EUROGRAPHICS 2006 / E. Gröller and L. Szirmay-Kalos
(Guest Editors)

Volume 25 (2006), Number 3

GPU-based Collision Detection for Deformable Parameterized Surfaces

Alexander Greß, Michael Guthe, and Reinhard Klein

Institute of Computer Science II, Universität Bonn, Germany

Abstract

Based on the potential of current programmable GPUs, recently several approaches were developed that use the GPU to calculate deformations of surfaces like the folding of cloth or to convert higher level geometry to renderable primitives like NURBS or subdivision surfaces. These algorithms are realized as a per-frame operation and take advantage of the parallel processing power of the GPU. Unfortunately, an efficient accurate collision detection, that is necessary for the simulation itself or for the interaction with and editing of the objects, can currently not be integrated seamlessly into these GPU-based approaches without switching back to the CPU.
In this paper we describe a novel GPU-based collision detection method for deformable parameterized surfaces that can easily be combined with the aforementioned approaches. Representing the individual parameterized surfaces by stenciled geometry images allows to generate GPU-optimized bounding volume hierarchies in real-time that serve as a basis for an optimized GPU-based hierarchical collision detection algorithm. As a test case we applied our algorithm to the collision detection of deformable trimmed NURBS models, which is an important problem in industry. For the trimming and tessellation of the NURBS on the GPU we used a recent approach [GBK05] and combined it with our collision detection algorithm. This way we are able to render and check collisions for deformable models consisting of several thousands of trimmed NURBS patches in real-time.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems; Splines; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality

1. Introduction

Efficient collision and self-collision detection of deformable 3D objects is required in numerous applications: in virtual prototyping it is used to determine clearances, in physically based simulations to ensure that the bodies do not penetrate each other and in computer games that colliding objects bounce and slide on contact instead of passing through each other. In these contexts a collision is defined as a configuration of two objects whose surfaces, that are allowed to deform and move over time, intersect. Since the deformation and movement is mostly simulated by discrete updates of the objects, standard collision detection algorithms are only interested in intersections at these discrete time stamps.

For the geometry of the objects different representations are used in these applications. While in most virtual reality applications meshes are still the predominant geometry representation for rendering and interaction, the use of higher order representations like NURBS or subdivision surfaces as rendering and interaction primitives has come into focus. The reason for this trend is that due to the rapidly growing parallel processing power that enables GPUs to perform tasks with inherent parallelism much faster than current CPUs, the conversion of these higher order geometry rep-

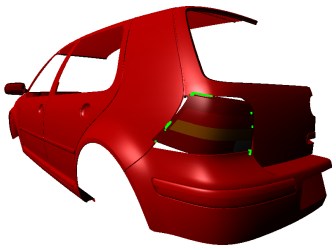

Figure 1: *Virtual assembly simulation (collisions highlighted in green).*

resentations into renderable primitives on the GPU became feasible in real-time. Especially in the automotive industry, using these surface representations in VR applications like the assembly simulation shown in Figure 1 instead of meshes can significantly shorten the preparation time. Similarly, physically based simulation tasks like the folding of cloth were realized directly on the GPU instead of the CPU. Therefore, there is a strong trend to move the collision detection to the GPU as well in order to integrate it seamlessly into such GPU-based approaches without transferring the generated geometry back to the CPU.

To detect collisions between rigid models, hierarchy-based methods have been shown to be the most efficient and accurate approaches. However, for deformable models, their disadvantage is that the hierarchy must be rebuilt or updated every frame. Since this hierarchy generation or update process involves a significant amount of sequential processing, existing techniques cannot be used on parallel architectures, as GPUs. An interim solution are GPU-based non-hierarchical screen-space approaches, which determine colliding or potentially colliding objects by rendering geometric primitives in screen-space, using depth or stencil buffer tests to check for collisions. Although these approaches are still quite efficient under certain conditions, their accuracy is limited by the current screen resolution and their performance is not always superior to CPU-based approaches.

To implement an efficient GPU-based hierarchical collision detection for deformable models, first of all a suitable representation for the bounding volume hierarchy that can be computed on the GPU must be found. It should not only allow to implement an efficient hierarchical collision test, but it should also make a GPU-based hierarchy construction as fast as possible, such that the benefit of hierarchical collision detection still applies to deformable models. To solve this problem, we suggest to use dynamic geometry images to represent the geometry, which allows to generate a quaternary bounding box hierarchy as fast as a conventional mipmap generation just by using different filters, even for deformable surfaces. All that is required is a parametrization of the objects similar to that used for texture mapping. Since virtually all parametrization techniques generate charts with complex boundaries, the boundaries are encoded in *stenciled* geometry images, which allow the use of arbitrary parameterizations (including multiple charts) as well as trimmed parametric surfaces. For the different types of parameterized surfaces such stenciled geometry images can be generated on the GPU either per frame by evaluating the surfaces at the regular samples of the image or during loading.

Since recent results show that the resulting quaternary hierarchy delivers better performance than a binary hierarchy and since this hierarchy provides good parallelism for breadth-first traversal, the use of geometry images is an excellent choice for a GPU implementation. In addition, for the collision test of two geometry images as well as for the self-collision test of a single geometry image a list of index pairs describing all pairs of geometry image texels representing intersecting surface elements can be reported. This list can be read back to the CPU very efficiently, if required, e.g. for allowing an arbitrary collision response. Based on the introduction of these dynamic stenciled geometry image hierarchies our main contributions are:

- A technique for real-time GPU-based generation of bounding volume hierarchies.
- A simultaneous hierarchy traversal method for the collision test based on an improved *non-uniform stream reduction* technique that is much more efficient than previous approaches. This is due to the usage of mipmaps and the automatic mipmap generation functionality as well as efficient vertex shader usage and combination of the stream reduction with the bounding volume overlap test.
- A GPU-based implementation of *hierarchical* self-collision detection (following the approach of Volino and Magnenat-Thalmann [VMT94]).
- As an application of these techniques, the first hierarchical collision detection system for trimmed NURBS surfaces on the GPU. This way the conversion of NURBS representations into an intermediate representation suitable for virtual prototyping becomes redundant.

2. Related work

Geometry images. Geometry images were introduced by Gu et al. [GGH02]. They represent an arbitrary surface as a simple 2D array / texture containing quantized points using a regular 2D grid sampling. This sampling is accomplished by parameterizing the entire surface onto a square. Alternatively, Sander et al. [SWG*03] use an atlas construction to map the surface piecewise onto charts of arbitrary shape. Praun and Hoppe [PH03] presented a technique to parameterize a genus-zero surface directly onto a spherical domain without cutting it into charts. Losasso et al. [LHSW03] presented an interesting application of geometry images, where they are used to represent a smooth genus-zero surface as a single bi-cubic B-spline surface patch, which can be rendered using subdivision performed in the fragment shader pipeline of the GPU.

Collision detection. Although there exist also a lot of non-hierarchical approaches to collision detection, for example methods based on distance fields or on spatial subdivision, bounding volume hierarchies have been shown to be among the most efficient data structures for collision detection [TKH*05]. Primarily, they have been applied to detect collisions of non-deformable models. They have also been shown to be quite well-suited for deformable models, despite the fact that in this case frequent updates of the bounding volume hierarchy are required. For deformable models, optimized updating and refitting techniques of the hierarchy have been devised [LAM01, MKE03, JP04]. Furthermore, hierarchies consisting of bounding volumes and

some additional data can be used to detect self-collisions efficiently [VMT94]. While many different bounding volumes have been used for rigid bodies, e.g. [PG95, GLM96, Zac98, AdG*02], it has been shown that axis-aligned bounding boxes (AABBs) are preferable over other bounding volumes in case of deformable models, since they can be updated more efficiently [vdB97]. Furthermore, 4-ary or 8-ary trees have been shown to perform better than binary trees for deformable models [LAM01, MKE03]. See [TKH*05] for a recent survey on various different methods for the collision detection of deformable objects. Concerning collision detection in the context of geometry images Beneš and Villanueva [BV05] used a binary bounding sphere hierarchy that exploits the special structure of the geometry image for their generation. But with respect to a GPU implementation quaternary trees are preferable since the GPU architecture is optimized to handle quad-tree hierarchies for 2D textures.

GPU-based collision detection. Several screen-space approaches for collision detection on the GPU have been proposed so far, e.g. [HTG03, KP03, GLM04, HTG04, WB04, GLM05, WB05]. They perform collision tests by rendering geometric primitives in screen-space. In many approaches this is done by the use of depth or stencil buffer tests, as for example in the approach by Knott and Pai [KP03], where edges of one object and polygons of the other object are rendered (based on the observation that an intersection can occur if and only if an edge of one object intersects a polygon of the other object). Many screen-space approaches have restrictions regarding the topology of an object: most of them require closed objects and some of them only work for convex objects. Another issue of screen-space approaches is the avoidance of geometric errors which can easily be caused by the discretization induced by rendering primitives in screen-space, for example if the size of a primitive is below the chosen screen resolution. Even if such errors are prevented [GLM04], the screen resolution has still significant impact on the performance of the algorithm. Besides, some screen-space approaches only report *potentially* colliding sets of objects [GLM05], which have then to be checked for exact collision of primitives on the CPU. In a recent work by Govindaraju et al. [GKJ*05], such a GPU-based collision culling technique has been integrated into an otherwise CPU-based collision detection approach for deformable models based on the partitioning of the given models into independent sets.

[GZ04] as well as [Hor05] presented GPU-based hierarchical collision detection approaches, but their implementations were not able to clearly outperform an optimized hierarchical CPU-based implementation. Both approaches try to expose the parallel processing capabilities of recent GPUs by a breadth-first traversal of the bounding volume hierarchies on the GPU using a technique, which Horn calls *non-uniform stream reduction* (cf. Section 5.3). Nevertheless, in both GPU-based approaches the hierarchy traversal has an additional overhead of $\log_2 n$ compared to a corresponding hierarchy traversal on the CPU. The implementation of [GZ04] is further restricted to the use of 1D textures (or a single line of a 2D texture per mesh), thus preventing efficient use for complex models because of texture size restrictions on current GPUs.

3. Overall Algorithm

Let us define in more detail what we mean by detecting collisions of parameterized deformable models (whose geometry is given in the form of geometry images). Depending on the application we differentiate between the following cases:

- *Approximate collision detection:* In this case we assume that the resolution of the geometry images is chosen in such a way that the distance of the positions stored at neighboring texels is always below a given threshold. We report a collision if any pair of the AABBs defined by single surface elements from the geometry images overlap. These AABBs build the leaf level of the AABB tree (see Section 4). Under the aforementioned assumption every reported collision corresponds to a pair of surface elements that are not farther from each other than the given threshold. Note that geometry images for parameterized models can be generated at any resolution because of the given parametrization.
- *Exact collision detection:* If we interpret the geometry image as an exact geometry representation where each surface element corresponds to a quad (or two triangles) defined by the positions stored at four neighboring texels, we could perform exact triangle intersection tests at the lowest hierarchy level. However, this assumption does not make sense for geometry images generated by tessellating higher order primitives. Therefore, the description of a GPU-based triangle intersection test (see e.g. [GZ04]) is omitted here.

When two deformable models have to be checked for collision against each other, the basic steps of the hierarchical collision detection, that are performed every frame after the generation/update of the geometry images are as follows:

1. *AABB tree generation:* The first step of the collision detection for deformable surfaces is to build the AABB trees for both geometry images. While for static geometry images it is sufficient to do this once, we have to repeat this step every frame for dynamic geometry images. Section 4 describes this step in detail.
2. *Hierarchy traversal and collision test:* Then both AABB trees are traversed simultaneously visiting only those node pairs whose parent nodes correspond to an overlapping AABB pair. A depth-first traversal order as used in many CPU-based approaches does not allow efficient parallel execution and is therefore unsuitable for a GPU implementation. Instead, a breadth-first traversal scheme is used since this way the AABBs of all node pairs on the same hierarchy level may be tested simultaneously for overlap. Section 5 describes the hierarchy traversal and the collision test in detail.

If a set of more than two objects has to be checked for pairwise collision, we first determine the pairs with overlapping (top level) bounding boxes on the CPU, using conventional acceleration techniques like a grid or hash table. The AABB tree generation is only performed for those objects whose bounding boxes overlap with any other object, and the collision test of course only for object pairs with overlapping bounding boxes. Furthermore, we can restrict the geometry image generation and thereby also the hierarchy generation and traversal to the overlapping area of the top level bounding boxes, if the geometry images are generated only for the collision detection process.

4. AABB tree generation

We represent an AABB by two corners, the one with minimum coordinates (*min point*) and the one with maximum coordinates (*max point*). Given a geometry image of size $r_u \times r_v$, we store the leaf level of the AABB tree in a min point and a max point geometry image, each of size $(r_u - 1) \times (r_v - 1)$, containing the min and max points of the AABBs, respectively. Thereby, a texel (i, j) stores the AABB around the quad with the corners defined by the texels $(i, j), (i+1, j), (i, j+1), (i+1, j+1)$ of the given geometry image (for $i \in \{0, \ldots, r_u - 2\}, j \in \{0, \ldots, r_v - 2\}$), as shown in Figure 2. Thus generating the leaf level of the hierarchy corresponds to calculating the component-wise minimum and maximum of four vectors for each texel, which can be done in a simple one-pass fragment shader.

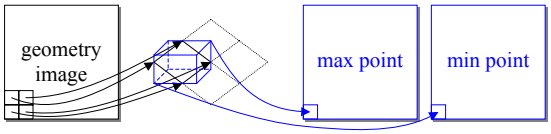

Figure 2: *Generation of min/max point textures.*

After setting up the AABBs for the leaf level, the rest of the hierarchy can be generated in a bottom-up manner. This way we can store the whole hierarchy in the mipmap levels of the two geometry images that already contain the leaf levels. Starting from the leaf level, we build the two mipmaps using minimum and maximum filters, respectively. These two filters can be implemented by a simple fragment shader, which requires one pass for each of the $\log_2 n$ mipmap levels. Since the number of written fragments during this $\log_2 n$ passes corresponds to the number of texels in the two mipmaps without the base level (i.e. $\frac{n-1}{3}$ texels for each of the two mipmaps, with $n = (r_u - 1) \cdot (r_v - 1)$), the hierarchy generation is a cheap linear time operation, comparable e.g. to the time required to render the surface given by the geometry image in corresponding resolution, thus, this operation can be easily performed per frame. In the following, we name the two mipmapped geometry images that contain the AABB tree *hierarchy images*.

5. Hierarchy traversal and collision test

In this section, we describe how the simultaneous traversal of two AABB trees and the AABB overlap tests at each level are implemented.

Assuming that the resolution of the geometry images and thus also of the hierarchy images was based on a common threshold for both models, the AABBs of equal hierarchy levels in both AABB trees (labeled bottom-up, starting from their leaf levels) are approximately of the same size. Therefore, an efficient traversal strategy is to descend both hierarchies simultaneously in each step of the breadth-first traversal, such that always nodes of equivalent hierarchy levels are tested for overlap. Since the root AABB overlap test was already done on the CPU and since hierarchy levels with less than 256 node pairs do not provide enough parallelism to be productive on the GPU, we skip the first few levels (assuming that all node pairs overlap), until at least 256 overlap tests are performed.

5.1. AABB overlap test

For a dynamic geometry image that is generated every frame, we can assume that the positions in the geometry image and thus also the AABBs in the hierarchy images are given in world space. Let $boxMin_{0,1}$ and $boxMax_{0,1}$ be the min and max points, respectively, of two arbitrary AABBs from the hierarchy. Then these two AABBs overlap if and only if the vector given by

$$\min(boxMax_0, boxMax_1) - \max(boxMin_0, boxMin_1) \quad (1)$$

with component-wise min and max operators has no negative component.

5.2. Hierarchy traversal

The basic idea of the hierarchical approach is to process the AABB trees via a breadth-first traversal in $\log_2 n$ steps, one for each level of the hierarchy. Contrary to the brute-force approach of testing all n_0 boxes of one object against all n_1 boxes of the other, now only those pairs of AABBs have to be checked for whose parent AABBs an overlap has been detected in the previous step of the traversal. Therefore, some references are required as additional input, which point to the overlapping pairs of AABBs from the previous level and thus indicate which pairs of AABBs from the current level have to be checked. In the following the term *reference* denotes a pair of 2D texture coordinates, t_0 and t_1, such that t_0 points to a texel in the first hierarchy image and t_1 to a texel in the second hierarchy image. Assuming that the previous step resulted in a list of such references to all overlapping AABB pairs, this list can be used as input for the current step of the traversal in such a way that for each entry in that list exactly 16 overlap tests are performed as shown in Figure 3: the four children of the AABB at texel t_0 are checked against each of the four children of the AABB at texel t_1.

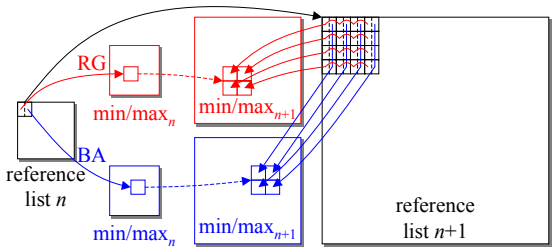

Figure 3: *Propagation of reference list from level n to n+1.*

Let us assume that in each step of the traversal, the shader outputs for every checked AABB pair, in case there is an overlap, the reference to this AABB pair, or otherwise a null reference (i.e. an invalid texture coordinate pair), marking the corresponding fragment irrelevant for the succeeding traversal step. It is obvious that we have to remove these null references from the *output stream* (i.e. from the render target texture) before using them as input for the succeeding traversal step, as otherwise the list would grow by factor 16 with each traversal step, finally having to process a fragment for each pair of leaf AABBs at the last traversal step, the same way as in the non-hierarchical brute-force approach. Note, that the traversal is stopped when no overlapping pairs remain, which can easily be tested using an occlusion query.

5.3. Non-uniform stream reduction

The removal of null references corresponds to the *non-uniform stream reduction* operation described by [Hor05]. To convert a non-compacted data stream (in this case the list of references where some of them are null-references) to the corresponding compacted stream, in a first step we determine for each non-null entry of the non-compacted stream at which location it will be stored in the compacted stream. The 0-based index of this location is exactly the number of non-null elements to the left of this entry in the non-compacted stream (assuming a 1D representation of the data stream here). Therefore, by counting the non-null elements to the left of each entry, we can construct a list of indices of the same size as the non-compacted stream, such that we can use this list in a succeeding step to actually transform the stream to its compacted equivalent. To overcome the $\log_2 n$ runtime overhead of previous approaches [GZ04, Hor05], we use a two-step approach with a total running time of $\frac{8}{3}n$ instead of $n \log_2 n$. Our approach is tailored to 2D streams (as the data stream that we have to compact is a 2D texture), while the previous approaches assumed 1D streams.

The first of these two steps is similar to the binary hierarchy construction of [GZ04]. However, since we are dealing with a 2D stream, we construct a quaternary hierarchy and store it in a mipmapped 2D texture. For simplicity, let us assume that the base level of this texture has exactly the same size as the non-compacted stream. In a simple shader pass, we set each entry of the base level to 0 if the corresponding entry in the non-compacted stream is a null reference and to 1 otherwise (Figure 4 left). The other levels of the hierarchy are constructed via mipmap generation. For efficiency reasons, this time we do not implement a custom mipmap generation via a multi-pass shader, but use the standard hardware mipmap generation functionality to construct the mipmap with a single API call and of course linear running time (Figure 4 right). This results in a quad-tree hierarchy where each node holds the percentage of non-null elements contained in a certain rectangular region of the non-compacted 2D stream, which is equal to the number of these elements apart from a scaling factor that is constant for each level. (The scaling is not shown in the figure.)

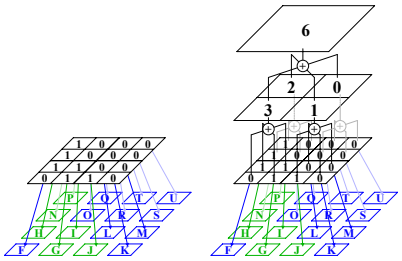

Figure 4: *Hierarchical non-null element counting.*

In a second step, based on this hierarchy we construct the index list in form of an additional mipmapped 2D texture, whose base level has the same size as the non-compacted stream and stores for each entry of the non-compacted stream an index of the location where it will be stored in the compacted stream. While in the 1D case the index could be computed by counting the non-null elements to the *left* of the respective entry, in this case we define the index to be the number of non-null elements that would be visited *before* the respective entry during a depth-first traversal of the mipmap quad-tree hierarchy. According to this definition we can calculate indices on all levels of the mipmap hierarchy, although only the indices at the base level are required later to determine the target locations of entries in the compacted stream. Of course, we do not perform a depth-first traversal to determine the indices, as the same values can also be calculated using a breadth-first traversal, i.e. via a top-down construction of the mipmap levels in $\log_2 n$ passes, namely by adding the index of the parent node (as that node would be visited before the current node in a depth-first traversal) to the number of non-null elements visited after the parent and before the current node. Assuming that the current node is the nth child of its parent according to an arbitrary but consistent ordering ($n = 1, \ldots, 4$), the latter number can be calculated by summing up $n-1$ non-null element counts from the previously computed mipmap of non-null element counts/percentages (see Figure 5).

Hence, in total we need one shader pass plus a hardware mipmap generation, followed by the construction of an additional mipmap in a top-down manner (in $\log_2 n$ shader

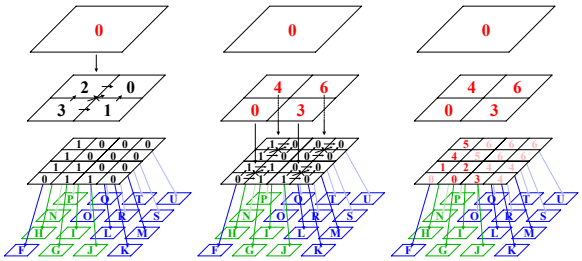

Figure 5: *Hierarchical calculation of the number of previously visited non-null leaves for each node.*

passes), and thus a total of $2 \cdot (n + \frac{n-1}{3}) \approx \frac{8}{3}n$ processed fragments to construct the index list. Finally, the compacted stream can be constructed by taking each entry from the non-compacted stream and writing it to the location determined by the corresponding index from this list. However, writing to arbitrary locations (also called *scattering* in the context of GPU/stream programming) is not possible from the fragment shader. Instead, we could adopt the gather search technique used in [Hor05], but this again would result in a running time of $n \log_2 n$. Another alternative would be to render point primitives consisting of single fragments, using the vertex shader to direct these points to the desired locations. However, rendering such single fragments is quite slow with current graphics hardware. Only when using larger point primitives, at least of size 4x4, it is possible to obtain the full fragment fill-rate on current GPUs.

The solution to this issue is to combine this scattering pass with the succeeding AABB overlap test. As described in Section 5.2, for each entry in the compacted list 16 overlap tests are performed. This fact allows us to efficiently use the vertex as well as the fragment shader to perform the scattering and the overlap tests simultaneously: For each point primitive (a 4x4 point sprite), the vertex shader reads an entry from the non-compacted stream, discards it in case of a null element, or otherwise determines the position of the point primitive (by reading the corresponding entry from the index list) and passes the stream element to the fragment shader; while the fragment shader performs the overlap tests for the 16 child AABB pairs of this stream element.

6. Extensions

In this section, we devise two extensions of our approach. First, we extend it to *stenciled* geometry images, a geometry representation which is easier to use and also more efficient for certain types of parameterized models, for example trimmed NURBS surfaces. Then we show how to detect self-collisions efficiently.

6.1. Stenciled geometry images

A stenciled geometry image denotes an RGBA texture, that stores a position in its RGB channels (just like an ordinary geometry image) and a stencil value in its alpha channel. We assume that the stencil value is either 0 or 1, where 1 denotes a texel that corresponds to an existing surface element and 0 denotes a texel that does not contribute to the actual surface.

6.1.1. Modification of the hierarchy generation

The stencil values from the geometry image have to be propagated throughout the hierarchy, such that we can skip AABBs with stencil value 0 during the overlap tests. Since an AABB contains existing surface elements only if *any* of its child AABBs has the stencil value 1, we can calculate the stencil value of an AABB in the hierarchy as the maximum of the stencil values from its corresponding child AABBs. At the leaf level, the stencil value of an AABB is set to 1 if *all* four surface vertices in the geometry image have the stencil value 1.

If we use the alpha component of the max point hierarchy image to store the stencil values of the AABBs, we could use the min and max filters to construct the mipmaps of the hierarchy images in the exact same manner as before. However, this way some of the AABBs – especially those containing only one or two children with stencil values of 1 – might be larger than necessary because the stenciling is not taken into account when calculating the AABB. Correctly fitting AABBs can be achieved by a slight modification of the shaders that we use to evaluate the min and max filters: they now calculate the minimum and maximum of only those min/max points that have a corresponding stencil value of 1.

6.1.2. Modification of the AABB overlap test

Let $boxMin_{0,1}$ be 4D vectors with the first three components set to the min points of the two AABBs to be checked (coming from the min point hierarchy images), and the fourth component set to $\frac{1}{2}$. Let $boxMax_{0,1}$ be 4D vectors coming from the max point hierarchy images, such that the first three components contain the max point of the two AABBs to be checked and the last component contains the stencil value. Then the same test as described in Section 5.1 can be applied: there is an overlap only if the 4D vector defined by eq. 1 has no negative component.

6.2. Self-collision

The hierarchical collision detection described in Sections 3 – 5 can also be used to detect self-collisions of a given deformable model. Instead of testing the AABBs of two different hierarchies against each other, we simply test one AABB tree against itself. As for the collision detection of two different models, the hierarchical self-collision results in a list of references to every intersecting pair of surface elements. Since in this case, the two surface elements of such a pair are part of the same surface, it makes no sense to return both, (t_0, t_1) and (t_1, t_0), for two intersecting surface elements with texture coordinates t_0 and t_1. Therefore

we can skip the AABB pairs (t_0,t_1) with $t_0 > t_1$, according to an arbitrary but consistent ordering, during the hierarchy traversal. Furthermore, we have to skip the pairs (t_0,t_1) with $t_0 = t_1$ at the leaf level since they obviously do not represent an intersection. However, at all other levels we cannot skip these pairs, as we otherwise would not find any intersection since all intersecting pairs of surface elements are located in common AABBs at some upper levels of the hierarchy. This means that using this method without modification always all levels of the hierarchy have to be traversed to detect self-collisions. In addition, the AABBs of neighboring regions are very likely to overlap, resulting in many overlap tests even for non self-intersecting surfaces.

Fortunately, there is a possibility to reduce the number of required AABB overlap tests for self-collision detection and to allow the hierarchy traversal to terminate earlier in case of non self-intersection surfaces. This is accomplished by performing an additional test during the hierarchy traversal for certain AABB pairs (t_0,t_1), following the approach of Volino and Magnenat-Thalmann [VMT94]. The basic idea is: If the section of the surface contained in the single AABB $t_0 = t_1$ or in the two adjacent AABBs t_0,t_1 cannot possibly contain self-intersection, there is no need to check the children of t_0,t_1. We will integrate such a test in the same shader pass as the AABB overlap test.

As observed in [VMT94], a surface can only self-intersect if either the surface has a sufficiently high 'curvature', i.e. it is bend in such a way that one part of the surface hits another one, or if its boundary has a sufficiently high 'concavity' such that its 2D projection intersects itself. The latter is a rather pathological case, such that, although not necessarily exact, for many applications it is sufficient to test the first condition only. Otherwise the latter condition, for which only the surface boundary is relevant, could be checked on the CPU prior to the hierarchy traversal (c.f. [VMT94]). Now we can identify a region of the surface to have no potential self-collision, if there exists a vector that has a positive dot product with the normal of every surface element in this region. This can be checked efficiently by using a modified version of the vector direction sampling method described in [VMT94] incorporated directly in the shaders used for the hierarchy generation and AABB overlap test.

6.2.1. Modification of the hierarchy generation

For the leaf level of the hierarchy we check the dot product of eight predefined normal vectors representing a well-distributed sampling of the unit sphere with the local surface normal calculated from the geometry image. The result of these tests can be propagated to the upper hierarchy levels such that we know for every AABB of the hierarchy whether it has a positive dot product with one of the eight normals. In principle, this could be done by using a bit-field consisting of eight bits [VMT94], however, since GPUs currently do not support bit operations, we use four signed bytes instead to store the results of the normal test. By choosing the eight normals in such a way that every two of them are opposite to each other, we effectively need to calculate only four dot products at the leaf level and store the *sign* of the result (-1, 0, or 1) in the corresponding byte. These sign values are transferred to the higher levels of the hierarchy as long as they are identical for all four children of a particular node and set to 0 otherwise.

6.2.2. Modification of the AABB overlap test

Together with the AABB overlap test described in section 5.1 we perform an additional test for those pairs of AABBs (t_0,t_1) that are adjacent to each other or where $t_0 = t_1$, based on the four sign values stored in the hierarchy for each of the two AABBs. If any of them is not 0 and identical for both AABBs, the corresponding surface region can have no self-collision and need not be further traversed (i.e. a null reference is returned by the shader). Contrary to [VMT94], in our setting it is trivial to determine whether two AABBs are adjacent: we only need to compare their texture coordinates t_0 and t_1.

7. Application to NURBS models

Trimmed NURBS surfaces are the standard representation used by the majority of CAD systems. The traditional approach for handling of such models is the conversion into triangular meshes as a preprocessing step. Since this is not only time consuming, but also often needs some degree of user input, direct handling of trimmed NURBS models is of great interest to industry.

7.1. Related work

There are some previous approaches for collision detection between trimmed NURBS surfaces [HBZ90, HDLM96, LCLL02], but these were tailored to the CPU and cannot efficiently be implemented on the GPU to process GPU-generated deformable models. Due to the high computational cost, they are all limited to models of low complexity.

Recently, a method for direct rendering of trimmed NURBS surfaces on the GPU was proposed [GBK05]. In this approach, the NURBS surface is first converted into a set of bi-cubic Bézier patches on the CPU. These patches are then approximated with quad meshes on the GPU. The trimming is achieved by generating a binary *trim texture* on the GPU, which is then used in the fragment shader to remove the trimmed-away portions when rendering the surface. The resolutions required for the quad mesh grid as well as for the trim texture are determined dynamically on the CPU such that even highly deformable NURBS surfaces are approximated appropriately at any time. Although some parts of the algorithm are performed on the CPU, the amount of data to be transfered per frame from CPU to GPU is only marginal.

7.2. GPU-based Collision detection for NURBS models

To apply our GPU-based collision detection algorithm to NURBS models (without any prior conversion of these models to mesh approximations on the CPU), we extend the approach by Guthe et al. [GBK05] such that it can be used to generate the geometry images of all possibly colliding NURBS patches, which are then used by our collision detection algorithm to detect the patch-patch collisions. In each frame, at first all possibly colliding pairs of NURBS patches are determined using pairwise bounding box tests. Afterwards, geometry images have to be generated for all NURBS patches belonging to at least one possibly colliding pair. This generation can be restricted to a clipping volume given by the overlapping region of the bounding boxes. The required resolutions for the geometry images are determined per frame on the CPU as described in Section 7.3.

As in [GBK05], each NURBS patch is first converted into a set of bi-cubic Bézier patches, and the surface approximation of these patches by quad meshes is calculated on the GPU. But instead of rendering these quad mesh approximations directly to the screen, we output their geometry into the corresponding geometry image. This is achieved by rendering the generated mesh into the geometry image using the Bézier parameter values as positions of the grid vertices. To handle trimmed NURBS models, we only need, in addition to that, to include the binary trimming information as stencil values into the stenciled geometry image. This is equivalent to the rendering of the trim texture described in [GBK05], but with the output directed to the alpha channel of the geometry image and writing to the RGB channels disabled.

7.3. Determining the geometry image resolution

The resolution $r_u \times r_v$ required for a geometry image depends on the desired accuracy ε in two ways: first, the resolution $t_u \times t_v$ of the trimming needs to be sufficient to guarantee that the distance between neighboring texels is not higher than ε, and second, the quad mesh approximating the surface must have at least a resolution of $s_u \times s_v$ to guarantee a geometric error of at most ε. If these are known, $r_u \times r_v$ is simply their component-wise maximum. The resolution $t_u \times t_v$ required for the trimming is the maximum of the resolutions $t_u^{ij} \times t_v^{ij}$ required for each Bézier patch B_{ij} inside the clipping volume. In u-direction these are defined by

$$t_u^{ij} = \left\lceil \frac{(u_{i+1} - u_i)}{\varepsilon} \sup_{p \in [0,1]^2} \left\| \frac{\partial B_{ij}(p)}{\partial u} \right\| \right\rceil,$$

where $[u_i, u_{i+1}] \times [v_j, v_{j+1}]$ is the domain interval of the bicubic patch $B_{ij}(p)$. The required resolution for the surface approximation is in u-direction

$$s_u = \max_{i=0}^{m} \max_{j=0}^{n} \left\lceil (u_{i+1} - u_i) \sqrt{\frac{M_u^{ij} + M_{uv}^{ij}}{8\varepsilon}} \right\rceil$$

with

$$M_u^{ij} = \sup_{p \in [0,1]^2} \left\| \frac{\partial^2 B_{ij}(p)}{\partial u^2} \right\|, \quad M_{uv}^{ij} = \sup_{p \in [0,1]^2} \left\| \frac{\partial^2 B_{ij}(p)}{\partial u \partial v} \right\|,$$

and analogously for the the v-direction.

8. Results

We tested our implementation on an AMD Athlon-64 X2 4200+, PCIe 16x PC with a GeForce 7800 GTX GPU. To avoid potential precision loss in our GPU-based collision detection compared to a CPU-based approach, all tests have been made using full precision 32 bit floating-point computations on the GPU and 32 bit float textures.

8.1. Examples

There are many fields of application for our approach. One example is the virtual assembly simulation shown in Figure 1. Although the rear light of the car model consists of 104 NURBS patches and the rest of the shown car body has 1620 NURBS patches, the determination of all contact points between the rear light and the car body up to a user-specified precision achieves interactive frame rates using our method – including the rendering of the model and the visualization of the contact points. The contact point visualization is implemented by a simple shader that directly reads the contact pairs from the compacted list determined by our approach. Thus no read back to the CPU is required. As the geometry used as input for the collision detection is generated from the NURBS patches of the original CAD model on the GPU in real-time, the user can change the precision threshold at run-time, such that he can interactively control the trade-off between frame rate and accuracy of the collision detection. This is not possible in previous virtual assembly applications, where the NURBS patches are converted to polygonal meshes on the CPU in a time-consuming preprocessing step.

Furthermore, the given input models are allowed to deform over time. This is especially important for physically based simulations. Figure 6a shows an example of a simple dynamics simulation, where several rigid balls are colliding with a deforming flag, which is a trimmed NURBS patch, where a spline shape was cut out, letting some of the balls pass. The whole simulation (including the collision response calculated on the CPU) runs at over 25 fps.

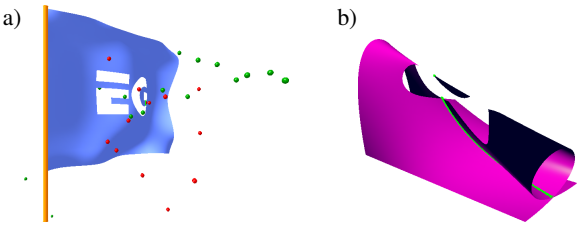

Figure 6: *Collision and self-collision detection for deformable trimmed NURBS surfaces.*

In addition to NURBS models, we also tested our approach on other kinds of geometry. For arbitrary rigid

meshes, geometry images can be generated in a preprocessing step on the CPU using the established techniques [GGH02]. Figure 7a shows an example for the determination and visualization of the contact points between two such geometry images. Once in geometry image representation, the geometry can be deformed on the GPU via the fragment shader using the same techniques that are usually applied in the vertex shader when rendering deformable meshes. For example, the vertex positions can be displaced dynamically using non-linear deformations as in Figure 7b. The whole application, including model deformation, collision detection, and contact point visualization runs at over 60 fps.

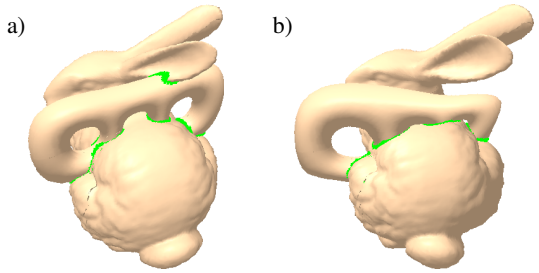

Figure 7: *Collision detection between static (left) and dynamic (right) geometry images.*

Finally, our approach allows the efficient detection of self-collisions of deformable models, important for example for applications like cloth simulation. We demonstrate this property on another deformable trimmed NURBS surface, as shown in Figure 6b. Here, the self-contact points determined by our approach are visualized.

8.2. Comparison

In the following, we compare the performance of our method to recent approaches. Since there exist no previous GPU-based algorithms that are able to determine all contacts of deformable parameterized models up to an arbitrarily high object-space precision, we can only compare to CPU-based collision detection.

Since most of the hierarchical collision detection packages for the CPU that are freely available do not support deformable models, we choose as a first test the scenario from Figure 7a, where two static geometry image models are moving through each other with constant speed. In this case, there is no need to rebuild the hierarchy every frame, since the geometry is constant over time. Of course a CPU-based approach that builds the hierarchy in a possibly time-consuming preprocessing step could be tuned to build a highly-optimized hierarchy for the given geometry images and has thus the potential to outperform our GPU-based approach for such static geometry. Astonishingly, our tests show that using our fast GPU-based traversal technique the simple AABB quad-tree hierarchy of our approach performs still very well compared to CPU-based approaches and in

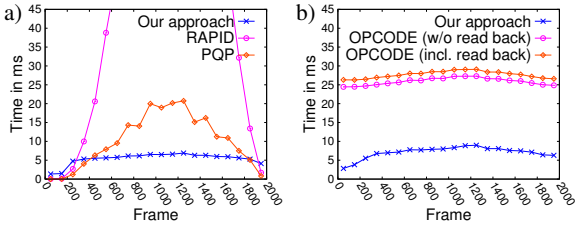

Figure 8: *Collision detection times for the two static (left) and the two dynamic (right) models from Figure 7.*

most cases even significantly better than the efficient collision detection packages *RAPID 2.01* and *PQP 1.3* (based on [GLM96]), when they are applied to the uniform quad meshes given by the geometry images, see Figure 8a.

When considering deformable models and dynamically generated geometry images, the advantage of our GPU-based approach gets even more evident. For comparison, we use a highly optimized CPU-based collision detection library with support for deformable models: *OPCODE 1.3* (`http://www.codercorner.com/Opcode.htm`). For maximum performance, we configure OPCODE to use AABBs as bounding volumes and disable the triangle intersection test at the leaf level of the hierarchy, just like in our approach. However, contrary to our approach, OPCODE handles only deformations caused by the change of vertex positions but does not allow a change of the mesh topology, similar to other previous hierarchical approaches for deformable geometry. Therefore we restrict the comparison to deformable models with constant topology, i.e. we have to keep the resolution of the stenciled geometry images constant as well as the stencil information contained therein. This allows OPCODE to construct the AABB tree for a model in a preprocessing step based on its topology and its initial vertex positions. At run-time, the AABB tree is refitted to the updated vertex positions of the model. In order to apply this CPU-based approach to GPU-generated deformable geometry, the generated geometry image must be read back to the CPU in each frame. Note that this is only practicable on a PCIe 16x system, where contrary to an AGP system the overhead of such a read back is significantly lower than the time required by OPCODE to refit the AABB tree. Figure 8b shows the performance of this CPU-based technique (including and excluding the read back overhead) in comparison with our GPU-based approach (including per-frame hierarchy image generation), using the GPU-generated deforming models from Figure 7b.

For more complex models such as the NURBS model shown in Figure 1 the CPU-based approach of building and refitting bounding volume hierarchies for the GPU-generated geometry images becomes totally impracticable, because of the high memory footprint required to store the hierarchies. In contrast, our GPU-based approach does not require any permanent memory for the hierarchy of deformable models since it is rebuilt completely in each frame.

9. Conclusion

We have presented a GPU-based hierarchical collision detection approach for deformable parameterized surfaces, consisting of a real-time generation of bounding volume hierarchies on the GPU, an efficient simultaneous hierarchy traversal method for the collision test, and a GPU-based hierarchical self-collision detection method. Based on these novel techniques, we developed the first GPU-based algorithm that allows to render and check collisions for complex deformable trimmed NURBS models in real-time. As shown by the evaluations, our approach permits accurate high-performance collision and self-collision detection for NURBS models or other types of rigid or deformable parameterized surfaces.

Acknowledgment. This work was partially funded by the EU under the Sixth Framework Programme (INCO-CT2005-013408).

References

[AdG*02] AGARWAL P., DE BERG M., GUDMUNDSSON J., HAMMAR M., HAVERKORT H.: Box-trees and r-trees with near-optimal query time. *Discr. Comp. Geom. 28*, 3 (2002), 291–312.

[BV05] BENEŠ B., VILLANUEVA N. G.: GI-COLLIDE: collision detection with geometry images. In *SCCG '05: Proc. of the Spring Conference on Computer Graphics* (2005), pp. 95–102.

[GBK05] GUTHE M., BALÁZS Á., KLEIN R.: GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Trans. Graph. 24*, 3 (2005), 1016–1023.

[GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. *ACM Trans. Graph. 21*, 3 (2002), 355–361.

[GKJ*05] GOVINDARAJU N., KNOTT D., JAIN N., KABUL I., TAMSTORF R., GAYLE R., LIN M., MANOCHA D.: Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. Graph. 24*, 3 (2005), 991–999.

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. In *Proc. of ACM SIGGRAPH '96* (1996), pp. 171–180.

[GLM04] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Fast and reliable collision culling using graphics hardware. In *VRST '04: Proc. of the ACM Symposium on Virtual Reality Software and Technology* (2004), pp. 2–9.

[GLM05] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware. In *VR '05: Proc. of the IEEE Conference on Virtual Reality* (2005), pp. 59–66.

[GZ04] GRESS A., ZACHMANN G.: Object-space interference detection on programmable graphics hardware. In *Geometric Modeling and Computing* (2004), Proc. of SIAM Conference on Geometric Design and Computing: Seattle 2003, Nashboro Press, pp. 311–328.

[HBZ90] HERZEN B., BARR A., ZATZ H.: Geometric collisions for time-dependent parametric surfaces. In *Proc. of ACM SIGGRAPH '90* (1990), pp. 39–48.

[HDLM96] HUGHES M., DIMATTIA C., LIN M., MANOCHA D.: Efficient and accurate interference detection for polynomial deformation. In *Proc. of Computer Animation Conference* (1996), p. 155.

[Hor05] HORN D.: Stream reduction operations for GPGPU applications. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005), Pharr M., (Ed.), Addison-Wesley, pp. 537–589.

[HTG03] HEIDELBERGER B., TESCHNER M., GROSS M. H.: Real-time volumetric intersections of deforming objects. In *VMV '03: Proc. of the Conference on Vision, Modeling and Visualization 2003* (2003), pp. 461–468.

[HTG04] HEIDELBERGER B., TESCHNER M., GROSS M. H.: Detection of collisions and self-collisions using image-space techniques. In *Journal of WSCG* (2004), vol. 12, pp. 145–152.

[JP04] JAMES D. L., PAI D. K.: BD-tree: output-sensitive collision detection for reduced deformable models. *ACM Trans. Graph. 23*, 3 (2004), 393–398.

[KP03] KNOTT D., PAI D. K.: CInDeR: Collision and interference detection in real-time using graphics hardware. In *Proc. of Graphics Interface* (2003), pp. 73–80.

[LAM01] LARSSON T., AKENINE-MÖLLER T.: Collision detection for continuously deforming bodies. In *Proc. of Eurographics* (2001), short presentation, pp. 325–333.

[LCLL02] LAU R. W., CHAN O., LUK M., LI F. W.: LARGE: a collision detection framework for deformable objects. In *VRST '02: Proc. of the ACM Symposium on Virtual Reality Software and Technology* (2002), pp. 113–120.

[LHSW03] LOSASSO F., HOPPE H., SCHAEFER S., WARREN J. D.: Smooth geometry images. In *Symposium on Geometry Processing* (2003), pp. 138–145.

[MKE03] MEZGER J., KIMMERLE S., ETZMUSS O.: Hierarchical techniques in collision detection for cloth animation. In *Journal of WSCG* (2003), vol. 11, pp. 322–329.

[PG95] PALMER I. J., GRIMSDALE R. L.: Collision detection for animation using sphere-trees. *Computer Graphics Forum 14*, 2 (1995), 105–116.

[PH03] PRAUN E., HOPPE H.: Spherical parametrization and remeshing. *ACM Trans. Graph. 22*, 3 (2003), 340–349.

[SWG*03] SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H.: Multi-chart geometry images. In *Symposium on Geometry Processing* (2003), pp. 146–155.

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 24*, 1 (2005), 61–81.

[vdB97] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2*, 4 (1997), 1–14.

[VMT94] VOLINO P., MAGNENAT-THALMANN N.: Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum 13*, 3 (1994), 155–166.

[WB04] WONG W. S.-K., BACIU G.: Hardware-based collision and self-collision for rigid and deformable surfaces. *Presence 13*, 6 (2004), 681–691.

[WB05] WONG W. S.-K., BACIU G.: GPU-based intrinsic collision detection for deformable surfaces. *Computer Animation and Virtual Worlds 16*, 3–4 (2005), 153–161.

[Zac98] ZACHMANN G.: Rapid collision detection by dynamically aligned DOP-trees. In *VRAIS '98: Proc. of IEEE Virtual Reality Annual International Symposium* (1998), pp. 90–97.