HCI 2007 튜토리얼

General Purpose Computation using Graphics Processing Units (GPGPU)

개요

최근들어 그래픽스 하드웨어 (GPU)의 계산 대역폭 및 데이터 전송 대역폭 증가와 GPU의 스트리밍 계산 능력의 덕택으로 많은 연구자들이 GPU를 그래피컬 데이터 처리외에 일반적인 연산 (General purpose computation)에 적용하는 방법을 연구해왔다. 이런 연구를 흔히 GPGPU라고 부른다. 본 튜토리얼에서는 GPGPU의 일반적인 접근법에 대한 소개와, 개발 과정에 필요한 각종 툴들과 연구자들의 시행착오, 그리고 국내의 GPGPU연구자들의 현황등을 살펴보고, GPU의 대표적인 벤더인 nVIDIA의 G80 GPU 아키텍쳐및 CUDA technology에 대한 소개를 겸한다. 본 튜토리얼의 목차는 다음과 같다.

1. GPGPU 연구의 배경
2. GPGPU 기본적 접근 방법 및 고려사항
3. GPGPU 개발을 위한 팁
4. nVIDIA GPU 소개
    a. G80 architecture
    b. CUDA technology
5. 국내 GPGPU연구 사례
    a. 변형 모델의 충돌검사
    b. 물리 시뮬레이션
    c. 뉴럴 넷웍, 4진 트리 탐색, 이미지 피라미드
    d. 스키닝 에니메이션

강사 소개

김영준 교수

이화여대 공과대학 컴퓨터학과
kimy@ewha.ac.kr

김영준 교수는 현재 이화여자대학교 공과대학 컴퓨터정보통신학과의 조교수로 근무중으로, 1993, 1996 년 서울대학교 계산통계학과에서 각각 학사, 석사 학위를 수여하였고, 2000 년 미국의 Purdue University 에서 박사학위를 수여하였다. 김 교수는 이화여대에 부임하기전에, 미국의 University of North Carolina at Chapel Hill 의 컴퓨터학과의 GAMMA 연구그룹에서 박사후 과정을 수행하였다. 김교수의 연구 분야는 interactive computer graphics, GPGPU, haptics, geometric and physically-based modeling, robot motion planning 등의 넓은 분야를 포함하고 있고, 이 분야의 저명 논문지에 45 편 가량의 논문을 발표하였다. 김교수는 2003 년에 ACM Solid Modeling 학회에서 최우수 논문상을 수상하였고, 2004 년에 과학기술부 주관의 젊은 과학자 연구자로 지정 되었으며, 2006 년에는 Geometric Modeling and Processing 학회에서 최우수 포스터 상을 수상하였으며, 2007 년에는 Geometric and physical modeling 분야의 연구업적으로 세계적인 인명사전인 Marquis who’s who in the world 2007 에 등재되었다.

한정현 교수

고려대학교 정보통신대학
jhan@korea.ac.kr

한정현 교수는 현재 고려대학교 정보통신대학에 재직 중으로, Interactive 3D Media Lab 을 지도하고 있다. 1986 년 서울대학교 컴퓨터공학과에서 학사, 1991 년 University of Cincinnati Computer Science 에서 석사, 1996 년 USC Computer Science 에서 박사 학위를 수여하였으며, 1996 년부터 1997 년 사이에 미국 상무성 National Institute of Standards and Technology (NIST) 연구원으로 근무하였고, 1997 년부터 2004 년까지 성균관대학교 정보통신공학부에서 전임강사, 조교수, 부교수로 근무하였다. 한정현 교수는 2001 년부터 2004 년까지 산업자원부 지정 게임기술개발지원센터의 연구책임자였고, 현재는 문화관광부 지정 게임연구센터 연구책임자이다.

오경수 교수

숭실대학교 미디어학부

oks@ssu.ac.kr

2001 서울대학교 전기컴퓨터공학부 박사

2000~2002 조이멘트 개발팀장

2003~현재 숭실대학교 조교수

김진욱 박사

한국과학기술연구원(KIST), Imaging and Media Research Center

jwkim@imrc.kist.re.kr

Research Interests

Real-time Physics based Animation, Human Modeling, Haptics, Robotics

Education

1995, B.S. Mechanical Design and Production Engineering, Seoul National University

1997, M.S. Mechanical and Aerospace Engineering, Seoul National University

2002, Ph.D. Mechanical and Aerospace Engineering, Seoul National University

Jeffrey Yen

nVIDIA APAC Technical Marketing Manager

강의 내용 1

GPU를 이용한 Proximity Computation

본 튜토리얼에서는 GPGPU의 일반적인 접근법에 대한 소개와, 이를 penetration depth computation, avatar interaction in virtual environments, self-collision detection between deformable bodies, streaming AABB collision detection, arrangement computation과 같은 다양한 기하 문제에 적용하는 방법을 소개한다. 특히, 이런 문제들에 GPU의 fast rasterization, occlusion query, fast texture lookup등의 방법이 어떻게 이용되었는지를 설명한다.

본 튜토리얼에서 참조한 논문들은 다음과 같다.

Young J. Kim, Miguel A. Otaduy, Ming C. Lin and Dinesh Manocha, Fast penetration depth computation for physically-based animation, *ACM Symposium on Computer Animation*, July 2002

Young J. Kim, Gokul Varadhan, Ming C. Lin and Dinesh Manocha, Fast swept volume approximation of complex polyhedral models, *ACM Symposium on Solid Modeling and Applications*, June 2003

Yoo-Joo Choi, Young J. Kim, Myoung-Hee Kim, Rapid pairwise Intersection tests using programmable GPUs, *Visual Computer*, 22(2), 2006

Xinyu Zhang, Young J. Kim, Interactive collision detection for deformable models using streaming AABBs, *IEEE Transactions on Visualization and Computer Graphics*, 13(2), Mar/Apr,2007

Young J. Kim, Stephane Redon, Ming C. Lin, Dinesh Manocha, Jim Templeman, Interactive continuous collision detection using swept volume for Avatars, *Presence: Teleoperators and Virtual Environments* Vol. 16.2, April 2007

강의 내용 2

GPU기반의 스키닝 에니메이션

본 강의는 GPU를 이용해 대규모 군중의 실시간 스키닝 애니메이션을 구현하는 기법을 소개한다. 픽셀 쉐이더, 렌더 타켓 텍스처, 정점 버퍼를 이용한 스키닝 애니메이션은 수만 군중 각각에 대해 독립적인 동작을 부여할 수 있다. 여기에 스프라이트 등을 사용한 LOD 기법을 적용하면 실시간 애니메이션 대상 객체의 수는 수십만으로 불어날 수 있다. 한편, 대규모 군중 렌더링 알고리즘은 행동 제어 기법과도 용이하게 통합된다.

본 튜토리얼에서 참조한 논문은 다음과 같다.

I. Kang and J. Han, “Real-Time Animation of Large Crowds,” *Proc. of 5th International Conference on Entertainment Computing*, 20-22 September 2006, Cambridge, UK, pp. 382–385.

강의 내용 3

GPU를 이용한 신경망, 4진트리 서치, 이미지 피라미드 구현

1 GPU를 이용한 신경망 구현
신경망 중 하나인 MLP(Multi Layer Perceptron)은 행렬곱으로 구현할수 있다. GPU로 행렬을 곱하는 법을 설명하고 MLP의 개념에 대한 설명, MLP를 행렬곱으로 구현하는 법, MLP를 GPU로 구현하기 유리한 상황을 설명하겠다.

2 GPU를 이용한 트리의 계층적 탐색
트리의 탐색은 GPGPU에서 중요한 topic이다. 4진 트리의 계층적 탐색방법을 설명하고 이를 이용한 간접조명 렌더링, subsurface scattering등의 렌더링 알고리즘들을 demo중심으로 설명한다.

3 Image Pyramid 기타 자료구조
Image Pyramid는 영상처리, 컴퓨터 그래픽스에서 유용한 자료구조이다. 이를 이용한 변위 매핑(displacement mapping)기법을 설명한다. 이 외에도 Octree, Quad Tree를 GPU상에서 구현한 예를 보인다. function을 texture에 저장하기, random number를 생성해서 texture에 저장하기등 기본적인 GPGPU 기술들에 대해서도 설명한다.

4 GPU프로그래밍시 디버깅 팁
화면에 찍어보기, depth buffer를 사용한 if문 제거, alpha blending을 이용한 코드 단순화 등의 코딩 팁과 컴파일 에러시 대처하는 팁들에 대해 알아본다.

관련 논문

Kyoung-Su Oh, Keechul Jung: GPU implementation of neural networks, International Journal of Pattern Recognition, Vol. 37, Issue 6, Pages 1311-1314, June 2004

Oh, K., Ki, H., and Lee, C. 2006. Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid. In Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Limassol, Cyprus, November 01 - 03, 2006). VRST '06. ACM Press, New York, NY, 75-82.

기현우, 오경수. 계층적 접근을 통한 반투명한 물체의 실시간 렌더링과 적응적인 화면 보간 기법. 2006년 하계 한국게임학회 학술발표대회, p.217-224. (Best Paper Award)

강의 내용 4

GPU를 이용한 non-convex geometry의 관성행렬 연산 및 부력 시뮬레이션

강체 동역학 시뮬레이션을 수행하기 위해서는, 강체의 질량, 질량중심, 관성행렬 등의 물성치를 알아야한다. 이러한 질량특성은 시뮬레이션을 수행함에 있어 변하지 않는 경우가 많기 때문에, 시뮬레이션을 수행하기 이전 초기화 단계에서 한번 구하는 것으로 충분하다. 그러나 강체를 표현하는 geometry가 시뮬레이션을 수행함에 따라 변하는 경우, 강체의 질량특성을 매 시뮬레이션단계마다 구해야하며, 만일 질량특성을 구하는 알고리즘이 효율적이지 못할경우, 전체적인 시뮬레이션 성능을 저하시킬수있다.
본 튜토리얼에서는 GPU를 이용하여 임의의 형상으로 표현된 강체의 질량특성을 효율적으로 근사하는 방법에 대하여 생각해보고, CPU를 이용한 해석적 알고리즘과 정확도 및 연산성능 측면에서 비교해본다. 또한 제안된 알고리즘을 확장하여, 부력 시뮬레이션에 적용시키는 문제를 다룬다.

튜토리얼은 다음과 같이 구성된다.

1. 강체의 질량특성 및 동역학 시뮬레이션
2. GPU를 이용한 질량특성 연산
3. Depth-peeling을 이용한 non-convex geometry 처리
4. CPU기반 알고리즘과 성능비교
5. 부력 시뮬레이션에의 적용
6. GPU 구현상의 몇가지 이슈들

관련 논문

Jinwook Kim , Soojae Kim , Heedong Ko and Demetri Terzopoulos, "Fast GPU computation of the mass properties of a general shape and its application to buoyancy simulation", The Visual Computer. Vol.22. No.8. 2006.

강의 내용 5

NVIDIA의 new 하드웨어 (8800 시리즈)와 CUDA

Interactive Collision Detection for Deformable Models using Streaming AABBs

Xinyu Zhang and Young J. Kim

Abstract—We present an interactive and accurate collision detection algorithm for deformable, polygonal objects based on the streaming computational model. Our algorithm can detect all possible pairwise primitive-level intersections between two severely deforming models at highly interactive rates. In our streaming computational model, we consider a set of axis aligned bounding boxes (AABBs) that bound each of the given deformable objects as an input stream and perform massively-parallel pairwise, overlapping tests onto the incoming streams. As a result, we are able to prevent performance stalls in the streaming pipeline that can be caused by expensive indexing mechanism required by bounding volume hierarchy-based streaming algorithms. At run-time, as the underlying models deform over time, we employ a novel, streaming algorithm to update the geometric changes in the AABB streams. Moreover, in order to get only the computed result (i.e., collision results between AABBs) without reading back the entire output streams, we propose a streaming en/decoding strategy that can be performed in a hierarchical fashion. After determining overlapped AABBs, we perform a primitive-level (e.g., triangle) intersection checking on a serial computational model such as CPUs. We implemented the entire pipeline of our algorithm using off-the-shelf graphics processors (GPUs), such as nVIDIA GeForce 7800 GTX, for streaming computations, and Intel Dual Core 3.4G processors for serial computations. We benchmarked our algorithm with different models of varying complexities, ranging from 15K up to 50K triangles, under various deformation motions, and the timings were obtained as 30∼100 FPS depending on the complexity of models and their relative configurations. Finally, we made comparisons with a well-known GPU-based collision detection algorithm, CULLIDE [4] and observed about three times performance improvement over the earlier approach. We also made comparisons with a SW-based AABB culling algorithm [2] and observed about two times improvement.

Index Terms—Collision Detection, Deformable Models, Programmable Graphics Hardware, Streaming Computations, AABB.

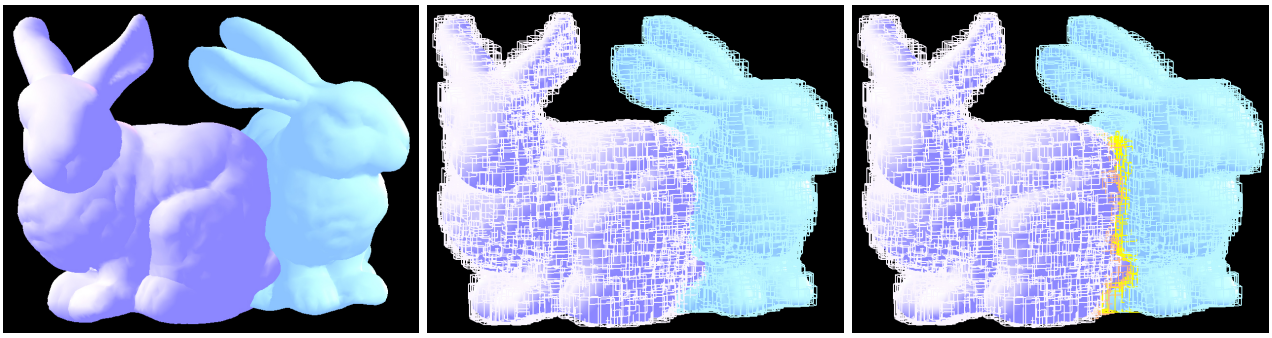
1 Introduction

The goal of collision detection is to determine whether one or more geometric objects overlap in space and, if they do, identify overlapping features, also known as *collision witness features*. Collision detection has been used for a wide variety of applications that attempt to mimic the physical presence of real world objects. The types of these applications include physically-based animation, geometric modelling, 6DOF haptic rendering, robotic path planning, medical imaging, interactive computer games, etc. As a result, many researchers have extensively studied the collision detection problems over the past two decades. An excellent survey of the filed is available in the work by Lin and Manocha [1].

At a broad level, the field of collision detection can be categorized differently depending on the nature of input models (rigid vs. deformable, linear vs. curved, or surface vs. volumetric), the existence of motion (static vs. dynamic), the type of collision query (discrete or continuous), and the type of computing resources that collision query utilizes(CPUs vs. GPUs). In principle, it is well known that the worst case computational complexity of any collision detection algorithm can be as high as quadratic in terms of the number of primitives contained in the input models. In practice, however, the actual number of colliding primitives tends to be a relatively small number. Therefore, the major efforts in most of existing collision detection algorithms have been focused on reducing the number of collision checkings between colliding primitives (e.g., triangles). Often, this goal is achieved through the use of bounding volume hierarchies (BVHs) such as axis aligned bounding box (AABB) trees, sphere trees, oriented bounding box (OBB) trees, discrete orientation polytopes (DOPs) or convex hull trees, or through the use of modern rasterization hardware.

Even though some researchers believe that collision detection is a solved problem, there are still quite a few challenges left. In particular, collision detection of deformable bodies is one of the remaining yet difficult challenges. The major difficulty of devising an efficient solution for deformable models lies in the fact that it is quite expensive to update the auxiliary collision querying structure such as BVH as the underlying model deforms over time. In order to address this issue, researchers have suggested a lazy update of BVHs [2], reduced deformation of models [3], or the use of GPUs based on image space computations [4], [5], [6], [7], [8]. However, the accuracy or the performance of the first three techniques are governed by image-space resolution and viewing directions. The efficiency of the GPU-based technique depends on the resolution of image space and it may not work well for highly deforming models that have many overlapping primitives and it often misses many colliding pairwise primitives. In-depth discussion of these challenges for collision detection of deformable objects can be also found in Teschner et al.'s work [9].

• The authors are with the department of computer science and engineering at Ewha womans university in Seoul, Korea. Email: {*zhangxy*, *kimy*}@*ewha.ac.kr*



(a) Intersecting bunny models (b) AABB streams of bunny models (c) Intersected AABBs

Fig. 1. Collision Detection using Streaming AABBs. (a) shows intersecting two bunny models (blue and cyan); (b) two bounding AABB streams (white and light blue boxes) are superimposed on the bunny models that they bound respectively; (c) highlights intersecting AABBs (shown as orange and yellow boxes). Using commodity graphics processors, our algorithm is able to find all the intersecting AABBs in the object space; using CPUs, the algorithm reports actually colliding triangles contained in the intersecting AABBs.

Recently, the streaming computation model has drawn much attention from different areas like computer graphics, image processing, geometric modelling, and even database [10]. The concept of a streaming model is not novel but it has been around for more than four decades. However, the recent introduction of powerful streaming architecture like GPUs revitalize the new era of streaming computations. This research trend is expected to continue and grow thanks to other emerging, new streaming processors like CELL processors[11], [12]. In contrast to the traditional, serial computation model like CPUs, a streaming computation model represents all data as one or more *streams*, which are defined as one or more ordered sets of the same data type. Allowed operations on streams include copying them, deriving sub-streams from them, indexing into them with a separate index stream, and performing computation on them with *kernels*. A kernel operates on entire streams, taking one or more streams as inputs and producing one or more streams as outputs. Moreover, computations on one stream element are never dependent on computations on another element [13], [14], [15], and thus can be performed in parallel with the same instructions.

1.1 Main Results

In this paper, based on the powerful concept of streaming computations, we present a novel collision detection algorithm for severely deforming objects. At a high level, the streaming computations in our algorithm can be split into three stages:

1. **Stream Setup:** As preprocess, for each deformable object, we calculate a set of axis aligned bounding boxes (AABBs) that bounds the object, and consider each set as an input stream to our collision detection algorithm.
2. **Stream Calculation:** At run-time, we perform massively-parallel pairwise, overlapping tests onto the incoming streams. Moreover, we use a streaming en/decoding strategy to get only the computed result (i.e., collisions between AABBs) without actually reading back the entire output streams
3. **Stream Update:** As the underlying models deform over time, we employ a novel, streaming algorithm to update the geometric changes in the AABB streams.

After determining overlapping AABBs at the stream calculation stage (step 2), we perform a primitive-level (e.g., triangle) intersection checking on a serial computational model, implemented using CPUs. The entire streaming computations are implemented using one of the highly successful streaming architecture of modern era, graphics processing units (GPUs).

One of the major distinctions between our algorithm and other GPU-based algorithms is that the entire pipeline of our approach performs collision detection in object space and never misses any pairwise, colliding primitives. More specifically, the main advantages of our approach include:

- **Streaming computations:** our algorithm performs massively parallel overlap tests on streaming AABBs by utilizing the high floating bandwidth of modern GPUs.
- **Tile-based rendering:** To cope with the limited memory (i.e., texture) size available in modern GPUs, our algorithm uses a tile-based rendering technique to handle a large AABB stream.
- **Hierarchical stream readback:** As a remedy for slow downstream bandwidth from GPUs to CPUs, the algorithm fetches minimal stream data from GPUs to CPUs using a hierarchical en/decoding stream readback strategy.
- **Generality of input models:** The algorithm can handle general polyhedral models and makes no assumptions about their topology and connectivity.
- **Accurate results:** The entire pipeline of our algorithm is performed in object space and can report all colliding primitives within a floating point precision of the underlying CPUs and GPUs.
- **Interactive performance:** Our extensive experiments show that the algorithm is robust and is able to report collision results of deformable models at highly interactive rates.

1.2 Organization

The rest of the paper is organized in the following manner. Section 2 surveys related work on collision detection of deformable objects. Section 3 gives a brief overview

of our approach. Section 4 describes the precomputation stage of our algorithm and section 5 presents our streaming collision detection algorithm. Section 6 provides our streaming update scheme and section 7 highlights our algorithm's performance on different benchmarks and analyzes its efficiency compared to other algorithms. In section 8, we conclude the paper and discuss a few limitations of the algorithm and suggest possible future work.

2 Previous Work

In this section, we give a brief overview of related work in collision detection for deformable objects. A more thorough, recent survey on collision detection for deformable models is available in [9].

2.1 CPU-based Algorithms

At a high level, collision detection (CD) algorithms can be classified into two categories: broad phase object-level CD and narrow phase primitive-level CD. For the broad phase CD, algorithms based on sweep-and-prune have been proposed in I-COLLIDE [16], V-COLLIDE [17] and SWIFT/SWIFT++ [18]. However, these techniques are designed mainly for rigid models. It is not clear whether they can handle large deformable models at interactive update rates.

For the narrow phase of CD algorithms, a variety of techniques have been presented such as the use of BVHs, geometry reasoning, algebraic formulations, space partitions, parse methods and optimization techniques [1], [19]. In particular, BVHs have been proven efficient and successful in collision detection. Examples of typical bounding volumes used in the literature are AABBs [2], [20], [21], spheres [22], [23], OBBs [24], DOPs [25]. By introducing AABB trees, the accurate algorithm suggested in [2] for deformable models has special advantages for slight deformations, because refitting AABB trees is much faster than rebuilding them. Recently, by combining BVHs with a cache-oblivious layout, the query time of collision detection for rigid bodies can be reduced significantly [26].

2.2 GPU-based Algorithms

CD algorithms based on GPUs can be classified into two different categories: image space- and object space-based approaches. The former approach exploits the powerful rasterization capability available in modern GPUs to perform intersection tests between object primitives in image space. The effectiveness of the approach is often limited by the image space resolution. The latter approach utilizes the high floating point bandwidth and programmability of GPUs and all the computations are performed in object space and thus are limited by the floating point precision of GPUs.

2.2.1 Image Space-based Techniques

The pioneering work of image-based collision detection has been introduced by [27] for convex objects. In this method, two depth layers of convex objects are rendered into two depth buffers and an interval between the smaller depth value and the larger depth value at each pixel is used for interference checking [9]. The work by [28] is able to detect collision for arbitrary-shaped objects, but the maximum depth complexity is limited and object primitives must be pre-sorted.

For cloth simulation, the first image-based collision detection algorithm has been presented in [29]. The algorithm generates an approximate representation of an avatar by rendering it from front to back and reports penetrating cloth particles. [8] uses a voxel-based AABB hierarchical method for highly compressed models.

The algorithm for virtual surgery operations [30] has been introduced to detect intersections between a surgical tool and deformable tissues by rendering the interior of the tool based on the selection and feedback mechanism available in OpenGL. However, selection and feedback can cause stalls in the graphics pipeline because it relies on the use of expensive picking matrices, thus resulting in a worse performance. The algorithms based on distance field computations [31] can report various proximity information such as interference detection, separation distance and penetration depth. In [32], they have presented a method to detect an edge/surface intersection in multi-object environments.

Layered Depth Images (LDIs) is used in [33] to approximately represent objects' volume and perform CD for models with closed surfaces. A method using GPUs-assisted voxelization is introduced in [34]. The approaches utilizing hardware-supported visibility queries [4], [6] have been proposed to significantly improve the efficiency of collision culling. However, the accuracy is governed by the image-space resolution and viewing directions. The issue of accuracy has been resolved by their improved algorithm, R-CULLIDE [5], but its performance is still governed by the resolution and viewing directions. A more recent algorithm [35] precomputes a chromatic decomposition of a model into non-adjacent primitives using an extended-dual graph. However, it requires a fixed connectivity for a model and can not be applicable to models with an arbitrary connectivity.

2.2.2 Object Space-based Techniques

Utilizing the high floating point bandwidth and programmability of modern GPUs, a hierarchical collision detection method for rigid bodies using balanced AABB trees has been devised in [36]. The algorithm maps AABB trees onto GPUs and performs a breadth-first search on the trees. During the traversal of hierarchy, occlusion query is used to count the number of overlapping AABB pairs and recursive AABB overlapping tests in object space is implemented using GPUs. However, traversing hierarchical structure on GPUs turns out to be a huge overhead for GPUs and this algorithm does not work at interactive rates. Moreover, the algorithm is designed only for rigid models, not for deformable models. A similar work using filtering operation has been suggested by [37]. [38] has proposed a GPU-based method to perform self-intersections between deformable objects. This method also fully uti-

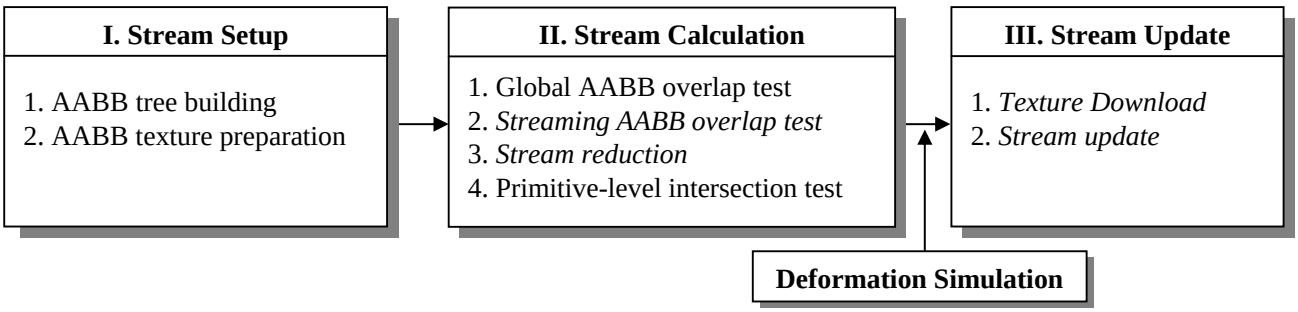


Fig. 2. The Streaming Collision Detection Pipeline. Stage I performs AABB stream setups. Stage II executes massively-parallel overlap test between AABB streams. Stage III updates the AABB streams as the underlying model deforms. The steps associated with streaming computations are *italicized*.

lizes the floating point bandwidth and programmability of modern GPUs but the input models are limited to around $1K$ triangles.

3 Algorithm Overview

The pipeline of our algorithm involves the following three steps to perform streaming collision detection between two deformable objects. The first step is performed as preprocess whereas the last two steps at run-time.

1. **Stream Setup:** (also see Section 4)
   (a) As preprocess, the 1D stream, $\mathcal{S}^X$, of AABBs is pre-built by building an AABB tree of a given model X in a top down manner such that each leaf node in the tree respectively corresponds to a unique element, $\square_i^X$, in $\mathcal{S}^X$ (i.e., $\mathcal{S}^X = \cup\square_i^X$). $\square_i^X$ may contain more than one triangle but each triangle belongs to a unique $\square_i^X$ in $\mathcal{S}^X$.
   (b) On GPUs, each $\square_i^X$ requires two texels to represent the bound (min/max) of an AABB and, as a result, $\mathcal{S}^X$ is stored at two floating point 1D textures $\{\mathcal{T}_{min}^X, \mathcal{T}_{max}^X\}$.
2. **Stream Calculation:** (also see Section 5)
   (a) **Global AABB Overlap Test:** We check for an intersection between the global AABB pairs of models. We further continue the following steps only if there occurs an intersection between the global bounding boxes.
   (b) **Streaming AABB Test:** All possible pairwise combinations between $\square_i^X$ and $\square_j^Y$ from models X, Y are examined for their possible overlap. This process is enabled by rendering a two dimensional rectangle onto an off-screen buffer while invoking a fragment shader to actually perform an AABB overlap test. More specifically, the rectangle is textured periodically with two 1D textures $\{\mathcal{T}_{min}^X, \mathcal{T}_{max}^X\}$ in vertical direction and two 1D textures $\{\mathcal{T}_{min}^Y, \mathcal{T}_{max}^Y\}$ in horizontal direction. The Boolean results of the above computation are stored at the off-screen buffer.
   (c) **Stream Reduction:** Our algorithm encodes the Boolean results into a packed representation to speed up the reading performance from GPUs back to CPUs. Based on a multi-pass rendering technique on off-screen buffers, we employ a hierarchical readback strategy that is a variant of [38]. The hierarchical readback structure is constructed in a bottom-up manner such that a single pixel in a higher level off-screen buffer encodes the Boolean results of a group of neighboring pixels in a lower level off-screen buffer. When we decode the Boolean results, we traverse the hierarchy in a top-down manner.
   (d) **Primitive-level Intersection Test:** Exact primitive-level intersection tests are performed on CPUs only for overlapping $\square_i^X, \square_j^Y$ pairs. We use a standard triangle/triangle intersection test such as [39]. This test does not rely on streaming computations.
3. **Stream Update:** (also see Section 6)
   As the underlying models X, Y deform, their associated AABB streams $\mathcal{S}^X, \mathcal{S}^Y$ should be updated. In our case, each of $\mathcal{S}^X, \mathcal{S}^Y$ is stored at two 1D min/max textures (e.g., $\mathcal{T}_{min}^X, \mathcal{T}_{max}^X$ for $\mathcal{S}^X$). Each texel in $\mathcal{T}_{min}^X$ (or $\mathcal{T}_{max}^X$) represents the lower bound (or upper bound) of associated geometry. We update $\mathcal{T}_{min}^X$ (or $\mathcal{T}_{max}^X$) by rendering a single 1D line and invoking a simple fragment program that performs pixel-wise min and max operations.

4 Stream Setup

An input stream $\mathcal{S}^X$ to our CD algorithm consists of an ordered set of AABBs $\square_i^X$'s that bound a given deformable model X. In this section, we explain how we initially create $\mathcal{S}^X$ and later describe in Section 6 how we update $\mathcal{S}^X$ as X deforms.

4.1 AABB Stream Construction

We start building an AABB tree using the method suggested in [2]. An AABB tree is built by recursively subdividing the given model in a top down manner. Each leaf AABB node in the tree contains no more than a user-provided, number of triangles and each triangle belongs to only one leaf AABB node. Fig. 3-(a) illustrates the structure of a typical AABB tree for a model X. Parts of actual model geometry (e.g., triangle list) are kept in each leaf node. Then, each leaf node produces one AABB, $\square_i^X$, and we collect these $\square_i^X$'s to form an AABB stream $\mathcal{S}^X$. Note that $\cup\square_i^X$ bounds X, but $\square_i^X$ is not necessarily

disjoint to each other; i.e., $\square_i^X \cap \square_j^X$ may not be empty. Besides, we also maintain a vertex list for all the vertices contained in $\square_i^X$. As we will see in Sections 4.2.2 and 6, these vertex lists are used to accelerate updating AABB textures on GPUs.

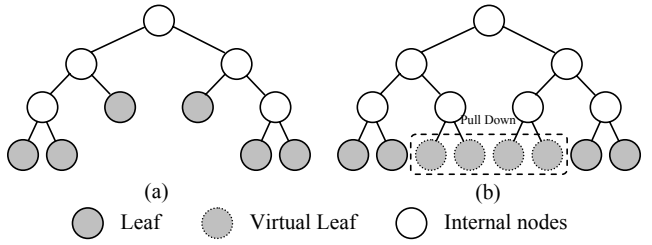


Fig. 3. Pulling Down Operation. (a) An original AABB tree before adjustment; (b) A complete binary AABB tree after adjustment by adding virtual leaves using pulling down operations.

In order to adequately map $\mathcal{S}^X$ to modern GPUs' architecture, however, we adjust each AABB tree to form a complete binary tree by generating extraneous, *virtual nodes* for a leaf level AABB node that does not exist in the original tree. Although latest GPUs support textures in non-power-of-two, power-of-two texture is still required for the stream reduction technique, described in Section 5.3. We illustrate the adjustment scheme in Fig. 3. To generate virtual leaves, we *pull down* a leaf node that is not located at a bottom level and create its virtual children. The values of the virtual children nodes are copied from their parent. We simply call these virtual children nodes, *virtual nodes*.

After the adjustment, at each level in the hierarchy, a complete binary tree has 2^d nodes including virtual ones, where d is the depth of the given level. The leaf level node of such a complete binary AABB tree corresponds to $\square_i^X$ and their union forms an AABB stream $\mathcal{S}^X$. $\mathcal{S}^X$ is represented as textures on GPUs.

4.2 Mapping AABB Streams to GPUs

The target streaming architecture on which we wish to implement our CD algorithms is the most successful and popular streaming architecture of all time, GPUs. Compared to CPUs, the floating point performance of GPUs has increased dramatically over the last four years. It has been reported that GPUs' performance doubles every six month. Moreover, GPUs has a full programmability that supports vectorized floating point operations at a quasi full IEEE single precision. The raw speed, increased precision, and rapidly expanding programmability make GPUs attractive platform for general purpose computation. On GPUs, stream data are subject to be bound to textures [40]. Now we explain how to prepare such textures on GPUs to represent AABB streams.

4.2.1 1D AABB Textures

Modern GPUs can support four color channels RGBA for textures where each color channel can be a floating point number. As a result, in order to store the bound (min/max) of each AABB element $\square_i^X$ contained in a AABB stream $\mathcal{S}^X$ at textures, we require two 1D textures $\mathcal{T}_{min}^X, \mathcal{T}_{max}^X$; let us call these textures $\mathcal{T}_{min}^X, \mathcal{T}_{max}^X$ *AABB textures*. More precisely, for each $\square_i^X$, its lower bound $(x_{min}, y_{min}, z_{min})$ is stored at one texel in $\mathcal{T}_{min}^X$ and its upper bound $(x_{max}, y_{max}, z_{max})$ at one texel in $\mathcal{T}_{max}^X$. Fig. 4-(a) illustrates a brief description of this procedure.

Meanwhile, for each AABB stream $\mathcal{S}^X$, we also prepare its corresponding 1D stencil array whose dimension is the same as the associated AABB texture, $\mathcal{T}_{min}^X$ or $\mathcal{T}_{max}^X$. Each element in the stencil array indicates whether corresponding AABB node (i.e., $\square_i^X$) is real or virtual: following the common OpenGL convention, zero denotes a virtual node such that the pairs with zeroed $\square_i^X$ will not be further considered being update in frame buffer after the AABB overlap test. However, notice that for a pair of virtual AABB nodes (say $\square_i^X, \square_{i+1}^X$) that share a common real AABB parent node, at least one of them should be considered for an overlapping test, but not necessarily both since they contain the same AABB value. Therefore, we mark the first virtual AABB node as one while keeping the second one as zero. For example, in Fig. 4-(a), the third and fourth nodes are virtual nodes sharing the same, real parent node in Fig. 3-(a). In this case, the third node will be marked as one while the fourth will be as zero. The created 1D stencil array fills up the stencil buffer that will be used to prevent the frame buffer from unnecessary update after the fragment processing in GPUs which actually performs an AABB overlapping test (see Section 5.2).

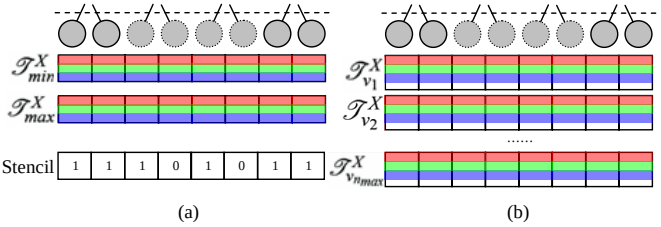


Fig. 4. Preparation of AABB Textures. (a) 1D AABB textures storing the upper and lower bounds of AABBs and a stencil array; (b) 1D vertex textures for unique vertices contained in AABB nodes.

4.2.2 Vertex Textures

As will be explained in Section 6 in detail, as a model X deforms, its geometry as well as its AABB stream $\mathcal{S}^X$ should be updated. The geometry of X, in our case, is represented using a list of triangles. These triangles are partitioned into a separate group and each group is bounded by different $\square_i^X$'s. Therefore, $\square_i^X$ can contain more than a single triangle. To update $\square_i^X$ under deformation, we need to store the triangles at separate textures, called *vertex textures*[1].

Let us denote n_{max} as the maximum number of triangles that any $\square_i^X$ can contain; i.e., $n_{max} = \max(|\square_i^X|), \forall i$. Then, we prepare n_{max} 1D vertex textures whose size is the same as that of an AABB texture. For example, as illustrated in Fig. 4-(b), the ith vertex contained in the

1. The vertex texture in our paper is a different notion from nVIDIA's vertex texture

jth AABB node is stored at the jth texel of the ith vertex texture; however, here, we use only RGB channels of the jth texel. The alpha channel (A) is reserved for representing an empty texel. In other words, if the size of $\square_i^X$, say $n_i = |\square_i^X|$, is smaller than n_{max}, we set the alpha channels of texels between $n_i + 1$ and n_{max} to zero and set the rest as one.

In practice, working with 1D textures on GPUs turns out to be less efficient than 2D textures. The main reasons are: (a) GPUs are equipped with 2D frame buffers, so 2D textures tend to be updated more rapidly than 1D textures [40] and (b) the maximum number of possible multiple 1D textures is limited by underlying hardware. Thus, we pack n_{max} 1D vertex textures into a 2D texture whose dimension is $2^{d_{max}} \times n_{max}$ where d_{max} is the height of the complete binary AABB tree.

5 Streaming Collision Detection

At run-time, our streaming CD algorithm reports all intersecting triangles between two deforming models. This run-time process can be subdivided into three stages: global AABB overlap test, streaming AABB overlap test, and fast readback of colliding results.

5.1 Global AABB Overlap Test

Let us denote the global AABBs that bound the entire models X and Y as $\square_G^X, \square_G^Y$, respectively. As trivial rejection, if $\square_G^X \cap \square_G^Y = \varnothing$, we immediately terminate the algorithm and report no collision between X and Y; otherwise, we continue the next steps described in Section 5.2. This test does not require a streaming computation and thus can be simply implemented on CPUs.

5.2 Streaming AABB Overlap Tests

The process of checking for overlaps between two AABB streams $\mathcal{S}^X, \mathcal{S}^Y$ proceeds in two steps:

1. Stream Pairing: we represent all possible pairwise combinations between $\square_i^X, \square_j^Y$ in $\mathcal{S}^X, \mathcal{S}^Y$ by texturing a squared rectangle with $\mathcal{T}_{min}^X, \mathcal{T}_{max}^X$ in vertical direction and with $\mathcal{T}_{min}^Y, \mathcal{T}_{max}^Y$ in horizontal direction.
2. Elementary AABB Overlap Test: rendering the textured rectangle invokes a fragment program on GPUs for each pixel that performs a simple interval overlapping test between $\square_i^X, \square_j^Y$.

More precisely, for the step (1), we render a $2^{d_{max}^X} \times 2^{d_{max}^Y}$ rectangle, where d_{max}^X and d_{max}^Y are, respectively, the heights of the AABB tree X and Y that were precomputed as preprocess. We texture-map the rectangle with four 1D textures $\mathcal{T}_{min}^X, \mathcal{T}_{max}^X, \mathcal{T}_{min}^Y, \mathcal{T}_{max}^Y$, as illustrated in Fig. 5. $\mathcal{T}_{min}^X, \mathcal{T}_{max}^X$ from X are used to periodically texture the rectangle in vertical direction and $\mathcal{T}_{min}^Y, \mathcal{T}_{max}^Y$ from Y in horizontal direction.

For the step (2), rendering the above textured rectangle invokes a same fragment program on every pixel in a SIMD fashion on GPUs. The fragment program performs an elementary overlap test for the corresponding pixel, which represents a pair of AABBs, $\square_i^X, \square_j^Y$. The overlap test is a simple interval overlap test along three principal axes of an AABB. However, as explained in Section 4.2.1, we prevent some fragments from being updated in the frame buffer since their associated AABB pairs are virtual nodes. We use two 1D stencil arrays from model X and Y to set up the stencil buffer to disable unnecessary update of frame buffer after the fragment processing for those pairs. For example, as illustrated in Fig. 5, the white blocks marked as '-' represent prevented AABB pairs (PAPs) by the stencil buffer.

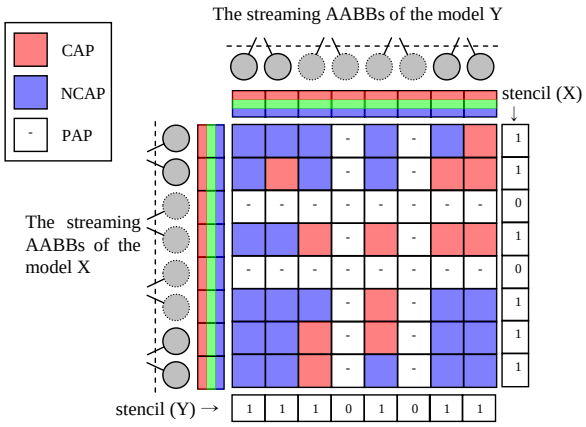


Fig. 5. AABB pair overlap tests on GPUs. The red blocks represent colliding AABB pairs (CAPs) and the blue blocks represent non-colliding AABB pairs (NCAPs). The white blocks marked as '-' represent prevented AABB pairs (PAPs) by the stencil buffer that is defined by the stencil array of the model X (the right column) and that of the model Y (the bottom row). The left AABB textures correspond to an AABB stream $\mathcal{S}^X$ of the model X and the top AABB textures to $\mathcal{S}^Y$ of the model Y.

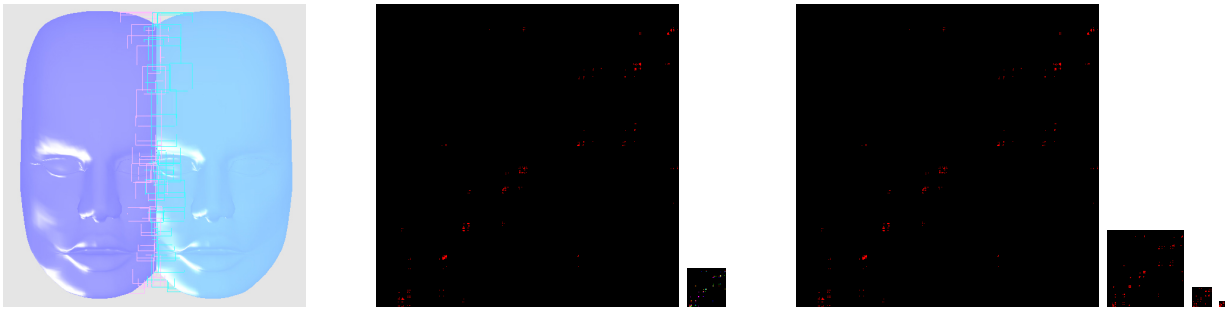
```
void streamingAABBTest ( float uvA: TEXCOORD0,
                         float uvB: TEXCOORD1,
                         out float4 color: COLOR,
                         uniform sampler1D minTextureA,
                         uniform sampler1D maxTextureA,
                         uniform sampler1D minTextureB,
                         uniform sampler1D maxTextureB)
{
  float3 aabbMinA = (float3) tex1D(minTextureA, uvA).xyz;
  float3 aabbMaxA = (float3) tex1D(maxTextureA, uvA).xyz;
  float3 aabbMinB = (float3) tex1D(minTextureB, uvB).xyz;
  float3 aabbMaxB = (float3) tex1D(maxTextureB, uvB).xyz;

  if(aabbMinA.x > aabbMaxB.x || aabbMaxA.x < aabbMinB.x ||
     aabbMinA.y > aabbMaxB.y || aabbMaxA.y < aabbMinB.y ||
     aabbMinA.z > aabbMaxB.z || aabbMaxA.z < aabbMinB.z )
     discard; //no overlap

  color= float4(1.0, 0.0, 0.0, 0.0);
}
```

TABLE I

ELEMENTARY AABB OVERLAP TEST IN CG

The code implements a simple interval overlap test between AABB textures addressed by uvA, uvB, and returns its Boolean result as color.

The rendering result of the fragment program contains a Boolean result of collision between a pair of AABBs: colliding AABB pairs (CAPs) and non-colliding AABB pairs (NCAPs). For example, in Fig. 5, the red blocks represent CAPs and the blue ones represent NCAPs. Note that PAPs are always NCAPs. Table I shows a simple, fragment



(a) Two intersecting models (b) Reduced hierarchical encoding (c) Hierarchical encoding

Fig. 6. Snapshots of streaming AABB overlap tests on GPUs and the hierarchical readbacks. (a) Snapshot of two intersecting models with CAPs in wire-framed boxes. (b) Reduced hierarchical readback. Left: collision result of AABB pairs stored at an off-screen buffer P_0; the red pixels indicate the CAPs. Right: encoded off-screen buffer by a 8×8 kernel P_1'. (c) Hierarchical readback. Left most: the same AABB collision results P_0. Right three images: hierarchically encoded off-screen buffers P_1, P_2, P_3.

program in Cg performing an elementary AABB overlap test for each pixel.

In Fig. 6, we show snapshots of our CD algorithm in action. In Fig. 6-(a), two deformable models X and Y intersect with each other. The left image in Fig. 6-(b) is a snapshot of a textured rectangle, where the red pixels are CAPs and the black ones are NCAPs.

5.3 Stream Reduction

5.3.1 Hierarchical Readback

One of the limitations to map the concept of streaming computations to GPUs is the limited bandwidth of data transmission between GPUs and CPUs, especially reading the stream data from GPUs back to CPUs, also known as *downstream bandwidth*. Therefore, when mapping streaming computations to GPUs, we need to carefully design the algorithm in such a way that the number of readbacks from GPUs should be minimized. In general, the readback time increases linearly in proportion to the size of a readback. For example, on nVIDIA GeForce 6800 with PCI express bus architecture, it takes 96.97 ms to read the entire contents of a 2048×2048 floating point color buffer whereas it takes only 6.58 ms to read a 512×512 color buffer [41].

To speed up the readback performance, a straightforward idea will be to split a readback buffer into smaller ones and read only relevant parts. A more intelligent way is to read the data in a hierarchical fashion, assuming that the relevant data is grouped together. In practice, collision results show a spatial coherence; i.e., colliding triangles tend to be in close proximity with one another. Based on this observation, in [38], a hierarchical en/decoding strategy has been suggested to speed up the readback performance. We use a variant of this approach.

In our readback scheme, we consecutively reduce the size of output stream in a hierarchical fashion. Initially, the 2D output stream whose element represents a Boolean collision result is stored at a textured rectangular buffer P_0, i.e., off-screen color buffer, as shown in Fig. 5. We render this buffer P_i to another 4×4 times smaller buffer P_{i+1} until the size of the rendered buffer P_{i+1} reaches a certain value; in practice, we use three layers of off-screen buffers to encode the original off-screen color buffer (i.e., $i_{max} = 3$). We encode a set of 4×4 adjacent pixels in a higher level buffer P_i as a single pixel in a lower level buffer P_{i+1}.

When we decode the encoded streaming CD result, we move backward from P_{i+1} to P_i, starting from reading the entire contents of $P_{i_{max}}$. Since each pixel in P_{i+1} indicates the contents of 4×4 pixels in P_i, we read only relevant portion of pixels in the hierarchy. In practice, this approach works quite well when the ratio ,η , of CAPs to the number of all AABB pairs is relatively small (say, 0.095% in our implementation). However, as the ratio increases, we might as well reduce the level of hierarchy. In fact, we maintain only a single level in the hierarchy. More precisely, if η is smaller than a certain threshold, we perform the hierarchical encoding strategy with $i_{max} > 1$. Otherwise, we encode P_0 into P_1' whose size is 8×8 times smaller than P_0, but with $i_{max} = 1$. As a result, a single pixel in P_1' encodes 8×8 adjacent pixels in P_0.

Our experiment has shown that a variable hierarchical method can provide a better readback timing than the fixed hierarchy. Fig. 6 shows snapshots of the contents in the hierarchical encoding at run-time. The left images in 6-(b) and 6-(c) are both P_0. The right image in 6-(b) is P_1', and the right three images in 6-(c), from left to right, denotes P_1, P_2, P_3, respectively (we also refer the readers to see the accompanying video).

5.3.2 Analysis

Now, we give a brief analysis of the variable hierarchical en/decoding strategy. Fig. 7 shows the performance of the variable hierarchical strategy. The x axis denotes the ratio η and the y axis is the encoding/decoding timing for different η's.

The hierarchical readback consists of two steps: encoding P_0 into three layers P_1, P_2, P_3 followed by decoding all the layers backwards. In Fig. 7, the readback time is a linear function of η. Moreover, encoding timing is almost constant if the size of P_0 is fixed whereas the decoding time is also a linear function of η. However, the encoding or decoding time of the reduced hierarchical readback

scheme takes a constant time since there is only one level of hierarchy.

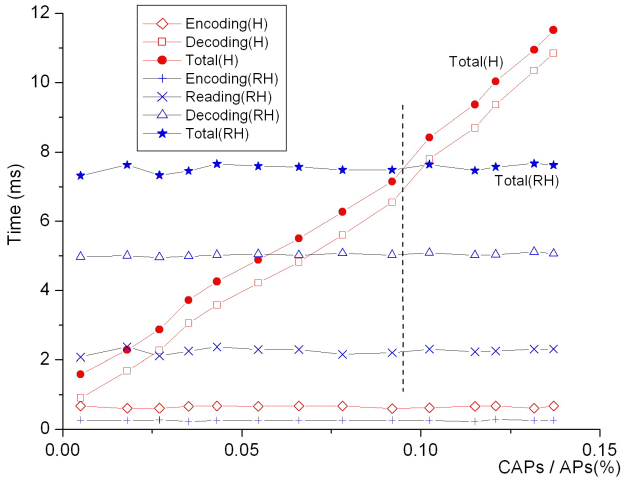


Fig. 7. Performance of variable hierarchical readback. Encoding(H) and Decoding(H) are the timings of encoding and decoding using the hierarchical readback scheme with three levels. Total(H) is their sum. Encoding(RH), Reading(RH) and Decoding(RH) are the timings of encoding, reading P_1', and decoding using the reduced hierarchical readback scheme with a single level. Total(RH) is their sum.

In order to improve the readback performance, when η reaches a threshold indicated by the dotted line (0.095%) in Fig. 7, we switch from a hierarchical scheme to a reduced one. As a result, in our implementation, we can always read 2048×2048 data in less than 7.2 ms. Note that an optimal threshold value η should be recalculated for different sizes of P_0.

5.4 Primitive-level Intersection Test

Once we find CAPs (say $\square_i^X, \square_j^Y$), we perform a triangle/triangle intersection checking for all pairs of triangles contained in $\square_i^X, \square_j^Y$. We do not use streaming computations for this primitive-level checking unlike [36], [38]; instead, we use a classical, CPU-based method suggested by Möller [39]. The main reason why we use a serial, CPU-based method is that a set of triangles contained in CAPs can be arbitrary such that the setup time to map the potentially colliding triangles to textures can be quite expensive [38] and this process can not be executed as preprocess. Moreover, since the number of triangles contained in CAPs is relatively small, the overhead of streaming computations for a primitive-level intersection checking can not be compensated for.

5.5 Handling Large Models

The maximum texture size specified by GPUs limits the maximum resolution of an AABB texture [14]. On modern GPUs like nVIDIA GeForce 7800, for example, the maximum texture size is 4096×4096. That means that the maximum height of the complete binary AABB tree in Section 4.1 is limited to 12. To overcome this limitation, we propose a tile-based method to render a large rectangle with as many as $\max\{1, d_{max}^A - 12\} \times \max\{1, d_{max}^B - 12\}$ texturing tiles. Each tile is rendered and read back independently. However, since the typical size of GPU memory is limited to 256MB or 512MB, it is difficult to allocate these many textures at one time. Thus, we create only a single rendering target (i.e., off-screen buffer) that can be used by all the tiles. In theory, the tile-based rendering should perform linearly with a respect to the number of tiles. However, due to the GPU memory/cache coherence and parallelism efficiency [42], the performance of tile-based rendering in our test increases super-linearly. We anticipate that this issue can be resolved in the future release of new GPU architectures and drivers.

6 Stream Update

```
void streamUpdate ( float2 uv: TEXCOORD0,
                    out float4 color0: COLOR0,
                    out float4 color1: COLOR1,
                    uniform samplerRECT vertexT,
                    uniform float nmax)
{
  float3 vmin = float3(1.0, 1.0, 1.0);
  float3 vmax = float3(0.0, 0.0, 0.0);
  float3 v;

  for (int row=0; row<nmax; row++)
  {
    v = texRECT(vertexT, uv + float2(0.0, row)).xyz;
    vmin = min(vmin, v);
    vmax = max(vmax, v);
  }
  color0 = float4(vmin, 0.0);
  color1 = float4(vmax, 0.0);
}
```

TABLE II
STREAM UPDATE USING A MIN/MAX OPERATION IN CG
The code implements a simple min/max operation for a 2D vertex texture, and returns its result as colors.

As a model deforms, its associated AABB stream should reflect the deformation. An earlier BVH-based algorithm such as [2] refits an entire AABB tree after each deformation step. Our approach does not maintain such hierarchy but has only an AABB stream (i.e., $\mathcal{S}^X$) corresponding to the leaf nodes in AABB hierarchy. In our case, this stream is mapped to AABB textures and the underlying triangle geometry is mapped to vertex textures. For a given model X, our goal is to store the element-wise minimums of vertex textures ($\mathcal{T}_i^X, 1 \le i \le n_{max}$) at $\mathcal{T}_{min}^X$ and the element-wise maximums at $\mathcal{T}_{max}^X$; i.e.,

$$\begin{aligned} \mathcal{T}_{min}^X[k] &= \min_{1 \le i \le n_{max}} \mathcal{T}_i^X[k] \\ \mathcal{T}_{max}^X[k] &= \max_{1 \le i \le n_{max}} \mathcal{T}_i^X[k] \\ \text{and} \quad & 1 \le k \le 2^{d_{max}} \end{aligned} \tag{1}$$

In order to perform element-wise min/max, we pack n_{max} of individual 1D vertex texture, $\mathcal{T}_i^X$, into a separate column i in one 2D vertex texture whose size is $n_{max} \times 2^{d_{max}}$. Then, we render a single line and texture map it with the 2D vertex texture, while redirecting its output to two different render targets (for min and max, respectively) using multiple render target technique (MRT) available in OpenGL 2.0 and DirectX 9.0. A fragment program is invoked to actually perform column-wise min and max operations for the 2D vertex texture, as shown in Tab. II. Utilizing MRT, we can calculate min and max concurrently. After rendering is completed, $\mathcal{T}_{min}^X$ and $\mathcal{T}_{max}^X$

are respectively stored in the first and second render targets. The first render target is named as COLOR0 and the second render target as COLOR1 in the code.

7 Experimental Results and Analysis

7.1 Implementation

We implemented the entire pipeline of our algorithm on a PC equipped with a Intel Dual Core 3.4GHz Processor, 2.75GB of main memory and nVIDIA GeForce 7800 GTX GPUs with 512M video memory and PCI-Express interfaces. As a choice for programming languages, we used Microsoft Visual C++, nVIDIA's Cg shading language with vp40 and fp40 profiles, and OpenGL 2.0 graphics library. Because no GPUs currently provide double-precision floating point numbers or double-precision arithmetic, for the purpose of fair comparison, we have used 32-bit floating point for both CPU and GPU computations throughout the entire paper. However, even though the storage format of floating point in GPU is the same as the IEEE 754 standard, the arithmetic operation might produce slightly different results.

7.2 Collision Benchmarking Scenario

In order to measure the performance of our streaming CD algorithm, we employ six different deformable bodies whose triangle count ranges from $15K$ to $50K$ triangles (as shown in Table III). Such complex models can model deformable simulation in most of applications. Also, we apply two different kinds of deformations to the deformable bodies to simulate their collisions:

- **Wavy Deformation:** Random bumps with wave functions are generated on the surfaces of the deformable bodies and the bumps are propagated to the entire surfaces. In our scenarios, sine and cosine functions are used to simulate wavy bumps. Local potential energy introduced will be damped out while the energy is being propagated to neighboring parts of the surface.
- **Pulsating Deformation:** Vertex positions periodically move up and down in the direction of a surface normal (bulging effect). Any random pulsating function can be chosen at user's discretion.

7.3 Performance Analysis

The statistics and some snapshots of our experiments are shown in Table III and Fig. 8-Fig. 9.

Table III shows the performance statistics of our algorithm (all timings were measured in ms). The first four columns indicate the triangle count of the tested models, the number of CAPs, the number of the potential colliding triangle pairs (PCTPs) and the number of actual colliding triangle pairs (CTPs), respectively. The following five columns are the timings of streaming AABB overlap tests, readback (encoding/decoding) by streaming reduction, primitive-level (i.e., triangle-level) intersection tests, texture download and stream update (i.e., AABB textures update). The last column indicates the total time including all the steps used in our algorithm.

In Fig. 8, we used two deforming torii to simulate three different configurations of deformations commonly occurring in many applications: interlocking bodies, touching bodies and merging bodies. Each torus consists of $15K$ triangles. The timing in our experimental results shows that the CD checking can be executed at the rates of 60-80 frames per second (FPS) for the interlocking torii (Fig. 8-(a)), 90-100 FPS for the touching torii (Fig. 8-(b)) and around 30 FPS for the merging torii (Fig. 8-(c)). For these benchmarks, the wave deformation was adopted to simulate the deformation.

We also have tested our algorithm with other models. The snapshots of these benchmarks are highlighted in Fig. 9. For these benchmarks, the pulsating deformation has been adopted. We refer to the accompanying video for a better visualization of our experiments. The experimental results have shown that our algorithm can be applied to highly real-time applications that need to return all colliding triangle pairs, accurately.

We analyze the time complexity of each step in our algorithm. The streaming AABB overlap test takes a constant time when the scenario is given. The hierarchical readback takes a linear time in terms of the number of CAPs when it is less than the precalculated threshold, and takes a constant time when it is greater than the threshold, as shown in Fig. 7. The primitive-level intersection test takes a quadratic time in terms of the number of triangles in AABBs. However, because each AABB $\square_i^X$ contains $n_i (n_i < n_{max})$ primitives where n_{max} is a fixed small constant number, the primitive-level intersection test is sensitive to the number of PCTPs in practice. The stream update takes always a constant time. As a result, the entire algorithm is sensitive to the number of PCTPs or the number of CAPs in practice.

7.4 Comparisons with Other Approaches

Collision detection is well-studied in the literature and a number of algorithms and public domain systems are available. However, none of the earlier algorithms provide the same capabilities or features as our streaming CD algorithm does. We compare some of the features of our approach with the earlier algorithms.

7.4.1 CPU-Based Algorithms

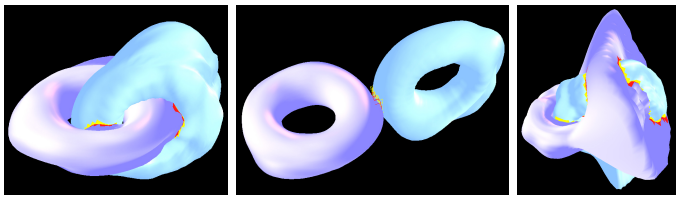
BVHs have been widely used for CD algorithms such as I-COLLIDE, RAPID, V-COLLIDE, SWIFT, SOLID 1.0, QuickCD, etc. However, these algorithms are designed for rigid bodies. In SOLID [2], AABB trees are used to handle collisions for deformable bodies. However, its timing statistics have showed that updating the entire AABB tree can be a bottleneck of the algorithm, because it uses a lazy refitting method to recalculate the new bounding box of each leaf AABB node and recalculate internal AABB nodes in a bottom-up manner. Our approach also uses AABB as a bounding volume, but does not keep any hierarchy at run-time unlike [2] such that we do not need to update

	nTris	nCAPs	nPCTPs	nCTPs	Overlap Test	Readback	Tri Test	Texture Download	Stream Update	Total
1	15000×2	475	129884	429	0.10	0.23/1.38	15.19	2.43	0.70	20.03
2	15000×2	167	30108	214	0.11	0.22/0.59	3.77	2.46	0.71	7.86
3	15000×2	781	204848	748	0.12	0.21/1.94	23.94	2.45	0.69	29.35
4	15000×2	743	41921	473	0.11	0.24/2.52	5.70	3.62	1.00	13.19
5	20000×2	1372	166600	865	0.10	0.24/3.24	22.27	5.45	1.18	32.49
6	50000×2	306	233104	416	0.11	0.24/0.75	26.28	10.34	1.48	39.20

TABLE III

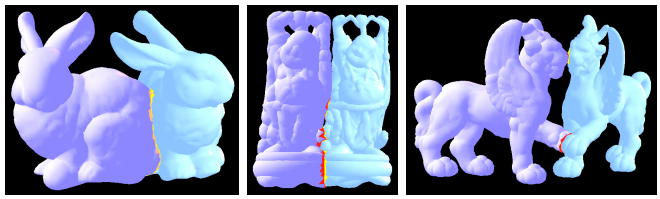
PERFORMANCE STATISTICS OF OUR ALGORITHM.

The benchmark models from 1 to 6 are interlocking torii, touching torii, merging torii, bump bunnies, happy buddhas and intimate animals. The first four columns: the triangle count of models, the number of CAPs, the number of potentially colliding triangle pairs (PCTPs) in CAPs, and the number of actual colliding triangle pairs (CTPs). The next four columns of timings measured in *msec*: streaming AABB overlap tests, readback by streaming reduction, primitive-level intersection tests and stream update. The last column: the total CD time.



(a) (b) (c)

Fig. 8. Benchmark Set I: Each torus consists of $15K$ triangles and the wave deformation is adopted to simulate the deformation. (a) Interlocking torii (60-80 FPS). (b) Touching torii (90-100 FPS). (c) Merging torii (25-30 FPS).



(a) (b) (c)

Fig. 9. Benchmark Set II: The pulsating deformation is adopted to simulate the deformation. (a) Bump Bunnies ($15K$ triangles/each, 50-60 FPS). (b) Happy Buddhas ($20K$ triangles/each, 25-40 FPS). (c) Intimate Animals ($50K$ triangles/each, 20-35 FPS).

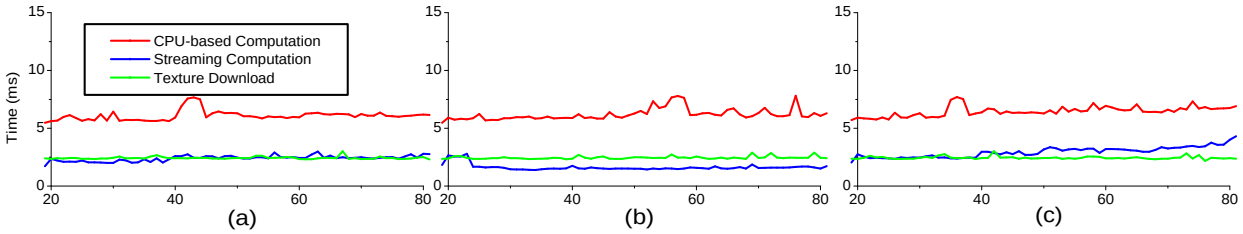


Fig. 10. Our Algorithm (StreamingCD) vs CPU-based AABB-tree Algorithm (SOLID). The graph compares the performance of SOLID [3] with ours for benchmarking set I: (a) Interlocking torii. (b) Touching torii. (c) Merging torii.

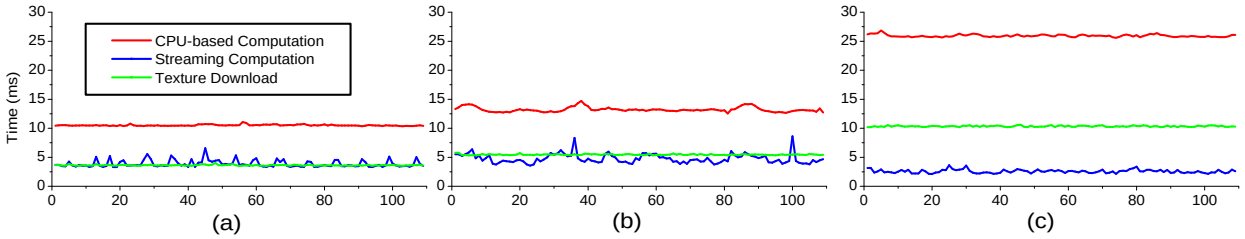


Fig. 11. Our Algorithm (StreamingCD) vs SOLID for Benchmarking Set II: (a) Bump Bunnies. (b) Happy Buddhas. (c) Intimate Animals.

such hierarchy. As a result, our update operation is more efficient and faster than [2] where AABB trees need to be updated in a bottom-up and serial manner on CPUs. Another bottleneck of the AABB tree scheme is in the process of traversal if two objects have many overlapping AABB nodes, for example, in severely deforming objects.

We have implemented the lazy AABB-update scheme employed in SOLID [2] and compare its AABB culling and AABB-tree update performance with our algorithm as shown in Fig.'s 10 and 11. In the figures, we did not include triangle-level intersection tests as they are used in both schemes. Moreover, in order to highlight the performance of our streaming algorithm, we separated the timing of texture download from CPU to GPU. Excluding the downloading time, our algorithm is 2∼10 times faster than SOLID. Including everything together, our algorithm is 1.4∼2 times than SOLID. Notice that for more complex benchmarking models such as *Intimate Animals*, our algorithm performs even better. Considering the performance growth rates of GPU compared to CPU, we expect that the performance gap of collision detection observed in this paper will be even wider in the future.

Finally, as new processors like CELL processors [11], [12] are being equipped with streaming computation capabilities, our algorithm can be adapted to other streaming processors in the future, not just for GPUs. Compared to our

algorithm, BD-tree [3] is an algorithm that is limited to reduced deformable models and suitable for only small deformations, whereas ours can handle severe deformations.

7.4.2 CULLIDE

The CULLIDE [4], [5], [6] uses GPU-supported, image-space visibility queries to perform visibility culling for potentially colliding sets. Since these methods are image-based methods, their effectiveness are subject to the rasterization resolution; however, to maintain a higher resolution in CULLIDE decreases the performance significantly [5]. In addition, the collision culling efficiency is also sensitive to the specified viewing directions. Finally, the original and quick CULLIDE [4], [6] may miss many colliding triangle pairs.

	Pruning	Tri Test	# of PCTPs	# of CTPs	Missing
1	61.01	1.91	13915	101	63%
2	35.66	4.88	36270	117	44%
3	79.87	13.13	100254	330	74%
4	64.45	6.40	291890	327	32%
5	65.50	10.67	71760	377	26%
6	127.60	21.52	165166	91	68%

TABLE IV
PERFORMANCE STATISTICS OF CULLIDE. (SEE TEXT FOR THE EXPLANATION OF THE ABBREVIATIONS)

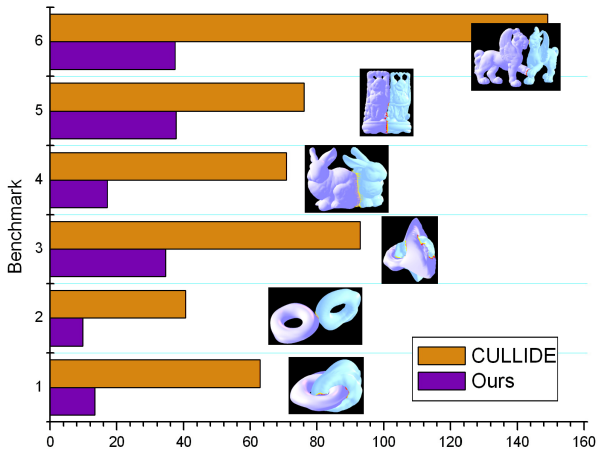


Fig. 12. Performance Comparison: Our Streaming CD vs CULLIDE. The graph compares the performance of CULLIDE with that of our algorithm to compute all the intersecting triangles under a same deformation scenario. On average, we have observed more than three times performance improvement of our algorithm over CULLIDE.

As an exact collision algorithm, however, our approach report all geometric contacts between deformable objects within a floating point precision. We have compared the performance of our algorithm with that of CULLIDE [4] on the benchmarks proposed in Table III. The CULLIDE library was provided by the authors of [4] and further optimized for better performance. Table IV shows the performance statistics of CULLIDE on our benchmarking models. In this table, 'Pruning' denotes the time spent on pruning (occlusion query) using the nVIDIA OpenGL extension GL_NV_occlusion_query; 'Tri Test' denotes the time spent on the exact pairwise triangle intersection test for the left triangles after Pruning; '# of PCTPs' denotes the potential colliding triangles; '# of CTPs' denotes the colliding triangles and 'Missing' denotes the percentage of the missing collisions. We test the performance of CULLIDE at an image space resolution of 512×512. During the tests, we observed that many missing collisions (30%-70%) arise in CULLIDE due to the image space resolution, even though we optimized visibility query by providing manually-optimized view directions. As mentioned in CULLIDE [4], the pruning efficiency largely depends upon the choice of view direction for orthographic projection. A view direction randomly selected will cause worse pruning performance in our scenarios. A higher image space resolution can reduce missing collisions, but then it require more time on visibility culling and pairwise exact triangle tests. Fig. 12 shows the performance results. In our experimental setting, we have observed about three times performance improvement over CULLIDE. As we increase the image space resolution for CULLIDE, we expect even higher performance gaps between ours and CULLIDE.

We expect better performance and higher accuracy from the improved versions of CULLIDE such as R-CULLIDE [5] or Quick-CULLIDE [6]. But since these methods rely on AABB-tree culling to narrow down the *potentially colliding sets*, a combination of our techniques with these methods is expected to provide even better performance.

7.4.3 Other Related Algorithms

Based on chromatic decomposition, CDCD [35] performs graph coloring on a polygonal mesh model that requires a fixed connectivity. Whereas, our approach makes no assumptions about input geometry and topology and works on arbitrary polygonal models, i.e., *polygon soups*. In [36], mapping AABB trees onto GPUs has been proposed by progressively building tree structure on GPUs and issuing HW-supported queries to check for the number of primitives to be read into the frame buffer. But this algorithm shows a poor performance because it relies on multi-pass rendering and a brute force readback from GPU memory. Moreover, this algorithm [36] is designed for only rigid bodies, and it is not clear whether it can handle severely deformable bodies because updating the entire AABB trees on GPUs can be a huge bottleneck.

8 Conclusion and Future Work

We have presented a fast, exact collision detection algorithm for severely deformable models using streaming AABBs. This approach has been implemented on programmable GPUs that perform massively-parallel streaming computations very rapidly. Our approach is applicable to arbitrary triangular models. The algorithm involves streaming AABB overlap tests and stream update using SIMD computations available on modern GPUs. In addition, to improve the performance and scalability of the algorithm, we have presented a stream reduction technique for efficient readback and a tile-based rendering. Compared to the earlier algorithms, our approach provides highly interactive update rates while being able to report all the colliding triangles in the deformable models.

Our algorithm has a few limitations. One of them is that the algorithm requires pre-setup time to prepare AABB streams and to map them onto textures in GPU's memory. Moreover, our algorithm may need more texture memory than other GPU-based CD algorithms. Finally, our algorithm can not report self-intersections occurring inside a model.

For future work, we want to extend our algorithm to provide separation distance and penetration depth to better support physically-based simulation. We would also like to investigate a possibility of haptic rendering of deformable models using our algorithm.

References

[1] M.C. Lin and D. Manocha, "Collision detection and proximity queries," in *Handbook of Discrete and Computation Geometry, 2nd Ed.*, 2004, pp. 787–807.

[2] G. van den Bergen, "Efficient collision detection of complex deformable models using AABB trees," *Graphics Tools*, vol. 2, no. 4, pp. 1–13, 1997.

[3] D.L. James and D.K. Pai, "BD-Tree: Output-sensitive collision detection for reduced deformable models," *Trans. Graphics*, vol. 23, no. 3, 2004.

[4] N.K. Govindaraju, S. Redon, M.C. Lin, and D. Manocha, "CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware," in *Proc. Graphics Hardware*, 2003, pp. 25–32.

[5] N.K. Govindaraju, M.C. Lin, and D. Manocha, "Fast and reliable collision culling using graphics processors," in *Proc. ACM Symp. VRST*, 2004, pp. 2–9.

[6] N.K. Govindaraju, S. Redonn, M.C. Lin, and D. Manocha, "Quick-CULLIDE: Efficient inter- and intra-object collision culling using graphics hardware," in *Proc. IEEE Virtual Reality*, 2005, pp. 59–66,319.

[7] G. Baciu and W. Wong, "Image-based techniques in a hybrid collision detector," *IEEE Trans. Visualization and Computer Graphics*, vol. 9, no. 2, pp. 254–271, 2003.

[8] G. Baciu and W. Wong, "Image-based collision detection for deformable cloth models," *IEEE Trans. Visualization and Computer Graphics*, vol. 10, no. 6, pp. 649–663, 2004.

[9] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," in *Proc. Eurographics*, 2004, pp. 119–135.

[10] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Proc. Eurographics*, 2005, pp. 21–51.

[11] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," in *IEEE Int'l Conf. Solid-State Circuits*, 2005, pp. 184–185,592.

[12] B. Flachs, S. Asano, S.H. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano, "The microarchitecture of the streaming processor for a CELL processor," in *IEEE Int'l Conf. Solid-State Circuits*, 2005, pp. 134–135.

[13] J. Owens, "Streaming architectures and technology trends," in *GPU Gems 2*, 2005, pp. 457–470.

[14] M. Pharr, *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation*, Addison-Wesley, 2005.

[15] R. Fernando, *GPU Gems: Programming techniques, tips, and tricks for real-time graphics*, Addison-Wesley, 2004.

[16] J.D. Cohen, M.C. Lin, D. Manocha, and M.K. Ponamgi, "I-COLLIDE: An interactive and exact collision detection system for large-scale environments," in *Symp. Interactive 3D Graphics*, 1995, pp. 189–196.

[17] T.C. Hudson, M.C. Lin, J. Cohen, S. Gottschalk, and D. Manocha, "V-COLLIDE: Accelerated collision detection for VRML," in *Proc. Symp. VRML*, 1997, pp. 117–125.

[18] S.A. Ehmann and M.C. Lin, "Accurate and fast proximity queries between polyhedra using surface decomposition," *Computer Graphics Forum*, vol. 20, no. 3, pp. 500–510, 2001.

[19] P. Jimenez, F. Thomas, and C. Torras, "3D collision detection: A survey," *Computers and Graphics*, vol. 25, no. 2, pp. 269–285, 2001.

[20] R. Bridson, R. Fredkiw, and J. Anderson, "Robust treatment for collisions, contact and friction for cloth animation," in *Proc. SIGGRAPH*, 2002, pp. 594–603.

[21] D. Baraff, A. Witkin, and M. Kass, "Untangling cloth," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 862–870, 2003.

[22] I. J. Palmer and R. L. Grimsdale, "Collision detection for animation using sphere-trees," *Computer Graphics Forum*, vol. 14, no. 2, pp. 105–116, 1995.

[23] Philip M. Hubbard, "Collision detection for interactive graphics applications," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 3, pp. 218–230, 1995.

[24] S. Gottschalk, M.C. Lin, and D. Manocha, "OBB-Tree: A hierarchical structure for rapid interference detection," in *Proc. SIGGRAPH*, 1996, pp. 171–180.

[25] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *IEEE Trans. Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.

[26] S.E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-oblivious mesh layouts," *Trans. Graphics*, vol. 24, no. 3, pp. 886–893, 2005.

[27] M. Shinya and M. C. Forgue, "Interference detection through rasterization," *Visualization and Computer Animation*, vol. 2, no. 4, pp. 132–134, 1991.

[28] K. Myszkowski, O. G. Okunev, and T. L. Kunii, "Fast collision detection between complex solids using rasterizing graphics hardware," *Visual Computer*, vol. 11, no. 9, pp. 497–512, 1995.

[29] T. Vassilev, B. Spanlang, and Y. Chrysanthou, "Fast cloth animation on walking avatars," *Computer Graphics Forum*, vol. 20, no. 3, pp. 260–267, 2001.

[30] J.C. Lombardo, M.P. Cani, and F. Neyret, "Real-time collision detection for virtual surgery," in *Proc. Computer Animation*, 1999, pp. 33–39.

[31] K. Hoff, A. Zaferakis, M.C. Lin, and D. Manocha, "Fast and simple 2D geometric proximity queries using graphics hardware," in *Proc. ACM Symp. Interactive 3D Graphics*, 2001, pp. 277–286.

[32] D. Knott and D. Pai, "CInDeR: Collision and interference detection in real-time using graphics hardware," in *Proc. Graphics Interface*, 2003, pp. 73–80.

[33] B. Heidelberger, M. Tescher, and M. Gross, "Detection of collisions and self-collisions using image-space techniques," *Journal of WSCG*, vol. 12, no. 3, pp. 145–152, 2004.

[34] W. Chen, H. Wan, H. Zhang, H. Bao, and Q. Peng, "Interactive collision detection for complex and deformable models using programmable graphics hardware," in *Proc. ACM Symp. VRST*, 2004, pp. 10–15.

[35] N.K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M.C. Lin, and D. Manocha, "Interactive collision detection between deformable models using chromatic decomposition," *Trans. Graphics*, vol. 24, no. 3, pp. 991–999, 2005.

[36] A. Gress and G. Zachmann, "Object-space interference detection on programmable graphics hardware," in *Proc. SIAM Conf. Geometric Design and Computing*, 2003, pp. 311–328.

[37] D. Horn, "Stream reduction operations for GPGPU applications," in *GPU Gems 2*, 2005, pp. 573–589.

[38] Y.J. Choi, Y.J. Kim, and M.H. Kim, "Self-CD: Interactive self-collision detection for deformable body simulation using GPUs," in *Proc. Asian Simulation*, 2004, pp. 187–196.

[39] T. Moller, "A fast triangle-triangle intersection test," *Graphics Tools*, vol. 2, no. 2, pp. 25–30, 1997.

[40] M. Harris, "Mapping computational concepts to GPUs," in *GPU Gems 2*, 2005, pp. 493–508.

[41] I. Buck, K. Fatahalian, and P. Hanrahan, "GPUBench: Evaluating GPU performance for numerical and scientifc applications," in *http://graphics.stanford.edu/projects/gpubench/*.

[42] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proc. Graphics Hardware*, 2004, pp. 133–137.

Real-Time Animation of Large Crowds

In-Gu Kang and JungHyun Han*

Game Research Center, College of Information and Communications,
Korea University, Seoul, Korea
kangin9@paran.com, jhan@korea.ac.kr

Abstract. This paper proposes a GPU-based approach to real-time skinning animation of large crowds, where each character is animated independently of the others. In the first pass of the proposed approach, skinning is done by a pixel shader and the transformed vertex data are written into the render target texture. With the transformed vertices, the second pass renders the large crowds. The proposed approach is attractive for real-time applications such as video games.

Keywords: character animation, skinning, large crowds rendering, GPU.

1 Introduction

In the real-time application areas such as video games, the most popular technique for character animation is *skinning*[1]. The skinning algorithm works efficiently for a small number of characters. On the other hand, emerging techniques for rendering large crowds[2, 3] show satisfactory performances, but do not handle skinning meshes. The skinning algorithm can be implemented using a vertex shader[4]. Due to the limited number of constant registers, however, the vertex shader-based skinning is not good for rendering large crowds. There has been no good solution to real-time skinning animation of large crowds, where each character is animated independently of the others. This paper proposes a GPU-based approach to independent skinning animation of large crowds.

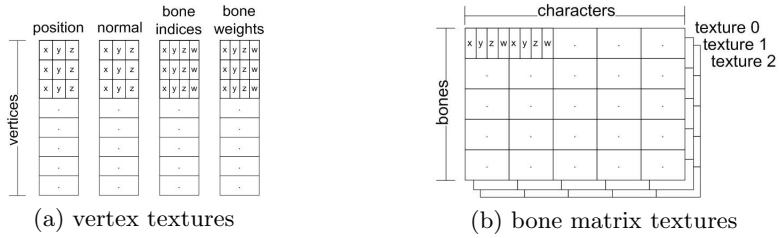
2 Pixel Shader-Based Skinning

This paper proposes a two-pass algorithm for rendering large crowds[5, 6]. In the first pass, skinning is done using a pixel shader and the transformed vertex data are written into the render target texture. With the transformed vertices, the second pass renders the large crowds.

The skinning data for a vertex consist of position, normal, bone indices and weights, and bone matrices. Fig. 1-(a) shows that position, normal, bone indices and weights are recorded in 1D textures. A vertex is influenced by up to 4 bones. The bone matrices are computed every frame, and each row of the 3×4 matrix is recorded in a separate texture, as shown in Fig. 1-(b).

* Corresponding author.

R. Harper, M. Rauterberg, M. Combetto (Eds.): ICEC 2006, LNCS 4161, pp. 382–385, 2006.



(a) vertex textures (b) bone matrix textures

Fig. 1. Texture structures for vertex and matrix data

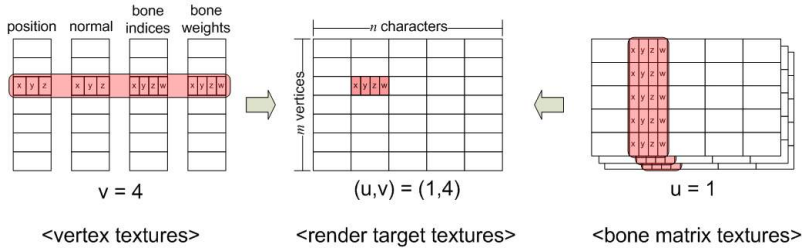


Fig. 2. Skinning and render target texture

Through a single *drawcall*, all vertices of all characters are transformed into the world coordinates, and then written into the *render target texture*. Shown in the middle of Fig. 2 is the render target texture for n characters each with m vertices. For implementing the skinning algorithm in the pixel shader, the vertex shader renders a quad covering the render target. Then, the pixel shader fills each texel of the render target texture, which corresponds to a vertex of a character.

The render target texture in Fig. 2 is filled row by row. All vertices in a row have the identical vertex index. Therefore, the vertex data from the vertex textures are fetched just once, and the cached data are repeatedly hit for processing n-1 characters.

When skinning is done, the render target texture is copied to a vertex buffer object (VBO)[7], and then each character is rendered by the vertex shader using a given index buffer. For all of the render target texture, VBO and pixel buffer object (PBO)[8], 32-bit float format is used for each of RGBA/xyzw for the sake of accuracy.

3 Implementation and Result

The proposed algorithm has been implemented in C++, OpenGL and Cg on a PC with 3.2 GHz Intel Pentium4 CPU, 2GB memory, and NVIDIA Geforce 7800GTX 256MB. Table 1 compares the frame rates of the vertex shader skinning and the proposed 2-pass skinning. For performance evaluation, view frustum culling is disabled and 'all' characters are processed by GPU. Fig. 3 shows snap-

Table 1. FPS comparison of vertex shader (VS) skinning and proposed 2-pass skinning

	soldier		horse	
# characters	VS	2-pass	VS	2-pass
1	2340	1545	2688	1571
16	580	1057	575	1179
64	200	565	163	649
256	56	200	42	219
1024	14	55	10	58
2048	7	27	5	29
4096	3	13	2	14

Fig. 3. Rendering 1,024 soldiers without LOD and frustum culling

Fig. 4. Rendering 10,240 soldiers with LOD and frustum culling

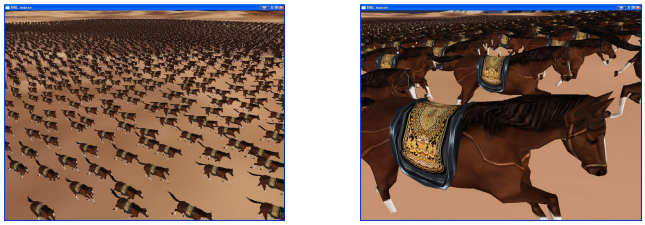


Fig. 5. Rendering 5,120 horses with LOD and frustum culling

shots of rendering 1,024 soldiers. The average FPS is 55, as shown in Table 1. In the current implementation, 3 LOD meshes are used: each with 1,084, 544 and 312 polygons, respectively. Fig. 4 shows snapshots of rendering 10,240 soldiers with LOD applied. The average FPS is 60 with view frustum culling enabled.

Finally, Fig. 5 shows snapshots of rendering 5,120 horses with LOD applied. The average FPS is 62 with view frustum culling enabled.

4 Conclusion

This paper presented a pixel shader-based approach to real-time skinning animation of large crowds. The experiment results show that the proposed approach is attractive for real-time applications such as games, for example, for rendering huge NPCs (non-player characters) such as thousands of soldiers or animals. With appropriate adjustments, the proposed approach can be used for implementing MMOGs (Massively Multi-player Online Games).

Acknowledgements

This research was supported by the Ministry of Information and Communication, Korea under the Information Technology Research Center support program supervised by the Institute of Information Technology Assessment, IITA-2005-(C1090-0501-0019).

References

1. Lewis, J.P., Cordner, M., Fong, N.: Pose Space Deformations: A Unified Approach to Shape Interpoalation and Skeleton-driven Deformation. SIGGRAPH2000 165–172
2. Microsoft: Instancing Sample. DirectX SDK. February 2006
3. Zelsnack, J.: GLSL Pseudo-Instancing. NVIDIA Technical Report. November 2004
4. Gosselin, D. R., Sander, P. V., Mitchell, J. L.: Drawing a Crowd. ShaderX3. CHARLES RIVER MEDIA. (2004) 505–517
5. James, D. L., Twigg, C. D.: Skinning Mesh Animations. SIGGRAPH2005 399–407
6. Dobbyn, S., Hamill, J., O'Conor, K., O'Sullivan, C.: Geopostors : A Real-Time Geometry / Impostor Crowd Rendering System. ACM Transactions on Graphics(2005) 933
7. NVIDIA: Using Vertex Buffer Objects. NVIDIA White Paper. October 2003
8. NVIDIA: Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL. NVIDIA User Guide. August 2005

PERGAMON

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Pattern Recognition 37 (2004) 1311–1314

PATTERN RECOGNITION
THE JOURNAL OF THE PATTERN RECOGNITION SOCIETY

www.elsevier.com/locate/patcog

Rapid and Brief Communication

GPU implementation of neural networks

Kyoung-Su Oh*, Keechul Jung

School of Media, College of Information Science, Soongsil University, 1, SangDo-Dong, DongJak-Gu, Seoul, 156-743, Republic of Korea

Received 6 January 2004; accepted 14 January 2004

Abstract

Graphics processing unit (GPU) is used for a faster artificial neural network. It is used to implement the matrix multiplication of a neural network to enhance the time performance of a text detection system. Preliminary results produced a 20-fold performance enhancement using an ATI RADEON 9700 PRO board. The parallelism of a GPU is fully utilized by accumulating a lot of input feature vectors and weight vectors, then converting the *many* inner-product operations into *one* matrix operation. Further research areas include benchmarking the performance with various hardware and GPU-aware learning algorithms.

Keywords: Graphics processing unit(GPU); Neural network(NN); Multi-layer perceptron; Text detection

1. Introduction

Recently graphics hardware has become increasingly competitive as regards speed, programmability, and price. Besides, graphics processing units (GPUs) have already been used to implement many algorithms in various areas, including computational geometry, scientific computation, and image processing, as well as computer graphics [1,2].

In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4].

Accordingly, the current paper presents a faster NN using common graphics hardware GPU. Although no graphics hardware is dedicated to NN computation, it can still be adapted to many pattern recognition problems with an inexpensive and minimal hardware overhead. The essential operation in an NN is the inner-product between a weight vector and an input vector in each layer. Therefore, to utilize the parallelism of a GPU, lots of input feature vectors and weight vectors are accumulated, then the *many* inner-product operations are converted into *one* matrix operation. As such, 'multiplication' and a 'non-linear threshold function, such as a sigmoid' can be effectively implemented using the *vertex shader* and *pixel shader* in a GPU.

2. Neural network architecture

An artificial neural network, usually referred to as 'neural network', is based on the concept of the workings of the human brain. There are many different types of NN, with the more popular being a multilayer perceptron, learning vector quantization, radial basis function, Hopfield, and Kohonen.

The current study focuses on using a GPU to implement a multilayer perceptron, which is usually fully connected between adjacent layers. The input layer receives the input features of a given application. Although the network

* Corresponding author. Tel.: +82-2-828-7260; Fax: +82-2-822-3622.
E-mail addresses: oks@ssu.ac.kr (K.-S. Oh), kcjung@ssu.ac.kr (K. Jung).

0031-3203/$30.00
doi:10.1016/j.patcog.2004.01.013

structure can vary as regards the number of layers, number of nodes in each layer, and input mask size, each layer performs the same inner-product operation between the given input vectors and the weight vectors, followed by a non-linear function. Moreover, many inner-product operations can be replaced with a matrix multiplication, which is more appropriate for GPU implementation.

3. GPU processing

Graphics hardware has only been used for rendering within the last few decades, however, its extended capabilities in supporting complex operations have also become useful in non-graphics applications. In particular, the advent of a programmable vertex shader and pixel shader enables flexible functions for general computation. Since GPUs are designed for high-performance rendering where repeated operations are common, they are more effective in utilizing parallelism and more pipelined than general purpose CPUs. Therefore, in areas where repeated operations are common, a GPU can produce a better performance than a CPU.

The mechanism of general computation using a GPU is as follows. The input is transferred to the GPU as textures or vertex values. The computation is then performed by the vertex shader and pixel shader during a number of rendering passes. The vertex shader performs a routine for every vertex that involves computing its position, color, and texture coordinates, while the pixel shader is performed for every pixel covered by polygons and outputs the color of the pixel.

As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix} = \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$

$$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}, \tag{1}$$

$$M = W \times X + B$$

$$= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$$

$$+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$$

$$= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \tag{2}$$

$$R = \mathrm{sigmoid}(M)$$

$$= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \tag{3}$$

where w_{ij} denotes the weight at the connection between the ith node of the output layer and the jth node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the ith feature value of the jth input vector and b_i is the bias term for the ith output node from L input vectors. The final result R_{ij} is the output of the ith output node for the jth input vector.

The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X, then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X. For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X, while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X, and so on. As a result of

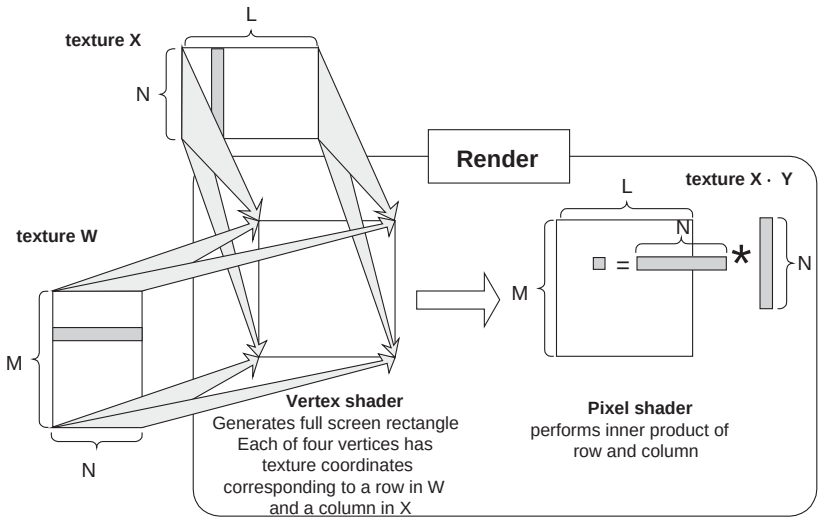


Fig. 1. Overview of matrix multiplication using GPU.

the vertex shader, every pixel (i, j) has texture coordinates corresponding to the ith row of W and the jth column of X. The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations.

The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, *texture* $W \times X$, are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex's texture coordinate for the *texture* $W \times X$ correspond to its position. For example, the upper left vertex has the texture coordinate $(0, 0)$, while the texture coordinate for the upper right vertex is $(1, 0)$. As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation.

If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation.

4. Application to pattern recognition

Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3].

Therefore, this paper briefly describes such a text detection method, and readers are referred to the author's previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU's performance is maximized by accumulating a large number of input vectors [1] to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M × M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU.

[1] It is dependent on the GPU configuration. The maximum texture size of an ATI RADEON 9700 PRO board is 2048.

(a) (b)

Fig. 2. Experimental Results: (a) test image, (b) result of MLP with GPU.

Table 1
Processing times per elementary operations

	Texture creation	Matrix multiplication	Sigmoid
GPU	0.469000	0.030000	0.031000
CPU		11.743	

As shown in Table 1, we get a 20-fold performance enhancement using an ATI RADEON 9700 PRO board compared to CPU-only processing.

Acknowledgements

This work was supported by the Soongsil University Research Fund.

References

[1] A. Moravanszky, Linear algebra on the GPU, in: W.F. Engel (Ed.), Shader X 2, Wordware Publishing, Texas, 2003.
[2] D. Manocha, Interactive geometric & scientific computations using graphics hardware, SIGGRAPH 2003 Tutorial Course #11.
[3] K. Jung, Neural network-based text location in color images, Pattern Recog. Lett. 22 (14) (2001) 1503–1515.
[4] J. Zhu, P. Sutton, FPGA implementation of neural networks —a survey of a decade of progress, Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003), Lisbon, 2003, pp. 1062–1066.

Jinwook Kim · Soojae Kim · Heedong Ko · Demetri Terzopoulos

Fast GPU Computation of the Mass Properties of a General Shape and its Application to Buoyancy Simulation

Abstract To simulate solid dynamics, we must compute the mass, the center of mass, and the products of inertia about the axes of the body of interest. These mass property computations must be continuously repeated for certain simulations with rigid bodies or as the shape of the body changes. We introduce a GPU-friendly algorithm to approximate the mass properties for an arbitrarily shaped body. Our algorithm converts the necessary volume integrals into surface integrals on a projected plane. It then maps the plane into a framebuffer in order to perform the surface integrals rapidly on the GPU. To deal with non-convex shapes, we use a depth-peeling algorithm. Our approach is image-based; hence, it is not restricted by the mathematical or geometric representation of the body, which means that it can efficiently compute the mass properties of any object that can be rendered on the graphics hardware. We compare the speed and accuracy of our algorithm with an analytic algorithm, and demonstrate it in a hydrostatic buoyancy simulation for real-time applications, such as interactive games.

Keywords General-purpose computation on GPUs · Mass property computation · Physics-based animation · Rigid-body dynamics · Buoyancy simulation

J. Kim, S. Kim, H. Ko
Imaging Media Research Center
Korea Institute of Science and Technology
39-1 Hawolgok-dong, Seongbuk-gu, Seoul, 136-791, Korea
E-mail: {jwkim, lono, ko}@imrc.kist.re.kr

D. Terzopoulos
Department of Computer Science
University of California
Los Angeles, CA 90095, USA
E-mail: dt@cs.ucla.edu

1 Introduction

The fast calculation of mass properties, including the mass, center of mass, and products of inertia, is necessary for the dynamic simulation of solids. In rigid body dynamics, the mass properties are usually assumed to be constant during the simulation. Therefore, the computation can be performed in an initialization step and the computed values are used in the subsequent simulation. Hence, the computational cost to calculate mass properties is often negligible. In certain important cases, however, the mass properties can change during the simulation and complex geometric shapes may require expensive mass property computations.

Among these cases is the simulation of hydrostatic buoyancy. Buoyancy is a natural phenomenon resulting from the interplay between a fluid system and a floating rigid body system. If we assume a hydrostatic pressure condition for the fluid system, then we can simulate the motion of the rigid body floating in the fluid by applying a buoyant force to the center of mass of the instantaneous submerged volume, which is known as the center of buoyancy. The buoyant force itself is proportional to the instantaneous submerged volume. A problem here, of course, is that the submerged volume changes continuously. Consequently, the computation of its mass properties can be a major bottleneck of the simulation.

Most of the research in computing the mass properties of solid shapes can be applied only to specific solid representation schemes and, therefore, it may involve an expensive representation conversion process [11]. Gonzalez et al. [6] combined a polynomial free-form surface representation with the Gauss divergence theorem to efficiently calculate the moments of the enclosed object. However, their approach allows only piecewise polynomial surface patches. Mirtich [13] proposed an efficient method to compute the center of mass and higher-order moments for polyhedral objects. The proposed algorithm is based upon a three step reduction of the volume integrals to successively simpler integrals. The final step of the algorithm computes the required integrals over a

face from the coordinates of the projected vertices. This means that the computation is done by algebraic operations with vertex coordinate values. Even though this method is computationally efficient for fixed polyhedral objects, its efficiency can suffer if the geometric structure changes frequently as it may require an expensive reconstruction of a set of vertices and faces. Unfortunately, the typical situation in buoyancy simulations requires repeated updates of vertex coordinates and even of the number of relevant vertices. This is because the submerged volume is defined as the intersection of a geometric object representing the fluid system with a geometric object representing the floating rigid body.

In this paper, we propose a GPU-friendly algorithm to compute the mass properties determined by general geometries. Our approach is essentially image-based. Because of this, it is not restricted by the mathematical or geometric representation of rigid bodies. Regardless of the geometric representations employed, whether they be polyhedral approximation, free-form surfaces, constructive solid geometry, etc., if it is possible to render an object of interest on the GPU, then our algorithm can approximate the object's mass properties, exploiting the efficiency of the GPU.

Recent advances in the programmability of graphics hardware have enabled its use for general purpose computation, not restricted to rendering [16]. Various problems in scientific computation, including fluid dynamic simulation, the solution of linear systems of algebraic equations, nonlinear optimization, and volume rendering, have been addressed by taking advantage of the parallelism and programmability of GPUs [1,7–9,14, 17]. Moreover, programmable GPUs are getting faster and cheaper. Our algorithm accrues these benefits by exploiting the GPU to calculate mass properties. It first computes the mass, the center of mass, and the products of inertia by reducing volume integrals into surface integrals. It projects surfaces of the rigid body onto a plane that corresponds to the frame buffer of a rendering process. Next, it computes the integrands on the GPU. Finally, it performs a summation operation using a buffer reduction to obtain the desired result.

To perform the required integral operations over all the surfaces representing the non-convex geometric object, we use a depth-peeling algorithm to obtain each of the surface patches regardless of convexity. The depth-peeling is a fragment level depth sorting algorithm, which achieves a correct rendering of transparent objects that are located order independently [4,12]. The objective of the method is to find the fragments of geometry in a systematic manner. We focus our attention on this method because it can access all the fragments representing the geometry regardless of its convexity. We modify the original depth-peeling algorithm to obtain surface peels, which are surface patches beneath the fluid in our buoyancy simulation, as well as the intersection surface between the fluid and the rigid body.

The remainder of the paper is organized as follows: Section 2 reviews rigid body mass properties and derives them in the form of surface integrals over the projected plane. Section 3 introduces our GPU-friendly algorithm for computing the mass properties determined by non-convex geometry. Section 4 presents an error and performance analysis of our approach compared to the analytic method proposed by Mirtich [13]. Section 5 modifies an original depth-peeling algorithm to deal with hydrostatic buoyancy simulation and shows an example of interactive rigid body dynamics simulation under buoyancy. Finally, Section 6 draws conclusions from our work.

2 Rigid body mass properties

2.1 Computing mass properties with volume integrals

The *mass* of a rigid body is given by

$$m = \int_V \rho(x, y, z)\, dV, \tag{1}$$

where $\rho(x, y, z)$ is the *mass distribution* function of the body and V is its *volume*. If we assume $\rho(x, y, z)$ to be constant over the volume, the expression for the mass simplifies to $m = \rho V$. In this paper, the mass distribution function will be considered a constant value for simplicity.

The *center of mass* $\mathbf{r}$ and the *inertia tensor* $\mathbf{I}$ are given by

$$\begin{aligned} \mathbf{r} &= \frac{1}{V}\int_V \begin{bmatrix} x \\ y \\ z \end{bmatrix} dV, \\ \mathbf{I} &= \rho \int_V \begin{bmatrix} (y^2+z^2) & -xy & -xz \\ -yx & (z^2+x^2) & -yz \\ -zx & -zy & (x^2+y^2) \end{bmatrix} dV. \end{aligned} \tag{2}$$

2.2 Reduction to surface integrals on a projected plane

To calculate the mass properties of a rigid body efficiently, we exploit the *divergence theorem* as suggested by Gonzalez et al. [6]. According to the divergence theorem, an integral over the three-dimensional volume can be transformed into an integral over its boundary surface as follows:

$$\int_V \nabla \cdot \mathbf{f}\; dV = \int_{\partial V} \mathbf{f} \cdot \mathbf{n}\; dA, \tag{3}$$

where $\mathbf{f}$ is a continuously differentiable vector field defined on a neighborhood of V, where $\mathbf{n} = [n_x, n_y, n_z]'$ denotes the exterior normal vector of V along its boundary ∂V, and where dA is the infinitesimal surface area of the boundary. When the volume is represented by a bounding polyhedron, its boundary is the set of planar

polygons comprising its faces. If we set $\mathbf{f} = [0, 0, z]'$, then we obtain the volume as $V = \int_{\partial V} z n_z \, dA$. Similarly, setting $\mathbf{f}$ in turn to $[0, 0, xz]'$, $[0, 0, yz]'$, and $[0, 0, \frac{1}{2}z^2]'$ yields $\int_V x \, dV = \int_{\partial V} xz n_z \, dA$, $\int_V y \, dV = \int_{\partial V} yz n_z \, dA$, and $\int_V z \, dV = \int_{\partial V} \frac{1}{2} z^2 n_z \, dA$, respectively.

Now, we slightly modify (3) by projecting the boundary surface area element dA onto the xy plane. From Figure 1, we see that the relationship between the infinitesimal surface area dA and the projected surface area $dx\,dy$ is $dx\,dy = |n_z| \, dA$ if the surface normal vector has unit length.

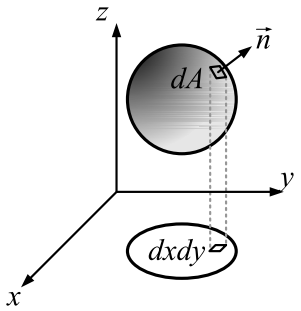


Fig. 1 Projection of the infinitesimal surface area element.

Finally, we obtain the volume V and the center of mass $\mathbf{r} = [r_x, r_y, r_z]'$ as follows:

$$\begin{aligned} V &= \int_{\partial V} \mathrm{sgn}(n_z) z \, dx\, dy, \\ r_x &= \frac{1}{V} \int_{\partial V} \mathrm{sgn}(n_z) xz \, dx\, dy, \\ r_y &= \frac{1}{V} \int_{\partial V} \mathrm{sgn}(n_z) yz \, dx\, dy, \\ r_z &= \frac{1}{2V} \int_{\partial V} \mathrm{sgn}(n_z) z^2 \, dx\, dy, \end{aligned} \tag{4}$$

where $\mathrm{sgn}(x)$ denotes the signum function which extracts the sign of a real number x. Note that the integrals are computed on the planar surface area, which is achieved by projecting the surface boundary onto the xy plane. When the surface area element dA is projected on the xy plane, it will be singular if $n_z = 0$. Hence, an improper choice of $\mathbf{f}$ (e.g., $\mathbf{f} = [x, 0, 0]'$ to compute the volume) can lead to a singularity at the boundary of a projected surface, where it would require division by a very small number. Our proposed $\mathbf{f}$s, however, only require multiplication by $\mathrm{sgn}(n_z)$, thus avoiding the singularity problem at the boundaries.

The inertia tensor $\mathbf{I}$ is

$$\mathbf{I} = \rho \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}, \tag{5}$$

where the *moments and products of inertia* are similarly given as follows:

$$\begin{aligned} I_{xx} &= \int_{\partial V} \mathrm{sgn}(n_z) x^2 z \, dx\, dy, \\ I_{xy} &= \int_{\partial V} \mathrm{sgn}(n_z) xyz \, dx\, dy, \\ I_{yy} &= \int_{\partial V} \mathrm{sgn}(n_z) y^2 z \, dx\, dy, \\ I_{xz} &= \frac{1}{2} \int_{\partial V} \mathrm{sgn}(n_z) xz^2 \, dx\, dy, \\ I_{yz} &= \frac{1}{2} \int_{\partial V} \mathrm{sgn}(n_z) yz^2 \, dx\, dy, \\ I_{zz} &= \frac{1}{3} \int_{\partial V} \mathrm{sgn}(n_z) z^3 \, dx\, dy. \end{aligned} \tag{6}$$

3 Computing mass properties on the GPU

3.1 Shader implementation

The programmability of recent graphics hardware and the various choices of precision and formats of framebuffers enable us to implement mass property computations on GPUs in an easy and flexible way. The integrands in equations (4) and (6) can be evaluated discretely at each pixel in a framebuffer by GPU programming. The process is straightforward:

1. Render the geometry with an orthographic projection onto the xy plane;
2. Evaluate the integrands on a fragment shader;
3. Encode the evaluated values at the output buffers.

The number of parameters that must be computed is 10 in total, including 1 for volume, 3 for the center of mass, and 6 for the moments and products of inertia. To store these parameters, we use three framebuffers, each of which can contain four values in the red, green, blue and alpha channel. This can be efficiently implemented using the "multiple render target" capability of recent graphics hardware, which enables the fragment shader to save per-pixel data in multiple buffers.

Hence, we obtain color buffers containing the values of integrands in equations (4) and (6). Furthermore, the integration of the values over the projected plane area can be performed by reading back fragment color values of the framebuffers and summing them up, or by using a buffer reduction algorithm as will be explained in the next section. A fragment shader can be implemented in the OpenGL Shading Language [15] very easily, as follows:

```
// homogeneous coordinate of a point on the surface
varying vec4 p;

// z component of the surface normal
varying float n_z;

void main(void)
{
  float c = sign(n_z) * p.z;

  // (rx, ry, rz, V)
  gl_FragData[0] = c * p;

  // (Ixx, Ixy, Ixz, .)
  gl_FragData[1] = p.x * gl_FragData[0];

  // (Iyy, Iyz, Izz, .)
  gl_FragData[2] = c * vec4(p.y * p.y, p.y * p.z, p.z * p.z, 0);
}
```

Note that the fourth components of `gl_FragData[1]` and `gl_FragData[2]` are not used.

A potential problem is how to generate color buffers covering all the surface fragments of the geometric shape. Consider the case of a sphere. The surface of a sphere can be divided into two patches—the north and south hemispheres—according to the direction of surface normals. If we look at the sphere from the negative z viewing direction, the line of sight will intersect the sphere twice. That is, the typical rendering pipelines will render two fragments from those two surface patches on one pixel in the framebuffer and, therefore, the resulting color buffer will contain only one of the fragments from the two surface patches regardless of the choice of the depth test function. To resolve this problem, we use the depth-peeling algorithm discussed in the next section.

3.2 Depth-peeling

Using the standard depth test function of the 3D graphics API, we can obtain the nearest surface fragment from the eye at each pixel. Although the second nearest or other fragments may be required in some areas, there is no straightforward way to obtain the nth nearest fragment. One possible solution is to use a depth-peeling algorithm, which is a fragment-level depth sorting technique [12]. Depth-peeling can be implemented as a multipass algorithm. In the first rendering pass, the geometries are rendered using a normal "less-than" depth function. This will yield a depth buffer containing the depth values of the nearest surface of the geometry. In the next rendering pass, only the fragment for which depth is greater than the depth values in the buffer from the previous pass are rendered. Then the depth buffer will contain the depth values of the next nearest surface of the geometry, and so on. The process repeats until the depth values of all the surface fragments are found. The depth-peeling technique introduced by Everitt [4] requires a shadow buffer to peel away the surfaces by comparing depth values. However, since recent GPUs and APIs support "render-to-texture" capabilities and the direct manipulation of pixel values on fragment processors using shading languages, depth-peeling can be implemented using programmable GPUs and the modification of the algorithm is even easier.

For our objective of computing mass properties, we can apply the standard depth-peeling algorithm with the shader developed in the previous section. As a result, we obtain n textures containing the enumerated integrands in equations (4) and (6), where n is a total number of peels.

3.3 Two-dimensional integrals over the projected area using buffer reduction

Using the textures obtained in the previous section, we compute the two-dimensional integrals over the projected surfaces in order to obtain mass properties. A straightforward way to perform the integration is to read all evaluated integrands from framebuffers and sum them. Given current graphic memory interfaces, however, reading back a texture memory directly into system memory can yield significant latency. To tackle this problem, we use buffer reduction [2]. To summarize the buffer reduction technique, a fragment program reads two or more values from the buffer and computes a new value using the reduction operator, which in our case is an addition operation. These passes continue until the output is reduced to a single value, the sum. In general, this process takes $O(\log n)$ passes, where n is the number of elements to reduce. Figure 2 illustrates a reduction operation to calculate the sum.

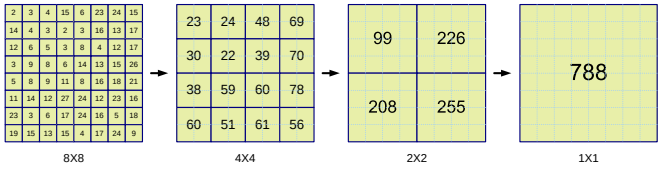


Fig. 2 Summation reduction procedure.

4 Performance

In this section, we compare our algorithm in terms of accuracy and speed with the analytic method developed by Mirtich [13]. Since the analytic method used in this test is restricted only to polyhedra, we use shapes approximated by polyhedra as test objects. Figure 3 illustrates some of these objects. It is important to note, however, that our algorithm can be applied to any model that can be rendered on graphics hardware. All the tests were run on a 2.53GHz Pentium 4 CPU with an NVIDIA GeForce 7800 GTX GPU. 32-bit floating point textures were used for framebuffers. Table 1 lists all the geometric test objects and the number of peels for each test object.

Fig. 3 Some polyhedral test objects.

object	vertices	faces	peels
cube	8	12	2
sphere	422	840	2
teapot	821	1628	6
torus	1024	1922	4
knot	1440	2880	8
bunny	2557	5110	6
pipe	4626	9252	6

Table 1 Geometric information for the test objects.

4.1 Error analysis

We measured the relative error of the mass and the moments of inertia I_{xx}, I_{yy}, and I_{zz} at various framebuffer resolutions. The other mass property values are very small for our test objects, because the shapes are approximately symmetric along axes. Our approach computes integrands for each fragment on the GPU, where we use texture memories as framebuffers. Hence, the resolutions of the framebuffers are critical for accurate results. As shown in Figure 4, a resolution of 32×32 was sufficient to compute the mass properties within a 5% error bound.

4.2 Performance analysis

We now compare the performance of our algorithm and the analytic method. If we assume that the cost of vertex processing on the GPU is negligible compared to the cost of fragment processing, the complexity of our algorithm is approximately $O(kn^2)$, where k is the number of rendering passes for the depth-peeling and n is the framebuffer resolution along its width or height. On the other hand, the complexity of the analytic method is $O(m)$, where m represents the number of faces of the polyhedron. Figure 5 shows a comparison of the computation times for the analytic method and our GPU-based method at three different framebuffer resolutions.

We observed that our GPU-based approach is comparable to the analytic method in terms of computational cost. At 64×64 resolution, our algorithm outperforms the analytic method for moderately complex shapes such as the bunny or the pipe. However it also shows the downside of quadratic complexity for a resolution of 128×128 or more. For example, it is obvious that the analytic method is preferable to our GPU-based method for low-polygon-count models such as a cube. The computational

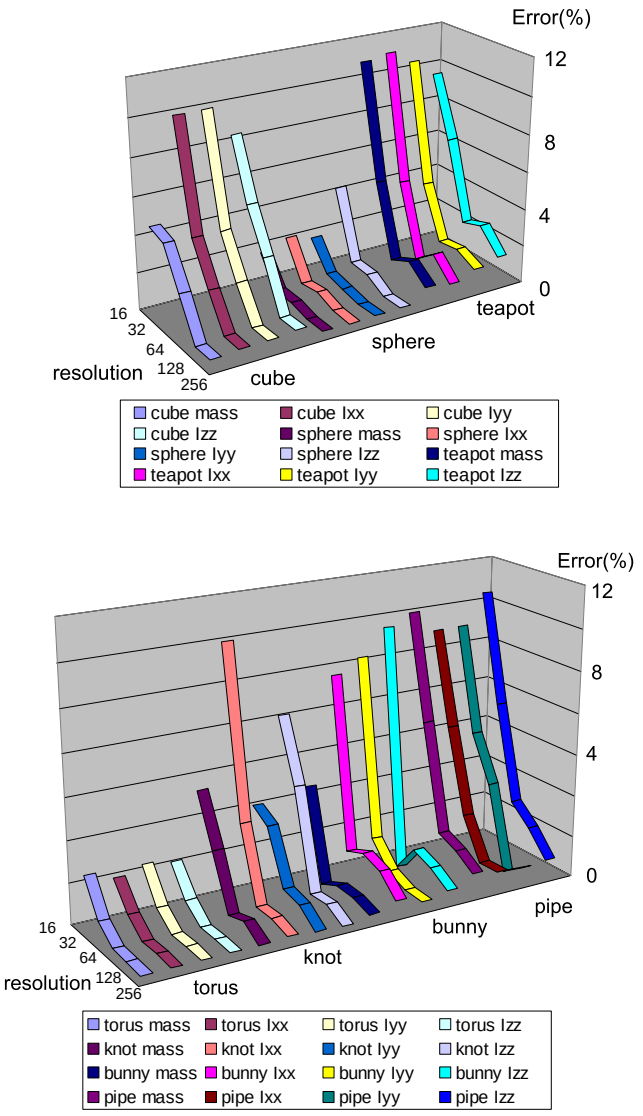


Fig. 4 Relative error comparisons.

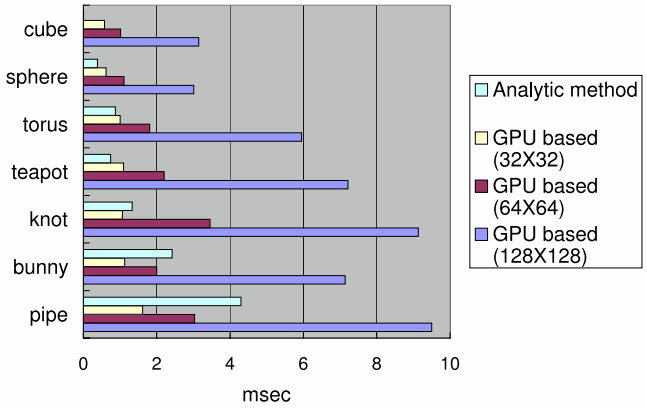


Fig. 5 Computational time comparison of analytic method and GPU-based method.

cost of the analytic method for the cube is so small that we could not distinguishably display it in the graph.

5 Case study: Buoyancy simulation

The beauty of our image-based approach is that it is not restricted to any particular mathematical or geometric shape representation. It can efficiently compute the mass properties of arbitrary objects, so long as they can be rendered efficiently on graphics hardware. As an example application of our algorithm, we will now demonstrate an interactive, hydrostatic buoyancy simulation.

5.1 Hydrostatic buoyancy

One of the most popular simplifications of fluid motion is the *shallow water model* [10] which assumes zero viscosity and considers only two-dimensional motions. An interesting fact of the shallow water model is that the pressure field is characterized by the *hydrostatic equilibrium condition*:

$$p = \rho g h, \tag{7}$$

where g is the gravitational acceleration and h is the depth of the fluid. This very simple pressure model works well with the shallow water model, and it corresponds exactly to the observation of Archimedes.

According to *Archimedes' principle*, a body immersed in a fluid experiences a vertical *buoyant force* equal to the weight of the fluid that it displaces. The buoyant force acts on the center of mass of the submerged volume. Figure 6 illustrates a rigid body partially immersed in a fluid. Assuming a stationary fluid system, two forces are acting on the body at this instant. The first is the force of gravity that acts downwards at the center of gravity C, while the second is a buoyant force which acts upwards at the *center of buoyancy* B, which is the center of mass of the immersed part of the rigid body (assuming that the immersed portion consisted of fluid). The magnitude of the buoyant force is proportional to the weight of the submerged volume of fluid. The imbalance between gravity and the buoyant force induces a torque that will rotate the body to restore a static equilibrium.

The simulation of fluid motion is out of the scope of our work.[1] Instead we focus on the rigid body motion of an object floating on fluid due to the hydrostatic buoyant force. To simulate hydrostatic buoyancy, we compute the volume and the center of mass of the submerged part

[1] Foster and Metaxas [5] demonstrate a simplified scheme for coupling buoyant objects to the results of a Navier-Stokes fluid simulation. Carlson et al. [3] simulate the interplay between rigid bodies and viscous incompressible fluid.

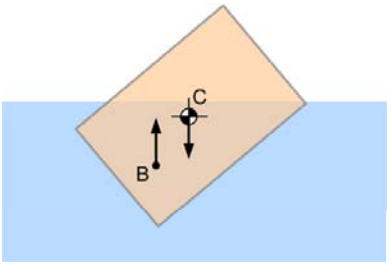


Fig. 6 Buoyant force and gravity acting on a partially submerged rigid body.

of the body at every simulation time instant. If the geometries of a fluid body and a rigid body are complicated, calculating their intersection requires a considerable amount of computation and can become a bottleneck in the simulation process. In the following section, we tackle this problem by modifying the depth-peeling algorithm.

5.2 Boundary surfaces of an intersection volume of a non-convex geometry and a fluid surface

We improve the original depth-peeling technique to account for all the projected fragments of the geometry below the fluid surface. For simplicity, let us assume that the signs of the z components of the fluid surface normal vectors do not change. Our algorithm considers surfaces from the rigid body and the fluid surface intersecting the geometry separately. The multi-pass rendering procedure to handle the surfaces of a submerged volume is as follows (note that an orthographic projection is applied to render the scene with the negative z viewing direction as shown in Figure 7):

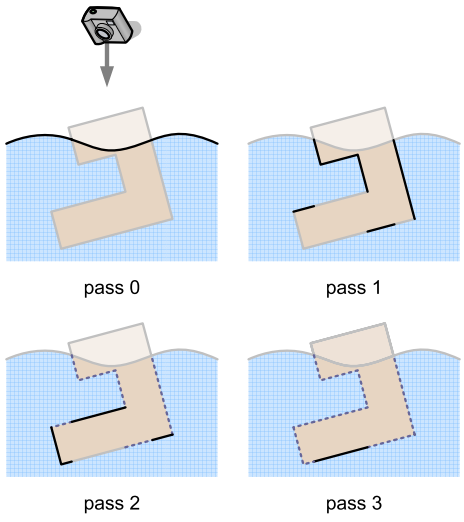


Fig. 7 Multi-pass rendering to obtain all surface patches.

Pass 0: Render the surface of the fluid, storing its depth values into a texture T_{dw} as a reference.

Pass 1: Render the object geometry to a texture T_1. In our fragment shader, the integrands in equation (4)—i.e., $\mathrm{sgn}(n_z)xz$, $\mathrm{sgn}(n_z)yz$, $\mathrm{sgn}(n_z)z^2$, and $\mathrm{sgn}(n_z)z$—are evaluated and the output color is composed of their values. Also, the depth values are stored in a texture T_{d1}. During this rendering pass, the fragments whose depth value is less than the fluid surface depth are discarded to inhibit the operation. The texture T_{dw} generated in rendering Pass 0, is used to lookup the depth value of the fluid surface. In Figure 7, the solid black lines correspond to the fragments.

Pass 2: Render the geometry to a texture T_2. As in the previous rendering pass, the integrands are evaluated and their values are assigned to the output color. Here, only the fragments whose depth value is greater than the fluid surface depth *and* the depth value of T_{d1} are accepted in order to peel away the surface patch obtained in the previous rendering pass. In Figure 7, the dashed lines indicate peeled away fragments. A depth texture T_{d2} is initialized with T_{d1} and overwritten with the depth values of the currently processed fragments.

Pass n: Repeat the same process as in rendering Pass 2 until all the object fragments are found and evaluated.

Thus, we obtain n textures, and the texture T_n contains the evaluated integrands of the nth surface patch.

Now, the only remaining surface patch is the fluid surface intersecting with the rigid body geometry. As illustrated in Figure 8, the surface patches of rendering Pass 1 consist of upward and downward faces. The fluid surface intersecting with the rigid body geometry can be obtained by drawing the fluid surface only for those fragments having a downward normal in rendering Pass 1. Note that a more efficient implementation results if the fragment shader can write a stencil bit into the output framebuffer in rendering Pass 1. Finally, we evaluate the integrand for the fluid surface patch intersecting the rigid body geometry and write the value in a texture T_w.

In summary, our algorithm requires a total of $n + 2$ rendering passes to cover all the surface patches of a partially submerged rigid body geometry, where n represents the maximum number of intersection points of the submerged part of the geometry with the z axis. The first rendering pass generates a reference depth texture from the fluid surface. In the next n rendering passes, integrands are evaluated for each fragment of the geometry surface and the resulting values are stored in textures T_i. The final rendering pass evaluates the integrand for the fluid surface patch that intersects the rigid body geometry and stores the values in a texture T_w.

Finally, we apply the summation reduction procedure described in Section 3.3 to evaluate the integral expressions for the volume of the immersed portion of the object and the center of buoyancy in order to evaluate the buoyant force and its point of application in the object.

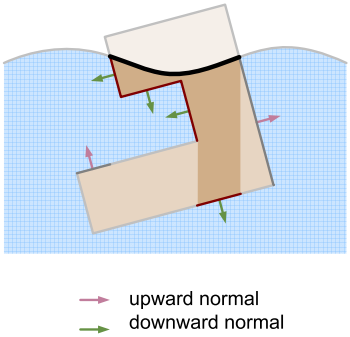


Fig. 8 Intersecting the surface of the fluid body with a rigid body.

5.3 Simulation example

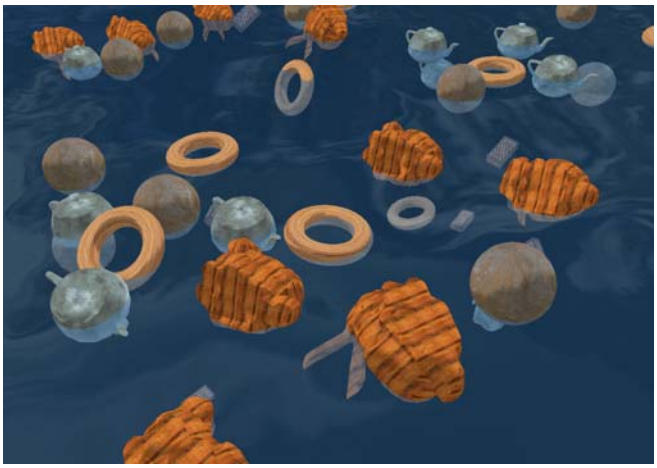


Fig. 9 Interactive simulation of 50 rigid bodies floating in water.

Figure 9 shows a typical scene from our interactive simulations of buoyant objects. Ten spheres, 10 rectangular boxes, 10 tori, 10 teapots, and 10 Stanford bunnies were tested. Boxes with density higher than that of the water were observed to sink as expected. We also modeled a viscous drag force acting at the center of buoyancy with magnitude proportional to the submerged volume and the square of the body velocity. The simulation runs on the CPU of a 2.53GHz Pentium 4 PC employing an NVIDIA GeForce 7800 GTX GPU. The average frame rate of the example shown in the figure was 16 frames/sec. Over 90% of the computational resources were consumed in calculating the buoyant force.

For spherical and rectangular bodies, depth-peeling was applied twice to compute the submerged volume of the object geometries. For the teapot and Stanford

bunny bodies, depth-peeling was applied a maximum of 6 times, but in most cases 3 or 4 peels sufficed to cover the submerged volume. The framebuffer resolution used in this example was 32×32, allowing at most 5% approximation error. The leftmost images in Figure 10 show the gravity force (downward blue arrow) and buoyant force (upward yellow arrow) acting on a bunny, a torus, and a teapot. The remaining images are color buffers that encode the integrands for each peel, as described in the previous section. Since the framebuffers use a floating point texture format that cannot be illustrated properly, we have transformed the values so that they map to a color range of [0,1].

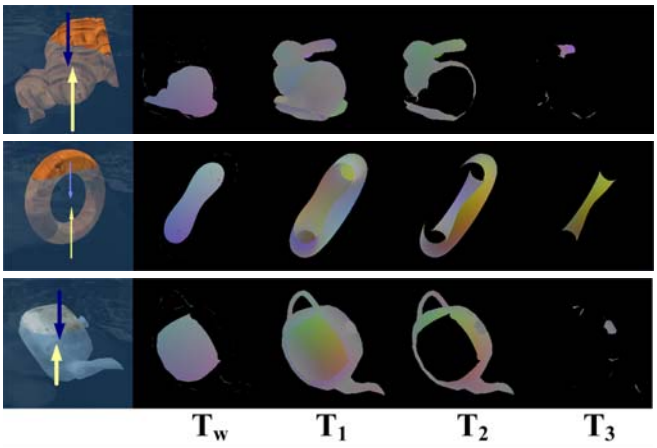


Fig. 10 Color encoded integrands of each peel for the buoyant bunny, torus, and teapot (left).

6 Conclusion

We have proposed a GPU-friendly algorithm for computing the mass properties of a rigid body represented by a general geometry. We formulated the mass properties as surface integrals on a projected plane, avoiding singularities at the boundaries. We also showed that depth-peeling techniques can be exploited to tackle non-convex geometries. Our approach is essentially image-based. Consequently, it can efficiently compute mass properties as long as the geometries can be rendered using graphics hardware.

We applied our algorithm to simulate rigid body motion in a real-time hydrostatic buoyancy simulation. The mass properties of the submerged volume were efficiently computed without an explicit reconstruction of the intersecting geometry between the fluid and the rigid bodies. Our algorithm approximates mass properties fairly accurately, even using low resolution framebuffers. Our interactive simulation demonstrates that the proposed algorithm can be applied to animate floating rigid bodies on a stationary fluid system in a fast and plausible way.

Acknowledgements The material presented herein is based upon work supported by the Information and Telecommunication National Scholarship Program supervised by IITA and the Ministry of Information and Communication, Republic of Korea.

References

1. Bolz, J., Farmer, I., Grinspun, E., Schröoder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans. Graph. **22**(3), 917–924 (2003)
2. Buck, I., Purcell, T.: GPU Gems 2, chap. A toolkit for computation on GPUs. Addison-Wesley (2004)
3. Carlson, M., Mucha, P.J., Turk, G.: Rigid fluid: Animating the interplay between rigid bodies and fluid. ACM Transactions on Graphics **23**(3), 377–384 (2004)
4. Everitt, C.: Interactive order-independent transparency (2001). URL citeseer.ist.psu.edu/everitt01interactive.html
5. Foster, N., Metaxas, D.: Realistic animation of liquids. Graphical Models and Image Processing **58**(5), 471–483 (1996)
6. Gonzalez-Ochoa, C., McCammon, S., Peters, J.: Computing moments of objects enclosed by piecewise polynomial surfaces. ACM Transactions on Graphics **17**, 143–157 (1998)
7. Hillesland, K.E., Molinov, S., Grzeszczuk, R.: Nonlinear optimization framework for image-based modeling on programmable graphics hardware. ACM Transactions on Graphics **22**(3), 925–934 (2003)
8. Krüger, J., Westermann, R.: Acceleration techniques for GPU-based volume rendering. In: VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), p. 38. IEEE Computer Society, Washington, DC, USA (2003)
9. Krüger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics (TOG) **22**(3), 908–916 (2003)
10. Layton, A.T., van de Panne, M.: A numerically efficient and stable algorithm for animating water waves. The Visual Computer **18**(1), 41–53 (2002)
11. Lee, Y.T., Requicha, A.A.: Algorithms for computing the volume and other integral properties of solids. I. Known methods and open issues. Communications of the ACM **25**(9), 635–641 (1982)
12. Mammen, A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. IEEE Computer Graphics and Applications **9**(4), 43–55 (1989)
13. Mirtich, B.: Fast and accurate computation of polyhedral mass properties. Journal of Graphics Tools **1**(2), 31–50 (1996)
14. Moreland, K., Angel, E.: The FFT on a GPU. In: HWWS '03: Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, pp. 112–119. Eurographics Association, Aire-la-Ville, Switzerland (2003)
15. Rost, R.J.: OpenGL Shading Language. Addison-Wesley Longman, Redwood City, CA (2004)
16. Thompson, C.J., Hahn, S., Oskin, M.: Using modern graphics architectures for general purpose computing: A framework and analysis. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture, pp. 306–317 (2002)
17. Wu, E., Liu, Y., Liu, X.: An improved study of real-time fluid simulation on GPU. Computer Animation and Virtual Worlds **15**(3–4), 139–146 (2004)

G80 Messaging – Final

NVIDIA® unified architecture revolutionizes PC graphics performance through unprecedented processing power and efficiency

- Fully unified shader core dynamically allocates processing power to geometry, vertex, physics, or pixel shading operations
    - Ground-breaking 128 parallel 1.35GHz* stream processors deliver amazing floating point processing power for unmatched gaming performance
    - Completely unified and optimized for current DirectX 9 and next generation DirectX 10 games and applications
- GigaThread™ technology provides extreme processing efficiency in advanced, next generation shader programs
    - Multi-threaded architecture supports thousands of independent, simultaneous threads, maximizing GPU utilization
- NVIDIA® SLI™-Ready – Up to 2x the performance of a single graphics card

World's first DirectX 10 GPU delivers unparalleled levels of graphics realism and film-quality effects

- Reference GPU for industry's DirectX 10 API development and certification
- Geometry shaders enable incredibly detailed, high polygon characters, shadows, and effects
    - Geometry creation and tessellation smooth curved surfaces and enable more lifelike character animation including realistic facial expressions and hair
    - GPU-generated shadow volumes deliver amazing performance improvements for shadow rendering
- Next generation geometry instancing provides extremely efficient batch processing of game objects and data and allows for richer and more immersive game environments
- New graphics data path enables rapid data storage (streamed output) for advanced shader calculations
- Full Shader Model 4.0 compliance delivers compatibility with DirectX 10 games

NVIDIA Lumenex™ engine delivers incredible image quality, floating point accuracy, and fast frame rates

- 16x full-screen anti-aliasing technology delivers superior AA quality while providing astounding performance
- 128-bit floating point HDR (high-dynamic range) with anti-aliasing provides twice the precision of previous generations while obliterating jaggies
- High speed memory interface and two dual-link DVI outputs enable extreme HD gaming up to 2560x1600 resolution at amazing frame rates

NVIDIA Quantum Effects™ technology enables a new level of physics effects to be simulated and rendered on the GPU

- Advanced shader processors architected for physics computation deliver amazing performance and visual effects such as smoke, fire, and explosions
- Realistic movement of hair, fur, and water is completely simulated and rendered by the graphics processor
- CPU is freed to run the game engine and AI, improving overall gameplay

PureVideo™ technology delivers the ultimate home theater experience on a PC

- Hardware acceleration for decoding H.264, VC-1, WMV and MPEG-2 movies delivers lifelike images that have up to six times the detail of standard DVD movies
- Dedicated video processor offloads the CPU and 3D engine of complex video tasks, providing a higher quality movie experience

- Post-processing support including advanced de-interlacing, scaling, noise reduction, and edge enhancement provides spectacular picture clarity and detail
- Provides world-class TV-out functionality via Composite, S-Video, Component, or DVI connections. Supports HD resolutions up to 1080p depending on connection type and TV capability

** GeForce 8800 GTX has 128 stream processors running at 1.35GHz. GeForce 8800 GTS has 96 processors running at 1.2GHz.*

NVIDIA CUDA for Thread Computing

NVIDIA® CUDA™ thread computing is a fundamentally new architecture to solve complex computational problems across consumer, business, and technical industries. CUDA technology gives data-intensive applications access to the tremendous processing power of NVIDIA graphics processing units (GPUs) through a revolutionary computing architecture unleashing entirely new capabilities. Providing performance increases up to 200% and simplifying software development through the standard C language, CUDA technology enables developers to create solutions for data-intensive processing to produce accurate answers, in less time. With the introduction of CUDA thread computing, consumers and professionals have faster access to powerful, decision making information that previously was simply not possible.

For more information on developing with CUDA technology, please visit http://developer.nvidia.com .

What is CUDA technology?

CUDA thread computing is an innovative combination of computing features in next generation NVIDIA GPUs that are accessible through a standard 'C' language. Where previous generation GPUs were based on "streaming shader programs", CUDA programmers use 'C' to create programs called threads that are similar to multi-threading programs on traditional CPUs. In contrast to multi-core CPUs, where only a few threads execute at the same time, NVIDIA GPUs featuring CUDA technology process thousands of threads simultaneously enabling a higher capacity of information flow.

One of the most important innovations offered by CUDA technology is the ability for threads on NVIDIA GPUs to cooperate when solving a problem. By enabling threads to communicate, CUDA technology allows applications to operate more efficiently. NVIDIA GPUs featuring CUDA technology have a parallel data cache that saves

frequently used information directly on the GPU. Storing information on the GPU allows computing threads to instantly share information rather than wait for data from much slower, off-chip DRAMs. This advance in technology enables users to find the answers to complex computational problems in real-time.

What applications benefit from CUDA?

CUDA thread computing is suitable for a wide range of applications that process massive amounts of information. For example, game applications take advantage of CUDA technology by leveraging the NVIDIA GPU to run the entire physics computation, letting gamers experience amazing performance and visual effects. In addition, commercial software applications used for product development or large data analysis, that previously required a supercomputer mainframe environment to run applications, can now benefit from using a standard workstation or server enabled with CUDA technology. This breakthrough in technology enables customers to make real-time analysis and decisions from anywhere. In addition, scientific applications which require the most intensive technical computing capability are no longer constrained by compute density; CUDA thread computing provides a platform with a higher level of performance from the same space requirements.

Developing with CUDA

The CUDA software development kit (SDK) is a complete software development solution for programming CUDA-enabled GPUs. The SDK includes standard FFT and BLAS libraries, a C-compiler for the NVIDIA GPU and runtime driver. The CUDA runtime driver is separate standalone driver that interoperates with OpenGL and Microsoft® DirectX® drivers from NVIDIA. CUDA technology is equally supported on both the Linux and Microsoft® Windows® XP operating system.

Why Use CUDA technology?

Performance. NVIDIA GPUs offer incredible performance for data-intensive applications. CUDA technology provides a standard, widely available solution for delivering new applications with unprecedented capability.

Productivity. Developers wanting to tap into the NVIDIA GPU computing power can now use the industry standard "C" language for software development. CUDA thread computing provides a complete development solution that integrates CPU and GPU software to enable developers to quickly provide new features and greater value for their customers.

Scalability. CUDA technology scales performance and features across the full line of NVIDIA GPUs from embedded form factors to high performance professional graphics solutions. The power of CUDA performance is now available in virtually any class system from large, computing installations to consumer products.

Technology Features

- Unified hardware and software solution for thread computing on CUDA-enabled NVIDIA GeForce® GPUs and NVIDIA Quadro® graphics boards
- CUDA-enabled GPUs support the Parallel Data Cache and Thread Execution Manager for high performance, thread computing
- Standard C programming language on a GPU
- Standard numerical libraries for FFT and BLAS
- Dedicated CUDA driver for computing
- Optimized upload and download path from the CPU to CUDA-enabled GPU
- CUDA driver interoperates with graphics drivers
- Supports Linux and Windows XP operating systems

- Scales from high performance professional graphics solutions to mobile and embedded GPUs
- Native multi-GPU support for high density computing
- Supports hardware debugging and profiler for program development and optimization