INTERACTIVE COLLISION DETECTION FOR DEFORMABLE AND GPU OBJECTS

Joachim Georgii   Jens Krüger   Rüdiger Westermann

Computer Graphics & Visualization Group, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
{georgii, kruegeje, westerma}@in.tum.de

ABSTRACT

If two closed polygonal objects with outfacing normals intersect each other there exist one or more lines that intersect these objects at at least two consecutive front or back facing object points. In this work we present a method to efficiently detect these lines using depth-peeling and simple fragment operations. Of all polygons only those having an intersection with any of these lines are potentially colliding. Polygons not intersected by the same line do not intersect each other. We describe how to find all potentially colliding polygons and the potentially colliding pairs using a mipmap hierarchy that represents line bundles at ever increasing width. To download only potentially colliding polygons to the CPU for polygon-polygon intersection testing, we have developed a general method to convert a sparse texture into a packed texture of reduced size. Our method exploits the intrinsic strength of GPUs to scan convert large sets of polygons and to shade billions of fragments at interactive rates. It neither requires a bounding volume hierarchy nor a pre-processing stage, so it can efficiently deal with very large and deforming polygonal models. The particular design makes the method suitable for applications where geometry is modified or even created on the GPU.

KEYWORDS

Collision detection, Graphics hardware, Texture packing.

1. INTRODUCTION

While it is clear how to detect collisions between polygonal models under weak time constraints, there is an ongoing effort to develop techniques for interactive or even real-time applications. The difficulty arises from the fact that the size of dynamic 3D objects that can be rendered interactively has dramatically increased. Today, real-time raster systems can render moving objects composed of many millions of triangles at interactive rates. Such systems are used in many different areas of entertainment, industry, and research. Moreover, as graphics capabilities become more advanced, the list of applications is growing rapidly. These applications impose significant requirements on the collision detection system, and they require algorithms and data structures to deal with hard time constraints.

Over the last years there is also a growing demand for interactive collision detection between objects that can deform, and can thus self-interfere. Typical applications include surgery simulators, cloth simulation, virtual sculpting, and free-form deformations. Interactive collision detection between deforming objects is complicated because it requires frequent updates of the data structures commonly used to accelerate the detection process.

Even more important, geometric changes are increasingly performed on programmable graphics hardware using vertex programs and access to displacement textures (Gerasimov et al. 2004, Shiue et al. 2005, Botsch and Kobbelt 2005, Guthe et al. 2005). In this case, the changes in geometry might not even be known to the application program, which makes it difficult to maintain a data structure that appropriately represents the modified geometry. This problem aggravates due to the extended functionality of recent graphics hardware. With Direct3D 10 compliant hardware supporting geometry shaders (Balaz and Glassenberg 2005) it is even possible to create additional geometry on the graphics subsystem. In particular, triangle strips or fans

composed of several vertices can be spawned from one single vertex. By using this functionality the renderable representation cannot only be modified but it can be constructed on the fly on the graphics chip.

The implications thereof with respect to collision detection are dramatic: As large parts of the geometry will continuously be modified and created on the GPU, CPU algorithms relying on the explicit knowledge of the object geometry can no longer be used. As a consequence, collision detection must either be performed entirely on the GPU, or the information required to perform the collision test on the CPU has to be created on the GPU and downloaded to the CPU.

1.1 Contribution

In this paper, we present a collision detection algorithm for closed manifold meshes that addresses the aforementioned issues. We also show how this algorithm extends to the detection of collisions between arbitrary open 2D-manifolds. The method is especially designed for interactive handling of deformable objects and GPU objects, i.e., polygonal objects that are modified or constructed on the GPU.

Our algorithm is developed in regard of the observation that the Achilles' heel of almost all collision detection algorithms for deformable objects is the dynamic data structure used to represent the changing object geometry. We avoid the construction and repetitive update of such a data structure by shifting parts of the collision detection algorithm onto the GPU. Our algorithm takes as input a renderable object representation, and it only requires a GPU array containing the polygons to be rendered. In particular, this allows the method to handle geometry that is arbitrarily modified or created on the GPU. The proposed method proceeds in five passes:

1. **Object sampling**: Colliding objects are sampled along a set of rays via depth-peeling, and all rays along which a collision occurs are detected. This is done by exploiting the intrinsic strength of recent GPUs to interactively render high-resolution polygonal meshes and to efficiently perform simple fragment operations.
2. **Ray merging**: On the GPU a texture mipmap containing screen-space bounding boxes of ray-bundles at an ever increasing width is built.
3. **Primitive separation**: Primitives access the mipmap to test whether they are potentially colliding or not. For each primitive P, potentially colliding areas are encoded as screen-space bounding boxes in a texture map. From this information, the set of primitives P might interfere with can be computed efficiently. In particular due to this pass, the precision of interference computations is not constrained by the resolution of the frame buffer we render into.
4. **Texture packing**: The results generated in the previous pass are transferred to the CPU. To keep bandwidth requirements and CPU processing as low as possible the sparse texture map is first converted into a packed representation.
5. **Intersection testing**: Exact intersection testing is performed on the CPU. Per-primitive screen-space bounding boxes computed in pass three are used to prune most of the remaining primitive tests.
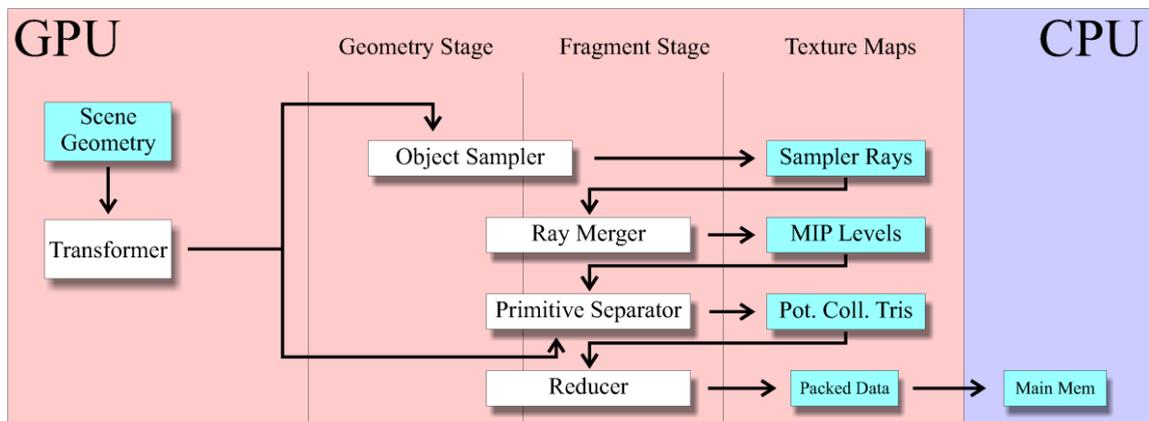

Figure 1: A diagrammatic overview of the proposed collision detection algorithm.

Figure 1 gives an overview of our collision detection process and illustrates how the proposed method fits the GPU stream architecture. It is designed as a pipeline of stages being successively applied to the stream of model geometry and generated fragments. In contrast to previous collision detection algorithms, it is a very unique feature of the proposed method that it can handle geometry that is arbitrarily modified or even created on the GPU. The input for the algorithm is a geometry stream as it is output by a geometry transformation unit, i.e., the vertex shader or the geometry shader on current graphics hardware. The result is a fragment stream consisting of only the potentially colliding primitives, which is written into a geometry texture.

This geometry texture is transferred to the CPU for exact intersection testing. Although it is in general possible to compute polygon-polygon intersections on the GPU, its data-parallel nature is not very well suited for the kind of operations required to determine the colliding partners out of the entire set of possible candidates. Therefore, a CPU-GPU hybrid method is employed, in which the CPU is responsible for exact intersection testing.

For many scenarios, the data transfer from the GPU to the CPU is most likely to become the bottleneck in the entire collision detection system. Thus, we present a novel GPU technique to convert a sparse texture into a packed texture of reduced size. The proposed technique converts an input stream containing a few randomly scattered valid data items into a stream consisting only of these items. As this stream is downloaded to the CPU, bandwidth requirements are considerably reduced. As the approach can also be used to filter out fragments not going to participate in upcoming rendering passes, it has a number of possible applications. At the end of Section 3 we will outline some of these applications.

2. RELATED WORK

Although a vast amount of literature has been published over the last decades, the efficient detection of collisions between large and dynamic polygonal models is still a fundamental problem in a number of different areas ranging from computer animation and geometric modeling to virtual engineering and robotics. For thorough surveys of the various species of collision detection algorithms let us refer here to the work by Lin and Manocha (2004) and Teschner et al. (2005).

According to the classification of collision detection algorithms into the three basic categories *static*, *pseudo-dynamic*, and *dynamic* (Held et al. 1995), our method belongs to the second class, as it detects collisions between moving objects at regular time intervals. While a large number of static and pseudo-dynamic techniques have been developed in the past, fewer approaches have explicitly addressed exact collision detection in time and space (Mirtich and Canny 1995; Lennerz et al. 1999; Redon et al. 2002; Coming and Staadt 2005).

Especially if a large number of objects have to be considered, collision detection algorithms usually proceed in a *broad phase* and a *narrow phase* (Smith et al. 1995; Cohen et al. 1995; Hubbard 1995). In the broad phase, approximate tests are performed to identify the potentially colliding objects out of the entire set of objects. This step effectively prunes the majority of all $O(\#objects^2)$ possible object-object tests. In the narrow phase, further tests identify the object primitives causing interference. Usually this is done in a hierarchical manner by considering several levels of intersection testing between two objects at increasing accuracy.

Hierarchical methods are often based on bounding volumes and spatial decomposition techniques. Such methods enable the efficient localization of those areas where the actual collisions occur, thus reducing the number of primitive intersection tests (Möller 1997). Over the last years a number of different variants of hierarchical representations have been used, such as bounding sphere hierarchies (Palmer and Grimsdale 1995; Hubbard 1996), axis aligned bounding boxes (van den Bergen 1997; Bridson et al. 2002), oriented bounding boxes (Gottschalk et al. 1996) and k-DOPs (Klosowski et al. 1998; Volino and Thalmann 2000).

In the context of deformable objects, the emphasis has been placed on the efficient update of hierarchical object representations (DeRose et al. 1998; Mezger et al. 2003; James and Pai 2004). In particular the idea to delay tree updates has been shown to be an efficient means to accelerate inter-object collision detection (Larsson and Akenine-Möller 2005). A method for continuous collision detection between deformable meshes based on graph theory has been suggested (Govindaraju et al. 2005a). At the core of this method is the partitioning of polygonal meshes into sets of independent primitives in which collisions can be detected in linear time. To avoid the time-consuming pre-processing of the previous approach, a method based on

Voronoi diagrams has been presented recently by the same group (Sud et al. 2006). The properties of Voronoi diagrams are utilized to determine the closest primitives in a given set of primitives. A GPU collision detection algorithm for deforming NURBS objects has been developed (Greß et al. 2006). This technique is based on a hierarchical AABB traversal scheme entirely implemented on the GPU.

The algorithm we propose finds its origin in the early idea of using rasterization hardware for interference detection between polygonal objects (Rossignac et al. 1992; Myszkowski et al. 1995). Building on this theory, methods based on voxels (Baciu and Wong 2002; Heidelberger et al. 2004) and view-frusta (Lombardo et al. 1999) have been proposed.

In comparison to previous approaches, our approach is similar to suggestions made by Knott and Pai (2003). By observing that scan-conversion algorithms only count the number of intersections between the polygon and the view ray up to the first point of the penetrating object, a variant that also detects additional points along these rays was proposed. This is achieved by rendering the penetrating object as wire-frame such that lines not entirely occluded by others "shine through" and leave a unique ID in the frame buffer. As the method distinguishes between penetrating and penetrated objects, it cannot handle self-intersections of one single deformable object.

Along a different avenue of research, hardware supported occlusion queries have been employed to accelerate collision detection (Govindaraju et al. 2003). Via an occlusion query the application program can request the number of fragments that survive the depth test in the rendering of a set of primitives. These queries can thus be used to accelerate the broad phase of collision detection, i.e., by testing whether two objects can be trivially rejected because they do not interfere in screen-space. To overcome sampling and precision errors that can result in collisions being missed, Govindaraju et al. (2004) proposed to "fatten" the objects, i.e., to extend the screenspace footprint of rendered primitives both with respect to the number of covered fragments and the primitives depth. The use of occlusion queries for intra-object collision detection including strategies to reduce the number of potentially colliding primitive pairs was proposed in (Govindaraju et al. 2005b).

Occlusion queries have been shown to be a very powerful means to reduce the number of potentially colliding primitive pairs. The reason why we decided not to use occlusion queries is threefold: First, as occlusion queries have to be issued by the application program, this mechanism seems to be problematic for the handling of collisions between GPU objects. Second, in the context of collision detection occlusion queries work most effectively if used in combination with a narrow-phase acceleration strategy on the CPU. This implies a dynamic data structure to be kept on the CPU, which is difficult to maintain efficiently in case of deformable objects or even GPU objects. Third, the effectiveness in pruning a majority of intersection tests strongly depends on the camera position and viewport. Thus, in general several views of the scene have to be rendered.

3. TEXTURE PACKING

Before we are going through the different stages of the proposed collision detection pipeline we will first describe the texture reduction stage—the Reducer in Figure 1. This stage implements a general method to convert a sparse texture into a packed texture that consists only of the non-empty texels in the sparse texture. The proposed method significantly distinguishes from the one presented by Greß et al. (2006) in that it does not rely on a global scattering pass and the sequence of operations is not data-dependent. In contrast to Ziegler (2006), our method has a better worst case complexity and thus performs better on denser textures.

The reduction stage, although it is not a mandatory stage in the proposed collision detection algorithm, is essential for our technique to perform most efficiently due to the following reasons: First, the packed texture can be downloaded to the CPU at much faster rates. Second, the processing of a large number of empty cells on the CPU can be avoided. The texture reduce operation on the GPU is accomplished in three stages:

- **Counting**: Non-empty texels per row are counted in a logstep reduce-add operation along texture rows. A singlecolumn texture storing these counts is read to the CPU (see Figure 2).
- **Shifting**: In each row non-empty texels are shifted to the right of the texture. All rows are processed in parallel by rendering a vertical line (see Figure 3).
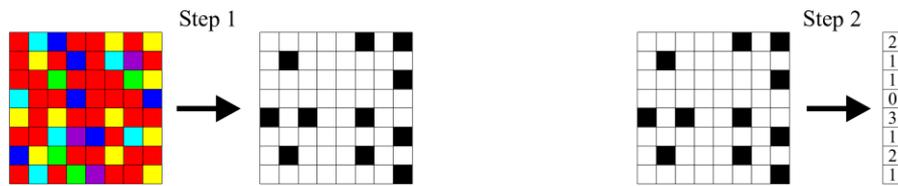
Figure 2: Counting: Step 1 executes a classification shader (selects only yellow texels here). Step 2 counts the positive texels per line and reads them back to the CPU.
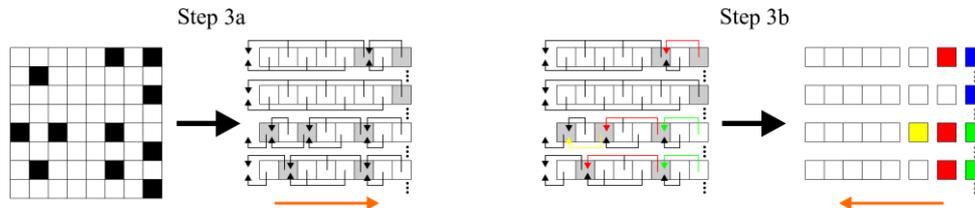

Figure 3: Shifting: First, a next-pointer list is constructed. Second, this list is traversed to gather texels from the appropriate position. The orange arrow depicts the traversal direction. For the sake of clarity only the shifting of every second line is shown.
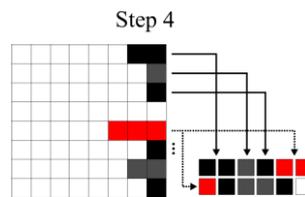

Figure 4: Moving.

- **Moving**: Packed rows are moved into the reduced texture. The texture is finally downloaded to the CPU (see Figure 4).

The application program computes the total number of non-empty texels from the single-column texture being read in the first step. This information is used to set the size of the reduced target texture. In addition, the maximum number M of non-empty texels per row is computed.

Then, the sparse texture is reduced horizontally. This is done in two passes, the first of which proceeds from left to right and the second from right to left (see Figure 3). To do such a reduction on the CPU we would simply traverse each row from left to right, keeping a pointer to the current element in the reduced row and copying the next non-empty element to this position. Unfortunately such a copying (or scattering) operation is not available on recent GPUs so that we have to convert this operation into a gather operation. Therefore we first sweep over the texture from left to right and store into each texel the position of the preceding non-empty texel in the same row. Before the first non-empty entry is encountered, a special key is stored. In the second pass we sweep from right to left for M steps. If the rightmost texel in a row is not empty we write zero into the texture, otherwise the address found in that texel is written (Figure 3, blue texels). In all subsequent sweeps the address at the preceding position in the same row is dereferenced first, and the retrieved address is written to the render target. Sweeping is accomplished by rendering a vertical line primitive covering as many pixels as there are rows in the sparse texture. As we only need to access a single address per line, which can be stored in one component of a RGBA color value, with every line four columns can be processed at once.

After the horizontal reduction, the contiguous sets of texels in each row of the sparse texture have to be copied into a render target of reduced size. This is done by rendering for each row in the sparse texture a horizontal line covering as many pixels as there are non-empty texels in this row. If the size of the packed

texture is $X \times Y$ and rendering the first L lines has generated N fragments, then the $(L + 1)$-th line starts at position (N mod X, N div X) in the target texture. If the length of this line is larger than $X − (N$ mod $X)$ it has to be split into two or more lines of reduced length. This is illustrated in Figure 4. The fragments generated for each line can fetch the corresponding values from the sparse texture via appropriately chosen texture coordinates. The read values are copied to the render target as shown in Figure 4. Due to performance issues, the application program creates a vertex array containing all the required information and renders this array using one single call.

Applications

The need for a texture reduction scheme on the GPU is paramount in a number of graphics applications, where data is modified or generated on the GPU and has to be processed further on the CPU. Popular examples include physics on GPUs (Green and Harris 2006), where the results of approximate occlusion queries and collision response calculations are used in the broad phase of collision detection on the CPU.

The texture reduction scheme provides a powerful mechanism to minimize the bandwidth requirements in the proposed collision detection algorithm. On the other hand it can also be used to discard fragments not going to participate in upcoming rendering passes on the GPU. In typical fragment-based rendering algorithms it is often the case that the fragment program is conditionally executed by only a small fraction of the entire set of fragments generated by the rasterizer. Potential applications include adaptive techniques for texture filtering, rendering, or numerical simulation on the GPU. A similar texture reduction technique based on histogram pyramids has been presented recently by Ziegler et al. (2006). In particular, they use their technique to generate point clouds of arbitrary geometries for particle explosions.

As an alternative to the proposed texture reduction technique, fragments can also be discarded using advanced GPU features like the early-z test or breaks in the fragment stage. Unfortunately, on current graphics cards early-out mechanisms introduce some overhead, i.e., either a branch instruction or additional rendering passes. Moreover, since the pixel shader hardware runs in lock-step, a performance gain can only be achieved if all fragments in a contiguous array exit the program early. These observations are backed up by latest GPUBench (Buck et al. 2004) results. Results for our target architecture, the GeForce 7800 GTX, are available at `http://graphics.stanford.edu/projects/gpubench/results/`. These results attest current GPUs a rather bad branching performance even if all fragments in a 4×4 block exit the program simultaneously.

As a matter of fact we believe that the proposed texture packing has the potential to become a general means to efficiently filter out fragments from a generated fragment stream. Thus, the method is not only highly beneficial in hybrid CPU-GPU approaches to reduce bandwidth requirements but also in pure GPU techniques to minimize the load in both the rasterization and the fragment stage.

4. OBJECT SAMPLING

In the first pass of the proposed collision detection algorithm the polygonal scene is sampled to detect rays along which at least one potentially colliding polygon is hit. These rays will subsequently be called collision rays. Here we are searching for rays that have either at least two consecutive hits with a front facing polygon or at least two consecutive hits with a back facing polygon, or that first hit a back facing polygon.

The underlying theoretical basis of this method is given by the generalization of Jordan's theorem to higher dimensions. A closed polyhedron P separates space into an "inside" and an "outside." If it has outfacing normals, any ray starting outside of P and intersecting P has alternating front and back facing intersection points (silhouette points are ignored). At the front facing intersection points the ray enters P and at the back facing intersection points it is leaving P. If along a ray two consecutive front or back facing intersection points are found, the ray enters a second object at the second front facing point before the first object was left, or it leaves an object but was still in another object 1. This also holds for an arbitrary number of objects. Examples of (self-) interference are illustrated in Figure 5 on the left.

To detect intersecting closed polyhedra, it is therefore sufficient to detect consecutive front facing polygons or consecutive back facing polygons (silhouette polygons are ignored), or a back facing polygon as first hit along any ray starting outside the potentially colliding set. To detect all collisions, the space in which the polygons exist has to be sampled as densely as possible.
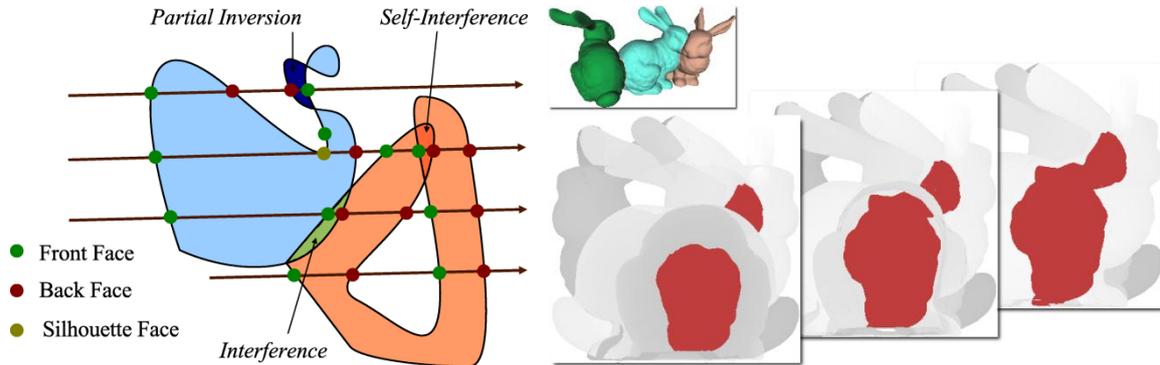
Figure 5: Illustration of interference, self-interference and partial inversion (left). By using depth-peeling and fragment operations to detect consecutive front or back faces, collision rays in each layer are determined (colored red, right image).

4.1 Depth-Peeling

Although the rays being used to sample the objects' faces can be chosen arbitrarily, an uniform sampling along parallel rays leads to the most isotropic sampling in object space. Scanline rendering algorithms simulate this by projecting the objects along an arbitrary, but constant direction. To detect consecutive pairs of front or back faces, all faces have to be rendered in correct visibility order with respect to an infinite viewer in the direction of projection. Depth-peeling (Everitt 2001) is employed to achieve this ordering.

The use of depth-peeling to track intersection events in an ordered way was suggested before by Guha et al. (2003) and Hable and Rossignac (2005) in the context of GPU-based CSG rendering. By tracking the state of intersection points between view rays and bounded solids, i.e., entering (1) or leaving (0), the state of a boolean expression of these events can be determined correctly at any point along the rays. We are interested in extending this idea to find consecutive events of the same state.

Depth-peeling requires multiple rendering passes. For each pixel, in the n-th pass the (n−1)-th nearest fragments are rejected in a fragment program and the closest of all remaining fragments is retained by the standard depth test. A floating-point texture map—the depth map—is used to communicate the depth of the surviving fragments to the next pass. A detailed description of this method is given by Everitt (2001). The number of rendering passes is equal to the objects' depth complexity, i.e., the maximum number of object points being projected to a single pixel. The depth complexity can be determined by rendering the objects once and by counting the number of fragments being projected to each pixel during scan-conversion. The maximum coverage of all pixels is then collected in a log-step reduce-max operation (Krüger and Westermann 2003).

To detect two front or two back faces in consecutive rendering passes it is sufficient to store for each entry in the depth map an additional tag that indicates the expected facing of the respective fragment. The expected facing is determined as alternating front and back facing states. In addition to only comparing the current depth of the fragment with the value stored in the depth map, a fragment shader now also compares the expected facing to the actual one. If they differ, the fragment is marked as a collision ray and it is discarded in upcoming rendering passes.

The modified depth-peeling technique generates a texture map—the sampler-ray—in which for all collision rays the status is set to "on", and "off" otherwise (see Figure 5 right). As the sampling rate is constrained by the resolution of the frame buffer we render into, some interfering primitives, and thus collision rays, might not be detected. This problem can be weakened by increasing the resolution to its maximum size, but it probably still exists. On the other hand, a collision ray is only missed if all interfering primitives along that ray are missed. If at least one of the intersections is detected, the upcoming stage of the collision detection process will find all intersecting primitives.

4.2 Advanced Depth-Peeling

Depth-peeling can be optimized by exploiting the functionality of recent graphics hardware. One of the key novelties of Direct3D 10 capable hardware (e.g. NVIDIA 8800) is a new programmable stage—the geometry shader—in the rendering pipeline. In contrast to the vertex shader, the geometry shader takes as input an entire graphics primitive (e.g. a triangle or a triangle and its neighbors) and outputs zero to multiple new primitives. This is achieved by letting the geometry shader append the new primitives to one or multiple output streams. In particular, multiple output streams called render targets (MRTs) can be bound to the geometry shader. Note that these MRTs are different from the ones known in the pixel shader stage. Each geometry shader MRT exhibits its own rasterizer, pixel shader stack, and depth/color buffers.

We can reduce the load of the geometry stage by peeling simultaneously from the front and from the back. This is achieved by issuing two output streams from the geometry stage, where one of the stream gets assigned the depth values z and the other the depth values $1-z$. In the fragment stage, for each of the peeling layers the facing of the fragment is compared with the expected facing (if peeling from the back, we initially start with back facing fragments). Finally, the two generated textures containing the collision rays are merged. Due to this optimization, only the half number of peeling steps is required, thereby nearly doubling the performance of the object sampling stage.

5. RAY MERGING

To determine potentially colliding primitives, i.e., polygons that are hit by a collision ray, the information being generated on a per-ray basis in screen-space has to be carried over to the set of polygons. One possibility is to rasterize one fragment for every polygon and to compute the rays intersecting the polygon in a fragment shader. The status of each ray can be retrieved from the sampler-ray, and the primitive gets assigned a flag indicating whether it is hit by at least one collision ray or not. Unfortunately, from this information alone it is quite cumbersome to determine the set of potentially colliding primitive pairs being needed for exact intersection testing. Moreover, due to the different size of triangles in screenspace, the GPU's fragment processing cannot reach its maximum performance.

Therefore, we propose a more efficient strategy. At the core of this strategy is the idea to generate the information required to efficiently determine the set of potentially colliding primitives for each polygon. This inter-object relation is established via the collision rays. In general, every polygon is hit by many collision rays. Thus, for a particular polygon several of these rays are required to detect all potentially colliding partners. In the following we describe a simple GPU data structure in which the relations between a primitive and all of its potential partners are encoded in one single ray bundle. An example of this mipmap texture is given in Figure 6.

The data structure consists of ray bundles at ever increasing width. For every collision ray that is "on" the screen-space bounding box bb = $(x_<, y_<, x_>, y_>)$ of the pixel this ray is passing through is computed. The first and last two components of the quadruple specify the left-bottom and the top-right corner of the bounding box. Bounding boxes of rays that are "off" are set to $(1, 1, 0, 0)$, such that they do not affect the union with any other box. Bounding boxes are rendered into an RGBA 16 Bit floating point texture of the same size as the sampler-ray. From this texture a mipmap hierarchy is generated by computing at each level l the union of bounding boxes of the 2×2 corresponding texels at level $l-1$. The union of two bounding boxes is calculated as

$$bb^1 \cup bb^2 = \left(\min(x_<^1, x_<^2), \min(y_<^1, y_<^2), \max(x_>^1, x_>^2), \max(y_>^1, y_>^2) \right)$$

This process is performed recursively until only one bounding box is left (see Figure 6). For the sake of simplicity we assume the initial texture size to be a power of two, and we limit ourselves to quadratic textures. The mipmap finally stores screen-space aligned bounding boxes of ray bundles, where a bundle only contains those rays that have been marked as "on". This mipmap can be used to find all potentially collision rays of a particular primitive as explained next.

Figure 6: Mipmap construction: On the finest level, the screen-space bounding boxes (red) are set to the pixel border of each collision ray (blue), or the texels are empty otherwise. For 2×2 texels the union of their boxes (green) is calculated to obtain a bounding box (red) at the next coarser level.

6. PRIMITIVE SEPARATION

To find the ray bundle that contains all colliding rays intersecting a certain primitive, we first compute the minimum mipmap level where the screen-space extent of one texel is larger than the screen-space bounding box of the primitive itself (see Figure 7).

To efficiently find this level, we rasterize one fragment for every polygon and we implement a pixel shader that computes the primitives screen-space bounding box. From the extent of this box the appropriate mipmap level is derived. As a primitive can overlap multiple ray bundles at this level, at every corner of the primitive screen-space bounding box one bundle along with its screen-space bounding box is fetched from the mipmap hierarchy. (If the corresponding texels are still neighboring at the next finer mipmap level, this level can be used instead.) The union of these boxes is then determined, and the resulting bounding box is intersected with the screen-space bounding box of the triangle. The coordinates of this box are stored in a target texture. If only empty ray bundles are fetched from the mipmap, the respective entry in the render target is set to zero, resulting in a texture that is sparsely filled.
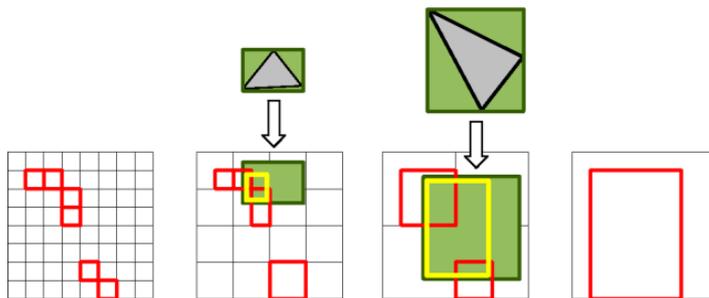

Figure 7: Primitive separation: The screen-space bounding box of two triangles is shown in green. The maximum extent of these boxes yields the mipmap level. The mipmap is sampled at every corner of the box and the resulting bounding boxes (red) are combined to a single box. The intersection with the triangle bounding box is output (yellow).

7. DATA TRANSFER AND INTERSECTION TESTING

The texture that is generated in the previous pass has to be transferred to the CPU for exact intersection testing. By applying the technique described in Section 3 a packed texture containing the set of potentially colliding triangles is generated on the GPU. Every texel stores an unique triangle ID as well as the screen-space bounding box of the set of rays intersecting this triangle. The packed texture is finally read to the CPU for intersection testing.

After having received the texture containing all potentially colliding primitives, a sweep-and-prune strategy (Cohen et al. 1995) is utilized to determine the colliding primitive pairs. These are the primitives whose bounding boxes overlap along each of the three screen-space axes. The extent of the bounding boxes in screen-space z-direction is only computed if an overlap along the x- and y-direction has been detected. For those primitives being detected a triangle-triangle intersection test is performed (Möller 1997), and for each triangle a response vector taking into account its own normal and the normal of the colliding partner is computed.

8. RESULTS

We have tested the proposed collision detection algorithm in three different scenarios consisting of several thousands up to a million triangles. All of our tests were run on an Intel Core 2 Duo equipped with a NVIDIA GeForce 7800 GTX using a 1K×1K frame buffer to sample the objects along parallel view rays. All objects are encoded as indexed vertex arrays stored in GPU memory.

On the basis of three test scenes we will demonstrate the overall performance of our algorithm before we present a more detailed analysis. We verified that in the examples shown all intersecting primitive pairs at one time step were detected. However, because our algorithm belongs to the class of pseudo-dynamic collision detection algorithms some collisions might be missed due to the movement of objects.

Deformable object collisions: In the first example the collision detection algorithm was integrated into an interactive deformable bodies simulation. Collision detection was performed on the boundary surfaces of tetrahedral objects. In Figure 8 two snapshots of an animation sequence with bunny and horse models are shown. The depth complexity of the scenes along the collision rays is 8. All collisions between the deforming objects were detected in less than 40 ms.

GPU object collisions: To demonstrate the ability of the proposed algorithm to deal with geometry that is modified or even created on the GPU we have used a scene that is made of dynamic meshes being procedurally deformed on the GPU. Figure 9 shows this scene consisting of artificial creatures made of a spherical body and a number of moving tentacles attached to it. Tentacles are animated and deformed by a vertex shader program on the GPU. Rigid starships try to pass these creatures. Upon collision with any other creature or any of the shuttles, tentacles retract and start growing again. The overall scene consists of 320k triangles.

In contrast to all other examples, for every potentially colliding triangle that is detected on the GPU the three vertex coordinates of this triangle along with its normal and a unique tentacle ID are stored in different texture maps. These textures are then packed and downloaded to the CPU. In this way the CPU can control the retraction of tentacles via shader parameters, and it can simulate rigid body motion of the shuttles. All collisions were detected in about 25 ms.

Rigid body collisions: Our last example demonstrates the capability of our method to handle collisions between rigid bodies (see Figure 10). Although we are aware of the fact that optimized CPU collision detection algorithms are probably more suited to this particular application, the given examples allow for a clear analysis of the different parts of our method.

The scene consists of 60 rigid bunnies moving through space due to gravity and collisions. The entire scene consists of half a million triangles. The depth complexity of the scene as seen in the image is 16. The detection of all collisions in the scene took 95 ms. We have also run the experiment by restricting the number of rendering passes performed by the depth-peeling routine to six. In this case, the performance increased to 68 ms, and even more interestingly only less than 5% of all penetrating triangle-triangle pairs were missed. Alternatively we have stopped depth-peeling after seven passes and all fragments with a depth count greater than 6 were marked as potentially colliding. In this case the performance dropped down slightly, but on the other hand all triangle-triangle collisions were again resolved. The efficiency of our approach is further demonstrated by the last example in Figure 10. Even for a triangle count of 800K and a number of 50K potentially colliding primitives processed on the CPU the method is still able to achieve about 4 frames per second.

8.1 Analysis

A detailed statistic of all test scenes is presented in Table 1. The overall triangle count is given in the first column. The following columns present the number of collision rays, the number of triangles which are downloaded to the CPU, the number of triangles that survive the CPU pruning of overlapping pairs of primitive bounding boxes, the number of primitive intersection tests and the number of detected collisions. Representative timings for collision detection in the example scenes are listed in Table 1 on the right. All timings are given in milliseconds (ms). The first column shows the amount of time spent by the GPU for *object sampling*, *ray merging* and *primitive separation*. The time required for reducing the sparse texture and
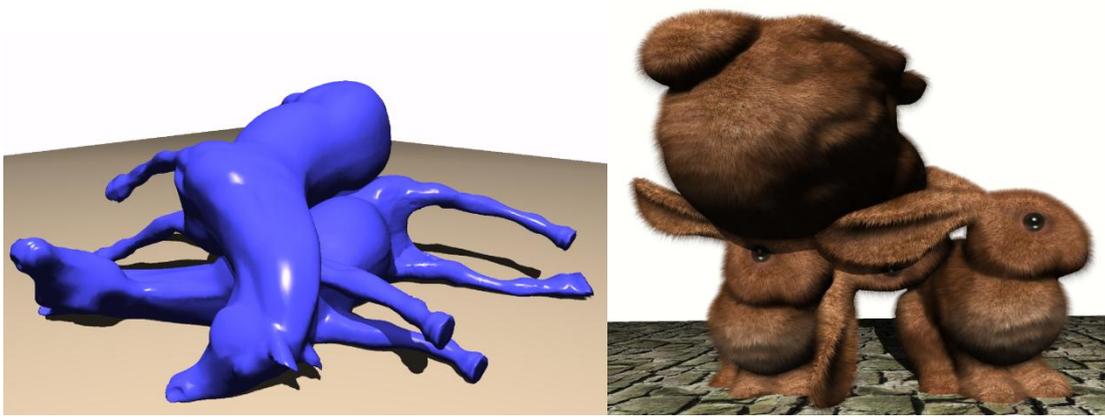
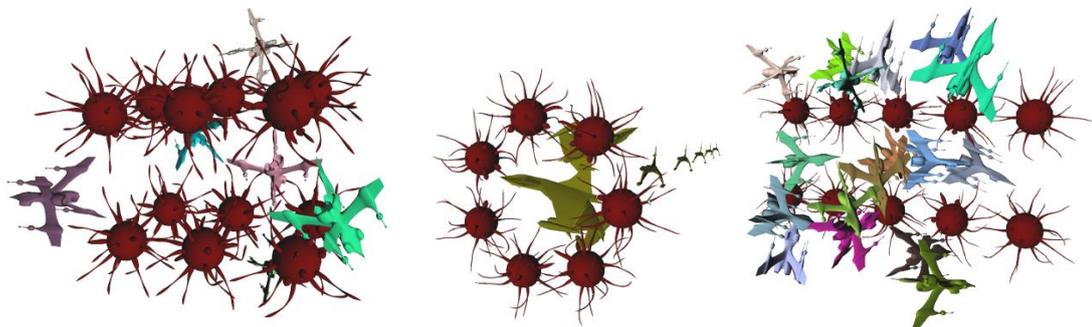Figure 8: (Self-) Collisions between deformable objects

Figure 9: Collisions between dynamic GPU objects.

Figure 10: Collisions between rigid objects.

downloading the packed texture to the CPU is given next, followed by the time required for CPU processing on a single core. Finally, the overall performance is specified in frames per second (fps).

In comparison to previous work on GPU-based collision detection between CPU objects, the examples we have shown indicate comparable performance. It can be observed, however, that our algorithm has the tendency to generate a larger set of potentially colliding primitives on the GPU, which then has to be processed on the CPU. Due to the fact that we sample the scene along parallel rays through each pixel, we might also miss collisions if the size of polygons is below the pixel size. On the other hand, such inaccuracies can only occur if all interfering primitives under one pixel are missed. If at least one of them is detected all intersecting primitives will be detected in the upcoming stage of the collision detection process. By "fattening" the objects similar to the method proposed in (Govindaraju et al. 2004) this problem can be solved entirely, but it comes at the expense of an increasing number of false positives.

An advantageous feature of our method is that only a very simple data structure is required to determine potentially colliding primitive pairs. Besides GPU-friendly geometry representations like vertex arrays or display lists that are used for depth-peeling, an indexed face set and a shared vertex array is needed. Updates of the geometry only require the vertex array to be updated accordingly.

As can be seen in the rigid bunny scene, depth-peeling can become the performance bottleneck if the depth complexity of the scene is increasing. On the other hand, this limitation only becomes severe if very large objects with high depth complexity have to be tested, because we can use GPU-resident geometry representations for rendering. In typical real-time scenarios, e.g., virtual surgery simulators or computer games, such scenes are not very likely to occur. It is also worth noting that collision detection between many rigid bodies can be improved by means of any appropriate broad-phase strategy. Moreover, due to the improved functionality of recent graphics hardware (e.g. NVIDIA 8800) the time required by the depth-peeling can be halved by exploiting multiple render targets as described in Section 4.2.

In contrast to previous collision detection algorithms, our method can handle geometry arbitrarily modified or created on the GPU. This is demonstrated by our second example above. In this case the GPU sends the modified geometry of all potentially colliding polygons to the CPU, thus taking advantage of the texture reduction we have presented. If additional geometry is created on the GPU, e.g., by using geometry shaders, only slight modifications to our algorithm are required. In particular, as the entire geometry has to be stored in one container, i.e., a texture map that can be accessed in step 3 of the collision detection algorithm, this container has to be generated in an intermediate rendering pass. All stages of the collision detection can then be executed as described. In case that the colliding geometry is not known to the CPU, computing an adequate collision response is of course a more complicated task.

Table 1. Triangle counts and timing statistics of the test scenes (NVIDIA 7800 GTX).

Scene	# tri's	# coll. rays	# tri's downl.	# pot. coll. tri's	# tri-tri tests	# pos. tests	GPU	Reduce	CPU	Overall time
defo bunnies	12K	0.2K	0.4K	0.2K	0.4K	0.1K	3ms	1ms	1ms	5ms/200fps
defo horses	32K	21K	13K	4.3K	9.0K	3.7K	5ms	4ms	24ms	33ms/30fps
art. creatures	320K	0.3K	1.0K	0.4K	1.5K	0.2K	17ms	6ms	1ms	24ms/42fps
rigid bunnies	500K	2.8K	12K	4.4K	7.7K	1.5K	56ms	18ms	21ms	95ms/11fps
dragon	800K	7.7K	54K	7.5K	6.6K	1.5K	53ms	32ms	139ms	224ms/4.4fps

8.2 Non-Closed Polygonal Objects

Our collision detection algorithm has been developed for closed polygonal objects. However, collision detection is also required in environments where GPU objects, deformable objects, and rigid bodies interact with each other. Moreover, these scenes often include non-closed objects, such as the ground, walls, or cloth patches. To be able to deal with such environments, we present an extension of our algorithm that allows for the handling of non-closed objects. This extension comes at the expense of an increasing number of false positives to be downloaded to the CPU. Therefore, if used to detect collisions between open 2D manifolds only, we do not expect the method to perform superior compared to CPU collision detection algorithms that have been optimized for the handling of such meshes.

In general, an arbitrary open polygonal object can be closed by constructing a tight enclosing hull around it. We avoid this construction by slightly changing the way the collision rays are determined in the object sampling stage: a potentially colliding ray is detected if two consecutive fragments are in close depth proximity. In general, due to this modification a greater number of collision rays, i.e., false positives, is produced, resulting in increasing bandwidth requirements as well as a higher load on the CPU to finally detect the intersecting triangle pairs. The result of the proposed modification is demonstrated in Figure 11, where a cloth patch self-interferes when colliding with a sphere object. The cloth patch consists of 2k triangles. Up to 2k triangle-triangle tests have to be performed on the CPU, and up to 300 intersecting triangle pairs have been detected. The maximum time spent for collision detection and response in one single time step of the animation sequence (the cloth patch falling down on the sphere) was 12 ms. The average time per-frame was only 5 ms.
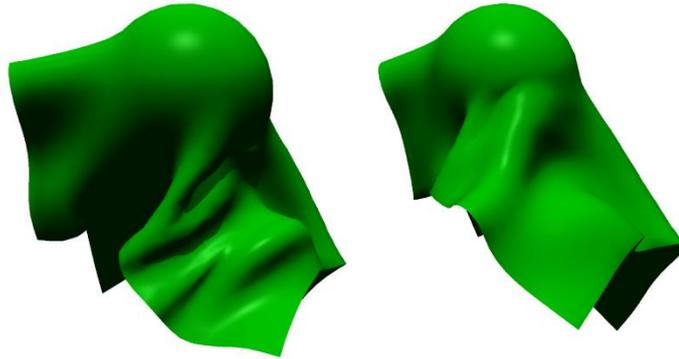
Figure 11: (Self-) Collisions between non-closed dynamic and rigid objects. The cloth patch consists of 2k triangles. Up to 2k tri-tri tests have to be performed on the CPU, and up to 300 intersecting triangle pairs have been detected.

9. CONCLUSION AND FUTURE WORK

In this paper we have presented a collision detection algorithm that is particularly designed for recent and future graphics hardware. It exploits the intrinsic strength of GPUs to scan-convert large sets of polygons and to shade billion of fragments at interactive rates. The design we suggest makes the method suitable for applications where geometry is deformed or even created on the GPU. The possibility to deal with dynamic scenery and scenery that is not known to the application program distinguishes the technique from previous approaches. In a number of different examples these statements have been verified.

We believe that the suggested algorithm is influential for future research in the field of collision detection. For the first time it has been shown that collision detection between objects modified or created on the GPU can successfully be accomplished. With Direct3D 10 compliant hardware and geometry shaders being available this feature will be required in many different applications. In contrast to previous GPU-based algorithms for collision detection, all objects can remain in their renderable representation and do not have to be converted into another format. The burden of frequently updating hierarchical data structures is taken off the application program.

The proposed texture reduce operation has the potential to greatly influence other applications. As this operation can be used to filter out fragments not going to participate in upcoming rendering passes, these passes can run more efficiently on the reduced set. In the future we will investigate the benefits of this operation to applications ranging from physics on GPU to adaptive rendering techniques.

REFERENCES

G. Baciu and W. S.-K. Wong, 2002, Hardware-assisted self-collision for deformable surfaces. In *Proceedings of the ACM symposium on Virtual reality software and technology*.

R. Balaz and S. Glassenberg, 2005, "DirectX and Windows Vista Presentations," http://msdn.microsoft.com/directx/archives/pdc2005/.

M. Botsch and L. Kobbelt, 2005, Real-time shape editing using radial basis functions. In *Proceedings of Eurographics'05*.

R. Bridson, R. Fedkiw, and J. Anderson, 2002, Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of SIGGRAPH '02*.

I. Buck, K. Fatahalian, and P. Hanrahan, 2004, GPUBench: Evaluating GPU performance for numerical and scientific applications. In *ACM Workshop on General-Purpose Computing on Graphics Processors*.

J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgil, 1995, I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the symposium on Interactive 3DGraphics '95*.

D. S. Coming and O. G. Staadt, 2005, Kinetic sweep and prune for collision detection. In *Workshop On Virtual Reality Interaction and Physical Simulation*.

T. DeRose, M. Kass, and T. Truong, 1998, Subdivision surfaces in character animation. In *SIGGRAPH '98*.

C. Everitt, 2001, Interactive order-independent transparency. NVIDIA Corporation, Tech. Rep.

P. Gerasimov, R. Fernando, and S. Green, 2004, Whitepaper: Shader Model 3.0 Using Vertex Textures. http://developer.nvidia.com/object/using_vertex_textures.html.

S. Gottschalk, M. C. Lin, and D. Manocha, 1996, OBBTree: a hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH '96*.

N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha., 2003, Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics conference on Graphics hardware*.

N. K. Govindaraju, M. C. Lin, and D. Manocha, 2004, Fast and reliable collision culling using graphics hardware. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*.

N. K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M.C. Lin, and D. Manocha, 2005a, Interactive collision detection between deformable models using chromatic decomposition. *ACM Transaction on Graphics*, 24(3).

N. K. Govindaraju, M. C. Lin, and D. Manocha, 2005b, Quick-CULLIDE: Fast Inter- and Intra-Object Collision Culling Using Graphics Hardware. *IEEE Virtual Reality Conference 2005 (VR '05)*.

S. Green and M. Harris, 2006, Physics simulation on NVIDIA GPUs. http://developer.nvidia.com/object/havok-fx-gdc-2006.html.

A. Greß, M. Guthe, and R. Klein, 2006, GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum*.

S. Guha, S. Krishnan, K. Munagala, and S.Venkatasubramanian, 2003, Application of the two-sided depth test to CSG rendering. In *Proceedings of the symposium on Interactive 3D graphics '03*.

M. Guthe, Á. Balázs, and R. Klein, 2005, GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Transactions on Graphics* 24(3).

J. Hable and J. Rossignac, 2005, Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes. *ACM Transactions on Graphics* 24(3).

B. Heidelberger, M. Teschner, and M. Gross, 2004, Detection of collisions and self-collisions using image-space techniques. *J. WSCG 12(3)*.

M. Held, J. Klosowski, and J. Mitchell, 1995, Evaluation of collision detection methods for virtual reality fly-throughs. In *Seventh Canadian Conference on Computational Geometry*.

P. M. Hubbard, 1995, Collision detection for interactive graphics applications. *IEEE Trans. on Visualization and Computer Graphics*, 1(3).

P. M. Hubbard, 1996, Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, 15 (3).

D. L. James and D. K. Pai, 2004, BD-tree: output-sensitive collision detection for reduced deformable models. *ACM Trans. Graph.*, vol. 23.

J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan, 1998, Efficient collision detection using bounding volume hierarchies of k-DOPs. In *IEEE Transactions on Visualization and Computer Graphics*.

D. Knott and D. K. Pai, 2003, CInDeR: Collision and interference detection in real-time using graphics hardware. In *Graphics Interface*.

J. Krüger and R. Westermann, 2003, Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3).

T. Larsson and T. Akenine-Möller, 2005, A dynamic bounding volume hierarchy for generalized collision detection. In *Workshop On Virtual Reality Interaction and Physical Simulation*.

C. Lennerz, E. Schömer, and T. Warken, 1999, A framework for collision detection and response. In *11th European Simulation Symposium*.

M. C. Lin and D. Manocha, 2004, *Collision and proximity queries. In Handbook of Discrete and Computational Geometry, 2nd Ed., J. E. Goodman and J. O'Rourke.* Eds. CRC Press LLC.

J.-C. Lombardo, M.-P. Cani, and F. Neyret, 1999, Real-time collision detection for virtual surgery. In *Proc. of the Computer Animation*.

J. Mezger, S. Kimmerle, and O. Etzmuss, 2003, Hierarchical techniques in collision detection for cloth animation. In *Journal of WSCG*.

B. Mirtich and J. Canny, 1995, Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 symposium on Interactive 3D graphics*.

T. Möller, 1997, A fast triangle-triangle intersection test. *Journal of Graphics Tools*.

K. Myszkowski, O. G. Okunev, and T. L. Kunii., 1995, Fast collision detection between complex solids using rasterizing graphics hardware. In *The Visual Computer*.

I. J. Palmer and R. L. Grimsdale, 1995, Collision detection for animation using sphere-trees. In *Computer Graphics Forum 14*.

S. Redon, A. Kheddar, and S. Coquillart, 2002, Fast continuous collision detection between rigid bodies. In *Proceedings of Eurographics*.

J. Rossignac, A. Megahed, and B.-O. Schneider., 1992, Interactive inspection of solids: Cross-sections and interferences. In *Proceedings of SIGGRAPH '92)*.

L.-J. Shiue, I. Jones, and J. Peters, 2005, A realtime GPU subdivision kernel. *ACM Transactions on Graphics*, 24(3).

A. Smith, Y. Kitamura, H. Takemura, and F. Kishino, 1995, A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*.

A. Sud, N. Govindaraju, R. Gayle, I. Kabul, and D. Manocha, 2006, Fast proximity computation among deformable models using discrete voronoi diagrams. *ACM Transaction on Graphics*, 25(3).

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, 2005, Collision detection for deformable objects. *Computer Graphics Forum 24*.

G. van den Bergen, 1997, Efficient collision detection of complex deformable models using AABB trees. In *Journal of Graphics Tools*.

P. Volino and N. M. Thalmann, 2000, Accurate collision response on polygonal meshes. In *Proceedings of the Computer Animation*.

G. Ziegler, A. Tevs, C. Theobalt and H.-P. Seidel, 2006, On-the-fly Point Clouds through Histogram Pyramids, In Proceedings of VMV '06.