Distance Fields Accelerated with OpenCL

Erik Sundholm

June 8, 2010
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Eddie Wadbro
Examiner: Fredrik Georgsson

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

An important task in any graphical simulation is the collision detection between the objects in the simulation. It is desirable to have a good general method for collision detection with high performance. This thesis describes an implementation of a collision detection method that uses distance fields to detect collisions. This method is quite robust and able to detect collisions between most possible shapes. It is also capable of computing contact data for collisions.

A problem with distance fields is that the performance cost for making a distance field is quite extensive. It is therefore customary to have some way of accelerating the computation of the distance field (usually by only computing select parts of the field). The application implemented in this thesis solves this performance problem by using the parallel framework OpenCL for accelerating the construction of the field.

OpenCL enables programmers to execute code on the GPU. The GPU is highly data parallel and a huge increase in performance can be obtained by letting the GPU handle the computations associated with the initiation of the field.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The ability to detect collisions between graphical objects is an important part of any graphical physical simulation. It is desirable to have collision detection that detects every collision that occurs and a collision response that makes the interacting objects behave as close to reality as possible. A problem with collision detection is that it often constitutes to a bottleneck on the simulation's performance when the amount of objects that needs to be taken into consideration is large. It is therefore desirable to offload the CPU of the burden of collision detection as much as possible.

A way to achieve this is to utilize the GPU and let the GPU handle the computations related to the collision detection. This is possible with the new generation of GPUs where a GPU can be programmed to perform other tasks than graphics related ones. A way to program the GPU is to use the OpenCL (Open Computing Language) framework [1]. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors [2]. OpenCL creates a level of abstraction above the hardware details of the processors and allows the programmer to view a processor as a general computing unit that can be programmed using OpenCL code.

Collision detection is usually achieved by abstracting an object's shape as a collection of common geometrical objects (like spheres, cubes or cylinders) and then checking these geometric objects for collisions. This is usually sufficient, but there are situations where this method gives a bad estimation of the object's shape. It is therefore preferable in some cases to use a more general method for detecting collisions which can relate the shapes of more oddly formed objects.

One such general method is to use distance fields to detect collisions. A distance field for an object divides space around the object into a 3 dimensional grid where each cell contains the closest distance to the object from the cell. The cells at the surface of the object have distance values that are around zero which provides a contour of the object which can be used to detect collisions.

The task of this Master Thesis consists of implementing a general collision detection method which employs distance fields and uses OpenCL to run tasks related to the method on the GPU when it is prudent.

The project is carried out at Algoryx Simulations AB, a company specializing in accurate physics simulations for the professional market. The company specializes in industrial simulations and Algoryx's main product is AgX [3], a physical toolkit. AgX is used in simulators for areas like vehicles and off-shore ships. Algoryx also develops physical simulators to be used in education and has developed Algodoo [4], a 2D physical sandbox.

This report consists of 4 parts:

- The first part describes the purpose and the goals of the project and provides a thorough problem description.
- The second part provides an in-depth study of General-Purpose Computing on Graphics Processing Units (GPGPU) and gives a case study of the OpenCL framework, the targeted platform for the implemented software.
- The third part, which consists of chapters 4–5 provides the theory behind the methods and data structures that are used in the implemented application. Chapter 4 describes the theory behind distance fields and describes their properties and the different methods used for creating them. Chapter 5 describes how distance fields can be used to perform collision detection for graphical objects.
- The fourth part consists of chapters 6–8 and describes the results of project. It gives an description of the implementation and its performance and provides an analysis of the project.

Chapter 2

Problem Description

2.1 Problem Statement

The problem to be solved in the project consisted of implementing a program capable of generating distance fields for graphical models. It also consisted of implementing collision detection functions that by using the distance field generated for a model would be able to detect collisions between the model and other types of graphical objects (like other models with distance fields, planes and geometric primitives). At least some parts of either the generation of the distance field and/or the use of the distance field to detect collisions with other objects was to be implemented with OpenCL.

An additional task of the project was to implement a demonstrator application which were to run a graphical simulation containing models of different shapes and sizes with all of the models having their own distance fields. These models were supposed to interact with each other and collide with the distance fields being used to detect the collisions between the models. This would demonstrate that the generation of the distance fields and their use to detect collisions was working properly.

The focus of the project was to be put on implementing distance fields and distance field based collision detection. Less weight was to be put on implementing the demonstrator application and the complexity of the simulation that the application was to run would be dependent on how long the implementation of the actual distance field method would take.

2.2 Goals

The main goals of the project consisted of implementing distance fields, distance field based collision detection (using OpenCL for increased performance) and constructing a graphical simulation which uses the implemented distance fields for detecting collisions between objects in the simulation.

2.3 Purposes

A general collision detection method that can handle any type of shape is useful when one is dealing with complex graphical objects. One typically wants to handle collisions in a graphical simulation by using primitive tests because of the efficiency of such tests. A primitive is a elementary geometry like a sphere, box or cylinder whose shape is well

know. A collision test between two primitives can be made very efficient because everything about the primitives shapes is already known and it is possible to use this information to analytically calculate the contacts between the primitives. A model is represented in the collision detection as a set of primitives which works as an abstraction of the shape of the model. This works well in many cases, but there are times when it is not appropriate to abstract a model as a set of primitives and when such an abstraction gives a poor performance. In these kinds of situations can a general collision detection method (such as using distance fields to detect collisions) be useful. A distance field based collision test does not produce as good performance as a primitive based collision test on simple models, but can in contrast handle any type of model (provided it does not have any holes) and can more accurately handle complex models. This makes a distance field based collision detection method a useful tool for a physics engine.

Distance fields can also be used for many other tasks than collision detection. It can be used to implement deformable objects, sculpting, model simplification, remeshing, path planning and for many other applications. A more thorough description of the different applications of distance fields is given by Sud, Otaduy and Manocha in [5].

2.4 Tools

Several tools will be used in the implementation of this project:

- C++ will be used for the implementation of the host code
- OpenCL will be used for the implementation of the kernel code
- Agx (Algoryx's physics engine) will used for the implementation of the simulation where the distance fields will be used
- LaTex will be used for writing the report
- Visual Studio 2008 will be the editor used for writing the C++ and OpenCL code
- TeXworks and gEdit will be the editors used for writing the report

2.5 Project Outline

The process of implementing the project involves several steps:

First must a solid understanding of distance fields and collision detection be obtained. Knowledge about distance field will be acquired by reading the different papers about distance fields that have been published. Knowledge about collision detection will be acquired both by studying literature about the subject and by studying the collision detection software implemented by Algoryx. After an understanding of the subject matter has been obtained should time be spent on looking up appropriate algorithms for preforming the different tasks related to the application such as the distance test, the sign computation, collision tests etc.

Later should a solid understanding of OpenCL be acquired so that written OpenCL routines will be efficient and so that the application can be designed with parallelism in mind so the host application will work well with the GPU code. An understanding of OpenCL will be obtained by studying the literature about OpenCL and by writing sample applications.

The sample applications will be very simple in the beginning, but will continually grow more complex and more close to what will actually be implemented in the project.

After an understanding of the OpenCL framework and GPGPU has been obtained should a detailed design of the application be made. The data types that will be needed by the application will then be specified and the functions needed by the application will be thoroughly planned.

After the design has been completed will the actual implementation of the project begin. The application will first be implemented solely on the CPU to make it easier to debug errors in the used algorithms and prevent errors in the kernel code from being mistaken as flaws in the method. This is also done to later have a reference implementation to compare the GPU implementation against. The application's algorithms will be thoroughly tested on the CPU and verified as being sound before any part of the implementation will be transfered to the GPU. The CPU implementation of the application will have data that is exportable to the GPU by using datastructures that OpenCL kernel code recognizes. This will make it possible to construct a GPU implementation of any of the application's functions without it affecting any other parts of the application. The cost of starting up an OpenCL function (kernel) and transferring data to the GPU is somewhat large and an application does not receive any real benefits from running a function on the GPU unless the amount of calculations that the function needs to have preformed is quite substantial. It is therefore appropriate to keep certain functions on the CPU. The decision whether to make a GPU implementation of a function will depend on the performance increase that can be achieved by porting the function to the GPU and how complicated it would be to implement the function on the GPU.

The tasks that will be prioritized to implement on the GPU will be the computation of the distance values of the cells of the distance field's grid and collision detection between distance fields and other distance fields. This is because these tasks would receive the greatest potential speed-up from a GPU implementation.

2.6 Related Work

There have been many works published about the generation of distance fields and the concept of distance fields has been around for quite some time. One of the first mentions of distance fields in scientific literature was in 1966 in a paper by Rosenfeld and Pfaltz about image processing [6]. The use of distance fields in computer graphics did however not come around until the 80s.

A recent paper done about distance field generation is the paper [5] written by Avneesh Sud, Miguel A. Otaduy and Dinesh Manocha where a fast algorithm (DiFi) for computing a distance field along a 3D grid is presented. The algorithm divides the grid into 2D slices and uses a scheme of culling and clamping to calculate distance values for as few cells as possible per slice. The algorithm also tries to minimize the amount of calculated distance values by enclosing each triangle in the mesh with bounding volumes and only calculating distance values for cells inside bounding volumes. The algorithm uses the GPU to obtain a speed-up and does so by using the OpenGL API and rendering the distance function and manipulating the result by using OpenGL functions.

J. Andreas Baerentzen and Henrik Aanaes presented a algorithm in [7] for producing signed distance fields which employs a novel method for calculating the sign (whether a cell is inside or outside the model) of the grid cells distance values which uses angle-weighted pseudo normals. This method for calculating the sign is easily integrable into the distance calculation which gives the advantage of being able to calculate a distance value and its

corresponding sign in one pass. This advantage combined with the good quality of this sign test has lead to this method becoming well-used in distance field generation and referenced in many other technical papers about distance fields.

Kenny Erleben and Henrik Dohlmann presented a distance field algorithm at the third GPU gems conference [8] which employs many techniques presented previously in other distance field papers. The algorithm is GPU accelerated and uses an approach similar to that used in the algorithm presented by Sud et all in [5] with a division of the grid into 2D slices and an enclosing of the mesh's triangles with bounding volumes. It improves on this approach by fixing the leaking artefacts that sometimes occur when this method is used where some cells in the grid receive the wrong sign. This makes their method able to handle "inconsistent" meshes, that are polygonal models which can have holes, flipped surfaces and overlapping triangle faces. They managed to accomplish this by using the angle-weighted pseudo normals presented by J. Andreas Baerentzen and Henrik Aanaes in [7]. Their method also does not require a scan line which most other algorithms that divide the grid into 2D slices do.

Kenny Erleben also explored the use of distance fields in collision detection in his thesis project [9] from 2004 where he made a survey of several different collision detection methods. He choose to focus on the use of distance fields in collision detection and didn't expand on how the distance fields themselves should be created.

Chapter 3

General-Purpose Computing on Graphics Processing Units

In this section is the concept of General-Purpose Computing on Graphics Processing Units (GPGPU) introduced and the reasons behind its use is explained. A case study about OpenCL is also given.

3.1 Overview

Graphics hardware has seen a massive development in the last few years and the computing power of the GPU is continually improving. The GPU's floating-point computational power exceeded that of the CPU several years ago and the performance gap between them is still steadily growing (see Figure 3.1). The reason behind the GPU's vast computational power is the GPU's relatively simple architecture and data processing which enables data parallel processing. The tasks (like transforming vertices and calculating lighting for pixels) that the GPU have been employed for traditionally have been naturally data parallel with one piece of data having little or no dependence with any other piece of data. This lack of data dependence has made it very easy to parallelize the tasks preformed on the GPU since threads can be started independently of each other. As a consequence is it simple to increase the performance of the GPU since a higher performance can be obtained simply by increasing the amount of threads that can be run concurrently and this can be achieved by adding more cores to the GPU.

The GPU has also surpassed the CPU in terms of memory bandwidth (see Figure 3.2). A typical GPU can achieve a bandwidth of around 100 GB/s while a CPU only can achieve a bandwidth of around 20 GB/s. This bigger memory bandwidth has been made possible with the GPU's more relaxed memory model and that the GPU does not have to satisfy requirements from legacy operating systems like the CPU has to.

A factor that limits the GPU's performance is the GPU's memory latency. Even though the GPU's computational ability and memory bandwidth have improved by leaps and bounds over the last few years have memory latencies not seen an improvement of the same magnitude. The computational power of the GPU increases with about 70 % each year and the memory bandwidth with about 25 % while memory latencies in comparison are only improving with about 5 % each year [10]. This makes it essential to minimize the amount of accesses made to the GPU's memory and maximize the amount of work that gets

performed on loaded memory to fully utilize the potential of the GPU.

The CPU is in contrast to the GPU not capable of the same amount of parallelism as the GPU. This is because the CPU has to be more general and perform a wider variety of tasks than the GPU. The tasks that the CPU has to perform, unlike the tasks the GPU is designed to perform are not always data parallel and may not be parallizable in nature. This variety of tasks limits the magnitude the CPU can be parallelized since it can not make any assumptions about its upcoming tasks. The CPU's increased generality comes with the price of a more complex hardware architecture with more hardware dedicated to control. The CPU also has larger areas dedicated to memory caches than the GPU. The CPU has as a consequence less space available for computational units such as ALUs (Arithmetic Logic Units) and FPUs (Floating-point Units) compared to the GPU where most of the transistors consists of computational units. [11] The differences between the CPU's and GPU's hardware architectures can be seen in Figure 3.3.

These differences in hardware architecture makes the CPU and GPU suited for different kinds of problems:

- A CPU excels at complex problems that needs to be processed sequentially, where the control flow is complex and contains many branches and only a small amount of data needs to be processed.

- A GPU works best with problems involving a lot of data where a lot of floating-point computations have to be preformed and where the processing of one piece of data is independent of the rest of the data. Problems given to the GPU should ideally have a simple control flow with as few branches as possible and use as few registers as possible (since registers can become sparse since every thread started by the GPU needs it own set of registers).

The potential of the GPU for solving data and computational heavy problems is what gave rise to GPGPU. GPGPU is the process of using the GPU for calculations unrelated to the rendering of graphics.

GPGPU began with the introduction of programmable shaders in 2001. GPGPU was then performed by using shader programs written in some shader language like GLSL or HLSL. A shader program is used to program the functionality of the rendering pipeline and has traditionally been used by graphical programmers to create visual effects not available with the default graphics pipeline. Non graphics related calculations with shaders are performed by first storing the data to be processed in textures or any of the graphics cards buffers (for example the z-buffer) and then processing the data with a fragment program. In the fragment program all of the wanted calculations are done on the data and the result is stored either as textures or in graphical buffers. The data is later extracted from the textures or buffers it was stored in and transformed to the format wanted by the user.

The need to use shaders to perform general purpose calculations was quite limiting for programmers and programming these shaders were not intuitive. It forced programmers to learn how to use a graphical API (like OpenGL or Direct3D) and learn shader programming and forced programmers use GPU related datatypes (like textures) to store data. These drawbacks gave rise to GPGPU frameworks which were intended to give programmers a more intuitive way to program the GPU to perform non graphics related computations and free the programmer from using graphics programming concepts. These frameworks usually provides the programmers with a C-like interface with similar datatypes and control flow statements (for-loops, if-statements, switch-statements etc) with the addition of vector type versions of the common C types like *int*, *float* and *double*.

One of the earliest of these frameworks was ATI's Close To Metal (CTM) that was introduced in 2006. [15] It provided programmers with a general interface for programming ATI GPUs. CTM was short lived and only got to the beta stages. It was later succeeded by Stream SDK as ATIs GPGPU framework. Nvidia's response to CTM was CUDA that was released to the public in febuary 2007 [14] and provided programmers with a corresponding interface for programming NVIDIA graphics devices. Other released GPGPU frameworks includes BrookGPU developed by Stanford University and DirectCompute developed by Microsoft.

A problem with most of the early GPGPU frameworks was that they required the user to use a GPU made by the company providing the framework. This gave rise to the motivation to have a unified framework independent of underlying hardware structure. This motivation is what gave rise to OpenCL. OpenCL will be discussed in detail in the next section.

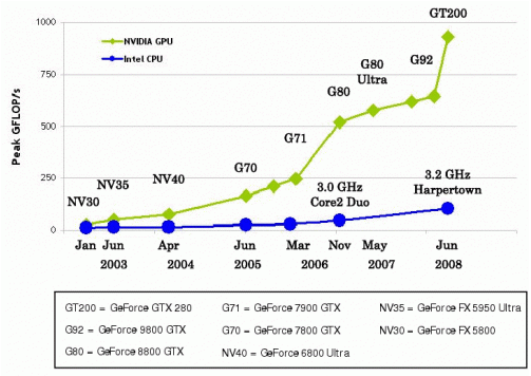


Figure 3.1: A graph of the number of floating point operations per second for CPUs and GPUs the last few years [12].

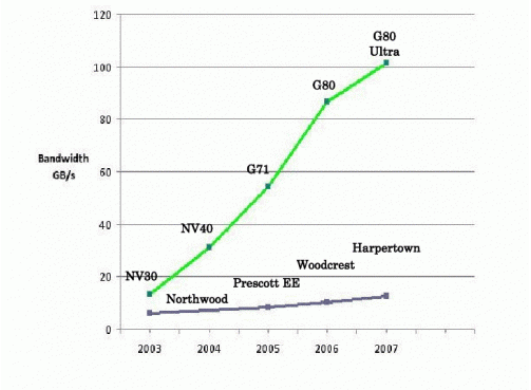


Figure 3.2: A graph of the memory bandwidth for CPUs and GPUs the last few years [12].

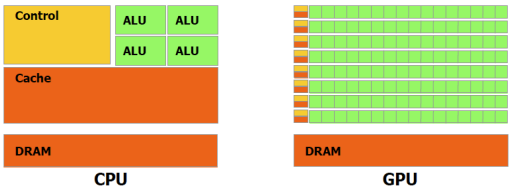


Figure 3.3: A picture displaying the hardware architecture of the CPU and the GPU [12].

3.2 OpenCL

3.2.1 Introduction

OpenCL was developed by Apple (in collaboration with technical teams at AMD, IBM, Intel and Nvidia) who wanted to have a means to exploit the computational power of the GPU for their Snow Leopard platform. Apple chose to hand over the rights and managing of OpenCL to the Khronos group after the initial proposal of the framework was completed and make it a open and royalty-free standard. The Khronos group maintains several other open standards such as OpenGL, OpenKODE and Collada [13]. OpenCL's first stable version was released in December 2008, but drivers were not provided by the major graphics card developers to the general public until several months later (26th November 2009 for NVIDIA and 21th December 2009 for ATI) [2]. Beta drivers were available earlier, but only to some specially selected developers. OpenCL is very similar to CUDA, but can unlike CUDA be used with any GPU from the last few years and not just GPUs made by Nvidia. OpenCL code can also be run on the CPU, but the focus of the framework is on using the GPU. OpenCL also gives developers the ability of running calculations on a variety of devices (which might be GPUs or CPUs or both) simultaneously.

3.2.2 Platform Model

The OpenCL specification defines a platform on which OpenCL programs are run. The platform consists of a *host* (most often the user's computer) that is connected to one or more OpenCL devices. A OpenCL application is run on a subset of the total number of devices residing on the host and the application divides the operations it needs performed between the devices allocated to the application.

A *device* is a processor capable of performing floating point calculations and is typically either a GPU or a CPU, but can also possibly be a NPU (Network Processing Unit) [16].

A device is in turn composed of one or more compute units. A *compute unit* can for example be a CPU-core or a Streaming Multiprocessor (SM)/SIMD engine in a GPU.

A compute unit is composed of one or more processing elements. A *processing element* is a virtual scalar processor and can for example be a ALU or a streaming processor. It is in these processing elements that the calculations issued by a OpenCL application are performed.

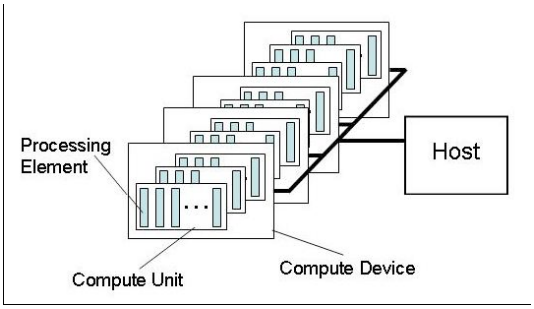


Figure 3.4: OpenCLs platform model [17].

3.2.3 Execution Model

An OpenCL application consists of two parts: a host application that executes on the host machine and kernel programs that executes on the devices allocated to the host application.

Host Application

The host application defines the kernels context and manages their execution. The host application's context consists of the following parts:

- **Devices:** A collection of devices allocated from the devices connected to the host.
- **Kernels:** The OpenCL functions that are to be run on the devices.
- **Program Objects:** The source and executables that implement the kernel functions.
- **Memory Objects:** A number of memory objects visible to the host and the devices allocated to the application. The memory objects contains values that instances of the kernels running on the devices are able to manipulate.

The context is created and later manipulated by the host application by using functions contained in the OpenCL API.

The host application uses a control queue to control the execution of the kernel functions on the devices (contained in the context). It is possible to have multiple command queues for a single context, but a context typically only has one. All queues associated with a context run concurrently and independently of each other. It exists no explicit mechanism in OpenCL for synchronization between command queues.

The host application places commands in the command queue which are scheduled onto the devices in the context. There are three kinds of commands that can be sent to a device:

- **Kernel execution command:** A command to execute a specific kernel function.
- **Memory command:** A command to transfer data to, from or between the memory objects stored in the host's or the context's devices memory or map/unmap memory from the host's address space
- **Synchronization command:** A command to constrain the order of execution of commands.

Commands passed to a device execute asynchronously in relation to the host. Kernel execution and memory commands do however generate event objects which can be used to coordinate execution between a device and the host. Events can also be used to control the order of the execution of commands. If the programmer does not explicitly change the ordering of commands via events are the commands ordered in relation to each other according to two different modes:

- **In-order Execution:** Commands are executed in the order they appear in the command queue and complete in the order they appear in the queue. This means that the application waits for the previous command to complete before issuing the next one.
- **Out-of-order Execution:** Commands are executed in the order they appear in the command queue, but the application does not wait for a command to complete before issuing the next command in the queue. Any synchronization between commands has to be enforced by the programmer through the use of synchronization commands.

Kernel Programs

A kernel program is a collection of kernel functions which get executed on the devices of a context. A kernel function is a parallel function which executes by running instances of itself on the device's processing elements.

There are two categories of kernel functions:

- **OpenCL kernels:** OpenCL kernels are written in the OpenCL C programming language and compiled with an OpenCL compiler. Some implementations may also provide other means of creating OpenCL kernels. OpenCL kernels are supported by all implementations of OpenCL and are the type of kernel programmers will most likely deal with.
- **Native Kernels:** Native kernels are accessed through a host function pointer. They share memory objects with OpenCL kernels and are queued for execution along with OpenCL kernels on devices. A native kernel function can be a function defined in application code or a function imported from a library. The ability to execute native kernels is a optional feature within OpenCL and the semantics of native kernels are defined in the devices OpenCL implementation. The OpenCL API contains functions for querying whether native kernels are supported on a specific device.

When a kernel function is submitted for execution (by placing a command to run the kernel function on the command queue) by the host application is a index space defined for the function. This index space can have up to 3 dimensions and is called the function's NDRange. For each point in the index space is an instance of the kernel function executed. A instance of a kernel function is called a work-item by OpenCL and is uniquely identifiable by its position in the index space. A work item's position in the index space is called its global ID and is defined as a tuple of size N where N is the number of dimensions of the index space. Each started work-item executes the same kernel code, but may take different paths through the code (by for example taking different clauses in if-statements) and the data the work-items operate on may vary.

Work-items are themselves organized into work-groups. The work-groups are assigned global IDs (tuples of size N where N is the number of dimensions of the index space) like the work-items to identify their position in the index space. Work-items are in addition to

their global IDs also assigned a local ID that signifies their position in their respective work groups.

Each work group is provided with one compute unit and the work-items in the work-group execute concurrently on the processing elements of the compute unit. The memory of the compute unit is shared among the work-items within the work-group. The work-items within a work-group are also capable of synchronizing their execution in relation to the other work-items in the work-group.

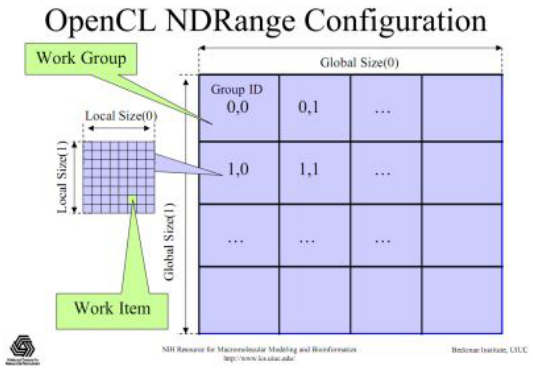


Figure 3.5: A figure displaying the concept of work-items and work-groups [16].

3.2.4 Memory Model

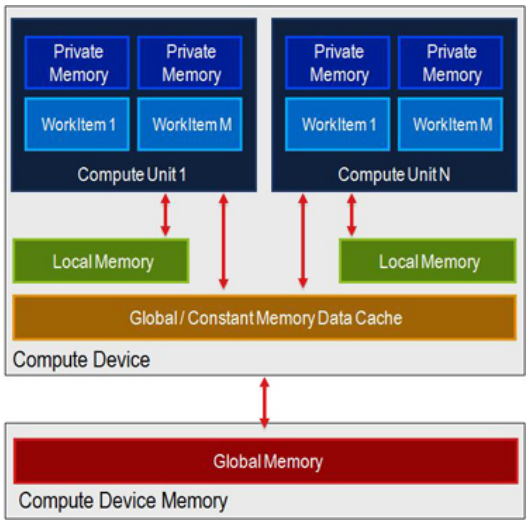


Figure 3.6: OpenCL's memory model [17].

There are four address spaces in OpenCL:

– **Global Memory:** The global memory is a memory region that is readable to all

work-items in all work-groups. Work-items can read from and write to every memory object that is stored in global memory.

- **Constant Memory:** A device's constant memory is allocated from the device's global memory. Every work-item has access to its own chunk of constant memory. The constant memory stays constant throughout the execution of a kernel and is used to store constants related to kernel functions.
- **Local Memory:** Local memory is memory shared among the work items within a work-group. A device's local memory is either implemented as dedicated regions of memory on the device or as regions of memory mapped from global memory. A work-group's local memory is accessible to all work items within that work-group and every work-item can read or write from/to it. A work-group's local memory is associated to the group's compute unit. Local memory can be significantly faster than global memory in some device architectures (for example on GPUs). This justifies moving data from global memory to local memory in some cases, because of the speed-up that can be gained by quicker memory accesses.
- **Private Memory:** Every work-item has access to its own chunk of private memory. A work-item's private memory is accessible only to that individual work-item and only that work-item can read from it or write to it.

The host and a devices memory models are for the most part independent of each other. This is necessary since the host's memory architecture is unknown to OpenCL and OpenCL can't make any assumptions about its structure. It is however necessary at times to transfer data from the host's memory to a device's global memory. This is done as mentioned earlier by transferring memory commands to the device via a command queue. There are commands for transferring data to or from memory objects stored in device memory and also for mapping/unmapping regions of memory objects. These memory commands can either be blocking or non-blocking. The OpenCL function call for executing a blocking command returns once the command has fully executed while the function for executing a non-blocking command returns as soon as the command has been enqueued into the command queue.

Memory Consistency

OpenCL uses a relaxed consistency model in its memory management. The model states that memory visible to a work-item is not guaranteed to be consistent across a collection of work-items at all times. The memory within a work-item has load/store consistency. Local memory within a work-group is consistent for work-items within that work-group at barriers. Barriers are explicitly placed points in the kernel code were the work-items within a work group should synchronize with each other. Global memory is also consistent at barrier points for the work-items within a work-group. There are however no guarantees for memory consistency between work-items in different work-groups.

Memory consistency for memory objects shared between different commands enqueued into the command queue is enforced at synchronization points.

3.2.5 Synchronization

There are two kinds of synchronization possible in OpenCL namely:

- Synchronization between the work-items in a work-group.
- Synchronization between commands enqueued into command queues in a single context.

Synchronization between the work-items in a work-group is done by using barriers (explained in the previous section). Barriers are typically used to achieve memory consistency, but can be used anywhere the programmer wants work-items within a work-group to synchronize with each other. A barrier must be reached by all work-items within a work-group or none at all. If just one work-item executing a kernel reaches a barrier must all work-items within that group execute the portion of the code (for example a loop or a conditional statement) containing the barrier even though they would not normally execute this portion of the code.

Synchronization between the commands enqueued into the command queues of a context is done with so called synchronization points. These synchronization points are:

- **Command-queue barriers:** A command queue barrier ensures that all commands enqueued before the barrier have finished executing and finished manipulating any memory objects before any subsequent commands are executed. This type of barrier can only be used to synchronize commands enqueued into the same command queue.
- **Waiting on events:** All OpenCL functions that enqueues commands into a command queue returns an event when a command has been enqueued that identifies which type of command that has been enqueued and which memory objects the command updates. Commands can wait on events before they begin execution and thereby receive a guarantee that they are synchronized with the commands generating the events.

3.2.6 Code Example

Sample Kernel

Here is a example of a simple kernel function:

Kernel 1 The vectorAdd kernel.

```
__kernel void vectorAdd(__global const float *a,
                  __global const float *b,
                  __global float *ans,
                  int vector_size)
{
    int iGID = get_global_id(0);
    if(iGID > vector_size) {
        return;
    }
    ans[iGID] = a[iGID] + b[iGID];
}
```

This kernel takes three equally-sized arrays of floating point numbers and the size of the arrays as input and adds the elements at corresponding positions in the two first arrays together and stores the resulting sums in the third array. All of the arrays given as input have the modifier "global" preceding them. This signifies that the arrays are stored in global memory. Because of the parallel nature of the kernel is there no need to use a loop to add the arrays together as one would have to do in a imperative language like C. OpenCL will start a thread (work-item) running the kernel for every index in the arrays and every thread will add two elements together. This eliminates the need for a loop. A thread knows which elements to add together by getting its global ID which tells it its position in the NDRange which in this case corresponds to the position in the arrays it should work on. The if-clause comparing the global ID to the array size is used as a bound check for the work-items similar to how a break-condition works in a loop. This bound check is used to compensate for the fact that the NDRange might not have the same size as the arrays. The NDRange might be bigger than the arrays because an other kernel function requires a bigger NDRange. The NDRange size might also have been rounded up to adapt to things like work-group size and hardware architecture (some GPUs likes to start a certain number of threads at a time). A work-item that is outside the bound of the arrays will simply return without doing anything.

Example Execution of the Kernel

A execution of the vectorAdd kernel can for example look like this:

```
float *a = |0|1|2|3|4|

float *b = |4|3|2|1|0|

clEnqueueWriteBuffer(cmdQueue, devA, CL_FALSE, 0, sizeof(cl_float) * globalWorkSize,
a, 0, NULL, NULL)

clEnqueueWriteBuffer(cmdQueue, devB, CL_FALSE, 0, sizeof(cl_float) * globalWorkSize,
b, 0, NULL, NULL)

clEnqueueNDRangeKernel(cmdQueue, vectorAddKernel, 1, NULL, &globalWorkSize,
&localWorkSize, 0, NULL, NULL)

clEnqueueReadBuffer(cmdQueue, devAns, CL_TRUE, 0, sizeof(cl_float) * globalWorkSize,
ans, 0, NULL, NULL)

float *ans = |4|4|4|4|4|
```

The a and b arrays are first written to the memory objects (devA and devB) stored in the context's devices global memory. This is done by enqueuing two memory write commands in the context's command queue. The command queue passes on these commands to the devices and the data in a and b is written to devA and devB. After the arrays have been written to device memory is a command to execute the kernel enqueued (this is done in the call to clEnqueueNDRangeKernel). The kernel is then executed on the devices and the result is saved in the memory object devAns stored in the devices global memory. Finally is the result read-back to the host (by enqueing a memory read command in the command queue) where it is saved in the ans array.

Chapter 4

Distance Fields

4.1 Introduction

A discrete distance field is a 3D grid of points (i.e. a voxel grid) [7] enclosing a triangle mesh. Every point (voxel) in the grid contains a scalar whose value is the distance from the cell to the closest point on the mesh. A distance field can either be signed or unsigned. In a signed distance field has a cell in addition to its distance value also a sign value. This sign value signifies whether a cell is inside or outside the triangle mesh. This sign value can either be 1 or -1 and is multiplied with the cells distance value. In this thesis are only signed distance fields used and a distance field is synonymous with a signed distance field. The sign of the cells distance values are needed when using distance fields for collision detection (for checking whether models are intersecting).

The naive approach for creating a distance field is to calculate the distance from every point in the grid to every triangle in the triangle mesh and pick the smallest distance for every point. The time complexity of this method becomes *O(nm)* where n is the amount of cells in the grid and m is the amount of triangles in the triangle mesh. This approach is sufficiently fast for relatively small triangle meshes and grids with limited resolutions. Its performance is however lacking for grids with large grid resolutions and for triangle meshes with large amounts of triangles. This is because of the large amount of calculations that has to be made. Implementations of distance fields normally solves this performance issue by either refining the algorithm so that not as many distances have to be calculated, by accelerating the process by using other hardware than the CPU (typically the GPU) or by using a combination of the two.

4.2 Definition of a Signed Distance Field

4.2.1 Triangle Mesh

A triangle mesh M is a union of triangles T_i where $i \in [1, N]$ with N being the number of triangles. This is defined as [7]:

$$M = \bigcup_{i \in [1,N]} T_i.$$

It is assumed in this paper that M is a closed, orientable 2-manifold in 3D Euclidian space (a model with a surface with no holes and a clearly defined inside and outside). This

assumption is important in the sign calculation for the cells in the distance field's grid because a cells sign specifies whether the cell is inside or outside the model, but the inside and outside of a model is only defined for models that are closed, orientable manifolds.

It is possible to enforce the manifold condition for meshes by requiring that:

- The mesh should not contain any self-intersections. Triangles may only share edges and vertices and must otherwise be disjoint.
- Every edge must be a part of exactly two triangles.
- Triangles sharing a vertex must form a single cycle around that vertex.

4.2.2 Signed Distance Field

A signed distance field is a scalar grid that specifies the minimum distance to a shape with the signs of the distance values acting as indicators of what is inside and outside the shape. It can be defined as follows [19]:

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

$$D(\mathbf{r}, M) \equiv S(\mathbf{r}, M) \cdot \min_{\mathbf{x} \in M} \{|\mathbf{r} - \mathbf{x}|\}, \qquad \forall \mathbf{r} \in \mathbf{R}^3$$

D takes a triangle mesh M as input and gets the distances between every point $\mathbf{r}$ in 3D space and the points on the triangle mesh they are closest to. The result of running S on $\mathbf{r}$ is multiplied with every calculated distance.

S takes a point in 3D space and a triangle mesh M as input and determines whether the point is inside or outside M. It is defined as follows [20]:

$$S(\mathbf{r}, M) = \begin{cases} -1 & \text{if } \mathbf{r} \in M, \\ 1 & \text{else.} \end{cases}$$

When the function D is used in practice does it usually not process every point in 3D space and does instead work on a subset of $\mathbb{R}^3$ (the discrete grid) which works as an approximation of $\mathbb{R}^3$. It must be decided on a case to case basis on how much of $\mathbb{R}^3$ that needs to be processed by D.

4.3 Sign Computation

4.3.1 Background

The task of computing the sign of a cell's distance value consists (as mentioned in the previous section) of determining whether the cell is inside or outside the triangle mesh. This can be accomplished in a number of ways: One can divide the grid into a number of z-level planes and then calculate the intersection between these planes and the mesh. This produces 2D contours that can later be scan-converted and used to calculate the sign of the cells. This was suggested by Payne and Toga in [21]. A simple method would be to cast a ray along each row of cells. At every cell where the ray has crossed the border of the mesh an uneven number of times do we know we are inside the mesh. Another method to calculate the sign was proposed by Mauch in [22]. This method consists of creating truncated Voronoi regions [23] for every face, edge and vertex of the mesh. The regions that corresponds to

the faces and edges will be either interior or exterior to the mesh depending on whether the mesh is locally concave or convex. This can later be used to calculate the sign for the cells.

These methods all have their own advantages, but most of them requires the distance field to be scan converted which adds complexity to the implementation. An alternative to these methods is to calculate the sign locally at the closest point on the mesh by using the normal at the point to determine the sign (by using the equation of the plane).

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0, \\ 1 & \text{else.} \end{cases}$$

$$S_{local}(\mathbf{p}, \mathbf{n}) = sgn(\mathbf{p} * \mathbf{n}).$$

The signum function (sgn) which is used for extracting the sign always returns a non-zero value. This is to prevent the multiplication of the sign with the distance value from producing zero which would result in errors in the distance field.

Local methods for calculating the sign have the advantage of being possible to integrate into the distance calculation. This makes it possible to construct the signed distance field in a single pass which saves time. Local methods does however have problems in some situations where it is hard to determine the correct normal to use in the sign calculation. These situations occur when the closest point on the mesh from a cell lies on one of the triangles edges or is one of its vertices. The edges and vertices of a triangle does not have any exact normals like the triangles face has and it is only possible to calculate approximate normals (also called pseudo normals) for these types of triangle features. The plane test can in some cases return the wrong sign when it is run on these approximate normals. A solution to this problem was proposed by Baerentzen and Aanaes in [7] where a local sign computation method using angle-weighted pseudo normals was presented. This method proved to be quite successful and solved the sign problem for most kinds of meshes.

This method was chosen to handle the computation of the sign in the implemented application. This was because of the quality of the methods sign computation and the possibility to integrate the sign computation into the distance test.

4.3.2 Angle-Weighted Pseudo Normals

An angle-weighted pseudo normal is an approximate normal for an edge/vertex that is a weighted sum of the face normals of the triangles neighboring the edge/vertex. The face normals are weighted with the incident angle of the face towards the edge/vertex.

In the case of an edge between the two faces i and j with the normals $\mathbf{n}_i$ and $\mathbf{n}_j$ is the angle weighted normal $\mathbf{n}_\alpha$ given by:

$$\mathbf{n}_\alpha = \pi\mathbf{n}_i + \pi\mathbf{n}_j.$$

An edge in a well-formed manifold is shared by exactly two triangles so the edge normal is never affected by more than two face normals. The incident angle between a triangle and one of its edges is always π so the expression for obtaining $\mathbf{n}_\alpha$ can be simplified as:

$$\mathbf{n}_\alpha = \mathbf{n}_i + \mathbf{n}_j.$$

In the corresponding vertex case (which is also the general case) is it not known how many triangles the vertex is a part of and n_α is given by:

$$\mathbf{n}_\alpha = \sum_i \alpha_i \mathbf{n}_i,$$

where α_i is the incident angle of triangle i and $\mathbf{n}_i$ is the face normal of triangle i.

It is not guaranteed that a computed angle-weighted normal is normalized (it is possible for an angle weighted normal to have coordinates that have an absolute length of more than 1). The scaling of the normal does not affect the sign computation (the plan test) so it is therefore not necessary to normalize an angle-weighted normal. This is however necessary if the normal should be used for anything other than the sign computation.

4.4 Distance function

An important part of any implementation of distance fields is the actual distance function. The quality of the distance function directly affects how good a distance field will perform in its application area. The distance function calculates the distance between a point (a cell) and a triangle. The easiest way to achieve this is to calculate the distance between the point and the triangles centroid point (see figure 4.1). This naive function is fast to compute, but gives a bad approximation when the triangle is large and might not give a result with the required accuracy for the application. The programmer must decide on how important accurate distance calculations are for his/her application in comparison to a fast computation of the distance field and pick a distance function that fits the requirements of the application. For the application implemented in this thesis was a precise distance function presented by Christer Ericsson in [24] chosen. This function was chosen because of the large accuracy requirements of collision detection where precise measurements are necessary for a good performance.

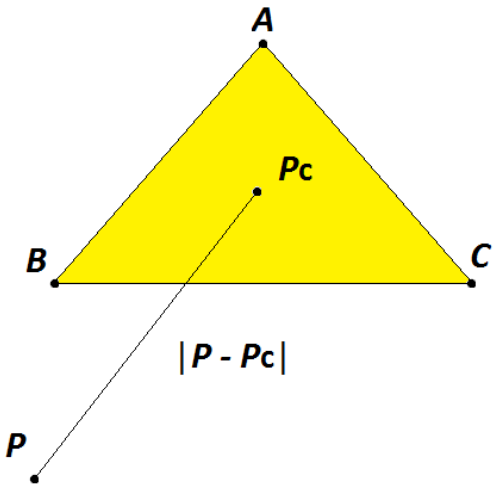


Figure 4.1: The distance between a point P and a the triangles centroid point P_C.

4.4.1 Distance to the Closest Point on a Triangle

The algorithm presented in [24] calculates the distance from a point and the closest corresponding point on the triangle. This point on the triangle can either lie on the triangle's face, one of its edges or be one of the triangle's vertices. The algorithm divides the space around the triangle into 7 Voronoi regions (3 vertex regions, 3 edge regions and 1 face region). The algorithm then tries to discern which region the source point lies in. When this is known is the point projected onto the feature (vertex, edge or face) of the region it lies in. The projected point is the nearest point on the feature from the perspective of the origin point and also the closest point on the entire triangle (because the origin point lies in the features region). The distance to the triangle is then computed by calculating the distance from the origin point to the projected point.

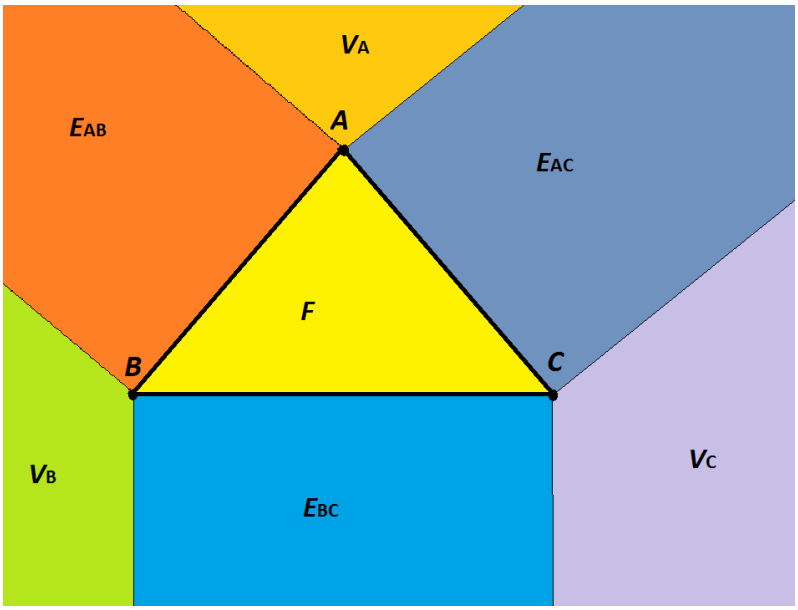


Figure 4.2: The area around a triangle is divided into the Voronoi regions F, E_{AB}, E_{AC}, E_{BC}, V_A, V_B and V_C.

Algorithm Description

Here follows a more detailed description of the algorithm. The algorithm has been slightly modified to also calculate and apply the sign of the calculated distance which the original algorithm does not do.

Algorithm 1 Signed distance from a point to a triangle.

procedure SIGNEDDISTANCETOTRIANGLE($\mathbf{p}$, $\mathbf{v_a}$, $\mathbf{v_b}$, $\mathbf{v_c}$)
  R_a = makeRegion($\mathbf{v_a}$)
  if insideRegion($\mathbf{p}$, R_a) **then** ▷ Check if $\mathbf{p}$ is in the region outside the vertex $\mathbf{v_a}$
    return sign($\mathbf{p}$) * $||\mathbf{p} - \mathbf{v_a}||$
  end if
  R_b = makeRegion($\mathbf{v_b}$)
  if insideRegion($\mathbf{p}$, R_b) **then** ▷ Check if $\mathbf{p}$ is in the region outside the vertex $\mathbf{v_b}$
    return sign($\mathbf{p}$) * $||\mathbf{p} - \mathbf{v_b}||$
  end if
  R_{ab} = makeRegion($\mathbf{v_a}$, $\mathbf{v_b}$)
  if insideRegion($\mathbf{p}$, R_{ab}) **then** ▷ Check if $\mathbf{p}$ is in the region outside the edge $\mathbf{e_{ab}}$
    $\mathbf{p_{AB_{proj}}}$ = projectPoint($\mathbf{p}$, $\mathbf{v_a}$, $\mathbf{v_b}$)
    return sign($\mathbf{p}$) * $||\mathbf{p} - \mathbf{p_{AB_{proj}}}||$
  end if
  R_c = makeRegion($\mathbf{v_c}$)
  if insideRegion($\mathbf{p}$, R_c) **then** ▷ Check if $\mathbf{p}$ is in the region outside the vertex $\mathbf{v_c}$
    return sign($\mathbf{p}$) * $||\mathbf{p} - \mathbf{v_c}||$
  end if
  R_{ac} = makeRegion($\mathbf{v_a}$, $\mathbf{v_c}$)
  if insideRegion($\mathbf{p}$, R_{ac}) **then** ▷ Check if $\mathbf{p}$ is in the region outside the edge $\mathbf{e_{ac}}$
    $\mathbf{p_{AC_{proj}}}$ = projectPoint($\mathbf{p}$, $\mathbf{v_a}$, $\mathbf{v_c}$)
    return sign($\mathbf{p}$) * $||\mathbf{p} - \mathbf{p_{AC_{proj}}}||$
  end if
  R_{bc} = makeRegion($\mathbf{v_b}$, $\mathbf{v_c}$)
  if insideRegion($\mathbf{p}$, R_{bc}) **then** ▷ Check if $\mathbf{p}$ is in the region outside the edge $\mathbf{e_{bc}}$
    $\mathbf{p_{BC_{proj}}}$ = projectPoint($\mathbf{p}$, $\mathbf{v_b}$, $\mathbf{v_c}$)
    return sign($\mathbf{p}$) * $||\mathbf{p} - \mathbf{p_{BC_{proj}}}||$
  end if
    ▷ Since $\mathbf{p}$ is not in any other region must it be inside the bounds of the triangles face
  $\mathbf{p_{ABC_{proj}}}$ = projectPoint($\mathbf{p}$, $\mathbf{v_a}$, $\mathbf{v_b}$, $\mathbf{v_c}$)
  return sign($\mathbf{p}$) * $||\mathbf{p} - \mathbf{p_{ABC_{proj}}}||$
end procedure

The actual OpenCL routine can be found in the appendix.

Chapter 5

Collision Detection Using Distance Fields

Collision detection is a common application for distance fields. A signed distance field provides a contour of the mesh it is representing and can both be used to detect collisions and get contact information (like the penetration depth and the contact normal) for the collisions that occur. The primary focus of this thesis is to implement collision detection between pairs of distance fields, but ways to detect collisions between distance fields and geometric primitives (like planes, spheres and cubes) are also considered. A variant of the collision detection method presented by Kenny Erleben in [9] is used for detecting collisions between pairs of distance fields. This method both detects collisions between pairs of distance fields and extracts the contact data for these collisions. The method used for detecting collisions between distance fields and geometric primitives varies with the primitive type. Collision detection between distance fields and geometric primitives is not explored that much in distance field literature, but is quite easy to implement for most primitive types because of the well-known shapes of the primitives.

This section will explain the components of the method used for detecting collisions between pairs of distance fields. It will also explore detecting collisions between distance fields and geometric primitives.

5.1 Distance Field-Distance Field Collisions

5.1.1 Sampling the Mesh

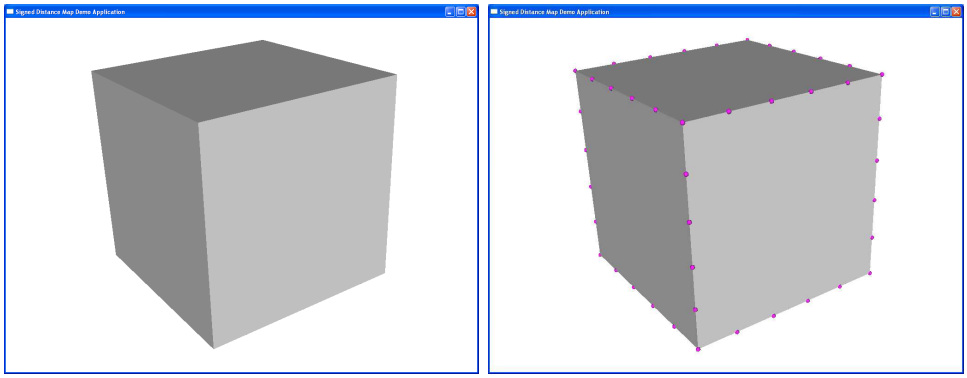
In addition to the distance field itself is also a set of points called sampling points needed in order to detect a collision between a distance field and another distance field. The sampling points are generated from the triangle mesh and are a sampling of the mesh's triangle features. This sampling acts as a simplified representation of the triangle mesh. The sampling points of a mesh are compared against another mesh's distance field in order to detect collisions between the meshes and locate where the collisions occur. The sampling points are generated from the mesh's triangle features in the following steps:

- **Vertices:** All of the mesh's vertices that lie in a non-flat region are added as sampling points. The vertices that lie in flat regions on the mesh's surface are however not added as sampling points. This is because if a collision occurs in such a region will

it be detected by a sampling point from the other mesh penetrating the zero-level-set surface. Vertices in flat regions can therefore be ignored. A vertex lies in a flat region when it has no concave or convex incident edges.

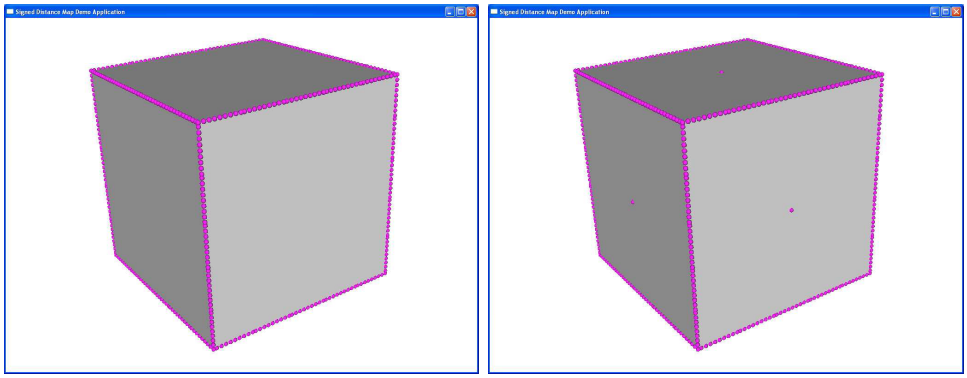
- **Edges:** An edge is sampled into a set of sampling points if it has at least one non-planar neighboring face and has a length greater than a sampling threshold. The sampling is done by adding sampling points along the edge a sampling threshold length apart from each other. The sampling threshold is chosen as the maximum of a user specified threshold and the diagonal of a grid cell (in the distance field's grid). In this way can the programmer achieve a sampling density along edges which corresponds to the resolution of the distance field grid (by setting the user specified threshold to zero) while also having the opportunity to have a coarser sampling for performance reasons.

- **Faces:** Vertex and edge sampling is usually sufficient for detecting collisions. However for some meshes does vertex and edge sampling not produce the contact points needed by the simulation to generate the appropriate response to a collision. An example of a situation where just vertex and edge sampling would be inadequate is when one cube lies perfectly aligned on top of an other cube. In this situation would only vertex and edge sampling fail to produce any contact point with a normal in the penetrating direction. This would result in the upper box would sinking right through the lower box. To prevent this are additional sampling points inserted on the flat surfaces of the mesh. A breadth first traversal is done over the mesh's surface to find regions of coplanar triangles. For each such region is a single centroid point computed which is the average point of all vertices of the triangles in the region. These centroid points are added as sampling points. In the case of the two cubes would a sampling point be placed on the surface of every side of the cubes. These sampling points would help remedy the problem and would result in a contact point with a normal in the contact direction being produced. To prevent too many sampling points from being made is the area of a flat region computed and only when a regions area is significantly larger than the largest side of a grid cell is a sampling point added for the region.



(a) Original mesh

(b) Vertex re-sampling



(c) Vertex and edge re-sampling

(d) Vertex, edge and face re-sampling

Figure 5.1: Figure from [9] by Kenny Erleben displaying different levels of mesh sampling.

Re-sampling Type	Sampling points
Only vertices	56
Edge sampling	536
Edge and face sampling	542

Table 5.1: A table of the sampling points generated for the cube mesh. The mesh has 152 vertices, 450 edges and 300 faces.

5.1.2 Detecting a Collision

The method chosen for detecting collisions deviates from the method used by Kenny Erleben in [9].

The method used in [9] inserts the sampling points of the mesh into a bounding sphere hierarchy (a sphere tree) where the sampling points themselves acts as leaves. At the root of the sphere tree is a bounding sphere that encompasses all of the vertices of the mesh. This sphere contains a series of smaller spheres who themselves contain even smaller spheres. This subdivision of the mesh into smaller and smaller spheres continues until the sampling points are reached which becomes the leaves of the tree. A collision query consists of traversing the bounding sphere hierarchy of a mesh and comparing it against the distance field of an other mesh. If a sphere's center lies outside the distance field, but its surface intersects the AABB (Axis Aligned Bounding Box) of the distance field does the method descend to the spheres children. The distance field's AABB is an axis-aligned box that encompasses all of the vertices of the distance field's mesh. If the center of the sphere lies inside the distance field does the method look up the distance values of the 8 surrounding grid nodes. If the minimum distance of these distance values is less than the spheres radius plus the collision envelope does the method descend to the spheres children. Otherwise is the sphere pruned. This descent through the tree continues until the tree's leaf nodes (the sampling points) are reached. It is checked if any of the reached sampling points is a contact point and if any contact point is found has a collision occurred.

The method presented in [9] was deemed as unsuitable for this thesis since it is not suited for data parallel processing and would be hard to implement as an OpenCL routine. A simpler more parallizable method was instead chosen.

This method simply treats the sampling points as a set and takes a mesh's sampling points and tests them against the distance field of another mesh. If a sampling point lies in a cell with a distance with a negative sign has a collision occurred and the sampling point should be added as a contact point. This can be described more formally as:

Algorithm 2 Detect collisions between two distance fields.

```
1:  procedure DetectDistFieldDistFieldCollision(DistanceField d, *sampPoints)
2:      for all s of sampPoints do
3:          s_cell = getCell(s, d)          ▷ Gets the cell a point occupies in a distance field
4:          if s_cell.dist ≥ 0 then
5:              continue          ▷ Ignore sampling points that are not inside the other mesh
6:          else
7:              GenerateContactPoint(s, d)          ▷ Explained in "Algorithm 3"
8:          end if
9:      end for
10: end procedure
```

The advantage of this method is that it is simple and data parallizable. The data items being processed (the sampling points) are independent of each other and a data item does not need to know anything about the other data items. The data can thereby be processed in parallel without threads ever having to synchronize with each other. The algorithm also has a simple control flow which also helps parallel processing.

5.1.3 Contact Point Generation

Contact points are generated from sampling points. When a sampling point of a mesh is located inside another mesh is a contact point generated from it. A contact point contains in addition to the position of the point, a contact normal and the penetration depth. The contact normal is found by approximating the gradient of the distance field at the contact point's position. The penetration depth is given by taking the absolute value of the distance value stored in the cell that the contact point is located in.

Algorithm 3 Generate a contact point from a sampling point in a distance field-distance field collision.

```
1: procedure GenerateContactPoint(s, DistanceField d)
2:     s_cell = getCell(s, d)            ▷ Gets the cell a point occupies in a distance field
3:     n = calcGradient(s_cell, d)       ▷ Calculates the gradient at a cell in a distance field
4:     depth = −s_cell.dist
5:     p_cont = Contact(s, n, depth)
6:     addContactPoint(p_cont)
7: end procedure
```

5.2 Distance Field-Plane Collisions

The sampling points introduced in section 5.1.1 can be used to detect collisions between distance fields and planes. To detect a collision with a plane are the distance field's sampling points tested against the plane. This test consists of calculating the distance between the sampling point and the closest point on the plane from the perspective of the sampling point. By looking at the sign of the calculated distance can we tell on which side of the plane the sampling point is located and if it has passed through the plane. If any of the distance field's sampling points has crossed the plane is a contact point generated from the point. The contact normal used will be the planes normal and the penetration depth will be the calculated distance between the sampling point and the closest point on the plane.

Algorithm 4 Detect collisions between a distance field and a plane.

```
1: procedure DetectDistanceFieldPlaneCollision( *sampPoints, Plane p)
2:     for all s of sampPoints do
3:         d = calcDistance(s, p)
4:         if ¬ testSign(d) then
5:             continue   ▷ Ignore sampling points that have not passed through the plane
6:         else
7:             n = p.normal
8:             depth = d
9:             p_cont = Contact(s, n, depth)
10:            addContactPoint(p_cont)
11:        end if
12:    end for
13: end procedure
```

5.3 Distance Field-Line Collisions

A method that utilizes the Cohen-Sutherland line clipping algorithm[25] is used to detect collisions between distance fields and lines. It first checks which cells in the distance field that the line's endpoints lies in. The endpoints cells defines a line through the distance field grid that mirrors the original line. The line through the grid is given to the Cohen-Sutherland algorithm which clips the line so that its endpoints are inside the distance fields AABB. If the line–clipping algorithm finds out that the line is totally outside the distance fields AABB is the line skipped from further processing because it could not possibly exist a collision between the line and the distance field.

Collisions between the distance field and the line is later found by going along the line cell by cell and checking if any cell along the line has a negative distance. The stepping along the line is achieved through the use of Bresenham's line drawing algorithm[26]. If one of the line's cells has a negative distance is the cell inside the distance field's triangle mesh and we have a collision between the line and the mesh. Only one contact point is extracted by the algorithm and this point is the center point of the first cell along the line that is inside the mesh. The contact normal that gets used is the normalized distance vector between the position of the contact point and the line's start point. The penetration depth is calculated by taking the absolute value of the distance value of the first colliding cell.

Algorithm 5 Detect collisions between a distance field and a line.

```
1:  procedure DetectDistanceFieldLineCollision(DistanceField d, Line l)
2:      p1_cell = getCell(l.p1,, d)
3:      p2_cell = getCell(l.p2,, d)
4:      l_cell = Line(p1_cell, p2_cell)
5:      if ¬clipLineCohenSutherland(l_cell) then
6:          continue ▷ Discard the line from processing if it is totally outside the distance
    field's AABB
7:      else
8:                                  ▷ The stepping along the cells occupied by the line is achieved
    with Bresenham's line drawing algorithm. nextCell() picks the next cell along the line.
    It tries to pick the cell which best fits the line.
9:          for p_cell := l_cell.p1, p_cell = nextCell(p_cell) to l_cell.p2 do
10:             p_cell_pos = getCellMidpointPosition(p_cell)
11:             n = p_cell_pos − l.p1
12:             depth = p_cell.dist
13:             p_cont = Contact(p_cell_pos, n, depth)
14:             addContactPoint(p_cont)
15:             break
16:         end for
17:     end if
18: end procedure
```

Pseudo code versions of Cohen-Sutherlands line clipping algorithm and Bresenham's line drawing algorithm can be found in the appendix.

5.4 Collisions between Distance Fields and Other Primitives

Lines and planes were considered the most important geometric primitives to implement collision detection(with distance fields) for. Other primitive types were also considered for implementation, but were cut out due to lack of time.

It would however be quite easy for most simple primitive types (like spheres and cubes) to implement collision detection with distance fields. Most of the simple primitive types have well defined methods for determining whether a point is inside or outside the primitive. To detect a collision can one simply check if any of the distance field's sampling points are inside the primitive. If any sampling point is inside the primitive do we have a collision. The intersecting sampling points can be used to generate contact points. The penetration depth for a contact point is computed by calculating the distance from the contact point to the closest point on the surface of the primitive. The contact normal would be possible to compute by getting the normalized distance vector between the contact point and the closest point on the primitive's surface.

This method would work in most cases, but would have problems with a mesh significantly larger than the primitive. The mesh could then have flat surfaces large enough for the primitive to pass through without ever hitting a sampling point. It is therefore necessary to also have points on the primitive that can be checked against the distance field. It is up to the programmer to find the points on a primitive that best relates its shape. For a cube could one for example place a point at each corner of the cube and at the center of the surface of each of the cube's sides.

Chapter 6

Implementation

The collision detection software developed in this thesis was implemented in AgX (see Section 6.1), a simulation toolkit for physics simulation and physics API developed by Algoryx. The choice to use AgX was made to have a platform for running physical simulations and to have utilities for computing the appropriate physical response for collisions. In this way could focus be put on the implementation of the distance fields with no time having to be put on programming an application capable of running graphical simulations and programming the physics of such an application. The use of AgX also eliminated a potential error source by handling the physical response for collisions. The functions used for computing the physical response for a collision in AgX have been extensively tested and generally delivers correct results. Potential bugs in the collision handling software can then likely be assumed to be in the implemented collision detection routines and not in the collision response.

6.1 AgX

6.1.1 Overview

AgX is a simulation toolkit for physics simulation and physics API written in C++, developed by Algoryx. AgX is usable on Windows, Linux and Mac OSX. The first version of AgX to be released to customers (AgX v0.5) was released in January 2008 [3]. AgX is continuously being improved by Algoryx and provides the core functionality needed to perform stable and robust simulations of mechanical systems in real-time. AgX have several different solvers to choose from. These solvers are suited for different kinds of purposes. There are for example iterative solvers for fast and approximate solutions and direct solvers for simulations requiring high accuracy. There also are hybrid solvers which tries to combine aspects of both iterative and direct solvers. AgX also have event handlers that allows programmers to create listeners for contact events, step events etc. AgX also allows users to save simulation data which can be imported into another simulation later. This is accomplished with a Collada Physics reader. This reader supports a large portion of the standardized items described in the Collada DOM 2.0 standard [13].

AgX consists of many different components, but the focus of this section will be to explain the components of AgX that are relevant to the distance field implementation.

6.1.2 Collision Primitives

AgX has several different representations of an object (for example a cube or sphere) in a simulation.

- **Graphical representation:** The graphical representation consists of the objects vertices, edges and colors. This representation is used to draw the object on the screen.
- **Physical representation:** The physical representation consists of the physical attributes of the object like weight, friction and inertia. This information is used by the solver to compute how physical laws like gravity affects the object and how the object should react when it collides with other objects in the simulation.
- **Collision primitive representation:** The collision primitive representation contains the information necessary for detecting collisions with the object. The structure of a collision primitive can be very simple in some cases, like for a sphere whose collision primitive consists of only its center point and its radius or for a cylinder whose collision primitive consists of only its height and radius. A collision primitive can also have a quite complex structure if the shape that it is representing has an complex structure and the appearances of two objects of the same shape type can vary quite much (which is the case for triangle meshes). All collision primitive classes in AgX extends the `Shape` class.

The collision primitives of the objects in the simulation are compared against each other by AgXs collision detection routines which are called colliders. A separate collider is implemented for every possible pair of primitive types. Colliders are explained in more detail in the next section.

The distance field implementation developed in this thesis have been implemented as a collision primitive. The distance field primitive provides collision data for triangle meshes.

6.1.3 Colliders

AgX have a number of collision detection routines called colliders. There is a collider for each possible pair of collision primitives. A collider takes an interaction pair as input. An interaction pair is a pair of collision primitives that might be colliding. Interaction pairs are generated by the broad-phase collision detection whose job is to find all potential collisions. The collider looks at the primitives in the interaction pair it has been given as input and tests if any collision has occurred. If a collision has occurred does the collider extract contact points (where a contact point contains its position, the contact normal and the penetration depth) for the collision. The generated contact points are added to a contact object (a set of contact points) that is later returned to the simulation which applies the appropriate physical response for the collision.

6.2 Distance Field

The distance field implementation that was developed in this thesis was implemented as a collision primitive in AgX. This was done in order to be able to use the implemented distance fields in AgX colliders and integrate support for distance fields into AgX.

6.2.1 System Overview

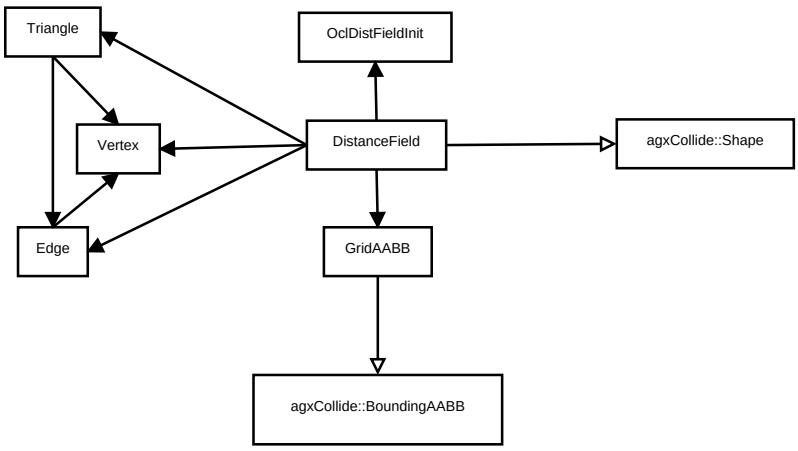


Figure 6.1: A class diagram displaying the classes the `DistanceField` class interacts with (excluding utility classes like vector and matrix classes).

The distance field implementation consists of a number of classes in addition to the actual distance field class (named `DistanceField`). Below follows a brief description of these classes and their relationship with each other.

DistanceField

`DistanceField` is the main class of the distance field implementation and distance fields are represented as `DistanceField` objects in the application. `DistanceField` extends AgX's `Shape` class. `Shape` is an abstract class representing a collision primitive. No `Shape` objects are ever initialized. A `DistanceField` object contains the distance field grid and the mesh's sampling points. The distance field's grid is a grid of scalar distance values where each distance value is the smallest distance from the cell to the corresponding object. The properties of the grid are defined by the distance field's `GridAABB` object. A distance field's `GridAABB` object acts as the distance field's AABB and also contains information about its grid. It defines where the grid is in space and can be used by the application to extract information about the grid and individual cells of the grid.

When a `DistanceField` object is created is a previously read-in triangle mesh taken as input, which consists of a vector of vertices and a vector of indices. The read-in mesh is processed and built into a triangle feature hierarchy. Each triangle feature type has its own class and the triangle features in the hierarchy are represented as objects of these classes (`Triangle`, `Edge` and `Vertex`). This hierarchy is needed by the application for calculating the face and pseudo normals of the mesh's triangles. It is also needed for computing the mesh's

sampling points. The processing of the mesh and construction of the triangle hierarchy is explained in more detail in Section 6.2.6.

The normals of the mesh's triangles are computed right after the triangle hierarchy has been completed. After the normals have been computed is the distance field's grid created and its cells distance values are computed. This is done by initializing a kernel for calculating the cells distance values on the GPU and starting it with the mesh's triangles and normals as input. After the grid has been initialized are the mesh's sampling points computed. This is done as mentioned in Section 5.1.1, with the help of the triangle hierarchy which is used to find flat regions on the mesh. After both the distance field's distance values and sampling points have been computed is the distance field cached to a file. This saves processing time in further executions since it prevents distance fields for complex meshes from having to be made multiple times. This process is explained in more detail in Section 6.2.6.

GridAABB

The `GridAABB` class is a special kind of AABB and extends AgX's AABB class called `BoundingAABB`. A `GridAABB` object encloses all of the vertices of the triangles of a distance field's mesh. `GridAABB` also defines the properties of the distance field's grid like the grid's dimensions, position in space and the distance between the cells in the grid. It also provides the application with functions for getting information related to the distance field grid and information about individual grid cells.

Triangle

The `Triangle` class is used for representing a triangle in the application. It is one of the classes used for building the triangle hierarchy used in `DistanceField`. A `Triangle` object contains pointers to all of its vertices and edges. The triangle's vertices and edges are represented in the application as `Vertex` and `Edge` objects. A `Triangle` object also contains the triangle's normal.

Edge

The `Edge` class is used for representing an edge in the application. An `Edge` object contains pointers to its source and destination vertices. It also contains the edge's pseudo normal. An `Edge` object also have pointers to all of the triangles it is a part of (which is two triangles if the mesh is a 2-manifold).

Vertex

The `Vertex` class is used for representing an vertex in the application. A `Vertex` object contains the vertex's position and its pseudo normal. It also contains pointers to all triangles it is a part of.

OclDistFieldInit

The `OclDistFieldInit` class is used for setting up and running the GPU kernel used for initializing the distance field grid. `OclDistFieldInit` provides the necessary host code for running the kernel. It sets up the components needed on the host side, like a command queue and input and output buffers. `OclDistFieldInit` also handles the bookkeeping related to running the kernel such as deallocating memory buffers and other allocated resources, like the command queue.

When the kernel is to be run does `OclDistFieldInit` allocate memory for the distance field grid. `OclDistFieldInit` then receives the mesh's triangles and normals as input from `DistanceField` and executes the kernel with the triangles and normals as input. The kernel computes the grid cells distance values and initializes the grid. The initialized grid is then read back to `OclDistFieldInit` from the GPU. `OclDistFieldInit` then relays the initialized grid to the `DistanceField` object.

6.2.2 Mesh Processing

The distance field implementation creates distance fields for triangle meshes extracted from .obj files. An AgX utility named `MeshReaderOBJ` is used for extracting a mesh from an .obj file. `MeshReaderOBJ` extracts two vectors when extracting a mesh: a vertex and index vector. The vertex vector consists of a number of 3D points (3 dimensional vectors of floating point numbers) which are the vertices of the triangles of the mesh. The index vector contains a number of 3 dimensional vectors of integers. These 3 dimensional vectors represents the mesh's triangles and the values of a vector represents the indices in the vertex vector where the triangle's vertices can be found.

These vectors are taken by `DistanceField` and made into a more complex structure (a triangle hierarchy). The first step in making this structure consists of converting every 3D point in the vertex vector to `Vertex` objects. These `Vertex` objects are stored in a vector mirroring the original vertex vector where a `Vertex` object in the new vector is stored at the same position in the vector as the original point was stored in the old vector.

The next step consists of going through the index vector 3 elements at the time and for every three indices create a `Triangle` object and add the the created triangles to a `Triangle` vector object. Each `Triangle` object contains pointers to the `Vertex` objects that the triangle's 3 indices corresponds to in the `Vertex` vector. When a `Triangle` object is created are pointers to the triangle added to the triangles vertices. A vertex is typically part of several triangles so a `Vertex` object has usually collected a series of `Triangle` pointers before the index vector has been fully processed. When a `Triangle` object is created are three `Edge` objects also created. These edges contains pointers to their source and destination vertices as well as a pointer to the triangle that created it. All created `Edge` objects are stored in a vector in `DistanceField`. When a `Triangle` object is about to create an `Edge` object does it first check in the `Edge` vector if that particular edge has been created before and if that is the case is this old edge used instead of creating a new one. The old edge receives a pointer to the triangle when a pointer to the edge is added to the triangle.

When the entire index vector has been traversed is the triangle hierarchy finished. The result is three vectors: a triangle, edge and vertex feature vector with pointers connecting related features to each other.

Here is an example of how the triangle hierarchy can look like:

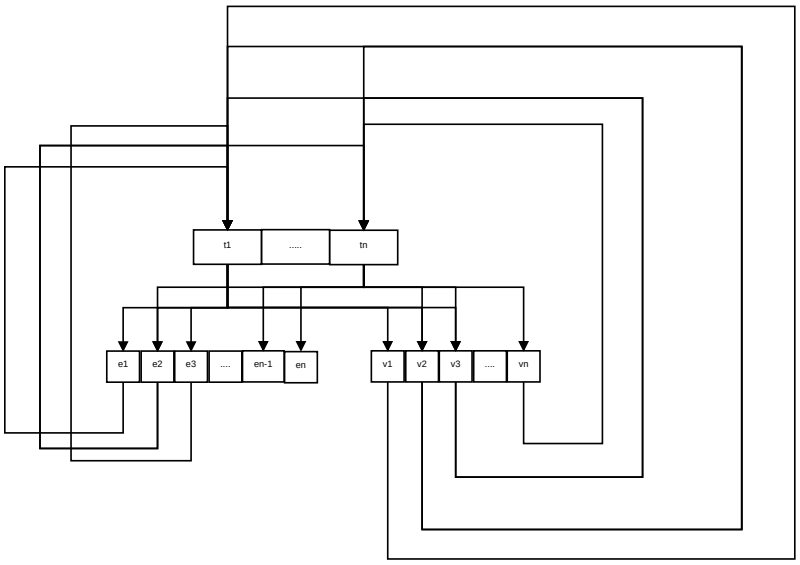


Figure 6.2: The triangle hierarchy. t_n shares the edge e_2 with t_1. e_2 consists of the vertices v_2 and v_3.

6.2.3 Generation of Triangle Feature Normals

Three types of normals are computed for the triangles of a mesh: face, edge and vertex normals.

The face (surface) normal of a triangle is a well defined concept and is calculated by taking the cross product of the distance vectors between two pairs of vertices. A triangle's face normal is calculated by the application when the triangle is created with the computation of face normal built into `Triangle`'s constructor.

Edge and vertex normals can not be calculated exactly and can only be approximated. This is because edges and vertices does not have a surface and does not contain enough information by themselves for it to be possible to calculate a normal. A normal for a vertex or an edge normal must therefore be approximated by using the face normals of its neighboring triangles.

The edge and vertex normals are calculated by using the triangle hierarchy. The vertex/edge uses its triangle pointers to compute a weighted sum of the face normals of the triangles it is a part of. The sum is weighted according to the scheme explained in Section 4.3.2.

6.2.4 Generation of Sampling Points

The mesh's sampling points are computed as a step of `DistanceField`'s constructor. The triangle hierarchy of the mesh is used when the sampling points are computed. `DistanceField`'s sampling point function goes through `DistanceField`'s feature vectors and generates sampling points for the stored features.

The function begins with processing the `Vertex` vector. It is first checked if a processed vertex is located in a flat region of the mesh. This is accomplished by using the vertex's triangle pointers to check if all of the triangles that the vertex is a part of has the same normal as the vertex's normal. If the vertex lies in a flat region is the vertex skipped and no sampling point is added. If the vertex lies in a non-flat region is the vertex added as a sampling point.

After every vertex in the vertex vector have been processed does the function iterate through the `Edge` vector. It is first checked if a processed edge lies in a flat region. This is done in the same way as for a vertex. The function then checks if the edge is long enough to be sampled. This is done by checking if the edge is longer than the sampling threshold (the distance between two sampling points on an edge). If the edge is both long enough and does not lie in a flat region should sampling points be generated for the edge. The function achieves this by placing sampling points along the length of the edge placed so that they lie one sampling threshold apart from each other.

The function finally goes through the `Triangle` vector and checks if any of the mesh's surfaces requires a sampling point. This is done by trying to find groups of triangles where each triangle in the group has the same normal. If a group with a big enough area is found is a sampling point added to the group's surface. An area is deemed large enough when it is considerably larger than any side of a cell in the distance field grid. The sampling point that gets added is a centroid point for the surface which is the average point of all of the vertices of the triangles in the triangle group.

A difference from the method presented by Erleben in [9] is that a sampling point does not just consist of its position in space. A sampling point has in addition to its position also a normal defined for itself. This normal is the same normal as the one used by the triangle feature the sampling point lies on. So when the sampling point is a vertex is the vertex's normal used, when the sampling point lies on an edge is the edge's normal used and when the sampling point lies on a flat surface is the surface's normal used.

The reason the sampling points have normals defined is to prevent contact points from getting invalid contact normals. The contact normal calculated with the distance field's gradient can sometimes point in the opposite direction of the collision. This usually happens at boundary regions of the mesh, i.e. the corners of the model. This is only a minor issue since it only happens to some models and only a few contact points get flipped normals. The solver can still produce a valid collision response even with a few faulty contact points as long as the majority of the contact points have valid contact normals. It does however still lower the quality of the collision detection so the sampling normals have been added to solve this problem. Before a contact point is added is it first checked if the calculated contact normal points in the same direction as the sampling point's normal. If that is the case is the contact normal faulty and the contact point should not be added to the simulation. This method has proven effective in solving the problem and eliminates all contact points with faulty normals.

6.2.5 Initialization of Distance Field Grid

The initialization of the distance field grid is handled by running an OpenCL kernel on the GPU as mentioned in Section 6.2.1. The kernel needs parts of the information stored in the triangle hierarchy in order to calculate the grid's distance values. It needs the triangles vertices and indices along with the normals calculated for the triangle features. So before the kernel can be run must first this data be extracted from the triangle hierarchy and converted to a format supported by OpenCL.

The 3 dimensional vectors used by the application (for example to store positions and normals) are represented as AgX vectors (`Vec3` for a 3D vectors of floats, `Vec3i` for a 3D vectors of integers) and have to be converted to OpenCL vectors before they can be transfered to the GPU. There exists no 3 dimensional vectors in OpenCL(because the GPU wants the bytes of all data to be evenly divisible by 4) so all 3 dimensional vectors have to be stored as 4 dimensional vectors on the GPU. The OpenCL data types that will be used to store the 3 dimensional vector on the GPU will be `float4` (for float vectors) and `int4` (for integer vectors). The vectors used to stored the triangle features are converted to arrays. This is because OpenCL does not support any list types and needs data either to be stored as arrays or as images.

The `Triangle` objects in the hierarchy are converted to `int4`s which consists of the indices in the `Vertex` vector where a triangle's vertices can be found. The hierarchy's `Vertex` objects are converted to `float4`s consisting of the vertices positions. The edges in the `Edge` vector are not needed by the kernel because the edges of a triangle can be derived from the triangle's indices. It is therefore no need to restructure the `Edge` vector and send it to the GPU.

The triangle features normals are extracted from the hierarchy and stored in an array of `float4`s. The triangles face normals are stored first in the array, followed by the edges normals and last the vertices normals. The normals are stored in this way so that the normals of one feature type can be differentiated from the normals of an other feature type. It is possible later, by knowing the amounts of triangles, edges and vertices to find the normal of a certain feature.

Here follows a table displaying the structure of the triangle hierarchy's data when it has been re-structured:

Triangles						
i_{11}	i_{12}	i_{13}	...	i_{n1}	i_{n2}	i_{n3}

Vertices		
v_1	...	v_n

Normals								
n_{t1}	...	n_{tn}	n_{e1}	...	n_{em}	n_{v1}	...	n_{vk}

Table 6.1: The structure of the input data to the distance calculation kernel.

After the data needed from the triangle hierarchy has been properly restructured is a `OclDistFieldInit` object made. The `OclDistFieldInit` object sets up the grid kernel on the GPU and initiates the necessary components on the CPU side for running the kernel, like the command queue and the output buffers. The triangle hierarchy data is sent to the `OclDistFieldInit` object along with information about the grid that is to made like the grid's dimensions and the min and max point of the distance field's AABB. The `OclDistFieldInit` object then transfers the triangle hierarchy data to the GPU and executes the kernel.

The kernel works by starting up a thread (work-item) for every cell of the grid. Each thread then calculates the distance between its cell and each of the mesh's triangles (by using the distance test described in Section 4.4.1) and finds the smallest distance to the mesh and makes this value the cell's distance value. The grid that the kernel produces is a one dimensional array of floats. The reason the produced grid is not 3 dimensional is that OpenCL does not support pointers to pointers and can only handle one dimensional arrays. It is however possible to simulate a 3 dimensional array by knowing the grid's dimensions

so this is not an issue.

6.2.6 Caching of Distance Fields

It can take a substantial amount of time to generate a distance field for a complex mesh, even with GPU acceleration. An utility for caching distance fields was therefore implemented to alleviate this problem. This caching utility takes a generated distance field and saves its grid and sampling points (with corresponding normals) on a file. When the same mesh is being processed again in an other execution of the application does the application instead of making a new distance field just load the cached distance field from the mesh's distance field cache file.

A mesh can have several distance fields with different configurations saved in its cache file. Fields can for example have different dimensions for their grid, different field or grid thickness (if the thickness parameters are used). A mesh only loads a cached distance field if it has the appropriate configuration for the current execution of the application and constructs a new distance field if it can't find such a field in the cache file. The application automatically checks if a mesh has a cache file and creates one if none can be found.

A disadvantage with caching distance fields is that the generated cache files can become quite large (tens of MBs of data) especially if multiple distance field configurations are saved. This is however an acceptable drawback considering that modern computers usually have hundreds of GBs or more of hard-disk space

The .dcache File Format

A file format called .dcache was constructed for storing distance fields. A .dcache file contains a number of distance field configurations along with the distance values of the fields grids and the fields sampling points with their respective normals. A .dcache file has the following format:

```
DISTCACHE %Identifies the beginning of a new block of distance field information.
Number of Grid Cells     Grid width     Grid Height     Grid Depth     Field Thickness
Grid Thickness
d1
...
dn
Number of Sampling Points
s1 snormal1
...
sn snomaln
```

Figure 6.3: The .dcache file format.

A .dcache file can store any number of distance fields structured in this manner with each new distance field being preceded with DISTCACHE.

The reason there is a separate entry for the number of grid cells when the number of grid cells can be calculated by taking $gridWidth \cdot gridHeight \cdot gridDepth$ is because not all grid cells in the grid might have a distance value defined. It is possible to sometimes prune cells depending on the application of the distance field if a cell does not contribute to what the distance field is being used for. This can help to reduce the computation time. In the

case of using distance fields for collision detection can cells a considerable distance from the mesh surface be pruned since the cells of interest are cells that are inside or on the surface of the mesh, cells which can be used to detect a collision with the mesh.

The current implementation does however calculate a distance value for every cell of the grid so, $numGridCells = gridWidth \cdot gridHeight \cdot gridDepth$ does apply. This was however not the case in earlier versions of the application (and might not be the case in future versions).

The "field thickness" and "grid thickness" parameters are currently not used, but were in earlier versions of the application. They were used in a scheme to minimize the amount of cells that needed to be processed. In this scheme were every triangle of the mesh encompassed by its own AABB and only cells that were inside some triangle's AABB had a distance value computed.

The "field thickness" was a parameter of how much each triangle's AABB should be expanded from the triangle AABB's default size. The default size of a triangle AABB was the smallest size were the AABB enclosed the entire triangle. The "field thickness" parameter was used to expand the size of the AABB by increasing the distance between the max and min point of the AABB by the "field thickness".

The "grid thickness" was a parameter of how much to increase the size of the AABB enclosing the distance field. The default size of this AABB is the smallest size in which the entire mesh is enclosed by the AABB. The "grid thickness" parameter was used to expand the size of the distance field's AABB the same way "field thickness" was used to expand the triangle AABBs sizes.

A future improvement of the application would be to use the aforementioned scheme once again and thereby decrease the amount of cells needed to be taken into consideration. If that were to happen would the grid and field thickness parameters be needed again so they are kept for the time being.

6.3 Collision Detection

Support for distance field assisted collision detection was integrated into AgX. Collision detection in AgX is accomplished with collision detection routines called "colliders". A collider is a collision test between a pair of collision primitives. Each collider is its own class and is called statically by the application. A collider can be described with this schematic figure:

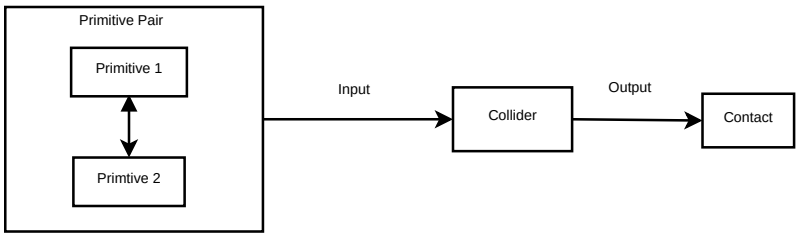


Figure 6.4: A collider routine. A collider takes a pair of primitives as input and returns a "contact" which is a set of contact points.

To implement support for collision detection with distance fields were colliders that compares distance fields against other collision primitives constructed. The primitive types for which colliders were implemented were the ones that it was deemed most likely that

a distance field would encounter. More primitive types will need to be supported before the implemented distance fields can be used professionally, but this amount of colliders was deemed adequate for this thesis.

6.3.1 Distance Field Distance Field Collider

Collision detection between a distance field and an other distance field is handled by the `DistancefieldDistancefieldCollider` class.
`DistancefieldDistancefieldCollider` is given a pair of distance field primitives (`DistanceField` objects) as input. `DistancefieldDistancefieldCollider` is also given the distance fields transformation matrices. A primitive's transformation matrix is used to place the primitive in space and transfer it to the world frame. The collider uses the distance fields transformation matrices to transfer the distance fields to the same coordinate system. This is required before the distance fields can be tested against each other for collisions.

The distance fields are tested against each other two times. Each field's sampling points are tested against the other field's distance grid. When a sampling point is tested against a distance field's grid is it first checked if the sampling point lies inside the distance field's AABB and thereby is inside the confines of the grid. If the sampling point is inside the grid is it checked which cell in the grid that the sampling point lies in and if that cell has a positive distance value. If the cell has a positive distance value is the sampling point not inside the other mesh and no collision has occurred. If the distance value is negative on the other hand has a collision occurred and a contact point should be generated. A contact point consists of the sampling point itself, a collision normal and the collision's penetration depth.

The collider begins with calculating the contact normal. The contact normal is computed by numerically approximating the gradient of the sampling point cell's distance value. This is done by using the finite difference approximation:

$$\mathbf{n} = \{dx, dy, dz\} = \left\{ \frac{c_{x+1,y,z} - c_{x,y,z}}{l_{cell_x}}, \frac{c_{x,y+1,z} - c_{x,y,z}}{l_{cell_y}}, \frac{c_{x,y,z+1} - c_{x,y,z}}{l_{cell_z}} \right\}.$$

The interval used for calculating the gradient is dependent on the position of the sampling point. The collider tries to calculate the gradient over as small interval as possible to achieve the best possible precision. This is accomplished by calculating the difference of the distance value of the cell that the sampling point lies in and the closest neighboring cell in relation to the sampling point.

After the contact normal has been calculated is it compared against the sampling point's normal. If the contact normal is found to be pointing in the same direction as the sampling normal is the contact point not added to the contact. This is because the calculated contact normal is then not pointing in the direction of the collision.

The collider then proceeds with calculating the collision's penetration depth. This is accomplished by getting the absolute value of the distance value of the cell that the sampling point lies in.

After the penetration depth has been calculated is the contact point (with all of its components) added to the contact.

It would be quite easy to implement the distance field-distance field test as an OpenCL kernel. There are no data dependencies between a distance field's sampling points and the sampling points can be processed independently of each other. This kernel could be structured almost exactly like the CPU version of the test. The kernel would however not

need the for-loop that processes each sampling point since each sampling point would be processed by a separate thread.

The collider does not use such a kernel since it would not benefit the performance of the application. No more than a few thousand sampling points are ever made for a mesh and it is more common for a mesh to have a couple of hundred sampling points. A kernel would not benefit the collider's performance with such a small amount of input data. Kernels comes with quite much overhead during their setup and therefore requires a relatively big data set to work on before they start to pay off. It is therefore more suitable to run the collision test on the CPU.

6.3.2 Distance Field Plane Collider

Collision detection between a distance field and a plane is handled by the `DistancefieldPlaneCollider` class. `DistancefieldPlaneCollider` is given a distance field and a plane primitive as input along with their respective transformation matrices. `DistancefieldPlaneCollider` uses the transformation matrices to move the plane to the distance field's coordinate system. The collider tests each of the distance field's sampling points against the plane. This is done by calculating the signed distance between the plane and the sampling point and negating the result. If the calculated distance is greater than or equal to zero has a collision occurred and the sampling point should be added as a contact point. The contact normal for this contact point becomes the planes normal and the penetration depth becomes the calculated distance.

6.3.3 Distance Field Line Collider

Collision detection between a distance field and a line is handled by the `LineDistancefieldCollider` class. `LineDistancefieldCollider` is given a distance field and a line primitive as input along with their respective transformation matrices. `LineDistancefieldCollider` uses the primitives transformation matrices to move the line to the distance field's coordinate system.

`LineDistancefieldCollider` then tries to clip the line against the distance fields bounding box using Cohen-Sutherland line clipping algorithm (included in the appendix). The line is clipped to make sure that the line's end points are inside the distance field's grid. After the line has been clipped does the collider get the cells that the lines endpoints lies in and calculates the difference between the indices of the endpoints cells. This difference is used to determine the slope of the line.

The collider then iterates through each cell between the line's endpoints and checks if any cell has a negative distance value. If a cell has a negative distance value does the line cross through the distance field's mesh at that point and a collision has occurred. A contact point is then added to the contact with the cell's center point used as the point, the distance vector between the cell's center point and the line endpoint that the line stepping algorithm is going to as the contact normal and the absolute value of the cell's distance value as the penetration depth. The collider stops iterating through the lines cells and stops executing after the first cell with a negative distance value has been found. The contact therefore always contains one contact point at most.

The collider handles the stepping through the line's cells with Bresenham's line drawing algorithm (can be found in the appendix). Bresenham's line drawing algorithm is normally used for drawing lines of pixels, but can also be used here where the pixels are exchanged for grid cells. The collider steps through the line's cells by iterating through the cells that

Bresenham's algorithm draws. Bresenham's algorithm works by trying to draw the pixels that best fits the direct line between the line's two endpoints.

Chapter 7

Results

The result of this thesis is software adding support for distance fields and distance field based collision detection to the AgX physics engine. Several simulations using the implemented distance fields were also created with the help of AgX's simulation utilities.

7.1 Screenshots

Several screenshots were taken displaying the various components of the developed software in action.

7.1.1 Distance Field

Figures 7.1 to 7.3 visualizes a distance field for an object. A distance field's grid's cells are rendered as points. The colors of the cells reflects their distance values. A cell's color goes from blue to green to red depending on how close the cell is to the object. When a cell is inside the object does the cell's color shift from pink to yellow to cyan depending on how far the cell is from the object's surface.

- **Figure 7.1** displays the object without any distance field.
- **Figure 7.2** displays the object with a distance field.
- **Figure 7.3** displays the object with a distance field while only displaying cells that have negative distance values i.e. are inside the object.

7.1.2 Mesh sampling

Figure 7.4 and figure 7.5 displays an object with its sampling points.

- **Figure 7.4** displays the object with its sampling points.
- **Figure 7.5** only displays the object's sampling points.

7.1.3 Collision detection

Figure 7.6 to 7.8 are figures displaying collisions between distance fields and other primitives.

- **Figure 7.6** displays a series of collisions between two triangle meshes, both having distance fields. The smaller mesh is dynamic and affected by physical forces (like gravity) while the larger mesh is static and locked in its initial position and not affected by physical forces.
- **Figure 7.7** displays a collision between a distance field and a line. The distance field's mesh has a line going through it and the first penetrating point on the line is made into a contact point.
- **Figure 7.8** displays a series of collisions between a triangle mesh with a distance field and a plane.

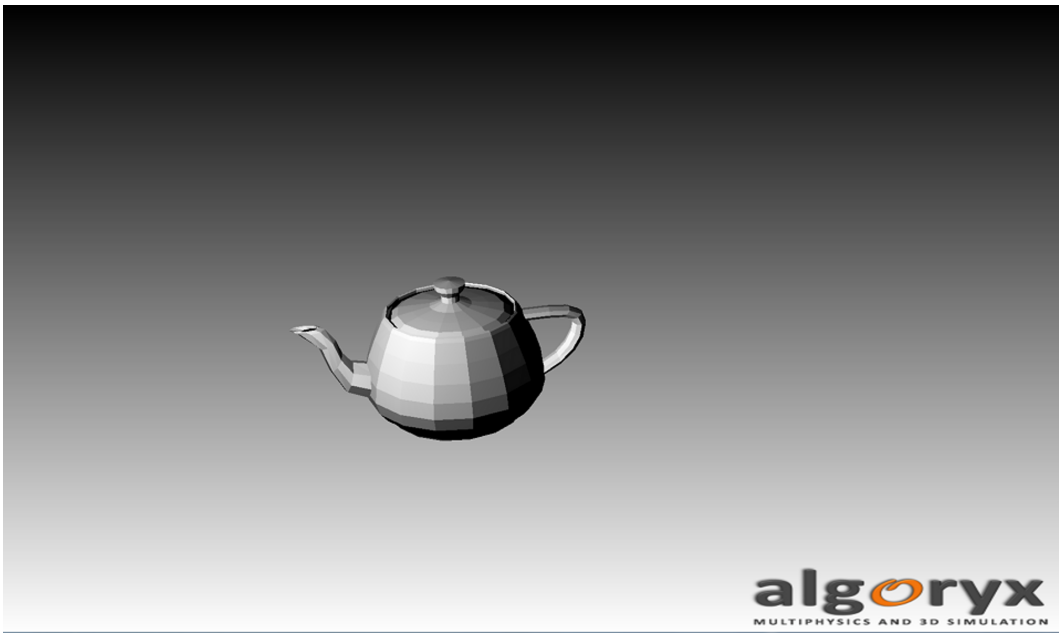


Figure 7.1: The Utah teapot.

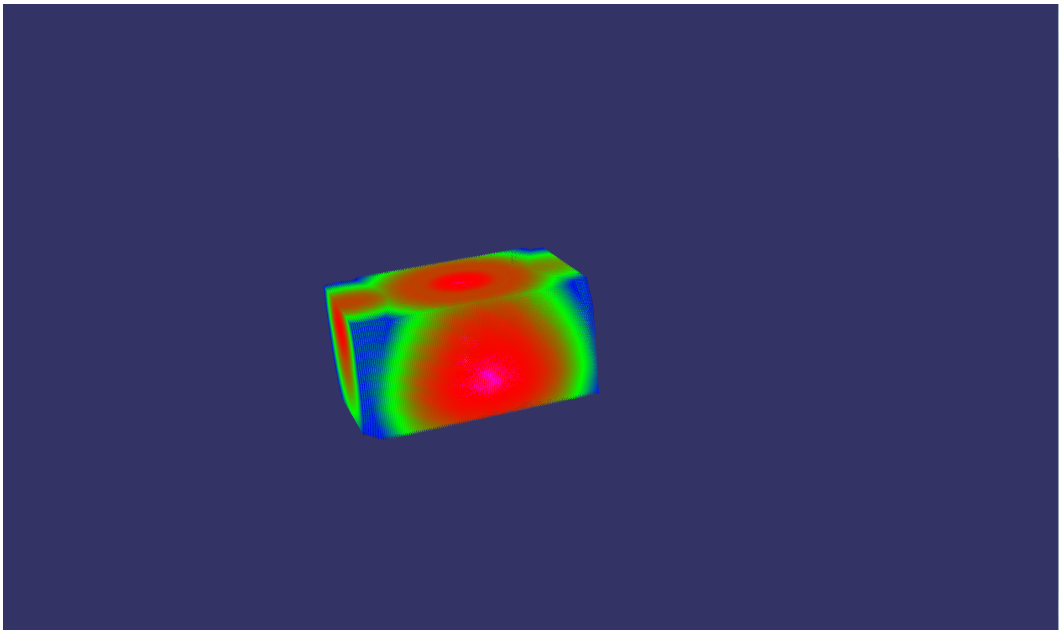


Figure 7.2: The teapot with a distance field.

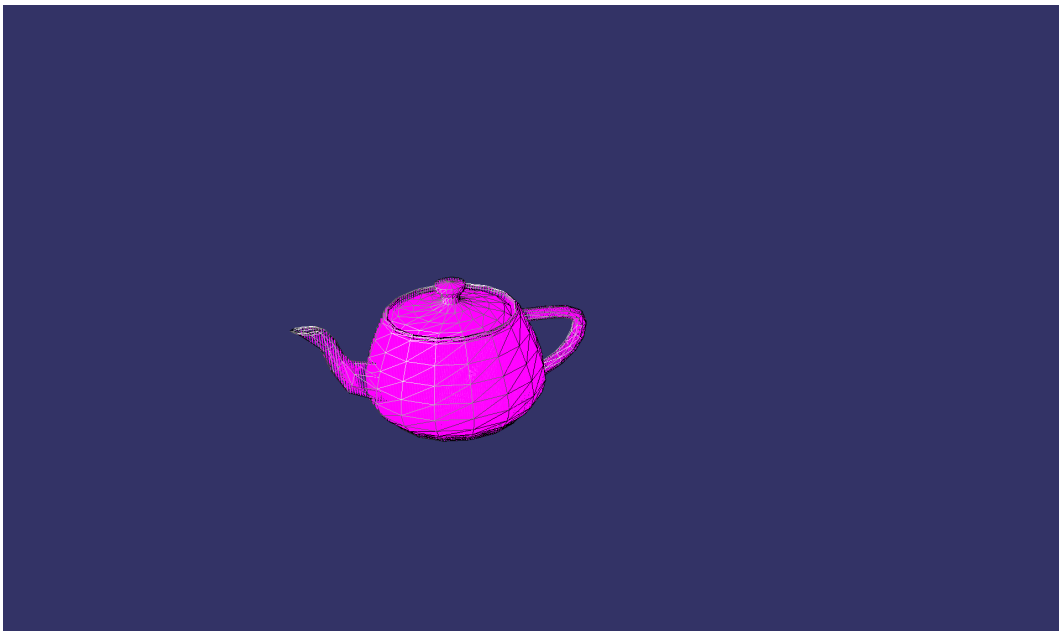


Figure 7.3: The teapot with a distance field where only cells with negative distances are shown.

Figure 7.4: The teapot with its sampling points. Some points are occluded by the objects surface.

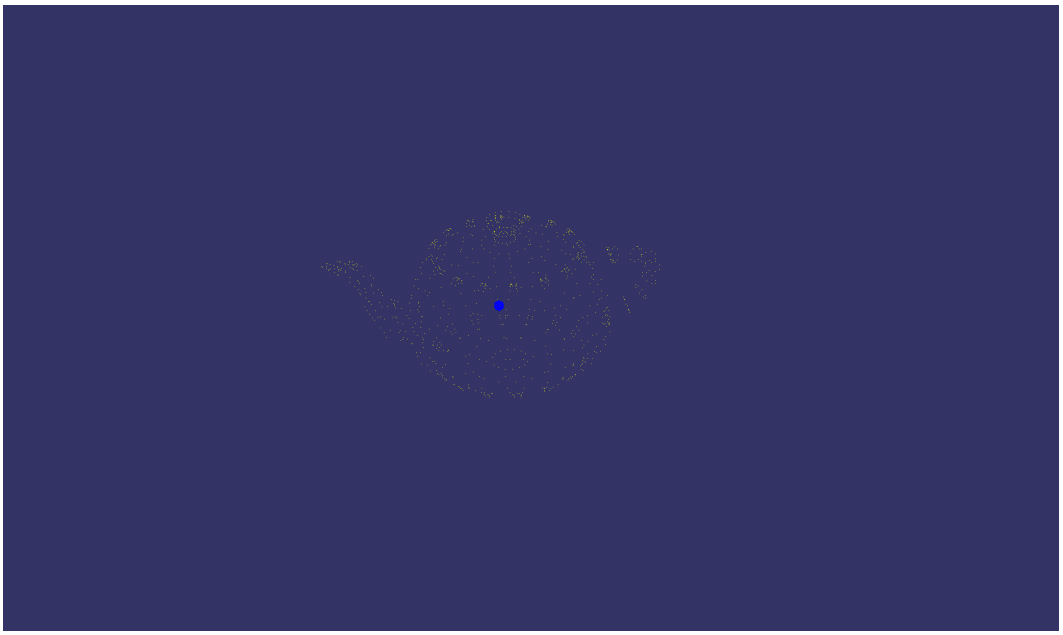
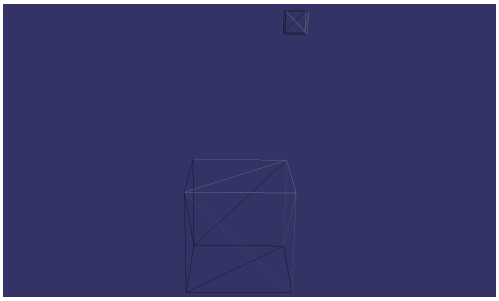
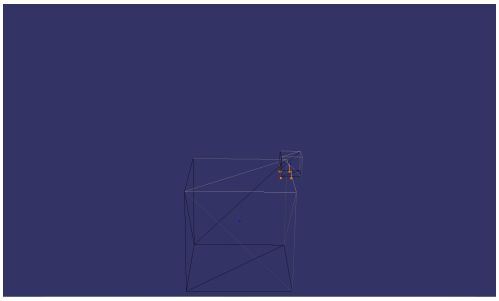


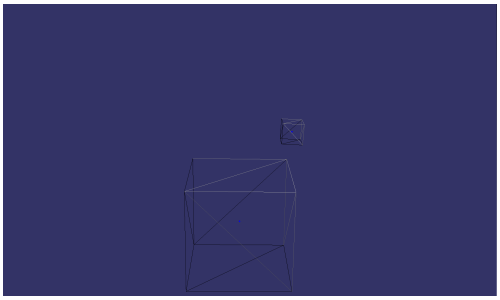
Figure 7.5: A figure only displaying the teapots sampling points. One can clearly see here that the entire mesh gets sampled.



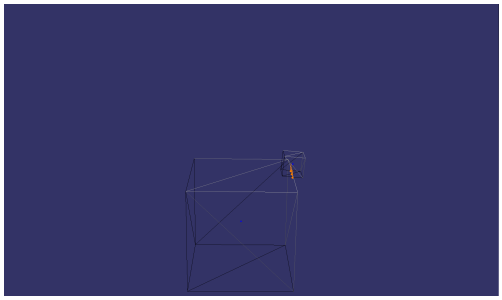
(a) The initial scene



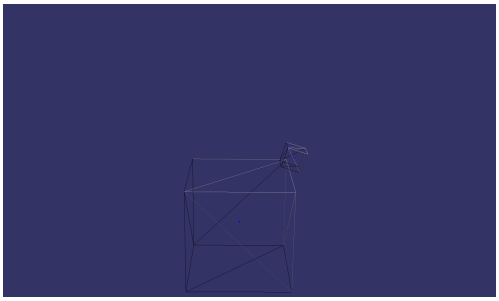
(b) The first collision



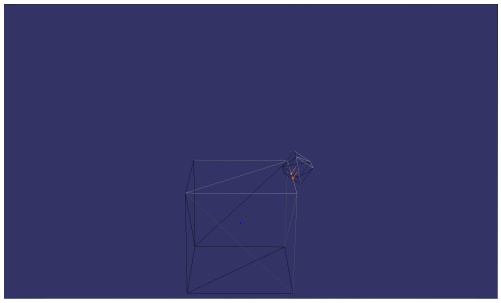
(c) After the collisions upward thrust has been canceled out



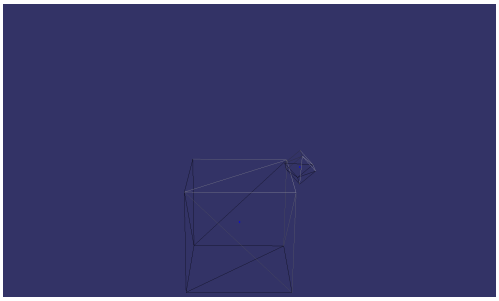
(d) The second collision



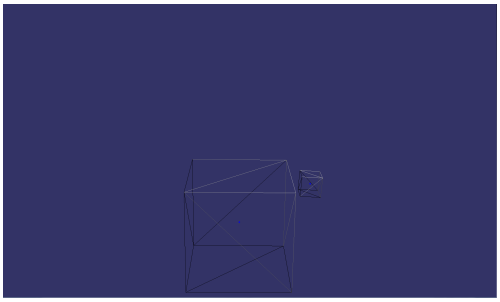
(e) After the collisions upward thrust has been canceled out



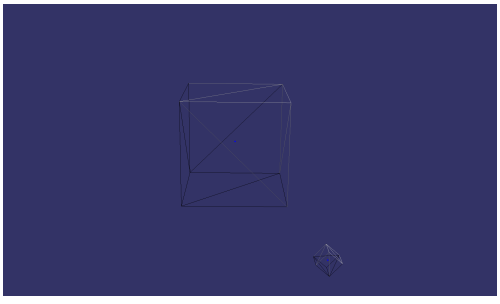
(f) The third collision



(g) After the collisions upward thrust has been canceled out



(h) The smaller mesh has begun to clear the larger mesh



(i) The smaller mesh has cleared the larger mesh

Figure 7.6: A series of collisions between two distance fields.

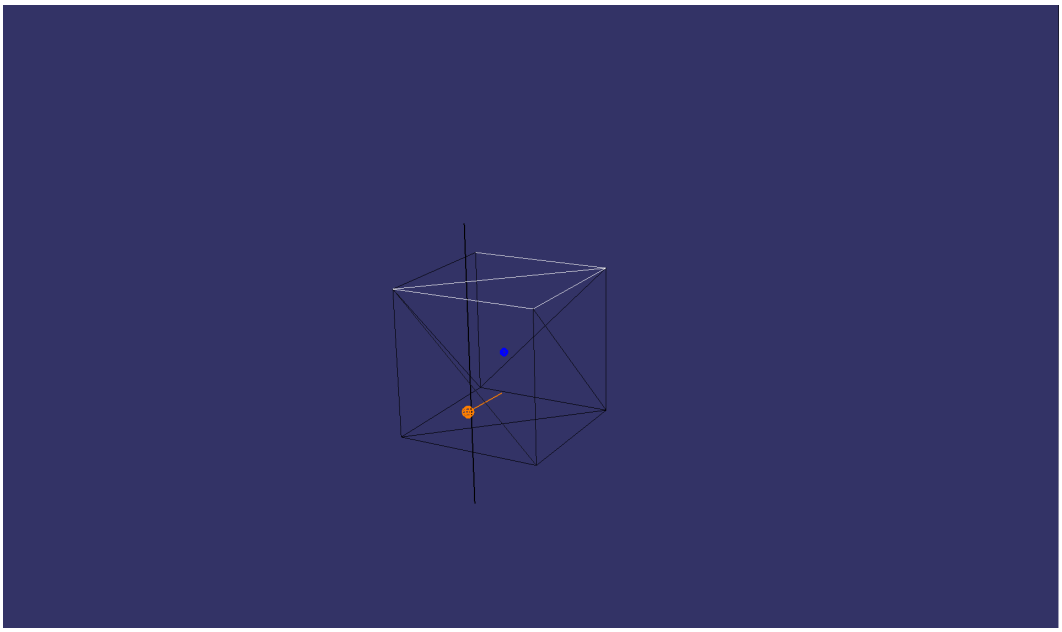
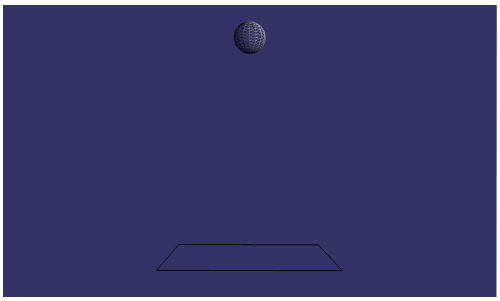
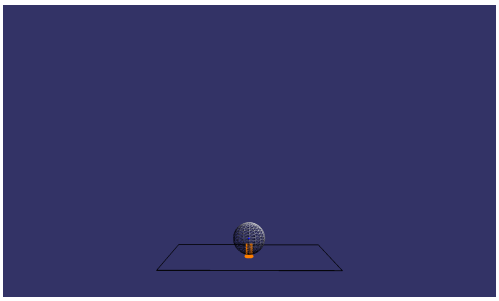


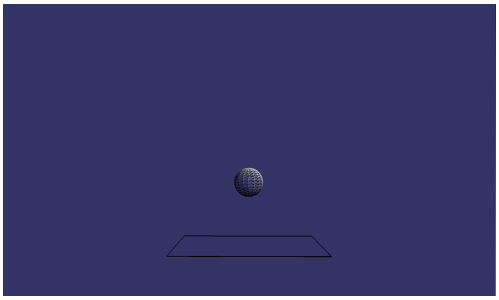
Figure 7.7: A collision between a distance field and a line.



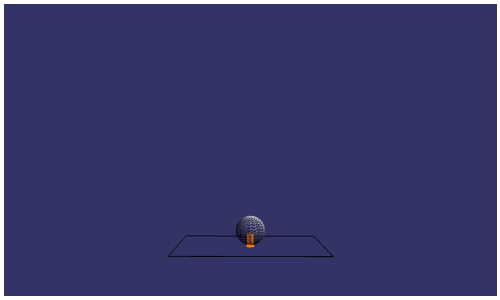
(a) The initial scene



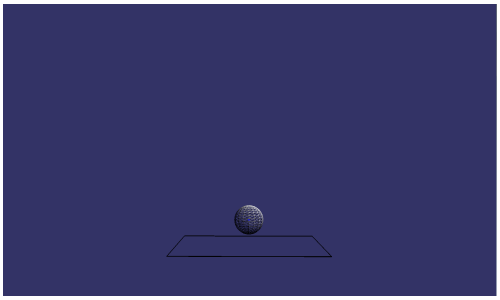
(b) The first collision



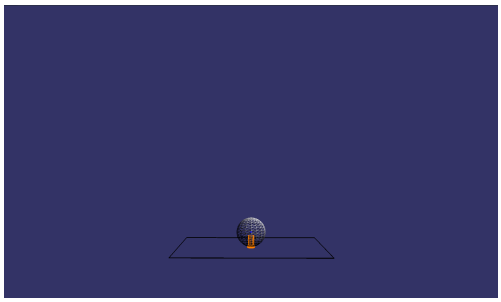
(c) After the collisions upward thrust has been canceled out



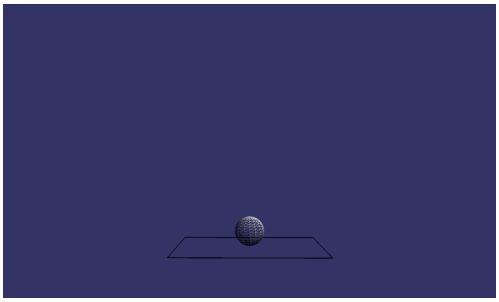
(d) The second collision



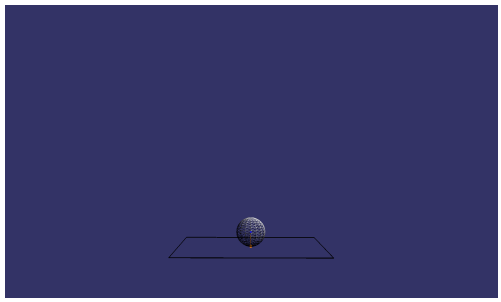
(e) After the collisions upward thrust has been canceled out



(f) The third collision



(g) After the collisions upward thrust has been canceled out



(h) The triangulated sphere has become resting

Figure 7.8: A series of collisions between a distance field and a plane.

7.2 Performance of Distance Field Initialization

It was anticipated that using the GPU for initializing the distance field's grid would have a performance superior to the performance of using the CPU for the same task. This hypothesis was tested by making a CPU implementation of the grid initialization function and comparing its performance against the performance of the GPU implementation of the function. It turned out that the task of initializing the distance field grid is very well suited for being processed by the GPU and a substantial speed-up can be gained by running the grid initialization as a GPU kernel. Below follows a table displaying the CPU and the GPU functions execution times for different models.

Model	N_t	Time CPU	Time GPU
Triangulated box with 2 triangles per side	12	2.09s	1.62s
Cone	64	8.70s	2.38s
Cylinder	127	18.29s	2.71s
Torus	399	52.61s	4.98s
Triangulated box	768	1min 29.42s	6.85s
Teapot	992	1min 54.86s	9.91s
Highly triangulated sphere	2499	8min 6.18s	40.33s
Robot	8454	17min 19.57s	1min 2.35s
Armadillo 3	27674	-	3min 16.03s

Table 7.1: Time required by the CPU and the GPU functions for making a distance field with a grid with the dimensions 128x128x128 for a variety of different models.

The execution times are about equal for models with very few triangles, but the GPU is significantly faster for models with large amounts of triangles. The advantage of using the GPU function over the CPU function grows with the amount of triangles a model has. This is visualized in Figure 7.9 and 7.10.

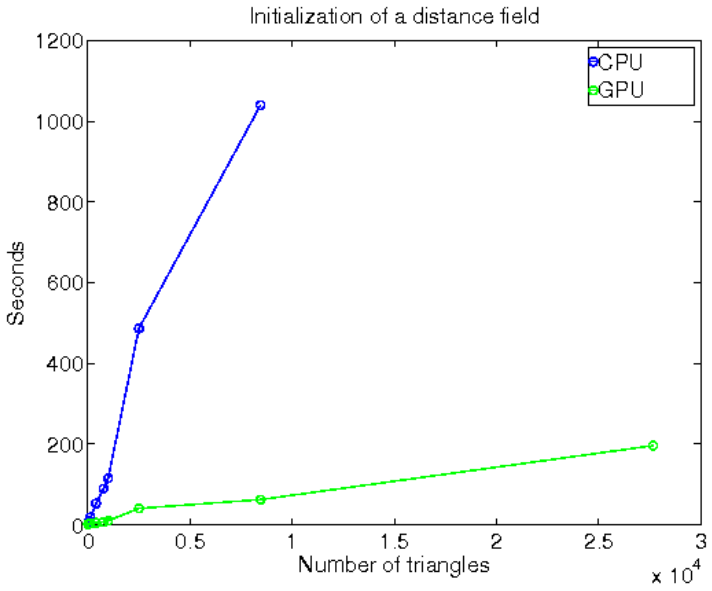


Figure 7.9: A graph displaying the difference in required time between a distance field initialization routine implemented on the CPU and on the GPU.

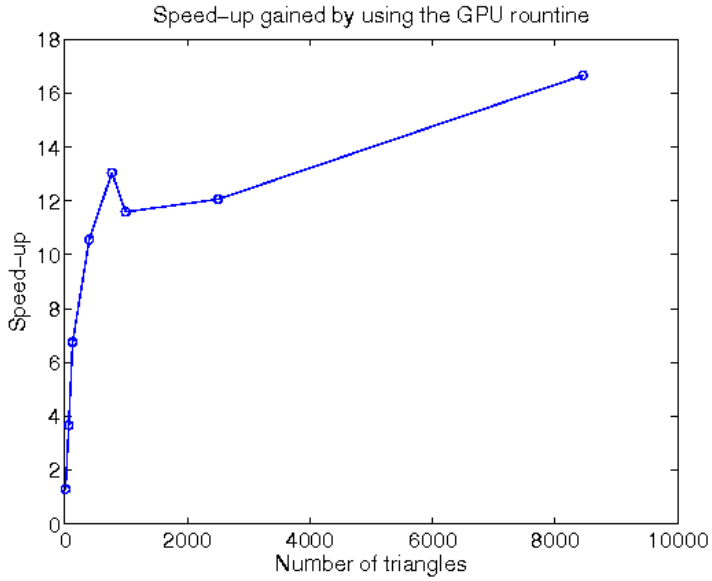


Figure 7.10: A graph displaying the speed-up of using the GPU routine as opposed to the CPU routine.

Chapter 8

Discussion and Conclusions

Most of the goals of this thesis have been realized. A GPU accelerated distance field implementation and software for using the implemented distance fields for collision detection have been developed successfully. Some problems were encountered during the project that slowed down progress, but these issues were all eventually resolved. The initial design of the software proved mostly sound, but some changes to the design were deemed necessary as more knowledge of the subject area was acquired. An example of a change in the design is not using a kernel for handling the distance field–distance field collision detection. This was initially the plan, but the idea was dropped when it was discovered that such a kernel would not bring any substantial benefits. It was possible to mostly follow the project plan made at the start of the project, but the project could not be finished at the aimed stop date due to unforeseen problems.

8.1 Limitations

The developed software have some limitations:

The collision detection routine used for detecting collisions between pairs of distance fields overestimates the penetration depth of computed contact points. It gives an approximation of the penetration depth that is mostly right, but is slightly off. This makes the application apply a bit larger collision impulse than necessary which adds energy to the simulation. This makes it harder for objects to come to rest and can make objects at rest shake very slightly.

Another limitation of the application is that colliders have not been implemented for most primitive types. It was however never part of the plan for the thesis to implement colliders for anything other than the distance field-distance field and distance field-plane case, so this limitation was excepted.

8.2 Future Work

A future improvement of the distance field implementation would be to implement some method for decreasing the amount of cells that needs to have distance values computed during the initialization of the distance field grid. The application currently calculates distance values for all the cells of the grid; this is unnecessary since only a portion of the cells in the grid are ever relevant to the collision detection. A method for decreasing the

amount of cells needed to be taken into consideration was used in an earlier version of the application. This feature had to be removed because it was hard to make the method work with the grid kernel. One could in the future try to fix the compatibility issues of this method or try to design an entirely different method for decreasing the workload.

Another future project would be to fix the problem with the calculation of the penetration depth, mentioned in the previous section. One could try to fix this either by improving on the already used method for computing the penetration depth, which is by using the distance value of the cell that the contact point lies in or implement an other method for computing the penetration depth. A way to improve the current method would be to adjust the used distance value with the distance from the contact point to the cell's center point. This would likely produce a better approximation of the penetration depth.

The implemented distance field software could also be improved by adding support for additional primitive types.

The application could also be improved if an efficient kernel for generating contact points for distance field-distance field collisions could be constructed. A contact point generation kernel for distance field-distance field collisions would not contribute to the applications performance (as described in Section 6.3.1) with the way collision detection is handled in AgX as of now (with colliders processing pairs of primitives individually). For such a kernel to be worthwhile would the kernel need to process several pairs of distance fields simultaneously. To accomplish this would the processing of distance field pairs have to be changed. Instead of passing each distance field pair individually to the distance field-distance field collider would all generated distance field pairs have to be stored in a vector with the vector later being passed to the collider. With the contact generation kernel handling all distance field pairs at once would the kernel have enough data to process for it to be worthwhile. This project would involve quite a lot of work, but the project's potential benefits would be large. If the contact point generation could be handled effectively by a GPU kernel would the distance field-distance field collision detection receive a significant speed-up.

8.3 Conclusion

Some conclusion were made during the course of the project.

It is possible to receive a large performance increase by letting the GPU handle the calculations related to the initialization of a distance field. The initialization of the distance field grid is a very data parallel problem that is very suitable for being processed on the GPU.

It was discovered that in order for distance field based collision detection implemented on the GPU to be worthwhile must multiple distance fields be processed on the GPU simultaneously.

Distance fields can represent a wide range of objects accurately and correctly detect collisions for these objects. The contact data extracted from distance fields can however in some cases be slightly off because of distance field based collision detection being an approximate method.

Chapter 9

Acknowledgments

I would like to thank a number of people who helped me during this project.

First, I would like to thank my supervisors: Eddie Wadbro at the university and Kenneth Bodin at Algoryx. Eddie provided me with continuous feedback and always had time for a meeting. His input on GPGPU and other subjects were very helpful and his feedback on the report was invaluable. Kenneth always made sure I had all the help I needed and always provided assistance when problems occurred.

I would like to thank all of Algoryx's employees for being so helpful and making me feel welcome. A special thanks goes to Michael Brandl at Algoryx for providing me with help on a daily basis and always taking the time to answer my questions about collision detection and various other subjects. I would also like to thank Nils Hjelte at Algoryx for answering all my questions about AgX and always lending me help when I needed it. Thanks also goes to Martin Nilsson at Algoryx for answering all of my questions about OpenCL and for teaching me the proper way to design kernels. And I would also like to thank Emil Ernerfeldth at Algoryx who had previously implemented distance fields for providing me with his insights on the subject and answering all of my questions.

I would like to give a special thanks to my family for being so supportive and always being there for me.

Lastly I would like to dedicate this thesis to my grandfather Lennart Sundholm. You will be missed.

Chapter 10

Terminology

- **AABB (Axis Aligned Bounding Box):** An axis-aligned box encompassing all the vertices of a graphical object.
- **Compute Unit:** A compute unit resides on a OpenCL device. It can for example be a core on a multi-cored CPU or a Streaming Multiprocessor(SM)/SIMD engine on a GPU. A compute unit consists of one or more processing elements.
- **Device:** A device is a processor capable of performing floating point calculations and is typically either a GPU or a CPU, but can also possibly be a NPU.
- **Contact:** A set of contact points.
- **Contact point:** A point where an object penetrates an other object. A contact point consists of the actual point, the penetration depth and the contact normal.
- **Contact normal:** The direction in which two object are colliding.
- **Distance field:** A distance field is a scalar field that specifies the minimum distance to a shape.
- **Global ID:** Each work-item has a global ID. A work-item's global ID gives the work-items position in index space.
- **GPU (Graphics Processing Unit):** A processor dedicated to graphics processing.
- **GPGPU (General-Purpose Computing on Graphics Processing Units):** The use of graphics hardware for non-graphics related calculations.
- **Group ID:** A work-groups group ID gives the groups position in index space.
- **Host:** The host is the machine that hosts the devices accessible by a OpenCL kernel. It is typically a desktop computer.
- **Kernel program:** A kernel program is a collection of kernel functions which gets executed on a contexts devices.
- **Kernel function:** A kernel function is a parallel function which executes by running instances of itself on a devices processing elements.

- **Local ID:** Each work-item has a local ID. A work-items local ID gives the work-items position in its work-group.
- **2-manifold:** A 2-manifold is a surface with no holes or edges crossing each other which unambiguously defines an inside and outside.
- **OpenCL:** OpenCL was developed by Apple (in collaboration with technical teams at AMD, IBM, Intel and Nvidia). OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors.
- **Penetration depth:** The depth of a collision.
- **Processing element:** A processing element is a virtual scalar processor and can for example be a ALU or a streaming processor. It is in processing elements that the calculations issued by a OpenCL application are performed.
- **Pseudo normal:** An approximate normal.
- **Sampling point:** A point generated in the sampling of a triangle mesh. A sampling point is either a vertex of a triangle, a point on a triangle edge or a center point on a flat surface of the mesh. A mesh's sampling points gives a representation of the mesh as a set of points.
- **Triangle face:** The surface of a triangle.
- **Triangle feature:** A triangle feature is a component of a triangle. A triangle feature is either a vertex, edge or the triangles surface.
- **Triangle mesh:** A triangle mesh is a set of triangles that are connected by their common edges or corners.
- **Voronoi region:** A region around an object where every point in the region is closer to the object than any other object in space.
- **Work-item:** A work-item is a instance of a kernel function.
- **Work-group:** A work-group is a group of work-items.

Bibliography

[1] Khronos group website, visited 2010-04-26: http://www.khronos.org/opencl/ 2010.

[2] Wikipedia OpenCL, visited 2010-03-25: http://en.wikipedia.org/wiki/OpenCL 2010-03-23.

[3] Algoryx website, visited 2010-04-21: http://www.algoryx.se/ 2008.

[4] Algodoo website, visited 2010-04-25: http://www.algodoo.com/wiki/Home 2009.

[5] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha: Difi: Fast 3d distance field computation using graphics hardware, EUROGRAPHICS 2004, Volume 23 (2004), Number 3. 2004.

[6] A. Rosenfeld and J.L. Pfaltz: Sequential Operations in Digital Picture Processing, J.ACM, Volume 13, Number 9, pages 623-628. 1966.

[7] J. Andreas Baerentzen and Henrik Aanaes: Generating Signed Distance Fields From Triangle Meshes, Technical Report, IMM. 2002.

[8] Kenny Erleben and Henrik Dohlmann: GPU Gems Chapter 34. Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra 2007.

[9] Kenny Erleben: Stable, Robust, and Versatile Multibody Dynamics Animation, Ph.D thesis. April 2005.

[10] John Owens: GPU Gems 2 Chapter 29. Streaming Architectures and Technology Trends. 2007.

[11] David Kirk and Wen-mei Hwu: CUDA Textbook, Draft. 2008.

[12] Nvidia: CUDA Programming Guide 2.3. 2009-08-26.

[13] Khronos group website ,visited 2010-03-26: http://www.khronos.org/ 2010.

[14] Wikipedia CUDA, visited 2010-03-25: http://en.wikipedia.org/wiki/CUDA 2010-03-23.

[15] AMD website, visited 2010-03-25, A Brief History of General Purpose (GPGPU) Computing: http://ati.amd.com/technology/streamcomputing/gpgpu_history.html 2009.

[16] Geek3D website, visited 2010-03-27, [TEST] GPU Computing GeForce and Radeon OpenCL Test (Part 1): http://www.geeks3d.com/20100115/gpu-computing-geforce-and-radeon-opencl-test-part-1/ 2010.

[17] Khronos: OpenCL specification version 1.0, 2009-06-09.

[18] AMD website, visited 2010-03-29, An Introduction to OpenCL: http://ati.amd.com/technology/streamcomputing/intro_opencl.html 2009.

[19] Kent Roger B. Fagerjord and Tatyana V. Lochehina:GPGPU: Fast and easy Distance Field computation on GPU. 2008.

[20] Mark W. Jones, J. Andreas Baerentzen, and Milos Sramek: 3D Distance Fields: A Survey of Techniques and Applications, IEEE Transactions on Visualization and Computer Graphics, Volume 12, Number 4. 2006.

[21] Bradley A. Payne and Arthur W. Toga: 3D Distance Fields: Distance field manipulation of surface models, IEEE Computer Graphics and Applications, pages 65-71. 1992.

[22] Sean Mauch: A fast algorithm for computing the closest point and distance transform, Technical report. 2000.

[23] MathWorld Voronoi Diagram, visited 2010-04-12: http://mathworld.wolfram.com/VoronoiDiagram.html .

[24] Christer Ericson: Real-Time Collision Detection, Elsevier Inc. 2005.

[25] G. Scott Owen, Siggraph website, visited 2010-04-20, Clipping: Sutherland-Cohen line clipping: http://www.siggraph.org/education/materials/HyperGraph/scanline/clipping/clipsuco.htm 2005-03-17.

[26] Mark Feldman, Gamedev website, visited 2010-04-20, Bresenham's Line and Circle Algorithms: http://www.gamedev.net/reference/articles/article767.asp 1999-07-10.

Appendix A

OpenCL Functions

A.1 Distance Test

The distance test presented by Christer Ericsson implemented as an OpenCL function:

```
float calcDistanceToTriangle(float4 point, int index, __local float4 *groupMem) {
            /*Check if the point is in the vertex region outside vertex A*/

            /*Get distance vectors*/
            float4 ab = groupMem[index * 10 + 1] - groupMem[index * 10];
            float4 ac = groupMem[index * 10 + 2] - groupMem[index * 10];
            float4 ap = point - groupMem[index * 10];

            float signDotProd;

            /*Return the distance between the point and vertex A(tri_verts[0])
            if the point is inside As vertex region*/
            if((dot(ab, ap) <= 0.0) && (dot(ac, ap) <= 0.0)) {
                        /*Calculate the sign*/
                        signDotProd = dot(ap, groupMem[index * 10 + 7]);
                        /*Return the signed distance*/
                        //barycentric coordinates (1, 0, 0)
                        return dot(ap, ap) * signNonZero(signDotProd);
            }

            /*Check if the point is in the vertex region outside vertex B*/

            /*Get the distance vector between the point and vertex B*/
            float4 bp = point - groupMem[index * 10 + 1];

            /*Return the distance between the point and vertex B(tri_verts[1])
            if the point is inside Bs vertex region*/
            if((dot(ab, bp) >= 0.0) && (dot(ac, bp) <= dot(ab, bp))) {
                        /*Calculate the sign*/
                        signDotProd = dot(bp, groupMem[index * 10 + 8]);
                        /*Return the signed distance*/
```

```
            //barycentric coordinates (0, 1, 0)
            return dot(bp, bp) * signNonZero(signDotProd);
}

float v;
float4 distVector;

/*Check if the point is in the edge region of AB and return a projection
of the point onto AB if that is the case*/

/*Get (ab * ap) * (ac * bp) - (ab * bp) * (ac * ap)*/
float vc = (dot(ab, ap) * dot(ac, bp)) - (dot(ab, bp) * dot(ac, ap));

/*Return the the distance from the point to a projection
of the point onto AB if the point is inside the edge region of AB*/
if((vc <= 0.0) && (dot(ab, ap) >= 0.0) && (dot(ab, bp) <= 0.0)) {
            /*Calculate v*/
            v = dot(ab, ap) / (dot(ab, ap) - dot(ab, bp));
            /*Get the distance vector*/
            distVector = point - (groupMem[index * 10] + ab * v);
            /*Calculate the sign*/
            signDotProd = dot(distVector, groupMem[index * 10 + 4]);
            /*Return the signed distance*/
            //barycentric coordinates (1 - v, v, 0)
            return dot(distVector, distVector) * signNonZero(signDotProd);
}

/*Check if the point is in the vertex region outside vertex C*/

/*Get the distance vector between the point and vertex C*/
float4 cp = point - groupMem[index * 10 + 2];

/*Return the distance between the point and vertex
C(tri_verts[2]) if the point is inside Cs vertex region*/
if((dot(ac, cp) >= 0) && (dot(ab, cp) <= dot(ac, cp))) {
            /*Calculate the sign*/
            signDotProd = dot(cp, groupMem[index * 10 + 9]);
            /*Return the signed distance*/
            //barycentric coordinates (0, 0, 1)
            return dot(cp, cp) * signNonZero(signDotProd);
}

float w;

/*Check if the point is in the edge region of AC and return a projection
of the point onto AC if that is the case*/

/*Get (ab * cp) * (ac * ap) - (ab * ap) * (ac * cp)*/
float vb = (dot(ab, cp) * dot(ac, ap)) - (dot(ab, ap) * dot(ac, cp));
```

```
            /*Return the the distance from the point to a projection
            of the point onto AC if the point is inside the edge region of AC*/
            if((vb <= 0.0) && (dot(ac, ap) >= 0.0) && (dot(ac, cp) <= 0.0)) {
                        /*Calculate w*/
                        w = dot(ac, ap) / (dot(ac, ap) - dot(ac, cp));
                        /*Get the distance vector*/
                        distVector = point - (groupMem[index * 10] + ac * w);
                        /*Calculate the sign*/
                        signDotProd = dot(distVector, groupMem[index * 10 + 6]);
                        /*Return the signed distance*/
                        //baycentric coordinates (1 - w, 0, w)
                        return dot(distVector, distVector) * signNonZero(signDotProd);
            }

            /*Check if the point is in the edge region of BC and return a projection
            of the point onto BC if that is the case*/

            /*Get (ab * bp) * (ac * cp) - (ab * cp) * (ac * bp)*/
            float va = (dot(ab, bp) * dot(ac, cp)) - (dot(ab, cp) * dot(ac, bp));

            /*Return the the distance from the point to a projection
            of the point onto BC if the point is inside the edge region of BC*/
            if((va <= 0.0) && ((dot(ac, bp) - dot(ab, bp)) >= 0.0)
                        && ((dot(ab, cp) - dot(ac, cp)) >= 0.0)) {
                        /*Calulate w*/
                        w = (dot(ac, bp) - dot(ab, bp)) /
((dot(ac, bp) - dot(ab, bp)) + (dot(ab, cp) - dot(ac, cp)));
                        /*Get the distance vector*/
                        distVector = point - groupMem[index * 10 + 1] +
                       (groupMem[index * 10 + 2] - groupMem[index * 10 + 1]) * w);
                        /*Calculate the sign*/
                        signDotProd = dot(distVector, groupMem[index * 10 + 5]);
                        /*Return the signed distance*/
                        //baycentric coordinates (0, 1 - w, w)
                        return dot(distVector, distVector) * signNonZero(signDotProd);
            }

            /*The point is inside the face region. The distance to the triangle is calculated by
getting a point projected to the triangles plane(done by using barycentric coordinates
and getting (u, v, w)) and calculating the distance to it from the point.*/
            float denom;

            /*Calculate the denominator, v and w*/
            denom = 1.0 / (va + vb + vc);
            v = vb * denom;
            w = vc * denom;

            /*Calculate the distance vector*/
```

```
            distVector = point - (groupMem[index * 10] + ab * v + ac * w);
            /*Calculate the sign*/
            signDotProd = dot(distVector, groupMem[index * 10 + 3]);

            /*Return the signed distance*/
            return dot(distVector, distVector) * signNonZero(signDotProd);
}
```

Appendix B

Algorithms

B.1 Cohen-Sutherlands Line Clipping Algorithm

Algorithm 6 Computes the bit code for a point.

```
1:  procedure ComputeOutCode(x, y)
2:      code :=INSIDE
3:      if y > y_max then
4:          code |= TOP
5:      else if y < y_min then
6:          code |= BOTTOM
7:      end if
8:      if x > x_max then
9:          code |= RIGHT
10:     else if x < x_min then
11:         code |= LEFT
12:     end if
13:     return code
14: end procedure
```

Algorithm 7 Cohen-Sutherlands line clipping algorithm in 2D.

```
1: procedure CohenSutherland(x0, x1, y0, y1)
2:     code0 := ComputeOutCode(x0,y0)
3:     code1 := ComputeOutCode(x1,y1)
4:     while (code0 ≠ INSIDE) ∨ (code1 ≠ INSIDE) do
5:         if (code0 & code1) ≠ INSIDE then
6:             return false
7:         else
8:             if code0 > 0 then
9:                 code := code0
10:            else
11:                code := code1
12:            end if
13:            if (code & TOP) = TOP then
14:                x := x0 + ((x1−x0)(ymax−y0))/(y1−y0)
15:                y := ymax
16:            else if (code & BOTTOM) = BOTTOM then
17:                x := x0 + ((x1−x0)(ymin−y0))/(y1−y0)
18:                y := ymin
19:            else if (code & RIGHT) = RIGHT then
20:                y := y0 + ((y1−y0)(xmax−x0))/(x1−x0)
21:                x := xmax
22:            else if (code & LEFT) = LEFT then
23:                y := y0 + ((y1−y0)(xmin−x0))/(x1−x0)
24:                x := xmin
25:            end if
26:            if code = code0 then
27:                x0 := x
28:                y0 := y
29:                code0 := ComputeOutCode(x0,y0)
30:            else
31:                x1 := x
32:                y1 := y
33:                code1 := ComputeOutCode(x1,y1)
34:            end if
35:        end if
36:    end while
37:    return true
38: end procedure
```

B.2 Bresenham's Line Algorithm

Algorithm 8 Bresenham's line algorithm in 2D.

```
1:  procedure BRESENHAM(x_0, x_1, y_0, y_1)
2:      boolean steep := abs(y_1 − y_0) > abs(x_1 − x_0)
3:      if steep then
4:          swap(x_0, y_0)
5:          swap(x_1, y_1)
6:      end if
7:      if x_0 > x_1 then
8:          swap(x_0, x_1)
9:          swap(y_0, y_1)
10:     end if
11:     dx := x_1 − x_0
12:     dy := abs(y_1 − y_0)
13:     e = dx/2
14:     y := y_0
15:     if y_0 < y_1 then
16:         y_step := 1
17:     else
18:         y_step := −1
19:     end if
20:     for x from x_0 to x_1 do
21:         if steep then
22:             plot(y, x)
23:         else
24:             plot(x,y)
25:         end if
26:         e := e - dy
27:         if e < 0 then
28:             y := y + y_step
29:             e := e + dx
30:         end if
31:     end for
32: end procedure
```