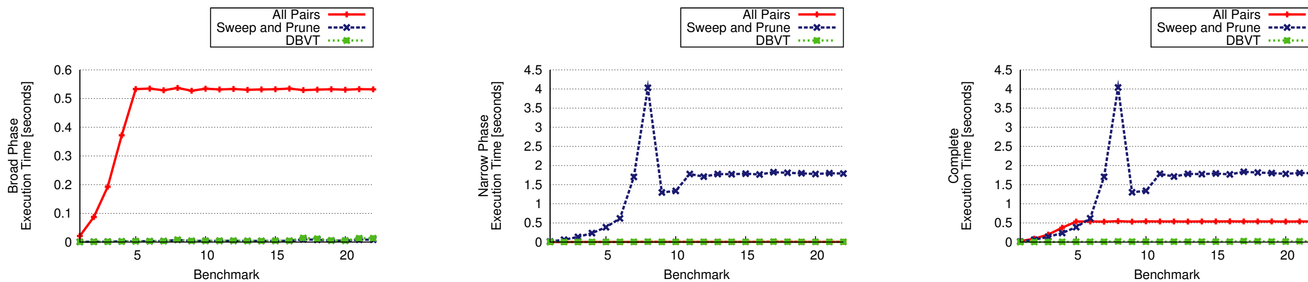


Figure 5: Results from the third experiment.

works well in the broad phase but poorly in the narrow phase. More significantly, from the complete execution time, it is clear that for low numbers of simple objects, sweep and prune performs better than all pairs, but for larger numbers of more complex objects, all pairs is more efficient. However, our primary finding is that DBVT works significantly better in all cases.

Our unexpected findings in these three experiments have demonstrated the need for our framework. Conventional wisdom suggests sweep and prune is the most efficient broad phase algorithm but we have discovered that, at least with Bullet's implementation, sweep and prune's efficiency may come at the expense of an inefficient narrow phase. Moreover, our findings recommend the use of DBVT for all scenarios similar to those found in the benchmarks used in our experiments. However, if a different scenario were to be found in an application utilising collision detection, we recommend recreating this scenario as a benchmark in our framework to determine whether DBVT is still the most efficient algorithm.

6 Conclusion and Future Work

We have outlined the design of a framework for benchmarking collision detection consisting of one generic benchmark that can be adapted by means of parameters to define a wide range of practical benchmarks. Using our framework, we tested three collision detection algorithms supplied with the Bullet Physics SDK. Our results demonstrate that the three algorithms do not behave consistently across all benchmarks, highlighting the utility of the framework. We also determined that contrary to popular belief, the sweep and prune algorithm is not the optimal solution for the benchmarks we tested. Therefore, we believe that our framework should prove invaluable to collision detection algorithm developers wishing to determine the performance of their algorithm across a wide spectrum of benchmarks, and to interactive application developers who wish to find the optimal algorithm for their scenarios.

To facilitate these developers, we are making our framework available for download under the BSD licence from `http://gv2.cs.tcd.ie/benchmarkcd/`. It compiles and executes under both Linux and Microsoft Windows. We also provide all of the experiments in this paper, which can either be used directly or as a starting point for more targeted evaluations. By providing our framework for download, developers do not have to reprogram the framework to test their own algorithms, allowing for consistent results across different algorithms and reducing development time.

In the future, we intend to analyse a further set of representative interactive applications, with a view to determining a minimal but complete list of parameters suitable for describing every scenario likely to be encountered by a collision detection algorithm. Adding these additional parameters will clearly make our framework more comprehensive but it will also significantly increase the possible test space. To manage this extra complexity, we intend to define a small number of benchmarks that very closely approximate the scenarios found in different types of interactive applications. Using these benchmarks, developers will be able to quickly test their algorithms against the most likely scenarios, while the full set of parameters will remain for those wishing to run more comprehensive analyses. This will also facilitate researchers, since they will be able to publish execution times of their algorithm for standardised benchmarks, allowing for the easy comparison of algorithms between papers.

We also intend to integrate additional collision detection algorithms into our framework. By adding these additional algorithms, developers would be provided with a greater range against which to test their new algorithms, without having to waste valuable development time. These algorithms should be relatively easy to integrate, as the framework's API has been designed to facilitate extensibility.

We currently provide an effective and practical framework for benchmarking the performance of rigid body collision detection. However, there exists no comparable framework for analysing deformable bodies, particle systems or articulated rigid body collision detection. We believe our framework could be easily adapted

1	1000 spheres
2	2000 spheres
3	3000 spheres
4	4000 spheres
5	5000 spheres
6	5000 cuboids
7	5000 cylinders
8	5000 cones
9	5000 Stanford Bunnies
10	5000 Stanford Armadillos
11	5000 uniformly-distributed object types
12	5000 uniformly-distributed object types location normal
13	5000 uniformly-distributed object types size uniform 50/500
14	5000 uniformly-distributed object types size normal 275/75
15	5000 uniformly-distributed object types mass uniform 1/500
16	5000 uniformly-distributed object types mass normal 249.5/83.5
17	5000 uniformly-distributed object types acceleration uniform (–250, –250, –250)/(250, 250, 250)
18	5000 uniformly-distributed object types acceleration normal (0, 0, 0)/(83.333, 83.333, 83.333)
19	5000 uniformly-distributed object types linear velocity uniform (–250, –250, –250)/(250, 250, 250)
20	5000 uniformly-distributed object types linear velocity normal (0, 0, 0)/(83.333, 83.333, 83.333)
21	5000 uniformly-distributed object types angular velocity uniform (–250, –250, –250)/(250, 250, 250)
22	5000 uniformly-distributed object types angular velocity normal (0, 0, 0)/(83.333, 83.333, 83.333)

Table 1: Benchmarks used in the third experiment. All parameters other than those specified are left at their defaults.

to support these object types, through the addition of parameters such as joint types, number of particles, elasticity and plasticity, while still supporting rigid bodies and parameterisation. Moreover, the framework's current support for detecting collisions between polygonal objects could be expanded to include parametric surfaces, implicit surfaces and voxel objects. In this way, we could extend our framework to comprehensively benchmark collision detection algorithms for all aspects of modern interactive applications.

Acknowledgements

This research is supported by the Irish Research Council for Science, Engineering and Technology (IRCSET) funded by the National Development Plan (NDP).

References

BARAFF, D. 1992. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, Ithaca, New York, USA.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept.), 509–517.

BOOST, 2009. Boost C++ libraries, 8 Feb. http://www.boost.org/.

CAO, X. R. 2006. *A Framework for Benchmarking Haptic Systems*. Master's thesis, Simon Fraser University, Burnaby, British Columbia, Canada.

CASELLI, S., REGGIANI, M., AND MAZZOLI, M. 2002. Exploiting advanced collision detection libraries in a probabilistic motion planner. *Journal of WSCG – Proceedings of the 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2002) 10*, 1–3 (4–8 Feb.), 103–109.

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 *Proceedings of the 1995 Symposium on Interactive 3D Graphics (SI3D 1995)*, M. Zyda, Ed., ACM, 189–196.

COUMANS, E., 2009. Bullet physics SDK, 22 Mar. http://www.bulletphysics.com/.

FRIEDMAN, J. H., BENTLEY, J. L., AND FINKEL, R. A. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software 3*, 3 (Sept.), 209–226.

FUTUREMARK, 2009. 3DMark Vantage. http://www.futuremark.com/benchmarks/3dmarkvantage/introduction/.

GILBERT, E. G., AND FOO, C.-P. 1990. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation 6*, 1 (Feb.), 53–61.

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

GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. 1996. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*, J. Fujii, Ed., ACM, 171–180.

GOVINDARAJU, N. K., KNOTT, D., JAIN, N., KABUL, I., TAMSTORF, R., GAYLE, R., LIN, M. C., AND MANOCHA, D. 2005. Interactive collision detection between deformable models using chromatic decomposition. *ACM Transactions on Graphics – Proceedings of the 32nd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2005) 24*, 3 (31 July–1 Aug.), 991–999.

HAINES, E. 1987. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7*, 11 (Nov.), 3–5.

HUBBARD, P. M. 1993. Interactive collision detection. In *Proceedings of the IEEE 1993 Symposium on Research Frontiers in Virtual Reality (VR 1993)*, S. Bryson and S. Feiner, Eds., IEEE Computer Society, 24–32.

JIMÉNEZ, P., THOMAS, F., AND TORRAS, C. 2001. 3D collision detection: a survey. *Computers & Graphics 25*, 2 (Apr.), 269–285.

JTC1/SC22/WG21. 2005. Draft technical report on C++ library extensions. Tech. Rep. DTR 19768, ISO/IEC, 24 June.

KHRONOS GROUP, 2009. OpenGL, 20 Mar. http://www.opengl.org/.

KLOSOWSKI, J. T., HELD, M., MITCHELL, J. S. B., SOWIZRAL, H., AND ZIKAN, K. 1998. Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (Jan.–Mar.), 21–36.

KRISHNAMURTHY, V., AND LEVOY, M. 1996. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1996)*, J. Fujii, Ed., ACM, 313–324.

LARSSON, T., AND AKENINE-MÖLLER, T. 2001. Collision detection for continuously deforming bodies. In *Proceedings of the 22rd Annual Conference of the European Association for Computer Graphics (Eurographics 2001)*, J. C. Roberts, Ed., Eurographics Association, 313–324. Short Presentation.

LEXT, J., ASSARSSON, U., AND MÖLLER, T. 2001. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications 21*, 2 (Mar.–Apr.), 22–31.

LIN, M. C., AND CANNY, J. F. 1991. A fast algorithm for incremental distance computation. In *Proceedings of the 1991 International Conference on Robotics and Automation (ICRA 1991)*, T. Hsia, Ed., IEEE, 1008–1014.

LIN, M. C., AND GOTTSCHALK, S. 1998. Collision detection between geometric models: a survey. In *Proceedings of the 1998 IMA Conference on the Mathematics of Surfaces*, R. Cripps, Ed., IMA, 37–56.

LIN, M. C., AND MANOCHA, D. 2004. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, second ed, J. E. Goodman and J. O'Rourke, Eds. Chapman & Hall/CRC, Boca Raton, Florida, USA, 787–807.

LUQUE, R. G., COMBA, J. L. D., AND FREITAS, C. M. D. S. 1996. Broad-phase collision detection using semi-adjusting BSP-trees. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games (I3D 2005)*, D. Luebke and H. Pfister, Eds., ACM, 179–186.

MIRTICH, B. 1998. Efficient algorithms for two-phase collision detection. In *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, K. Gupta and A. P. del Pobil, Eds. Wiley, Chichester, Sussex, UK, 203–223.

MIRTICH, B. 1998. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics 17*, 3 (July), 177–208.

QUINLAN, S. 1994. Efficient distance computation between non-convex objects. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation (ICRA 1994)*, H. Stephanou, Ed., IEEE Computer Society, 3324–3329.

REDON, S., KHEDDARY, A., AND COQUILLART, S. 2002. Fast continuous collision detection between rigid bodies. *Computer Graphics Forum – Proceedings of the 23rd Annual Conference of the European Association for Computer Graphics (Eurographics 2002) 21*, 3 (2–6 Sept.), 279–287.

REDON, S., LIN, M. C., MANOCHA, D., AND KIM, Y. J. 2005. Fast continuous collision detection for articulated models. *Journal of Computing and Information Science in Engineering 5*, 2 (June), 126137.

TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of the 2003 International Fall Workshop on Vision, Modeling, and Visualization (VMV 2003)*, T. Ertl, Ed., AKA, 47–54.

TESCHNER, M., KIMMERLE, S., HEIDELBERGER, B., ZACHMANN, G., RAGHUPATHI, L., FUHRMANN, A., CANI, M.-P., FAURE, F., MAGNENAT-THALMANN, N., STRASSER, W., AND VOLINO, P. 2005. Collision detection for deformable objects. *Computer Graphics Forum 24*, 1 (Mar.), 61–81.

TRENKEL, S., WELLER, R., AND ZACHMANN, G. 2007. A benchmarking suite for static collision detection algorithms. *Journal of WSCG – Proceedings of the 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2007) 15*, 1–3 (29 Jan.–1 Feb.), 105–110.

TURK, G., AND LEVOY, M. 1994. Zippered polygon meshes from range images. In *Proceedings of the 21st International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1994)*, D. Schweitzer, A. Glassner, and M. Keeler, Eds., ACM, 311–318.

VAHORA, F., TEMKIN, B., KRUMMEL, T. M., AND GORMAN, P. J. 1999. Development of real-time virtual reality haptic application: Real-time issues. In *Proceedings of the 12th IEEE Symposium on Computer-Based Medical Systems (CBMS 1999)*, IEEE Computer Society, 290–295.

VAN DEN BERGEN, G. 1997. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2*, 4, 1–14.

WXWIDGETS, 2009. wxWidgets: Cross-platform GUI library, 19 Mar. http://www.wxwidgets.org/.

ZACHMANN, G. 1998. Rapid collision detection by dynamically aligned DOP-trees. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS 1998)*, G. Burdea and S. Tachi, Eds., IEEE Computer Society, 90–97.