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.

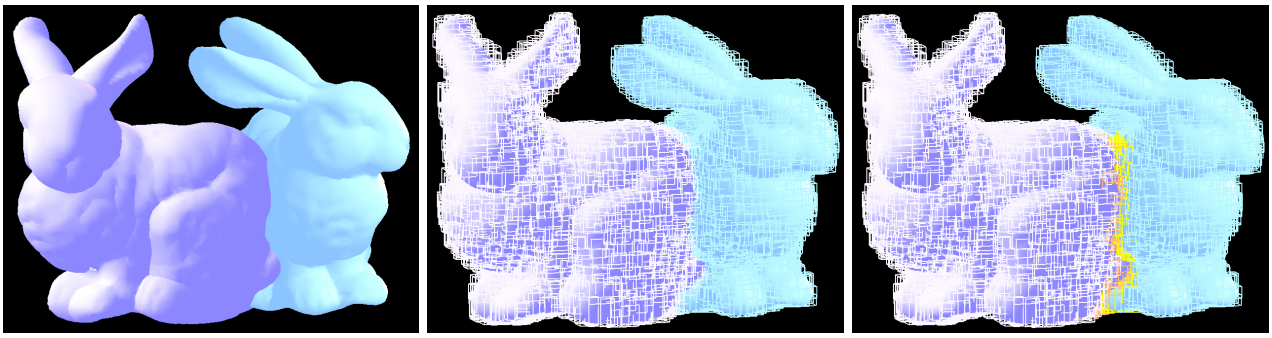
1 Introduction

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-

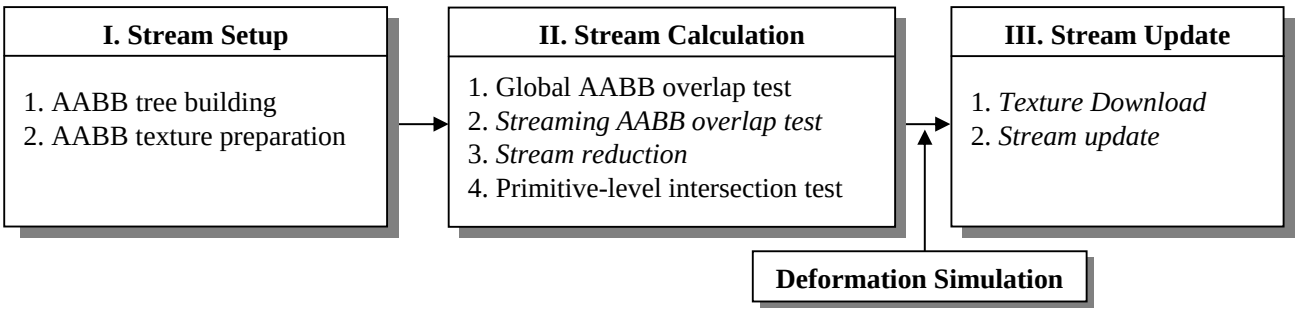


Fig. 2. The Streaming Collision Detection Pipeline. Stage I performs AABB stream setups. Stage II executes massively-parallel overlap test between AABB streams. Stage III updates the AABB streams as the underlying model deforms. The steps associated with streaming computations are *italicized*.

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.

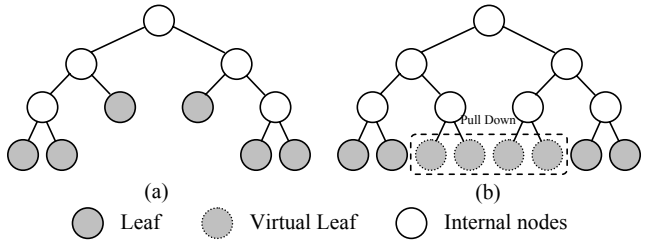


Fig. 3. Pulling Down Operation. (a) An original AABB tree before adjustment; (b) A complete binary AABB tree after adjustment by adding virtual leaves using pulling down operations.

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).

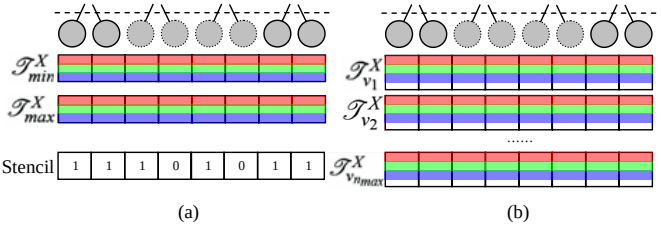


Fig. 4. Preparation of AABB Textures. (a) 1D AABB textures storing the upper and lower bounds of AABBs and a stencil array; (b) 1D vertex textures for unique vertices contained in AABB nodes.

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.

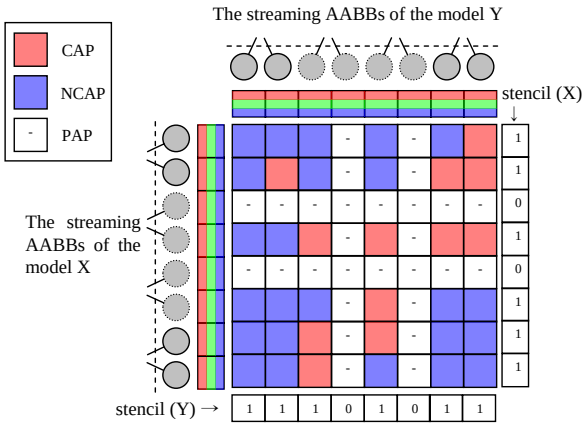


Fig. 5. AABB pair overlap tests on GPUs. The red blocks represent colliding AABB pairs (CAPs) and the blue blocks represent non-colliding AABB pairs (NCAPs). The white blocks marked as '-' represent prevented AABB pairs (PAPs) by the stencil buffer that is defined by the stencil array of the model X (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.

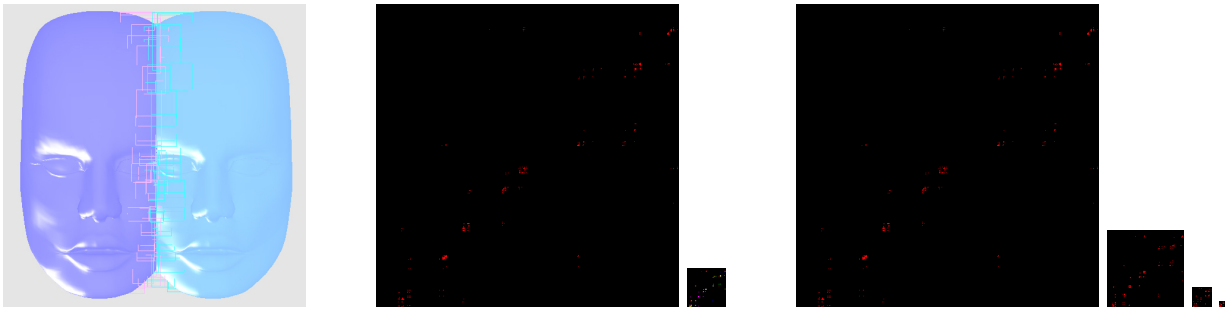
```
void streamingAABBTest ( float uvA: TEXCOORD0,
                         float uvB: TEXCOORD1,
                         out float4 color: COLOR,
                         uniform sampler1D minTextureA,
                         uniform sampler1D maxTextureA,
                         uniform sampler1D 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.

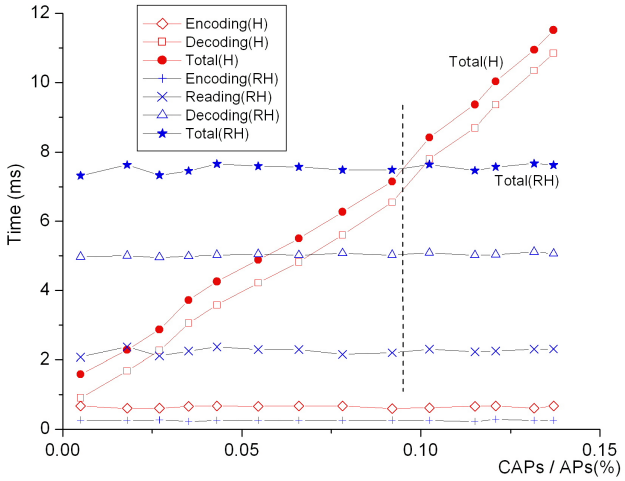


Fig. 7. Performance of variable hierarchical readback. Encoding(H) and Decoding(H) are the timings of encoding and decoding using the hierarchical readback scheme with three levels. Total(H) is their sum. Encoding(RH), Reading(RH) and Decoding(RH) are the timings of encoding, reading P_1', and decoding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

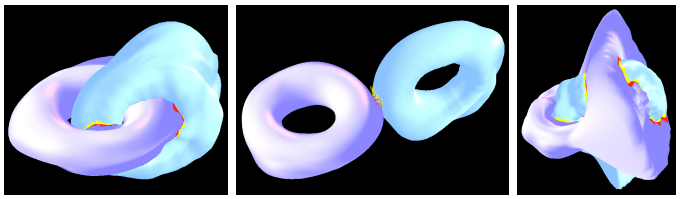
BVHs have been widely used for CD algorithms such as I-COLLIDE, RAPID, V-COLLIDE, SWIFT, SOLID 1.0, QuickCD, etc. However, these algorithms are designed for rigid bodies. In SOLID [2], AABB trees are used to handle collisions for deformable bodies. However, its timing statistics have showed that updating 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

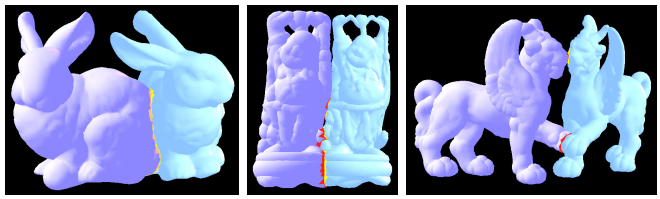
PERFORMANCE STATISTICS OF OUR ALGORITHM.

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).

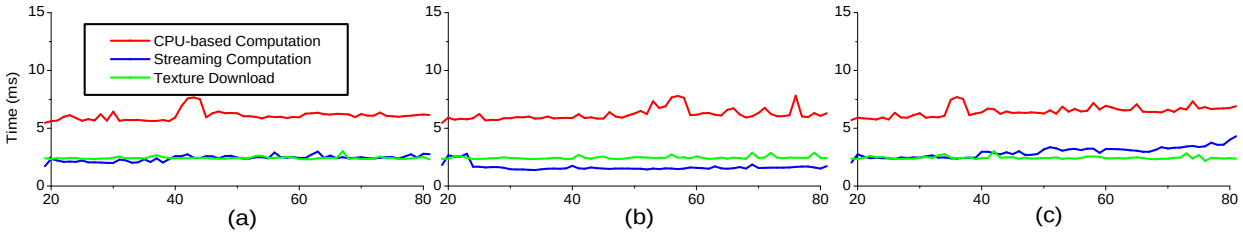


Fig. 10. Our Algorithm (StreamingCD) vs CPU-based AABB-tree Algorithm (SOLID). The graph compares the performance of SOLID [3] with ours for benchmarking set I: (a) Interlocking torii. (b) Touching torii. (c) Merging torii.

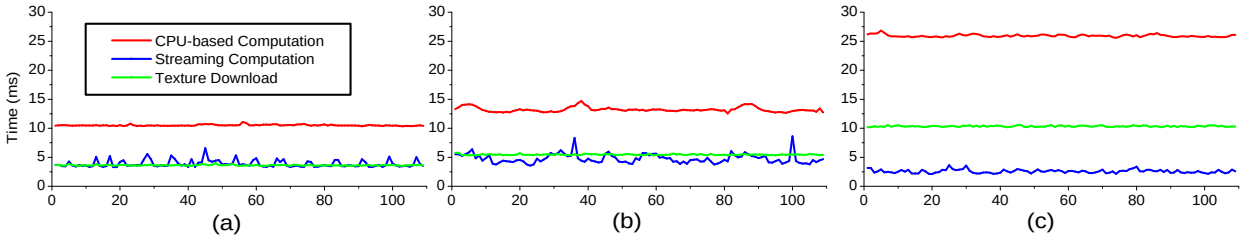


Fig. 11. Our Algorithm (StreamingCD) vs SOLID for Benchmarking Set II: (a) Bump Bunnies. (b) Happy Buddhas. (c) Intimate Animals.

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)

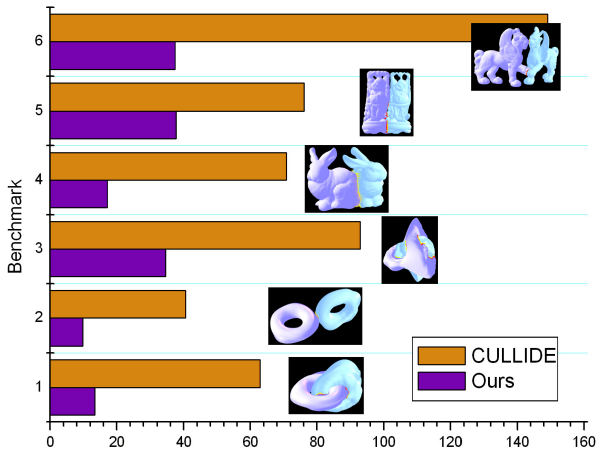


Fig. 12. Performance Comparison: Our Streaming CD vs CULLIDE. The graph compares the performance of CULLIDE with that of our algorithm to compute all the intersecting triangles under a same deformation scenario. On average, we have observed more than three times performance improvement of our algorithm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