GPGPU: Fast and easy Distance Field computation on GPU

by

Kent Roger B. Fagerjord

and

Tatyana V. Lochehina

NARVIK UNIVERSITY COLLEGE
JULY 2008

Contents

Abstract

The main purpose of this thesis is to create a fast and effective distance field computation algorithm and implement it on the graphics device (GPU). Our goal is to achieve the most accurate collision detection of large and complex objects with almost no cost.

In order to propose a solution we studied all previous works that we could find about distance fields and related areas (they all summarized in chapter 2 of this work). Most of them could not help us since the algorithms were created specially for central processing units (CPU) and thus either take much time or suffer great inaccuracy.

Lately, a number of works directly related to our problem were published. We combined some of their techniques together with the modern technology and finally created a simple, accurate and fast algorithm that we describe in details in chapter 4.

Our experimental implementation of the algorithm showed really amazing results. Briefly, the algorithm allowed us to run collision avoidance simulations in real-time with meshes containing several millions of triangles. Of course, this is not only the algorithm that matters when we talking about real-time simulations. Detailed results are described in chapter 5.

In theory, our distance field algorithm could greatly improve the performance and accuracy of the applications utilizing large and complex models in dynamic environments. It could be applications that simulate dense and highly detailed environments where collision avoidance is vital.

Chapter 1

Introduction

In this paper we investigate the problem of rapid distance computation between rigid bodies, which is important in the context of physically based simulation.

To propose a solution we take a look at three main features of our problem: GPU programming, distance field computation algorithm and collision detection algorithm. This combination is a relatively new field of research that opens many opportunities in fast and precise simulations of large complex models.

A main research area of this work is a distance field computation algorithm (DiFi), since the exactness of collision detection areas is often a bottleneck in many real-time applications. Many CPU based distance field computation methods have been developed since the beginning of the 1990s. The graphics hardware based algorithms came first with the work of Hoff et al. in 1999. We combine some already known techniques about distance fields and propose in this paper a new GPU based distance field computation algorithm for triangulated meshes that is simple, accurate and fast.

GPU programming was a kind of a wish dream before due to its unavailability. Now we use this magnificent technology to solve this particular problem. The results of using GPU in comparison with CPU are amazing. Heavy calculations accelerate dozens of times, CPU time decreases since computations are performed on the GPU, a 100% accuracy in distance computations achieves with almost no cost and, possibly the most important, it is able to run real-time physically based simulations with really large objects.

In Figure 1.1 an example is given. The example is a distance field generated entirely on GPU around a triangulated mesh 'Bender'. Distances are colored increasing from red to green and to blue. A model has 12k polygons and time taken to compute the distance field around this model with our algorithm is 0,02 sec.

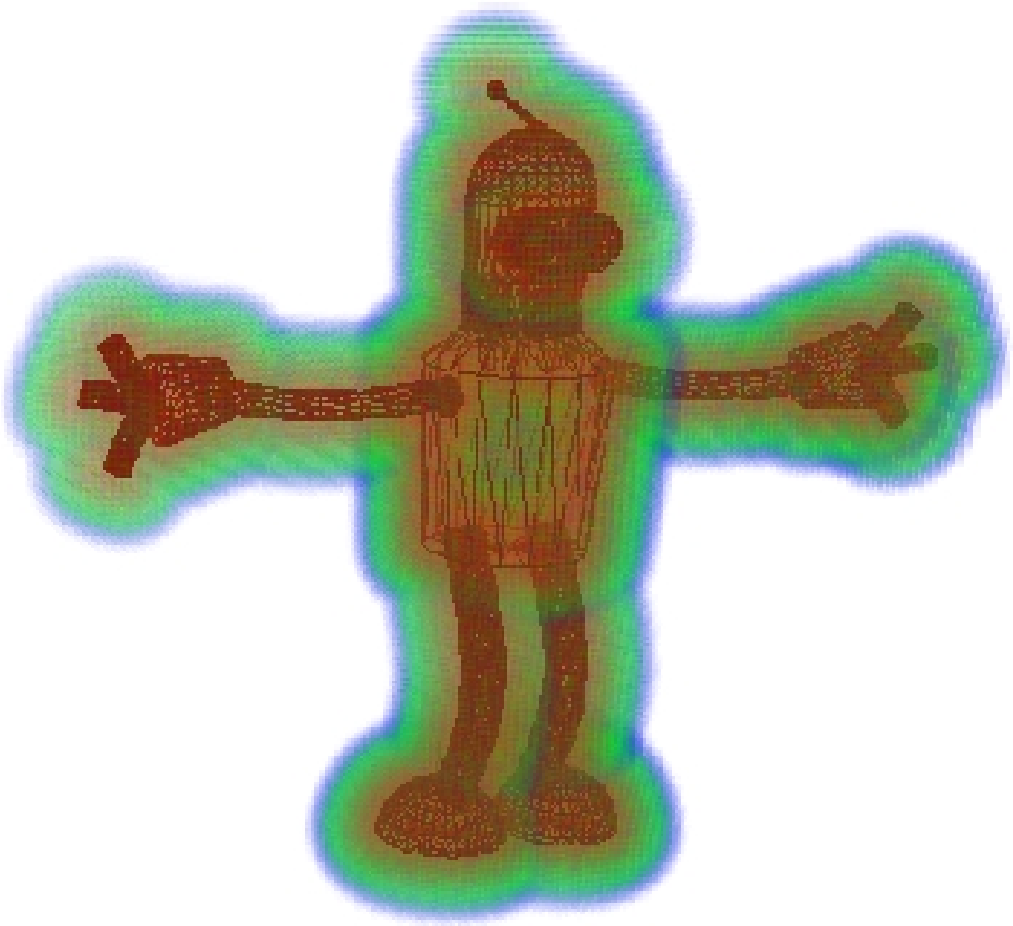

Figure 1.1: *3D Distance Field of Bender model (12k polygons): distance to the surface is color coded, increasing from red to green and to blue. Time taken to compute the distance field on 150x150x50 grid using our algorithm is 0,02 sec.*

Chapter 2

Related work

The purpose of this chapter is to give a brief description of previous research works on distance fields and distinguish the most important ones that affected finally our distance field computation algorithm.

The problem of computing a distance field can be broadly classified by the type of input object representation. This can either be a data on a voxel grid (volumetric models, binary image) or an explicit surface representation (geometric models).

2.1 Volumetric models

This is probably the earliest research area in distance field computation algorithms. With a given voxel data, many exact and approximate algorithms for distance field computation have been proposed.

In an early paper on discrete distance fields, Payne and Toga proposed a surface manipulation technique that uses distance fields [18]. Mark Jones found that using the above method resulted in a fourfold increase in efficiency when compared to naively computing the distance in 3D. Jones is only interested in voxels within a certain distance of the mesh and performs an initial scan conversion to find voxels that are adjacent to the mesh. Distances are only computed for those voxels. Furthermore, the accurate distance to a given triangle is only computed if the voxel is within a given distance of the triangle plane [9] [7]. Dachille and Kaufman proposed a method that is suitable for hardware implementation [5] .

Sethian introduced the so-called level set method to the field of computer graphics, which describes an evolving front by a partial differential equation. Once a distance shell has been computed it can be used as initial value for a level set based fast marching algorithm [18] [10] . The algorithm has become very popular in the intervening period. It is extremely adaptable and has found many uses in computer vision and computer graphics. Examples include stereo, image and volume segmentation, statistical modeling, simulation of water, shape metamorphosis and shape modeling.

A completely different approach has been suggested by Sean Mauch [13] . The idea is to create (truncated) Voronoi regions for every face, edge and vertex in the mesh. These Voronoi regions are represented as polyhedra which, in turn, are scan converted. The regions corresponding to edges and faces will be either interior or exterior to the mesh depending on whether the mesh is locally concave or convex. The Voronoi region of a face will straddle the mesh, but if the closest point is on a face there will be no sign ambiguity. Thus, Mauch's method handles sign correctly, although the author does not discuss the issue.

Recently, André Guéziec proposed a new method called the "Meshsweeper" algorithm. This algorithm is based on a hierarchy of simplifications of the mesh. The distance is first computed at a coarse level, and then we proceed to finer levels. The advantage is that at any level, it is possible to prune branches if they cannot contain the shortest distance [6] .

R. Satherley and M. Jones came with a hybrid distance field computation method based on distanse transforms passed through the dataset. The sub-voxel accurate distance shell is propagated in this method using the most accurate 8 pass VCVDT, and then those vectors are used to direct a further pass through the data in order to calculate the distance to the correct sub-voxel surface. This method produces the correct distance for 90% of voxels, whereas the previous best (EVDT on binary segmented data) produces the correct distance to less than 1% of voxels [20].

Huang et al. proposed a new distance field representation called CDF (Complete Distance Fields). The central idea is to divide a volume into cells, and in each cell to store references to all triangles that might contain the point closest to any point inside the cell [8].

A simplified algorithm for generating a discrete, signed 3D distance field was proposed by J. Andreas Bærentzen and Henrik Aanæs in 2002. The algorithm is based on a level set method proposed by Osher and Sethian [7].

2.2 Geometric models

Many distance field computation algorithms are known for geometric models represented using polygonal or higher order surfaces. These algorithms use either uniform grid or an adaptive grid.

Adaptively sampled distance fields (ADFs) were proposed as a new fundamental graphical data structure [22]. A key issue is the underlying sampling rate used for adaptive subdivision. Many adaptive refinement strategies use trilinear interpolation or curvature information to generate an octree spatial decomposition [24][22][21][19][11].

The graphics hardware based algorithms for distance field computation came first with the work of Hoff et al. [12] . They render a polygonal approximation of the distance function on the depth-buffer hardware and compute the generalized Voronoi Diagrams in two and three dimensions.

Characteristics/Scan-Conversion (CSC) algorithm is a scan-conversion method to compute the 3-D Euclidean distance field in a narrow band around manifold triangle meshes. The CSC algorithm uses the connectivity of the mesh to compute polyhedral bounding volumes for the Voronoi cells. The distance function for each site is evaluated only for the voxels lying inside this polyhedral bounding volume. An efficient GPU based implementation of the CSC algorithm is presented by Sigg et al. [3]. The number of polygons sent to the graphics pipeline is reduced and the non-linear distance functions are evaluated using fragment programs [24].

Recently, a fast algorithm (DiFi) was proposed by Sud et al. to compute a distance field of complex objects along a 3D grid. They used a combination of novel culling and clamping algorithms that render relatively few distance meshes for each slice. They also exploit spatial coherence between adjacent slices in 3D and perform incremental computations to speed up the overall algorithm. Their novel site culling algorithm uses properties of the Voronoi diagram to cull away primitives that do not contribute to the distance field of a given slice. The two-pass approach and culling using occlusion queries is used. Furthermore, they present a conservative sampling strategy that accounts for sampling errors in the occlusion queries. The clamping algorithm reduces the rasterization cost of each distance function by rendering it on a portion of each slice [24] . The algorithm gives pleasing results but is rather complicated.

The algorithm that resembles somewhat our algorithm was proposed by Fuhrmann et al. [1]. For every triangle of the mesh they construct a prism by moving its vertices along the face normal. The axis-aligned integer bounding box enclosing the prism is determined. Then for all grid points that lie in the bounding box the distance to the triangle is computed. The resulting distance field is signed.

Chapter 3

Preliminary study of GPUs

In this chapter we give a brief introduction into the field of GPU, explore its architecture, study GPU programming principles and a new architecture for computing on NVIDIA graphics processing units CUDA.

GPU is an acronym which stands for Graphics Processing Unit. Simply put, it is a processor that resides on the graphics card. Graphics cards starting with the GeForce 8 series have become very programmable and are considered to be highly parallelized computing devices. Nowadays GPUs are able to accelerate certain algorithms >10x and are able to solve many problems not only within computer graphics, but also general computing problems that are not related with computer graphics at all. This is the next generation General Purpose GPU (GPGPU).

The architecture is briefly described in section 3.2. GPU programming methods in general and a description of CUDA that was used in our application are presented in section 3.3.

3.1 Hardware

A GPU is a processing unit, which consists of many integrated processing units. The actual numbers depends on model and make. It is organized in a way that a single instruction is executed on a group of processors simultaneously, but operating on different data. This technique is called "Single Instruction Multiple Data" – SIMD for short.

The graphics hardware utilized in this diploma is a NVIDIA GeForce EN8800 GTS 512MB (G92). It has 128 processing units which runs at a core speed of 1625MHz. 512MB of GDDR3* memory is present which runs at 1000MHz with a 256-bit interface. Theoretical transfer rate is 64GB/sec [16]. Complete specification of the system used for this diploma thesis is presented in table 3.1.

Many improvements have been made in the architecture compared to previous platforms.

*GDDR3 - third-generation double-data-rate memory for graphics

Component	*Description*
CPU	AMD Athlon 64 X2 Dual Core 3800+
RAM	4094MB
GPU	GeForce EN8800 GTS 512MB
OS	Windows XP 64-bit, Gentoo 64-bit

Table 3.1: *Specification of the system used for application implementation*

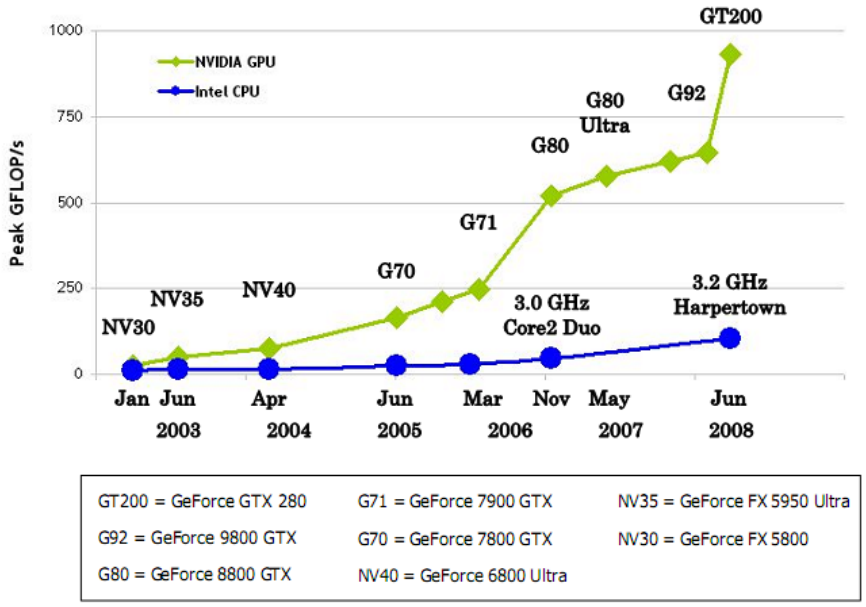

Figure 3.1: *Peak GFLOP/s comparison between Intel CPUs and NVIDIA GPUs [17]*

The cost of branching has decreased[†], which means that more complex programs performs better [14] .

As illustrated in figure 3.1 and 3.2, the computational power and memory bandwidth have outperformed regular CPUs for several years already, and the GPUs are not waiting for the CPUs to catch up. In the future more and more applications will benefit from the increasing GPU computational power.

3.2 GPU Architecture

The CPU and GPU architectures are quite different. While the CPU is built up with strong control flow capabilities, the GPU is built with performance and throughput in mind, which is ideal for graphics oriented applications.

The CPU performs very well with logic and code path branching. To gain parallel pro-

[†]Some would argue not enough

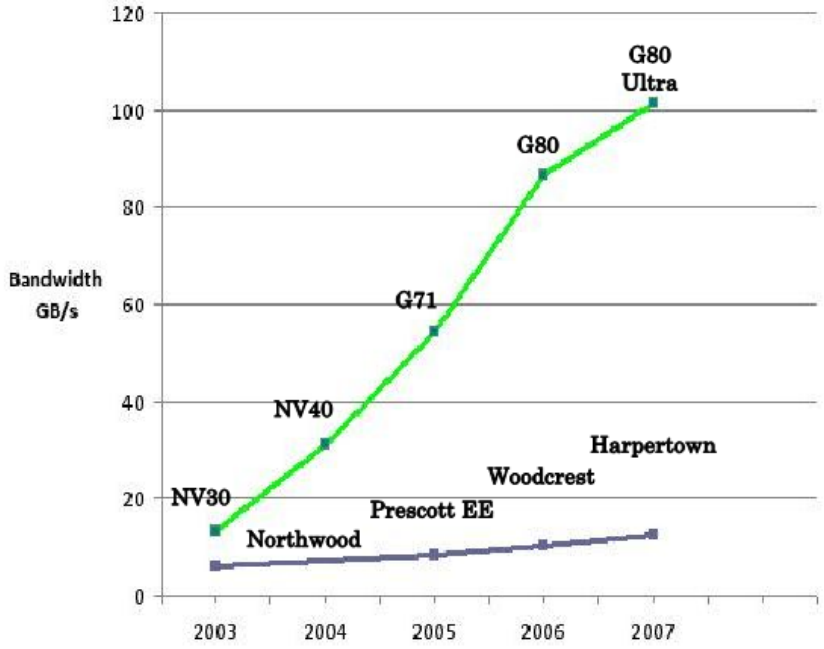

Figure 3.2: *Bandwidth comparision between Intel CPUs and NVIDIA GPUs [17]*

cessing performance, we have to have a defined problem domain with a solution requiring many repetitive tasks operating on different data. The way GPUs are designed, fits these problems well. The GPU have many processors and a few instruction units controlling a set of processors.

The instruction unit has a "pool" of instructions waiting in a queue, and the processors have a data pool with data to be processed. The same instruction is issued from an instruction unit to the processors controlled by that unit, and then the processors perform that instruction on different data. That is why the code instruction path must follow the same instruction path as every other thread. If there are two threads following different code paths, the code paths must be serialized, losing the benefit and performance of parallel processing.

Figure 3.3 explains this very well, although somewhat simplified for illustration purposes. On the GPU, DRAM is the memory referred as device memory later on. On the CPU the DRAM is the system memory, while the cache is on-chip fast access memory on both GPU and CPU. The control unit on the CPU controls the rest of the CPU. The control unit decodes the instructions, fetches data to and from system memory, issues instructions to the logic units, stores intermediate results and so on.

The cache and control unit on the GPU have a different layout to enable effective and true parallel processing, as the GPU figure shows us. A GPU have multiple control units with their own cache. Each control unit have several processors and logic units. This enables true parallelism because each control unit issues the same instruction to all the processors it controls after having prepared the (different) data for each processor.

Figure 3.3: *CPU and GPU architecture as illustrated in the CUDA Programming Guide [17]*

Different methods of memory access are available with different kinds of caching and with lower latency [15]. Some regions in memory can be accessed using a 2D cache. This means that small regions of an image are cached rather than a single line or a part of a single line. Another memory type is shared memory. Shared memory is shared between threads and is the fastest memory available for threads (and thread communication). Also the transfer rate between host and device is very high. This makes it more economical to divide work between CPU and GPU, and also to work on very large datasets. A medical dataset can easily reach 30GB or more [15].

3.3 GPU-programming

Even nowadays some GPU architecture details are hidden, although less now than previously. OpenGL and DirectX provides a state machine that represents the rendering pipeline. Early GPU programs used properties of the state machine to "program" the GPU. Recent GPUs provide high level programming languages to work with the GPU as a general purpose processor.

3.3.1 Programming methods

All started with "programming" in OpenGL using state variables like blend functions, depth tests and stencil tests [26]. As the rendering pipeline became more complex, new functionality was added to the state machine (via extensions). With the introduction of vertex and fragment programs, full programmability was introduced to the pipeline.

Finally, high level programming became possible and high level languages like HLSL, Cg, GLSL, CUDA, BrookGPU and Sh were created. With this general purpose programming has become easy (see Table 3.2 for an example).

```
// overlap test between AABBs
inline __host__ __device__ bool testAABBAABB(const AABB &a, const AABB &b)
{
        //Exit with no intersection if separated along an axis
        if (a.mx.x < b.mn.x || a.mn.x > b.mx.x) return false;
        if (a.mx.y < b.mn.y || a.mn.y > b.mx.y) return false;
        if (a.mx.z < b.mn.z || a.mn.z > b.mx.z) return false;

        //Overlapping on all axes means AABBs are intersecting
        return true;
}
```

Table 3.2: *Fragment of the implementation of collision detection on GPU with CUDA (See Chapter 5)*

Programming on GPU is rather different from that on CPU. Some things the programmer have to have in mind when programming the device are,

1. Conditional statements do not work well on parallelized processors.
2. There is also a limit on how many variables one can use in the kernel.
3. Device memory is limited.

3.3.2 CUDA Introduction

CUDA (Compute Unified Device Architecture) is technology that allows a programmer to use the C programming language to implement parallel algorithms for execution on the GPU [27]. CUDA represents a new way of programming the graphics device. A developer doesn't need any knowledge of graphics programming or a specific graphics API (OpenGL/Direct3D) to program a GPU with CUDA. All the programmer need to know, is how to use an (external) compiler, some C and general understanding of parallel programming.

With CUDA, there are two separate execution environments. There are additional ones if you have multiple CUDA-enabled graphics devices. In CUDA terms, the “host” is the CPU with system memory (RAM). The “device” is the GPU with GPU memory.

CUDA is at present point something relatively new. And as with other new technologies, there aren't many resources available on the Internet. The only resources for us have been the programmers manual, the examples provided with the CUDA SDK and API and some sparse topics at the CUDA part of NVIDIA's forums. One key step in this diploma, was to understand the basics and features of CUDA before we could design and implement an algorithm. This includes not only how the host and device code of CUDA works, but also setup and usage of nvcc (NVIDIA CUDA Compiler), linking CUDA C to existing C++ code and integration with OpenGL.

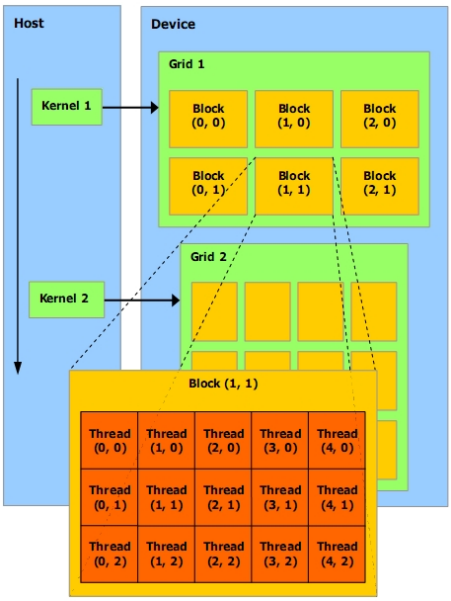

Figure 3.4: *Each kernel is executed as thread in a batch of threads organized as a grid of thread blocks [15]*

The 8-Series (GeForce 8x) GeForce-based GPU from NVIDIA is the first series of GPU to support CUDA. These cards features hardware support for 32-bit single precision floats. CUDA supports the C double data type, however on GeForce 8x series these types will get demoted to 32-bit floats.

At the core of CUDA computing, are the kernels. A kernel is a function intended to run on the device, invoked from the host. These kernels are executed in true parallel. A kernel can also be viewed as a set of instructions running on the device, but working with different data in parallel. This is also known as SIMD (Single Instruction, Multiple Data). The idea is that a set of processors performs a given instruction in parallel, but with different data.

Advantages:

1. Ability to program the device directly without a graphics API.
2. Easy to debug the kernel in emulation mode.
3. Can utilize multiple devices not in Sli.
4. Extra functionality like scattered write

Disadvantages:

1. New technology, few resources on the Internet.
2. Vendor limited to NVIDIA.

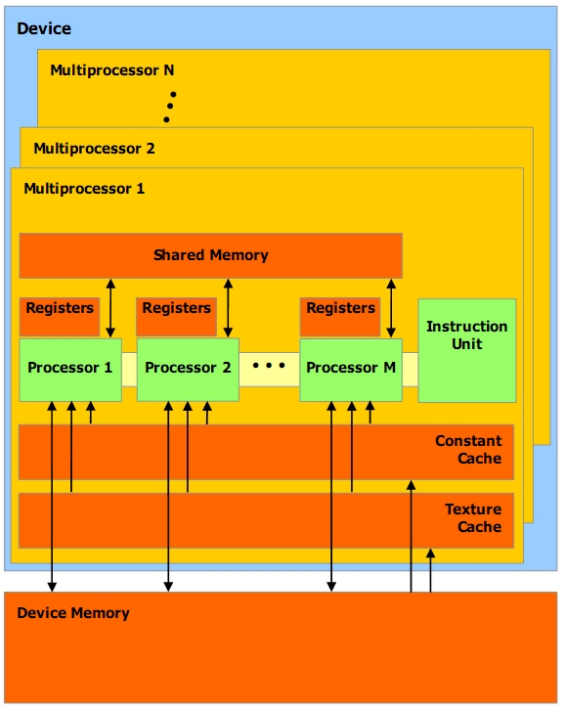

Figure 3.5: *Schematic overview of the processor architecture.*

3.3.3 CUDA execution details

A **kernel** is a set of instructions set to run on the GPU in parallel in blocks of threads. For optimum performance, the data can be different, but the instructions can not differ. But due to the complexity of certain problems, the cost of branching has decreased in later revisions of GPUs because they have better branch prediction and are able to mark instruction in threads with a predicate to allow or deny a thread to actually execute that instruction. A CUDA kernel is compiled and linked to the main program in the build process, and not compiled and linked during runtime like GLSL. When the program is executed, the compiled kernel is downloaded to the device[17]. See Figure 3.4

Each kernel represents a thread running one execution path. The number of instructions in a thread and the number of threads determine how expensive a computation is. Readbacks from the device to the host are somewhat different from traditional texture reads. There are optimized CUDA API calls designed to operate on linear, 2D and 3D memory. The most efficient way is problem specific and have to be tested in each case.

Some other things to keep in mind when designing and implementing a kernel. The function parameter list holds a maximum of 256 bytes of data, which is stored in shared memory when the thread executes. One block is executed on one multiprocessor, each multiprocessor has a set of processors with registers shared between the threads in the block. If a kernel have many variables and the number of threads in a block is high, the resources are exhausted, resulting in a kernel launch failure.

Our GPU contains 16 multiprocessors and 128 stream processors.

A warp is a subset of a block, which consist of the threads actually running in parallel. Consequently, a warp size is the number of threads in a warp, and a half-warp is either the first or last 16 threads in a warp.

The programmer must determine how many threads that the block should have. This amount is most optimal if it is a multiple of the warp size. Devices with compute capability 1.x have a defined warp size of 32.

Something which is related to the block size, is a bank conflict. A bank conflict occur when two threads access the same memory bank simultaneously. Then the access must then be serialized. This happens in certain configurations of block sizes where the half-warps access the same memory bank at the same time. To avoid wasting clock cycles, the recommended block size must be a multiple of 64. The CUDA Programming Manual [17] explains this in detail.

Also, an access to a register usually have no overhead per instruction, but there are circumstances the access could be delayed. The manual states that when then are at least 192 active threads per multiprocessor, this delay could be ignored. This implies there should be at least 192 threads per block, if possible.

The number of blocks in a grid must also be taken into consideration. The number of blocks per grid should be at least the number of multiprocessors , most preferably two times or more blocks per multiprocessor so that the execution can be pipelined. The manual states that 100 or more blocks per grid should cover future revisions, while 1000 or more blocks per grid will scale across several generations.

Chapter 4

Distance Field

This chapter is the main research area of our paper. The purpose of this chapter is to introduce our distance field computation algorithm, give definitions, explore some help algorithms, study possible applications of the algorithm and compare our method with some other distance field computation methods.

4.1 Distance Field algorithm

Let S be a surface such that $S \in \mathbb{R}^3$ then a distance field of this surface will be a scalar function.

Definition 4.1.*A distance field is a scalar field that specifies the minimum distance to a shape and is defined as follows:*

$$D : \mathbb{R}^3 \to \mathbb{R},$$

$$D(r,S) \equiv \min_{s \in S} \{|r-s|\}, \forall r \in \mathbb{R}^3$$

The distance may be signed to distinguish between the inside and outside of the shape but it makes sense only in the case of a closed surface. Here we present an unsigned distance field for all kind of triangulated surfaces.

The algorithm can be summarized as follows (see Algorithm 1):

Step 1. For every triangle of the mesh an axis-aligned bounding box is calculated by determining a minimum and a maximum points of the bounding box from the points of the triangle. The box is expanded by the thickness of the distance field around the triangle. See Figure 4.1.

Step 2. Optimizing. The calculated bounding box of the triangle compares with the bounding boxes of previous and next triangles of the mesh (see Figure 4.2). If the box belongs to one of the other bounding boxes, the triangle skips from further calculating. The loop continues until all possible triangles are skipped. This step is very important for further parallelized computing due to the nature of the CUDA kernel operations. The

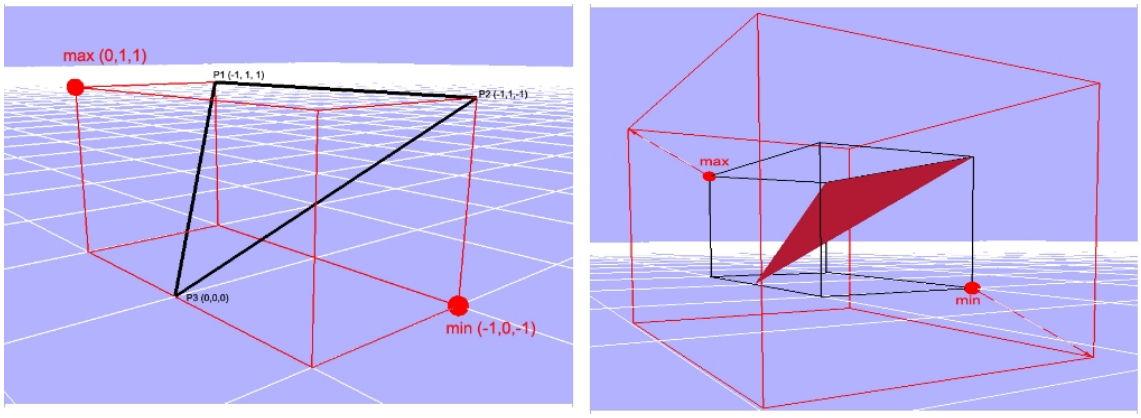

Figure 4.1: *An axis-aligned box (left) of the triangle is calculated by determining a minimum and a maximum points of the bounding box from the points of the triangle. The box is expanded by the thickness (threshold) of the distance field (right).*

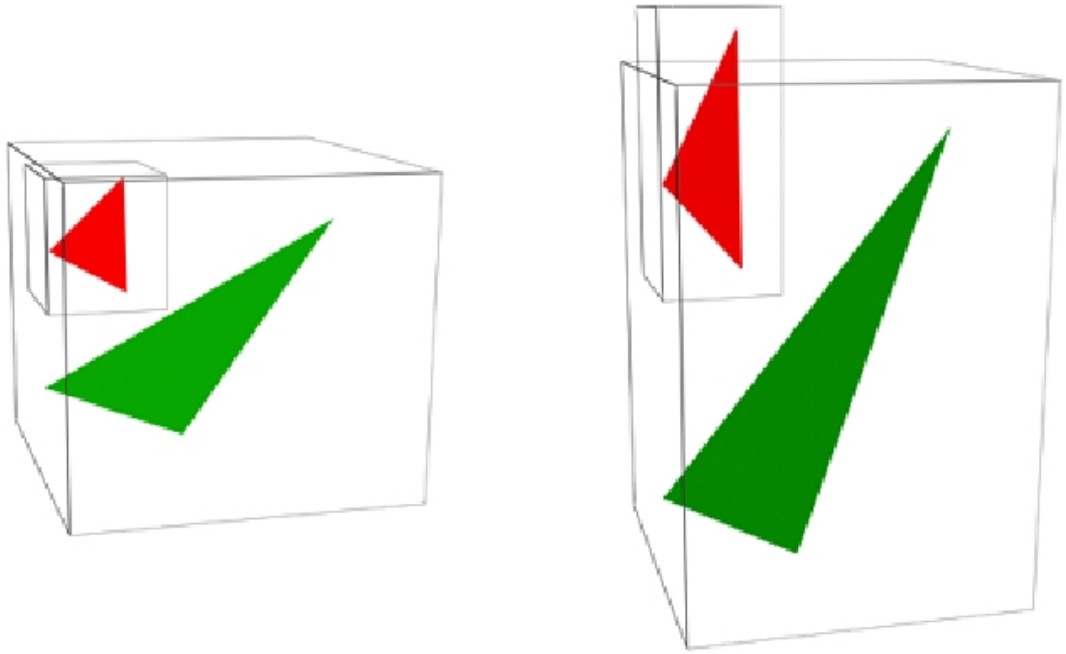

Figure 4.2: *Triangle skip. If the bounding box of the smaller triangle belongs to the bounding box of the other triangle (left), the triangle skips from calculation. But if we stretch the mesh in one way, a triangle may not already be skipped.*

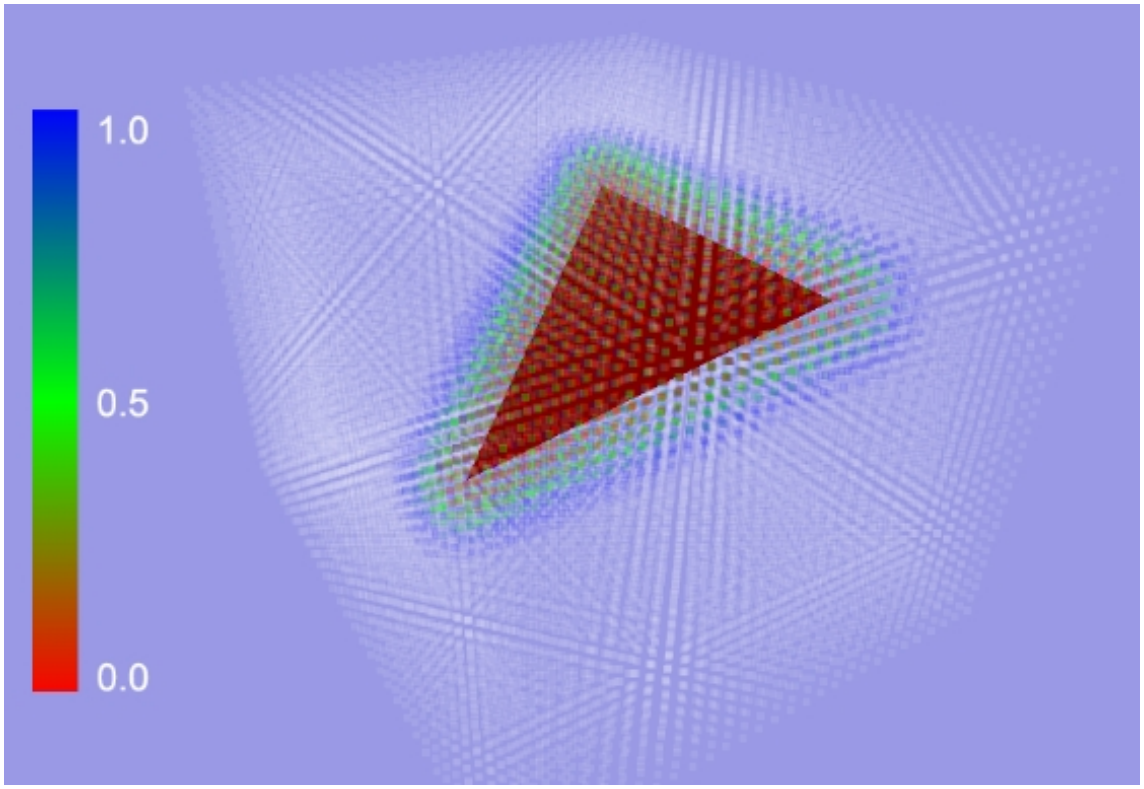

Figure 4.3: *For all grid points that lie in the bounding box the squared distance to the triangle computes. The distance values are colored from red (nearest) to blue and white (far away).*

CUDA kernel launches the threads in warps. Our warp size is 32 [17]. The runtime of a warp is equal to the runtime of the longest living thread. And because of the fact that threads are run in warps, no threads will be launched when a thread is finished before other threads. All 32 new threads in the new warp, will be launched when the old warp is finished. If there are 31 triangles in a warp marked for skipping, the 1 remaining triangle will still be calculated, wasting at least 31 triangles worth of computing time. It's therefore important to rearrange the entire triangle memory structure, eliminating the unneeded triangles from memory.

Step 3. If the triangle is not skipped from calculating, for all grid points that lie in the bounding box the squared distance to the triangle computes (see Figure 4.3). Here comes some optimizing opportunities with the computing of the minimum distance between a given point and a triangle. The current distance updates to a new value if the calculated absolute value is less than the current.

The asymptotic time complexity of the algorithm is $O(nm)$ where n is the number of triangles that are taken to calculation and m the number of grid points in bounding boxes.

One might wonder if this algorithm produces a valid distance field despite triangle skip. Indeed, some errors in distance values can become a problem if the size of the triangles becomes too large. Thats why this algorithm prefers many small triangles against few

large (see section 5.2.1.2 for details).

A triangle skip operation in this algorithm is also highly customizable. Per default it is meant to skip only those triangles whose bounding boxes completely belong to the bounding boxes of other, larger triangles. But in cases where distance field thickness is large, other belongings percent can be used. For example, we can determine that the triangle skips if its bounding box belong to the bounding box of other triangle more than 75%.

Input: triangular mesh T, three-dimensional Cartesian grid G
Output: distance field D

forall $t \in T$ **do**
  construct bounding box $G_b \leftarrow AABB(t)$
  if $G_b \in G_{b-1} \vee G_b \in G_{b+1}$ **then**
    break
  end
  else
    $set(t^{'}) \leftarrow t$
  end
  $T(t^{'}) \leftarrow set(t^{'})$
end
forall $t^{'} \in T$ **do**
  forall $p \in G_b \cap G$ **do**
    $d \leftarrow min\left\{\|p - s(t^{'})\|\right\}$
    $D(p) \leftarrow min\left\{\text{abs}(d), \text{abs}(D(p))\right\}$
  end
end

Algorithm 1: Distance field (T, G)

4.2 Closest point to triangle test

One of the bottlenecks in our distance field algorithm (see section 5.2.2) is a closest point to triangle test. This problem can be seen as a minimization problem. One approach is to formulate the minimization problem and solve it using methods of calculus (such as the method of Lagrange multipliers). We preferred here a more geometric approach proposed by Christer Ericson [4] although it is not the most effective one.

Given a triangle ABC and a point P, let Q describe the point on ABC closest to P. One way of obtaining Q is to compute which of the triangle's Voronoi feature regions P is in. Once determined, only the orthogonal projection of P onto the corresponding feature must be computed to obtain Q.

A detailed description of this test, including pseudocode, is given in [4]. A preliminary version of the test contains four cross-product calls. Because cross products are often more expensive to calculate than dot products, we have implemented an optimized version

of the test where four cross products are changed into six dot products by the Lagrange identity:

$$(a \times b) \cdot (c \times d) = (a \cdot c)(b \cdot d) - (a \cdot d)(b \cdot c)$$

There are some other ways of obtaining the closest point. We have not studied all of them in order to find the most effective one, since it is not an issue in this diploma work.

4.3 Minimum Distance Field

During our experiments we found out that it is possible (and desirable) to construct a minimum discrete distance field that will still be effective for collision detection.

Let S be a surface such that $S \in \mathbb{R}^3$ and p be a grid point such that $p \in G$

Definition 4.2. *A minimum distance field is a minimum scalar field that specifies the minimum distance to a shape and is defined as follows:*

$$D^{min} : \mathbb{R}^3 \rightarrow \mathbb{R},$$

$$D^{min}(G,S) \equiv \min_{s \in S} \{|p-s|\}, \forall p \in G \wedge p \rightarrow 0$$

More precisely, a *minimum distance field* is a minimum number of grid points in a grid. The minimum size of grid can be calculated by the formula:

$$G^{min} = \frac{(p^{max} - p^{min})}{dist} * e, \text{ where}$$

p^{max} and p^{min} – maximum and minimum points of the mesh extended with a threshold value;

dist – thickness of the distance field, threshold;

e – correcting value that we recommend as 2-4 (depends on the model and speed of dynamic objects).

Note that G^{min} is a vector and the number of minimum grid points calculates for all three components that are x, y and z.

The calculation of a minimum distance field accelerates the whole computation in dozens of times (see section 5.2). We can for some medium sized models even run a real-time simulation without any difficulties for presentation speed.

A minimum distance field has same characteristics as a normal distance field and the most important is that it detects a collision in the same effective way as a more tight distance field (see Figure 4.4). Red points on the figure show that they have detected a collision with the distance field of another object.

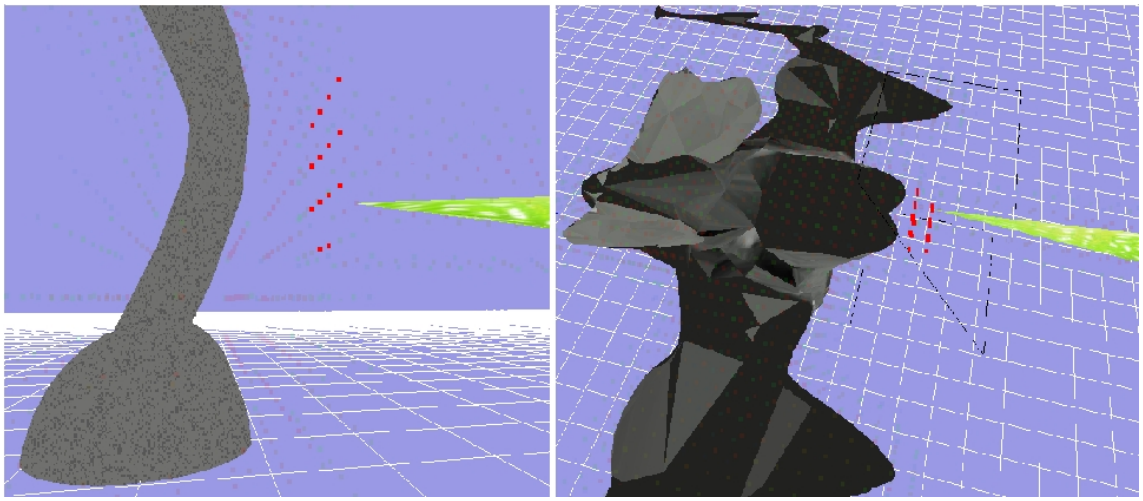

Figure 4.4: *Collision detection with a minimum distance field is still effective*

4.4 Comparison to other methods

Our algorithm described in this paper has some very important new features that give it obvious advantages over previous works. First, the algorithm is specially made for parallel computing devices that accelerates the whole computation dozens of times and gives the opportunity to work with really large meshes. Furthermore, the meshes are preferred to be oriented, but it is not obligatory. The algorithm works fine with all kind of triangulated meshes, the only thing is that if the mesh is not oriented almost all triangles will be taken into computations and it will increase a time complexity of the algorithm to $O(nm)$ where n is the number of triangles in the mesh and m the number of grid points in bounding boxes.

We compute a minimal needed for efficient collision detection distance envelope around the surface (see section 4.3). This accelerates a runtime of the whole physically based simulation and still gives truly fast collision response.

The graphics hardware based algorithms for distance field computation came first with the work of Hoff et al.[12]. They render a polygonal approximation of the distance function on the depth-buffer hardware and compute the generalized Voronoi Diagrams in two and three dimensions.

Breen et al. [2] first built up the whole Voronoi diagram for the faces, edges and vertices of the triangular mesh. They used scan conversion to determine which cells of the grid lie in each Voronoi region. Unfortunately they do not give any information about computation times.

A fast algorithm (DiFi) by Sud et al. [24] used a combination of novel culling and clamping algorithms that render relatively few distance meshes for each slice. They also exploit spatial coherence between adjacent slices in 3D and perform incremental computations to speed up the overall algorithm. Their novel site culling algorithm uses properties of the Voronoi diagram to cull away primitives that do not contribute to the distance field of

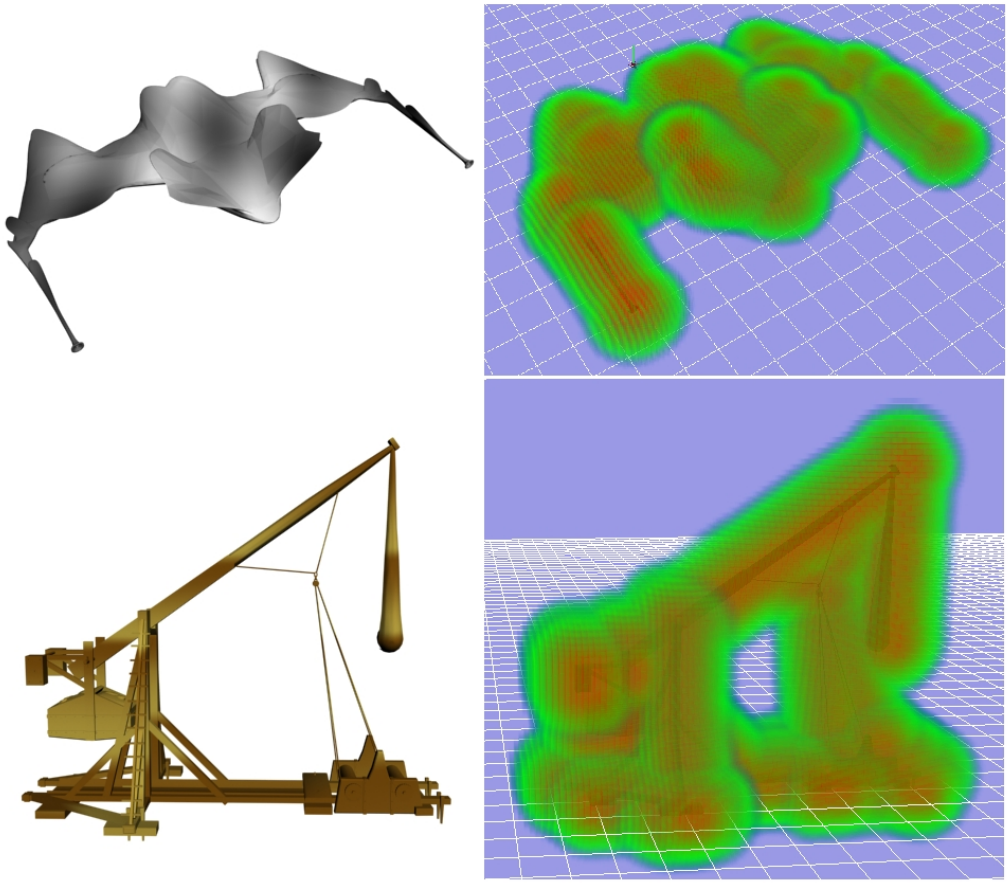

Figure 4.5: *Distance field computation colored results (Plane and Catapult)*

a given slice. The two-pass approach and culling using occlusion queries is used. Furthermore, they present a conservative sampling strategy that accounts for sampling errors in the occlusion queries. The clamping algorithm reduces the rasterization cost of each distance function by rendering it on a portion of each slice. The algorithm gives pleasing results but is rather complicated.

In comparison to our method Fuhrmann et al. [1] first for every triangle of the mesh calculates a prism by moving its vertices along the face normal. Then the minimum distance between a point and a triangle algorithm divides the triangle bounding box volume into a positive and a negative part. The asymptotic time complexity of their algorithm is *O(nm)* where *n* is the number of triangles and *m* the number of grid points. The algorithm also suffers of sign errors if the size of the triangles becomes too large.

4.5 GPU based collision detection algorithm

In this section we propose a GPU based collision detection method between rigid bodies with distance field calculated. The important thing here is that the distance field is going

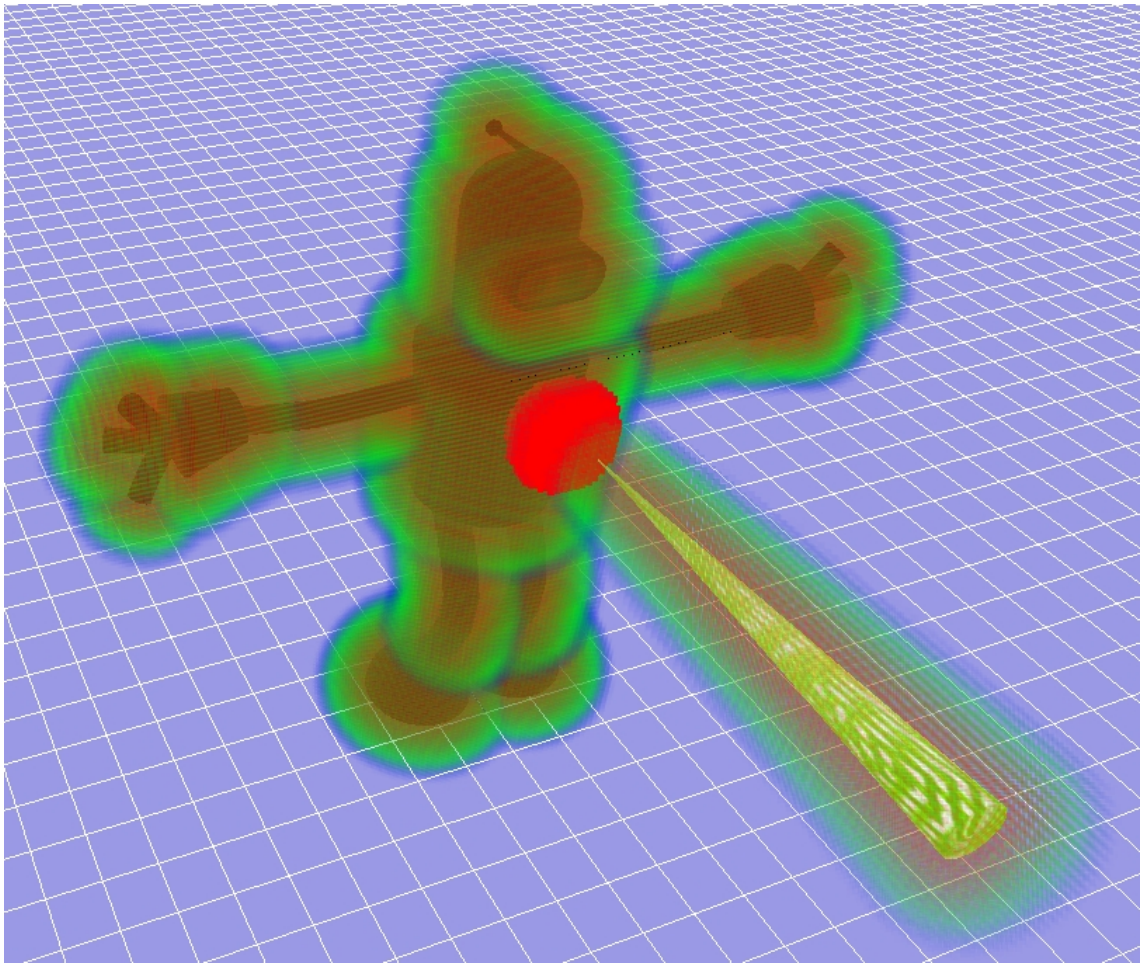

Figure 4.6: *Example of collision detection with distance fields (Bender and Cone)*

to help us in collision detection operation.

The most effective collision detection takes place when one have inexpensive intersection tests along with tight fitting. The fastest and cheapest test that we can think about today is AABB-AABB intersection test, where AABB is an axis-aligned bounding box. The best feature of the AABB is its fast overlap check, which simply involves direct comparison of individual coordinate values [4].

4.5.1 The simulation loop

Since we have already calculated a distance field ,we are going to use its benefits in our collision detection algorithm for GPU (see Algorithm 2 and Figure 4.6).

Step 1. We construct an axis-aligned bounding box around models that are going to collide. They have already bounding boxes that are called “grids” and are saved in the graphic card memory. We take the parameters of these boxes and construct bounding volumes of type AABB.

Step 2. With any movement of one of the objects we do overlap tests between AABBs. These tests are straightforward and regardless of representation. Two AABBs only overlap if they overlap on all three axes, where their extent along each dimension is seen as an interval on the corresponding axis [4]. For the min-max representation that we use here, this interval overlap test becomes:

Step 3. When bounding boxes overlap each other, we take the common volume and check grid points within this volume. If the point doesn't have a value, it skips from calculation. If there is a value in some point, we find it a nearest point in this volume that has a value and belongs to other distance field.

Step 4. If the sum of values in two nearest points that belong to different distance fields is less than a defined threshold value, we have a collision.

Note: Steps 1 and 2 are done entirely on the CPU. Due to the simple nature of our AABB representation (min/max position), the small number of distance field checks (<10), and the fact that the AABB is already stored in the host's memory due to our software design, it wouldn't be beneficial to copy the distance field data onto the device's memory and then invoke a kernel to perform the AABB-AABB intersection test on the GPU.

Input: distance field D, three-dimensional Cartesian grid G
Output: collision detection C

```
1  forall (D, G) do
2      construct axis-aligned bounding volume AABB(G_i) ← G_i
3      if AABB(G_i) ∩ AABB(G_{i+1}) then
4          forall p ∈ G_i ∩ V do
5              if p! = 0 then
6                  c ← Σ{p + q}
7                  if c ∈ D_i then
8                      C = true
9                      return
10                 end
11             end
12         end
13     end
14 end
```

Algorithm 2: Collision detection algorithm (D, G)

Chapter 5

Results

In this chapter we evaluate our implementation of the distance field and collision detection algorithms and their possible improvements. It is not possible to directly compare our results with the results obtained by other researchers mainly because of hardware differences. But we can show whether or not our distance field algorithm can be used for real-time physically based simulations of large and complex geometries.

5.1 Experimental environment setup

A number of experiments were conducted to evaluate the algorithm and its implementation. To do this, some evaluation material was gathered and then processed to extract useful information. This was necessary for our algorithm to find proper parameters of the objects that give best runtime results.

5.1.1 Obtaining evaluation material

A number of 3DS objects of various shapes and resolutions were gathered from different sources. Some were related to industrial applications and some were not related to anything in particular.

The main purpose of collecting different objects was to evaluate our implementation of distance field algorithm and to find some proper parameters for the objects that give best results. Another purpose was to test our collision detection algorithm along with the minimum distance field in order to define proper borders of the correcting value e that is used in our minimal distance field calculation. Therefore, there was a need for different objects exposed to various resolutions.

A total of 10 objects were produced and tested with different scaling factors (see Figure 5.1). The objects are gathered so that we could entirely test both distance field and

collision detection algorithms. Such parameters as number of triangles, size of triangles, object complexity, resolution and shape were taken into consideration.

Objects were created with minimum resolutions so that they could later be tested with different resolutions and grid sizes. Some objects were created with 3D Studio Max and some (Bender, Catapult, Fly and Plane) were downloaded from the Internet [25] .

5.1.2 Parameters

There are sometimes large differences between objects in the evaluation material which makes it difficult to choose general parameters for the object to be ideal for our algorithm. Almost for every object some tests had to be done to find the most optimal solution. The actual parameters of the objects used for evaluation were created in a way that they could be used in a real-time application.

Here is a list of parameters that were taken into consideration when taking the objects to evaluation:

- the number of triangles since this is the most obvious factor that affects the runtime.
- the size of triangles since this factor is quite important in our algorithm.
- resolution of the object that affects triangle sizes and thus runtime.
- complexity of the object that can include different sets of large and small triangles in the object.
- a shape of the object that was important in collision detection tests.

For this diploma work we tried to choose the objects so that we could test the algorithm in a comprehensive way and find the most general parameters for the “ideal” objects.

5.2 Distance Field computation algorithm

We have tested our distance field algorithm with several geometries (triangular meshes up to S2,8M faces) at variable resolutions (see Table 5.1). The resolutions marked as ^{min} are actual resolutions of 3DS objects. All other values are made by scaling the object ^{20} means scaling in 20 times). Distance thickness here is set to 2 units (note that runtime is different with different thickness and resolution values).

The evaluation material described in section 5.1 was used. It contains a number of objects with different parameters. Time was measured from the point the algorithm starts (the kernel invocation function is being called that starts with preparing a memory for reading(mapping)) to the point all device memory is ready for being used elsewhere (unmapped).

The raw results from the tests of our algorithm are runtime in seconds, the number of triangles in a mesh, the number of skipped triangles, grid, resolution and the number of

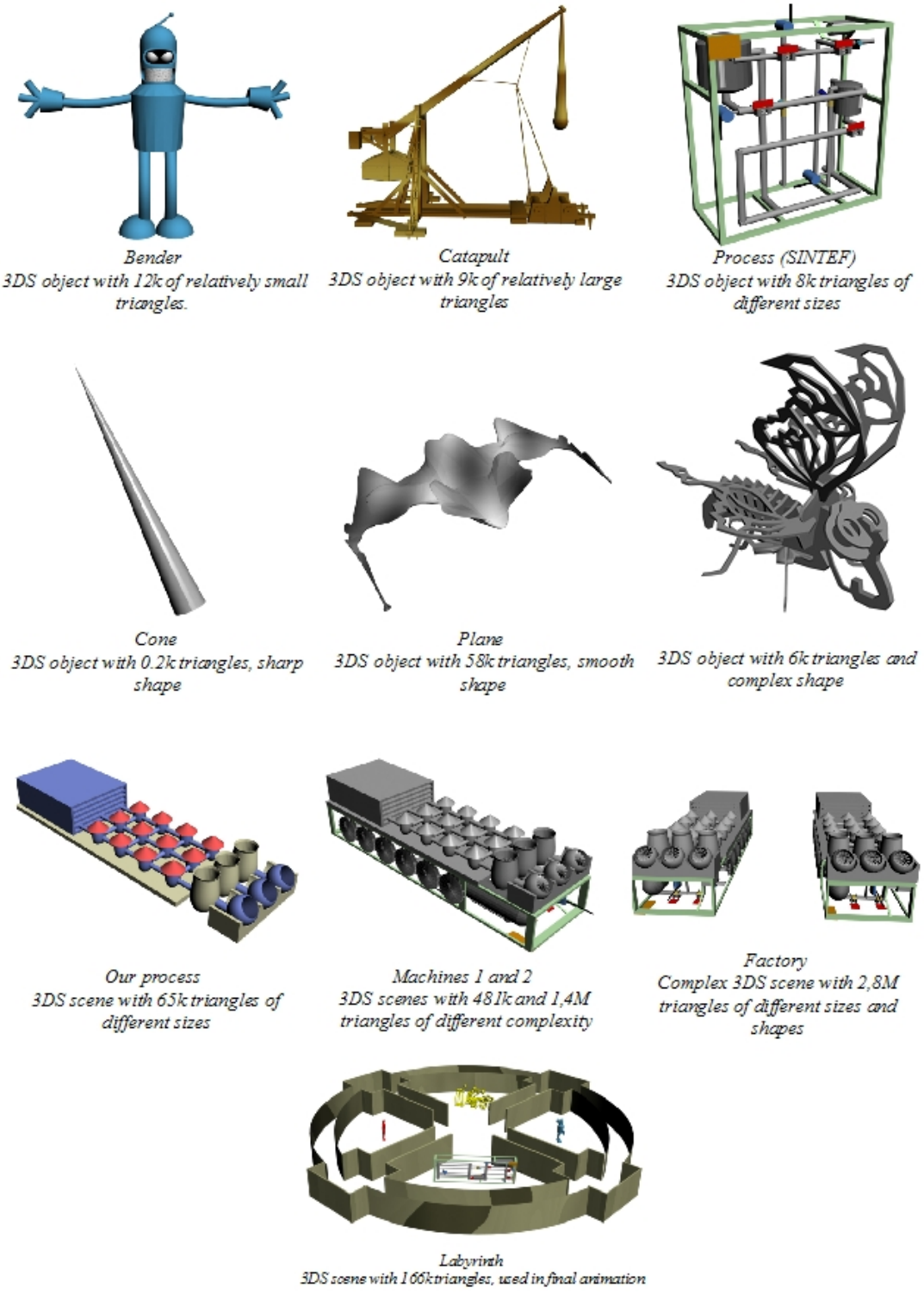

Figure 5.1: *Model 3DS objects that were used in algorithm tests*

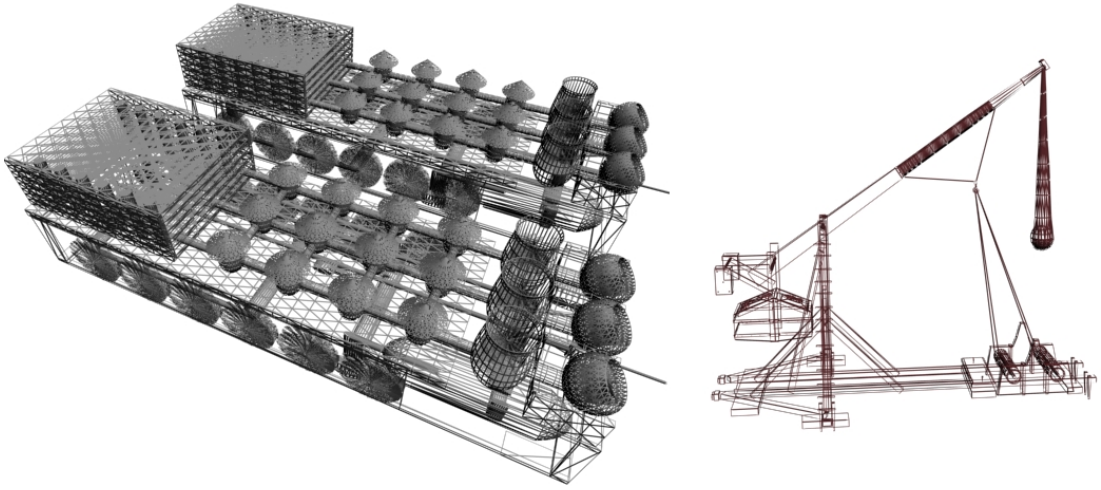

Figure 5.2: *Comparison of objects with small (left) and large (right) triangles - (a) Factory (2.8M faces). The distance field computation (414x406x413) took 0.64 secs (b) Catapult - (9.6K faces, 412x404x416, 0.90 secs).*

examined during an algorithm run points (Table 5.1). These data are gathered further in tables and give us an opportunity to extract more information.

Experiments show that for complex triangular meshes with predominantly small triangles the algorithm gives better results than for the meshes with large triangles. This statement illustrates Figure 5.2. We compare a mesh "Factory" that has 2,8M relatively small triangles and "Catapult" that has only 9k triangles, but they are rather large. Both objects have resolutions like 400x400x400. The incredible results that we see from comparing distance field computation times around these objects (0,64 sec for Factory in comparison to 0,90 for Catapult) prove our statement.

Runtime differs for different models. With a minimum distance field computation this is not a resolution that determines runtime, but rather a number of triangles and (more important that we have seen earlier) the size of triangles. The larger the size of a triangle, the more grid points its bounding box includes and thus more computations a device must perform. The worst case scenario is when all triangles are large and their bonding boxes contain almost all grid points (but not all such that the triangles will not be skipped by each other). In this case a time complexity of the algorithm increases to its maximum value that is *O(nm)* where *n* is the total number of triangles and *m* is the total number of grid points.

5.2.1 Experiments with different types of objects

To find the most optimal parameters of the objects for our algorithm we tested our evaluation material in different environments.

	Triangles	Skipped triangles	Resolution	Time, sec	Examined points
Process (SINTEF)	8 100	849	5x5x5[min]	0,00	58 008
			203x204x208	0,03	1 804 706
			402x400x403	0,12	6 079 154
Catapult	9 618	3 916	9x8x7[min]	0,00	51 670
			208x204x222	0,14	10 788 584
			412x404x416	0,90	70 235 291
Bender	12 034	384	12x10x6[min]	0,00	102 902
			218x210x207	0,04	2 589 580
			404x415x409	0,23	11 288 112
Plane	58 421	958	16x6x11[min]	0,00	486 327
			217x219x212	0,03	2 682 831
			408x405x414	0,07	7 279 068
Our process	65 356	6 709	10x7x25[min]	0,00	552 860
			217x208x205	0,06	8 794 401
			406x412x405	0,21	29 389 962
Machine 1	481 888	9 714	11x10x27[min]	0,04	3 998 838
		9 557	132x112x463[20]	0,15	12 087 377
		9 716	202x209x211	0,14	13 456 447
		9 715	419x408x417	0,37	35 323 779
Machine 2	1 444 768	42 694	11x10x27[min]	0,15	11 666 345
		41 696	131x112x463[20]	0,26	25 249 839
		42 696	201x209x211	0,29	28 166 209
		42 700	416x408x417	0,53	59 041 858
Factory	2 889 536	39 675	15x7x17[min]	0,28	23 511 890
		39 680	209x205x215	0,40	48 650 012
		39 679	414x406x413	0,64	89 116 249

[min] - actual resolution of a mesh , [20] - scaling in 20 times.

Table 5.1: *Duration of minimum distance field computation for triangular meshes on GeForce EN8800 GTS 512MB*

	Triangles	Skipped triangles	Resolution	Diff (Grid/Res)	Grid	Time, sec	Examined points
Process (SINTEF) (not scaled)	8 100	849	5x5x5	1^3	$5x5x5^{min}$	0,00	58 008
			5x5x5	20^3	100x100x100	0,09	103 455 366
			5x5x5	40^3	200x200x200	0,65	732 062 729
			5x5x5	60^3	300x300x300	2,25	2 578 104 812
Process (SINTEF) (scaled 30x30x30)	8 100	849	16x18x9	1^3	$16x18x9^{min}$	0,00	93 532
			16x18x9	$\approx 7^3$	100x100x100	0,03	8 062 816
			16x18x9	$\approx 14^3$	200x200x200	0,20	46 795 934
			16x18x9	$\approx 21^3$	300x300x300	0,60	140 797 081

Table 5.2: *Experiments with the process submitted for this diploma thesis by SINTEF IKT*

5.2.1.1 Process S

This process has a very small resolution and thus we tested it with different scaling factors to find whether it affects the runtime results.

From Table 5.2 we see that the most optimal set between resolution and grid is when resolution is equal to grid. Otherwise runtime is directly proportional to the difference between grid and resolution.

5.2.1.2 Our process

Here we tested a production equipment that is a kind of a big “Pulp and paper machine”. We tested the object with different triangle sizes (see Figure 5.3), resolutions and grid.

An optimized model of the process has 4 times lesser triangles that the normal one. It can look a little bit clumsy but we test it to find out if it gives much better results then the normal one.

From the test results of “Our process” (Table 5.3) we can see that our algorithm prefers small triangles rather then large with relatively large resolutions. In cases with small resolutions and minimum grids the situation is opposite: a small number of large triangles gives better runtime then a large number of small triangles.

If we look at grid sizes, we see that our minimum distance field is obviously gives best results, otherwise a large number of small triangles is better then the opposite.

5.2.2 Bottlenecks

Here we perform some bottleneck tests that will help us to find places in our algorithm that can be improved.

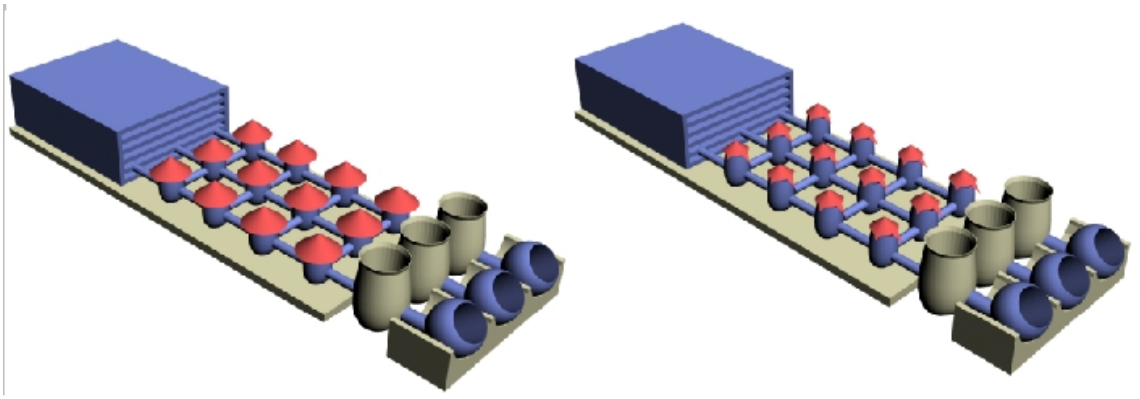

Figure 5.3: *Our process - (a) A model with 65k triangles (left), (b) An optimized model with 16k triangles (right)*

Triangles	Skipped triangles	Resolution	Grid	Time, sec	Examined points
65 356	6 709	10x7x25	10x7x25	0,00	552 860
		10x7x25	126x55x426	0,31	216 427 795
	6 533	126x55x426[20]	126x55x426	0,01	4 284 496
74 745 (small triangles)	6 353	14x21x26	14x21x26	0,01	637 776
		14x21x26	126x332x427	0,17	105 070 872
	6 299	126x332x427[20]	126x332x427	0,01	4 633 701
16 599 (large triangles)	2 792	14x11x28	14x11x28	0,00	144 180
		14x11x28	159x96x426	0,48	41 883 500
	2 800	159x96x426[20]	159x96x426	0,20	3 155 861

[20] – *scaling in 20 times*

Table 5.3: *Our process with different parameters*

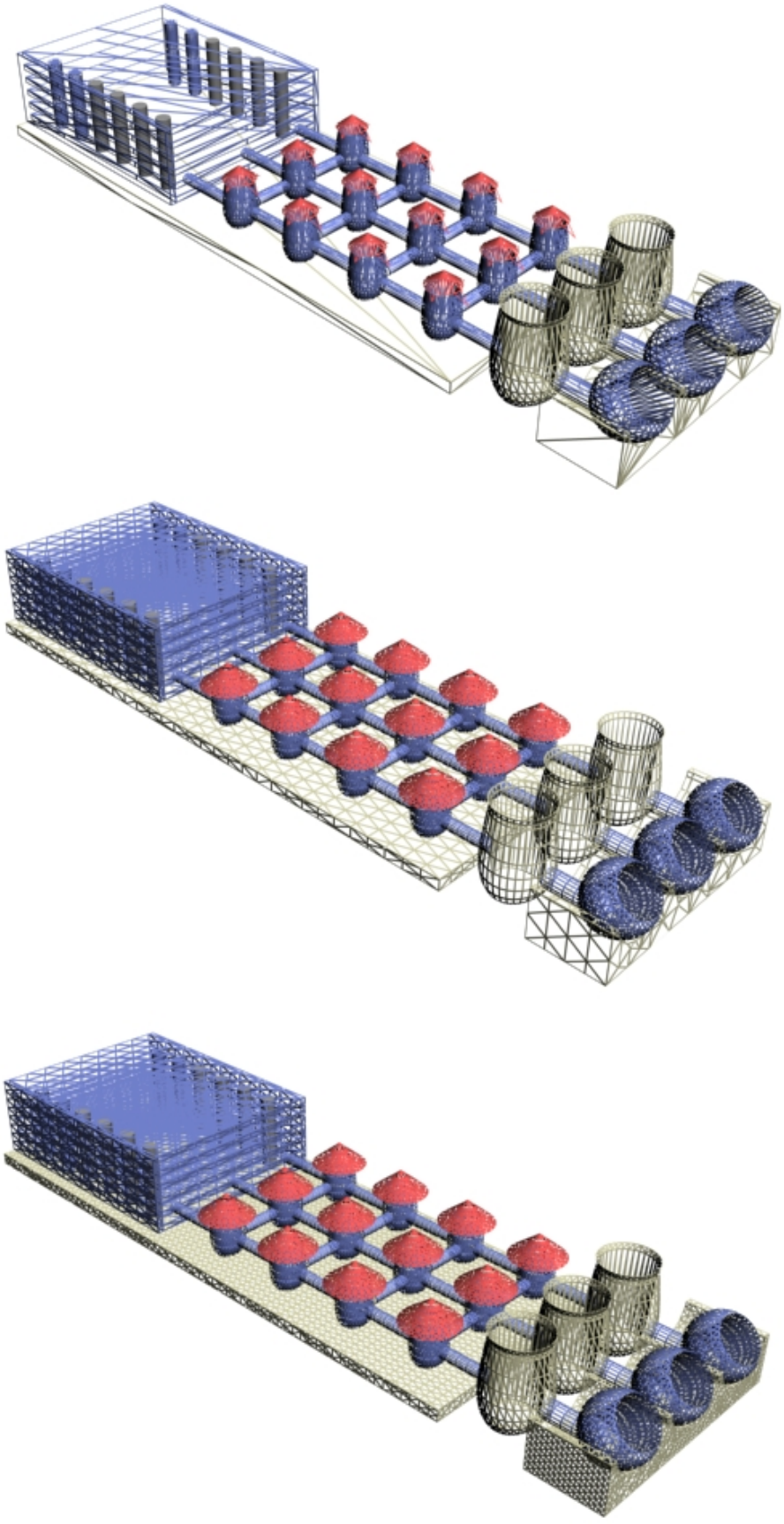

Figure 5.4: *Our process with large, normal and small triangles*

Part excluded	Runtime, sec	Weight, sec	Weight, %
-	**0,64**		
Kernel	0,18	0,46	71,8
The whole points-triangles distances calculation	0,18	0,46	71,8
Closest point-triangle test	0,48	0,16	25,0
Finding a squared distance	0,48	0,16	25,0
Calculating a position of a current point	0,53	0,11	17,1
Calculating a distance vector between a point and a triangle	0,50	0,14	21,9
Global memory read	0,61	0,03	4,7
Global memory write	0,18	0,46	71,8

^{min} *- actual resolution of a mesh*

Table 5.4: *Bottlenecks tests with the Factory model (2.8M triangles, 414x406x413 resolution)*

Recall from the Scott's first rule of bottlenecks: A bottleneck is a slowdown, not a stoppage. A stoppage is a failure. And “bottlenecks don't only exist under load” [23] .

Having all this in mind we test our largest model “Factory” in order to find these bottlenecks (see Table 6.4).

During this test we found a huge opportunity to improve our implementation. From the results we see that the main bottleneck in our implementation is a global memory write. After looking up the code we found that in our implementation every time after comparing distance values from a point to a triangle we wrote a new distance value to the memory whether or not the value should be updated. After correcting this part we succeeded in improving the performance in 2 times, and for some models in 3 times.

There is also an opportunity to improve a performance by improving a “closest point to triangle test”. The algorithm implemented here is described in section 4.2.

5.3 Collision detection algorithm

We validated our method for collision detection by integrating it into an animation “Fly in the labyrinth”. The animation scene includes a static labyrinth object (217k triangles) and a fly (6,5k triangles) that acts as a dynamic object. It is able to produce real-time animations (see Figure 6.5), but frame rates depend on different factors (we explored the number of grid points and the speed of the dynamic object).

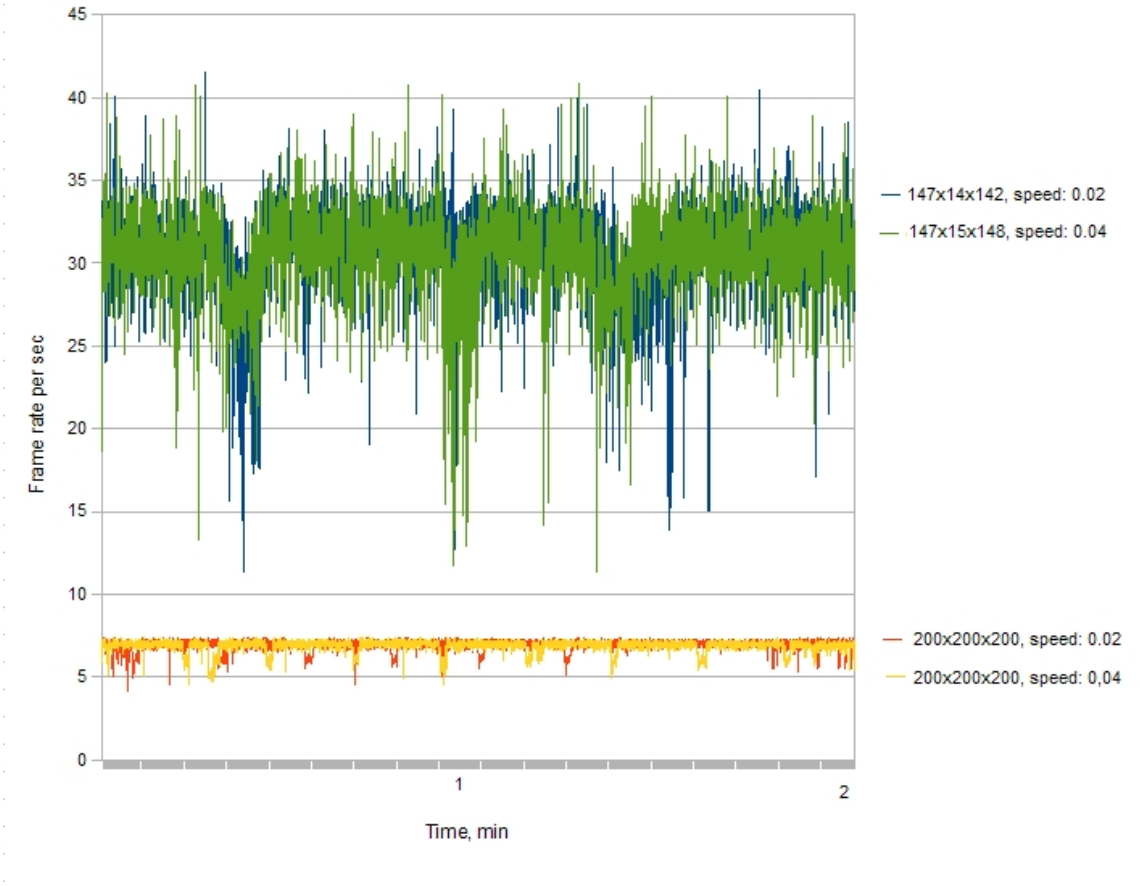

Figure 5.5: *Comparison of frame rates during a simulation run of "Fly in the labyrinth" (see Figure 6.6 and 6.7). When increasing a number of grid points, frame rates decreases significantly by a factor of 5.*

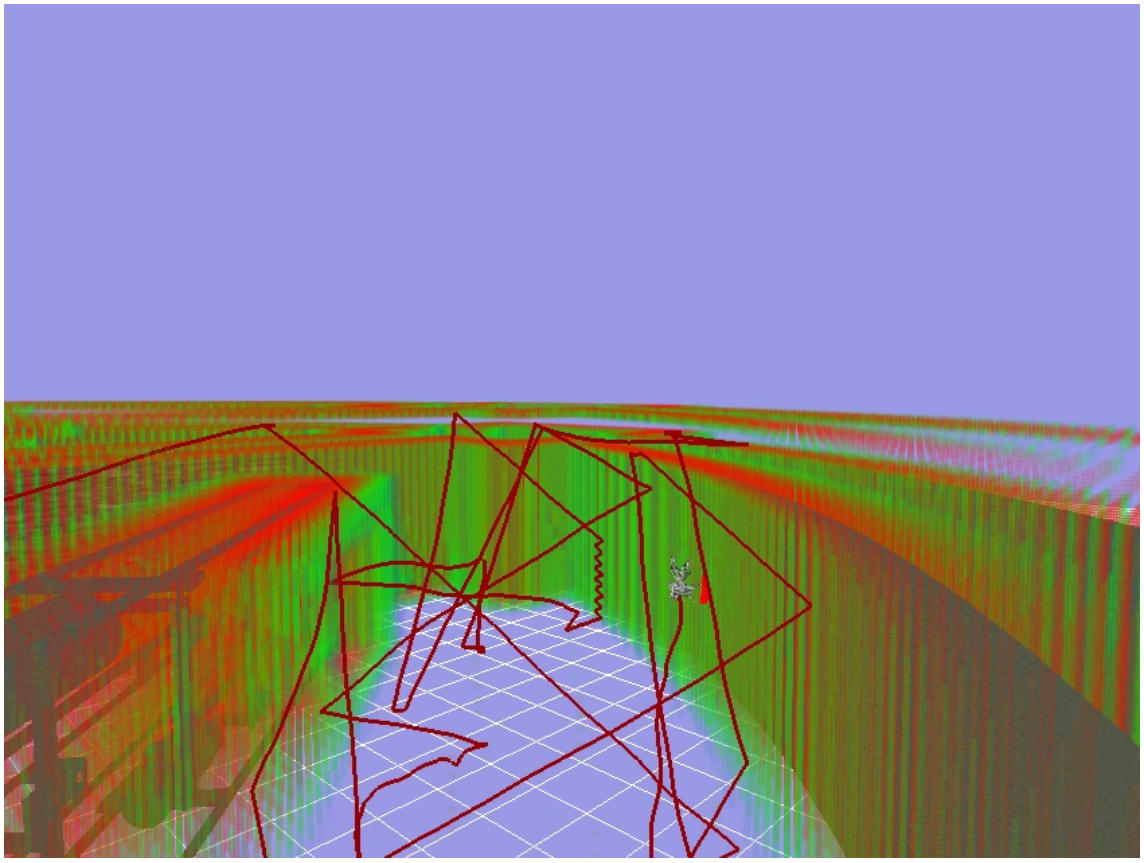

Figure 5.6: *Collision avoidance in the labyrinth*

We also found that there is a dependence between collision detection accuracy and a speed of dynamic objects. The lower the speed of dynamic objects, the higher the accuracy of collisions detections. More detailed, if we look at Figure 6.5, the same object ("Labyrinth") was tested in different environments, bun minimum grid sizes are different for speeds 0,02 and 0,04. We needed to correct a correcting value in minimum distance field computation formula (see Chapter 4.3) with a speed factor. Thats why in our experimental environment a minimum grid when a speed of the fly is 0,02 is 147x14x142 and a speed of 0,04 gives us a grid 147x15x148.

In order to demonstrate a precision of our collision avoidance algorithm, a path of the fly is shown in Figure 6.6 and 6.7. The fly does not follow any predefined path, but flies in random direction. If a collision occurs (one difi meets another difi), the fly just changes its direction into the opposite.

The implementation was carried out in C++ and uses OpenGL and CUDA for visualization. The system runs on dual Athlon 64 processor. All calculations of the animation run on the graphic card and are parallelized.

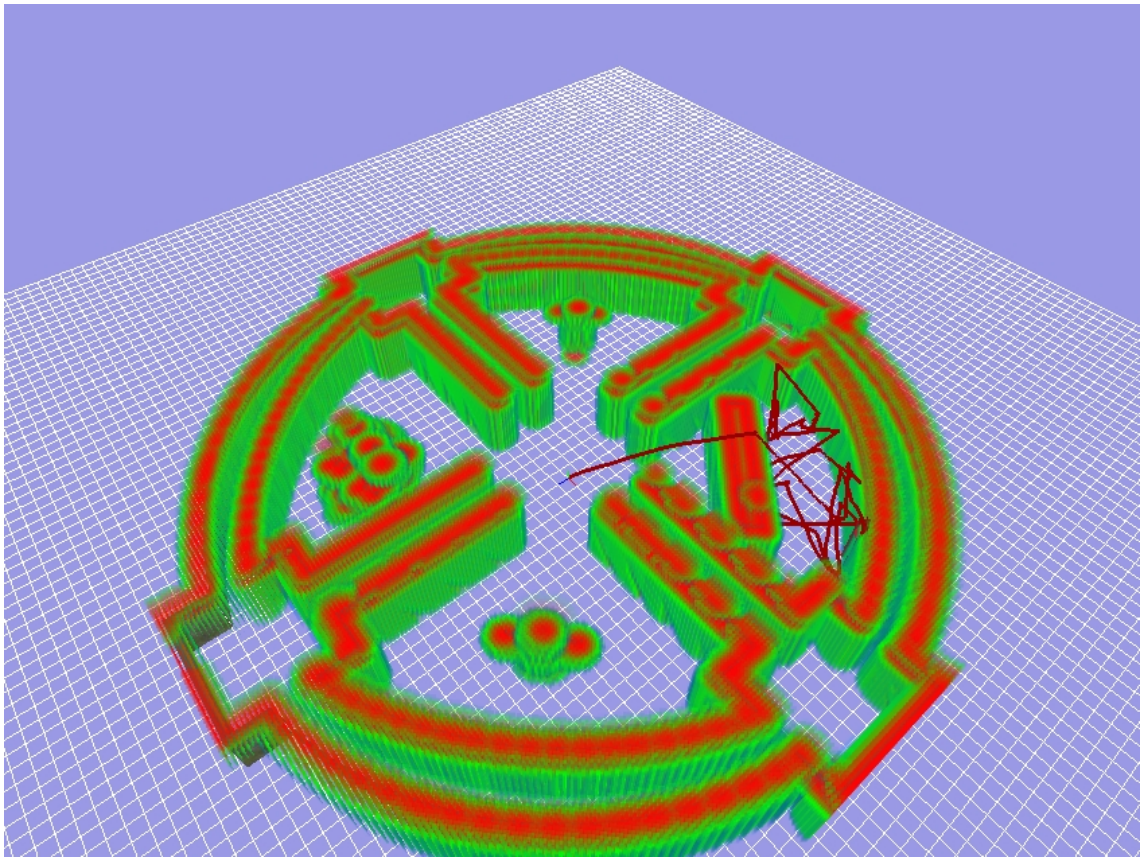

Figure 5.7: *Animation "Fly in the labyrinth" with distance field and path of fly showed*

	Triangles	Skipped triangles	Resolution	GPU time, sec	CPU time, sec	Examined points
Bender	12 034	384	$12x10x6^{min}$	0,00	0,73	102 902
Plane	58 421	958	$16x6x11^{min}$	0,00	2,44	486 327
Our process	65 356	6 709	$10x7x25^{min}$	0,00	2,58	552 860
Machine 1	481 888	9 714	$11x10x27^{min}$	0,04	17,53	3 998 838
Machine 2	1 444 768	42 694	$11x10x27^{min}$	0,15	50,84	11 666 345
Factory	2 889 536	39 675	$15x7x17^{min}$	0,28	101,72	23 511 890

^{min} *- actual resolution of a mesh*

Table 5.5: *Comparison of duration of minimum distance field computation for triangular meshes on GeForce EN8800 GTS 512MB and AMD Athlon 64 X2 Dual Core 3800+ .*

5.4 GPU implementation evaluation

To evaluate the results of utilizing a GPU, a device emulation had been used for reference that completely runs on CPU (see Table 5.5). A device emulation was also used to debug a GPU code. A specification of the system used is presented in table 2.1. Details of the GPU were discussed briefly in Chapter 3.

Runtime results from Table 5.5 shows us that even with emulation device CPU time increases rapidly as soon as the number of triangles and examined points increases and thus GPU with its parallelized calculations performs same tasks much more effective then CPU.

Of course, device emulation is not an effective comparison material. Running a highly parallelized GPU code on CPU with device emulation is never beneficial. This has to do with the nature of device emulation, and not the algorithm implementation. Each thread, which runs on the GPU, is also executed as a thread on CPU in serial. Most of the time the CPU starts and stops threads, and not executes the instructions.

With the design of the CPUs today, they will have a hard time beating the GPUs in creating a distance field with equal conditions. With the algorithm we presented in this thesis, the current design of the CPUs limits the performance because our algorithm is made with parallelism in mind. The CPU is only able to run 2-4 threads in parallel, while the GPU is able to run as many threads as there are processors in the device. Even if the clock speed of a CPU is higher than the clock speed of the GPU, the advantage of very low cost and massive true threading beats the CPU.

If the CPU were ever to beat the GPU, the algorithm must be designed to branch out most of the code paths and points in grid, which are not needed for the specific problem. This also implies that all special cases must be implemented, tested and optimized. Among these cases are the shape of the triangle and the orientation of the triangle. This in turn translates into very much work, which is error prone.

There is also a possibility of using a much better triangle skip algorithm, but then again, this algorithm could also be used to speed up the GPU implementation since this is a pre-distance field generation feature.

The conclusion is then, with the current CPU designs, the GPU is the best choice to solve such problems as distance field and collision avoidance computations. This not only relates to the speed of the algorithms, but also development costs. A GPU implementation is simple and easy to maintain, while a CPU algorithm could be very costly to develop and difficult to maintain.

5.5 Summary

Our distance field computation and collision detection algorithms and their implementation were evaluated with different tests. Main results can be summarized as follows:

- For complex triangular meshes with predominantly small triangles the algorithm gives better results than for the meshes with large triangles. Although in cases with small resolutions and minimum grids a small number of large triangles gives better runtime then a large number of small triangles.
- Runtime differs for different models. With a minimum distance field computation this is not a resolution that determines runtime, but rather the number and the size of triangles.
- The most optimal set between resolution of the object and its grid is when resolution is equal to grid. Otherwise runtime is directly proportional to the difference between grid and resolution.
- The lower the speed of dynamic objects, the higher the accuracy of collision detection algorithm.
- CPU time increases rapidly as soon as the number of triangles and examined points increases and thus GPU with its parallelized calculations was the ideal choice for our algorithm's implementation.

Chapter 6

Conclusion

We investigated the problem of rapid distance computation between rigid bodies and came with a solution based on distance fields. Distance fields had been frequently used in computer graphics, geometric modeling, robotics and scientific visualization. Here we use them for rapid distance computation and collision detection between large, complex objects. In this thesis we introduced two algorithms: distance field computation algorithm and based on it collision detection algorithm, that together propose a solution to our problem.

Our distance field computation algorithm has some very important new features that give it obvious advantages over previous works. First, the algorithm is specially made for parallel computing devices that accelerates the whole computation dozens of times and gives the opportunity to work with really large meshes. Furthermore, the meshes are preferred to be oriented, but it is not obligatory. The algorithm works fine with all kind of triangulated meshes, the only thing is that if the mesh is not oriented almost all triangles will be taken into computations and it will increase a time complexity of the algorithm to $O(nm)$ where n is the number of triangles in the mesh and m the number of grid points in bounding boxes.

A number of experiments were conducted to evaluate the algorithms and their implementation. Experiments showed some times really amazing results. We could run collision avoidance simulations in real-time with meshes containing several millions of triangles.

Thus the main purpose of this thesis was achieved.

Chapter 7

Discussions and future work

In this paper we proposed a new algorithm for distance field computation. The main difference from that proposed by for example Fuhrman et al. is that our algorithm proposes also to skip some triangles in the mesh from calculations during initialization and thus accelerates runtime.

One must also note that our triangle skip algorithm is not the definitive way of skipping triangles. There is also possible to analyze the triangles in different ways to eliminate unneeded triangles for a certain problem. We have merely created the framework to enable the possibility for triangle elimination, if desirable.

There are a number of improvements that can be made to our algorithm. The ones we find most likely to lead to something are listed here:

- closest point to triangle algorithm. We implemented a geometric solution of this problem proposed by Christer Ericson [4] . This is not the most effective algorithm and it can be improved to speed up a runtime.
- triangle skip algorithm. This can be sufficiently improved by creating a method of comparing one triangle with all the others. What we do now is to compare bounding box of one triangle with the bounding boxes of previous and next triangles. We must note that one must be very careful when skipping triangles. We do not recommend to do it on meshes with large triangles since significant errors can occur and reduce accuracy of the collision detection algorithm.
- as the experiments show, the main bottleneck of our distance field algorithm is a memory write procedure. We tried to reduce the number of memory write calls but maybe this is possible to restructure the algorithm's implementation and use even smaller number of these expensive calls.

A very important future work could be an implementation of other distance field computation algorithms and testing them all together on same hardware. We can not exactly say now how good our algorithm is in comparison to other algorithms developed recently mainly because of the differences in the hardware.

Terms

ADF	–	Adaptively Sampled Distance Field
API	–	Application Programming Interface. All functions starting with cuda* are a part of the CUDA API.
CUDA	–	Compute Unified Device Architecture.
Cutil	–	CUDA Utility. A set of macros simplifying CUDA programming.
device	–	The graphics device with it's own memory.
DRAM	–	Dynamic Random Access Memory.
emulation	–	Device emulation enables the developer to debug the kernel code.
GLSL	–	GL slang / OpenGL Shading Language.
GPGPU	–	General-purpose computing on graphics processing units
GPU	–	Graphics Processing Unit. The chip with all the “magic”.
host	–	The host of the graphics device, aka where the CPU, mainboard and RAM is located.
kernel	–	A kernel contains the code the GPU runs in parallel on the device.
nvcc	–	NVIDIA CUDA Compiler.
SIMD	–	Single Instruction, Multiple Data
SDK	–	Software Development Kit. The CUDA SDK contains nvcc, docs, cutil and examples.
VBO	–	Vertex Buffer Object. A general data storage stored on the device memory.
VCVDT	–	Vector-city Vector Distance Transform
EVDT	–	Efficient Vector Distance Transform

Bibliography

References

[1] A.Fuhrmann, G..Sobotfka, and C.Grob. Distance fields for rapid collisions detection in physically based modeling. Moscow, Russia, 2003.

[2] D. Breen, S. Mauch, and R. Whitaker. 3d scan conversion of csg models into distance volumes. In *Proc. 1998 IEEE Symposium on Volume Visualization*, pages 7–14, 1998.

[3] M.Gross C.Sigg, R.Peikert. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization*, 2003.

[4] C. Ericson. *Real-Time Collision Detection*. Elsevier Inc, 2005.

[5] A. Kaufman F. Dachille. Incremental triangle voxelization. In *Proceedings of Graphics Interface 2000*, pages 205–212, 2000.

[6] André Guéziec. "meshsweeper": Dynamic point-to-polygonal mesh distance and applications. In *IEEE Transactions on Visualization and Computer Graphics*, pages 47–60, 2001.

[7] J.A.Bærentzen. Generating signed distance fields from triangle meshes. Technical report, IMM, 2002.

[8] R. Crawfis Shao-Chiung Lu Jian Huang, Yan Li and Shuh-Yuan Liou. A complete distance field representation. In *Visualization, 2001. VIS '01*, pages 247–254, 2001.

[9] Mark W. Jones. *The Production of Volume Data from Triangular Meshes Using Voxelisation*, pages 311–318. Computer Graphics Forum, 1996.

[10] J.Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, Cambridge, UK, 1999.

[11] M.Overmars J.Vleugels. Approximating voronoi diagrams of convex sites in any dimension. *International Journal of Computational Geometry and Applications 8*, 1997.

[12] J.Keyser M.Lin D.Manocha K.Hoff, T.Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of ACM SIGGRAPH*, pages 277–286, 1999.

[13] Sean Mauch. A fast algorithm for computing the closest point and distance transform. Technical report, Applied and Computational Mathematics, California Institute of Technology, 2000.

[14] NVIDIA. *GeForce 8800 GPU Architecture Overview*, 2006.

[15] NVIDIA. *CUDA Programming Guide 1.0*, 2007.

[16] NVIDIA. *GeForce 8800 spesifications*, 2007.

[17] NVIDIA CORPORATIONG. *CUDA Programming Manual BETA 2*, June 2008.

[18] B. Payne and A. Toga. Distance field manipulation of surface models. In *IEEE Computer Graphics and Applications*, pages 65–71, 1992.

[19] S.Frisken R.Perry. *Kizamu: A system for sculpting digital characters*, pages 47–56. Proc. of ACM SIGGRAPH, 2001.

[20] M.W.Jones R.Satherley. *Hybrid distance field computation*. Volume Graphics 2001, 2001.

[21] R.Yagel F.Cornhill R.Shekhar, E.Fayyad. Octree-based decimation of marching cubes surfaces. In *Proc. of IEEE Visualization*, pages 335–342, 1996.

[22] Alyn P. Rockwood Thouis R. Jones Sarah F. Frisken, Ronald N. Perry. *Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics*, pages 249–254. TR2000-15, 2000.

[23] S.Barber. Beyond performance testing part 7: Identifying the critical failure or bottleneck. *http://www.ibm.com/developerworks/rational/library/4259.html*, 2004.

[24] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha. Difi: Fast 3d distance field computation using graphics hardware. Department of Computer Science, Univeristy of North Carolina, Chapel Hill, NC, USA, 2004.

[25] TURBOSQUID. 3d models. http://turbosquid.com/, 2008.

[26] Suresh Venkatasubramanian. *The graphics card as a stream computer*. Computer Science Department, Florida State Unoversity, 2005.

[27] Wikipedia. Cuda. *http://en.wikipedia.org/wiki/CUDA*, july 2008.