Pomona College
Department of Computer Science

Constructing 3D Distance Fields on GPUs

Joshua Landgraf

May 5, 2016

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Professors Jeff Amelang and Tzu-Yi Chen, advisors

Abstract

Many fields benefit from and rely on being able to find the distance from a point to the surface of a mesh. 3D distance fields provide a quick way to look up this vital information, but have been expensive to compute depending on the approaches and hardware used. This paper presents several methods to improve the efficiency of computing 3D distance fields in parallel, especially on GPUs, and analyzes their performance.

Acknowledgments

I am deeply grateful to Professor Amelang for his advice and support throughout the process of working on my thesis and for his help with developing and implementing my methods. I would also like to thank Professor Chen for her help with managing Pomona's thesis requirements and for her feedback on my thesis.

Contents

List of Figures

List of Tables

Preface

Distance fields are used to solve many different problems in the modern world. They are currently used to represent, modify, and display objects by the computer graphics and volume graphics communities. Computer vision researchers also use 3D distance fields to help process images and robotics researchers have found them useful for planning paths and navigating environments. Distance fields have even been used by physicists to solve Eikonal equations and have become quite popular in medical imaging.

However, it can take a lot of computational power to generate 3D distance fields. Because of this, there has been a significant amount of research on different approaches to computing distance fields efficiently. More recently, researchers have been working to ways to take advantage of highly parallel GPUs to accelerate these calculations. In this paper, we explain what exactly 3D distance fields are (Section 1.1) and cover some of these distance field algorithms (Section 1.2). We will also cover the basics of GPU architecture (Section 1.3) and discuss some previous attempts at adapting distance field calculations for GPU hardware (Section 1.4).

After providing a background for out work, we will present our own methods for efficiently creating 3D distance fields on GPUs. This includes the data structures used by our methods (Section 2.1) as well as the actual approaches to computing the distance fields (Sections 2.2, 2.3, and 2.4).

In addition to explaining our methods, we will also demonstrate some of the results of using out methods (Section 3) and do an in-depth analysis of the performance of our methods (Section 4).

Finally, we will wrap up our paper (Section 5) and discuss potential continuations of our work (Section 5.1).

Chapter 1

Background

In this chapter, we provide a background on what exactly distance fields are (Section 1.1) and some of the major approaches to computing them efficiently (Section 1.2). We also discuss elements of GPU architecture and programming that are relevant to the methods presented in this paper (Section 1.3) and previous work in implementing distance field calculations on GPUs (Section 1.4).

1.1 Distance Fields

A distance field is simply a collection of distances from certain points to a surface. While this may seem like a fairly straightforward definition, there is actually a lot of variability in what exactly constitutes a distance field. For instance, the distances in a distance field can actually be signed or unsigned. While unsigned distances may be easier to compute, signed distances allow for detecting whether a point is inside an object or not, as the distance to a

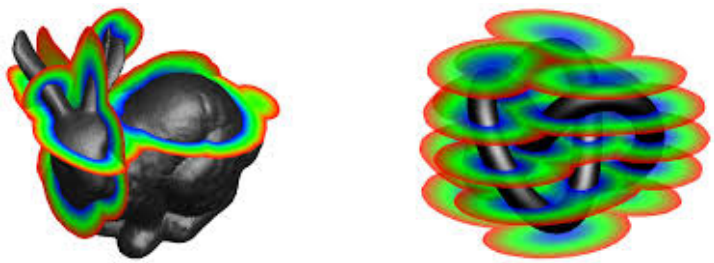


Figure 1.1: Two meshes and slices of their distance fields [PS06]

surface from the inside is negative. The distances in a distance field are also the shortest distance from a point to the surface, not the smallest. This is important when working with signed distances because a negative distance with a greater magnitude than another distance will be smaller, but not shorter. Finally, some distance fields may also contain data on which point on the surface was closest to the starting point.

Distance fields can also take on a variety of dimensions. A distance between just one point and a surface could be considered a zero-dimensional distance field. However, two-dimensional and three-dimensional distance fields are much more useful when working with images and surfaces as they contain more points in higher dimensions. While 2D distance fields are still of interest today (Valve has used them in their Source engine [Gre07]), 3D distance fields involve significantly more computation to generate and are more relevant to processing 3D graphics, especially with the increased use of volume graphics. It is also worth noting that the points in a distance field do not necessarily have to fill the bounding box around the image or mesh. Sometimes calculations only need distances to points within a threshold distance from the image or surface and a significant amount of space can be saved by storing just these distances in the field. This is demonstrated in Figure 1.1. While not required, the points in a distance field are often evenly distributed over a grid or lattice. That is, there is a constant spacing between points along each axis. This regularity makes it easy to convert between distance fields and voxel representations of data (like the one shown in Figure 1.2) because the area closest to each point is identical.

The surfaces that distance fields are generated from can also vary significantly. These surfaces are often represented as a mesh of triangles or polyhedra, but they can also be represented as a grid of voxels. Figure 1.2 shows the differences between these two methods. They are so dissimilar that different approaches are necessary when working with them. For the purpose of this paper, we will focus on surfaces represented as meshes of triangles as this seems to be the most common type of surface used when generating 3D distance fields.

1.2 Previous Work in Algorithms

There has been a great deal of research on algorithms for computing 3D distance fields. In fact, there are at least 10 different algorithms which vary

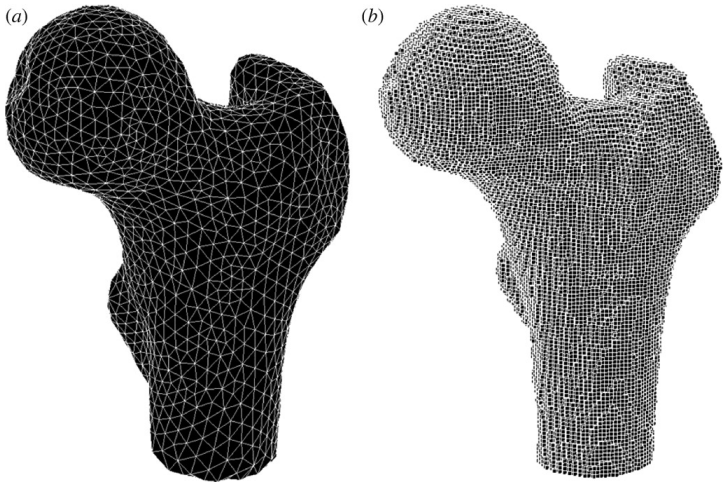


Figure 1.2: The surface of a femur represented using (a) a mesh and (b) voxels [LvL09]

```
for each point in grid:
  best_distance = infinity
  for each triangle in mesh:
    distance = distance from point to triangle
    if abs(distance) < abs(best_distance):
      best_distance = distance
  distances[point] = best_distance
```

Figure 1.3: Pseudocode for the naive algorithm.

```
for each point in grid:
  distances[point] = infinity
for each primitive in triangle in mesh:
  polyhedron = bounding polyhedron for primitive
  grid_points = scan_convert(polyhedron)
  for each point in grid_points:
    distance = distance from point to primitive
    if abs(distance) < abs(distances[point]):
      distances[point] = distance
```

Figure 1.4: Pseudocode for Mauch's algorithm.

significantly in the time they take to compute and in the range of the their accuracy [JBS06]. Before going on to cover some of the more advanced 3D distance field algorithms, I will cover a naive algorithm to show how to convert meshes into distance fields.

1.2.1 Naive algorithm

Perhaps the simplest naive algorithm would be to iterate over every point in a grid and, for each point, find the shortest distance from a triangle in the mesh to that point (by iterating over every triangle in the mesh). The pseudocode in Figure 1.3 helps explain how this algorithm would work (note that abs represents the absolute value function).

While a naive algorithm will correctly compute the distance transform, it is also likely to be very slow. For instance, the above algorithm spends a lot of time computing distances to triangles that are nowhere near the point of interest. However, Sean Mauch found that these problems can be solved using a technique called scan conversion [Mau03]. Scan conversion takes an object and turns it into grid points. In computer graphics, scan conversion can refer to rasterization, the process of converting objects into a 2D image. However, in this case, we are interested in converting 3D objects into the 3D points contained within them.

1.2.2 Mauch's algorithm

Mauch's algorithm is able to use scan conversion to be more efficient than a naive algorithm because it can find polyhedra that contain the points closest to a triangle within a certain distance [Mau00]. Once it finds these

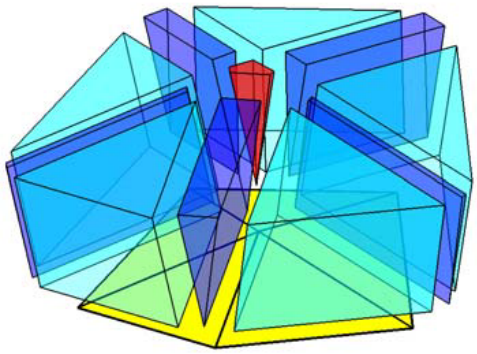


Figure 1.5: Bounding polyhedra for points closest to different primitives from a mesh. Taken from [PS06].

polyhedra, it can scan convert them into points that the triangle might be closest to and only find the distances from those points to the triangle. Mauch's algorithm is also more efficient because it handles the different components of triangles separately. For instance, when handling a vertex of a triangle, Mauch's algorithm generates a polyhedron that will contain grid points that may be closest to that vertex. After scan converting this polyhedron, the algorithm only needs to calculate the distances from the grid points to the vertex from the triangle. This process is significantly faster than finding the distances from these grid points to the triangle as a whole. Mauch's algorithm also generates polyhedra so that they do not significantly overlap with each other (i.e. the polyhedron for a line from a triangle will not contain many points in the polyhedron for the face of the same triangle). This way work is not wasted on calculating distances from grid points to one part of the triangle when the distance to another part will be smaller anyway. Figure 1.5 shows what such polyhedra might look like. Note that the figure uses an expanded view and the polyhedra would normally be flush with the surface of the mesh. The pseudocode in Figure 1.4 should help show how Mauch's algorithm works.

1.2.3 Meshsweeper algorithm

Another algorithm that takes advantage of spatial locality is the "meshsweeper" algorithm [Gue01]. Unlike Mauch's algorithm, the meshsweeper algorithm works by accelerating individual distance calculations. However,

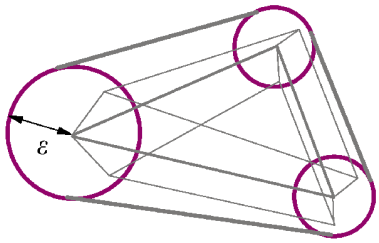


Figure 1.6: The bounding volume from the meshsweeper algorithm [Gue01]

this method is significantly more efficient than using octrees or k-D trees (which accelerate lookups using trees whose branches contain objects in progressively smaller regions of space). Instead of using traditional bounding regions (which are usually just bounding boxes), the meshsweeper algorithm uses regions whose points are within a certain distance ϵ of a triangle in space. Figure 1.6 shows what this bounding volume looks like. This definition of a bounding region allows the meshsweeper algorithm to easily break up the mesh into bounding regions and calculate minimum and maximum distances to the objects inside them (if d is the distance from a point to the triangle defining the bounding region, then objects inside the bounding region cannot be closer than $d - \epsilon$ or farther than $d + \epsilon$ from the point).

The algorithm starts by breaking up the mesh into high-level bounding regions and storing them in a priority queue in the order of their minimum distance to the query point. The algorithm is then able to prune regions whose minimum distance is greater than the maximum distance to objects in the first region. Once the queue is initialized, the algorithm can remove the first region in the queue, break it up into smaller regions, enqueue them, and prune the queue again. This process can be repeated to narrow the estimate of the distance from the query point to the mesh until the regions at the front of the priority queue only contains one object. At this point, a bounding sphere can be found which will only contain points that share the same closest object in the mesh as the query point. Unfortunately, since the regions in the queue cannot be recombined, the algorithm has to eventually restart its mesh refinement when it receives a query for a point sufficiently far from the first query point. However, before then the algorithm can take advantage of its existing refinement and speed up queries for points near the first point queried.

The pseudocode in Figure 1.7 helps demonstrate how the algorithm works for a single point. This algorithm could be used to make a 3D dis-

```
queue = new priority_queue
region = bounding region containing entire mesh
enqueue region onto queue
while region at front of queue contains more than one object:
  region = dequeue region from queue
  regions = split region into subregions
  enqueue each region in regions
  max_distance = max distance from point to first region in queue
  prune regions from queue with min distance greater than max_distance
distances[point] = distance from point to object in first region in queue
```

Figure 1.7: Pseudocode for the Meshsweeper algorithm.

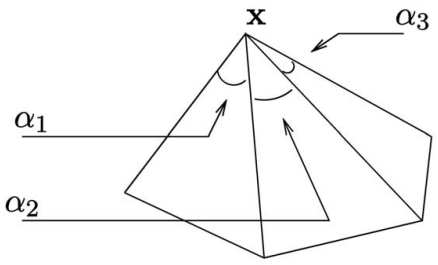


Figure 1.8: The three triangles shown above all contribute to the angle-weighted pseudonormal at vertex x with weights α_1, α_2, and α_3 [BA05]

tance field by doing queries for points in the distance field and filling out the distances one point at a time. However, this process can be accelerated for points near each other by reusing the priority queue for a point nearby and increasing the minimum and maximum distances of the regions in the queue by the distance between the points. This algorithm could also potentially be parallelized by having different threads handle different regions of the distance field and then combining the results.

1.2.4 Angle-weighted Pseudonormal

The final algorithm this paper will cover focuses on calculating the signs of distances from arbitrary points to points on a mesh [BA05]. The result of this paper stems from the observation that, if given an arbitrary point p in space, the closest point c on the mesh to p, and the normal n at point

c, $n \cdot (p - c)$ will have the correct sign (positive if p is outside the mesh, negative is p is inside the mesh, and 0 if p is on the mesh). This is because n points away from the mesh and $(p - c)$ points towards p, so the only way $n \cdot (p - c)$ will be positive is if $(p - c)$ points in the same direction as n and p is therefore outside the mesh. However, n is not always defined. If c is on a vertex or an edge, we cannot just use the normal of any of the faces of the triangles that contain c. This is where the key contribution of the paper comes in: the authors prove that if we use use the angle-weighted pseudonormal for the normal at c, we can still get the correct sign for the distance (although we cannot use $n \cdot (p - c)$ to calculate the magnitude of the distance anymore because n and $(p - c)$ will not necessarily point in the exact same direction, resulting in $|n \cdot (p - c)|$ not necessarily equalling $||p - c||$).

We can calculate angle-weighted pseudonormals for points on vertices and edges by taking the normals for the triangles that share that vertex or edge, weight them by the incident angles of the triangles at that point, and combine them to get the "normal" at that point using the following equation: $n_c = \frac{\sum_i a_i n_i}{||\sum_i a_i n_i||}$ where n_c is the angle-weighted pseudonormal at point c and i is used to iterate over all the triangles that contain c. As the authors point out, when c is on the face of the triangle, the incident angle is 2π and the angle-weighted pseudonormal is the same as the original normal of the triangle face. For edge cases, the incident angle for each triangle will just be π, and the angle-weighted pseudonormal will be the average of the normals of the two triangles. The authors go on to show that using their method to compute the signs for distances added practically no overhead to distance calculations and $20 - 40\%$ overhead to preprocessing the mesh. However, in all of their example meshes, the time it took to generated the tree hierarchies was significantly less than the time the distance calculations took.

1.3 GPUs

GPUs, or Graphics Processing Units, are massively parallel processors that are highly optimized for rendering graphics. However, programmers have found that they are also useful for accelerating a wide variety of highly parallel tasks that extend far beyond computer graphics. There now exist a variety of libraries and programming languages to help programmers write parallel programs for GPUs including OpenCL, CUDA, and Kokkos. While these languages may use different terminology to refer to different

parts of GPUs, the concepts are generally the same. In this chapter, I will use CUDA's terminology to discuss the elements of GPU hardware and programming that are relevant to the methods presented in this paper.

1.3.1 GPU Architecture

At a very high level, a GPU can look a lot like a CPU. Just like how many modern CPUs have multiple cores, each GPU can be broken down into streaming multiprocessors, each of which has its own cache and registers. However, while most CPUs only have 2 or 4 cores, a GPU can have over one or two dozen streaming multiprocessors (SMs). Also, just like how many CPUs now support running multiple threads on the same core (called simultaneous multithreading or hyperthreading), each SM can quickly switch between "GPU threads", called warps. However, instead of being able to switch between two or eight warps, a SM can store up to hundreds of warps simultaneously.

One final similarity between CPUs and GPUs is their ability to process data in vectors. Just like how CPUs have special vector units that can perform the same operation on multiple pieces of data at the same time, each warp can perform operations on 32 or 64 numbers at once (32 in Nvidia's CUDA and 64 on AMD's GCN architecture). However, warps have the added benefit of being able to selectively disable some of these operations, so that a vector operation appears to only happen to some of the pieces of data instead of all of them. This allows a warp to appear as if it is actually made up of 32 independent threads. To prevent confusion between these two types of threads on a GPU, we will refer to warps as GPU threads (since they are managed like threads by the GPU hardware) and refer to the "threads" within warps as CUDA threads (because CUDA exposes each as an independent thread to programmers). While this powerful feature allows each CUDA thread to have its own control flow, performance will suffer significantly if threads within a warp take different branches and try to execute different instructions (this phenomenon is called warp divergence). For this reason, it is good practice to view warps as a single thread that uses vector processing rather than a collection of 32 independent threads.

Finally, it is also worth noting that GPUs have their own RAM that is separate from CPU RAM. While GPU RAM has a much higher bandwidth than CPU RAM (up to around 1TB/s in Nvidia's Pascal architecture vs. 100GB/s for an Intel Xeon CPU), GPU RAM has traditionally been limited in size and required explicit transfers to and from the CPU over the relatively slow PCIe bus. While technologies like Nvidia's NVLink are making it faster

and easier for GPUs to access CPU RAM, NVLink is only supported on IBM POWER CPUs at the moment and explicit memory transfers will still be necessary to optimize the performance of some applications.

1.3.2 GPU Programming

When doing general-purpose programming on GPUs (GPGPU) with CUDA, computations are broken down into threads, blocks, and kernels. As explained earlier, a CUDA thread corresponds to one of the 32 "threads" in a warp. A block is a collection of CUDA threads that can work together. Finally, kernels are the actual programs that are run on GPUs. A kernel specifies a number of blocks to spawn on the GPU, how many CUDA threads are in each block, and the code that will run on each of its CUDA threads.

CUDA allows programmers to access fours kinds of memory: global, constant, shared, and local. Global memory is memory that is accessible by all CUDA threads in a kernel. It is allocated in GPU RAM and is the slowest form of memory on the GPU. Constant memory is also available to all CUDA threads in a kernel. However, it cannot be written to and is designed so that it works best when all threads read from the same location. While the code presented in this paper do not explicitly use constant memory, CUDA does use it to pass kernel arguments to each CUDA thread launched by the kernel. Shared memory is a chunk of block-specific memory that can only be accessed by threads in the same block. Shared memory is significantly smaller and faster than GPU RAM (it actually shares the same hardware as the L1 cache in Nvidia GPUs) and it is automatically freed when a block finishes executing. Finally, local memory is specific to each CUDA thread and is used when the thread cannot use registers for thread-specific memory (e.g. when the thread runs out of registers or accesses arbitrary indices from a thread-local array). Like global memory, local memory is allocated from GPU RAM and is not as fast as shared or constant memory.

CUDA also includes operations that are useful for organizing threads within kernels and blocks. The only way for threads cooperate across an entire kernel is through GPU RAM. In cases where multiple threads could access the same memory location, atomic operations and global memory loads come in handy. Atomic operations prevent threads from updating the same location in memory at the same time. They are available for both global and shared memory. Global memory loads are useful when reading from global memory that may have been updated by another block. This is because CUDA does not guarantee that its L1 caches are up-to-date. However, global memory loads are not necessary for read-only data

as the cached value should be correct. CUDA also provides block-wide synchronization barriers that guarantee all threads in a block have reached a certain point in the program. This is handy for preventing threads from reading from shared memory until every warp in a block has finished writing to it (since warps are scheduled independently from each other on SMs).

1.3.3 Kokkos

Kokkos is a relatively new programming model for both CPUs and GPUs that is being developed by Sandia National Laboratory. Kokkos can change memory layouts and execution patterns to optimize programs for the architecture it's being compiling for. This allows programmers to achieve consistently good performance across architectures without having to specifically optimize their code for each. We used Kokkos to implement the methods presented in this paper as it can be easier to program in than CUDA. However, under the hood, Kokkos uses CUDA to execute programs on GPUs and can easily get similar performance to some programs written in CUDA.

As mentioned earlier, while Kokkos uses different terminology than CUDA, it uses the same concepts for breaking down computations. Instead of blocks, Kokkos has "teams" that can work together using shared memory. In fact, a team in Kokkos literally turns into a CUDA block when compiling in the current version of Kokkos. Just like CUDA, Kokkos also supports team-wide barriers, atomics, and global memory loads. While some more advanced features of CUDA are not available in Kokkos, it would defeat the purpose of trying to create a programming language that runs well across architectures.

1.4 Previous Work in Implementations

While algorithms research is important to reduce the amount of computation required to generate a 3D distance field, implementation research is also important as it can significantly speed up the rate at which algorithms are computed. One area that has gathered a lot of interest recently is using GPUs to accelerate portions of distance field algorithms, especially since they are designed for graphics calculations and have a lot of parallel compute power. This section will detail some of the attempts to use GPUs in 3D distance field calculations.

One of the most obvious ways to use GPUs to speed up distance field calculations is to use them to do what they were originally designed for: scan conversion. In Mauch's algorithm, scan conversion is used to convert polyhedrons to the points inside them. A naive approach to using GPUs to

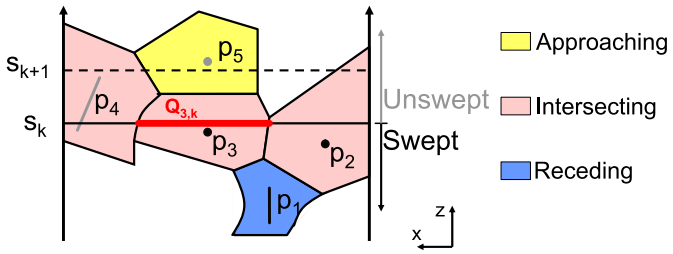


Figure 1.9: Different classifications of regions when sweeping across the z-axis [SOM04]

handle this scan conversion would be to send the polyhedrons (or 2D slices of them) to the GPU and have it render them. Unfortunately, this idea does not work very well in practice due to limited data bandwidth between CPUs and GPUs [SP03].

Sigg et al. found that they could significantly reduce the amount of data that had to be transferred by only generating one bounding prism per triangle in the mesh (instead of a bounding polyhedron for each triangle face, edge, and vertex). In doing so, they were able to significantly reduce the bandwidth required to send the data to the GPU and were able to get a 3.3-16.7x speedup over Mauch's algorithm depending on factors like the mesh size and grid resolution. Although, it is worth noting that in a later paper Sigg and Peikert pointed out that their method can lead to gaps where there are no prisms at all [PS06]. They found that this can be fixed by making the prisms bigger so that they covered the normals of the vertices, even in the worst cases. While this made the polyhedra overlap more than necessary, the authors were still able to get speedups similar to those in their earlier paper.

Sud et al. came up with a different approach for speeding up calculations on the GPU [SOM04]. For the first part of their method, they classified regions as either approaching, intersecting, or receding when sweeping across z-axis (Figure 1.9). Approaching sites might be relevant to calculate the distance field after certain z values, intersecting sites are definitely necessary at certain z values, and receding sites can be ignored after certain z values. These classifications allow their algorithm to ignore sites that would not contribute to their distance field calculations as it sweeps through slices of the grid.

The second part of Sud et al.'s involves using the maximum distance to

a site to restrict the distance calculations to that site. To calculate these maximum distances for each site for a slice, the authors start by taking the maximum distance from the last slice and add the distance between slices to get an upper bound on the potential maximum distance for the current slice. This potential maximum distance allows the algorithm to ignore calculating distances from a site to regions of a slice where the distance to that site is guaranteed to be greater than this maximum distance, reducing the number of distance calculations. Using these two approaches, Sud et al. were able to significantly improve the efficiency of using the GPU to do distance field calculations.

However, Sud et al. found that other GPU hardware could also be used to accelerate distance field calculations as well. They were able to take advantage of the GPU's interpolation hardware so by feeding data to the GPU as vertex attributes and assigning points in slices to pixels in the output of the GPU [SGGM06]. In doing so, Sud et al. were able to recover distance vector data from the GPU. When used to identify proximity between objects, this method achieved a 7-12x speedup over a previous one that also used GPU hardware. However, they also found that they could use the GPU's vertex processors, stencil capabilities, and texture memory to compute surface distance maps [SGG$^+$07]. After doing so, they were able to achieve a speedup of 8x over their results published the previous year.

Chapter 2

Methods

As seen in the previous work on distance field algorithms (Section 1.2), there are two major approaches to computing distance fields. In one method, the program iterates over the elements of the mesh and finds relevant points to find distances to. In the other method, the program iterates over points and finds the closest element in the mesh to get the distance to. Unfortunately, neither of these methods will work well in all situations. If the user only cares about points within a small distance of the surface, the first method makes the most sense and will be the most efficient. However, if the user wants the distance to every point in a large area, this will obviously not work very well as each point will have a distance calculated to it many times over. In the worst case, each primitive will calculate the distance from itself to almost all the points in the distance field, which approaches the massive amount of work that would have to be done in the naive algorithm. Instead, the second method would work better here because a good algorithm could find the closest primitive in the mesh in at least logarithmic time. However, the second method will obviously not work well in the original problem because it will calculate the distance to many more points than necessary and each distance calculation will be relatively expensive. For the purposes of this paper, we chose to focus on methods that work best for small distances from the mesh surface. These kinds of distance fields are useful for a variety of graphics applications and utilize limited GPU memory better (we assume that distances to points that are close to the surface are probably more relevant than distances to points that are farther away from the mesh).

When iterating over elements of the mesh, we also had the choice of iterating over whole triangles or their primitives (vertices, edges, and faces). Since the first method involves a lot of repeated work (edges are processed twice and points are processed at least 3 times over), we decided to iterate

over them separately and optimize our methods for each primitive. Our methods for each primitive (points, edges, and faces) are covered below (Sections 2.2, 2.3, and 2.4).

Some final, but very important aspects of our methods, are the data structures used to store and recall distances to points and to hold the mesh data. Not only is it important for the point data structure to be compact, but it must also be easily be built at run time on the GPU. We discuss all of these data structures below (Section 2.1).

2.1 Data Structures

Our code uses two static data structures to store mesh and geometry data on the GPU. Our mesh data consists of the number of vertices in the mesh; the number of triangle faces in the mesh; 3 arrays of floats for the X, Y, and Z, locations of the points; and 3 arrays of unsigned integers indicating the first, second, and third indices of the points that form each triangle. We went with a "struct of arrays" instead of an "array of structs" layout as the former results in more efficient fetching of data from GPU RAM (having threads load sequential indices from memory at the same time reduces the number of cache lines that must be retrieved from relatively slow global memory).

The geometry data describes the grid / lattice that determines the number and locations of the points in the distance field. The geometry data contains the minimum and maximum X, Y, and Z coordinates for points in the lattice; the number of points along each axis (also called extents); and the distance between adjacent points, as measured along each axis. For example, a lattice could be described as having minimum coordinates $(0f, 0f, 0f)$ and maximum coordinates $(1f, 1f, 1f)$ (the f indicates that the coordinates are floating point values in space). If there are 3 points along the X axis and 2 points along the Y and Z axes, then the coordinates of every point in the grid would be $(0f, 0f, 0f)$, $(0f, 0f, 1f)$, $(0f, 1f, 0f)$, $(0f, 1f, 1f)$, $(0.5f, 0f, 0f)$, $(0.5f, 0f, 1f)$, $(0.5f, 1f, 0f)$, $(0.5f, 1f, 1f)$, $(1f, 0f, 0f)$, $(1f, 0f, 1f)$, $(1f, 1f, 0f)$, and $(1f, 1f, 1f)$. In order to reference each point in a consistent manner, we can identify them by their index along each axis. For example, the point $(0.5f, 1f, 1f)$ would have indices $(1i, 1i, 1i)$ since it is the 1th point with respect to the X axis, the 1th point with respect to the Y axis, and the 1th point with respect to the Z axis (the i indicates that the coordinates are integer indices). The means we can reference points by both their coordinates in space (e.g. $(1f, 0f, 1f)$) and their order (e.g. $(2i, 0i, 1i)$). Since integers are

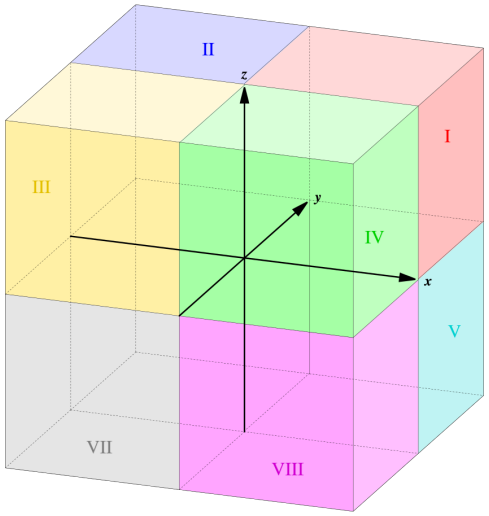


Figure 2.1: Box-like regions of space can be divided into 8 box-like octants [Wik13]

not susceptible to the errors that floating point numbers can accumulate over time, we try to reference points by their integer coordinates as much as possible and only generate their floating point coordinates when needed. We can do this for each axis by calculating $min + i * (max - min)/(num\_points - 1)$, where min and max are the minimum and maximum coordinates for the axis, num_points is the number of points along the axis, and i is the index of the point along the axis (therefore ranging from 0 to $num\_points - 1$). Note that $(max - min)/(num\_points - 1)$ is simply the distance between adjacent points along an axis and will have already been precomputed, meaning that the conversion from integer indices to floating point coordinates is fairly efficient.

For our distance field data structure, we decided to use an octree. In our specific implementation, our octree consists of two types of nodes: non-terminal nodes and leaf nodes. Non-terminal nodes correspond to ranges of points along each axis. For example, the root node for the octree representing the grid defined earlier would represent points with X indices ranging from 0 to 2, Y indicies ranging from 0 to 1, and Z indices ranging from 0 to 1 (all of the points in the grid). Nonterminal nodes consist of 8 references to subnodes, each of which corresponds to one of the 8 octants that the node can be divided into (Figure 2.1). For example, one subnode of the root node

would represent points with X indices ranging from 0 to 1, Y indices ranging from 0 to 0, and Z indicies ranging from 0 to 0. If a subnode's region is narrowed down to a single point in the entire grid, the subnode will not be a non-terminal node, but a leaf node. A leaf node simply contains the floating point distance from the surface to the sole point in the region represented by the leaf node.

This approach to storing distance data on GPUs works well for a variety of reasons. First of all, it is fairly compact (at least when each axis has a similar extent). Since it is unambiguous which point each leaf node refers to, the X, Y, and Z index of the point to not have to be explicitly stored in the data structure, saving a significant amount of space. We can also save space with octrees by not storing distances to grid points that are irrelevant. For many meshes, the majority of the points are not even close to the surface represented, which means massive space savings compared to a naive data structure that allocates space for the distance to every point in the grid. Due to the nature of our octree, all leaf nodes will be at roughly the same depth. This means that threads in a warp will not have to wait long for the rest of the threads to finish traversing the octree. Furthermore, if multiple threads in a warp are processing points near each other, they will likely make similar traversals through the tree, resulting in less data having to be fetched from relatively slow global memory to the warp. Finally, the calculations required to determine which subnode contains a point of interest are fairly lightweight, which helps reduce the overhead of accessing data from the octree.

Octrees also have one more significant advantage that makes them a good fit for calculating distance fields: they can easily be built on demand, even on GPUs. Since our code generates grid points to find distances to at run time, new nodes to store these distances will have to be added to the tree as run time as well. This can be accomplished as follows. We first start by reserving the majority of the free memory on the GPU for the octree. Since we don't know how large it will eventually get, it is better to allocate as much as possible than to run out of space. In our code, we reserve 85% of memory left over after copying over our mesh and geometry data. While we would prefer to allocate more, we found that CUDA would not let us allocate all of the "free" memory on the GPU. We also allocate space for another global variable that is used to keep track of how much of space reserved for the octree is actually used. To finish initializing the octree, we zero the memory and set the allocated space to 8 (meaning 8 spaces have been reserved for the 8 node references that make up the root node). It is worth noting at this point that the octree is actually made up of a union of (unsigned)

integer indices and floating point distances. Since it is indeterminate at compile time which memory locations will store which kinds of values and we are managing our own memory, we have to be prepared to be able to store either kind of value in any space in the octree. This means that non-terminal nodes are really just 8 contiguous integer values (which can hold the indexes of other nodes), leaf nodes are just sole floating point values (which can hold a distance value), and references to nodes are just the index of the first element in the node. While the references could just be replaced by pointers to the start of the nodes, it is easier to offset indices as required when accessing all but the first element of non-terminal nodes. It is also worth noting that using structs would be difficult in this situation as the structs for non-terminal nodes would be 32 bytes in size while the structs for leaf nodes would be 4 bytes in size. Since each index in the octree could contain either of these nodes, C++ would require 32 bytes of space to be allocated, even if we just wanted to store a leaf node. Therefore we decided to not use this approach.

As our program runs, it will eventually identify grid points within the cutoff distance of the mesh and it will need to be able to save the distance between these points and the surface of the mesh. Once a CUDA thread has a distance to store, it starts by traversing the octree at the root node and identifying which of the eight subnodes corresponds to the region that contains the grid point of interest. Once it has the subnode, it grabs one of the eight indices in the node that corresponds to the starting index of the subnode of interest. This process repeats until the thread either finds the index of the leaf node for the distance value of interest or it finds that the index is uninitialized (i.e. still 0). In the second case, space needs to be allocated for this node. In order to prevent other threads from allocating space for the same node (or changing the index for the node once it has been initialized), the thread tries to "lock" the index variable by atomically attempting to replace the 0 with a reserved value. If the locking operation succeeds, the thread knows it is safe to allocate space for the new node. This is accomplished by increasing the size of the data structure (through the global size value mentioned earlier) by the size of the desired node and getting the original size value back. This size, which doubles as an index values, is guaranteed to be unique, unused, and large enough to hold all the node data. The thread then updates the index for the subnode with the index of the space it just allocated for it. In the case that a thread finds that an index is uninitialized but it does not successfully acquire the lock for that index (because another thread acquired it first), the original thread simply waits until the index has been updated, loads the index value, and

proceeds as before. This process repeats until the thread finally allocates space for the data value of interest and initializes the data value with the distance it wanted to store there. If space for the distance value was already allocated, the thread must then repeatedly attempt to atomically replace the old distance value with its own until the stored value is either less than or equal to the value the thread originally wanted to store.

2.2 Points

When enumerating grid points within the cutoff distance triangle vertex, there is one optimization that we can make that is not possible with the other two triangle primitives. Since points do not vary in size like triangle edges or faces, they do not require any kind of team work and can actually be processed by individual threads within a warp.

Traditionally, the lattice points within the cutoff distance of a vertex were found by using bounding boxes or GPU rendering hardware. However, both of these methods process far more lattice points than necessary because they process points in cubes and squares, which contain many more points than spheres and circles. We came up with a novel method that almost exclusively enumerates lattice points within the cutoff distance of the triangle point. This method approximately halves the number of lattice points that need to be processed without introducing any overhead from team / block synchronization or warp divergence.

Our method starts by having each thread find the minimum and maximum X, Y, and Z values for lattice points around its point. Each thread then iterates over the range of possible X values for lattice points around the point. For each X value, the thread starts with a Y value that is in the middle of the minimum and maximum Y values (placing it towards the center). The thread then increases the Y value until the distance to the original triangle point is greater than the cutoff distance, restarts with a Y value 1 index smaller than the original starting value, and decreases the Y value until the distance exceeds the cutoff (we assume an ideal Z value during the distance-calculating process). This enumerates every Y value for the given X value that could result in a lattice point that is within the cutoff distance of the point. Finally, for each Y value, the thread then iterates over Z values in the same way as it does for Y values (except we can calculate the actual distance now because we have an X, Y, and Z value to work with). By starting with Y and Z values towards the middle and working our way outward, we guarantee we can stop enumerating more points once

we increase or decrease the Y or Z value too much. This prevents us from enumerating much, if any, points outside the cutoff distance. If the thread finds that a specific X / Y / Z value pair has a distance to the triangle point that is less than the cutoff distance, it updates the distance to that lattice point using the method discussed earlier (Section 2.1).

2.3 Edges

Unfortunately, processing edges is much more complex than processing points. Since edges can be many different lengths and have many different orientations in space, we cannot use optimizations like those from our point methods. However, we did find some ways to utilize some of the more advanced features of GPUs to help process edges in parallel.

Our edge processing method involves three separate stages to help break down a complex problem into a simpler, more parallel one. In a naive approach, one would just make a bounding box around the cylinder formed from all the points within the cutoff distance of the edge. You could then just iterate over every point in that bounding box and only save the distances to points within the cutoff distance of the line. You would also have to filter out points that do not project onto the line segment of interest (since lines are infinite, you have to be careful not to use the distance from a lattice point to part of the line that is not included in the edge of the triangle). To reduce the amount of computation spent on processing bounding boxes, we will only use the naive bounding box approach on the smallest chunk of the cylinder / edge that we can.

The team / block first starts by figuring out which axis the cylinder / edge is most aligned with. For the purpose of explaining this method, let's assume that this is the X axis. The code would then find the lattice point that is closest to the edge for each step along the X axis. For example, assume we have a lattice with a minimum X coordinate of 0, a maximum X coordinate of 5, and a width of 6 points along the X axis. Let us also assume that we have an edge that goes from $(0.5f, -, -)$ to $(3.5f, -, -)$ (Y and Z coordinates are omitted because they are irrelevant). The code would then find that the first lattice X coordinate on the line would be $1f$ (or $1i$) and the last lattice X coordinate on the line would be $3f$ (or $3i$). For each coordinate in this range ($1i$ to $3i$), the code would then find the Y and Z coordinates of the lattice points that place them closest to the line for each X coordinate. This is accomplished by first finding the point on the line with the given X coordinate, which can be done with linear

interpolation. Simply figure out how far along the line you are by calculating $percent = (x\_coord - line\_start\_x)/(line\_end\_x - line\_start\_x)$ and use this to weight each Y and Z coordinate (e.g. $y = line\_start\_y + (1 - percent) * line\_end\_y$). To get the lattice Y and Z coordinates, you then simply round the actual coordinates to the nearest lattice coordinates (which can be done by converting the floating point coordinates to integer coordinates and back while paying careful attention to reduce rounding error). These lattice points are our "starting points" and we will return to them later. To complete this stage of the method, we simply find the distance between each pair of neighboring starting points and find the largest of all of them (which can be done with atomics on shared memory or by having the team do a simple reduction).

It then comes time to find which points are inside a chunk of the cylinder. We will accomplish this by creating a bounding box around a small portion of the cylinder and testing every point inside the bounding box to see if it actually falls within the cylinder. We start by pretending that there is a lattice point that lies in the center of the cylinder of interest (and therefore on the line through the cylinder) and construct a bounding box around this point that will contain any lattice point within the cutoff distance of the central point (plus some error since the actual lattice points will likely not fall perfectly on the line). For each lattice point in the bounding box, the team then finds the euclidean distance to the central point and the distance to the central point along the edge (i.e. project the lattice point onto the edge and then measure the distance to the central point). We can imagine the lattice point, the projection of the lattice point on the edge, and the central point form a right triangle, with the central point and lattice point connected via the hypotenuse. We know that the distance along the line should not exceed half of the maximum distance between points that we found earlier. We also know that the distance between the projected point on the line and the lattice point should not exceed the cutoff distance (plus some error because the starting points will not be perfectly on the line whereas the central point is assumed to be). This gives us a maximum distance for the hypotenuse, which can be used to throw out points that have too high of a euclidean distance. We can also throw out points that are too far away when their distance is measured along the edge as these will be handled by another starting point. If a lattice point passes these tests, then the thread processing it atomically allocates space for it in shared memory and stores it there. If there is not enough room in shared memory for all the lattice points, we can move on to the next step and revisit this one later once we are done with the points we have just generated.

The final step is to "copy" the lattice points that we generated in the previous stage and "paste" them around the starting lattice points along the edge. This allows us to take a few points inside a small slice of a cylinder and turn them into many points that surround the entire edge. We do this by assigning one thread to each starting point. Each thread then iterates over the lattice points from the last step (the ones that were relative to the central point) and moves them so that they are now relative to the starting point that the thread is processing. Almost all of these points will lie in the actual cylinder around the edge, which means this part of the process is very efficient. The threads then find the distances from the newly generated lattice points to the edge and add them to the point data structure if the distance is within the cutoff distance. Note that the first and last starting points may need to also check for points that are beyond the edge of the line. It is also worth noting that implementing the method in this fashion may not work well for short edges. For example, if there are less than 32 starting points for an edge, then at least one warp will not be fully utilized. However, some experimentation in splitting up the work so the threads now iterated over starting point / lattice point pairs suggested that this approach is more efficient.

2.4 Faces

Unlike our edges method, we decided to keep our faces method simple. While it should be possible to only enumerate points that are above and below the triangle face (we actually came up with at least one potential method to do so), the amount of work it would require to enumerate just those points would likely outweigh the benefits of avoiding processing irrelevant lattice points. Instead, we determined that a very lightweight point enumeration scheme would actually run faster (and would be much easier to implement and debug).

Like in our edges method, our faces method uses a bounding box approach to enumerate lattice points around the triangle. We do this by first finding the normal to the triangle face. We then scale the normal vector so that it is the length of the cutoff distance. With this normal, we can find the six points that make up the prism that contains all the lattice points within the cutoff distance of the face. To get the coordinates of the bounding box, we simply take the min and max of the coordinates of all of the six points, which results in two points at either size of the bounding box. From there, we have the team iterate over every point in the bounding box. Each point

is checked to make sure it is directly above or below the triangle face and, if so, its distance is calculated to the surface and recorded in the point data structure.

Chapter 3

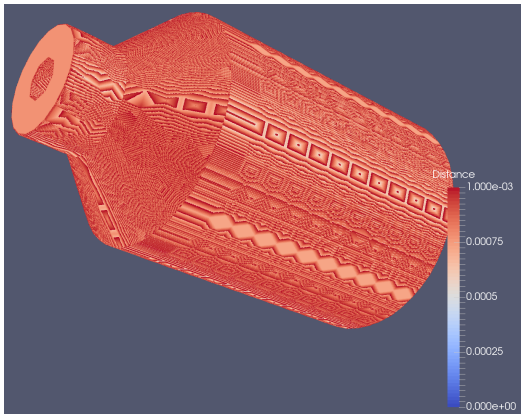
Examples

In this chapter, we present rendered outputs of the 3D distance fields produced by our code. We accomplished this by converting the point data structure generated by our program into a format compatible with Paraview, a data visualization program often used for scientific computing.

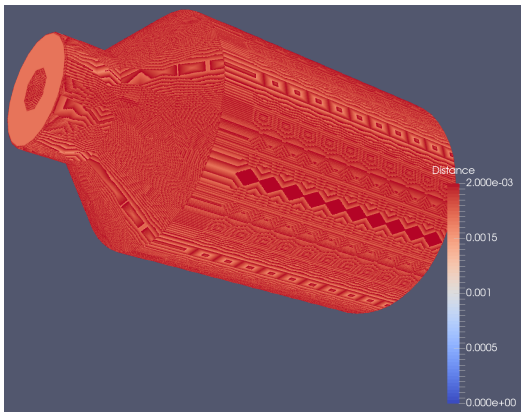
3.1 Canister Mesh

The following figures show the results of converting Sean Mauch's canister mesh into a 3D distance field. In Figure 3.1, we can see the effects of changing the cutoff distance on the mesh. When the distance is small, the differences in distance between lattice points near the surface are more pronounced. For significantly larger cutoff distances, the relative distances between lattice points and the mesh are reduced and the points at the extremity of the distance field have a more uniform color.

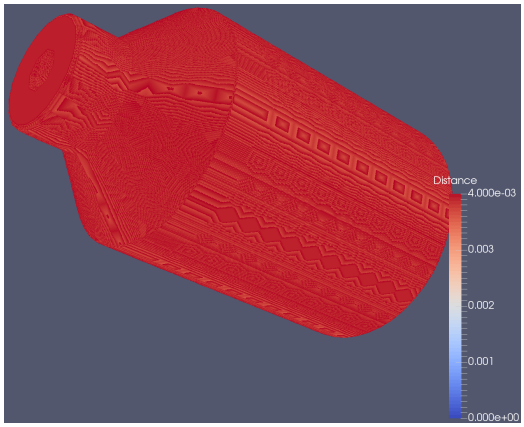
Figure 3.2 demonstrates the ability of the code to handle a wide range of lattice extents. Lattices with smaller extents are faster to compute, but come out very pixelated. Lattices with larger extents have much more detail, but take up a lot of space. The Paraview file for the last distance field is over 4GB in size.



(a) Maximum distance of 0.001

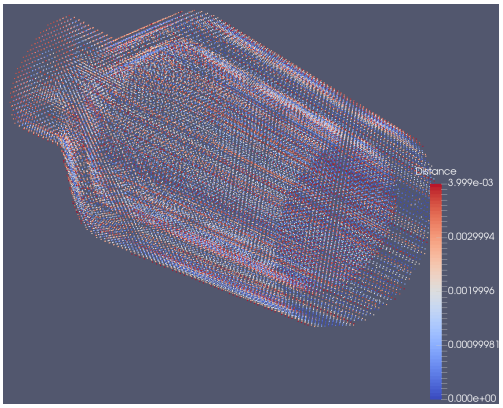


(b) Maximum distance of 0.002

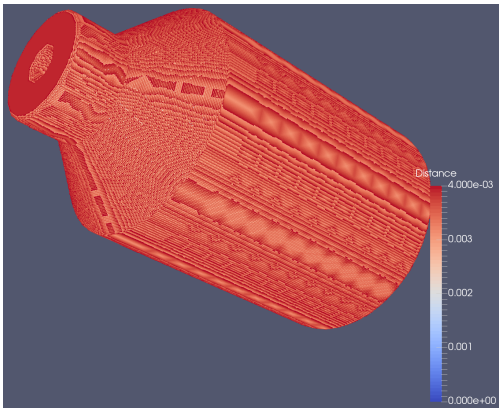


(c) Maximum distance of 0.004

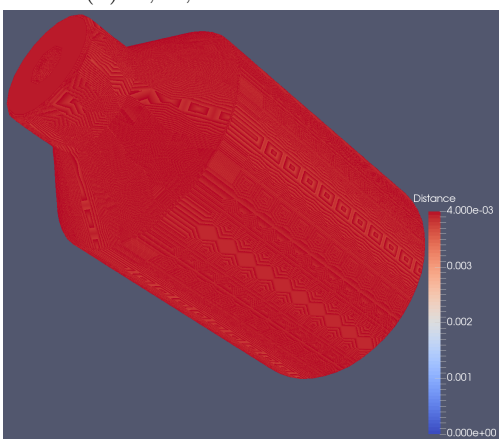
Figure 3.1: Canister mesh processed with X, Y, and Z extents of 1000 points.



(a) X, Y, & Z extents of 100



(b) X, Y, & Z extents of 500



(c) X, Y, & Z extents of 1500

Figure 3.2: Canister mesh processed with maximum distance of 0.004.

Chapter 4

Results

In this section, we attempt to characterize the performance of our code under a variety of conditions and analyze how different parts of our methods contribute to the overall run time of the distance field computation. Performance times are only for the actual distance calculations on the canister mesh (so time spent copying or zeroing data is not included, but time spent building the distance field is). Each time is the average of three runs.

4.1 Scaling with Problem Size

Figures 4.1 and 4.2 show how the run time of the distance field calculations varies with the cutoff distance and lattice extents (points along each axis). Both trends are third order in nature, which makes sense because of the three-dimensional nature of the problem (doubling the cutoff distance increases the area around the mesh by roughly eight times and doubling the extent of each axis increases the number of points by eight times). Surprisingly, the relationship between cutoff distance and time was perfectly characterized by the equation shown, which is often uncommon in performance data. The parameterization of the relationship between extent and time also closely matches the data.

Figures 4.3 and 4.4 show how the size of the generated point data structure scales with the cutoff distance and lattice extent. Note that data does not need to be averaged as the code consistently generates the same distance field for the same input. In the first figure, we can see that the size increases almost linearly with cutoff distance. However, for the last cutoff distance, we can see that the scaling is not truly linear. Perhaps this is because the lattice points generated around each triangle are starting to overlap more

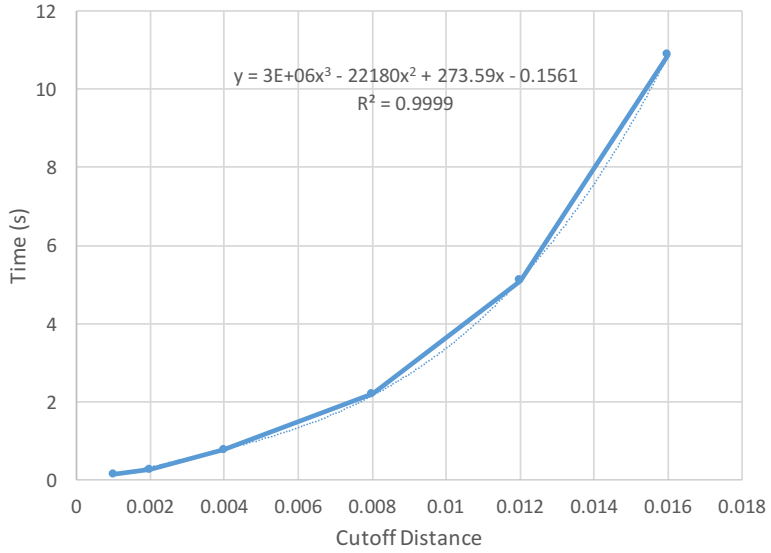


Figure 4.1: Time taken to generate distance field (with a lattice extent of 1000) vs. cutoff distance

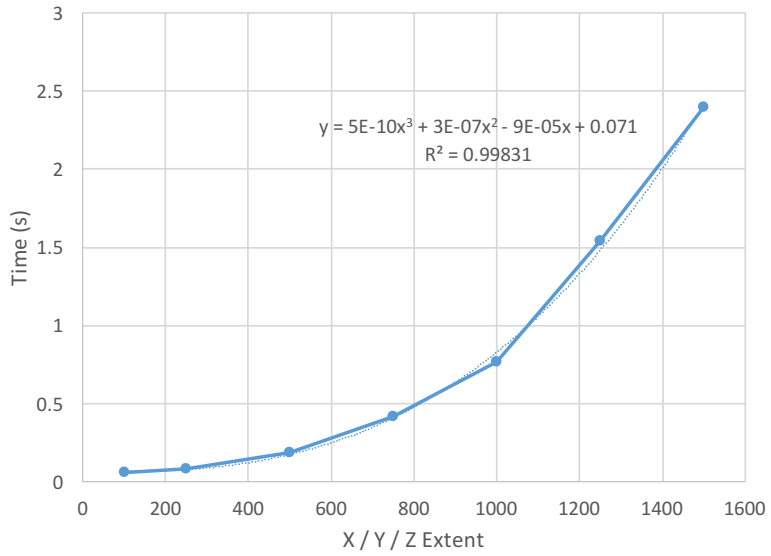


Figure 4.2: Time taken to generate distance field (with a cutoff distance of 0.004) vs. lattice extent

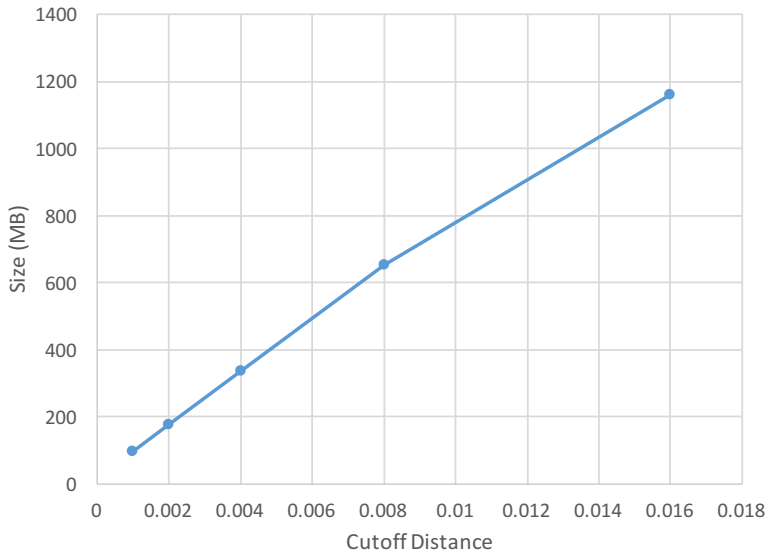


Figure 4.3: Size of the point data structure (with a lattice extent of 1000) vs. cutoff distance

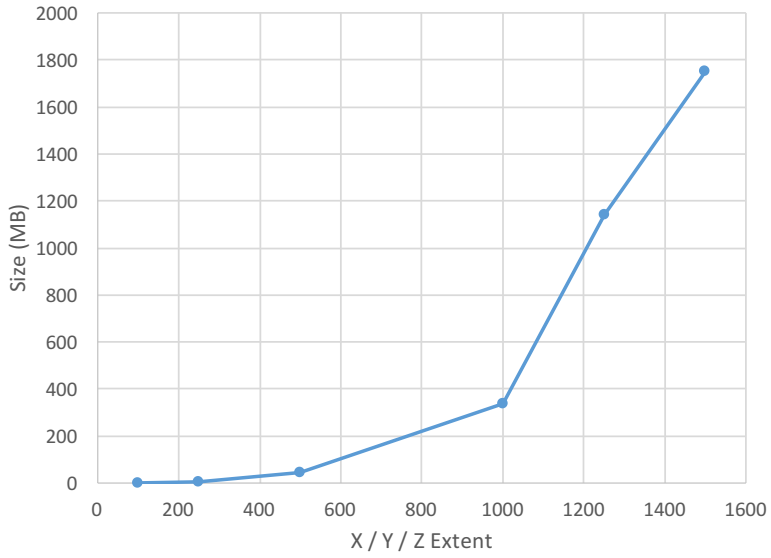


Figure 4.4: Size of the point data structure (with a cutoff distance of 0.004) vs. lattice extent

and less new points are being generated. In the second figure, we can see that, at first, the size seems to increase exponentially. However, there is an inflection point around a lattice extent of 1000 and the increase in size starts to slow down. Since this data is more erratic, trendlines are not included in the graphs.

4.2 Performance Breakdown

In Section 4, results were reported for the time it took to process the points, edges, and faces in a mesh. However, we found that there were significant differences between how much each method contributed to the total runtime. In this section, we breakdown the performance of our method as a whole into the performance of the point, edge, and face processing methods.

Table 4.1 shows how much each method contributes to the total runtime. Point data structure size and run time are presented as percentages of the result of performing all three methods. This data reveals a huge difference between the costs and contributions of each method. For instance, processing the faces enumerates almost all the points in the final distance field, but only takes about a fifth of the run time. However, processing the edges generates fewer points than the processing faces stage, yet takes three times longer. Processing points is probably the most balanced as it only enumerates about a fifth of the points, but only takes about a fifth of the run time. It is worth noting that the edges method should be generating close to as many points as the faces method because the cutoff distance used causes many of the points to overlap around the edges and faces. Data is also available for pairs of methods since points generated by different stages often overlap, which saves later stages from having to add them to the octree.

Table 4.1: Performance breakdown when processing the canister mesh with a cutoff distance of 0.004 and X / Y / Z extents of 1500

Points	Edges	Faces	Percent Size	Percent Time
yes	no	no	22.87%	18.08%
no	yes	no	90.42%	62.81%
no	no	yes	98.37%	21.73%
yes	yes	no	90.47%	79.60%
yes	no	yes	99.01%	37.54%
no	yes	yes	99.97%	82.92%
yes	yes	yes	100.00%	100.00%

4.3 Compression

One nice feature of using octrees for representing the distance field on the GPU is that they are very space efficient. In this section, we will compare the size of our octree against the size of the final output files that are generated for Paraview. These output files mostly consist of the X, Y, and Z coordinates of the lattice point and the distance to that point with a small amount of metadata. Ideally, our GPU data structure would be a quarter of the size of the Paraview data structure because it would only contain distance data.

Table 4.2 shows how the sizes of the two data structures compare and gives the compression ratio for the octree representation over the Paraview one. In the first half of the table, we can see that the compression ratio increases with cutoff distance (and thus the size of both data structures). For larger cutoff distances, the compression ratio gets quite close to the ideal ratio of 4x. The second half of the table contains data on how the data structure sizes and compression ratio change with lattice extent. For the smallest X / Y / Z extent, the octree representation does not help as much (in fact, octree nodes likely take up more space than the actual distance values in this case). For larger extents, the compression ratio is much higher and generally increases with the extent. However, for the last two extents, the compression ratio is much lower than expected.

Table 4.2: Compression ratios of different canister distance fields

Cutoff Dist.	Extent	Size on GPU	Size on disk	Compression
0.001	1000	97.3 MB	323.7 MB	3.3x
0.002	1000	176.4 MB	629.9 MB	3.6x
0.004	1000	335.3 MB	1,242 MB	3.7x
0.008	1000	654.8 MB	2,472 MB	3.8x
0.016	1000	1,161 MB	4,414 MB	3.8x
0.004	100	0.6 MB	1.0 MB	1.7x
0.004	250	6.0 MB	18.0 MB	3.0x
0.004	500	43.4 MB	146.9 MB	3.4x
0.004	1000	335.3 MB	1,242 MB	3.7x
0.004	1250	1,143 MB	2,511 MB	2.2x
0.004	1500	1,750 MB	4,044 MB	2.3x

4.4 Distance Field Data Structure and Atomics

In addition to the three major triangle primitive processing methods, the data structure they all share is another potential factor in program performance. In this section, we analyze two different ways using octrees and atomics impact computation run times.

In order to quantify the impact of adding data to the octree (allocating new nodes and atomically updating distance values), we ran our distance field computation twice: once to initialize the data structure and once with the data structure already initialized by the previous run. Since the first run would fill out the entire data structure with the minimum distance to each point, the second run would only have to enumerate points, calculate distances, and traverse the octree only to find that the distance stored does not need to be updated. We calculated how much slower the original computation ran compared to the second run, which can be seen in the "Writing Slowdown" column of Table 4.3. Note that a slowdown of 0% would indicate no change in run time, a slowdown of 100% would indicate the original program took twice as long to run and so on and so forth. As can be seen in the data, the impact of having to add nodes and update values is biggest for relatively small computations. Perhaps this is because the impact of initializing high-level nodes is greater for smaller computations whereas this cost is amortized in larger computations where more threads benefit from them having been initialized earlier.

Table 4.3: Slowdowns due to adding data to the octree and accessing it in general

Cutoff Distance	Extent	Writing Slowdown	Accessing Slowdown
0.001	1000	84.50%	121.14%
0.002	1000	45.10%	254.62%
0.004	1000	23.04%	498.46%
0.008	1000	15.56%	779.48%
0.016	1000	5.29%	1,449.70%
0.004	100	1,008.83%	3.25%
0.004	250	196.74%	45.06%
0.004	500	60.59%	163.41%
0.004	750	31.93%	351.32%
0.004	1000	23.04%	498.46%
0.004	1250	20.17%	766.33%
0.004	1500	17.32%	910.09%

We also attempted to quantify how much of an impact the data structure made as a whole. To do this, we disabled all accessing of the data structure (threads weren't even allowed to traverse it). Instead, we just had the thread save the distances and lattice point indices to volatile variables, which should prevent them from being optimized out of the computation (this would also approximate the run time of the computation with a "perfect" data structure that could store data in the minimal amount of time). We then compared the run times under these conditions to the original run times by calculating the slowdown caused by having to traverse and add data to the octree. These results are shown in the "Accessing Slowdown" column of Table 4.3. Interestingly, unlike in the previous case, the slowdown actually increases with problem size. However, this actually supports out earlier theory: threads benefit from not having to add as much data in larger computations, but still have to pay the price of traversing the data structure. However, this does not explain why the slowdown gets worse when the lattice extent stays the same. Since traversing the data structure shouldn't take longer (because data is at the same depth), it is unclear why a larger percentage of the time is spent traversing the data structure. Regardless, this data shows that reducing the time it takes to traverse the distance field data structure would significantly improve run times for computations.

Chapter 5

Conclusion

Accelerating 3D distance field calculations is becoming more relevant than ever as researchers continue to add to the wide variety of applications for them. However, the complexity and computational power required to generate these distance fields continues to grow as we are able to create increasingly complex 3D models to process and demand higher fidelity outputs than before. In this paper, we presented methods for taking advantage of modern parallel hardware (GPUs) to efficiently accelerate these intense calculations and analyzed their outputs and performance. While we found that our methods were generally useful, we did identify some areas that could use further improvement, which are discussed below.

5.1 Future Work

There are a few major tasks that deserve further consideration. Section 4.2 showed that the edges method has significant room for improvement (it runs about 67% slower than the other two methods combined). However, another major concern is the complexity of its implementation. Even my own code is not perfect and likely fails to enumerate a few lattice points around the ends of the edges. It would be very worthwhile to spend more time testing other approaches to processing edges.

However, Section 4.4 suggests that developing better data structures may have even more of an impact on performance than improving the edges method. In the worst case, having to interact with an octree slowed down computations by 15x, which is massive. One particularly interesting potential replacement for octree is a GPU-optimized implementation of OpenVDB. OpenVDB has many of the benefits of octrees, but is able to reduce

access times by shortening the height of the tree through the use of a map to find subnodes at the root level and by storing multiple data values in leaf cells.

Finally, it would be nice to implement signed distance calculations and support for the closest point transform. This could easily be done by generating pseudonormals for points and edges in a mesh preprocessing step. Then the triangle primitive processing methods could determine whether distances should be positive by taking the dot product of the pseudonormal vector and the vector between the lattice point and corresponding point on the surface. The closest point transform could then be added by saving the corresponding point on the surface instead of throwing it away once the distance calculations is complete.

5.2 System Configuration

All computations in this paper were performed on a computer with a Xeon E5-2630 v2 CPU, Nvidia Tesla K20m, and 64GB of RAM. Code was compiled with optimization level 3 using GCC 4.7.4, Cuda 7.5.6, and a build of Kokkos from June 2015. The canister mesh used in this paper can be found in Sean Mauch's stlib library, which is available at `https://bitbucket.org/seanmauch/stlib`.

Bibliography

[BA05] J. Bærentzen and Henrik Aanæs. Signed distance computation using the angle weighted pseudonormal. In *IEEE Transactions on Visualization and Computer Graphics*, volume 11, pages 243–253, 2005.

[Gre07] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, pages 9–18. ACM, 2007.

[Gue01] A. Guezlec. "meshsweeper": dynamic point-to-polygonal mesh distance and applications. In *IEEE Transactions on Visualization and Computer Graphics*, volume 7, pages 47–61, 2001.

[JBS06] Mark Jones, J. Bærentzen, and Milos Sramek. 3d distance fields: A survey of techniques and applications. In *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[LvL09] Leen Lenaerts and G. Harry van Lenthe. Multi-level patient-specific modelling of the proximal femur. a promising tool to quantify the effect of osteoporosis treatment. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 367(1895):2079–2093, 2009.

[Mau00] Sean Mauch. A fast algorithm for computing the closest point and distance transform. Technical report, California Institute of Technology, 2000.

[Mau03] Sean Mauch. *Efficient algorithms for solving static hamilton-jacobi equations.* PhD thesis, California Institute of Technology, 2003.

[PS06] Ronald Peikert and Christian Sigg. Optimized bounding polyhedra for gpu-based distance transform. In *Scientific Visualization: The visual extraction of knowledge from data*, pages 65–77, 2006.

[SGG+07] Avneesh Sud, Naga Govindaraju, Russell Gayle, Erik Andersen, and Dinesh Manocha. Surface distance maps. In *Proceedings of Graphics Interface 2007*, pages 35–42, 2007.

[SGGM06] Avneesh Sud, Naga Govindaraju, Russell Gayle, and Dinesh Manocha. Interactive 3d distance field computation using linear factorization. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 117–124, 2006.

[SOM04] Avneesh Sud, Miguel Otaduy, and Dinesh Manocha. Difi: Fast 3d distance field computation using graphics hardware. In *Computer Graphics Forum*, volume 23, pages 557–556, 2004.

[SP03] Christian Sigg and Ronald Peikert. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization '03*, pages 83–90, 2003.

[Wik13] Wikipedia. Octant (solid geometry) — Wikipedia, the free encyclopedia, 2013. [Online; accessed 4-May-2016].