INTERACTIVE GPU-BASED COLLISION DETECTION

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, and with graphics capabilities becoming 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, access to displacement textures and geometry shaders (Balaz and Glassenberg 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.

The implications thereof with respect to collision detection are dramatic: As large parts of the geometry will permanently 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. The method is especially designed for the 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 makes the method amenable to the handling of 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 every primitive the information required to determine the set of primitives it might interfere with is computed and stored in a texture map. In particular due to this pass the precision of interference computations is not constrained by the resolution 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.

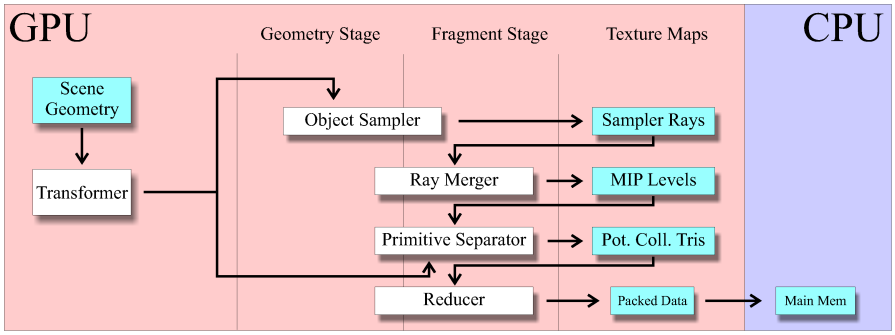

Figure 1: A diagrammatic overview of the proposed collision detection algorithm.

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. As a result it writes a fragment stream consisting of only the potentially colliding primitives 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 to 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.

As the data transfer from the GPU to the CPU is most likely to become the bottleneck in the entire collision detection system, 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.

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. Especially if a large number of objects have to be considered, collision detection algorithms usually proceed in a *broad phase* and a *narrow phase* (Cohen et al. 1995). 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).

In the context of deformable objects, the emphasis has been placed on the efficient update of hierarchical object representations (James and Pai 2004; Larsson 2005). A GPU collision detection algorithm for deforming NURBS objects has been developed (Greß et al. 2006). 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 off this theory, voxel (Baciu and Wong 2002; Heidelberger et al. 2004) and view-frustum (Lombardo et al. 1999) based methods have been proposed.

In comparison to previous approaches our method is closest in spirit to suggestions made by Knott and Pai (2003). As the method distinguishes between penetrating and penetrated objects, it can not 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, too (Govindaraju et al. 2003).

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, on the CPU the processing of a large number of empty cells can be avoided. The texture reduce operation on the GPU is accomplished in three passes:

- **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 left).
- **Shifting**: In each row non-empty texels are shifted to the right of the texture (see Figure 2 right).

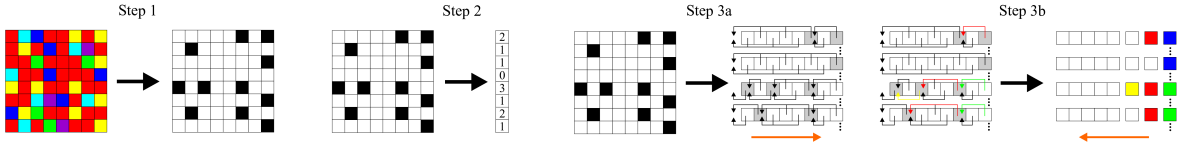

Figure 2: Counting and Shifting.

- **Moving**: Packed rows are moved into the reduced texture. The texture is finally downloaded to the CPU.

From the single-column texture being read in the first Step, the application program computes the total number of nonempty texels. 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 2 right). 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 2 right, 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. Via appropriately chosen vertex texture coordinates the fragments being generated for each line can fetch the corresponding values from the sparse texture and output these values to the render target. Due to performance issues, the application program creates a vertex array containing all the required information and renders this array using one single call.

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 testing for rays that have at least two consecutive hits with either a front or a back facing polygon.

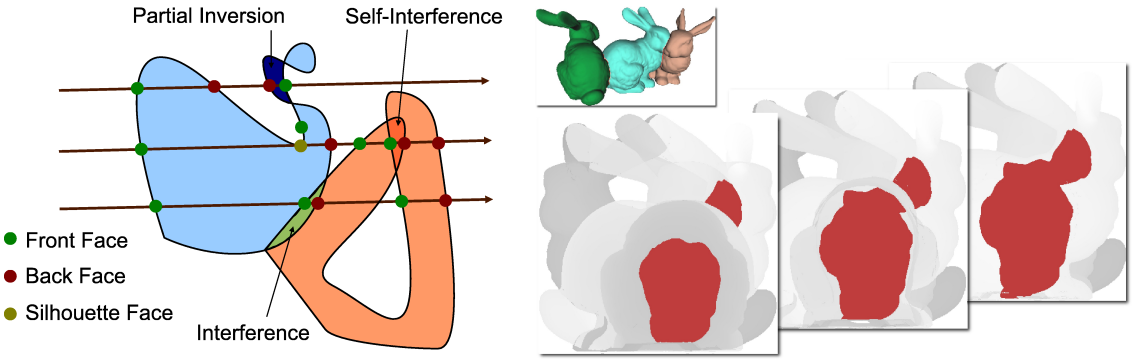

Figure 3: Illustration of interference, self-interference and partial inversion (left). By using depth-peeling andfragment operations to detect consecutive front or back faces, collision rays in each layer are determined (colored red, right image).

The underlying theoretical basis of this method is given by the generalization of Jordan's theorem to higher dimensions. If a closed polyhedron P separates space into an "inside" and an "outside" and has outfacing normals, it follows that any ray starting outside of P and intersecting P has alternating front and back facing intersection points. 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, then 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. This also holds for an arbitrary number of objects. Example cases of (self-) interference are illustrated in Figure 3 on the left.

To detect intersecting closed polyhedra it is thus sufficient to detect consecutive front or back facing polygons 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.

4.1 Implementation

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.

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 3 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 (2K × 2K), 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. This stage is described next.

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 now has to be carried over to the set of polygons. One possibility is to let the rasterizer generate 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. Therefore we propose a more efficient strategy. At the core of this strategy is the idea to generate for each polygon the information it requires to efficiently determine the set of primitives it collides with. This inter-object relation is established via the collision rays. As every polygon is hit by many collision rays in general, it requires several of them to detect all potentially colliding partners of a particular polygon. 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 4 on the left.

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 a 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 4 left). 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”. Now the idea is to find for every primitive the bundle that relates that primitive to all potentially colliding partners via the associated screenspace bounding boxes.

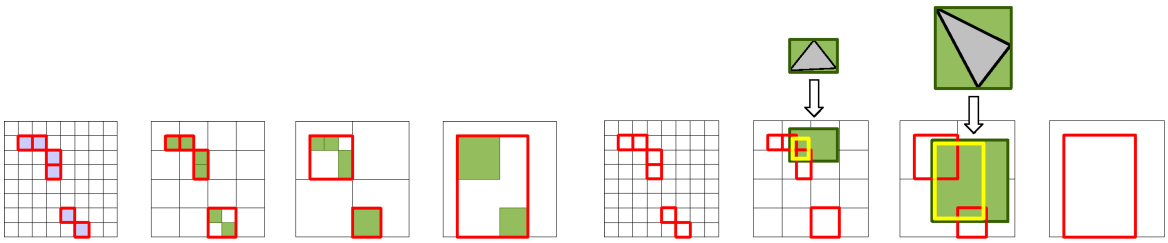

Figure 4: Mipmap construction (left) andprimitive separation (right).

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 4 right).

To efficiently find this level, we let the rasterizer generate 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. 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.

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 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 a single core Pentium 4 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 5 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 51 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 5 (middle image) 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. All collisions were detected in about 25 ms.

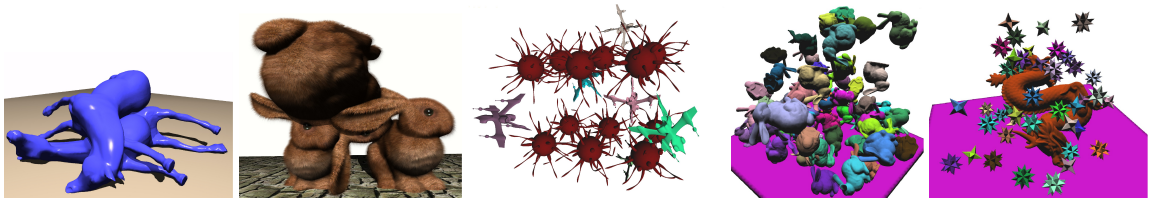

Figure 5: (Self-) Collisions between deformable (first two images), dynamic GPU and rigid objects (last two images).

Rigid body collisions: Our last example demonstrates the capability of our method to handle collisions between rigid bodies (see right of Figure 5). 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 120 ms. The efficiency of our approach is further demonstrated by the last example in Figure 5. 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 3 frames per second.

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

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.

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.

Table 1. Triangle counts and timing statistics of the test scenes

Scene	# tri's	rays # coll.	downl. # tri's	coll. tri's # pot.	tests # tri-tri	tests # pos.	GPU	Reduce	CPU	time Overall
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	42ms	51ms/20fps
art. creatures	320K	0.3K	1.0K	0.4K	1.5K	0.2K	17ms	6ms	2ms	25ms/40fps
rigid bunnies	500K	2.8K	12K	4.4K	7.7K	1.5K	56ms	18ms	44ms	118ms/8.5fps
dragon	800K	7.7K	54K	7.5K	6.6K	1.5K	53ms	32ms	248ms	333ms/3fps

9. CONCLUSION AND FUTURE WORK

In this paper we have presented a collision detection algorithm that is particular designed for recent and future graphics hardware. It exploits the intrinsic strength of GPU 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.

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/.

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

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

N. K. Govindaraju et al, 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 '03*.

A. Greß et al, 2006, GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum*.

B. Heidelberger et al, 2004, Detection of collisions and self-collisions using image-space techniques. In *Journal of WSCG 12(3)*.

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

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

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

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*.

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 et al, 1999, Real-time collision detection for virtual surgery. In *Proceedings of the Computer Animation*.

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

K. Myszkowski et al, 1995, Fast collision detection between complex solids using rasterizing graphics hardware. In *The Visual Computer*.

J. Rossignac et al, 1992, Interactive inspection of solids: Cross-sections and interferences. In *Proceedings of SIGGRAPH '92)*.

M. Teschner et al, 2005, Collision detection for deformable objects. *Computer Graphics Forum 24*.

G. Ziegler et al, 2006, On-the-fly Point Clouds through Histogram Pyramids, Proceedings of VMV'06, 137-144