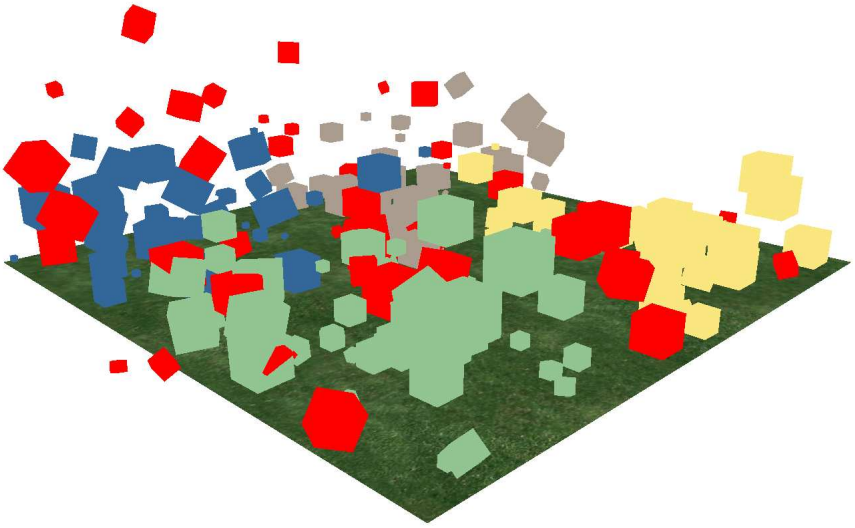
SODA CD: A Model for Scalability-Oriented, Distributed and Anticipative Collision Detection

Steve Dodier-Lazaro

SODA CD: A Model for Scalability-Oriented, Distributed and Anticipative Collision Detection

Master's Thesis



Keywords: virtual reality, collision detection, parallelism, scalability, optimistic computing

Author:
Steve Dodier-Lazaro[†§‖]

Supervisors:
Quentin Avril[‡]
Valérie Gouranton[§]

August 31, 2012

[†] ENSI of Bourges
[§] INSA Rennes
[‖] Inria Rennes
[‡] University of Rennes I

I would like to warmly thank my two supervisors Quentin Avril and Valérie Gouranton, for their unconditional support, their trust in my ability to fulfill my mission, their many tips both on research methodology and laboratory life, and for their always cheerful mood. In very few occasions I have met people that nice to work with.

I would also like to thank Bruno Arnaldi for his insightful comments on my work and his advice, and Nathalie Denis for her support with the many daily life issues that I've had to deal with.

Finally, I would like to address my thanks to the many interns that shared an office with me, as well as my colleagues from the Mimetic and VR4I teams for the time spent together, inside and outside of Inria.

Summary

In this master's thesis, we first analyze existing parallel algorithms for the problem of collision detection in virtual reality applications, as well as larger scale parallel algorithms taken from the particle simulation research field. Based on the properties we noticed to favour high scalability in these algorithms, we propose a novel approach to parallel collision detection.

In order to allow the execution of interactive applications on systems made of tens of processors with an efficient solution, our parallel collision detection framework is designed to be decentralized and to exploit all available computing resources.

More precisely, we improve the potential for scalability of virtual simulations primarily by loosening the synchronism constraints between processing units, moving from the current "synchronize after each detection and constraint solving round" scheme to several autonomous worlds that process a part of the application's simulation space at their own rhythm and synchronize with each other when an object moves from one to another. Our second contribution is a new rollback algorithm for speculative computing, that allows partial saving of the anticipated computations that get invalidated because of synchronization mechanisms.

Both of these proposals are making use of a spatial subdivision uniform grid. Each processing unit is assigned a territory made of contiguous cells from this grid, and simulates the objects within this territory. The rollback algorithm makes use of the grid by integrating rollbacks cell by cell, thus saving some independant computations whenever not all cells are interconnected as a result of objects coexisting in adjacent cells.

Processing units can work autonomously, using their own local clock and communicating exclusively with their direct neighbors in the grid, and making the results of their computations available via a circular buffer. This buffer is read by a rendering thread that allows users to visualize with the simulation. User input is also taken into account in the model, permitting the execution of interactive applications.

We also introduce theoretical leads for load balancing in our new framework, for continuous and discrete algorithms. The proposed load balancing mechanism is distributed and does not involve global communication, in order not to hinder the scalability of the framework. It is based on the exchange of spatial subdivision grid cells that compose the area managed by each processing unit.

INVALID: Finally, we discuss preliminary experimentations performed with our implementation of the framework, despite the implementation not being complete enough to allow deterministic simulations yet. We also discuss limitations of our framework and directions for future work.

Contents

Chapter 1

Introduction

Collision detection is a fundamental component of applications such as computer animation, e-learning applications, video games, and all kinds of virtual reality systems. It consists of determining if, where and when two or more objects may collide in a 3D virtual environment [Eri05]. For instance, simulating a ball that bounces over a surface requires knowing the exact moment and points of impact between the ball and the surface, as well as the trajectory and speed of the ball for the computation of a *collision response*. The interest in performing such a task can range from basic physics response in interactive applications and video games to accuracy critical applications such as building resistance checking or car crashing simulations.

Despite considerable research effort, collision detection remains a bottleneck for the performance of virtual environment simulations. Indeed, present-day applications include hundreds to thousands of objects of ever-increasing topological complexity. Handling all potential collisions between them in a timely fashion is one of the main challenges to overcome, as algorithms likewise grow in complexity. Consequently, collision detection has been organized as a *pipeline* [Hub95] for over 15 years, from simple filters that determine which objects are potentially colliding, to expensive and precise tests that allow the computation of an accurate response to these collisions. Such an architecture has become quickly mandatory to enable interactive simulations.

Notwithstanding the pipeline architecture and advances in geometry processing algorithms, current progress in processing unit design does not suffice to retain interactive performance with sequential collision detection frameworks, as Moore's Law is not verified anymore for single-core processors. Collision detection research has thus focused onto parallelism in the past few years, bringing about algorithms based on parallel bounding volume traversal tree (BVTT) traversal [TPB08, TMT09], parallel broad-phase algorithms [LHLK10, AGA11, TLW11], or task scheduling solutions [HRF09]. GPUs are also used by state of the art software, either on their own [LMM10, AGA12], or combined with CPUs [KHH+09, HRF+10, PKS10].

On one hand, poor scalability in parallel algorithms arise from overly strong synchronization constraints, and from the difficulty of efficiently extracting work that can be performed in parallel from sequential algorithms. On the other hand, it is expected that computer architectures with up to tens of CPUs become popular in the future, especially in virtual reality centers. In this thesis, we will present a parallel detection collision framework that is thought to be scalable on such architectures.

The SODA framework is intended for interactive use, which implies GPUs being sollicited for high framerate rendering. In SODA, processors independently process a subset of the simulation space, and synchronize locally when a simulated object moves from a processor's area to another. All processors anticipate computations and save object positions to a buffer then read by the rendering thread, which is the second case of synchronization in SODA CD. We will also explain how to manage synchronization and how one can use a spatial subdivision grid to setup the subsets simulated by each core, limit the framework overhead and perform load balancing.

In Chapter 2, we will first describe collision detection environments and algorithms, focusing on

parallel solutions and canvassing existing structures for spatial subdivision. We will also introduce related work in particle physics simulations. Chapter 3 will describe our novel collision detection framework, its synchronization mechanisms and possible methods for load balancing. Then, we will present our implementation and describe the experiments that will be performed to validate the model in Chapter 4, before concluding in Chapter 5.

Chapter 2

State of the Art

In this chapter, we will first present the main characteristics of collision detection tasks, before elaborating on the existing algorithms and their properties. A particular focus will be made on existing parallel algorithms, and on the use of spatial subdivisions in collision detection. Finally, two algorithms from the field of particle physics simulation will be described.

2.1 Properties of Collision Detection Environments

The family of collision detection algorithms seeks to provide computational methods for checking whether 3D representations of objects in a simulated environment are in contact. As simulations can contain multiple objects in motion, collisions must be detected between all pairs of said objects.

Environments Properties Many other aspects of the simulated applications have an impact over the choice of available algorithms, such as performance requirements or types of simulated objects. For instance, *interactive* applications require the scene to be rendered at frequencies that give the illusion of flawless motion. This means that one pass of the collision detection algorithm should never last more than a few tens of milliseconds, regardless of computational peaks that may occur. Reaching interactive performance generally implies either to limit the quality of the 3D models, or to run the simulation on a bigger system. In contrast, in *offline simulations* the time taken by collision detection is not critical.

When a single object is moving and has to be checked against a static environment, one speaks of *2-body* collision detection. Conversely, *n-body* algorithms manage higher amounts of objects in motion, either by performing pairwise collision checks, or by exploiting *spatial consistency* (for instance, sorting coordinates of objects on an axis, and avoiding checks between objects that both don't collide with one in between them on this axis). In the same way, *temporal coherence* consists of reusing in an algorithm data from the previous call as it is likely to be still valid.

Some kinds of objects - especially clothes and fluids - can change shape over time, tear apart or merge when colliding: they are *deformable*, which causes two problems: they may start colliding with other objects because of their deformations, and even collide with themselves. For instance, a piece of cloth that is folding on a surface will have multiple self-contact points, all of which need be handled. In comparison, *rigid bodies* do not require their surface to be checked for deformations. One may refer to Teschner et al. [TKH+05] for a survey of collision detection algorithms adapted to deformable models.

Finally, some algorithms compute collisions at a frequency coupled with that of the 3D rendering engine. They are called *discrete algorithms*, and on each pass, they compute the interpenetration between objects that have collided and apply an appropriate force to each object to correct it. On the other hand, *continuous algorithms* will detect the next time of impact between all objects, and need not be called again before the impacts do occur. Mirtich and Canny [MC95]'s paper is a good introduction to continuous collision detection.

Representation of 3D Object Models There are different possible representations for a 3D model, each of which is suitable for given algorithms. Figure 2.1 shows the four possible representations for 3D models are polygonal models, constructive solid geometry ones, parametric functions and implicit functions. Polygonal models are used in the vast majority of algorithms, as they are simple to understand and easy to process. There are three families of polygonal models: unstructured, convex, and non-convex. Convex polyhedra are the models for which the more algorithms are available. Lin and Gottschalk [LG98] will provide the curious reader with a comprehensive list of 3D models and approaches to collision detection.

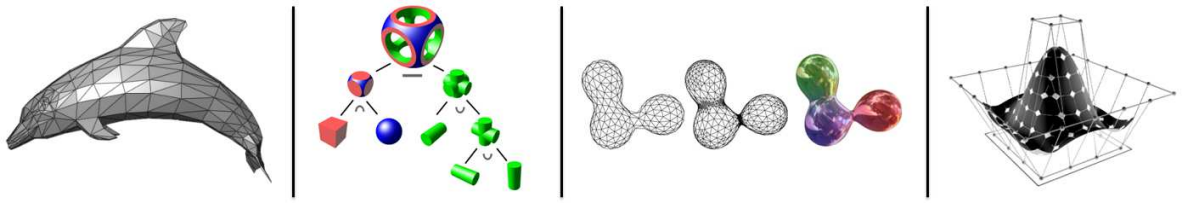


Figure 2.1: The different kinds of 3D model representations of objects (from left to right: polygonal models, CSG models, implicit functions, NURBS) [Avr11].

2.2 Collision Detection Algorithms

In order to improve performance, Hubbard [Hub95] proposed to organize algorithms in a pipeline, with increasing accuracy and complexity at each stage. Algorithms can generally be classified according to their position in the pipeline: the *broad phase* stage identifies pairs of objects that could potentially collide at a very low cost ; the *narrow phase* one finds the areas where contacts may occur on two objects from the broad phase with more complex techniques ; finally, *exact* algorithms compute the contact information for all pairs of actually colliding objects.

In this Section, exact collision detection algorithms will not be discussed. Instead, our attention will focus on some of the most widely used broad-phase and narrow-phase algorithms, that are building blocks of more complex solutions presented in the next Sections. A comprehensive review on all existing algorithms may be found in Avril's PhD thesis [Avr11], and many excellent surveys are available: Kockara et al.'s [KHI+07] focused mostly on narrow-phase algorithms, while Lin and Gottschalk [LG98] focus on exact methods.

2.2.1 Broad-Phase Algorithms

The broad-phase stage, sometimes called *proximity search*, uses approximate representations of models to find out which pairs cannot collide, and to prune them from the list of pairs to be investigated by the next stage. Broad-phase algorithms are truly efficient provided that they are orders of magnitude faster than exact ones, and that they return a reasonable amount of false positives. An obvious approach to broad-phase is called *all-pair tests* and consists of performing pairwise checks on rough approximations of the actual objects, such as axis-aligned bounding boxes (AABBs) [Ber97]. Other families are topological approaches, cinematic approaches and spatial subdivision [KHI+07, Avr11].

Sweep and Prune (SaP) Also called *Sort and Sweep*, this popular algorithm consists of sorting the coordinates of objects among axes, and retaining only those whose coordinates overlap. In a first time, the algorithm updates a bounding volume (usually an AABB) for each object. Then, it projects the coordinates of every bounding volume on each axis, and sorts these coordinates into a list. Whenever the range between two bounding volumes' extremities overlaps on every axis, the pair is labeled as possibly colliding.

Many improvements of SaP exist: with I-Collide [CLMP95], Cohen et al. improved SaP to exploit temporal coherence, by using a matrix representing overlaps, and sorting each moving object locally in the matrix instead of starting over at each time frame. It performs well particularly when few objects are in motion. Tracy et al. [TBW09] further improved it for large-scale environments, by combining it with spatial subdivision (see Subsection 2.2.1). The axis-sorted lists are segmented, so that insertion or removal of an AABB only affects one segment of the list, making updates significantly faster.

Indexing structures and spatial subdivision Indexing structures are height-balanced trees (such as R-Trees [Gut85]) in which each node represents a rectangle of the scene and contains objects lying in this rectangle. A drawback of such structures is that they require frequent updates, from a single leaf of the tree to those at the root, in order to remain balanced. This makes them unpractical to manage in parallel systems.

Similar to indexing structures, spatial subdivisions are also based on spatial coherence: if two objects are far from one another, they have no chance of colliding within a close future. By subdividing the space into separate regions, objects may only collide with those within identical or close areas. possibly colliding objects. Some methods are close in design to spatial databases and use some of the structures discussed above, regardless of the information provided by the environment. Samet's book [Sam90] provides an overview of spatial data structures. The simulated 3D scene is usually represented as a uniform or heterogeneous grid [Ove92], quad-trees, octrees or k-d trees [KMSZ98] (see Figure 2.2). Binary Space Partitioning is also a popular environment-dependant method for spatial subdivision, especially adapted to static environments. Spatial subdivisions will be presented in depth in Section 2.4.

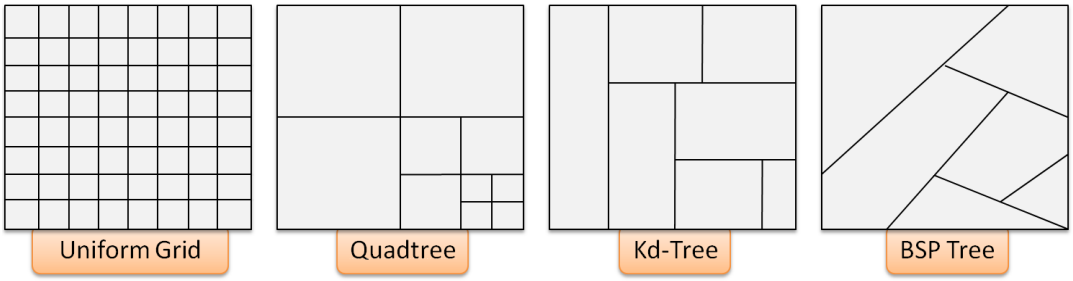


Figure 2.2: The main hierarchical structures used for spatial subdivision [Avr11].

Cinematic methods The last family of approaches is based on the motion of objects to tell which objects may collide. For instance, Vaněček [Van94] uses back-face culling to avoid checking some polygons of objects that may collide: the faces of an object not directly facing another one cannot collide with it and can be exempted from being checked. Another cinematic approach consists of foreseeing the moment when collisions will occur using upper bounds on the velocity or distance of objects [CK86]. Finally, Kim et al. [KGS97] modelled objects as spheres of a same subspace, with an event-driven algorithm that detects collisions and subspaces changes (a common approach in molecular physics simulation).

2.2.2 Narrow-Phase Algorithms

The narrow-phase consists of determining, for all input pairs, whether they have a high likelihood of colliding, and most often what are the possible points of contacts between the objects. Kockara et al. [KHI+07] performed a survey of the main narrow-phase algorithms. In this Section, the four families identified by Kockara et al. will be briefly presented.

Feature-based algorithms These methods directly exploit the geometric features of objects. Lin and Canny [LC91] first made use of polygonal model primitives by tracking the closest features of two polyhedra rather than their closest points for distance calculation. This allowed exploiting temporal

coherence better, as the closest features are less likely to change between successive time frames. V-clip [Mir98] is an improvement over Lin-Canny based on the idea that the closest points of two polyhedra should be within the Voronoï regions of their closest features. It is more robust and also provides support for non-convex polyhedra.

Simplex-based algorithms The GJK algorithm [GJK88] computes the Minkowski difference between the closest points of the polyhedra (using a simplex). If the difference is null, the polyhedra collide. It is at the origin of the simplex-based family of algorithms. A variety of GJK-based algorithms have been proposed for collision detection [GF90, Ber99, Ber01].

Image-Space Methods Algorithms from the image-space family use the special computing capabilities of GPUs (e.g. ray tracing, depth peeling, etc...) in order to identify colliding objects. Cinder [KP03] is a famous algorithm exploiting ray tracing, that casts semi-infinite rays from the viewport of the rendered scene, and counts the number of polygons that each ray passes through. Hermann et al. [HFR08] cast rays directly from within objects and stop when the rays meet a surface. If it belongs to another object, then a collision has been detected. Other image-space based solutions include occlusion queries [GRLM03] or Layered Depth Images.

Bounding Volume Hierarchies Bounding Volume Hierarchies (BVHs) are a very commonly used structure to represent objects in collision detection, by recursively partitioning them. They approximate the geometry of objects with simpler shapes to speed up overlap tests, with a granularity that increases as one digs through the hierarchy. A BVH is made of a root node coarsely representing the whole object, child with nodes that bound a subset of that object. The leaves of a BVH usually bound a few primitives only. The different shapes of BVH offer a trade-off between culling efficiency and storage and computation costs (see Figure 2.3). Teschner et al. [TKH+05] and Kockara et al. [KHI+07] wrote surveys that present BVHs in more details.

Among the most commonly used volumes, from fastest to slowest, the following can be found: *Spheres* [Hub96] are the simplest volume available, and are extremely lightweight. *Axis-Aligned Bounding Boxes* (AABBs) [Ber97] are rectangles aligned on the axes of the environment (which makes them very easy to manage). They provide better culling than spheres, at an acceptable cost. Similarly, *Oriented Bounding Boxes* (OBBs) [Got00] are bounding rectangles, but they can be arbitrarily aligned, which makes them require 15 elementary tests to check for overlapping, against 3 for AABBs. However, they fit objects more tightly.

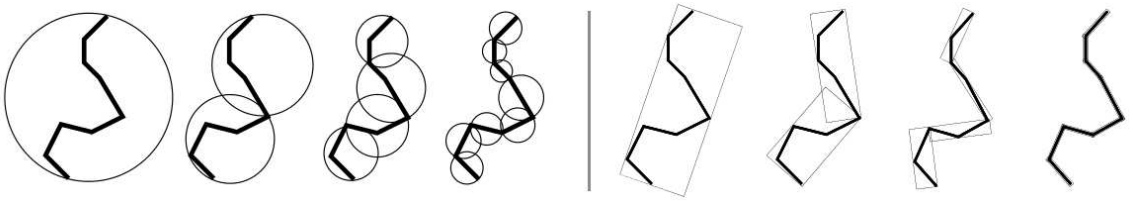


Figure 2.3: Right: examples of BVHs at different granularities using Spheres and OBBs [Got00].

BVH trees can be built in several different ways (top-down, bottom-up or by inserting volumes arbitrarily). Besides, R-Trees, BD-Trees [JP04] for deformable objects, octrees, and many other structures can be used to store trees.

Once BVHs are constructed for two objects, collision detection can be performed by traversing them top-down. Each time two nodes overlap, all their pairs of children nodes need to be tested against one another, until the primitives of objects are reached (then, exact algorithms can be used upon these). At each time frame, BVHs must be updated to reflect the motion of objects, and then traversed again. Another approach is to build a single tree for all objects, and check for collisions between nodes of the same depth. Temporal and spatial coherence can be exploited in BVHs by the means of Generalised

Front Tracking [EL01]. Many other improvements exist, for fast-moving objects, critical-time detection, etc.

2.3 Parallel Collision Detection

2.3.1 Collision Detection and Parallel Computing

Parallel systems have nowadays taken a noteworthy place in the world of collision detection algorithms. Indeed, the evolution of sequential processors faces a technological limitation, as it is presently impossible to manage the heat dissipation problems incurred by transistors smaller than those on current CPUs. Consequently, multiple core processors are now an unavoidable target architecture for all virtual reality applications with significant computing needs. Besides, GPUs have turned in the past few years into many-core processors with blocks of cores sharing global memory (allowing SIMD processing), opening doors for massively parallel computing.

However, the performance gain induced by parallelism is not proportional to the number of processors used. The overheads of data transfer between processors, synchronization required by the algorithm and processor idling due to imperfect load balancing have to be taken into account. If not, adding more processors to a system may not be cost-efficient at all, as the speedup ratio over a single core will start to stagnate. Hence, the new trend in detection collision is to take into consideration processors' architectures, focusing on combinations of CPUs and GPUs, to enable high parallelism with a lower overhead. Oppositely, we will explore in Section 3 the possible performance improvements that can be brought from focusing on loosening needs for synchronization, with the hope of better scalability on generic architectures likely to emerge in a near future.

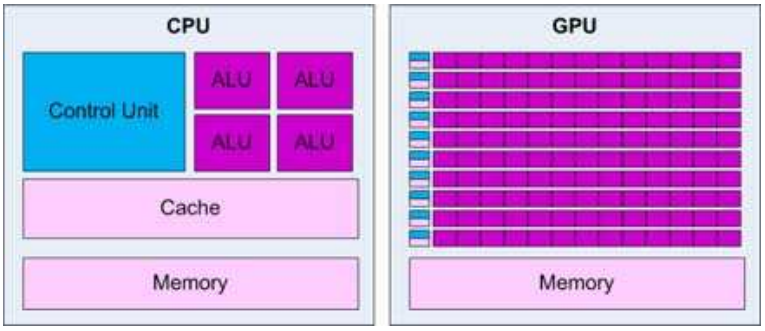


Figure 2.4: Multi-core CPUs share a global cache, while GPUs are organised in blocks of processors with a local cache and high global memory transfer latencies.

There are several ways of achieving parallelism: message-passing architectures allow separate cores to exchange messages and data in order to work together. For example, an hybrid CPU/GPU architecture can be seen as message-passing since GPUs cannot read the CPU memory directly (and vice versa). Shared-memory programming can be performed by several means. It is the natural model for multi-threaded processes running on the same multi-core CPU or GPU block, and it can be achieved with an abstraction layer such as OpenMP or Intel TBB, which are both software libraries for task parallel programming (HMPP being an equivalent for GPGPU). Finally, data parallel programming can be performed on multi-core processors or on Single-Instruction-Multiple-Data (SIMD) processors. It consists of running an algorithm on independent blocks of data on available processors.

Collision detection tasks are not easily parallelized: the work load is irregular, both spatially (non uniform distribution of objects in space), and temporally (many collisions may happen simultaneously), let alone the strong framerate constraints. Parallel algorithms need to be scalable at least for a given architecture, and ideally for any number of processing units, although both message-passing and distributed memories will account for limitations in scalability if algorithms are not built to be distributed: no algorithm has to our knowledge shown close-to-optimal scalability for high numbers of cores and com-

plex objects, albeit providing excellent performance gains on current quad-core architectures coupled with GPUs.

Static partitioning of data among available processors is thus not an appropriate approach to parallel collision detection. Architectural particularities look promising in enabling better performance: shared-memory architectures allow for fast exchange of data and are very adapted to job stealing approaches, but not GPUs, for instance. However, these can perform complex tasks very quickly on a set of data, so they are suited for fine-grained partitioned data.

In this Section, recent advances in parallelization of collision detection frameworks on CPU-based architectures will be presented, along with the aspects of these frameworks that may hinder their potential for scalability. We will introduce, but not insist on, GPU-based and hybrid algorithms in order to present the concepts retained by researchers to educe parallelism on these architectures. Indeed, we did not retain the use of GPUs in SODA CD, as explained in Section 3.1.

2.3.2 State of the Art Parallel Algorithms

Multi-Core Algorithms

Task Dependency Graph and Scheduling Hermann et al. [HRF09] explored the construction of a task dependency graph for the whole simulation pipeline rather than just collision handling, so as to be able to parallelize it without having to synchronize between each stage because of data dependencies. The graph is replayed among several time frames to limit the overhead of this approach, and dynamic loops with conditional breaks are also supported. The approach shows close-to-optimal results in terms of scalability for particle simulation, but about 4 fold speedup for 8 CPUs when simulating heterogeneous environments more representative of virtual reality applications. Hermann et al.'s approach shows that consequent performance gains can be obtained by ensuring that synchronization occurs only when necessary, but also demonstrates that scalability gains are bounded in non-anticipative solutions.

Multi-core Bounding Volume Traversal Tree (BVTT) traversal Many algorithms exist for parallel depth-first tree traversal. However, they may not be adapted to the dynamism of work load between subtrees of a BVTT traversal task. Consequently, specific algorithms were developed for collision detection. Thomaszewski et al. [TPB08] traverse a BVTT in the broad-phase stage of their collision handling algorithm, and acknowledge the need for a dynamic task decomposition. Every couple of BVH root nodes is tested and decomposed into subtasks each time an overlap is detected (one per pair of child nodes). All but one of these subtasks are then pushed on a global stack. Authors then create new tasks executed on new threads for each stacked pair of nodes. They use a programming model that manages thread instantiation cleverly and forbid task decomposition when a task is considered too fine-grained in order to avoid being penalized by the overhead of thread creation. By using temporal coherence to estimate the number of children in the subtrees of a task's nodes, they can decide whether the task is fine-grained enough. They achieve a speedup of about 3.5x on shared-memory systems with four cores, but do not report speedup results on higher amounts of cores.

Tang et al. [TMT09] also perform BVTT traversal, but incrementally, by building and updating the front being traversed in parallel. The authors monitored the run-time of a sequential BVTT traversal algorithm to determine that adjacent triangle pair checks did not need to be parallelized, so these are kept out of the traversal tree using orphan sets and representative triangles [CTM08]. BVH updating is based on a static task decomposition, as the cost of refitting an internal node can be pre-processed. The BVTT is then built in parallel using an cost estimation function for collision and self-collision queries. This allows decomposing tasks finely enough to ensure a balanced load on each core. A limitation to the authors' work is the high memory usage and front size, even though they reached a speedup between 4x and 6x on an octo-core CPU. Running Tang et al.'s algorithm on larger scale, possibly message-passing based architectures might be counterproductive if the front cannot be represented as several partial chunks distributed to the cores accountable for updating these parts. This seems unlikely to be efficiently scalable as BVTTs are hierarchical structures.

Filters for SIMD Processors Tang et al. [TLW11] proposed to improve continuous collision detection algorithms by adding two filtering phases adapted to SIMD CPUs (that can execute the same instructions on multiple data at the same time). Their approach is based on the following theorem: any two volumes that overlap in a 3D space will also overlap in a lower dimension space. Hence, any two primitives that do not overlap in a 1D (resp. 2D) space can be filtered out (*linear filter*, resp. *planar filter*). This technique removes up to 99% of pairs of objects. Despite being very suitable for SIMD processors, and having a speedup between 3 and 6 fold compared to k-DOP BVTT traversal, this approach may start losing efficiency on architectures with high quantities of more traditional cores.

General Purpose Computations on GPUs (GPGPU) for Collision Detection

GPUs are processors with a particular architecture. They are many-core SIMD processors where each core is a block of physical threads sharing a local cache and currently capable of handling up to 64 data pieces at a time. They dispose of hardware mechanisms for task switching within a core when the current one is blocked by a slow memory operation. Global cache access suffers a high latency, and the CPU to GPU bandwidth is limited. Besides, launching a kernel on a GPU core is costly. In order to compensate for these limitations, GPUs are traditionally used for heavy parallel computing, with no need for synchronisation between tasks and limited input data size.

Many algorithms are available on GPU architectures, including popular broad-phase methods [LG07], algorithms for BVH or ray-tracing hierarchy construction, the image-space techniques presented in Subsection 2.2.2, etc. Most GPU algorithms have a sequential base, and GPU capabilities are used for faster massively parallel processing of some parts of the pipeline. For instance, *gProximity* [LMM10] is an algorithm where BVH traversal is performed on OBB and RSS bounding volumes, that are more complex than traditionally used AABBs but also much tigher-fitting. It is a good illustration of the interest of using available computing resources to their fullest.

In the same fashion, Avril [Avr11] massively parallelizes AABB BVH overlap tests as a broad-phase stage, with optimized AABB CPU to GPU transfers. This method is more efficient than CPU-based ones for up to 3000 simulated objects, and uniform spatial subdivision (see Section 2.4.1) is then used to allow interactive performance for tens of thousands of objects, avoiding the combinatorial explosion of this brute-force approach.

Besides, Liu et al. [LHLK10] built up on the idea of Tracy et al. [TBW09] to combine SaP and spatial subdivision, in order to reduce the complexity of the swapping operation. They acknowledged the main problems of parallel SaP: high memory transfers due to the size of the sorted lists, low arithmetic intensity for the sweep part. These are addressed with spatial subdivision. The sorting of lists is done using a radix sort, and the algorithm is improved by using PCA to find the sweep direction that maximizes the variance of object coordinates on the selected direction, which further improves performance three fold over using an arbitrary axis.

Finally, Tang et al [TMLT11] proposed to consider GPUs as stream processors, and this for each step of the collision detection pipeline. The idea is to provide better culling by parallelising every step of the pipeline rather than just one as is usually done. Computation kernels for each stage of the pipeline can easily process the input streams in parallel, and streams can be compacted using one of many available algorithms. They then implement a BVTT traversal with deferred front tracking to reduce the storage size of the front. Making it possible for one GPU to process the next stage of the pipeline before all others managed their work load is vital in providing good scalability, by getting rid of a costly synchronization barrier between stages.

Hybrid Solutions

There exist relatively few solutions combining CPUs and GPUs in the literature, which can be explained by the young age of GPGPU. Some of them assign tasks to PUs based on their affinity, while others specialize each PU and make them work collaboratively. These algorithms usually reach higher scalability since they reduce the processor idling phenomenom by exploiting all kinds of available work units. They

are, along with those presented in Section 2.5, the state of the art of collision detection.

Kim et al. [KHH+09] presented a hybrid method where CPUs specialize in BVH traversal and culling and GPUs on elementary tests (cubic equation solving). Consequently, the main loop of the algorithm is lock-free. BVH traversal is done using task units that are guaranteed to access different nodes, removing the need for locks, and the work load is balanced by the means of dynamic task reassignment. Primitive pairs to be tested are stored in a *triangle index queue* managed by a master thread that sends them in batch to GPUs for computation, using a *GPU task queue* to monitor GPU usage (see Figure 2.5). When there are more segments in the GTQ than GPUs, CPUs are used to help empty it.

The master thread seems to be the weak link of *HPCCD* with regard to scalability, but distributing it over several physical threads is very likely a trivial engineering task. It is interesting to note that *PCCD*, the CPU-only version of Kim et al.'s algorithm, reaches a 7.3 fold speedup on an octo-core system over a single-core one, which is the highest speedup measurement report for 8 cores to the best of our knowledge. Drawing from their experience, we want to investigate other approaches for less synchronized and more independent simulations, as explained in Section 3.1.

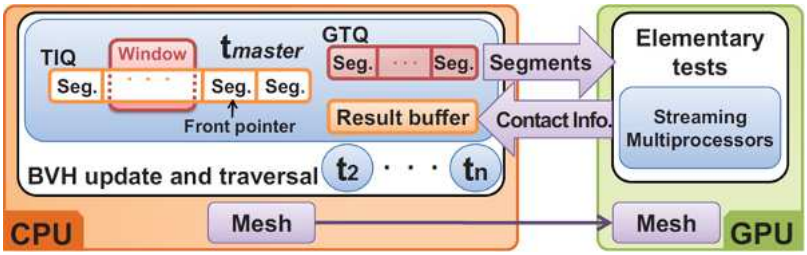


Figure 2.5: The hybrid architecture of *HPCCD*, where a master thread schedules work on the GPUs [KHH+09].

Hermann et al. [HRF+10] improved their previous CPU algorithm by combining GPUs and CPUs and providing a second level of scheduling and improved load balance via a work stealing strategy taking temporal and spatial locality into account (relying on the Owner Compute Rule and reusing of task mapping over frames). Authors developed a Distributed Shared Memory mechanism to simplify memory transfers in their programming model, implemented each task of their pipeline both for CPUs and GPUs, and used a task scheduler to spread tasks among PUs, using *task affinity lists* to improve the data locality of work stealing. Besides, tasks are scheduled on the PU where they will run fastest (run-times being collected for CPUs and GPUs during a *warming phase*). A noteworthy aspect of this algorithm is the apparition of *cooperative speedups* where the speedup of an hybrid system is larger than the sum of speedups of the equivalent CPU and GPU systems.

In an opposite direction, Pabst et al. [PKS10] focused on developing a high parallel spatial subdivision method, used as a broad-phase prior to GPU-optimized exact collision detection. Scalability is ensured by the absence of communication between GPUs during the computationally intensive parts of the algorithm. Spatial hashing (see Section 2.4) is used to partition the space into a grid, where each object belongs to a single cell. Then, detection collision is only performed between objects within the same and adjacent cells. It is thus possible to calculate the number of collision tests to run, to map each of these tests to a GPU thread and to perform SIMD computing on GPUs to detect actual collisions. This spatial hashing technique reduces the number of tests to perform by more than 95% both for discrete and continuous collision detection. A master/slave approach is used for GPU computing, where some parts of the pipeline are run on a master GPU, which distributes data to compute to slave GPUs for the parallel parts. Although Pabst et al. use spatial subdivision to increase the scale of simulations their algorithm manages, its master-slave architecture and synchronization mechanisms make it inherently not scalability-friendly.

2.4 Spatial Subdivision for Parallel Collision Detection

Spatial subdivision is the process of decomposing the space in which the simulation takes place into subsets, and assigning every object or object primitive to the subset in which it lies. We have seen from previous algorithms that it can play a crucial role both in load balancing of collision detection tasks, but also in enabling larger scale use of algorithms, be it brute-force approaches. Spatial subdivision algorithms can be classified based on the type of spatial data structure used and on the method that maps objects to the chosen structure. There are two families of spatial subdivisions: those based on hierarchical structures (such as those in Figure 2.2), and grids with cells of uniform sizes. The latter have the particularity of being object-independent, which may turn in useful when manipulating deformable objects, but may be suboptimal in simulations with a high object size variance. The main issues in spatial subdivision for collision detection are update of hierarchical structures and optimal uniform grid cell size [TKH+05].

The first occurrence of spatial subdivision in physics simulation is for molecular dynamics simulations, where Levinthal [Lev66] accelerated neighbourhood queries in spatial databases, using a uniform 3D grid. Overmars [Ove92] proposed a hashing-based spatial data structure to store objects in disjoint set of boxes adapted to their size. We will now review uses of spatial subdivision closer to our field of research.

2.4.1 Uniform Spatial Subdivision

Turk [Tur89] used a uniform 3D grid to subdivide space in a molecular collision detection application. Every cell of the grid contained pointers to any spheres partly or fully present in the cell, and molecules were tested for collision on every other one lying within the same cells. In a similar fashion, Lawlor and Kalé [LK02] map the bounding volumes of objects to a uniform grid of axis-aligned voxels. They acknowledge the importance of not repeating collision checks for two large objects mapped into several identical voxels. Voxel size is chosen so that most objects fit within a voxel.

Teschner et al. [THM+03] modelled deformable objects with tetrahedral meshes, and used a spatial hashing function to assign tetrahedrons to a 1D hash table depending on their coordinates. They also map vertices of all objects to axis-aligned 3D cells, and map the cells to the same hash table. This is an efficient way of avoiding duplicate work, since vertices will belong to a single cell, and tetrahedrons will thus not be checked for collision against the same vertex twice, even if present in several cells. Authors also determined that the cell size is the parameter that most influences performance. Liu et al. [LHLK10] used spatial subdivision to partition objects before sorting lists in SaP, cutting the space in planes parallel to the direction found using PCA (see Subsection 2.3.2). Within obtained cells, authors perform a second 2D spatial subdivision to further improve SaP performance.

Le Grand [LG07] presented a parallel GPU implementation for spatial subdivision using CUDA. Le Grand calls *home cells* those where the centroid of bounding volumes lie, and *phantom cells* the other ones. By setting the cell size to be at least twice the radius of the largest object, he avoids collision checks between two objects lying in the same phantom cell. This allows significantly reducing inter-cell collision checks.

2.4.2 Hierarchical Approaches

Band and Thalmann [BT95] presented a model combining a uniform 3D grid and an octree. The grid is made of 2^4 voxels per axis (said to be *level 4*), and objects are placed inside it using a process called *digitisation*. Whenever a voxel contains several object faces, these are digitized to the next level grid (thus dividing the voxel into 8 subcells). This step is repeated until every voxel contains only one face, or until a user-set threshold is reached (in which case an elementary test between faces is performed).

Mirtich [Mir97] built up on Overmars' works, by creating a hierarchy of cells of various sizes, linked to a spatial hashing function. Objects are mapped to the cell with an identical hash bucket that best fits their size. A 1D example is given in Figure 2.6. An object A must be checked for collisions against all objects within cells of lower or equal resolution than the one containing it. Processing cells with a

higher resolution is unnecessary since any objects within these will apply the same procedure. Mirtich then proposes storing the object also in lower resolution cells matching its hash bucket, and maintaining a list of pairs of close objects on which to perform collision detection. Reinhard et al. [RSH00] use a hierarchical octree in which objects also belong to a single cell. They determine the most appropriate resolution by dividing the length of the grid's diagonal by the one of the object's diagonal.

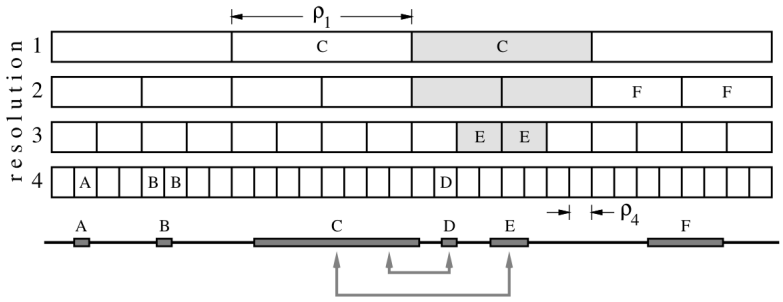


Figure 2.6: A 1D example of a hierarchical spatial hash table. The 1D objects are labelled A to F. Objects are stored in the hash table at the resolution that best fits their size. The shaded cells are those containing objects close to E [Mir97].

Melax [Mel00] developed a BSP tree traversal method for static environments and with few moving objects. The BSP tree leaf nodes are convex cells that are solid if they overlap with an object of the static environment, or empty otherwise. Each tree node contains a plane to determine in which subtree a given object is located. Thus, collision detection consists of traversing the tree with the moving object, and of checking whether it reaches a solid node. This form of spatial subdivision consists of directing moving objects towards the area where they lie, in order to eliminate the need for collision checks with the whole environment.

2.5 Related Work in Particle Simulation

Molecular physics simulations are different from traditional ones in the sense that the simulated objects do not have a complex topology but can always be represented by points or spheres. This means that both detection algorithms and collision response equations to solve are significantly simpler. This fact allowed molecular physics simulation researchers to design algorithms that run on systems that have orders of magnitude more processors than in the collision detection research field. Moreover, complex objects have higher memory storage requirements, and their topology cannot easily be transferred on-the-fly from a process to another if they don't share a memory from the start. This imposes an additional limit on the scalability of collision detection algorithm.

In this Section, we will present two papers on molecular physics simulation that are similar to the approach followed in SODA CD, and develop on the challenges that exist in utilizing such algorithms for collision detection between objects of complex and heterogeneous topologies.

Parallelism using a Uniform Grid with Dynamic Load Balancing Miller and Luding [ML04] developed a system for parallel molecular dynamics situation that is event-driven and similar in design to SODA CD. Event-driven algorithms sort future events in a priority queue that allows knowing which event is the next to be processed, in the form of a heap tree. A uniform grid is used to distribute the work load among processes, which are all assigned an area of contiguous cells. Authors redistribute cells between processes as a means of load balancing.

Processes have their own local clock, and when an interaction occurs with a neighbor process, the state of particles in the current process is rolled back to that of the time given by the neighbor process' clock, if earlier. The authors acknowledge the fact that this rollback algorithm is not infinitely scalable, however, and that if a simulation contains a chain of particles crossing over all areas, then this simulation

would run sequentially. We will propose in SODA CD a solution to partially limit the effects of a high amount of interactions between neighbor processes.

As we earlier stated, objects in molecular physics simulations are located by a position and do not have any particular topology. In our case, a single object can be made of tens of thousands of polygons, and collision checks between two such objects involve many more computations than the solving of a single equation. Consequently, the load balancing metrics used by Miller and Luding (the number of particles per area) is not sufficient for use in traditional collision detection frameworks. Other mechanisms for load balancing have to be developed.

Distributed Simulation and Optimistic Computing Li et al. [LJS+07] proposed to use speculative computation on a parallel event-driven particle collision system. Particles have a local event priority queue, and space is divided into blocks of cells. Each block keeps its own event queue made of the root events of local queues of the particles it contains. A global scheduler thread then keeps the root event of all block queues, and is used to let every process know up until when it can compute events in its block before risking a violation of causal order.

Besides, Li et al. propose to use speculative computing: each block has an additional speculative computing thread that keeps computing future particle states in order to avoid processor idling. When the global event queue does not contain any event prior to the currently speculated state of a particle, this state can be directly adopted as the current particle state. An algorithm determines how much further should speculative threads be allowed to go. This solution is preferred to storing particle states after each computation, although such a choice would not be available with discrete collision detection algorithms.

Chapter 3

Scalable, Distributed and Anticipative Collision Detection

In this chapter, our contribution is presented, starting in Section 3.1 from the hypotheses that we retained and that influenced our design choices, followed by the details of the implementation in Section 3.2. Finally, Section 3.3 presents possible means for load balancing in our model.

3.1 Positioning

The high computational irregulary in time and space of typical applications using collision detection, and the strong framerate constraints of such applications make customized algorithms preferrable to ready-to-use code parallelization or distribution solutions (as seen in the direction taken by recent research). As seen above, the most performant CPU-based algorithms are not yet fully scalable, and the best results (up to 7.3 [KHeY08] speedup over a single core) suggest that synchronization must be trimmed down to the strict necessary to further improve scalability. In fact, we could only find two pieces of research presenting scalability results for 12 to 16 core computers [TMT09, HRF09].

Asanovic et al. [ABC+06] advise researchers to design systems that scale well on any number of cores. We do not expect that algorithms properly scaling on current and on thousand-core computers be identically designed, however, considering the specific constraints of collision detection that we described before. This can already be observed between GPUs that are vector processors in need of fine-grained task decomposition to can process data in parallel, and CPUs that allow more flexible workflows.

We make the assumption that future architectures for virtual reality systems will need to use larger scale computing systems in order to keep up with growing application needs. We also acknowledge that many applications require very fast communication between computing units, so that the rendering framerate of these applications can be kept high enough for human interaction. These two constraints lead us to expect that efficient collision detection frameworks will be needed on architectures with tens of central processing units in a near future. Hence, the framework we present is designed to maintain a decent scalability on such computers.

We also want our framework to be independent from GPUs, because they may be considerably sollicited for rendering in some typical virtual reality applications, again with the constraint of interactive framerates. Besides, the General-Purpose computation on GPUs programming model requires a master CPU per GPU, which incurs synchronisation for access to the GPU resource. Finally, the architecture of GPUs makes them efficient only for massively parallel computing of data that holds in restricted memory with high CPU↔GPU transfer latencies. Consequently, we suspect that GPUs may be a hindrance for scalability on architectures with many more CPUs than GPUs. We want to leave inclusion of GPUs to further research, possibly after more flexible programming models are available for general purpose computation on GPUs.

Besides, we noticed that several algorithms were not capable of making full use of the available computing resources, having some processing units idling while others were finishing their computations. Even though we are not in a position to quantify such a loss, the molecular physics community addressed it by the means of speculative computing [ML04, LJS+07], which seems to be an interesting lead to follow. Hence, we chose SODA CD to be designed in a way that inherently permitted anticipative computations, while proposing clever solutions to the problem of necessary rollbacks described by Miller and Luding.

3.2 The SODA CD Framework

In this Section, we present SODA CD, a collision detection framework that is distributed among independant processors, that performs anticipative computation, and that is designed with high scalability in mind.

Unlike in previous works where a synchronization bareer is kept between each iteration of the collision detection pipeline, we propose a framework where each core is responsible for simulating a subset (referred to as a *world*) of the entire simulation, defined by a set of adjacent cells in a spatial subdivision grid (named *territory*), and who communicates exclusively with cores that manage territories adjacent to his own one (called *neighbors*, while the planes between distinct territories are called *borders*). In order to avoid processor idling - that occurs as a consequence to imperfect load balance between the different cores, each world precomputes the next simulation steps of its territory and saves them to a buffer.

We propose new synchronization constraints that suffice to guarantee determinism of the simulation in this model: one of them is linked to the arrival of an object in a territory (either through user input or from a neighbor territory), and the other relates to updating object positions in a 3D engine used for rendering. Synchronization is always performed at the oldest time step of the involved processing units, and all the precomputed time steps that follow are invalidated, until the changes involved by the synchronization are progressively propagated into these future time steps, using the spatial subdivision grid.

The framework is independant of the narrow-phase and exact collision detection algorithms used, while the broad-phase algorithm should make use of the spatial subdivision grid for better efficiency. In this first version of the framework, we only consider architectures where all cores share a unified memory, in which case no cache synchronization mechanisms have to be considered for object positions and transformations (especially for deformable models).

We will first explain territory management, and continue with a description of the worlds, and of input/output in SODA CD. Then, we will cover the initialization of a simulation in our framework. Finally, theoretical methods for load balancing are proposed, as well as some properties that must be held in order to maintain proper balance and stability of the framework.

3.2.1 Territories and Borders

From Spatial Subdivision to Territories

We define in our framework a spatial subdivision uniform grid. Uniform grids are interesting in that they offer the following properties:

- since uniform grid cells have the same dimensions, they can be described with only a set of 3 coordinates and an owner world identifier. This makes it practical for exchanging cell information between worlds (during synchronization or load balancing for instance).
- unlike regular trees, uniform grids do not require any topology update (which would be particularly costly with non-uniform dynamic spatial subdivisions, as updating them may involve communication between all processing units if the root node's children are modified by the update).

- updating the position of any given object can be done by the world managing it without global knowledge of the simulation space distribution.

Worlds are responsible of a territory made of adjacent spatial subdivision cells, in order to exploit spatial consistency to reduce the task scheduling overhead of current parallel spatial subdivision solutions. Indeed, most objects transiting from a cell to another will stay within the same territory, which means the object location update involves only the core that manages this territory. The overall overhead of this parallel solution will be lower than the one of concurrent work provided that the world to world synchronization and load balancing overheads are kept minimal (see Sections 3.2.1 and 3.3). Besides, each world only needs a memory representation of the subset of the grid that they own, with a margin of one cell that will allow them to know which neighbors they have. This is important for scalability as it allows very large scale simulations where the whole scene and objects could not be kept within the memory of a single computer (as long as all different 3D meshes of simulated objects are available to all computers for rendering). Territories are not necessarily rectangle-shaped as cells can be exchanged between worlds as a load balancing mechanism. Hence, in each world's local grid, cells are marked with a unique world identifier, so that each world knows which cells it actually manages.

Optimal Cell Dimensions Different criterias can be used and combined to choose the dimensions of each cell, although there is no optimal solution to this problem [LG07]. SODA CD is not sensitive to the method used to choose cell dimensions, although our implementation works in the following way: cells cannot be smaller than the dimensions of the biggest object in the simulation. This ensures that objects can only collide with those located in their own or in directly adjacent cells [PKS10]. Another possible constraint for minimal cell dimensions is to ensure a minimal density of n objects per cell. n may be simulation-dependant, and an implementation could allow tuning of this parameter, or even offline parameter optimization. The gains of including density criteria for cell size will be studied in future work.

Use of hierarchical grids In order to maintain efficiency of the grid in presence of very large objects, one can setup a hierarchical grid [Mir97], using the following function to determine its depth: $depth = \lfloor log_2(biggest\ to\ smallest\ object\ size\ ratio) \rfloor + 1$. This would guarantee that the number of different resolutions is kept low, and hierarchical grids are necessary to be able to efficiently run simulations with a high disparity in object sizes. However, the use of hierarchical grids significantly complexifies the implementation of border collisions and synchronization mechanisms presented thereafter, and was thus left for further work.

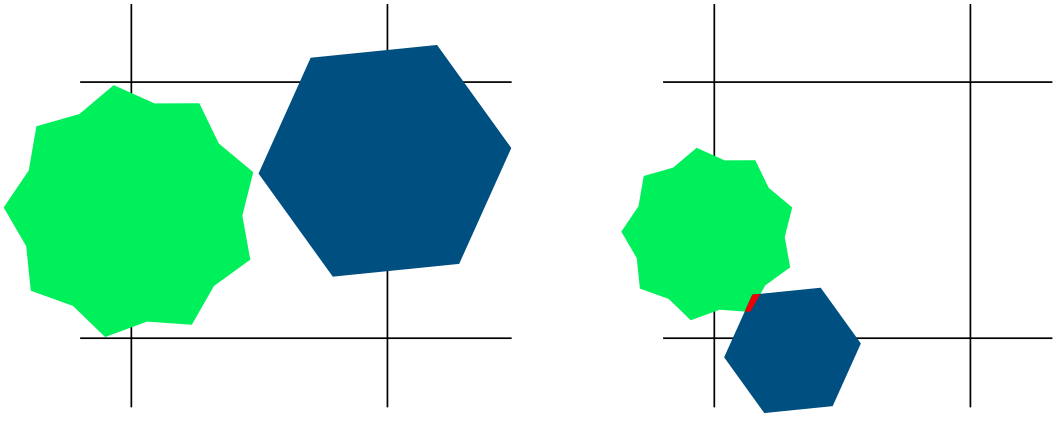


Figure 3.1: On the left, objects from non-adjacent cells cannot collide because of the minimal size constraint on cells. On the right, objects from adjacent cells collide, even though the point of contact is located in neither cell.

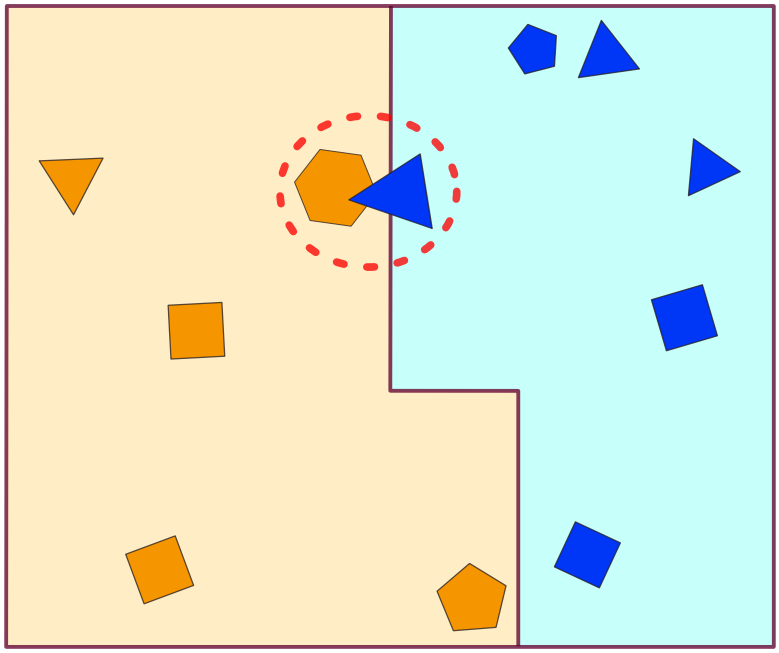


Figure 3.2: An object from the blue world crosses the border towards the orange world, causing the need for synchronization between both.

Border Traversal Detection

Collisions may occur between two objects located within adjacent cells, each at a side of the border of their respective worlds. If these collisions are left unmanaged, then some contact responses will not be applied, and the simulation will no longer be deterministic. However, worlds would need to be constantly synchronized if we cannot tell whether objects located at borders overlap with cells from another world or not, thus preventing anticipated simulation. For this reason, we only synchronize between two worlds if one object from a border cell of these worlds overlaps with the border.

Hence, borders between two territories are represented by static physics objects and normal detection collision is used to detect border traversals, although no collision response is applied with territory border objects. The synchronization mechanism is explained in Section 3.2.2.

Cost of the Detection: The worst possible setup for our framework is when territories are made of a single cell. In such a case, every object will have to be tested for collision with the 6 borders of its containing cell. In the worst case, all objects will collide with 3 of these borders (i.e. they will lie in the corner of their cell), which means that they will overlap in up to 7 cells belonging to distinct worlds, which brings up to 9 more borders with which they may be colliding (in total, 8 worlds will have to synchronize for each object, and if enough worlds contain an object, then the whole simulation will be synchronous and without anticipation). The sum of additional collision checks to perform is thus 15 per object, which remains of linear complexity in number of simulated objects.

Nevertheless, we argue that in practice such a situation is unlikely to arise, and that specific algorithms can be used for collision checks between territory borders and real objects (such as axis-aligned-bounding-box (AABB) vs. plane overlap, followed by vertice-face, edge-face or face-face exact detection algorithms), and that the cost of border traversal detection is likely to be low enough for practical efficiency.

Besides, consider the example of perfectly square territories made of 3x3x3 cells: the average number of borders per cell would be 0.5, which means that on average each object would have to perform 0.5 border detection checks rather than 6 to begin with.

3.2.2 Autonomous Worlds

Independent Physics Engines and Transform Buffers

Worlds in SODA CD are physics engines that run on a single core (Bullet engines in our implementation). They are responsible for detecting and processing collisions of a subset of the simulation being run. Each engine uses its own local clock and performs simulation steps independently from others. The results of a simulation step are made available to other threads as a list of transforms (i.e. the movement and rotation to apply to an object) and records of linear and angular velocy for each object simulated in the world's territory, associated to a timestamp (the value of the world's local clock once the simulation step was over). The lists are saved to a circular buffer, which can then be read to retrieve the positions of objects at each step.

The use of a *circular transform buffer* offers a trade-off between the desired robustness of the application and the memory size needed to store the buffers. Indeed, a large buffer enables the precomputation of a longer time span, which can help absorb the performance impact of a computational peak due to many simultaneous collisions, but the size of the buffer must be bounded depending on the amount of memory available on the system where SODA CD runs. Besides, the most important the anticipation is, the most likely it is to be invalidated by further synchronizations, up until a point where anticipative computation is pointless [ML04].

A rendering thread may read entries from the buffer in order to enable user feedback, as explained in Section 3.2.3. An entry will be removed from a buffer by the reading thread provided that it read another entry that is further in time.

Synchronization With Neighbors

Determinism is guaranteed in traditional parallel collision detection frameworks by making sure that all collision responses that occured before a time step t have been applied prior to detecting collisions at time step $t+1$ (i.e. by forbidding anticipation). We need to redefine this postulate in our application if we want all world simulations to run at their own pace.

Consequently, we must detect the penetration of a foreign object within each world's territory. Such an object will be associated with a timestamp indicating at which moment it starts to exist in the penetrated territory. From this timestamp, it must be included in all collision detections within the world. A first problem may arise from this: an object may be integrated within a world at a time preceding the one of the world's local clock, thus invalidating all circular transform buffer entries between the object's timestamp and the world's current clock value. We present in Section 3.2.2 a rollback algorithm to manage this problem.

In any case, objects that move from a world to another will coexist in both these worlds for some time. They may thus undergo collisions in both worlds at the same time, which means the collision response phase of these worlds' collision detection pipelines must be synchronized. It is acceptable that collision detection itself is not synchronized, since the detection algorithm will produce sets of inter-connected objects (called *simulation islands* in the physics engine our implementation uses) for which a collision response must be computed, on both worlds.

The collision response is then computed for each of these islands. A synchronized physics engine simulation step function thus only needs to merge simulation islands of synchronized worlds, schedule the response computation for each island on one of the involved worlds, and then share the results to all worlds so that they can update their local view of the objects.

Algorithm 1 presents our novel, synchronization-enabled, Bullet collision detection and response pipeline in pseudo-code. It detects border collisions and synchronizes with corresponding worlds, and also reads from a synchronization queue the queries made by other worlds for synchronization at a given time step. This queue contains the border collisions that trigerred the query, as well as the identity of all involved worlds for these collisions.

When a notification is received, the notified world will add the attached information to its synchronization queue, and it will restart the current simulation step if it matches the one of the notification. If the timestamp of the notification corresponds to a past simulation step, the Rollback and Propagate algorithm is used.

Algorithm 1: The new Collision Detection pipeline with synchronization on object border crossing. Collision detection is performed between objects and borders, and in case of collisions, the involved neighbors are identified and synchronized with. Synchronization also occurs with worlds that notified the need for synchronization due to border collisions in their own territory. Next, simulation islands are locally computed, and then merged and distributed among involved worlds for constraint solving. Once the constraints are solved, worlds exchange the position and speed of all border-crossing objects.

```
Input: TimeStep t
Variables:
List of ContactInfo borderCols;
List of Worlds neighbors;
List of SimulationIslands islands;
boolean syncNeeded = ⊥

Code:
interObjectCollisionDetection();
borderCols = borderCollisionDetection();

if borderCols not empty then
    neighbors = findAndNotifyNeighbors(borderCols);
    syncNeeded = ⊤;
end

if syncQueue contains entry for t then
    borderCols = borderCols ∪ syncQueue.getBorderCollisions(t);
    neighbors = neighbors ∪ syncQueue.getWorldsToSyncWith(t);
    syncNeeded = ⊤;
end

if syncNeeded then
    waitForNeighbors(neighbors, t);

    foreach World n ∈ neighbors do
        exchangeBorderCollisions(n, borderCols, t);
    end
    foreach ContactInfo info ∈ borderCols do
        detectCollisionsAround(info.getObject());
    end

    islands = calculateSimulationIslands();
    mergeAndScheduleIslands(neighbors, islands, t);
    foreach island i ∈ islands do
        if i managed by this world then
            solveIslandConstraints(i);
        end
    end
    synchronizeSolvedIslands(neighbors, t);
    solveConstraints();
else
    calculateSimulationIslands();
    solveConstraints();
end
```

A world that is very ahead of others may remain stuck in the waitForNeighbors(...) function waiting

for the slowest world (the one with no anticipation), should it be its neighbor. In fact, in future versions of the framework, a world waiting for neighbors would use its processing time to compute better load balancing between it and its neighbors, and would then take over some of its neighbors' territory as explained in Section 3.3. The Rollback and Propagate algorithm presented in Section 3.2.2 will be used to compute the simulation in the ahead world's new cells up until where it needed to synchronize.

The worst case of synchronization is a simulation where all objects are constantly in contact with each other, in which case all worlds will be synchronizaed and thus anticipation unavailable. The SODA CD framework would then be equivalent to a classical parallel spatial subdivision with a potentially important synchronization overhead. We yet have to evaluate the exact cost of our framework in such a situation, compared to existing spatial subdivision implementations such as Le Grand's [LG07]. Our framework might still present an interest in the case of message-passing based implementations, however, since the cost of sending messages to schedule all collision checks may be more important than the one of scheduling only constraint solving on simulation islands.

Modifying Anticipated Entries

A direct consequence of synchronization between two worlds w_1 and w_2 is that the fastest of these two worlds (the one whose local clock is most advanced) has to drop some of its anticipated computations from its circular transform buffer. In order to limit the incurred losses, we propose a method based on the spatial subdivision grid to integrate modifications that a world's objects undergo due to synchronization into previously computed buffer entries, called *Rollback and Propagate* (see Algorithm 2).

Algorithm 2: The *Rollback and Propagate* algorithm of a world object, in pseudo-code. The algorithm is presented void of synchronization mechanisms for easier understanding. rollbackAt() prevents other threads from reading the buffer starting at the given time, while propagateChanges() makes the entry readable again after modifying it.

```
Input: Stack of Objects L, TimeStep t
Variables:
List of Objects temp;
List of Objects nowColliding;
CircularTransformBufferEntry newEntry;
Code:
 if LocalClock() ≥ t then
    m_Buffer.rollbackAt(t);
    while t < LocalClock() do
        while L not empty do
            Object o = L.pop();
            List temp = detectCollisionsAround(o);
            nowColliding = nowColliding ∩ temp;
        end
        resolveConstraints(nowColliding);
        foreach Object o ∈ nowColliding do
            newEntry.insert(o.getPosition());
        end
        m_Buffer.propagateChanges(t, newEntry);
        L = nowColliding;
        nowColliding = {};
        t = t+1;
    end
end
```

The *propagateChanges*() function must not only update the transforms of an entry before making it readable again. It should also, for each object changed in the entry, notify worlds that own surrounding

cells whether that object overlaps with these cells, based on the result of the collision detection that was just performed. Indeed, neighbors must be notified of new overlappings to rollback and integrate the changes, but also of objects that do not overlap anymore, in order to avoid false positives in future simulation steps.

Although it is hard to quantify the gains of the rollback integration algorithm over simply invalidating buffer entries and running the collision detection pipeline again, one may expect performance improvements for any simulation which has a low density or irregular distribution of objects.

3.2.3 Management of Input and Output

Rendering

High frequency rendering of the simulated scene is a requirement for interactive applications. This task is performed by a 3D engine that keeps its own track of object positions and orientations. In normal applications, the 3D engine and physics engine object information would be synchronized after each physics simulation step.

In our case, there are several physics engines, each of which is allowed to be several seconds ahead in the simulation. We setup a thread responsible for rendering, that manually reads the CircularTransformBuffer of all physics engines to retrieve object positions prior to rendering one frame. This thread works with a fixed timer allowing a smooth framerate, and uses the closest in time step available in CircularTransformBuffers. If the simulation is extremely slow, then the same position information will be used, but the interface will still allow user interaction at a high framerate.

As explained earlier, the rendering thread is in charge of removing entries from CircularTransformBuffers once they are no longer needed (i.e. once the rendering thread has read the next entry in time in the buffer). In order to maintain the desired framerate, it is necessary to limit the time overhead of this position updating step. If too many objects are being simulated for one thread alone to provide the desired level of performance, CircularTransformBuffer reading can be spread on several threads, provided that the 3D engine is thread-safe.

The cost of reading positions from a buffer and writing to the 3D engine's memory is linear in number of objects, and does not suffer from any computational outbursts like collision detection itself. On message passing architectures where the size of the data transferred is critical, implementations can use lesser precision coordinates so that each object position and orientation information fits in eight integers, while on shared memory architectures, simple pointers to data can be exchanged. Besides, performance issues at rendering level will be contained in the threads allocated for rendering and will not impact the simulation itself, as data flows only from the physics worlds to the rendering threads.

Discrete vs. Continuous Algorithms for Circular Transform Buffers Whether the physics engine in use provides continuous or discrete detection algorithms has an important impact over the management of the circular transform buffer, and the best option is dependant on the system's hardware: a discrete algorithm will simulate at least 24 steps per second to provide interactive performance. Consequently, it is important to limit the storage size of the transform and velocities of an object. One possible solution is to limit the accuracy of the physics engine by storing position, orientation and velocities of objects on integers rather than on floats. Binary shift operations can be used to provide cheap conversion from the physics engine API's expected values to those stored in the buffers.

On the contrary, continuous algorithms simulate bigger time lengths per step. However, the rendering thread will still need to have object transforms for as many different time stamps as would be available with discrete algorithms. Linear interpolation between last known past and next known future transforms can be used to solve this problem, at the cost of some processor time to compute the interpolation. Nevertheless, the thread that performs rendering reads the buffer at different moments than when it may use GPU operations for actual rendering. Hence, linear interpolation might be implemented using GPUs, although it is unclear to us whether the CPU to GPU transfer times would thwart the speed gains from using GPU processors rather than a single CPU core.

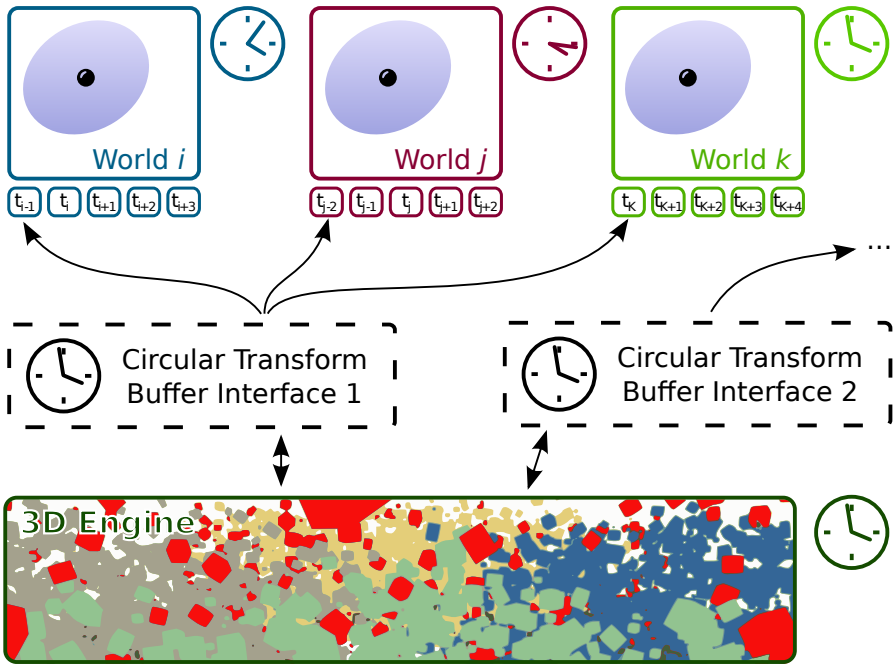


Figure 3.3: Global architecture of SODA CD. Worlds i, j and k's buffers are read by Interface 1, and the 3D Engine's thread reads object transforms from Interfaces 1 and 2.

In conclusion, discrete collision detection algorithms are a sound choice for architectures with large amounts of memory, whereas continuous algorithms are more adapted to systems with high processing power and low memory in SODA CD.

User input

Users may interact with an application by adding, removing, or modifying the trajectory of an object. Either way, the modifications that occurred will cause the invalidation of the anticipated computations in a world, and the summoning of the *Rollback and Propagate* algorithm. It is not possible to avoid the overhead of integrating user interactions in the anticipated computations, but it is still more efficient than not anticipating at all.

3.2.4 Simulation Setup

A critical point in showing that SODA CD is viable is its ability to setup a virtual reality application's simulation space, so that the algorithms discussed above can be used for this simulation. Creating a simulation consists of setting up a basic static environment and loading the objects to be simulated. The framework then automatically computes uniform grid parameters, runs a clustering algorithm on the entities, and then starts creating the worlds and their local grids.

Clustering algorithm: The algorithm used should be able to produce equally weighted clusters (either unweighted with an equal size, in which case it can be used directly on object positions, or weighted, with the weight being the number of objects per cell ; the latter solution has the advantage of rendering unnecessary the following steps of cell owner assignment and empty cell assignment). Ideally, the clustering algorithm should be runnable in parallel. However, our SODA CD implementation currently only ships a sequential equal-sized EKMeans algorithm.

Local grids: Each world owns its own grid that will contain all objects within its managed simulation space, and that grid will retain minimal information about its neighbors, in order to avoid memory

overheads. Moreover, we want to limit the knowledge a world has of its neighbors to an extent where all new information will be provided by current, known neighbors, rather than a centralized thread or an arbitrary number of unknown other worlds.

The size with which local grids are created corresponds to the one necessary to contain their whole cluster's points, plus a margin of one cell per direction that will allow them to know their direct neighbors. Once local grids are created, it may be that some cells overlap because two objects from different clusters have the same coordinates in cell space. Hence, we sort objects per cell coordinates and browse them to determine for each cell which world owns the most objects. This world takes full ownership of the cell and of its objects, and all other worlds are notified that it owns the cell at the given coordinates (they can thus update their margin cell information if they are neighbor with this world).

Other steps are performed, using a clustering algorithm again, to assign each empty cell to a world and to finish notifying ownership of every cell to every world.

Performance bottlenecks: Several operations are currently performed sequentially, with a linear complexity in number of objects. Albeit one could argue that it is acceptable for an application to require some time to start, we consider it is important for scalability that the simulation setup process is improved in future work. For instance, it is common for physics or 3D engines to propose the serialization of data that has to be precomputed before launching a simulation (e.g. a scene bounding volume hierarchy). The same approach could be used to save optimal clusters for a given simulation and runtime system.

3.3 Load Balancing

As we previously explained, the simulation space is divided into several *territories* made of contiguous *cells*. Each territory is simulated by a unique *world*, and the objects that it contains can *transit* from it onto another territory. Load balancing is performed by the means of giving or stealing parts of territory, in a similar fashion to broadly-known task stealing and giving mechanisms in task-parallel programming models.

Many criteria can be used to determine which territory from which worlds must be given to which neighbors (see Section 3.3.5). Coherence of the territories must also be taken into account so that no world obtains a split or thin territory, but rather that all territories stay as compact as possible (e.g. by minimizing their perimeter [ML04]). Even though it is planned that future versions of SODA CD make use of a hierarchical grid, load balancing is always performed on a flat uniform grid.

When using hierarchical grids, an ideal depth is computed and the associated grid used to reason on. To choose the optimal grid, the retained criterion is that the size of one cell should be as close as possible to a range between one and five percent of the total average territory size, in order to allow fine-grained load balancing.

3.3.1 Load Balancing Metrics

Load Metrics for Continuous Algorithms

The metrics used to evaluate the load of each core can be defined as a combination of several parameters, when a continuous collision detection algorithm is used. We will first define values associated with each world p_i:

TOI the next Time Of Impact that tells when to relaunch the detection algorithm

CDT the time spent computing the next collision detection step to be used by the renderer

Several levels of load may be defined for each world, using a ratio of the time spent computing compared to the one that is available.

Underloaded	when the world spends more time idling than its neighbors (ie. the $\frac{CDT}{TOI}$ ratio is lower than the average by more than a given threshold)
Normal load	occurs when the world's $\frac{CDT}{TOI}$ ratio is close to the average
High load	when the world's $\frac{CDT}{TOI}$ ratio is below one but higher than the average by more than a threshold
Overloaded	happens if the world cannot cope with the requirements of interactive simulation, and has a $\frac{CDT}{TOI}$ ratio above one.

On The Variance of CDT and TOI One should note that the CDT metrics varies little over time, and tends to increase as more objects are likely to be in collision. However, the TOI metrics may vary a lot from frame to frame (consider the case where three objects far from one another move in the same direction, two of them finally colliding, and one of the colliding objects bouncing on the trajectory of the third as a result: the first TOI will be very large, but the second one will be much smaller). Hence, the $\frac{CDT}{TOI}$ ratio may, when the density of a cell of territory varies greatly, increase extremely fast. This means it is not safe for use as a sole way to predict future load, without any smoothing mechanism.

In any case, weighting the average load by a function of the density or average of load of its own vicinity may contribute to both spread towards neighbors a newly emerging irregularity in a single world's load, and to avoid the effect of some cells being sent back and forth between two worlds because of a small disbalance between both worlds' load.

Likewise, the CDT of the previous frame can be compared to the new estimated TOI to have a rough idea on whether the next $\frac{CDT}{TOI}$ ratio may be over one (requiring the world to be immediately alleviated from part of its workflow), but the load should never be balanced in last second, in order to be able to leave time for inter-process communication when relevant. From these observations, we can deduce that strictly forbidding overload is not enough, and other parameters should be taken into account for computing the load estimation of each world, such as:

- **the density of the world's territory** is an indicator of the risk of undergoing collisions and of having a much higher CDT in the near future
- **the territory density of neighbor worlds** (as a function that increases for each neighbor that is denser than a given threshold) may be used to smooth future changes of load by anticipating the arrival of more objects within the world's own territory
- **the advance of a world's local clock compared to others** (which is representative of how much they can afford to suffer from higher computational costs)

Load Metrics for Discrete Algorithms

With discrete algorithms, the next time of impact between any objects in a world cannot be predicted, which renders scheduling more difficult. Instead, a good indication of the actual work load is the ratio between the time spent detecting collisions for the time step to be directly rendered (Just-in-time Detection Time, **JDT**) and the total processor time spent in the thread (Total Time Spent, **TTS**).

JDT is a metrics exclusively from the past, whereas the last computed TOI permits spotting cases of emergency when it is obvious that without giving part of its territory, a world will not be able to compute its next step in time. However, many physics engines do not provide continuous collision detection algorithms, thus making it necessary to provide metrics for discrete algorithms as well.

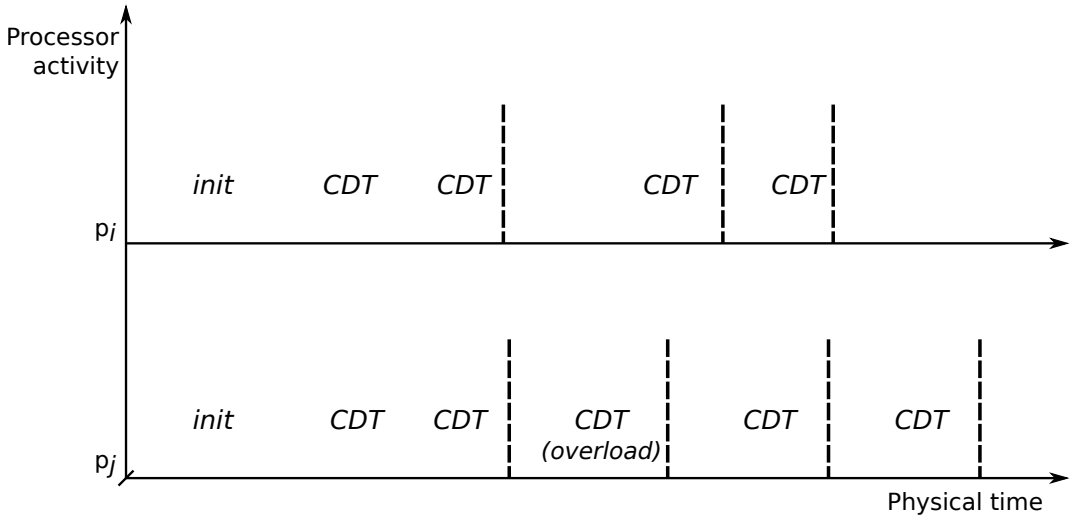


Figure 3.4: Simplified scheme showing the CDT of two worlds for their last few simulation steps and their next estimated TOI. p_j is overloaded in the second step, and highly loaded the rest of the while, while p_i is almost always underloaded. The areas in white are time that is free for anticipated computations.

3.3.2 Optimal Runtime

Implications of Interactions on Load Balancing

Collision detection algorithms have a complexity more than linear in number of objects to check. Consequently, taking territory (more precisely taking objects) from a world will decrease the runtime of its next simulation step more than linearly.

We define the *optimal runtime* as the one taken to end a simulation when the load is perfectly balanced all along. It is by essence lower or equal than the average runtime of separate worlds (with no consideration for the time spent in synchronizing, load balancing or in the rollback and propagate algorithm). Figure 3.5 illustrates how an object moving from a zone to another would affect the simulation time of a world p_i in the absence of load balancing mechanisms. Each object crossing from a slower to a faster world's territory costs the whole simulation to be slower, which justifies setting up both load balancing mechanisms (to avoid losing too big an advance) and an efficient rollback algorithm (to reduce the cost of revalidating precomputed data).

Convergence Towards Optimal Runtime

Load balancing can thus be seen as the following process: whenever a world p_i has T milliseconds of advance, it should be given enough territory so that it computes V ms of simulation in the time it will take for the slowest worlds to compute $T+V$ ms. In other words, every world should try to have enough work so that after some time, all worlds are exactly at the same time step (the one of the slowest of all, called reference time step), but in a best effort mode rather than enforcing this property like on previous models for parallel CD.

The reference time step will progress more quickly as the slowest worlds will be relieved from some of their work load. One factor to take into account is the cost of territory cession, which must be evaluated. Then, the amount of territory to give away could be expressed as a function of the amount of time diverged from optimal runtime and of the *territory cession overhead* (CPU time spent computing load balancing mechanisms and spent exchanging territory). The more important the divergence, the faster the load must be rebalanced, unless it becomes costlier than slower load balancing (e.g. implementations on message-passing based systems should take care about the time it takes to send object data between two physics engines running in separate processes).

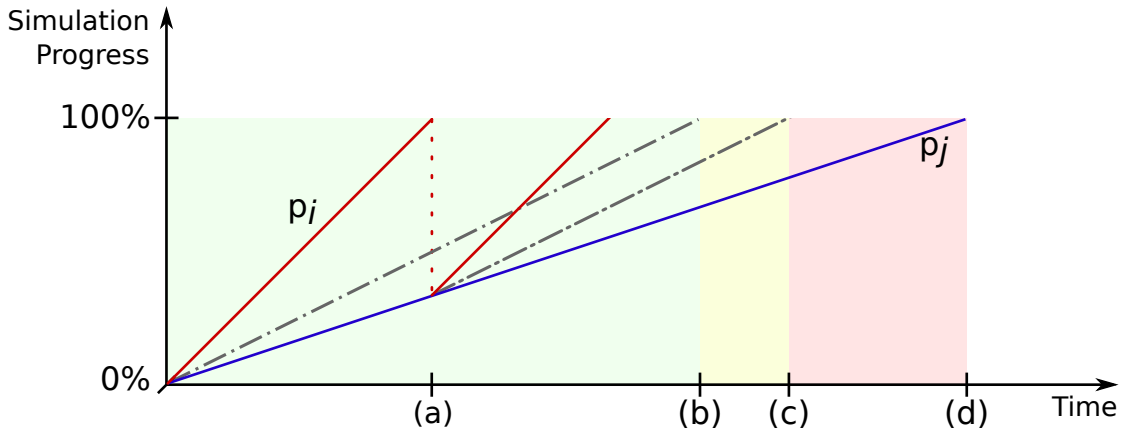


Figure 3.5: **(a)** an object from p_j moves to p_i. **(b)** max time taken if optimal load balancing. **(c)** max time taken if optimal load balancing after frontier crossing. **(d)** time taken without load balancing. p_i loses its advance on p_j as an object crosses the frontier between both worlds, causing the average runtime (which is for 2-body algorithms an upper bound on the optimal runtime) to increase. Here, Rollback and Propagate is voluntarily ignored, and the slight changes in average runtime caused by one object transiting are also not taken into account.

3.3.3 Monitoring Load and Taking Decisions

General Case Principle Current most elaborated parallel algorithms show heterogeneous performances depending on what is being simulated. If using one such algorithms on each world of our model, we may then assume that some worlds may perform collision detection much slower than their neighbors because they manage more complex objects. Thus, monitoring the current load of the system should be done after each collision detection step.

Furthermore, in the adverse of one continuous-algorithm-enabled world knowingly being slower than its estimate TOI by x milliseconds, it may even poll to check the global load of its neighbors, and see if it can give up part of its territory before finishing the current step in order to speed it up (in which situations such a strategy could be fruitful has yet to be determined).

For discrete worlds, x may be determined based on the wanted framerate of the application (more precisely as a number of missed frames since the current step started). Using this approach, every world should be able to determine when it is overloaded, and if its neighbors also are, and then take action as described in Section 3.3.5.

Detecting Underload and Anticipating Load Imbalance There are two distinct situations with regard to which a world p_i may be underloaded: the overall worlds, or its direct neighbors. In the former case, p_i may indicate all of its neighbors that it wants to expand (and how much), which may in turn asynchronously concede part of their territory provided that they themselves are not underloaded according to the same metrics (in which case they will also expand towards their overloaded neighbors and be able to satisfy p_i's request either immediately or some iterations later).

The case where only direct neighbors are overloaded is simpler: p_i has to immediately start expanding towards these neighbors. The threshold at which neighbors are considered overloaded will define how early p_i will try to anticipate poor balance in the system.

A lot of work can be done on determining how exactly it is best to expand, depending on the density of population of the areas towards which p_i expands, on the capacity of its neighbors to also expand if they later are underloaded, etc. Besides, the pace at which expansion can be done is very dependant on the architecture, and may range from instantaneous on shared-memory architectures without processor-specific cache (e.g. a multi-core CPU with a *world* being a core) to slower than collision detection itself on message-passing systems made of several computers.

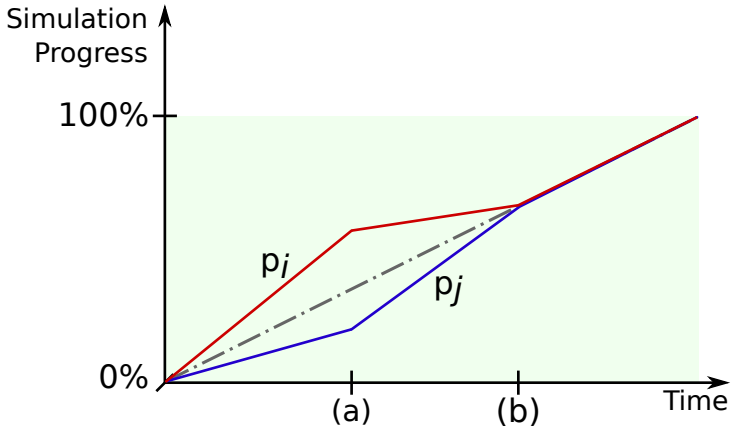


Figure 3.6: **(a)** load is balanced in order to help p_j catch up. **(b)** load is balanced so that worlds have the same running time. Here, load is balanced twice, in order to reduce the cost of object insertions and removals by reducing the advance of fastest worlds, and in order to make the reference time step progress as fast as possible.

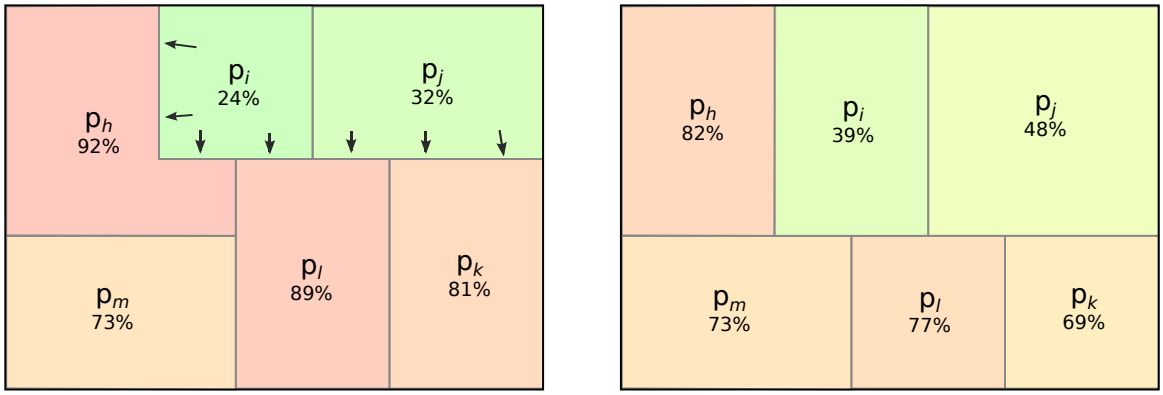


Figure 3.7: Example of two underloaded worlds, p_i and p_j, asking for expansion (the percentage represents the $\frac{CDT}{TOI}$ ratio). If the threshold for considering a world underloaded is 20% below the average, then only p_i will be considered underloaded after this iteration.

3.3.4 Territory and Neighbors Representation

Territories and Cells The territory of a world is a 3D grid made of cells, each of them containing pointers to list of dynamic, static and border objects that exist inside it. The dynamic objects are those actually being simulated, the static ones are the environment on which collisions can also occur, and the border ones are used to detect border traversals but do not cause any physics response. Each cell is associated with an owner id assumed to be always valid.

On simulation setup, we ensure that each world's 3D grid has a margin of one cell over the territory, so that the world id of the owner of all cells surrounding one's territory is always known. Load balancing algorithms are responsible for notifying the new owner id of every cell that changes property as a result of load balancing.

Each cell will be associated with a *computing cost*, that can be defined in many ways. In preliminary implementations, we will only consider the number of triangles within the cell as the computing cost, in order to retain at least a little of the impact of the complexity of objects over collision detection runtime. Other criteria for computing cost could be the distance to a number of closest objects (a higher number of such neighbors already outside the territory of the cell owner increasing the computing cost), and also the direction of objects within the cell (the more moving outwards, the higher the computing cost).

Neighbors Some information on neighbors is helpful in order to know which worlds should be given territory. For instance:

- the list of cells from that neighbor that are adjacent to one's territory, that allows knowing what can be given to maintain the neighbor's perimeter low
- the density and size of the surface of the neighbor, in order to give territory in priority to little worlds, that face more risks of being underloaded since the objects they contain may eventually leave their territory
- an approximation of the shape of neighbors' territory (or a function that tells how a neighbor's perimeter varies given that it is offered a set of cells) would help in knowing whether some cells are more profitable to it than others, not only using the (poorer) information already accessible via the list of adjacent cells

3.3.5 Territory Cession

The general approach to the cession of territory is that underloaded worlds may, if they wish, indicate that they are looking for more work to their neighbors, and give information about themselves to support this demand. Overloaded worlds may, either after a request or on their own initiative, give some territories to any of their neighbors, with these being unable to refuse it unless they have a higher average load. This compulsory aspect makes the algorithm more reactive in case of emergency load balancing from a world that is knowledgedly unable to cope with interactive framerate constraints.

The general case algorithm for giving territory away will be split in three parts: first, the world will determine how much computing cost it wants to give. Then, it will decide a list of territory recipients, each associated with a priority and amount of computing cost to give. Finally, for each candidate recipient, the world will choose which cells best satisfy the amount of computing power and territory shape requirements of both worlds. This last step is kept running until the global amount of computing cost to give is reached.

Selection of Recipient Worlds

The following criteria could be combined in order to assign a priority to every potential recipient world. The impact of their respective weight on the quality of load balancing is left for further research once a full prototype is available.

- amount of neighbors the candidate has (the lesser the neighbors, the higher the priority since the candidate has less opportunities to expand)
- smallest work load among candidates
- smallest territory surface
- a random value as a last resort to differentiate between two worlds with similar characteristics

Locally Choosing Cells to be Given

The criteria to choose which cells must be given are as follow. They are used to determine the next cell (or group of cells) to give until the amount of computing cost is reached. Some of these criteria may be precomputed and updated rather than computed every time.

1. maximal number of foreign neighbors for the cell, as it means there are more chances this cell causes us to synchronize
2. minimizing the perimeter of both involved worlds after giving the cell

3. if no unique cell satisfies previous conditions by a certain threshold, consider pairs of cells
4. maximize the amount of computer cost given
5. consider proximity to scene border and give those closer in priority

Fighting Famine

Famine is a term used in mutual exclusion algorithms to describe the fact that one agent never accesses a shared resource that it needs in order to be able to work. In SODA CD, we define famine as the phenomenon of not having large enough a territory for a world not to be underloaded, and of not having any means of increasing one's territory.

Although a load balancing algorithm should never tolerate the settling of famine, it is better to have emergency mechanisms to overcome it. To propose one, we build up on the solution presented by Miller and Lunding [ML04] as a general means of load balancing. An underloaded world that cannot obtain any territory from its direct neighbors can just bestow them all of its territory, and send a request to all worlds, looking for one of those with the highest load average (the lookup can be based on P2P algorithms). Once such a world has been identified, the underloaded one may takeover part of its territory, in order both to decrease the load of this world and to escape from famine.

Chapter 4

Implementation and Future Experimentations

We describe our shared-memory implementation of the SODA CD framework in Section 4.1. As a first proof-of-concept, we target a twelve-core computer, since we have access to scalability values from several other implementations for such a number of cores. This allows positioning of the implementation itself in terms of raw performance. Besides that, we also want to evaluate the scalability and robustness properties of SODA CD. However, the short duration of the internship did not allow us to develop a complete implementation. Hence, we will describe in Section 4.2 the properties that we want to evaluate and the protocol that we will follow, but we are unable to presently provide the results of a performance evaluation.

4.1 Implementation Status

Our current implementation of SODA CD is designed for shared-memory commodity computers with high numbers of CPUs, as these systems were available for experimentations in our virtual reality center. We decided to use the *Ogre 3D* engine for rendering, and the *Bullet* physics engine for world implementations, as they were readily available, and as we wanted to have the possibility to modify the codebase of the engines we used.

Nevertheless, this choice of engines wasn't without consequences on our implementation. For instance, *Ogre 3D* is not implemented in a thread-safe way, which slowed down the development process and forced us to postpone parallelization of the position and rotation update of 3D engine object representations in the rendering loop. This limited the number of simultaneous objects to a few tens of thousands in preliminary experiments, even when physics engines were obviously underused and when more physical threads were available for object position updating.

Likewise, *Bullet*'s continuous collision detection algorithms are not mature enough for simple use. Consequently, we use only discrete collision detection, which as previously discuted impacts the memory size requirements of circular transform buffers. We will investigate various optimizations in future versions of the implementation, such as discretization of object transforms and speeds for lighter storage cost, to match the constraints imposed by the type of algorithms available in *Bullet*.

Our shared-memory implementation runs on a single system. This limits the scale of simulations that we can run since a single system has typically less memory available than clusters, but it seemed important to have proper implementations of the framework on an architecture with low communication time between worlds. Once this step will be reached, we can assess how much bandwidth is needed to synchronize worlds running on different systems fast enough to retain interactive performance.

4.1.1 Uniform Grid Broad Phase

SODA CD's synchronization algorithm differentiates between border collisions and normal inter-object collisions. Hence, we wanted the broad phase to return, additionally to a list of possibly colliding pairs of objects, a list of potential border collisions, in order not to have to browse all pairs of colliding objects to identify them.

Besides, cell borders are more easily implemented as one static object per cell face, for each cell that is adjacent to a foreign territory. We only need to check against face neighbors (in opposition to corner neighbors). Indeed, it is not possible for an object to overlap a cell located at its home cell's corner without also overlapping those on the sides of its home cells, as a consequence of the rectangle shape of cells. When a world is notified that a foreign object collides with one of its borders, it can test on its own whether it also collides with face-neighbors of the cell where it landed, by transitivity. Either way, cell borders are static objects that never collide with each other. Two families of broad phase algorithms are particularly adapted to such a setup: spatial subdivisions, and BVTT traversal with a single tree for static objects.

Since we already use and maintain a spatial subdivision uniform grid for territory management, we decided to also use it in our broad phase implementation. Considering the minimal cell size constraint that we set, we also know that only directly adjacent (face and corner) cells need to be considered for collision checks against a given cell's object. By browsing our 3D array in topological order in our broad phase, we limit the number of tests even further. Objects from one cell are checked against objects from the same cell, and from cells whose coordinates are ranked higher in our defined topological order (see Figure 4.1).

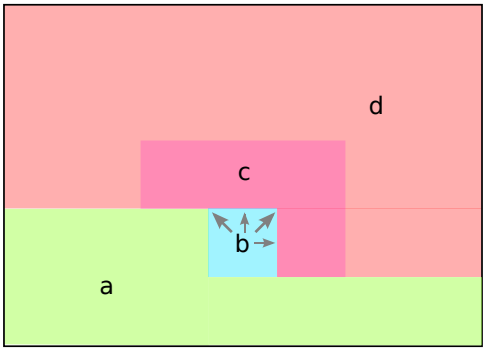


Figure 4.1: 2D example of our uniform grid broad-phase algorithm. **(a)** cells that have already been checked against. **(b)** cell currently being checked. **(c)** cells whose objects will be checked against b's. **(d)** cells not checked yet.

4.1.2 Thread Scheduling

Although inter-process communication is not yet finalized in SODA CD, the ability to perform thread placement on a shared-memory, fully thread-based implementation was of crucial importance to us. Indeed, the interactive framerate constraint lead us to choose the lightest IPC solution available: threads sharing a process address space.

However, a multiple core system is typically shared between many applications (especially since virtual reality applications massively rely on I/O with various devices, often requiring a user space to be setup to run, such as graphics or haptic devices). Hence, there is no guarantee that several worlds will run on different physical threads in practice. We postulate that this may hinder the predictability of available runtime for each world, and would rather make sure that there are as many worlds as physical threads and that each world is always assigned the same physical thread.

For instance, consider a twelve-core system, with a parallel-rendering enabled 3D engine that makes use of three threads. A random scheduling strategy would imply that any of the twelve worlds can be

scheduled on the same core as a rendering thread, and so any of them could suffer a thread context switch because of this scheduling strategy. This may hinder the regularity of the CPU time available for each core, in contrast with a scheduling strategy where exactly the same three worlds would be systematically penalized by context switches. Even though unintuitive, such a strategy guarantees that worlds have a more stable amount of CPU time. Hence, load balancing algorithms will not have to rebalance territories because of scheduling issues, but only as a result of actual changes in the distribution of collision events.

Consequently, we make use of a lightweight thread library called *Marcel* [TNW07]. Its scheduler BubbleSched allows the definition of bubbles of threads that are then scheduled instead of the threads themselves. Using this, we hope to be able to tie all CPU-intensive computations not related to physics engines to a fixed set of worlds (that will thus manage smaller territories). Among the experiments to run, we want to evaluate the efficiency of such a mechanism.

4.2 Future Experimentations

In this Section, we do not present results of performance evaluation as the implementation of SODA CD is still ongoing at the time of submitting this thesis. We will however describe the performance metrics that we want to setup in addition to the feature tests that we presented as necessary in the previous sections.

Input benchmark simulations Many benchmark simulations exist for testing the performance of collision detection frameworks, even though no review describe what production use case each of these simulations are designed to match. Besides, each simulation can be run with different model qualities that will strongly impact performance. Hence, comparison with another algorithm is possible only when the simulated data exhibits the same properties (topology, number of polygons, density, regularity of distribution, etc.). The sample simulation that we use during development is simply a simulation of randomly placed cubes falling on the ground.

Measurement of scalability Scalability is traditionally defined as the runtime or throughput performance improvement of an application compared to its performance using a sequential processor. We are interested in comparing our results to those presented by Avril [Avr11] in his thesis as his code base is available to us for comparison, and to the results presented by Hermann et al. [HRF09] since they include speedup results for twelve-core computers.

In order to evaluate scalability, we can retain for each time step the runtime of the slowest world. The average of these worst runtimes can then be compared with the runtime of a sequential collision detection algorithm. This scalability measurement will be comparable to the one of precited papers.

Raw performance The raw performance of our framework can be evaluated by assessing how many cubes it can simulate before degrading the rendering framerate below twenty-four frames per second (a sane minimum for interaction with users). However, this measurement is only useful as a means of comparing how SODA CD behaves when a feature is enabled or not. Even so, we will see that certain other metrics are sometimes more adapted.

Resistance to computational peaks One desired property of SODA CD is robustness, and it is this property that lead to the choice of anticipated computations. Hence, we want to evaluate whether the gains from anticipative computing are real, focusing on two aspects: whether they help in making clever use of processor idle time in cases of work load unbalance, and whether they help in absorbing and making less noticeable to users the performance loss that occurs during a computational peak.

The former objective will be reached by evaluating whether worlds ahead of others do not reach the maximal capacity of their circular transform buffer before balancing load better and how many of their anticipated computations are later invalidated.

For evaluating the quality of the property respectively to the second objective, we need to associate peaks with a quantified cost. The cost of a peak can simply be defined as the number of collisions that occur during the peak compared to the average number of collisions before the peak. Then, we can verify experimentally how the circular transform buffer size is correlated to the robustness of SODA when facing peaks. For instance, what is the cache size that allows keeping 2 seconds of advance in simulations with a 4-fold cost computational peak. It might even be possible to express the correlation between the duration and cost of the peak (in simulations where it is known) and the necessary buffer size to resist them without users noticing anything.

Synchronization Moreover, evaluating the time a world spent synchronizing may give useful information about how to make use of it. Likewise, the percentage of objects in a simulation that cause worlds to synchronize can give indications on the minimal bandwidth required between two worlds on message-passing implementations. Another important thing to evaluate is the efficiency of the Rollback and Propagate algorithm. This can be performed by computing the time ratio spent in the rollback algorithm and by computing the time saved by anticipated computations and Rollback and Propagate (time it would have taken to compute collisions just in time minus time spent in the rollback algorithm). Another approach would be to evaluate the percentage of a territory in which collision checks are necessary when calling the Rollback and Propagate algorithm, the lower being the better.

Chapter 5

Conclusion

In this master's thesis, we have presented SODA CD, a novel collision detection model, that is built around hypotheses formulated after an analysis of the weaknesses of existing state-of-the-art parallel collision detection algorithms. This model is designed to be particularly scalable on multi-cpu systems, and to be anticipative as a means to eliminate processor idling phenomena.

We showed how to distribute collision detection and response among several independant physics engines with loose synchronization mechanisms that do not require constant clock synchronization between all engines, how to anticipate computations in a given engine when it has run out of work, and how to intelligently roll back anticipated computations when they are invalidated as a consequence of synchronization.

We have proved that it was possible to initialize a virtual reality simulation by partitioning it onto different physics engines. Then, we have also showed that there exists a strong potential for load balancing in our model.

We have then drawn an overview of the status of our implementation of the SODA CD model, and described the performance metrics that we expected to be useful in validating the hypotheses on which SODA CD has been built.

Possible future work include a better formalization of the load balancing mechanisms and of the impact and design consequences of the running system's and implementation technologies' properties on SODA CD (such as message-passing vs. shared-memory architectures or discrete vs. continuous algorithms), the completion of an implementation and performance comparison with existing algorithms for large-scale simulations.

Bibliography

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[AGA11] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. Dynamic adaptation of broad phase collision detection algorithms. In *VR Innovations (ISVRI) - IEEE International Symposium*, pages 41–47, March 2011.

[AGA12] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. Fast collision culling in large-scale environments using gpu mapping function. In *ACM Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2012.

[Avr11] Quentin Avril. *Détection de Collision pour Environnements Large Échelle : Modèle Unifié et Adaptatif sur Architectures Multi-cœur et Multi-GPU.* PhD thesis, INSA Rennes, 2011.

[Ber97] Gino Van Den Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4):1–13, 1997.

[Ber99] Gino Van Den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2):7–25, 1999.

[Ber01] Gino Van Den Bergen. Proximity queries and penetration depth computation on 3d game objects. In *Game Developper Conference*, 2001.

[BT95] Srikanth Bandi and Daniel Thalmann. An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies. *Comput. Graph. Forum*, 14(3):259–270, 1995.

[CK86] R. K. Culley and K. G. Kempf. A collision detection algorithm based on velocity and distance bounds. In *IEEE International Conference on Robotics and Automation*, pages 1064–1069, 1986.

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *SI3D*, pages 189–196, 218, 1995.

[CTM08] Sean Curtis, Rasmus Tamstorf, and Dinesh Manocha. Fast collision detection for deformable models using representative-triangles. In Eric Haines and Morgan McGuire, editors, *SI3D*, pages 61–69. ACM, 2008.

[EL01] Stephen A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Computer Graphics Forum*, 20(3), September 2001.

[Eri05] Christer Ericson. *Real-time Collision Detection.* Morgan Kaufmann, 2005.

[GF90] Elmer G. Gilbert and Chek-Peng. Foo. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation*, 6(1):53–61, February 1990.

[GJK88] Elmer G. Gilbert, Daniel W. Johnson, and Sathiya S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4:193–203, 1988.

[Got00] Stefan Aric Gottschalk. *Collision queries using oriented bounding boxes.* PhD thesis, The University of North Carolina at Chapel Hill, 2000. AAI9993311.

[GRLM03] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In M. Doggett, W. Heidrich, W. Mark, and A. Schilling, editors, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 025–032. Eurographics Association, 2003.

[Gut85] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In Shamkant B. Navathe, editor, *Proceedings of the 10th ACM International Conference on Management of Data (SIGMOD)*, pages 47–57. ACM Press, 1985.

[HFR08] Everton Hermann, Francois Faure, and Bruno Raffin. Ray-traced collision detection for deformable bodies. In *GRAPP*, pages 293–299, 2008.

[HRF09] Everton Hermann, Bruno Raffin, and François Faure. Interactive physical simulation on multicore architectures. In Kurt Debattista, Daniel Weiskopf, and João Comba, editors, *EGPGV*, pages 1–8. Eurographics Association, 2009.

[HRF+10] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-GPU and multi-CPU parallelization for interactive physics simulations. *Europar*, September 2010.

[Hub95] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995. ISSN 1077-2626.

[Hub96] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph*, 15(3):179–210, 1996.

[JP04] Doug L. James and Dinesh K. Pai. BD-tree: output-sensitive collision detection for reduced deformable models. *ACM Trans. Graph.*, 23(3):393–398, 2004.

[KGS97] Dong-Jin Kim, Leonidas J. Guibas, and Sung-Yong Shin. Fast collision detection among multiple moving spheres. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 373–375, New York, June 4–6 1997. ACM Press.

[KHeY08] DukSu Kim, Jea-Pil Heo, and Sung eui Yoon. Pccd: Parallel continuous collision detection. Technical report, Dept. of CS, KAIST, 2008.

[KHH+09] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-Eui Yoon. HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs. *Comput. Graph. Forum*, 28(7):1791–1800, 2009.

[KHI+07] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and Richard Rowe. Collision detection: A survey. *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 4046–4051, Oct. 2007.

[KMSZ98] James T. Klosowski, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[KP03] Dave Knott and Dinesh K. Pai. Cinder: Collision and interference detection in real-time using graphics hardware, 2003.

[LC91] Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation. Technical report, University of Berkeley, California, March 19 1991.

[Lev66] Cyrus Levinthal. *Molecular model-building by computer.* Scientific American, 1966.

[LG98] Ming C. Lin and Stefan Gottschalk. Collision detection between geometric models: a survey. In Robert Cripps, editor, *Proceedings of the 8th IMA Conference on the Mathematics of Surfaces (IMA-98)*, volume VIII of *Mathematics of Surfaces*, pages 37–56, Winchester, UK, September 1998. Information Geometers.

[LG07] Scott Le Grand. Broad-phase collision detection with cuda. *GPU Gems 3 - Nvidia Corporation*, 2007.

[LHLK10] Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J. Kim. Real-time collision culling of a million bodies on graphics processing units. *ACM Trans. Graph*, 29(6):154, 2010.

[LJS+07] Ruipeng Li, Hai Jiang, Hung-Chi Su, Bin Zhang, and Jeff Jenness. Speculative and distributed simulation of many-particle collision systems. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems - Volume 01*, ICPADS '07, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

[LK02] Orion Sky Lawlor and Laxmikant V. Kalée. A voxel-based parallel collision detection algorithm. In *Proceedings of the 16th International Conference on Supercomputing (ICS-02)*, pages 285–293, New York, June 22–26 2002. ACM Press.

[LMM10] C. Lauterbach, Q. Mo, and D. Manocha. gproximity: Hierarchical gpu-based operations for collision and distance queries. *Computer Graphics Forum*, 29(2):419–428, 2010.

[MC95] Brian Mirtich and John F. Canny. Impulse-based simulation of rigid bodies. In *SI3D*, pages 181–188, 217, 1995.

[Mel00] Stan Melax. Dynamic plane shifting bsp traversal. In *Graphics Interface*, pages 213–220, 2000.

[Mir97] Brian Mirtich. Efficient algorithms for two-phase collision detection. Technical report, Mitsubishi Electric Research Laboratories, December 1997.

[Mir98] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Trans. Graph*, 17(3):177–208, 1998.

[ML04] S. Miller and S. Luding. Event-driven molecular dynamics in parallel. *J. Comput. Phys.*, 193(1):306–316, January 2004.

[Ove92] Mark H. Overmars. Point location in fat subdivisions. *IPL: Information Processing Letters*, 44, 1992.

[PKS10] Simon Pabst, Artur Koch, and Wolfgang Straβer. Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. In *Comput. Graph. Forum*, volume 29, pages 1605–16212, 2010.

[RSH00] Erik Reinhard, Brian E. Smits, and Chuck Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 299–306, London, UK, UK, 2000. Springer-Verlag.

[Sam90] Hanan Samet. *Applications of Spatial Data Structures.* Book, 1990.

[TBW09] Daniel J. Tracy, Samuel R. Buss, and Bryan M. Woods. Efficient large-scale sweep and prune methods with AABB insertion and removal. In *VR*, pages 191–198. IEEE, 2009.

[THM+03] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomeranets, and Markus Gross. Optimized spatial hashing for collision detection of deformable objects. In T. Ertl, B. Girod, G. Greiner, H. Niemann, H.-P. Seidel, E. Steinbach, and R. Westermann, editors, *Proceedings of the Conference on Vision, Modeling and Visualization 2003 (VMV-03)*, pages 47–54, Berlin, November 19–21 2003. Aka GmbH.

[TKH+05] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, Marie-Paule Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Straßer, and Pascal Volino. Collision detection for deformable objects. *Comput. Graph. Forum*, 24(1):61–81, 2005.

[TLW11] Chen Tang, Sheng Li, and Guopin Wang. Fast continuous collision detection using parallel filter in subspace. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 71–80, New York, NY, USA, 2011. ACM.

[TMLT11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 63–70. ACM, 2011.

[TMT09] Min Tang, Dinesh Manocha, and Ruofeng Tong. Multi-core collision detection between deformable models. In Willem F. Bronsvoort, Daniel Gonsor, William C. Regli, Thomas A. Grandine, Jan H. Vandenbrande, Jens Gravesen, and John Keyser, editors, *Symposium on Solid and Physical Modeling*, pages 355–360. ACM, 2009.

[TNW07] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *EuroPar*, Rennes, France, 2007.

[TPB08] Bernhard Thomaszewski, Simon Pabst, and Wolfgang Blochinger. Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics*, 32(1):25–40, 2008.

[Tur89] Greg Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, North Carolina, 1989.

[Van94] George Vaněček, Jr. Back-face culling applied to collision detection of polyhedra. *The Journal of Visualization and Computer Animation*, 5(1):55–63, January–March 1994.