A Multi-Pass Multi-Stage Multi-GPU Collision Detection Algorithm

Jose M. Juarez-Comboni, Andy M. Day
School of Computing Sciences,
University of East Anglia,
Norwich, United Kingdom
J.Juarez-Comboni@uea.ac.uk

Abstract

We present the work in progress of a collision detection (CD) algorithm on a multi-threaded environment and based on two graphics processor units (GPUs); the first GPU is used as a normal graphics processor, and the second GPU is doubled as a 'collision detection' coprocessor. Our approach makes use of the stream parallel engine provided by modern GPUs together with their native architecture for handling vector operations; thus we aim to perform fast and reliable collision detection. Our initial work involves multi-threading and incorporates a cheap pre-processing stage for storing essential scene data in a format suitable for GPUs (i.e. as textures). It also involves multi-pass rendering in multiple stages. The first stage of collision detection is based on a simple boundary volume collision query. If the result is positive, a second stage is then executed, involving objects pair-wise computations on a per-vertex basis. The approach is being implemented on a single Processor NVIDIA Single-Link Interface (SLI) [1] System with two NVIDIA GeForce 6600GT graphics cards.

Keywords: Real-Time Rendering, Collision Detection, Graphics Hardware, SLI, Multi-Threading, general-purpose GPU.

1. INTRODUCTION

In a virtual environment, computers have to emulate the conditions of the real world. This does not only involve using realistic computer generated models, but also the simulation of physical laws that allow a proper interaction between the user and every single object present in the environment. In other words, the better the approximation of the physics, the higher the fidelity of the virtual environment. Current graphic- subsystems allow the use of very detailed models at interactive framerates; nevertheless, they are always bound to the CPU computing power, which has to be shared for the calculation of physics and artificial intelligence (A.I.). The work in progress presented in this paper is an attempt to use the streaming parallel architecture of the state of the art GPUs, in combination with the new features brought forward by the latest generation of peripheral's interconnect (i.e. PCI-Express and SLI) and the future introduction of Dual-Core CPUs, to do very fine physics calculations, more specifically, collision detection computations for interactive virtual environments. We define as interactive, a virtual environment that is able to refresh the scene and give feedback to the user's inputs at least 30 times per second.

With the introduction of 3D graphics hardware, virtual environments have benefit with more detailed and complex models. Nowadays, a single object can easily contain 100K triangles of complexity; further, a single scene can contain several of these objects easily increasing the scene complexity to over 1M triangles. Intersecting surfaces and penetration depth algorithms have been very well-studied in the last few decades to generate efficient algorithms that can work in interactive real-time applications. Some of the commonly used methods involve a pre-computing stage where all objects in a particular scene are analysed and stored in special structures known as boundary volume structures (i.e. OOBB, AABB, distance/field, etc.); they work well on models undergoing rigid motion.

In the past few years graphics hardware has overcome Moore's curves predictions [2]. This and the continuous growth in graphics hardware requirements have pushed manufacturers to incorporated programmable units inside their GPUs. This has given developers the flexibility to create customized effects through the use of specialized programs known as 'shaders'; GPUs can now be seen as a stream parallel computer. Based on this new computing power offered in desktops and workstations, several algorithms for tackling the collision detection problem have been proposed. The remaining of this paper will be presented as followed:

- Section 2 is a brief review of related collision detection algorithms;
- Section 3 introduces the basic assumptions to solve the simplest case with our method, and the stage involved in our ongoing work;
- Section 4 presents a discussion and future work;

2. RELATED WORK

Collision detection is the heart for simulating interactions between objects, and it is the main component that gives the user a feeling of presence. It is also the most difficult aspect of a physical engine to implement correctly, and invariably, it is the main consumer of CPU power. The earliest applications of 3D collision detection are found in robotics and automation [3][4]. In computer animation, the first uses of collision detection are found in physics-based simulations, where it is essential to determine collisions in a physically convincing manner and at interactive rates. Because of the complex nature of this task the use of accelerating techniques is required for interactive environments. Lin and Canny [5] presented one of the first algorithms that exploit temporal coherence to reduce the cost of collision detection; their method caches an update the closest features (vertices, edges, facets) of objects in every new frame. Lin and Canny realized that if frame coherence is high in a scene it is faster to update the closest filters of a pair of objects from the previous frame, than to calculate everything from zero. This technique is applied in I-COLLIDE [6], the first the public library of collision detection. The most commonly CD algorithm used are based on boundary volume hierarchies (BVH). Basically, these algorithms enclose the model of interest in a 'watertight' volume.

Volume structures such as spheres [4], axis-aligned bounding boxes (AABB) [7], object-oriented bounding boxes (OOBB) [8], discrete orientation polytopes (k-dops) [9], have been frequently used. These structures aim to accelerate the CD computations to allow interactive applications. However, with the increasing model's complexity and increased scene detail users expect finer and more realistic collision detection.

With the advent of computer graphics hardware, various approaches employing graphics hardware for collision detection have been introduced. We can divide these approaches into two different categories; image-based, and object-based. Due to the nature of computer graphics and the different optimizations present in graphics hardware (i.e. depth, accumulator and stencil buffers), image-based collision detection has been of more interested for researchers. An early attempt of using hardware depth buffer was presented by Shinya et al [10]. Myszkowski et al. [11] presented another algorithm that used a combination of stencil buffer and depth buffer operations to implement collision queries. These early attempts were only applicable to convex objects and could not be applied for more than two objects. Baciu et al. [12] extended the latter to compute the area of overlap between two interfering solids. Recently, more efficient image-based collision detection methods have been introduced. Govindaraju et al. presented an image-based collision detection approach known as CULLIDE [13] Their methods do not require a pre-processing stage, making it ideal for handling non-rigid motion. They perform a set of visibility queries from different views in a multiple-pass rendering technique to compute what they call a Potentially Colliding Set (PCS).. Once the PCS is created, a second stage is carried out at primitive level (i.e. triangles) to obtain a PCS at sub-object level. Thus the amount of pair-wise computation needed to perform exact collision tests in the CPU is reduced. Q-CULLIDE [14] and R-CULLIDE [15] present optimizations over the original method. They also extend CULLIDE to perform self-collision detection (self-CD). Although these methods are suitable for manifold and non-manifold objects, they cannot compute overlap information and depth of penetration in a collision.

Heidelberger et al. have also presented image-space based approaches relying on volumetric collision queries for non-manifold objects [16], [17], and [18]. Their algorithm proceeds in three stages: the first stage computes the Volume-of-Interest (VoI) as an AABB representing the volume where collision queries are performed (intersection of two objects' AABB, or for self-CD, the objects AABB). The second step is to compute a LDI (Layer Depth Image) for objects inside the VoI. A LDI consists of images storing the depth values, front-face and back-face classification of the objects' primitives. The LDI is used for: self-collision, collision between objects, and vertex-in-volume tests. For self-CD, the entry and exit points registered in the LDI are tested. If the evaluation does not record an entry point followed by an exit point, self-collision is detected; for collision detection between objects, their LDIs are combined using Boolean intersection. Finally, individual vertices are tested against the volume of the object. The vertex is transformed into the local coordinate system of the LDI. If a transformed vertex intersects with an inside region, a collision is detected.

The main drawback of image based methods is that they are limited to image-space resolution i.e. a resolution of 640x480 pixels may miss more collisions than a 1024x768pixels resolution. In contrast, Object-based methods do not have this downside; however they have not been very well studied yet. Choi et al. [19], rather than offering an interactive collision detection algorithm for large and complex environment, they offer a GPU-based approach to detect self-collisions inside a deformable object. The technique detects pair-wise collisions among vertices within the object, and renders the results in a texture of dimensions $m \times m \ \left(m = 2^{\log_2 N}\right)$, where N is the number of triangles of the object, and $m \geq N$. To implement collision detection on the GPU, they store the position of a triangle in three different 1D-textures of size m (one per vertex). These textures are continuously mapped into a quad both, horizontally and vertically. Thus, a pair-wise comparison of the object's primitives is undertaken in the GPU. The Stencil buffer is used to avoid processing adjacent and distant triangles. Kolb et al. [20] offer another approach for collision detection fully performed on a GPU. Although the technique is focused on simulating state-preserving particle systems, it offers a novel technique to represent directional data applied to store indexed normal vectors. The technique consists of a heap structure that stores all available indices and that it is optimised to always return the smallest available index. Kolb's collision detection technique uses depth maps to store the information relative to a particle (distance to the collider object, relevant object surface point, transformation matrix, z-scale factor) in video memory, and compute distance queries and collision between a particle and the collider object. Since math operations are executed on the GPU, this approach is dependent on hardware resolution (amount of bits assigned per component) rather than image-space resolution. For instance, a GPU able to handle 128-bit colours (i.e. 32-bits per component) will give better CD results than a GPU only capable of 96-bits colours (i.e. 24-bits per component). This approach presents a full framework for collision detection/collision reaction based completely on general-purpose GPU computations (GPGPU).

3. METHOD

This section presents an overview of the system setup followed by a set of basic assumptions for our approach and a detailed description of the Multi-pass Multi-stage Multi-GPU Collision Detection or M^{3}CD algorithm

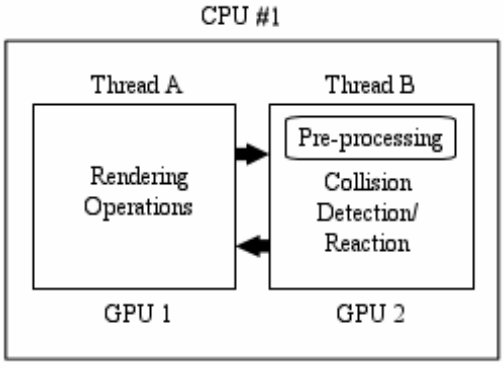

Figure 1: Environment setup

Figure 1 shows the programming structure we have decided to follow. The platform selected is WIN32[1] because it was the first platform to offer SLI enabled drivers. As of today, neither the

[1] Microsoft® Windows® XP Professional Edition

OpenGL API nor the graphics hardware drivers offer a direct way to select a GPU to carry our an arbitrary process; hence, our approach creates one window per thread, each one with its own device context and rendering context. This way we can target a specific GPU for the operations we want it to perform. Furthermore, the window in thread B is an invisible window, since it is only the off-screen buffers we are interested to handle here. The multithreading approach also allows us to perform collision detection in a virtually independent way. Besides, it will benefit of the introduction of dual-core CPUs.

3.1 Basic Assumptions

In order to simplify the problem some basic assumptions need to be made. Although problems may arise from these assumptions, they help solving the simplest case. Later on the algorithm will be improved to circumvent any existing problems.

1. The objects that the algorithm is dealing with are convex 3D objects only (in the near future the algorithm will work for non-manifold objects)
2. Detection of any pair of bounding volumes and vertices colliding is a well-defined and solved problem.
3. Both objects have a pre-calculated center point and a bounding sphere radius (BSR).
4. Object A has N vertices while object B has M vertices.
5. N = M
6. Objects are subject to a rigid motion only and the direction of this motion is known at any time (in the future the M^3CD algorithm will be extended to handle non-rigid motion).

If assumption 3 is not meet, then a pre-processing stage is engaged to calculate the geometric centre of the object together with the radius of the bounding sphere.

3.2 The M^3CD Method

Input. Our approach takes the geometric centre of every 3-D object in the scene, plus the radius of the smallest sphere surrounding the object.

Output. The algorithm computes collision queries in two different stages. If queries are positive in both stages, then the properties of the objects involved in the collision are modified (i.e. direction and origin) and results are feedback to the rendering thread.

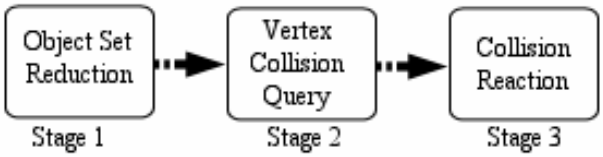

Figure 2: Algorithm Overview

As shown in Figure 2, our approach proceeds in three stages.

3.2.1 Sphere Intersection

The centre point (a 3D-vector) and BSR (a scalar) of every object are stored in a 2D RGBA floating point texture, so that one object corresponds to one texel. The stage is divided in two passes; the first pass will set the stencil buffer so that only the texels containing object's data. This first pass is very fast and helps minimizing overheads in the fragment shaders. The second pass will render a full-screen quad into the off-screen buffer and map the texture to it. Figure 3 shows an example of how the textured quad should look before and after stencil test is done.

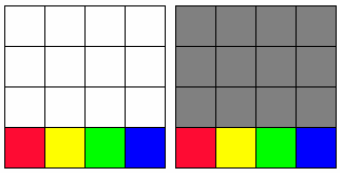

Figure 3: Original Texture without stencil test (right) and Texture read after stencil buffer test [texels in grey are not considered] (left).

The mechanism employed in the fragment shader to perform sphere intersections is very simple: a) intersections are done pair-wise, b) intersections are done following a permutation basis i.e. object B can only be compared with object A once, regardless of the order. Hence if we have N objects, the number of operations are reduced to the factorial of (N-1).

If two spheres do not overlap, the corresponding objects cannot collide. Otherwise, the objects involved are stored pair-wise in a stack structure, creating a 'Potentially Colliding Batch' (PCB) fur further processing.

3.2.2 Vertex Collision Query

Stage 2 executes a finer collision query between each pair of objects inside the PCB. Following the basic assumptions earlier stated, in order to detect a collision between two objects, the most simple but inefficient method of checking every vertex in object A against every vertex in object B is performed. For the brute force method therefore an order magnitude of $O(N^2)$ represents the time complexity of the algorithm. For this stage vertices of both objects are stored in a single texture or Vertex-Texture Map (VTM). The texels located in the first half of the texture represent the vertices belonging to object A; likewise, the second half of the texture represents object B as shown in Figure 4.

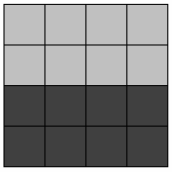

Figure 4: Vertex-Texture Map (VTM)

The collision queries performed in this stage use the Euclidean distance between vertices. If the distance between any pair of vertices is less or equal to zero, a collision is detected and both objects are marked.

3.2.3 Collision Reaction

Stage 3 checks the result from the previous stage. When a pair of objects inside the PCB is found as marked, their centre point coordinates and motion's direction are uploaded into texture memory (following the multi-pass rendering process mentioned in stage 1. A fragment program is then executed to calculate the new centre point coordinates (if necessary), and the new direction. This data is read back from the graphics hardware, and is used to update the state of each object. Results are then submitted to the main thread for updating the scene.

4. CONCLUSSIONS AND FUTURE WORK

This paper has presented the first stage of this project, which is to provide the simplest framework for accelerating collision detection using the streaming parallel computing features of current graphics hardware. The main disadvantage of the technique described in the computing intensive task currently employed for calculating fine collision detection. Nevertheless, there is a lot of room for this algorithm to be improved. The aim is to create a novel and robust method for accelerating collision detection for complex models in large scale. Furthermore, due to the parallel nature of the algorithm it can also be suitable for applications such as cloth simulation and medical simulation among other physically based simulations. Unfortunately, we are at an early stage of the project and proper results cannot be presented in this paper yet. However, results will be available before the presentation of this paper.

4.1 Future work

It is obviously that the brut force approach used to compute the collision detection at vertex level can overwhelm the processing capabilities of the latest generation of GPUs; especially when complex models are involved. Ongoing work is already concentrating on a set of algorithms to reduce the set of vertices involved in the vertex collision detection stage. Furthermore, the algorithm could benefit of the use of multiple-render targets (MRTs), which allow the graphics hardware to output up to four 4-vector values per processed fragment. Finally, different methods for storing data in texture memory are being studied and tested [21], [22].

5. REFERENCES

[1] http://www.slizone.com/content/slizone/learn.html

[2] Moore, G. E., “Cramming more components onto integrated circuits”, Electronics Magazine, Volume 38, Number 8, April 1965.

[3] Boyse, J. W., “Interference detection among solids and surfaces”, Communications of the ACM, vol. 22, pp. 3-9, 1979.

[4] Quinlan, S., “Efficient Distance Computation Between Non-Convex Objects”, Proc. IEEE Int. Conference on Robotics and Automation, IEEE, 3324-3329, 1994.

[5] Lin, M. C., and Canny, J. F., “A fast algorithm for incremental distance computation”, Proc. IEEE International Conference on Robotics and Automation, pp. 1008-1014, 1991.

[6] Cohen, J., Lin, M., Manocha, D., and Ponamgi, M., “I-COLLIDE: An interactive and exact collision detection system for large-scale environments”, Proc. ACM interactive 3D Graphics Conference, pp. 189-196, 1995.

[7] Van Den Bergen, G., “Efficient Collision Detection of Complex Deformable Models Using AABB trees”, Journal of Graphics Tools 2, 4, 1-13, 1997.

[8] Gottschalk, S., Lin, M. C., and Manocha, D., “OBBTree: A hierarchical structure for rapid interference detection”, Proc. SIGGRAPH ’96, pp. 171-180, 1996.

[9] Fünfzig, C., and Fellner, D. W., “Easy Realignment of k-Dop Bounding Volumes”, Institute of Computer Graphics, Technical University of Braunschweig, Germany 2003.

[10] Shinya, M., and Forgue, M.-C., “Interference Detection Through Rasterization”, The Journal of Visualization and Computer Animation, 2(4):131-134, 1991.

[11] Myszkowski, K., Okunev, O. G., and Kunii, T. L., “Fast collision detection between computer solids using rasterizing graphics hardware”, The Visual Computer, 11:497-511, 1995.

[12] Baciu, G., Wong, W., and Sun, H., “RECODE: an image based collision detection algorithm”, The Journal of Visualization and Computer Animation, pp. 181-192, 1999.

[13] Govindaraju, N. K., Redon, S., Lin, M. C., Manocha, D., “CULLIDE: interactive collision detection between complex models in large environments using graphics hardware”, ACM SIGGRAPH/Eurographics Graphics Hardware, pp. 25-32, 2003.

[14] Govindaraju, N.K., Lin, M.C., Manocha, D., “Q-CULLIDE: Fast self-collision culling in general environments using graphics processors”, Technical Report TR03-044 of University of North Carolina at Chapel Hill, 2003.

[15] Govindaraju, N., Lin, M. C., and Manocha, D., "RCULLIDE: Fast and Reliable Collision Culling using Graphics Processors", UNC-CH Technical Report, Jan 2004.

[16] Heidelberger, B., Teschner, M., and Gross, M., “Volumetric Collision Detection for Deformable Objects”, Technical Report No. 395, Institute of Scientific Computing, ETH Zurich, Switzerland, April, 2003.

[17] Heidelberger, B., Teschner, M., and Gross, M., “Detection of collision and self-collisions using image space techniques”, Proc. Computer Graphics, Visualization and Computer Vision WSCG'04, pp. 145-152, 2004.

[18] Heidelberger, B., Teschner, M., and Gross, M., "Real-Time Volumetric Intersections of Deforming Objects", Proc. Vision, Modelling, Visualization VMV'03, pp. 461-468, 2003.

[19] Choi, Y.-J., Kim, Y J., Kim, M.-H., “Self-CD: Interactive self-collision detection for deformable body simulation using GPUs”, Simulation Conference (LNCS), 2004.

[20] Kolb, A., Latta, L., and Rezk-Salama, C., “Hardware-based simulation and collision detection for Large particle systems”, Eurographics/SIGGRAPH Workshop on Graphics Hardware, 2004.

[21] Lefohn, A., Kniss J., and Owens, J., “Implementing Efficient Parallel Data Structures on GPUs”, GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation, 33:521-545.

[22] Harris, M., and Buck, I., “GPU Flow-Control Idioms”, GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation, 33:547-555.

About the authors

Jose Juarez-Comboni is a PhD student at the School of Computing Sciences in the University of East Anglia. His contact email is J.Juarez-Comboni@uea.ac.uk.

Andy M. Day is a lecturer at the School of Computing Sciences in the University of East Anglia. His contact email is amd@cmp.uea.ac.uk.