Abstract

Seismological applications use a 3D grid to represent the subsea rock structure. Many computations, such as detecting layers of rock in the seismic, can be done using the 3D grid exclusively. However, some algorithms for detecting vertical dislocations in the seismic require computations over a discretized polygon surface imposed over the 3D grid to assist geophysicists in interpreting the seismic data.

When using seismological applications on clusters, the 3D grid data is distributed between several cluster nodes. This thesis considers how algorithms involving discretized polygon surfaces can efficiently utilize the parallelism provided by clusters, and provides a general framework such algorithms can utilize.

The framework consists of three main parts: 1) efficient caching and transfer of voxels between cluster nodes, 2) efficient discretization or voxelization of polygon surfaces, and 3) efficient load-balancing.

First, three algorithms for caching and transferring voxels between nodes are introduced. The strategy which only transfers necessary polygon voxels is shown to be superior in most cases for our workloads, obtaining a speedup of 24.28 over a strategy which caches the bounding volume of the polygon, and a speedup of 2.66 over a strategy which transfers small blocks surrounding each polygon voxel.

Second, a new voxelization algorithm which may be suitable for Graphics Processing Units (GPUs) and multi-core CPU implementations is presented. On the GPU, a speedup of 2.14 is obtained over the corresponding algorithm on the CPU. For multi-core architectures without shared memory buses, a speedup of 2.21 is obtained when using 8 cores.

Finally, three algorithms for load-balancing the computations are introduced and future work is discussed. Our load-balancing algorithms achieve a speedup of 5.77 compared to not using any load-balancing for our workloads.

Acknowledgements

This thesis could not have been written without support from many other people.

First, I thank Associate Professor Dr. Anne C. Elster at IDI-NTNU for providing consistent support and encouragement throughout the semester, in addition to always being enthusiastic about my thesis subject — which led to many great discussions, comments and ideas which improved the quality of the work.

Second, thanks to everyone at Schlumberger Trondheim for providing me with this task, supporting this thesis by allowing me to use their hardware equipment and providing me with a place to work throughout the semester. As in the pre-Master's project, Tore Fevang, who was my co-supervisor at Schlumberger, has done an excellent job both in providing comments and ideas and in being a tough (and, unfortunately, unbeatable) competitor in table football. I also thank Wolfgang Hochweller, manager at Schlumberger Trondheim, for allowing me to write a thesis with them. Finally, thanks also goes to Jan Tveiten of Schlumberger Stavanger, who in collaboration with Tore Fevang came up with the initial ideas for this assignment.

Third, thanks to all my fellow students in room ITV-458 for providing both interesting and entertaining discussions throughout the semester.

Finally, since this thesis completes a five-year degree, a special thanks goes to all my other friends and family for the support throughout the previous five years of studying.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer clusters are increasingly being used as a cheap way of getting increased processing power and larger memory. Often, applications running on clusters do computations on a 3D grid of points. The application can assign to each cluster node an equal amount of grid points, and subsequently, each node can do computations on its grid points in parallel. However, in some cases, a need arises for imposing higher-level structures, such as polygons, over the grid structure. For example, tools for seismological analysis may use a polygonal structure to represent a surface below the ground containing a particular type of rock or seismic dislocations. How can such geometric structures imposed over the grid structure be efficiently represented on a cluster? And how can computations utilizing the geometric structure efficiently be done? These are the main questions we explore in this thesis.

In Section 1.1, we describe our motivation and problem in greater detail. Next, in Section 1.2, we describe the goals we seek to achieve. Subsequently, in Section 1.3, we describe the contributions of our work. Finally, in Section 1.4, we give an outline for the rest of this thesis.

1.1 Motivation and Problem Description

To spread data evenly between nodes, an application running on a cluster can assign each node in the cluster an equal share of the grid points. Formally, if there are $n \times m \times k$ grid points and p nodes in the cluster, assign to each node in the cluster a sub-grid of size $(n/a) \times (m/b) \times k$, where $ab = p$, $n \bmod a = 0$ and $m \bmod b = 0$[1]. Figure 1.1 on the following page illustrates the idea for $p = 4$ and $a = b = 2$.

The grid structure works well for many computations, but in many cases, we require higher-level geometrical objects to be imposed over the grid. For example, algorithms for applications such as seismic fault[2] detection require imposing a geometrical structure over the grid. Instead of operating on a point-by-point basis, these algorithms operate with lines, triangles, vertices or even complete objects, and do computations over the points intersecting the geometrical structures.

[1] This may require padding n and m appropriately.

[2] A *seismic fault* is a vertical dislocation in the rock structure below the crust.

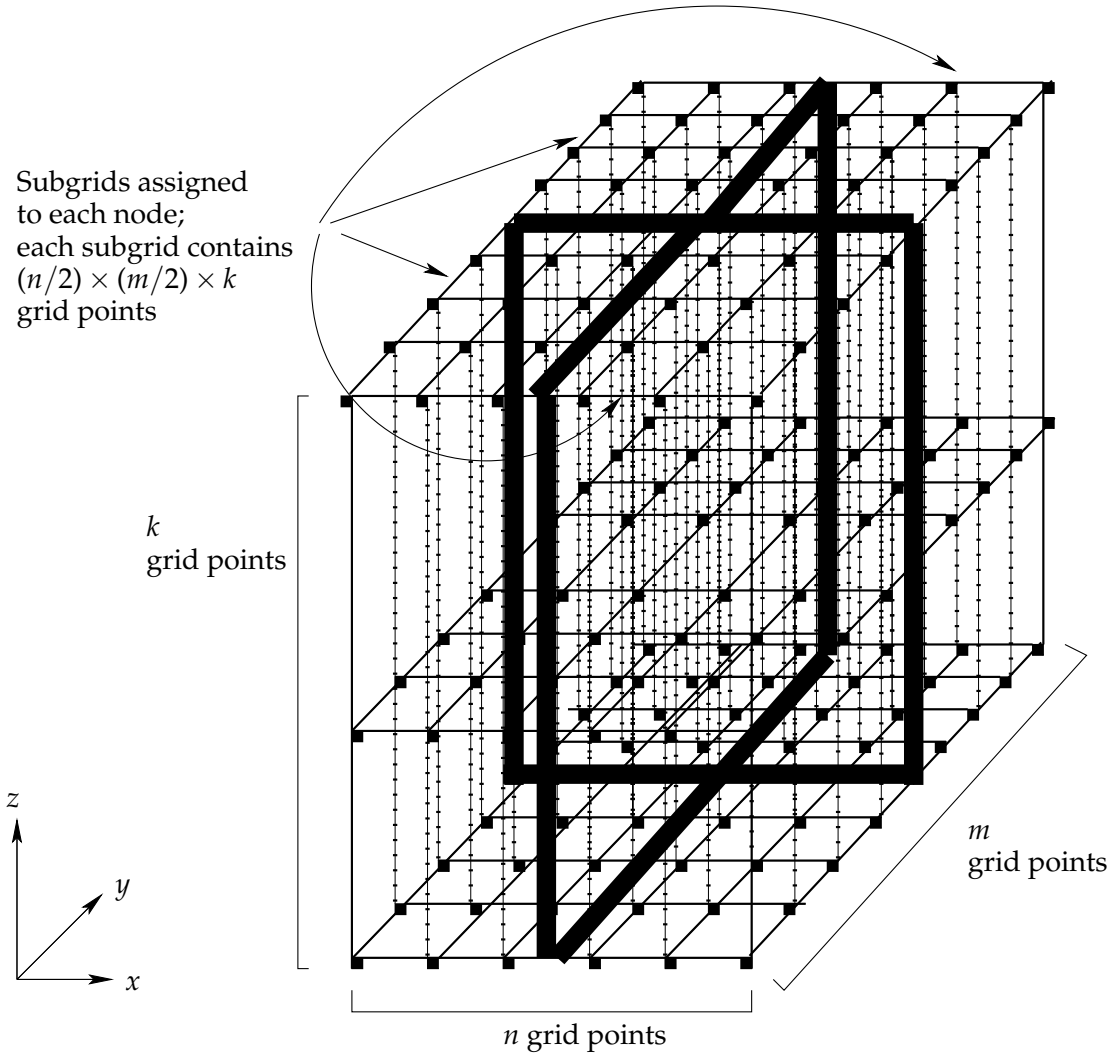


Figure 1.1: Representing 3D space by a 3D grid. In this case, $p = 4$ and $a = b = 2$. Each grid point can be seen to represent a subcube of the 3D space (a voxel).

The most usual way of representing geometry on computers is by using *polygons*. But how can computations be done over polygonal structures imposed over the grid in Figure 1.1 on a cluster? Several problems arise. Consider Figure 1.2.

In the figure, a polygon — in this case, a triangle — intersects grid points of both node n_0 and n_1. Suppose that we want to do some computation over the grid points intersecting the polygon's surface. One possibility is to simply split the polygon into two polygons, one on n_0 and one on n_1, and let n_0 and n_1 do computations over their respective polygons.

This approach has some problems:

1. First, computations done over a polygon's grid points will not be available on a single node after the computation is done. Therefore, communication must occur in order to exchange and merge partial computation results. Large amounts of communication will be required if there are many polygons.
2. Second, this problem becomes worse if combined with load-balancing and dynamic operations. For example, in Figure 1.2, there may be some other node, n_2, which is underutilized. Suppose that one of n_0 or n_1's partial polygons is migrated to n_2. From

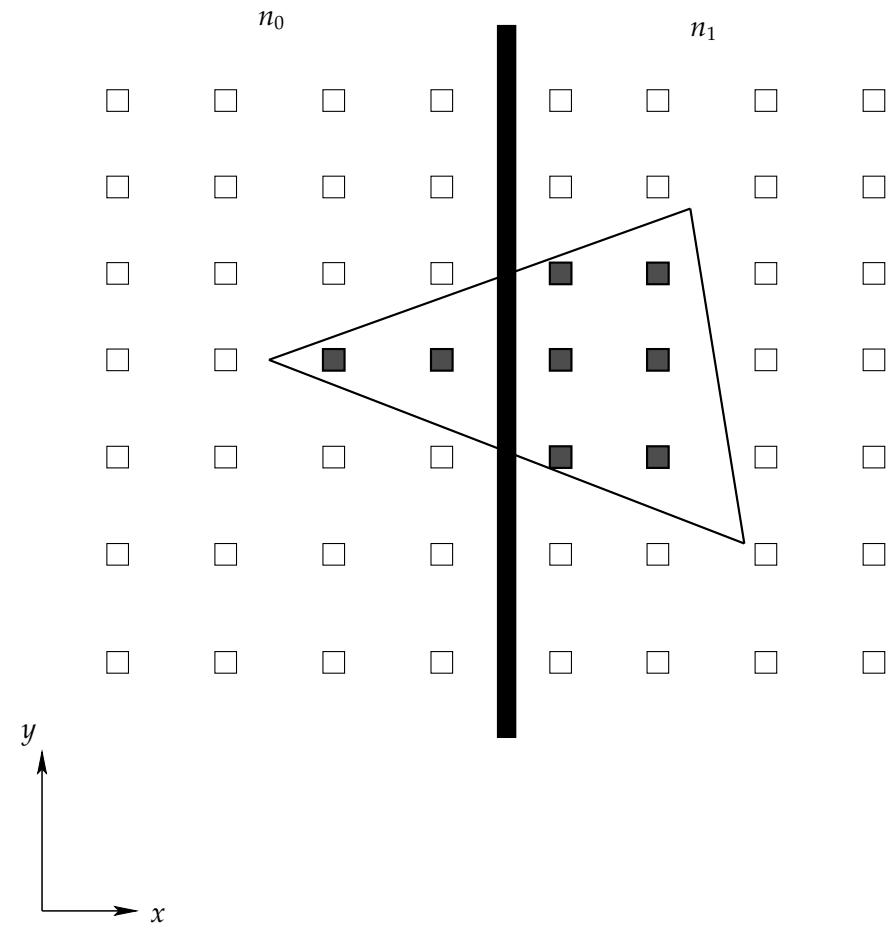


Figure 1.2: A triangle in a plane parallel to the xy plane, where one part intersects grid points (voxels) on node n_0 and the other part intersects grid points on node n_1.

the view of the algorithm utilizing the structure, there is still only one polygon. If one vertex of the original polygon is moved by the user, subsequently to partially moving the polygon to n_2, and a new computation is done over the polygon surface, the polygon may have to be split again.

3. Third, since the algorithms utilizing the polygonal structure only sees one polygon, which may be split into several polygons internally in the structure, we would have to somehow conceal the many smaller polygons spread on different nodes as one polygon to users of the structure. This will likely induce overhead.
4. Finally, some computations over the polygon surface may be dependent on neighboring grid points. In this case, border exchanges between the nodes will have to be done during computation. If several polygons are fragmented across several nodes, this will induce large amounts of communication.

Another, more elegant approach, is to not split the actual polygons but keep all grid points of a polygon on a *single node*. In this case, grid points intersected by a polygon are always available on at least one node, which eliminates the need for exchanges during or subsequently to the computation. Furthermore, there is no need to conceal anything from the algorithm utilizing the structure. Finally, load-balancing computations done over the voxelized polygon surface becomes easier since the entire polygon is treated as a single unit.

But this approach also has its problems. How do we decide which node we should store a polygon's grid points at? Furthermore, how do we implement support so that adding and removing polygons, moving vertices of polygons and extracting the points intersected by a polygon can operate as quickly as possible? Finally, we may have tens of thousands of polygons. How can the computations be load-balanced so that all nodes are approximately equally loaded?

These are the questions we answer in this thesis.

1.2 Goals

The goals of this thesis are:

1. *Develop a framework for representing polygonal structures used for computations over volume data on clusters, supporting dynamic operations.* The framework will aim to solve the problems we considered in the previous section; namely,
   (a) The framework will attempt to efficiently transfer grid points (voxels) intersected by a polygon on one node only, in a way that ensure that computations done over grid points intersecting the polygon's surface are done as efficiently as possible.
   (b) The framework will attempt to distribute polygons equally among the nodes, so that load is balanced when doing computations over the grid points intersecting the polygon. This requirement must be balanced with the time it takes to actually migrate polygons between nodes.

   Note that although our original goal was originally to merely discuss how the framework could support dynamic operations, which means that polygons may be added, moved or deleted, the framework we present supports *all* these operations.
2. *Evaluate efficiency*. We will also consider the efficiency of the framework and different caching and load-balancing algorithms on an actual cluster. We also consider how off-loading parts of the framework to multi-core and GPU architectures impacts the framework performance.

1.3 Contributions

The contribution of this thesis is a framework for polygonal structures on clusters, supporting efficient voxelization[3] of triangles of all polygons and doing computations over the voxelized polygon surfaces. The framework comprises:

1. *A fully dynamic framework for polygonal structures on clusters*, made for doing computations over voxelized surfaces efficiently and supporting dynamic operations. The main components of the framework are:
   (a) *Three caching and transfer strategies* for efficiently caching and transferring data between cluster nodes, in order to do computations over the voxelized polygon surfaces.

[3]Voxelization is the process of finding discrete grid points intersecting a geometric surface.

   (b) *Three scalable load-balancing algorithms* for efficiently load-balancing the computations.

2. *A new triangle voxelization algorithm optimized for GPUs and multi-core CPUs*. We presented a new triangle voxelization algorithm which exploits the rasterization hardware of GPUs. Compared to previous approaches to do GPU voxelization, our algorithm can handle larger triangles and enables the list of voxelized coordinates to be downloaded to CPU memory to be used for other purposes than visualization. Compared to previous CPU voxelization algorithms, our algorithm removes inner-loop branches in exchange for more floating-point computations.

The framework is mainly targeted at seismological applications, but may be used in any situation where efficient computations over voxelized polygons is required.

1.4 Outline

This thesis is structured in the following manner:

- **Chapter 2 — Background and Previous Work** describes background material we will use in the rest of the thesis. First, a workflow in a seismological application which motivated the creation of our framework is described. Next, polygonal structures and ways of representing such structures on computers are described. Subsequently, we give an introduction to parallel programming and load-balancing, which are techniques we will use in our implementations. Finally, we state precisely which operations our framework shall support.
- **Chapter 3 — Framework Design Part I: Cache and Transfer Strategies** describes different strategies for efficient transfer and caching of grid values, suitable for different workloads. Additionally, we describe how our framework exploits the fine-grained parallelism provided by GPUs and multi-core CPUs to do *voxelization*, which is an important part of the framework. We propose new, previously unsuggested voxelization algorithms that are suitable for such architectures.
- **Chapter 4 — Framework Design Part II: Load-Balancing** describes different load-balancing strategies for load-balancing the computations done over the voxelized polygon surfaces. We present, analyze and discuss three different strategies for load-balancing.
- **Chapter 5 — Benchmarks and Discussion** describes benchmarks of our implementations of the methods described in Chapters 3 and 4, followed by a discussion of the benchmark results.
- **Chapter 6 — Conclusions and Future Work** summarizes the findings in this report, and suggests future work.

In addition, the following appendices, which will be referred to throughout the thesis, are included:

- **Appendix A — Related Papers** gives an overview of some of the papers we refer to in the thesis.
- **Appendix B — Software User's Guide** gives instructions for compiling and running the programs included in the electronic attachment of this thesis.

- **Appendix C — Source Code Overview** gives an overview of the source code included in the electronic attachment and developed in this thesis work. This appendix is intended to be a starting point for future extensions of the software.
- **Appendix D — Other Data Structures for Polygon Meshes** gives alternative data structures for representing polygon meshes on computers. If an application utilizing our framework requires a structure which is more flexible than the structure we chose to use, or requires a structure which is less flexible but uses less memory, this appendix gives an overview of alternative structures.
- **Appendix E — Additional Algorithms, Methods and Code** describes some of the algorithms and methods (developed by others) used in our implementations, in addition to providing additional pseudocode for some of the algorithms we present in the thesis.
- **Appendix F — Additional Benchmarks** gives some extra benchmarks in addition to those we present in Chapter 5. We will refer to this appendix in Chapter 5.

Also, the thesis consists of an *electronic attachment*. This attachment contains source code and programs developed in the thesis work. Installation and running instructions for the electronic attachment are given in Appendix B, while an overview of the source code developed is given in Appendix C.

Chapter 2

Background and Previous Work

In this chapter, we give background information which the rest of the chapters built on. Section 2.1 describes the application that motivated the creation of the framework. Next, in Section 2.2, we investigate previous work done to find if similar frameworks have been created previously. Then, polygonal structures and efficient ways of representing polygonal structures on computers are explored in Sections 2.3 and 2.4.

We also introduce the reader to parallel programming techniques we will use in our implementations in Section 2.5. Finally, we present the interface our framework shall support — which can be viewed as a high-level requirements specification — in Section 2.7.

2.1 Target Application: Seismic Fault Detection

In order to operate on large 3D data sets in a cost-effective manner, applications for seismological analysis and visualization use *computer clusters*, often built from commodity PCs linked in a network, to do analysis computations. A cluster is used to increase both the processing power and available memory space of the application. Commodity PCs are cheap, but lack enough memory and processing power to process such large data sets by themselves. Therefore, connecting several such PCs into a network to form a cluster is a cheap way of gaining performance.

The cluster operates on a 3D data set partitioned evenly among the nodes. The 3D data set is obtained executing seismic shots on the seabed and measuring the acoustic reflection or, more recently, measuring the electromagnetic energy propagation through the subsurface[1]. A user, who is typically a geophysicist or geologist, uses a client to connect to the cluster, and can subsequently pass commands and retrieve data from the cluster nodes for rendering and inspection.

One operation done on the cluster, which is important in seismic interpretation, is detecting *horizons* and *faults* in the underground structure below the crust. Consider Figure 2.1 on the next page, which shows a 2D slice of the 3D data set. In the figure, green lines represent horizons, while the red line is drawn along a fault. A part of the acoustic signal retrieved from the seismic shot is also shown in the box slightly to the left.

[1]Prior to being used in an analysis- and visualization applications, the raw seismic data undergoes heavy preprocessing to remove noise.

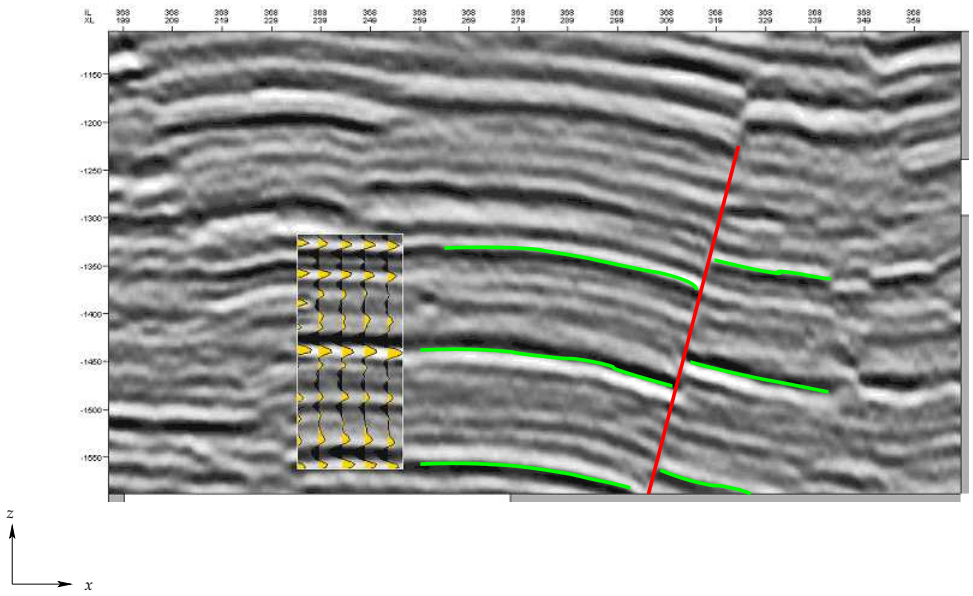


Figure 2.1: Faults and horizons in seismic data. (Seismic image is © Schlumberger. Used in accordance with acceptable use policy.)

In Figure 2.1, the green lines represent *horizons* in the seismic data, which are layers of a particular type of rock. Additionally, we can observe *seismic faults*, one of which is indicated by the red line, caused by vertical relative movement between two blocks of rocks. If the seismic data set is very large, detecting both horizons and seismic faults manually is a difficult task. Therefore, it is desirable to automate this process. Consequently, automated *horizon trackers*, which are able to automatically detect horizons (even across faults) in the seismic data, were developed through the 1990s see e.g. [1, 2].

Finding seismic faults automatically, however, turns out to be a more challenging task than finding horizons. Detecting faults is complicated by the fact that the seismic data contains more noise around faults and the property that several faults may intersect at several locations. Additionally, faults may be very small or very large.

One effect faults have on oil production is that faults prevent hydrocarbons from moving across fault boundaries [3]. To maximize reservoir output and be able to plan drilling sites, it is therefore of great importance to detect the faults in the seismic data. Manually, this task takes several hundred hours of work by geologists — even for relatively small oil fields.

The goal of any fault detection process is to extract from the 3D data set a *fault surfaces* or *fault polygons*, which subsequently to detection can be filtered, inspected or edited by a user or an algorithm attempting to interpolate the fault surfaces to match the actual fault. The fault polygons can either be large or small, depending on the resolution the user wants. Figure 2.2 on the facing page illustrates a set of faults, which are modeled as polygon meshes. Each polygon can cover hundreds or thousands of grid points.

Although some approaches to fully automatic fault detection have been proposed, best results are still obtained when the fully automatic detection methods are combined with user-intervention into semi-automatic methods [3]. For example, certain fault detection methods require user intervention to, subsequently to automatic detection, manually process automatically generated, estimated fault patches [4]. Other semi-automatic approaches, requir-

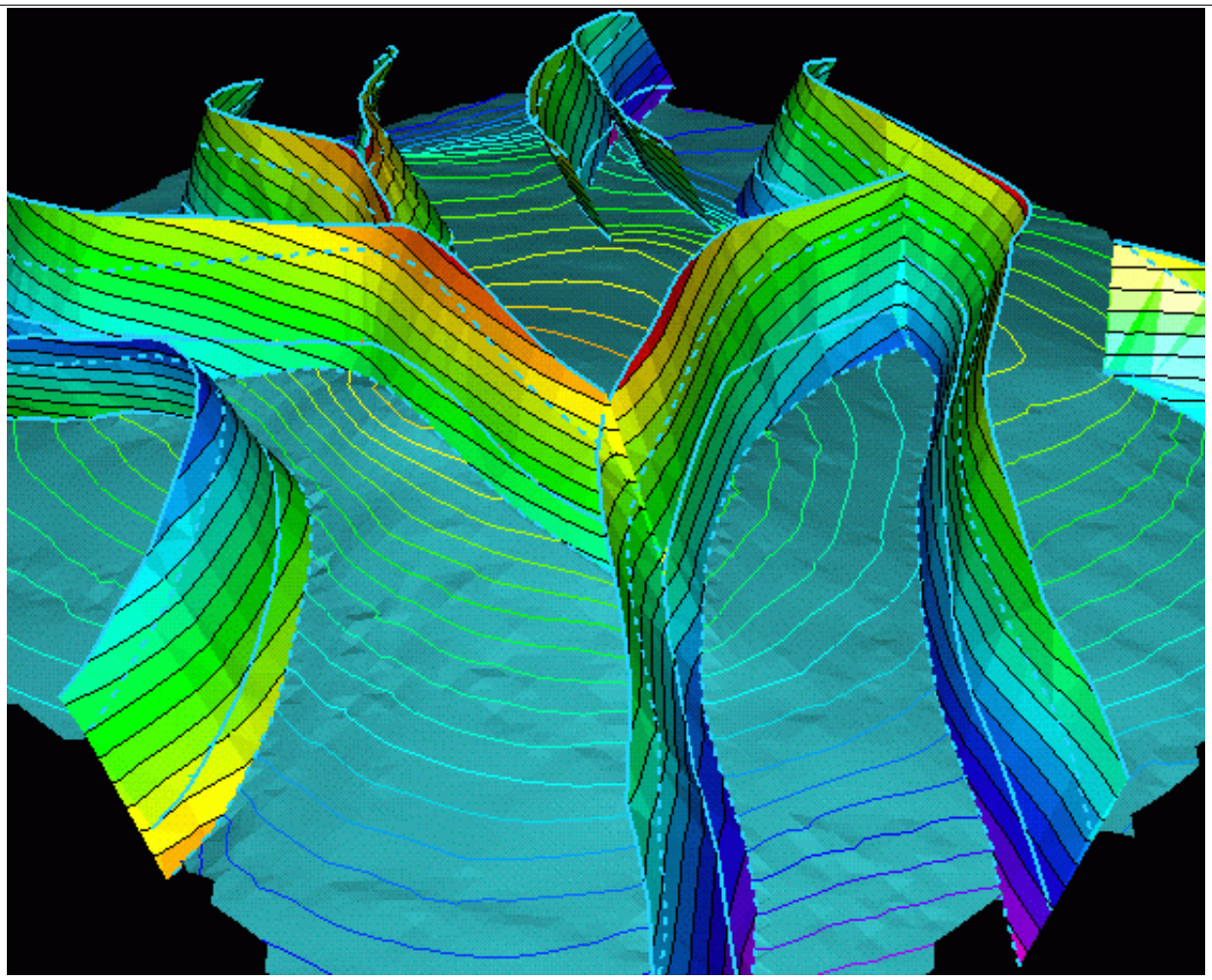


Figure 2.2: Seismic faults, modeled as polygon meshes, in a seismological modeling application. (Image © SINTEF and Roxar Software Solutions AS. Used in accordance with acceptable use policy. From `http://www.math.sintef.no/Geom/siscat/`)

ing similar user interventions, have been proposed [5].

One approach to semi-automatic fault detection is known as *pillar gridding*, and consists of the following steps:

1. Initially, a geologist marks *fault pillars* or *fault sticks*, which are near-vertical lines intersecting a fault, in the seismic application. The vertical lines can be viewed as sticks stuck into a fault at different points. Since the initial fault pillars are manually chosen, they are referred to as *interpreted pillars*. Fault pillars are shown in Figure 2.3 on the next page.
2. Next, an algorithm is run which:
   (a) Inserts additional fault pillars between the interpreted pillars (by a proprietary algorithm). The algorithm may also correct the interpreted pillars placed by the user, if these were placed somewhat off the actual fault, by moving the vertices of the fault pillars.
   (b) Connects all the fault pillars together by a triangle mesh. In sections of a fault that are near-planar, few and large polygons are used, while more polygons can

be used in sections which are not planar[2]. The detail level used in the mesh can be adjusted by the user, so that a more accurate mesh is generated at the cost of increasing running time.

3. Then, some computation is done over all points intersecting each triangle. For example, the points may be:
   - Transferred to the client to be rendered and viewed by the user,
   - filtered to be sharpened by using e.g. a Fourier transform,
   - compressed by using transform coding compression (Appendix E.4) for storing or transfer, or
   - do some analysis computation — for example, compute the average value of all the voxels intersected by the fault triangles to determine the average type of rock located in the fault, or compute the variance in order to determine how much the rock structure varies along the fault.
4. The user may subsequently refine the generated triangles, by for example moving points of triangles that the algorithm in Step 2 wrongly detected as being parts of the fault. Optionally, the algorithm may do this refinement process automatically, and Steps 2 and 3 may be repeated several times to iteratively refine the fault surface to match the actual fault.

After the process above, a fault surface has been generated.

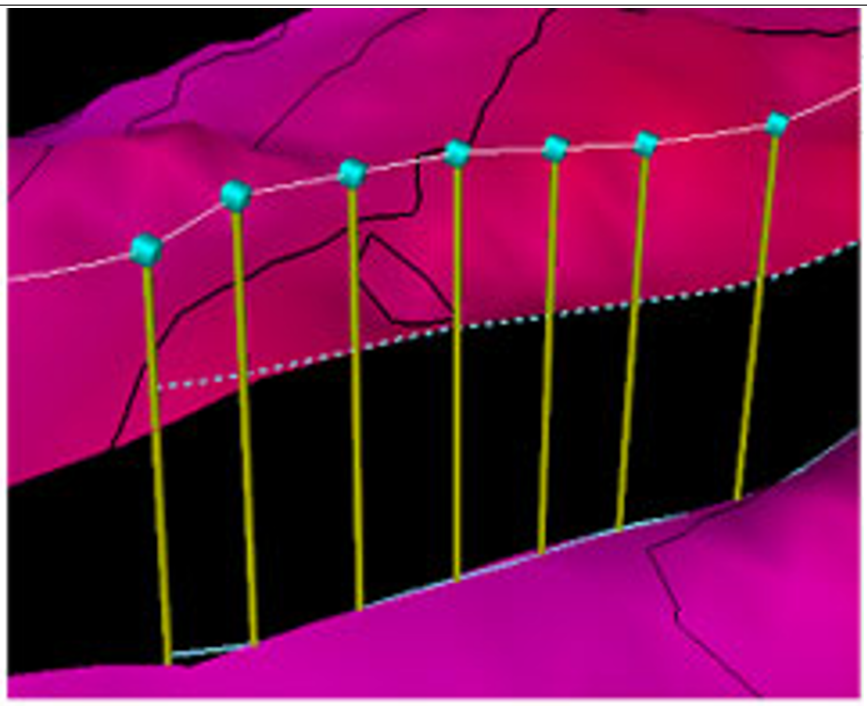


Figure 2.3: Fault pillars in a seismological modeling application. (Image is © Schlumberger. Used in accordance with acceptable use policy.)

From the user's point of view, using polygons can increase the responsiveness of the application, and are also intuitive to use. For example, the cluster nodes can compute an average

[2]Some methods for fully automatic fault detection [3] essentially do the reverse: First, relatively small fault patches are detected, which are subsequently merged to form larger fault patches.

value of all the voxels intersected by each polygon. Then, when a client wants to visualize a fault, instead of sending all the voxels intersected by the fault polygons to the client, only the average value of each polygon is sent along with the coordinates of the polygon's vertices. This reduces the amount of data to be transferred through the network, and enables quick visualization of the fault. If a more detailed view of the fault is required, more polygons can be used.

If the method we describe was to be run on a cluster, it would require a framework for imposing polygonal structures over the data set. In particular, we can use the cluster to increase the speed of detecting all voxels intersected by all polygons, and do the computations quicker by assigning to each cluster node a set of polygons in the fault polygon mesh. For these types of problems, the framework we describe in the rest of this thesis is well suited. However, as we mentioned in the previous chapter, the framework is not necessarily limited to such applications: *Any* application imposing polygonal structures over 3D grids can use the framework. This includes other application using a mix of volume and surface data, such as medical visualization applications, which use polygons to represent the plane where radiation therapy beams intersect the human body [6].

2.2 Previous Work on Polygon Representations for Computations on Clusters

In this section, we consider previous work related to bringing polygonal structure representations to clusters. The problem we seek to solve was inspired by problems arising in seismological client-server applications running on a cluster. These applications typically combine visualization and computation/analysis, and have traditionally used a voxel representation with no explicit polygonal model; see e.g. [7].

We do not know of any previously publicized research on bringing polygonal structures imposed on a voxel grid to clusters, perhaps since the primary users for such structures are seismological applications, which traditionally have been proprietary and few in number. We therefore look to other fields which have traditionally been the largest users of polygonal structures:

1. *3D Graphics Rendering Applications*, running on clusters, are used for rendering detailed, high-quality 3D scenes, using a polygon structure. Below, we investigate previous work done in bringing polygon structures used for rendering to clusters.
2. *Geographic Information Systems* or GIS, which store very large polygon models for representing geography, often do not run on clusters, *but* use a distributed database. How the model is distributed in such a distributed database is of clear relevance to our problem. We therefore investigate previous work done with polygon distribution in GIS systems below.

One fair question to ask is whether previous work done in parallelizing Finite Element Method (FEM) applications should be included in the above list. FEM methods also use polygons, but with FEM methods, values are stored on polygon vertices and not along surfaces [8]. Additionally, polygons are very small [9]. This is a different problem from what we are solving with our framework.

2.2.1 Previous Work in 3D Graphics Rendering Applications

When considering polygon representations, the natural place to start is 3D graphics rendering applications, which are arguably the most heavy users of polygon structures. Such applications have used clusters before, but for different purposes than us. Commercial movies, the first of which was "Toy Story" from 1996, was rendered using clusters using Pixar's RenderMan application [10]. RenderMan and similar animation systems use *frame-based parallelization* [11]: Each cluster node is assigned to render an individual *frame* of a movie or animation sequence. Each node is then given a polygonal description of its frame, and then renders the frame. Since we have no frames in our case, this strategy does not work.

Samanta et.al. presented in [11] a method for partitioning polygons for rendering purposes on clusters, which does not use frame-based parallelization. The assignment of polygons to each node is based on grouping several polygons into objects and subsequently, each node renders a portion of the 2D screen space. Since we do not have a screen space in our application, this method solves a fundamentally different problem (rendering) than we do (imposing polygon structures over a voxel grid).

Another field of 3D graphics is *volume rendering*, which is perhaps more relevant to our problem. In volume rendering, a set of voxels is rendered (instead of polygons as in [11]), usually using the Marching Cubes algorithm [12], which converts voxels to polygon structures.

Neumann suggests in [13] a taxonomy for parallel volume rendering methods. Roughly, the methods may be separated into *image partitioning methods*, where each node in a cluster is assigned a specified screen area to render, and *object partitioning methods* where each node renders a local image of the data set stored on the node, followed by a composition step which merges all the node images into a single, fully rendered image. The techniques used in parallel volume rendering methods differ from what we do, since the techniques are exclusively used for visualization. Such methods consider only the distribution of voxels and does not consider the resulting polygon distribution at all (if the volume rendering method run on each node is the Marching Cubes algorithm). This research is therefore of limited use. Consult Appendix A.1 for a brief discussion of papers we have considered.

Although we have not found research in graphics which solve our specific problem, we will use some techniques from this field in our framework: In particular, we require algorithms for voxelizing a polygon face. Research has been done in computer graphics on voxelization. We describe the specific previous work in the next chapter.

Furthermore, we will also use the Winged-Edge model to store polygon meshes on each node. The Winged-Edge data structure is described in the next section.

2.2.2 Previous Work in Geographic Information Systems (GIS)

A *Geographic Information System*, or GIS, is a "*computer-based information system that enables capture, modeling, manipulation, retrieval, analysis and presentation of geographically referenced data*" ([14], page 1). GISs are also avid users of polygonal structures, although the use is usually for a different purpose than in 3D rendering applications. Typically, a geographic map is stored as a set of polygons in a database system. Since the database or processing part of a GIS may be distributed, and since GIS applications often do operations on the polygon model, problems in GIS are somewhat analogous to our present problem.

A GIS system typically uses a polygonal representation such as the winged-edge structure, along with other structures supporting certain operations quickly. Example GIS operations on a polygon structure are [14]:

- *Point queries*. Given a point P in n dimensions, find all (polygonal) objects containing P.
- *Range queries*. Given a point P in n dimensions and an n-dimensional rectangle R, find all (polygonal) objects such that, for all points P' in an object, P' is contained in R.

Research on using GISs on clusters focuses on implementing the above to operations efficiently. While this feature may be desirable in our framework, this is a structure built on top of the polygonal data structure. GIS-systems also typically store relatively planar polygons and not volume data. Therefore, GIS research is of limited use to us currently. However, if an application requires point- and range-queries, R-trees may be easily built on top of our polygonal model. We include pointers to relevant papers in Appendix A.2.

2.2.3 Previous Work: Summary

The motivation for our framework arose in seismological applications. We want to use polygons to represent faults, and extract voxels intersecting the polygons quickly. As 3D rendering methods either work with boundaries (polygons) or volume (voxels), and partition data between nodes either dynamically, often based on the view point of the image to be rendered, or a static division, often based partitioning the screen-space to equally-sized blocks. Rendering methods for clusters which work with boundaries do not consider volume data — and methods working with volume do not consider polygons. Our problem is different, since we want to impose a polygonal structure over an already-existing voxel structure. There is no view-point or screen space. Parallel 3D rendering methods focus on a different problem (fast rendering). We do not focus on rendering at all, since in seismological applications on clusters, rendering is often done on the client machine connecting to the cluster and utilizing its local GPU.

On the other hand, GIS systems on distributed systems focus on optimizing range- and point-query performance and do not consider volume data — which is of limited relevance to us now, but may be interesting as an extension to our framework.

Therefore, we have not found any previous work which solve this specific problem. As mentioned, however, we will consider techniques from other fields — in particular, voxelization techniques from computer graphics — in designing our framework.

2.3 Polygonal Structures

There are several ways of representing 3D structures on computers. The two main classes of object representations are *boundary representations*, or *B-reps*, which represent structures by the structure boundary, and *space-partitioning representations*, which represent structures using a set of small elements [15]. The most frequently used B-rep is the *polygonal representation*, while one of the most well-known space-partitioning representations is a *voxel representation*. Polygonal representations use polygons — which are planar figures consisting of line segments, bounded by a closed path — as the fundamental building block, while

voxel representations use small subcubes (Figure 1.1). Polygonal representations are used for dealing with surfaces, while voxel representations are used when dealing with volumes. In some applications, both polygonal and voxel representations are used. We described one application using both representations in greater detail in Section 2.1.

The main reason the polygonal representation has achieved popularity in computer graphics applications is that polygons are simple, linear structures. Other representations, such as Bézier and B-spline curves, are slower to work with, and are usually converted to polygons [9, 12] when used on a computer. With a polygon representation, a structure surface is modeled as a network of interconnected polygons, called a *polygon mesh*.

There are several types of polygons. Polygons can either be *simple* or *complex*. A polygon is simple if the boundary of the polygon can be described by a single line with no self-intersections. Polygons which are not simple are complex. Complex polygons may have self-intersecting boundaries and holes. Simple polygons can either be *convex*, which is the case if the interior angle between two consecutive boundary edges of the polygon is at most 180 degrees[3]. Polygons which are not convex are *concave*. A polygon is *cyclic* if all vertices lie on a circle, and *equilateral* if all edges are of the same length. Note that all cyclic polygons are simple and convex, while equilateral polygons may be either simple convex, simple concave, or complex. Polygons which are both cyclic and equilateral are said to be *regular*. Different polygon types are illustrated in Figure 2.4.

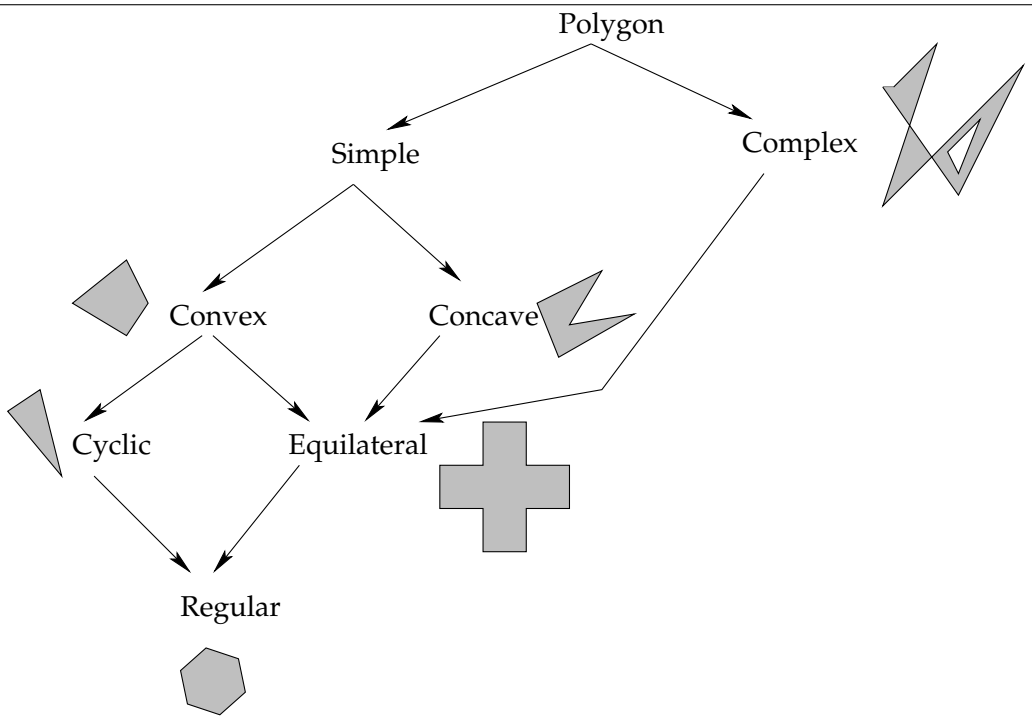


Figure 2.4: Different types of polygons, based on a similar figure from Wikipedia [16].

In the rest of this thesis, we focus exclusively on triangles, which are cyclic, convex and simple polygons. Considering triangles is sufficient, since all complex, concave and simple polygons can always be represented by a set of triangles using *polygon triangulation* algorithms. Several efficient triangulation algorithms exist. In 1990, Chazelle gave an $O(n)$

[3]If there is some angle equal to 180 degrees, or if two or more vertices are at the same coordinates, the polygon is also said to be *degenerate*.

algorithm for triangulating *simple* polygons with n vertices [17], which is asymptotically optimal since all triangulations of simple polygons add exactly $n-2$ vertices to the original polygon[18]. However, simpler and more general $O(n \log n)$ algorithms exist for triangulating *complex* polygons with holes [18][4].

When using polygons to represent an object, several polygons are joined to form a *polygon mesh*. A polygon mesh can be considered as a graph $G = (V, E)$ with vertices V and edges $E = \{(u, v) | u, v \in V\}$ connecting the vertices of V. A polygon mesh and the corresponding graph is shown in Figure 2.5. Note that the graph only concerns *topological* relationships between vertices and edges, and is independent of the actual Cartesian coordinate of any of the vertices and edges.

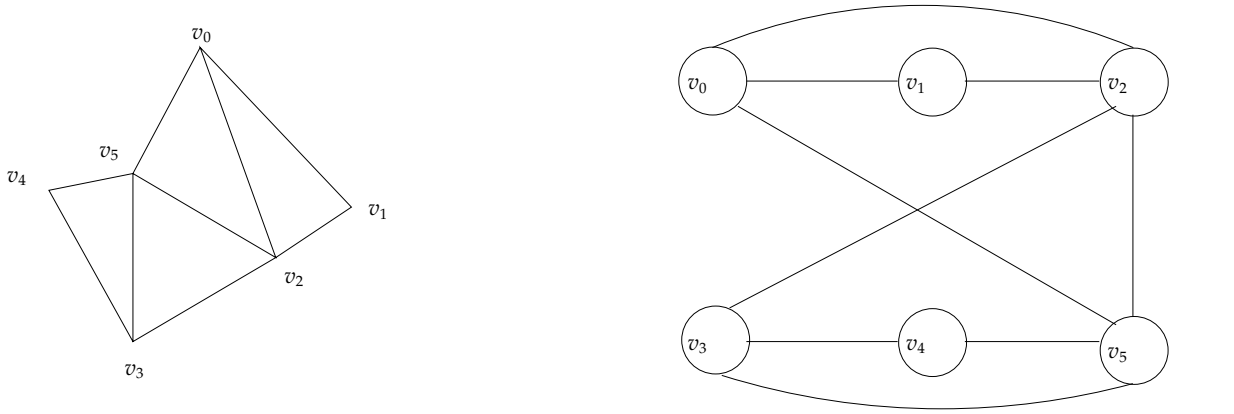


Figure 2.5: A polygon mesh (*left*) and corresponding graph (*right*).

As previously mentioned, we will only consider meshes consisting of triangles. We also only consider *simply connected planar meshes*, which have no self-intersecting edges and all vertices are connected to at least one edge. A graph is *simple* if it does not contain multiple edges nor loops[5], which are edges which start and end at the same vertex. The graph is *connected* if for all vertices u in the graph, there is a path from u to any other vertex v in the graph. Finally, the graph is *planar* if it can be drawn in a plane with no intersecting edges. The graph in Figure 2.5 is a simply connected planar graph. It is simple since there are no multiple edges and no loops, it is connected since all vertices can be reached from all other vertices, and it is planar since it can easily been drawn with no intersecting edges. The corresponding mesh is therefore also said to be a simply connected planar mesh.

In simply connected planar meshes, regions on the left or right side of an edge (if the edge is not at a boundary of the mesh) are called *faces*.

Polygon meshes can also be classified to be *manifold* (no two vertices coincide and sequential vertices are always connected by an edge, and each edge is shared by at most two polygons) or *non-manifold*. In this thesis, we only consider manifold meshes, which are usually assumed in applications using polygonal representations and structures for representing polygons. If a non-manifold surface needs to be represented, it can easily be split into several manifold surfaces [19].

[4] If the complex polygon has self-intersecting boundaries in addition to holes, performing "inside-outside"-tests and introducing additional edges in the polygon can be used to remove the self-intersections. See [15] for further descriptions of inside-outside tests.

[5] Loops are edges with the same start and end vertex.

We do *not* require the meshes to be *conforming* [18], that is, the interior of a triangle in a mesh in our framework *can* intersect a vertex of another triangle.

Next, we consider how polygon meshes can be efficiently stored on a computer.

2.4 The Winged-Edge Data Structure

Several data structures have been proposed for representing polygon meshes. These structures can be viewed as data structures for storing planar graphs, as in Figure 2.5. However, the data structures can store additional information. As an example, consider the problem of finding all triangles in the mesh represented by the graph in Figure 2.5. To do this, all cycles of three edges in the graph must be found. This is slow compared to storing with each edge a separate list of faces. Therefore, data structures for polygons also store lists of *faces* in addition to edges and vertices to keep track of which vertices and edges form a specific face in the structure, but strictly speaking, storing lists of faces is redundant. Memory space is thus sacrificed for computation speed.

First introduced in 1979, Baumgart's *winged-edge structure* [20] is one of the most popular data structures used for representing manifold polygon meshes on computers. Figure 2.6 shows a UML class diagram[6] of the winged-edge structure. Strictly speaking, the winged-edge structure originally only specified the edge relationships and not the relationships of the mesh, vertex and face tables, and thus the diagram is actually a common extension of the winged-edge structure.

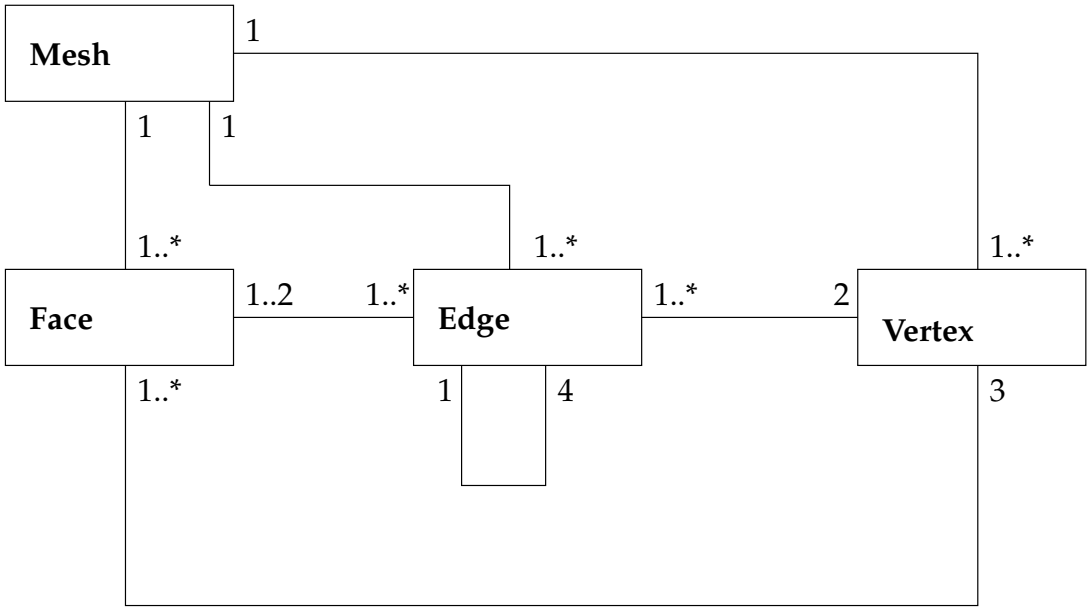


Figure 2.6: UML diagram of the winged-edge structure

In general, all data structures for representing polygon meshes contain mesh, edge, face and vertex tables. The difference between the representations we describe lies mostly in how the

[6]Consult [21] for more information on UML diagrams. Briefly, boxes represent objects, while edges represent associations between objects. The numbers indicate the cardinality of the association. For example, in Figure 2.6, each face is associated with at least 1 edge, while each edge is associated with 1 to 2 faces.

edge table is structured, while the *mesh*, *face* and *vertex* tables are similar for almost all the data structures.

In the winged-edge structure of Figure 2.6, a mesh object has pointers to all the polygonal faces, edges and vertices of which the mesh consists. Each entry in the face table stores the full list of edges and vertices which each polygon face consists of. Each vertex stores the Cartesian coordinates of the vertex, in addition to pointers to all edges having that vertex as a starting or ending vertex, in addition to storing pointers to all faces incident to the vertex. Each edge, in turn, consists of pointers to two faces — the faces to the *left* and *right* of the edge — and two vertices — the starting and ending vertices of the edge. Additionally, with each edge object, we store pointers to up to *four* other edges. To understand why, consider Figure 2.7.

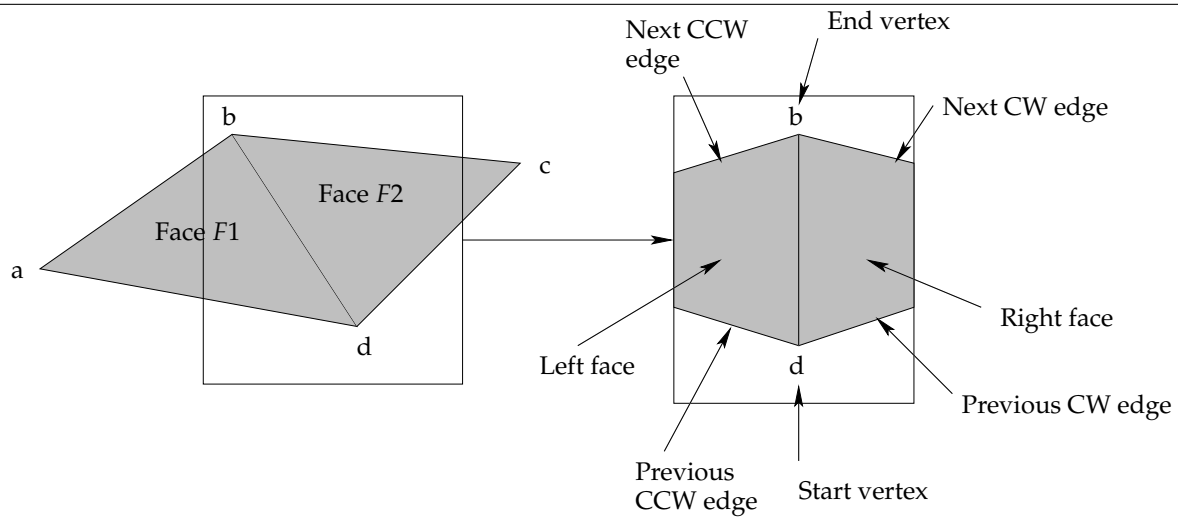


Figure 2.7: A simple polygon mesh, illustrating concepts used in the edge tables of the winged-edge structure.

In the figure, a very simple polygon mesh is shown. With the edge (d, b), we store pointers to $F1$ (left face) and $F2$ (right face), in addition to d as the starting vertex and b as the ending vertex. Additionally, with the edge, we store *next clockwise* (next CW), *previous clockwise* (previous CW), *next counterclockwise* (next CCW) and *previous counterclockwise* (previous CCW) edges as indicated in the figure. This allows for efficient traversal of *all* the edges in the entire mesh, in an order such that each edge visited is connected to the previous edge visited. The terms clockwise and counterclockwise refer to the "topological direction" in which we are moving when traversing. For example, to traverse all edges of a polygon, we may either start at a polygon edge and follow either only the counterclockwise (traversing the left polygon) or clockwise (traversing the right polygon) edge links. Thus, to traverse over all edges of $F2$, we follow all the clockwise edges. Analogously, all counterclockwise edges are followed for $F1$.

Why is this information stored? It might seem unnecessary to store this much information, when all that is really needed is storing a simply connected planar graph. The reason is the same as we mentioned earlier: this redundancy allows for more efficient algorithms. Consider, for example, the problem of finding whether an edge belongs to a face or not. Since we store all edges with each face, this is a quick operation involving scanning the face's edge list.

Also, with the winged-edge structure, additional edges and vertices may be easily inserted. We can, for instance, splice in a vertex on an edge (and thus create a new edge) in $O(1)$ time. Consider Figure 2.8. In the figure, we split an edge into two parts and add two edges to the mesh. This is an important operation, for example, if we want to increase the detail of the polygonal model. If low detail is required, few polygons are used. If higher detail is required, polygon edges are split into several smaller polygons. With the winged-edge structure, the operation of Figure 2.8 amounts to:

1. Insert e_1, e_2 and e_3 into the edge table, and insert v_2 into the vertex table,
2. Set e_1's next CCW/CW fields to e_0's next CCW/CW fields, and the previous CCW/CW fields to e_3 and e_2, respectively,
3. Set e_0's next CCW/CW fields to e_3 and e_2, respectively,
4. Set e_2's previous CW field to e_0, next CW field to e_1, and the CCW fields according to the structure at the other end of e_2,
5. Set e_3's previous CCW field to e_0, next CCW field to e_1, and the next CCW/CW fields according to the structure at the other end of e_3.

Assuming that the edge and vertex tables allow adding and looking up edges and vertices in $O(1)$ time — which is possible using hash tables [22] — this is an $O(1)$ operation[7]

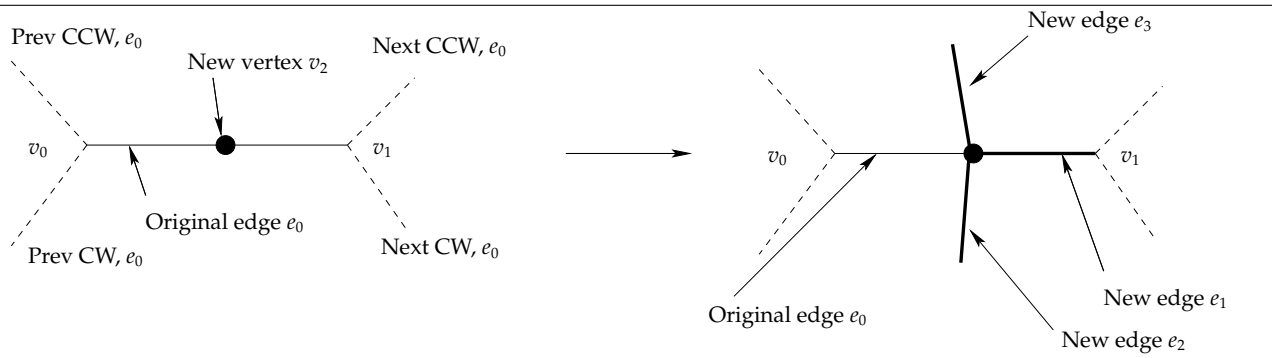


Figure 2.8: Adding vertices and edges to the Winged-Edge structure

Finally, the winged-edge structure allows for efficient traversal of the *entire* mesh in an order where the next edge is always connected to the previous edge. To traverse the entire mesh in this manner, start at an arbitrary edge. Then traverse its next CW/CCW edges recursively. Eventually, all edges will be traversed.

The price paid for this increased flexibility and speed is increased memory usage. We have chosen the winged-edge structure since it is a commonly used structure which balances reasonably between memory requirements and flexibility. In Appendix D, we describe some alternative structures and why we did not use them.

[7]Conceptually, this operation is equivalent to inserting new elements in doubly-linked lists — also an $O(1)$ operation [22]. In that operation, next and previous pointers are also updated to insert new elements. In the winged-edge structure, however, we have two next and two previous pointers.

2.5 Parallel Programming Techniques

We mentioned that clusters are commonly used as a cheap alternative to obtaining increased processing power. The increase in processing power comes from being able to do several tasks on each node in parallel. We refer to this as *coarse-grained parallelism*. On the other hand, techniques for *fine-grained parallelism*, which refers to doing parallel computation on *each node*, have recently become more popular through the usage of multi-core CPUs and the use of heavily parallel Graphics Processing Units (GPUs). In Section 2.5.1, we discuss some of the techniques for coarse-grained parallelism which we have used in our implementations, while we discuss fine-grained techniques that have been utilized in Section 2.5.2.

2.5.1 Coarse-Grained Parallelism

The most commonly used programming interface for communicating between nodes in parallel computer systems, which we have used in our implementations, is the Message Passing Interface (MPI) programming interface [23, 24]. As the name suggests, MPI centers around *message passing*: Processes communicate by explicitly transmitting messages to each other.

MPI can be used for communication between processes also on shared-memory systems. Thus, MPI can also be used for fine-grained parallelism. The primary difference of using MPI compared to OpenMP is that in the latter, no explicit messages are sent between processes from the application. Everything is controlled through preprocessor directives and simple OpenMP API calls. In MPI, however, messages must be explicitly sent between processes. MPI thus gives more control to the programmer, but programming can be more time-consuming.

MPI provides calls for sending and receiving messages to and from other processes. The sends and receives can either be *blocking* or *nonblocking*. In blocking communication, the application calls an MPI routine for either sending or receiving. The routine returns when the send or receive operation has completed. In nonblocking communication, the calls to the MPI routines return immediately and the application may subsequently poll or wait for the operation to complete. This is advantageous since the application can do other operations while the communication takes place. In our applications, we have used this technique the maximum extent.

MPI also provides several other functions. One feature we will use is *Cartesian communicators*. In MPI, each node is assigned an identification number, called a *rank*. Using Cartesian communicators allows allow a program to easily translate from coordinates in an n-dimensional Cartesian coordinate system, where each node is assigned one coordinate, to its rank. In our implementations, we will by default store all voxel values of a subcube of the 3D space on each node. As we mentioned in the previous chapter, with $p = ab$ nodes and a space of size $n \times m \times k$, the subspace of each node will be of size $(n/a) \times (m/b) \times k$[8].

When using Cartesian communicators, MPI provides functions for converting between the Cartesian coordinates and the corresponding rank, as illustrated in Figure 2.9 on the following page. In the figure, nodes are split in a 2D grid. Each node stores the voxels of a 3D

[8]While we could have used a 3D subdivision, so that we have $p = abc$ nodes and each node is assigned a subcube of size $(n/a) \times (m/b) \times k$, for seismological applications, k is usually significantly smaller than n and m. Therefore, we will use a 2D grid of nodes in our implementations.

subcube as shown in the figure. The functions `MPI_Cart_rank` and `MPI_Cart_coords` convert from coordinates in the 2D grid of nodes to the corresponding rank and vice versa, respectively. Note that the MPI library may optimize the layout of the Cartesian coordinates of each rank. On supercomputers, for example, a specific rank may be able to communicate very efficiently with another rank, while communicating with a third rank may take longer time due to increased distance between the processors. The MPI library may thus optimize the Cartesian grid so that ranks which can communicate quickly are assigned Cartesian coordinates which are close.

We have not used a 3D grid of nodes, since for seismological applications, the number of points in the z direction is typically much less than the number of points in the x and y directions. Therefore, we have opted for the 2D subdivision. However, the algorithms we present in subsequent chapters are easily generalized to 3D.

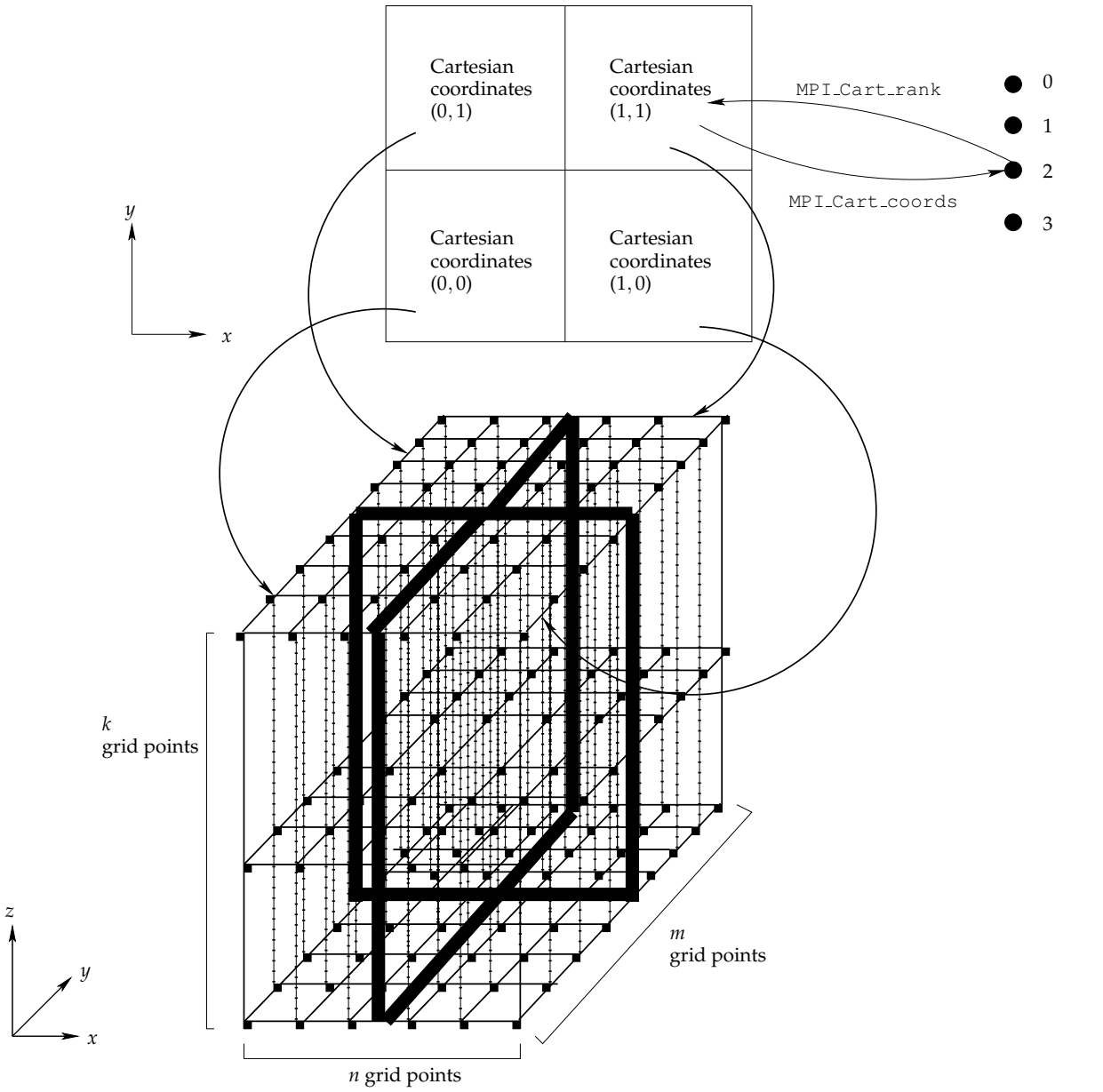


Figure 2.9: Cartesian communicators in MPI.

We refer to [24] for further information on MPI. We comment the MPI functions used in our implementation source code.

2.5.2 Fine-Grained Parallelism

We will use two techniques for fine-grained parallelism: multi-core programming and GPUs.

Multi-Core CPUs

Recently, CPUs with multiple execution units — or multiple *cores* — have become increasingly common. We will use the OpenMP application programming interface (API) [25] for utilizing multi-core CPUs (and also machines with multiple CPUs) with shared memory, referred to as *Symmetrical multiprocessing* (SMP) machines. OpenMP allows for easy utilization of SMP by introducing preprocessor directives into the source code of the program. Figure 2.10 illustrates parallelization using OpenMP.

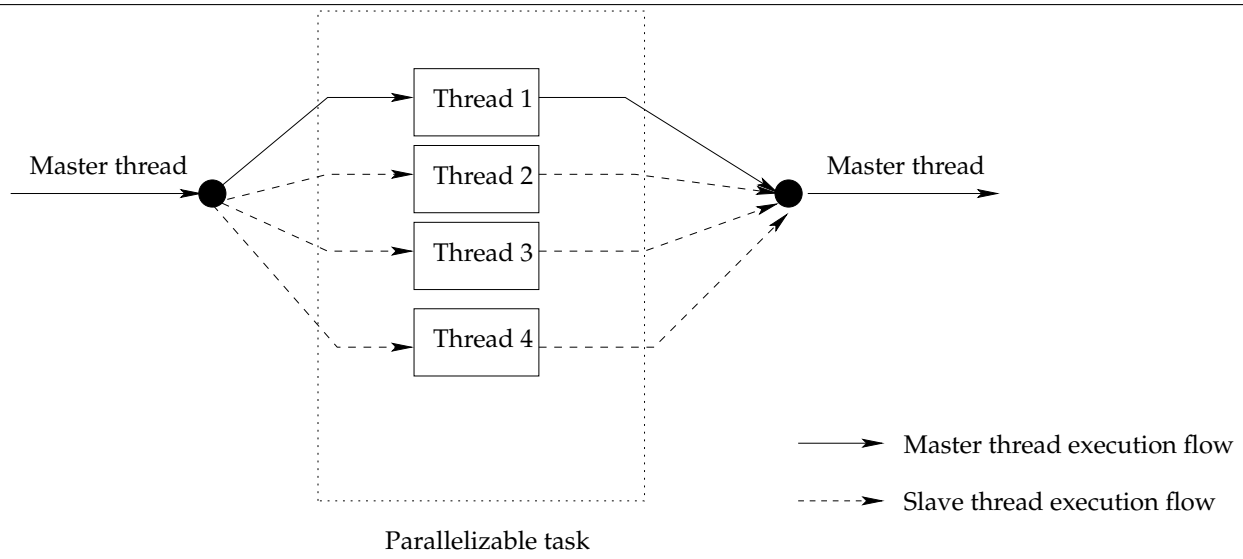


Figure 2.10: Parallelization using OpenMP. Initially, only the master thread in a program runs. Then, when a parallelized block of code is encountered, new slave threads are spawned (a *fork*). Each thread then executes in parallel, before the parallel task has been completed and the slave threads stop executing (a *join*).

To fork processes using OpenMP in ANSI C, the `#pragma omp parallel` preprocessor directive can be used before a block of code. At the end of the code block, a join operation is done. For instance, to parallelize a **for** loop iterating over N elements in C, the following code segment could be used:

```
  int i;
#pragma omp parallel for
  for(i = 0; i < N; i++) {
    // ... each thread executes the loop N/#threads times
  }
```

Listing 2.1: OpenMP-parallelized loop

This loop assigns to each OpenMP thread — where typically one thread is run on each CPU core — to approximately $N/\#threads$ values of i, which are run in parallel on all cores. OpenMP also includes directives for declaring variables which are private or shared among threads, in addition to synchronization directives. We refer to [25] for the complete OpenMP specification.

Graphics Processing Units

Another recent trend is to use Graphics Processing Units (GPUs) to offload computations from the CPUs. This development is due to GPUs becoming increasingly programmable. Consider Figure 2.11, which shows the logical architecture of a modern GPU[9].

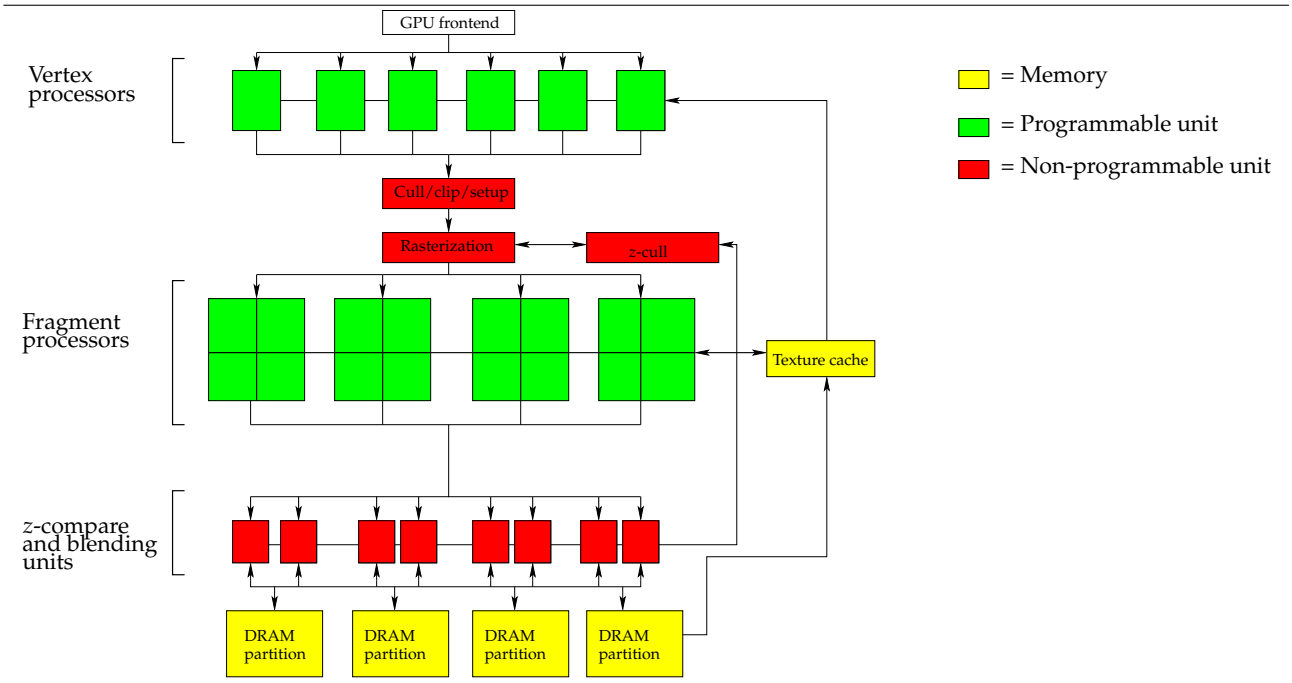


Figure 2.11: Modern GPU Architecture, adapted from *GPU Gems 2* [26]

We now briefly describe the GPU architecture; for more details, consult e.g. [27, 26]. A host machine passes a set of vertices to the GPU, which are the vertices of a set of triangles or quadrilaterals. These vertices are passed to the vertex processors, which can be programmed to do per-vertex operations and may modify certain properties of the vertex or pass additional values further down the GPU pipeline. The vertex processors have the ability to read from the GPU's texture memory, so that, for example, the texture color value associated with the vertex of a triangle can be read from the vertex program. This is useful when, for example, producing certain lighting effects which use per-vertex operations [15].

Next, a cull/clip/setup block groups vertices into *primitives*. For instance, three vertices can be grouped into a triangle. In this block, culling (removing very near or very distant

[9]New GPUs, such as the NVIDIA G80 series, do not have distinct vertex and fragment processors, and also introduce a new type of processor (geometry processors) which are of no use to us. Thus the physical implementation of the GPU today does not quite match the logical view of the architecture (Figure 2.11), but the logical view is still valid.

objects) and clipping (removing objects outside the screen space) are first performed. Subsequently, the primitives are passed to the *rasterization* block, which *rasterizes* all primitives. In rasterization, a primitive is converted into a discrete set of *fragments*[10] to be drawn to the screen.

Each rasterized fragment is then passed to one of many[11] *fragment processors* in parallel, which can execute a program modifying the color value of each pixel. The fragment program may write four components (red, green, blue and alpha) to GPU memory, and may also read from the GPU's texture memory to blend or apply lighting to the fragment's color value. Finally, the data is passed through blending and z-compare units. The blending units can blend pixels to produce various effects, such as fog. The z-compare units decide which fragments are to be displayed as pixels[12]. The pixels are subsequently written to the GPU's frame buffer memory for display on screen. By using *render to texture* techniques, we can write to the GPU's texture memory instead, and then, the pixels may be downloaded to the CPU instead of being displayed on screen.

The GPU hardware may be exploited for general computations, mainly by using the fragment processors. The fragment processors execute programs for each rasterized pixel in a massively parallel fashion. Recent GPUs have hundreds of such fragment processors. The fragment programs can thus be viewed as the inner kernel of a loop of a computation over a grid of discrete values, if each element in the grid is computed independently from the other elements. The input data is stored in texture memory in advance, from which the fragment programs can read. The output is written to texture memory, which can be downloaded to the CPU subsequently. Figure 2.12 on the following page illustrates the idea.

[10]Fragments are candidate pixels. A fragment may become a pixel, but can also be discarded in the z-compare block after the fragment processors

[11]Recent GPUs may have up to 128 fragment processors.

[12]The blending and z-compare units are seldomly used in non-graphics GPU applications.

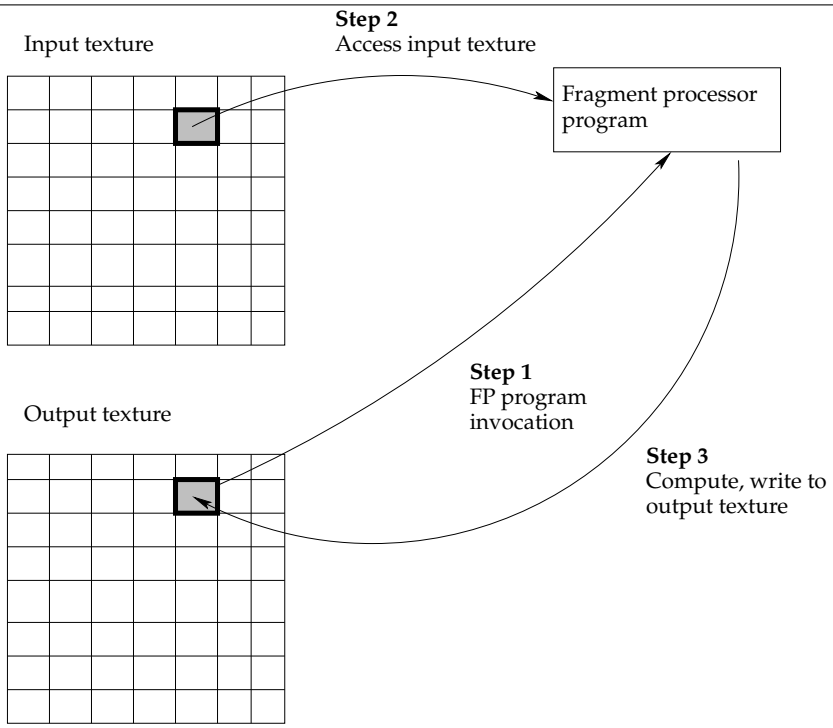


Figure 2.12: Using GPU fragment processors for computations.

This technique has been used for GPU computations of protein structure prediction [28], linear algebra [29, 30], financial algorithms [31], flow simulation [32], fast Fourier transforms (FFTs) [33], data compression [27] and even sorting [34]. Speedups over corresponding CPU computations can range anywhere from 2 to over 50. In general, it is advantageous that the computations done on the GPU are of high arithmetic complexity, since the significance of the data upload and download time from and to RAM diminishes as the arithmetic complexity increases [27].

In the very recent NVIDIA G80 GPUs, a framework called CUDA (Compute Unified Device Architecture) [35] has been introduced. CUDA is tailored for general-purpose programming on the GPU. When a G80 GPU are in CUDA mode, the concepts of triangles and vertices vertices are not used. Instead, the GPU acts as a set of parallel processors with a small amount of shared memory. While CUDA does make general-purpose programming easier and gives access to shared memory between the processors, for our use, it actually *complicates* the programming: In our case (as we shall see in Section 3.6), we are dealing with polygons and triangles. When using CUDA, the GPU has no concept of triangles and polygons at all. Therefore, in our case, it is easier to program GPUs using the regular pipeline instead of using the CUDA mode. However, note that for other applications not tied to triangles and polygons, such as linear algebra and FFT, using CUDA is advantageous since unnecessary stages of the pipeline are skipped.

2.6 Load-Balancing Techniques

We now give a brief description of load balancing techniques. As mentioned in the previous chapter, load balancing is a component of our framework.

Meyer [36] gives an overview of load-balancing approaches on clusters. In general, there are three types of approaches [36]:

1. *Static load-balancing*, where problems which can be solved in parallel are explicitly programmed to be independent work units in the program code.
2. *Semi-static load-balancing*, which do load-balancing at the start of the program execution or at specific time intervals.
3. *Dynamic load-balancing*, where load imbalances are continuously detected at runtime. There are two primary categories of dynamic load-balancing [37]:
   (a) *Peer-based methods*, where work is initially distributed among processes and data migration from different processes is subsequently done on a by-need basis.
   (b) *Pool-based methods*, where one process, which may be called a load distributor, holds all the work and other processes request work from the load distributor.

Static load-balancing induces little overhead at runtime, but may produce a work distribution which is suboptimal — certain nodes may be idle, while other nodes are overloaded. Dynamic load-balancing, on the other hand, can induce some overhead at runtime but in return gives better work distribution. The semi-static approach is a compromise between the static and dynamic strategies.

As we will see in Chapter 3, our load-balancing strategies are dynamic and peer-based. In general, fully dynamic load-balancing consists of five steps [36] (citing [38]):

1. *Load evaluation*. First, a dynamic load-balancing algorithm should determine if there is any imbalance at all. If there is no imbalance, the algorithm should terminate quickly. If an imbalance is detected, the source of the imbalance (i.e. which nodes are underutilized or overloaded) should also be detected.
2. *Profitability determination*. Although an imbalance is detected, it is not certain that load-balancing will increase performance. Migrating data and tasks from overloaded processes to underutilized processes costs time. If the migration time is greater than the savings gained by load-balancing, clearly, load-balancing should not be done.
3. *Work transfer calculation*. Next, a set of tasks which *may* be migrated from overloaded nodes to underutilized nodes is computed. This computation is done on the basis of the preceding two steps — the first step gives the source of the imbalance, and the second step determines whether work should be transferred or not.
4. *Task selection*. In this step, the destination to which the set of tasks computed in the previous step is computed. In computing the destination node, parameters such as task size, communication bandwidth and locality of the source and destination node may be considered.
5. *Task migration*. In this step, the tasks of the previous step are actually migrated.

Load-balancing schemes may only be partially dynamic, in which case only some of the steps above are implemented. Furthermore, load-balancing can also be considered in the context of fault-tolerance [39].

There are no silver-bullet solutions to the load-balancing problem, since general approaches, which attempt to work independently of the actual problem being solved, fail to utilize problem-specific knowledge [36]. General solutions suggested either introduce system-level software, such as network schedulers [40], or only apply to a specific class of problems [41].

We have therefore created our own algorithms to do load-balancing, which will be described in Chapter 3.

2.7 Framework Interface

We now describe the interface our framework supports. This can be viewed as a high-level requirements specification for the framework. Our framework aims to support *efficient computation of functions over a set of voxelized triangles*, in such a way that each polygon is handled by a single node. To achieve this, we require efficient voxelization techniques, efficient caching and transfer techniques, and efficient load-balancing techniques.

We describe notation used in subsequent chapters in Section 2.7.2. Next, the actual interface is described in Section 2.7.2.

2.7.1 Notation

We are given a discrete 3D coordinate space $N \times M \times K$, and $p = ab$ cluster nodes, where a and b are integers. Let $n = |N|$[13], $m = |M|$ and $k = |K|$. Then a, b, n and m are assigned such that $n \bmod a = 0$ and $m \bmod b = 0$[14].

A function $V(p)$ is defined from $p = (x, y, z) \in N \times M \times K$ on the set of values G. In principle, G can be any value, but we will only use single, real numbers, so in our case, $G = \mathfrak{R}$. Thus, $V(p)$ associates with each point an application-dependent value. In seismological applications, for example, $V(p)$ is the seismic signal value at point p.

Node i is assigned a set of $(n/a) \times (m/b) \times k$ grid point coordinates C_i, corresponding to a subcube of the 3D space. We assume that initially, all values in G of the points $p \in C_i$ are stored at node i. Thus, the values associated with each grid point coordinate are stored as in Figure 1.1 on page 2.

If we use polygon meshes to represent n_m meshes, each mesh M_i, $i = 0, \ldots, n_m - 1$ can be viewed as a triple $M_i = (V_i, E_i, F_i)$ with sets representing vertices V_i, edges E_i and faces F_i. These sets contain the winged-edge structure. Each polygon of M_i corresponds to one element in F_i.

We require a function $V^*(F)$ from a face F in some F_i to the power set of the set $\{p | p \in N \times M \times K\}$, which associates with each face the coordinates which intersect the face surface. Hence, the function V^* extracts the coordinates of which intersect a face. This is essentially a discretization of the face, and $V^*(F)$ is called the *voxelization* of F. Note that each point is taken to represent a small cube; hence, when we say that a point intersects a face surface, we refer to the cube representing the point intersecting the surface. We will also, for convenience, use the notation

$$V^*(\mathfrak{P}) = \bigcup_{P \in \mathfrak{P}} V^*(P) \tag{2.1}$$

[13]Throughout this thesis, we will mainly use N to denote the set of processors and $|N|$ to denote the number of processors. It will always be clear from the context whether we mean N to be the set of the coordinates in the x-direction or the set of processing nodes.

[14]This may require padding of n and m, so that we use $n + (a - n \bmod a)$ instead of n and similarly for m.

where P is a set of polygon faces.

Figure 2.13 illustrates the above concepts. In the figure, the coordinates of the grid points surrounded by solid lines in the right part are returned by the V^* function when applied to the polygon face, which is a part of some mesh M_i, which contains the winged-edge structure. V maps the coordinates to user-defined values in G. The figure illustrates a polygon in a plane parallel to the xy plane, but in principle, polygons can have any orientation.

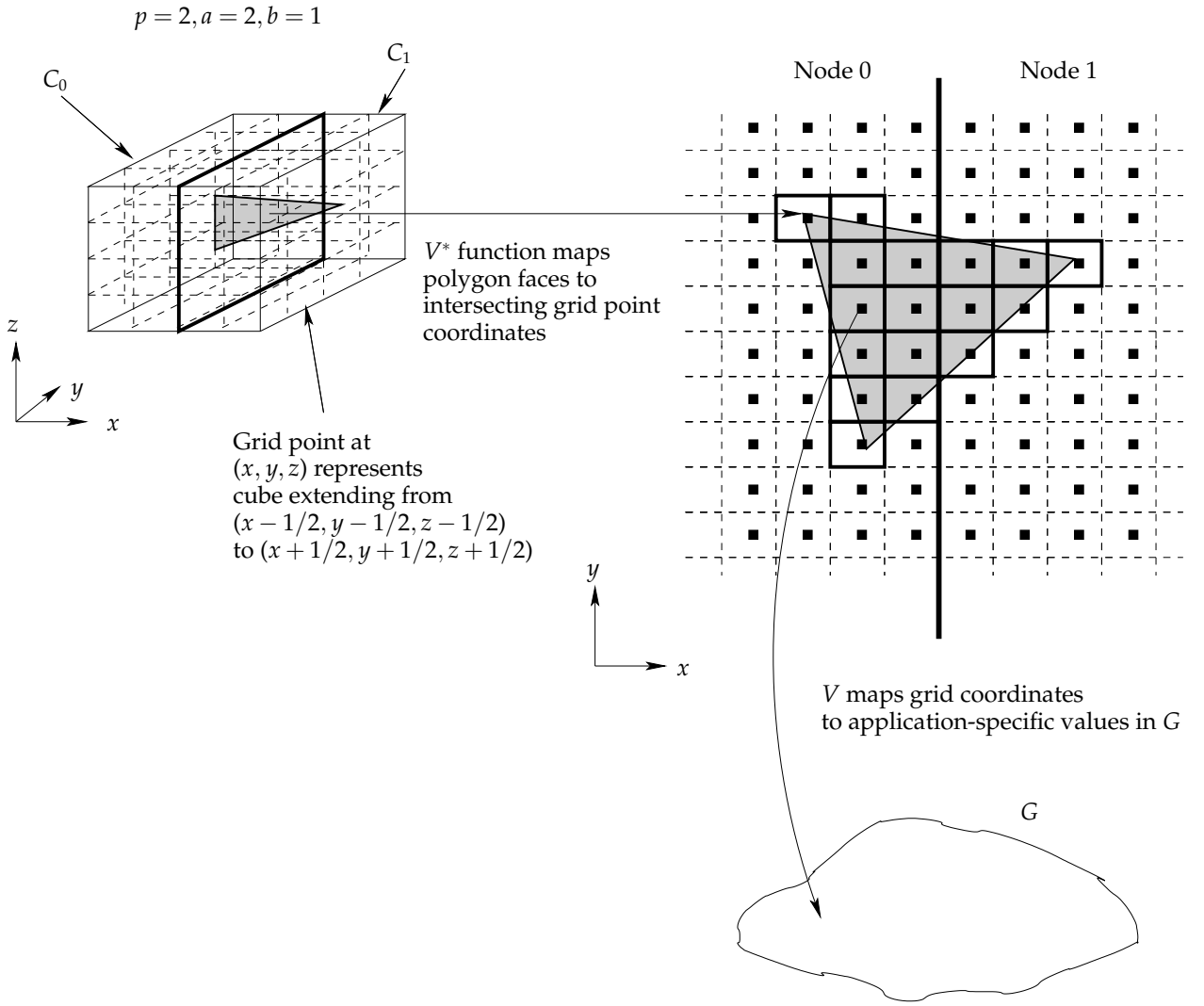


Figure 2.13: Notation used in the rest of this thesis.

To solve the problem requires consideration of both:

1. How to transfer voxels between nodes. If the $V(p)$ values for $p \in C_i$ are stored on node i, then, if node i requires the result of $V(p)$ for some $p \in V^*(F)$ for some face F partially on node i and partially on other nodes, communication among the nodes is required, since the $V(p)$ values for points not in C_i is located on other nodes.
2. How to ensure that load is balanced. That is, if all faces F^i whose voxels are stored on node i is such that the number of voxels in the voxelization of all faces in F^i, $|V^*\left(F^i\right)|$, is very large compared to $|V^*\left(F^j\right)|$ for some other node $j \neq i$, it is perhaps wise to distribute some of the faces in F^i to node j to ensure equal load.

We discuss these problems and solutions in Chapter 3.

Now, we describe precisely the set of operations our framework supports.

2.7.2 Framework Operations

Formally, we may separate between the two classes of topological and geometric operations — or *operators* — which take geometric entities such as vertices, edges, faces or entire objects as operands [14, 42]:

1. *Static operators*, which do not change topological relations or geometric attributes. Examples of static operators are comparing two objects for equality, set-based relationships, such as computing the union, intersection and difference between the points of two objects, computing the boundary of an area, checking whether a point is inside an area, and computing distances and angles between points. The V^* function in the previous section can also be viewed as a static operator.
2. *Dynamic operators*, which may change topological relations or geometric attributes. Dynamic operators can be further subdivided into the following classes:
   (a) *Creation operators*. These operators create new geometric entities, which corresponds to creating faces, edges or vertices in polygonal representations. The new entities may be related to existing entities, in which case a *dependent creation* occurs, or unrelated to other existing entities, which is an *independent creation*. For example, constructing a new polygon with no connections to a polygon mesh is an independent creation, while adding a new vertex or face to an existing mesh or polygon is a dependent creation.
   (b) *Update operators*. These operators change an already-existing object. For example, an operator to move the vertex of a polygon to another location is an update operator.
   (c) *Destruction operators*, used for destroying geometric entities such as vertices, edges, faces or entire objects.

We can also classify operators as [14][15]:

1. *Geometric operators*, which relate to geometric attributes, such as the position of a vertex and the length of an edge. Transformations such as translation, rotation and scaling are geometric operators.
2. *Topological operators*, which relate to topological relationships and not geometric attributes. Creating a new vertex and linking vertices with edges are example topological operators.

Below, we describe which operations our framework implements. We will use a small number of operators. While the set of operations we describe is sufficient for supporting the needs of the application in Section 2.1 on page 7, other applications may require other operators. However, had we implemented all operators all conceivable applications might need, the interface of the framework would be large and complicated. Furthermore, from the implementations of the operators we describe, implementing other, similar operators should be simple.

[15] More general operators which do not assume Euclidean or topological spaces also exist [14], however, we only consider discrete Euclidean spaces, which are also topological spaces, in this thesis.

2.7.3 Static Operators

We will consider two static operations, which can be considered to be topological operators. These operations are computing the V^* function and ensuring that all $V(p)$ values for $p \in V^*(F)$ for all faces are copied to a single node.

- VOXELIZE(F), where F is a polygon face object. This operation returns the coordinates of each grid point intersecting F (see Figure 2.13). This is equivalent to computing $V^*(F)$. As mentioned, this process is commonly called *voxelization*.
- EXTRACT-ALL computes VOXELIZE(F) for all faces in all meshes, and makes sure that for each face F, all voxel values $V(p)$ for $p \in V^*(F)$ are available on a single node. That is, the procedure makes sure that all the values associated with each coordinate of each face are available on a single node. Thus, each face F has a single node, called the *responsible node*, on which all $V(p)$ values for $p \in V^*(F)$ are available subsequent to the EXTRACT-ALL operation. If doing so, computations over each face's surface (done after this operation) can be done on the face's responsible node. If responsible nodes are assigned wisely, load will be balanced and the computations can proceed in parallel. A major part of the EXTRACT-ALL operation is to ensure that load is balanced.

Our goal is thus to compute EXTRACT-ALL as efficiently as possible, such that all voxels of a particular face are stored on a single node, and such that each face has a node which is responsible for doing computations over the voxelized representation of the face. Precisely which computation to do is application-dependent. In our benchmarks, we use computations commonly done in seismological applications.

2.7.4 Dynamic Operators

Mesh Creation and Destruction Operators

We will use the following operators for creating and destroying meshes.

- CREATE-MESH(V_0, V_1, V_2), which creates and returns a new mesh containing a single triangle defined by the three vertices V_0, V_1 and V_2. The mesh returned is $M = (V, E, F)$, containing the winged-edge representation of the single polygon created, so that the set V contains vertex coordinates and an adjacent polygon edge, E contains the winged-edge edge list structure, and F contains the faces. The edges of the face will be (V_0, V_1), (V_1, V_2), and (V_2, V_0). This can be viewed as an independent creation operation.
- DESTROY-MESH(M), which removes the entire mesh M. The same effect can be obtained by removing all the faces of the mesh. This can also be done by a sequence of REMOVE-FACES operations (see below).

Polygon Creation and Destruction Operators

In order to create and destroy polygon faces, we require the following two functions:

- ADD-FACE(M, V_0, V_1, V'), which adds a new face to the mesh M. This operation can be viewed as a dependent creation operation. The new face will be connected to already-existing vertices of other polygons in the mesh, which is specified by two vertices V_0

and V_1. Since we are assuming a manifold mesh, it is assumed that the edge (V_0, V_1) exists and is a part of exactly one polygon prior to applying this operation. Then, a new face can be created by specifying a single new vertex, V'.

- REMOVE-FACES(M, V), removing *all* faces and edges incident to V, in addition to V itself.

Figure 2.14 illustrates ADD-FACE and REMOVE-FACE. As described above, adding a face is done by specifying two vertices already in the mesh, along with a new vertex. Removing a face is done by specifying a vertex in a mesh and removing all incident faces and edges. Before adding or removing faces, CREATE-MESH must be executed.

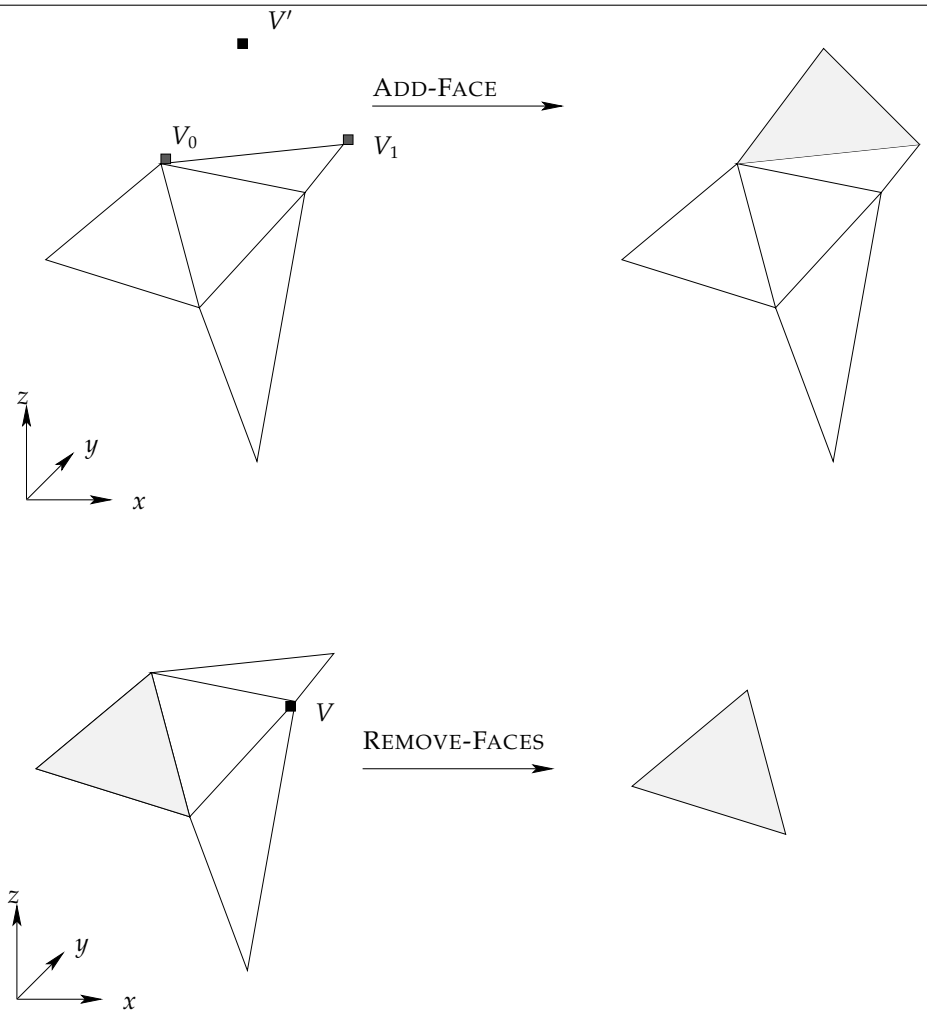


Figure 2.14: Adding and removing faces.

Update Operators

We will only require one update operation:

- MOVE-VERTEX(M, V, C), which moves vertex V of mesh M to a new location C. Note that this operation might lead to faces incident with V intersecting entirely different grid points; that is, the $V^*(F)$ function may change for all faces F of M, incident to V.

Operation	Brief Description
VOXELIZE	Compute $V^*(F)$ for a face
EXTRACT-ALL	Compute VOXELIZE for all faces and ensure that all $V(p)$ values for $p \in V^*(F)$ for each face are available on at least one single node, assign each face a node responsible for doing computations over the voxelized representation of face
CREATE-MESH	Create a new mesh consisting of one polygon
DESTROY-MESH	Remove a mesh
ADD-FACE	Add a new face to a mesh
REMOVE-FACES	Remove faces incident to a specified vertex
MOVE-VERTEX	Move a vertex of a face to a new position

Table 2.1: Summary of framework operations.

Notice that with this single update operator, many geometric transformations are supported. One commonly used class of transformations is *affine transformations* [15], which include translation (moving a point), rotation (rotating a point about an axis) and scaling (increasing or decreasing the Euclidean distance between points). We will primarily be concerned with the translation transformation applied to the vertices of a mesh, since this is what would be used in a pillar gridding application — but in principle, all kinds of transformations applied to single vertices, edges or entire faces (in which case the MOVE-VERTEX operation is applied to each vertex of the edge or face) are supported through this single operation.

Figure 2.15 illustrates the MOVE-VERTEX operation.

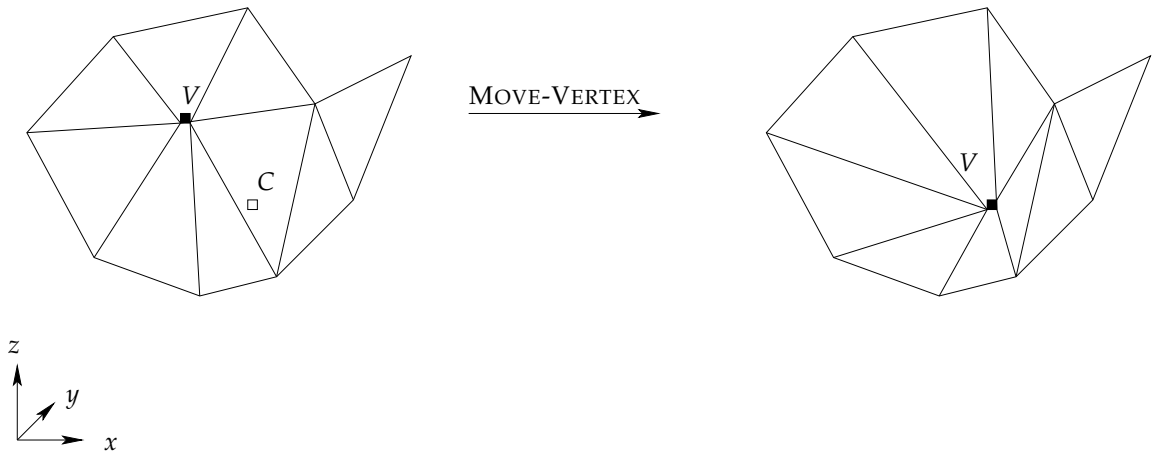


Figure 2.15: Moving a vertex in a polygon mesh using the MOVE-VERTEX procedure. The vertex marked with a black square, V, is moved to new coordinates C.

2.7.5 Summary of Operators

Table 2.1 summarizes the operations our framework supports.

Chapter 3

Framework Design Part I: Cache and Transfer Strategies

In this chapter, we consider how to voxelize many polygons in parallel, using multiple computer nodes, such that computations over the voxelized representation of the polygons in a set of meshes are executed efficiently.

Voxel data is distributed on several nodes. Hence, if a polygon intersects the boundary between two nodes, which node should be responsible for extracting the voxels of that polygon and doing computations over the voxelized surface? First, we give an introduction to how the polygonal structures have been implemented in our framework Section 3.1. Next, Section 3.2 introduces terminology used in this and the next chapter, and Section 3.3 describes the primary workloads we will consider. Then, in Section 3.4, we present three strategies for how voxels should be transferred among the nodes when executing the EXTRACT-ALL operation and subsequent computations and discuss advantages and disadvantages of using each strategy. Subsequently, the strategies are compared in Section 3.5.

The first sections of this chapter assume the presence of a method for discretizing triangles, i.e. doing the VOXELIZE operation described in the previous chapter. This is necessary to do in order to find the grid points intersecting a triangle surface. Figure 3.1 illustrates voxelization. In Section 3.6, we consider approaches for doing the voxelization, and introduce a new triangle voxelization algorithm suitable for GPUs and multi-core CPUs.

3.1 Framework Interface Implementation

The values we operate on — the $V(p)$ values — are distributed among cluster nodes. This is necessary because the data set size exceeds the memory size of single nodes, and in order to facilitate coarse-grained parallelism. But how should the *polygonal data*, that is, the winged-edge model, be distributed? Figure 3.2[1] shows two approaches: a distributed/local polygon model, or a global polygon model.

Either, each node can store the entire polygonal model (a *global* model — left side of Figure 3.2), or, the model can be distributed so that each node stores only a part of the model (a *local* or *distributed* model — right side of Figure 3.2).

[1]Throughout this and the next chapter, we will show figures in 2D. This is done just to make the figures viewable. Polygons can in principle have any 3D orientation.

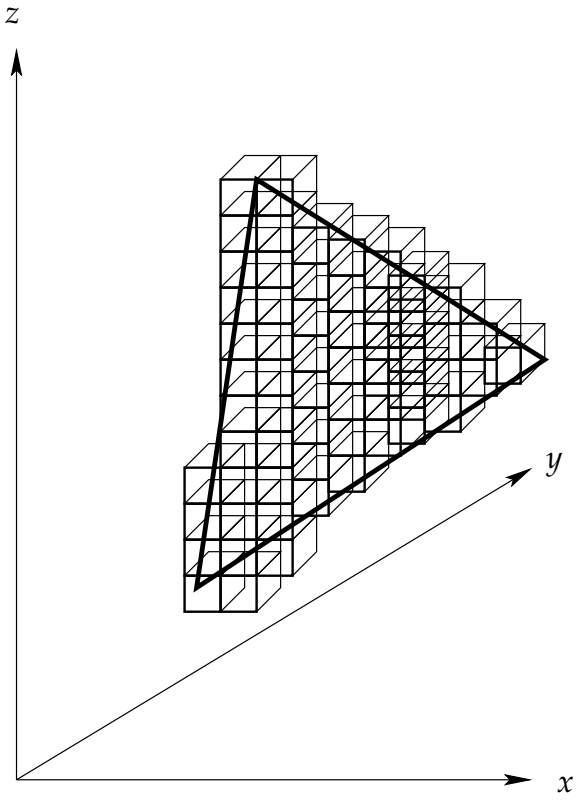


Figure 3.1: Discretizing, or *voxelizing* a 3D triangle

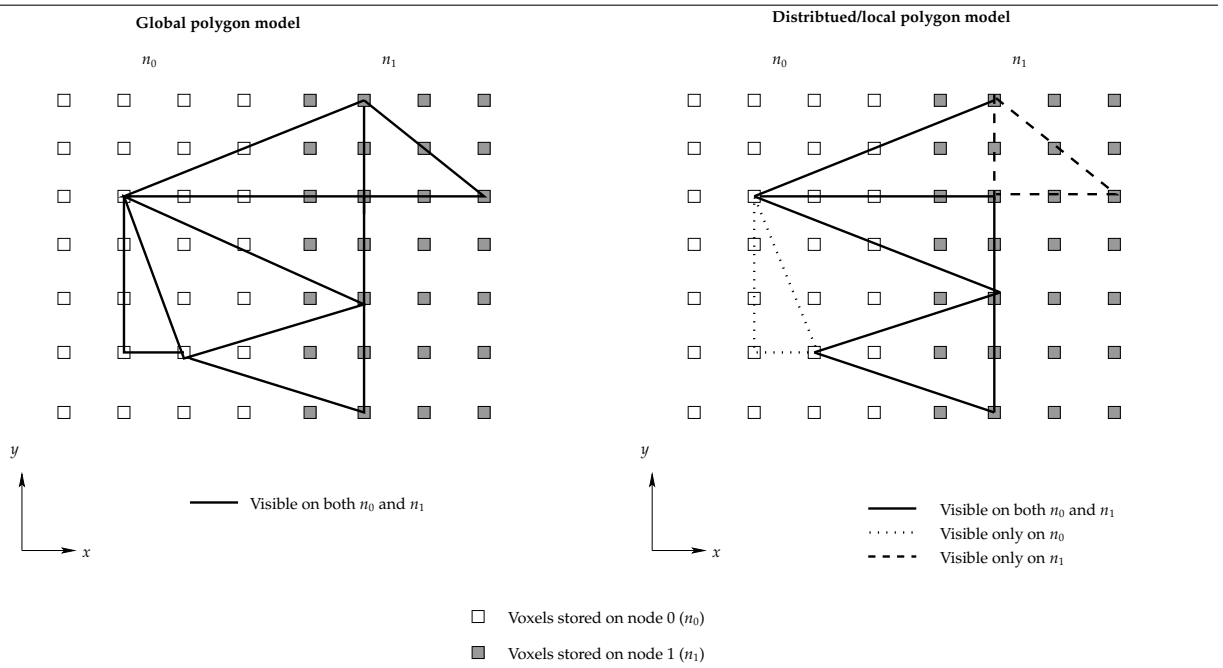


Figure 3.2: (*Left*) Global polygon model. Each node stores the entire model. (*Right*) Distributed/local polygon model. Each node only stores parts of the polygon model.

In our framework, we have chosen to *keep the polygonal model global*. That is, we assume that *each* cluster node has the *full* winged-edge model of all meshes, faces, edges and vertices

stored locally in its memory. In contrast, a distributed or local polygon model would only store the polygons that lie inside a node's coordinate space. This implies that we assume that all operations we described in the previous chapter (Table 2.1 on page 31) are broadcasted *to all nodes* when they are executed.

There are several reasons for keeping the polygon model global:

1. Each cluster node has a full view of all topological relationships between all polygons. Thus, applications running on one node knows about all other polygons stored on other nodes. If the polygon model had been distributed, the application would only have a limited view of the other polygons.
2. With global polygon models, load-balancing can happen without communication between the nodes, as we will discuss in Section 4. This is impossible when using only a local model, since one node can not possibly know how much other nodes are loaded.
3. Interactive operations become quicker, since all operations are broadcasted to all nodes. This way, when a user for example executes a MOVE-VERTEX operation, the nodes do not need to communicate at all. With a distributed model, if a vertex of a polygon was in node i's coordinate space C_i prior to moving but was moved to node j's coordinate space, C_j, communication between i and j would be required since the information about the polygons incident to the moved vertex might be unknown to node j.
4. There is little gain with distributed polygon models. The only reason for using a distributed model would be to save memory. However, the polygon models are small compared to the memory used to store the voxels. Therefore, there is little to gain (and much complexity to add) by having local polygon models.

This choice implies that all operations except the EXTRACT-ALL operation are done with no communication between the nodes. All communication is deferred to executing the EXTRACT-ALL operation. Recall that this operation computes $V^*(F)$ and moves all coordinates $p \in V^*(F)$ to a single node, for each face F in all meshes. After this operation has completed, typically, some computation will be done over the values returned. However, the methods we describe in the following section also apply to cases where we want to execute VOXELIZE for a single face.

3.2 Centroid, Centroid Nodes and Responsible Nodes

One central concept we use throughout this section is that of the *centroid* of a triangle, defined in Definition 1 (based on the definition in [15]) below.

Definition 1 *Given a triangle with vertices $v_i = (v_{i,x}, v_{i,y}, v_{i,z})$ for $i = 1, 2, 3$, the **centroid** of the triangle is the position of the center of mass, assuming the triangle has constant density. For a triangle in three dimensions, the centroid coordinates $\bar{c} = (\bar{c}_x, \bar{c}_y, \bar{c}_z)$ are given by*

$$\bar{c} = \frac{1}{3}\left(\sum_{i=1}^{3} v_{i,x}, \sum_{i=1}^{3} v_{i,y}, \sum_{i=1}^{3} v_{i,z}\right) \tag{3.1}$$

In our discrete coordinate space, we define the centroid to be $\bar{c}$ with each component rounded to the nearest integer.

■

We also need to define the concept of a triangle's *centroid node*, in Definition 2 below.

Definition 2 *Given a triangle T with vertices $v_i = (v_{i,x}, v_{i,y}, v_{i,z})$ for $i = 1, 2, 3$ with centroid $\bar{c}_T$, the **centroid node** of T is the node j whose coordinate space C_j is such that $\bar{c}_T \in C_j$.*

■

The concepts of centroid and centroid nodes are illustrated in Figure 3.3. In the figure, since $\bar{c} \in C_1$, node 1 is the centroid node of the triangle.

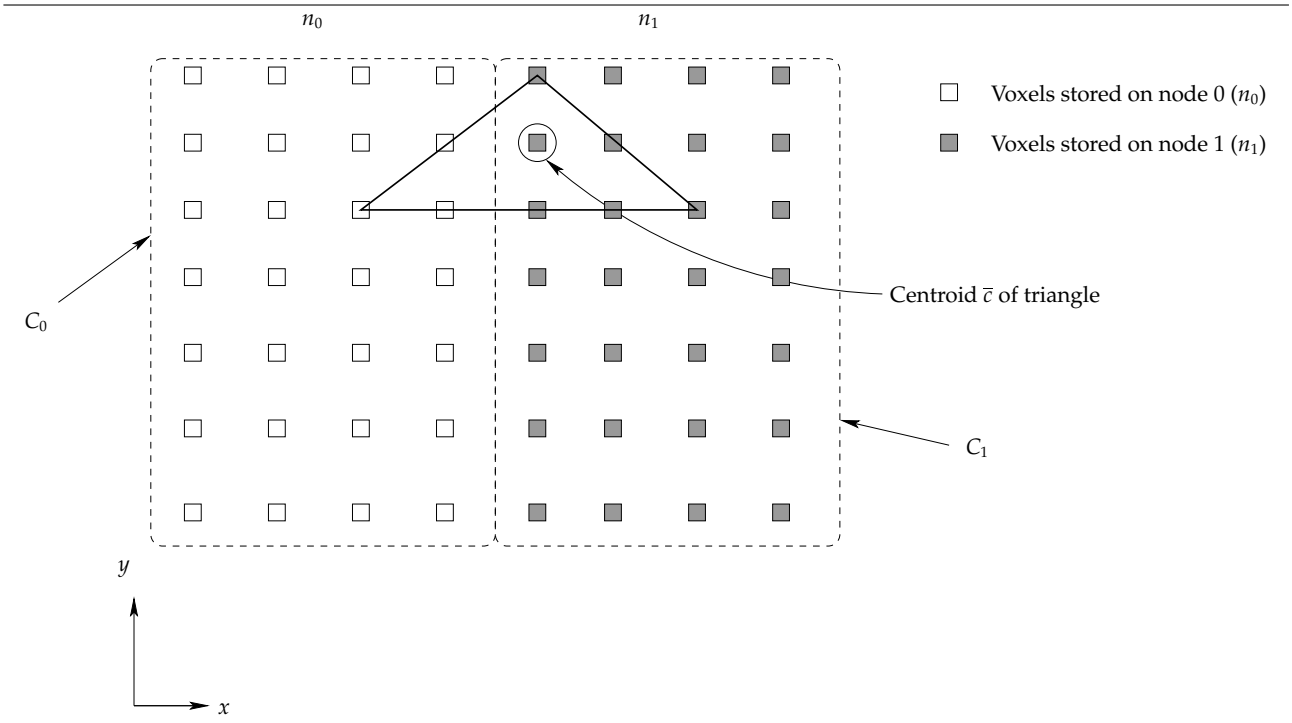


Figure 3.3: Centroid and centroid node

Now, consider Figure 3.3 again. Since we require having all voxels available on one node, we have two choices when executing the VOXELIZE operation on the triangle:

1. Either, node n_0 can do the VOXELIZE operation and transfer the voxels not stored on node n_0 from node n_1. Thus, node n_0 does all the computational work.
2. Or, node n_1 can do the computational work.

We will also use the concept of *responsible nodes*, defined in Definition 3 below.

Definition 3 *A node n_i is **responsible** for a triangle T if, when executing the* EXTRACT-ALL *operation, node n_i will compute* VOXELIZE(T)*, and subsequently to running* EXTRACT-ALL*, all voxels values $V(p)$ for $p \in V^*(T)$ will be available on node n_i. Any computation done subsequent to* EXTRACT-ALL *over the voxelized surface of T will be done only on node n_i.*

■

Each triangle has exactly one responsible node — if not, redundant work might be done during computation.

In the case of Figure 3.3, we can thus choose to have either node n_0 or n_1 responsible for the triangle shown in the figure. Furthermore, computations done over the triangle's surface after executing EXTRACT-ALL will be done on the triangle's responsible node.

In the rest of this section, we adopt the convention that for each triangle T, T's responsible node equals its centroid node T. This is intuitive, since if the centroid is on a node i, that node already has the most[2] voxels intersected by T available locally, by definition of the centroid. Consequently, by adopting this convention, *the number of voxels that must be transferred is always kept at a minimum.* In the next chapter, we will discuss load-balancing — where we might set the responsible node to not be the centroid node if that leads to better work distribution between the nodes.

3.3 Workloads

In our discussions, we will primarily consider the following workload, which resembles the workload in Pillar Gridding applications, as described in the previous chapter.

1. First, a polygon mesh is created using a series of ADD-FACE operations.
2. Next, an EXTRACT-ALL operation is done, and a computation over the voxelized polygon surfaces is done on each node.
3. After this, all or some vertices of the mesh are moved using the MOVE-VERTEX operation to correct or interpolate the mesh to fit the actual fault.

After Step 3, Step 2 may be repeated. Next, Step 3 is executed, and so on, until the polygons sufficiently match the fault. We refer to one execution of Steps 3 followed by Step 2 as a *move-extract cycle*, and we will primarily discuss workloads consisting of a sequence of such move-extract cycles.

3.4 Three Cache and Transfer Strategies

We now discuss three strategies for transferring and caching voxels from other nodes.

To understand why we need different strategies, consider Figure 3.3 again. Node n_1 needs to transfer voxels from node n_0. Should we *just* transfer the voxels n_0 requires, or should we transfer even more voxels? Why should we transfer more voxels than we need? A commonly used model for measuring the communication time $T_{\mathsf{comm}}(n)$ for sending n bytes over a network is [23]

$$T_{\mathsf{comm}}(n) = \alpha + (1/\beta)n \tag{3.2}$$

where α is the latency and β is the bandwidth of the link. For commonly used Gigabit Ethernet interconnects (and for many other types of links), α dominates the $(1/\beta)n$ term completely even for medium-sized n. Therefore, it is in many cases better to transfer n voxels once rather than transferring $n/2$ voxels twice.

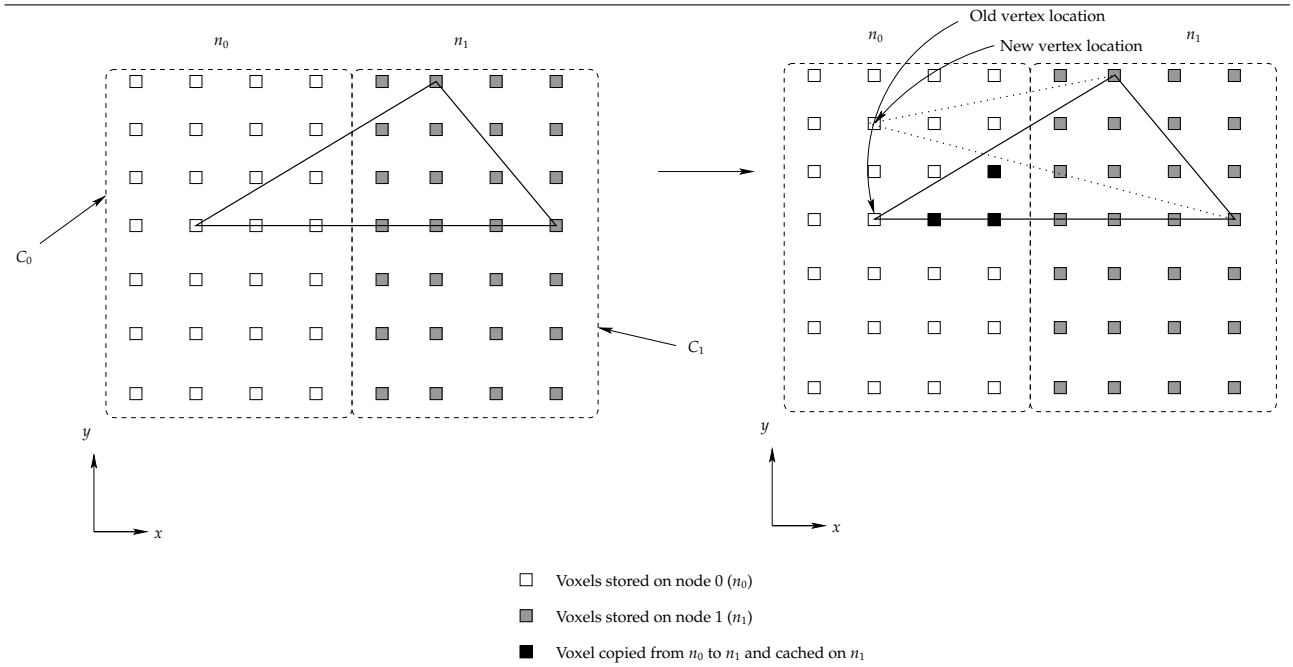


Figure 3.4: Moving a vertex, illustrating the need for caching. (*Left*) Original triangle. (*Right*) Triangle after the MOVE-VERTEX operation is dashed. If VOXELIZE is now executed, another transfer between n_0 and n_1 must occur.

But do we need more voxels from n_0 on n_1? Perhaps. Consider Figure 3.4.

Suppose we do an EXTRACT-ALL operation on the triangle mesh in Figure 3.4 (in this case, the triangle mesh is a single triangle). Then, n_1 transfers voxels from n_0 since n_1 is responsible for the triangle. After this, the configuration is as in the right part of Figure 3.4. But suppose the user of the framework now executes a MOVE-VERTEX operation to a new location. The new triangle is indicated by the dotted line. Next, EXTRACT-ALL is again executed. Now, more voxels must be transferred from n_0 to n_1! If we cached some voxels in the neighborhood of the original triangle, this might not be necessary.

Now, we describe three specific strategies for caching and transferring voxels in Sections 3.4.1, 3.4.2 and 3.4.3. All the strategies below have been implemented in the attached source code. To pass messages between nodes, we use the Message Passing Interface (MPI), which we gave an introduction to in Section 2.5.1 on page 19. We use nonblocking transfers exclusively so that multiple sends and receives are always "in flight". In one strategy, we also exploit nonblocking transfers to do voxelization while data is being transferred.

3.4.1 Strategy 1: Minimum Communication Strategy

Description

The first strategy is to use the least communication. For each polygon, each responsible node requests from other nodes the voxels of the polygon which are missing on the responsible

[2]One could construct special cases where the centroid node has the same amount of voxels as other nodes. However, in that case, choosing the centroid node as the responsible node is also an optimal choice in terms of the number of voxels needing to be transferred.

node. In this case, when EXTRACT-ALL is executed, we do the following steps:

1. Let F^i be the set of faces node i is responsible for. On each node i, compute VOXELIZE of all faces in F^i.
2. For each node i, let $C_{i,j}$ be the union of the sets of coordinates returned by VOXELIZE applied to each face in F^i, which is in some coordinate space C_j with $j \neq i$ *and* is not already cached on node i. That is, $C_{i,j}$ is all the voxels node i needs to retrieve from node j in order to store all the voxels of all the faces node i is responsible for. For all $j \neq i$, node i transmits a request to node j to retrieve the voxels in $C_{i,j}$.
3. Node j and other nodes then receive the requests from i, and transfers the requested voxel values to node i. Meanwhile, node i receives similar requests from other nodes.
4. Node i and all other nodes retrieve the voxel values, which are also cached for future use. If we need to do some computation over the voxels of all the faces, this operation can now proceed in parallel on all nodes.

This is the strategy shown in Figure 3.4 on the preceding page, where only the voxels actually required are transferred. In the figure, the black voxels are in $C_{1,0}$ the first time the Minimum Communication strategy is executed.

A diagram showing the communication between the nodes is shown in Figure 3.5 on the following page.

Analysis

Let us compute an expression for the time-complexity of this algorithm. If node $i \in N$ is responsible for a set of faces F^i with a total area of $A(F^i)$, where N is the set of nodes, an upper bound on the time Step 1 takes is

$$T_{\mathsf{comp,Step1}} = O\left(|F| + \max_{i \in N} A(F^i)\right) \tag{3.3}$$

$$= k_1|F| + k_1' \max_{i \in N} A(F^i) + k_1'' \tag{3.4}$$

for constants k_1, k_1' and k_1', since voxelization and computing the $C_{i,j}$s take time proportional to the area of a triangle and is dominated by the node having the greatest total area (since the voxelization is done in parallel on all nodes). Also, in order to find F^i, a scan of the face table, taking $O(|F|)$ time where F is the set of all faces, is required — although the other term will usually dominate the $|F|$ term.

Note that even if we use multiple cores to do the voxelization (described in Section 3.6), the voxelization might still have this time complexity, as the problem may be bounded by the speed of the memory bus rather than the computational power of the CPU. On GPUs, we do not know how the rasterization unit works internally other than likely being very parallel — but as we shall see, the CPU will have to iterate over $O(A(F^i))$ coordinates in this case as well.

Now, $|C_{i,j}|$ is the number of voxels node i requires from node j. Notice that $|C_{i,i}| = 0$ and that $|C_{i,j}|$ need not be equal to $|C_{j,i}|$. In the communication between Steps 2 and 3, node i

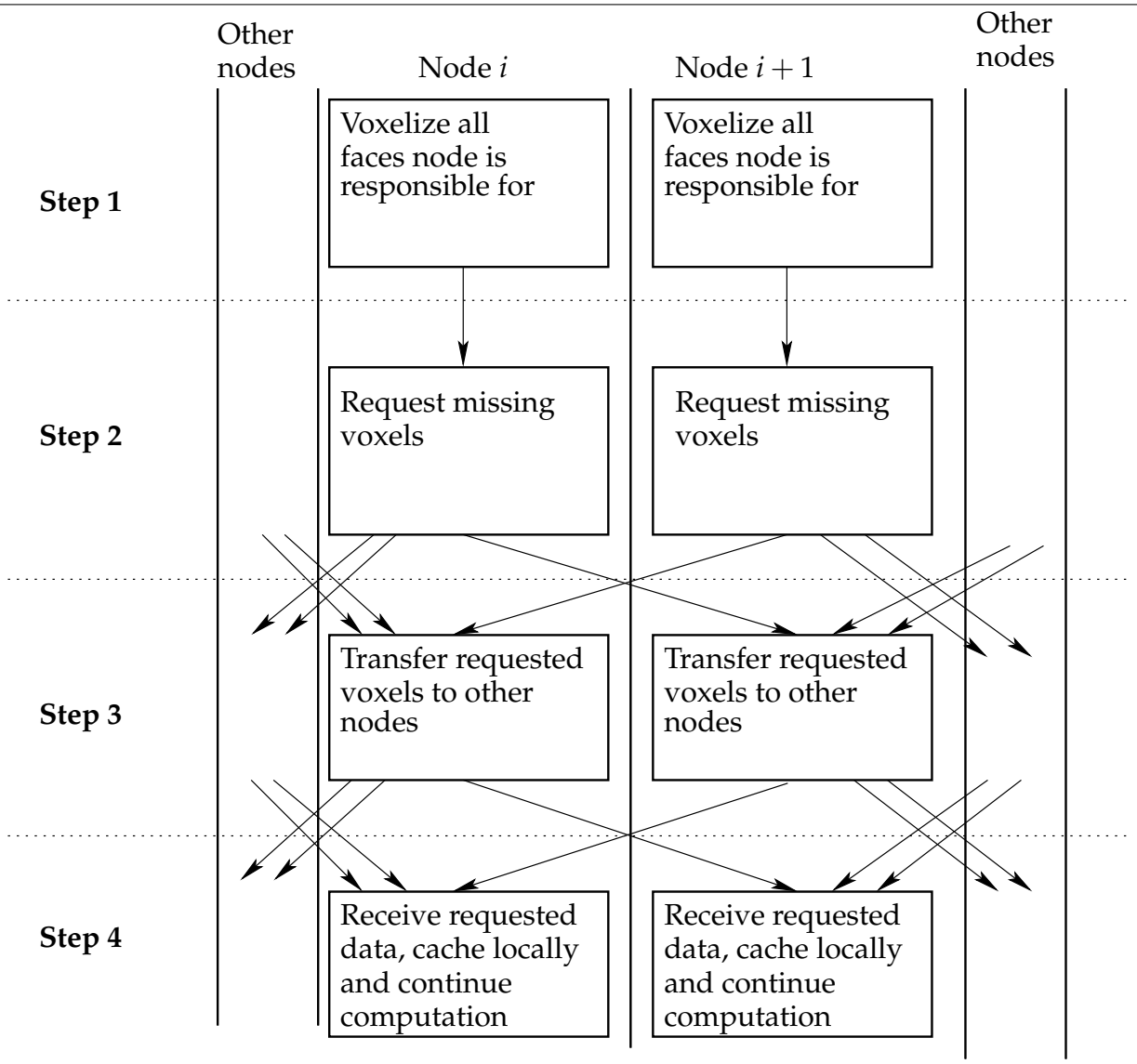


Figure 3.5: The Minimum Communication strategy flow diagram.

transfers $3\sum_{j\in N}|C_{i,j}|$ integers, since for each voxel node i requires from another node, the x, y and z-coordinates of the voxel are requested[3] Assuming 4-byte integers and that the communication time is approximately modeled by Equation 3.2, and the scenario that all nodes must transfer to all $|N|-1$ other nodes, an upper bound on the communication time between Step 2 and 3 is

$$T_{\mathsf{comm,Step23}} = (|N|-1)\alpha + 12\beta \max\left(\max_{i\in N}\sum_{j\in N}|C_{i,j}|, \max_{i\in N}\sum_{j\in N}|C_{j,i}|\right) \tag{3.5}$$

This formula has the following rationale: All nodes $i \in N$ can send the $|C_{i,j}|$ voxel requests

[3]One additional number is sent in order to transmit the size of the $C_{i,j}$ set, so that the receiver can allocate memory. This time is, however, negligible and does not induce additional latency since it is sent simultaneously with the actual $C_{i,j}$ sets.

in parallel. Therefore, the latency is $(|N| - 1)\alpha$ if we assume that all nodes need to transfer voxels to other nodes. The time is therefore dominated by the node i which has the largest sum of $|C_{i,j}|$ over all js. The time a node spends sending requests in Step 2 is $\sum_{j \in N} |C_{i,j}|$. Taking the maximum of these values for all i gives the longest time a node uses for sending requests. However, the time could also be dominated by one node having to *receive* a large amount of voxels in Step 3. The time node i spends on receiving requests is $\sum_{j \in N} |C_{j,i}|$ — and again, taking the maximum with respect to i gives the longest time a node will spend receiving requests. The formula in Equation 3.5 follows.

The communication in Steps 3 and 4 is analogous, but with reverse directions: node i will transfer $\sum_{j \in N} |C_{j,i}|$ voxels in Step 3, and will receive $\sum_{j \in N} |C_{i,j}|$ voxels in Step 4. In our case, we assume that each voxel is a 4-byte floating-point number, so the time is:

$$T_{\mathsf{comm,Step34}} = (|N| - 1)\alpha + 4\beta \max\left(\max_{i \in N} \sum_{j \in N} |C_{i,j}|, \max_{i \in N} \sum_{j \in N} |C_{j,i}|\right) \tag{3.6}$$

The computation time in Steps 2, 3 and 4 are bounded by

$$\begin{aligned}
T_{\mathsf{comp,Step2}} &= O(\max_{i \in N} \sum_{j \in N} |C_{i,j}|) && (3.7)\\
&= k_2 \max_{i \in N} \sum_{j \in N} |C_{i,j}| + k_2' && (3.8)\\
T_{\mathsf{comp,Step3}} &= O(\max_{i \in N} \sum_{j \in N} |C_{j,i}|) && (3.9)\\
&= k_3 \max_{i \in N} \sum_{j \in N} |C_{j,i}| + k_3' && (3.10)\\
T_{\mathsf{comp,Step4}} &= O(\max_{i \in N} \sum_{j \in N} |C_{i,j}|) + T_{\mathsf{other}}(\max_{i \in N} A(F^i)) && (3.11)\\
&= k_4 \max_{i \in N} \sum_{j \in N} |C_{i,j}| + k_4' + T_{\mathsf{other}}(\max_{i \in N} A(F^i)) && (3.12)
\end{aligned}$$

where $T_{\mathsf{other}}(\max_{i \in N} A(F^i))$ is an application-dependent computation over all the voxels on node i subsequent to transfer, and k_i and k_i' are positive constants.

The rationale for these equations are as follows: In Step 2, each node i must iterate over $\sum_{j \in N} |C_{i,j}|$ voxels in order to fill the request buffer. The time in this step is thus bounded by the node with the maximum number of voxels to be transmitted to other nodes. This is also the case in Step 4, where received coordinates must be inserted into a local voxel table. Also, in Step 3, a node iterates over all the voxels to be transmitted to other nodes.

Writing $K = k_1'' + k_2' + k_3' + k_4'$, the total computation time for this method $T_{\mathsf{MinTransfer,comp}}$ is thus bounded by

$$
\begin{aligned}
T_{\mathsf{MinTransfer,comp}} &= T_{\mathsf{comp,Step1}} + T_{\mathsf{comp,Step2}} + T_{\mathsf{comp,Step3}} + T_{\mathsf{comp,Step4}} && (3.13)\\
&= k_1|F| + k_1' \max_{i \in N} A(F^i) + (k_2 + k_4) \max_{i \in N} \sum_{j \in N} |C_{i,j}| && (3.14)\\
&\quad + k_3 \max_{i \in N} \sum_{j \in N} |C_{j,i}| + K && (3.15)
\end{aligned}
$$

The total time for executing the EXTRACT-ALL operations (and computations done subsequently, represented by the application-dependent T_{other} function) is therefore

$$
\begin{aligned}
T_{\mathsf{MinTransfer}} &= T_{\mathsf{MinTransfer,comp}} + T_{\mathsf{comm,Step23}} + T_{\mathsf{comm,Step34}} && (3.16)\\
&= k_1|F| + k_1' \max_{i \in N} A(F^i) + (k_2 + k_4) \max_{i \in N} \sum_{j \in N} |C_{i,j}| + k_3 \max_{i \in N} \sum_{j \in N} |C_{j,i}| && (3.17)\\
&\quad + 2(|N| - 1)\alpha + 16\beta \max \left(\max_{i \in N} \sum_{j \in N} |C_{i,j}|, \max_{i \in N} \sum_{j \in N} |C_{j,i}| \right)\\
&\quad + T_{\mathsf{other}}(\max_{i \in N} A(F^i)) + K
\end{aligned}
$$

Note that this is a worst-case *upper bound*. Since there are no synchronization barriers between the steps in Figure 3.5 on page 40, many of the steps may actually happen in parallel.

Discussion

From Equation 3.17, we can observe:

- The computation time for transferring the voxels and doing associated computations is bounded by the node which is responsible for the faces which in total have the largest area. This is the case since $\sum_{j \in N} |C_{i,j}| \leq A(F^i)$ for all nodes i, since the number of voxels requested from other nodes will never exceed the total area of all faces. In very special cases, the $|F|$ term will dominate, but this is unlikely since the number of faces is likely much less than the total area of all faces. Also, since the running time is $O(A(F) + |F|)$ when running the algorithm using a single-core machine, the theoretical running time of the parallel version using the Minimum Communication strategy is less than the single-core version.
- The more computation, the better: If the last term of Equation 3.17 increases, which means that the application does more computations, the transfer time will be of less significance.
- The efficiency of this strategy depends largely on the $|C_{i,j}|$s. Making a general formula for $|C_{i,j}|$ is in general difficult since $|C_{i,j}|$ depends entirely on the orientation, size and shape of each triangle, how large each node's coordinate space is and how many voxels have been previously cached. In general, though, the less triangles that intersect the boundaries between nodes, the less will $|C_{i,j}|$ be and the less $T_{\mathsf{MinTransfer}}$ will be. The $|C_{i,j}|$s can be reduced by:

- Increasing the coordinate space C_i of each node i. This implies either using *less* nodes, or larger data sets. If the coordinate space grows larger, there is less probability that a triangle will intersect a boundary between two nodes. But then again, $A(F^i)$ will increase, and so will $T_{\mathsf{other}}(\max_{i \in N} A(F^i))$.
- Decreasing the average triangle size. If the average triangle size decreases (and the number of triangles are held constant), the probability of intersecting a boundary also lowers.
- Decreasing the number of triangles — also lowering the probability of intersecting boundaries.
- Moving vertices small distances. If the workload consists of an EXTRACT-ALL operation, followed by computation, followed by MOVE-VERTEX operations repeated several times (as is the case in pillar gridding applications), the less distance each vertex is moved, the less will the $|C_{i,j}|$s be. Therefore, this strategy works best if vertices are moved very small distances. Additionally, if several such cycles are done, the $|C_{i,j}|$s will likely be reduced in each cycle.

The last point in the list above, which is that caching more data reduces the $|C_{i,j}|$s, give rise to the Block Transfer strategy, which we describe in the next section.

Let us summarize by considering the main advantages and disadvantages of the Minimum Communication strategy. Advantages are:

1. *Only voxels required are actually transferred*. With this strategy, we always transfer the minimum amount of voxels between nodes, since each node requests from other nodes precisely the voxels it requires and nothing more.
2. *There is a certain amount of caching*. Consider Figure 3.4 again. For many applications, when moving a vertex, the vertex will not be moved very far. Consequently, it is likely that the new polygon will still intersect some of the voxels which have already been transferred from another node and thus each $|C_{i,j}|$ will be reduced. Therefore, despite the caching of this strategy being minimal, it is probable that some of the voxels of the new polygon already have been transferred.

Disadvantages of this strategy are:

1. *The cache may be too small*. Although there is a certain amount of caching, it may be too limited. Given a triangle whose centroid is on one node's coordinate space but has a vertex in another node's coordinate space (Figure 3.4), if one vertex is moved just slightly, and an extraction operation is performed subsequently, no reduction in the $|C_{i,j}|$s occur and thus there is no reduction in $T_{\mathsf{MinTransfer}}$ the second time.
2. *Voxelization and communication does not happen in parallel*. The first step, which is voxelization of all faces, happens in parallel on all nodes. But this operation takes time, especially if there are many or large polygons. With the Minimum Communication strategy, we must first voxelize all polygons, and *then* communication can start.
3. *Transferring coordinates adds overhead*. From Figure 3.5 and Equation 3.17, notice that:
   - each node transfers a list of coordinates it requests from the other nodes, followed by
   - each node receiving requests and transferring the associated values $V(c)$ of each coordinate c back to the requesting node.

This is time-consuming. For each data value we need, we need to first transfer one coordinate value. Each coordinate consists of three 32-bit integers, while the data is a single 32-bit floating-point number. Thus, we spent more time transferring information about the coordinates than transferring the actual data.

Let us now consider another strategy, which is better at reducing the $|C_{i,j}|$s for subsequent transfers, but requires transferring more data initially.

3.4.2 Strategy 2: Block Transfer Strategy

Description

One of the problems of the previous strategy was that the cache might be very small. Another strategy, then, is to also request neighbors in each dimension for each voxel found by VOXELIZE. This is illustrated in Figure 3.6. In the figure, a block size of b is used. Instead of requesting only each voxel found on the polygon, request a cube extending diagonally $(x - b/2, y - b/2, z - b/2)$ up to and including $(x - b/2, y - b/2, z - b/2)$.

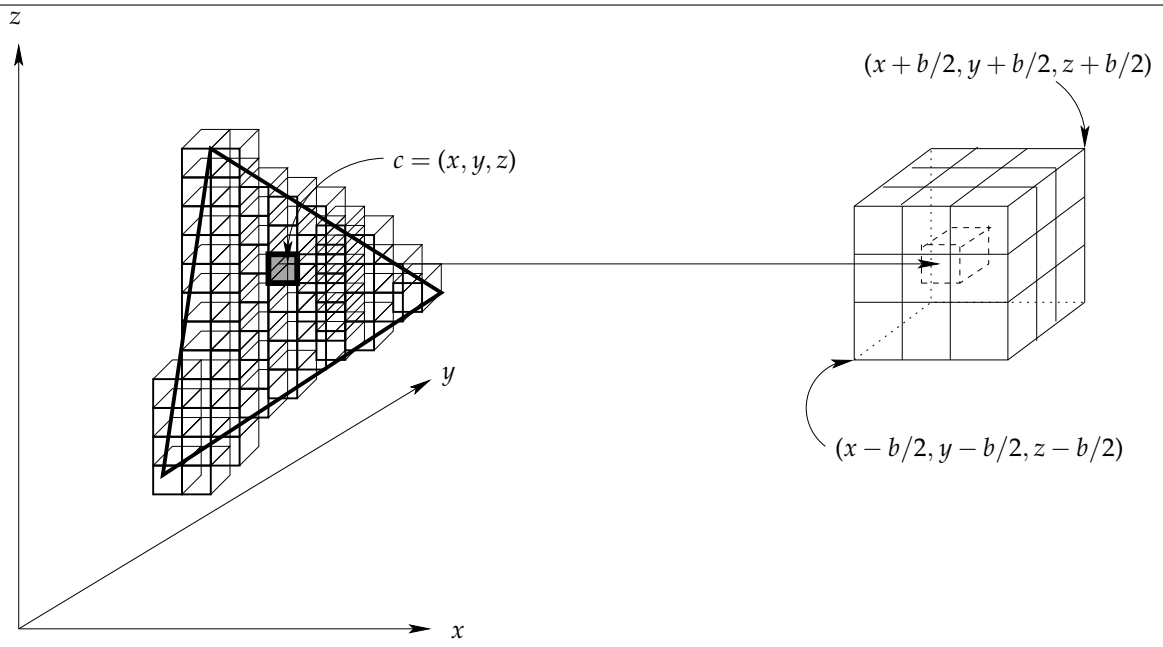


Figure 3.6: The Block Transfer strategy.

The communication between nodes is precisely the same as that of the Minimum Communication strategy of Figure 3.5 on page 40, with one exception: With a block size parameter of b voxels, define the $C'_{i,j}$ sets to be the union of $C_{i,j}$ and the set of voxels $D_{i,j}$ which for each $c \in C_{i,j}$ consists of all voxels in a cube of size b^3 with c in the center. Thus, $C'_{i,j} = C_{i,j} \cup D_{i,j}$. The Block Transfer strategy is precisely the same as the Minimum Communication strategy, except that the $C'_{i,j}$ sets are used instead of $C_{i,j}$.

Analysis

Since the Block Transfer strategy is the same as the Minimum Communication strategy but with $C'_{i,j}$ instead of $C_{i,j}$, the expression for the time of this strategy is already given by Equation 3.17 in the previous section, but with $C'_{i,j}$ instead of $C_{i,j}$. Can we say anything about $|C'_{i,j}|$? In general, we can estimate that b^2 of the voxels in $D_{i,j}$ will already be in $C_{i,j}$. Consider Figure 3.6 again. For the coordinate $c = (x, y, z)$, clearly, some of the voxels in the block surrounding c are also intersecting the polygon. Since the polygon is planar, a reasonable estimate is that b^2 of the voxels in the surrounding block also intersect the polygon.

Therefore, we have

$$|C'_{i,j}| \approx (b^3 - b^2)|C_{i,j}| \qquad (b > 1) \tag{3.18}$$

$$|C'_{i,j}| = |C_{i,j}| \qquad (b = 1) \tag{3.19}$$

Since the expression for the time of executing this strategy $T_{\mathsf{BlockCache}}$ is precisely as $T_{\mathsf{MinTransfer}}$ in Equation 3.17 except using $C'_{i,j}$ instead of $C_{i,j}$. By inserting Equation 3.18, we have for $b > 1$

$$\begin{aligned} T_{\mathsf{BlockCache}} \approx\ & k_1|F| + k'_1 \max_{i \in N} A(F^i) + (k_2 + k_4)(b^3 - b^2) \max_{i \in N} \sum_{j \in N} |C_{i,j}| \\ & + k_3(b^3 - b^2) \max_{i \in N} \sum_{j \in N} |C_{j,i}| + K \\ & + 2(|N| - 1)\alpha + 16\beta(b^3 - b^2) \max \left(\max_{i \in N} \sum_{j \in N} |C_{i,j}|, \max_{i \in N} \sum_{j \in N} |C_{j,i}| \right) \\ & + T_{\mathsf{other}}(\max_{i \in N} A(F^i)) \end{aligned} \tag{3.20}$$

For $b = 1$, $T_{\mathsf{BlockCache}} = T_{\mathsf{MinTransfer}}$. The Minimum Communication strategy is just a special case of the Block Transfer strategy with block size equal to 1.

Discussion

The advantage of using this strategy is that more voxels are cached for future use and thus $C_{i,j}$ will be lower as more EXTRACT-ALL operations (and as b gets higher). However, with increasing b, more voxels are transferred in the initial move-extract cycles.

The appropriate block size depends on how far an average MOVE-VERTEX operation moves a vertex, and how many such operations are done. In general, though, we can say that the less distance a MOVE-VERTEX operation moves a vertex, the less the block size needs to be used in order for cache hits to occur. But this does not necessarily mean that performance is improved: If large block sizes are used, and vertices are moved long distances, communication time may become exceedingly high. Additionally, the more times we move vertices and subsequently do EXTRACT-ALL operations, the more will the benefits be from using

larger block sizes since the $|C_{i,j}|$s will more quickly reduce in size in subsequent executions, provided that vertices are not moved too far.

This strategy improves the caching of the Minimum Communication strategy and may be advantageous if the workload is to interchangeably extract all points of all faces on all nodes, then move vertices, then extract points again, and so on several times, but only if the gain from the increased caching is sufficient to save the increased communication costs. Also, the other disadvantages of the Minimum Communication strategy remain.

Let us now look at another strategy which caches very large blocks, but in turn lets us do voxel extraction and transfer in parallel.

3.4.3 Strategy 3: Bounding Volume Strategy

Description

For a triangle T with vertices $v_i = (v_{i,x}, v_{i,y}, v_{i,z})$ for $i = 1, 2, 3$, the *bounding volume* is the cube which extends from

$$v_{\mathsf{min}} = \left(\min_{i=1,2,3} (v_{i,x}) , \min_{i=1,2,3} (v_{i,y}) , \min_{i=1,2,3} (v_{i,z}) \right) \tag{3.21}$$

to

$$v_{\mathsf{max}} = \left(\max_{i=1,2,3} (v_{i,x}) , \max_{i=1,2,3} (v_{i,y}) , \max_{i=1,2,3} (v_{i,z}) \right) \tag{3.22}$$

The bounding volume strategy operates as follows:

1. Each node i detects all polygon faces whose responsible node is some *other* node $j \in N$, but whose bounding volume intersects the coordinate space of node i, C_i. Let the coordinates of the bounding volume of all faces whose responsible node is j but are located in C_i be $B_{i,j}$. Note that $B_{i,j}$ might be empty.
2. On each node i, the node sends all voxels in $B_{i,j}$ to node j that have not been transferred previously[4]. Also, each node i also starts a receive operation from the other nodes to receive the $|B_{j,i}|$s.
3. Using nonblocking calls, the previous call does not block the application. Therefore, all nodes proceed to voxelize all the faces they are responsible for simultaneously to the data transfer.
4. Next, all nodes wait until the transfers started in Step 2 complete. Then, computations may continue.

A 2D example is shown in Figure 3.7 on the facing page. The figure assumes that no voxels have been cached previously. The black voxels are transferred from node 0 to 1 simultaneously to the polygon being voxelized on node 1.

A flow diagram of the steps in this strategy is shown in Figure 3.8 on page 48.

[4]This is kept track of by associating a bit mask with each voxel coordinate. When a voxel is transferred to node j from i, bit position j is set to 1 and the voxel is not transferred again.

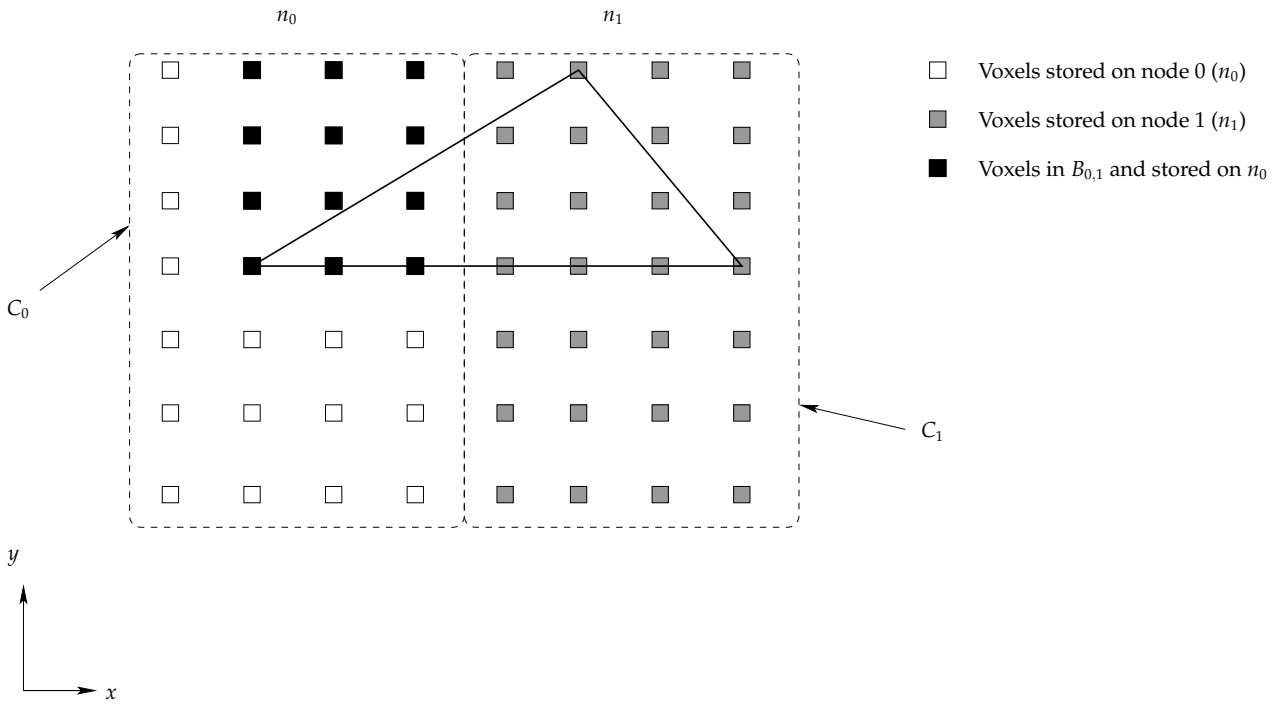


Figure 3.7: The Bounding Volume strategy, shown in 2D for a polygon whose responsible node is n_1.

Analysis

An upper bound for this strategy on a node i is the time it takes to compute the $B_{i,j}$, in addition to the time it takes to check if $B_{j,i} > 0$ for some other node j. The last calculation involves a traversal over all the $|F|$ faces in the polygon model and checking in $O(1)$ time if a face whose responsible node is j has a bounding volume which intersects the coordinate space of node i. The first calculation takes $\sum_{j \in N} |B_{i,j}|$ time. An upper bound on $T_{\mathsf{Step1,BV}}$ is therefore

$$T_{\mathsf{Step1,BV}} = k_1|F| + k_1' \max_{i \in N} \sum_{j \in N} |B_{i,j}| + k_1'' \tag{3.23}$$

where k_1 and k_1' are positive constants. In Steps 2 and 3, sending and receiving $B_{i,j}$ values which are nonempty and voxelizing faces is done in parallel. Hence, the time for these steps is the maximum of the transfer time and the voxelization time. Assuming that the worst-case scenario — all $B_{i,j}$s are nonzero for all nodes i and $j \neq i$ — the time can be expressed as

$$T_{\mathsf{Step23,BV}} = \max \left((|N| - 1)\alpha + 16\beta \sum_{j \in N} |B_{i,j}|, k_2 \max_{i \in N} A(F^i) + k_2' \right) \tag{3.24}$$

Note that we with each voxel value send the coordinate values. On a node i, several faces intersecting C_i but whose responsible node is $j \neq i$ may have intersecting bounding volumes.

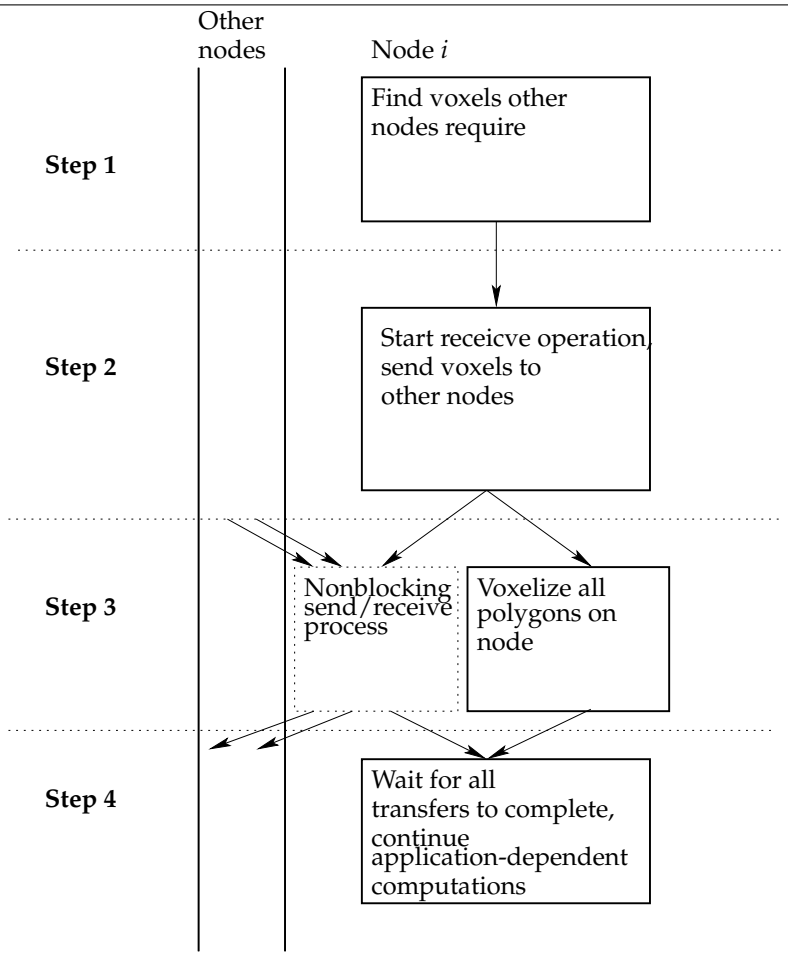


Figure 3.8: Flow diagram for the steps in the bounding volume strategy, when executing EXTRACT-ALL.

In this case, node j does not know precisely how the bounding volume intersect. Furthermore, some coordinates may be already cached on node j so that node i decides not to send them. Therefore, each coordinate is transmitted with each voxel.

Step 4 on a node i consists of inserting the $\sum_{j \in N} |B_{j,i}|$ voxels received into its local cache, and doing other computations:

$$T_{\mathsf{Step4,BV}} = k_2 \max_{i \in N} \sum_{j \in N} |B_{j,i}| + k_2' + T_{\mathsf{other}}(\max_{i \in N} A(F^i)) \tag{3.25}$$

Putting these equations together we have the following estimate for the time this strategy takes:

$$
\begin{aligned}
T_{\mathsf{BoundingVolume}} &= T_{\mathsf{Step1,BV}} + T_{\mathsf{Step23,BV}} + T_{\mathsf{Step4,BV}} && (3.26)\\
&= k_1|F| + k_1' \max_{i\in N} \sum_{j\in N} |B_{i,j}| + k_1'' && (3.27)\\
&\quad + \max\left((|N|-1)\alpha + 16\beta \sum_{j\in N} |B_{i,j}|, k_2 \max_{i\in N} A(F^i) + k_2' \right)\\
&\quad + k_2 \max_{i\in N} \sum_{j\in N} |B_{j,i}| + k_2' + T_{\mathsf{other}}(\max_{i\in N} A(F^i))
\end{aligned}
$$

Can we estimate $|B_{i,j}|$ in terms of $|C_{i,j}|$? Recall that $C_{i,j}$ denotes the voxels located on a node j but whose responsible node is i. $B_{i,j}$ denotes the set of coordinates which belong to a bounding volume of a face whose responsible node is j, but are in node i's coordinate space C_i. One possible approximation is to let

$$
|B_{i,j}| \approx |C_{j,i}|^{\frac{3}{2}} \tag{3.28}
$$

Had we used actual polygon voxel coordinates rather than the bounding volume when defining $B_{i,j}$, we would have had $B_{i,j} = C_{j,i}$. But $B_{i,j}$ measures a 3D space, while $C_{j,i}$ measures only the corresponding 2D space. Therefore, we may approximate the voxels added by multiplying with one 1D unit, which could be estimated to be $\sqrt{|C_{j,i}|}$. From this, Equation 3.28 follows.

Inserting Equation 3.28 into Equation 3.27 yields

$$
\begin{aligned}
T_{\mathsf{BoundingVolume}} &= T_{\mathsf{Step1,BV}} + T_{\mathsf{Step23,BV}} + T_{\mathsf{Step4,BV}} && (3.29)\\
&\approx k_1|F| + k_1' \max_{i\in N} \sum_{j\in N} |C_{j,i}|^{\frac{3}{2}} + k_1'' && (3.30)\\
&\quad + \max\left((|N|-1)\alpha + 16\beta \sum_{j\in N} |C_{j,i}|^{\frac{3}{2}}, k_2 \max_{i\in N} A(F^i) + k_2' \right)\\
&\quad + k_2 \max_{i\in N} \sum_{j\in N} |C_{i,j}|^{\frac{3}{2}} + k_2' + T_{\mathsf{other}}(\max_{i\in N} A(F^i))
\end{aligned}
$$

Discussion

By comparing Equation 3.30 to the expressions for the time taken by the Minimum Communication and Block Transfer strategies (Equations 3.17 and 3.20), we can conclude that the method has the following advantages compared to the other two strategies:

1. *Many voxels are cached*. Therefore, if doing many move-extract cycles, this method should be better than the Minimum Communication strategy (and the blocked strategy for relatively small block sizes), since communication time is eventually minimized.
2. *Communication and voxelization is done in parallel*. Voxelization and communication is done in parallel, which is not the case with the other two methods.

3. *Less messages are passed.* As previously described, the other two methods first send requests and then receive voxels. This induces a latency of $2(|N| - 1)\alpha$ if we assume that all nodes must transmit data to all other nodes, whereas the latency is only $(|N| - 1)\alpha$ with the bounding volume method under the same assumptions.

This method also has some disadvantages compared to the other strategies:

1. *The amount of voxels cached depends on the polygon orientation.* If a polygon lies in either of the principal xy, xz or yz planes, the bounding volume will of the polygon will not be a box (and therefore, the approximation in Equation 3.28 will be an overestimation), but a plane.
2. *The amount of voxels transferred may be very large.* The Minimum Communication strategy transfers data in the order of $O(A(F))$ voxels for each face F, while the bounding volume strategy generally transfers data in the order of $O(A(F)^{3/2})$. With large polygons, this can potentially be a very large number — and thus take very long time to transfer, destroying any potential cache benefits the Bounding Volume strategy provides.

3.5 Comparison of Strategies

The strategy that should be used depends on the application. If the workload consists of a series of move-extract cycles, the Block Transfer or Bounding Volume strategies may be superior to the Minimum Communication strategy since the increased caching will lead to less communication occurring after several such cycles. However, if vertices are moved such that the increased cache does not reduce the communication sufficiently, or if the number of cycles is small, the Minimum Communication strategy will likely win.

Note that the Block Transfer strategy with a large block size and Bounding Volume strategy in general do not scale well with the polygon size and the number of polygons: If a polygon has A voxels on another node which must be transferred, then approximately A voxels are transferred by the Minimum and Block Transfer strategies with a small block size. With the Block Transfer strategy using a large block size or the Bounding Volume strategy, a *volume* instead of a plane is transferred, and the voxels to be transferred are in the order of $A\sqrt{A}$. Therefore, the efficiency of these strategies hinges on the efficiency of the caching mechanism.

The optimal strategy will thus depend on many application-dependent factors: The number of move-extract cycles, the average polygon size and the number of polygons, and the distance each vertex is moved in a move-extract cycle. It is therefore difficult to generally compare the relative performance of the three strategies. We therefore conclude this discussion by stating that if few dynamic operations are used, if vertices are moved relatively far in each dynamic operation, or if there are many polygons or if each polygon has a relatively large area, the better the Minimum Communication strategy (or Block Transfer with small block size) will perform. With an increasing number of dynamic operations, fewer polygons with smaller areas, *and* if vertices are not moved too far in dynamic operations, the performance of the other strategies increases since many of the voxels used in a move-extract cycle have already been transferred in previous cycles.

In Chapter 5, we benchmark the different strategies .

3.6 Efficient Voxelization

So far, we have assumed that the VOXELIZE operation, which takes a polygon face F as argument and returns $V^*(F)$, has been implemented. This procedure, which is used when executing the EXTRACT-ALL operation, must thus convert a face, which in our case is a triangle of three vertices, and detect all discrete points intersected by the face. Now, we consider efficient ways of doing this operation by exploiting the fine-grained parallelism provided by multi-core CPUs and GPUs.

In Section 3.6.1, we explore previous work and alternative approaches to for converting mathematical objects to a discrete set of values. Next, in Section 3.6.2, we propose a new voxelization algorithm for planar objects. Subsequently, Sections 3.6.3 and 3.6.4 describe how the algorithm can be implemented on multi-core CPUs and GPUs, respectively. Finally, we briefly compare the properties of the algorithm to Kaufman's algorithm in Section 3.6.5.

3.6.1 Previous Approaches

Although our framework is not specifically tied to visualization, converting a mathematical object (of two or three dimensions), such as a line, circle or polygon, to a discrete set of 2D values, which is called *rasterization*, has been extensively studied in computer graphics research. Rasterization is routinely done by today's GPUs, since the most common use for rasterization is to compute the color value of each 2D pixel on a display device, such as a CRT or LCD screen. See e.g. [15] for an overview of rasterization techniques.

But since we work with voxels instead of pixels, we need to discretize in 3D. The process of converting mathematical objects to a set of discrete 3D voxels is called *voxelization*. Recall that we used $V^*(F)$ to denote the voxelization of the triangle face F.

Less research has been done on voxelization compared to rasterization. Although there is use for voxelization in volumetric graphics research (focusing on visualizing volume data), the use is limited compared to the very significant use for rasterization. Without efficient rasterization algorithms, displaying any type of graphics based on geometrical descriptions on a computer would be very slow.

A second driver behind voxelization research is 3D screens. But few such screens are in existence compared to the number of 2D screens [43]. However, 3D screens are getting cheaper [44], but some types of screens use merely a set of 2D screens to simulate a 3D effect [45] and thus only require 2D rasterization techniques. But true 3D screens, based on holographic techniques, are under research [45].

If such screens are to become common, voxelization techniques are of more use. Our framework does not require voxelization of solid objects, which in general is a difficult problem [43], but only planar triangles. Some (but relatively few) techniques have been proposed for voxelization of planar objects, which we describe in the following sections. We suggest a previously unsuggested technique for voxelization of triangles, which is suitable for implementation on GPUs later in the chapter. Now, we consider some previous voxelization approaches. A brief overview of the papers referred to below is given in Appendix A.3.

Kaufman's Algorithm

The first polygon voxelization algorithm, suggested by Kaufman in 1988 [46], builds on similar 2D rasterization algorithms. This algorithm computes a bounding *plane* of the polygon. Then, the algorithm iterates over all voxels of the bounding plane and uses a flag to detect whether or not it is inside the polygon. When starting at an edge of the bounding plane, the flag is set to false. As soon as an edge of the polygons is reached, the flag is set to true and the algorithm records subsequent voxels visited, until another edge of the polygon is reached, when the flag again is set to false.

Since the algorithm iterates over a bounding plane of the polygon, the running time is asymptotical in the area of the polygon. The algorithm is illustrated in Figure 3.9. In the figure, all voxels intersecting the bounding plane of the triangle (dashed) are visited. Note that the polygon and bounding plane may have any 3D orientation, and is not necessarily aligned with the xy-plane as shown in this figure.

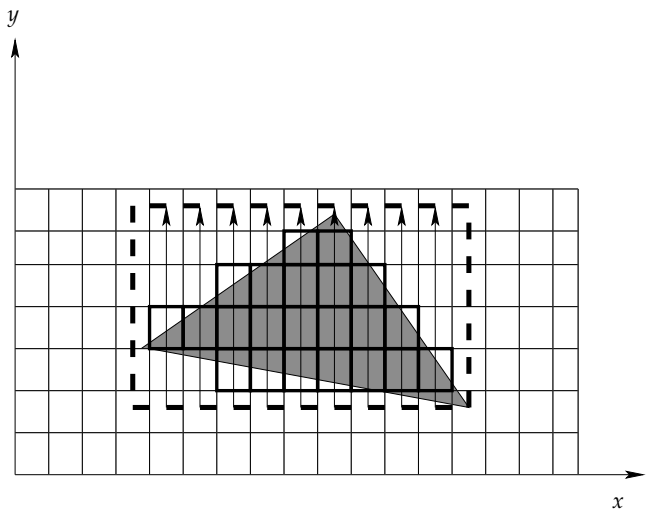


Figure 3.9: Kaufman's polygon voxelization algorithm.

This algorithm is incapable of taking advantage of the GPU rasterization hardware (since it works in 3D). Additionally, it uses two branch instructions in the inner loop: One branch for toggling the flag saying whether the algorithm is inside or outside of the polygon, and one for checking the value of the flag.

Recent Voxelization Research

Recent research has focused on voxelizing more complicated objects than polygons; see e.g. [43, 47]. There exists a more sophisticated polygon voxelization algorithm which guarantees that voxelized polygon is always of a specific thickness [48]. The idea is to put a polygon between two planes which have a specific distance from each other, and assign all voxels in the volume spanned by the two planes as the voxels of the polygon. The algorithm is relatively slow. We do not require a thickness guarantee in our applications, so we do not use this algorithm.

We also focus exclusively on convex triangles. Therefore, more recent (and computationally slower) voxelization algorithms such as [49], which is able to handle all polygons (including complex ones), are unnecessary.

Other voxelization techniques have been proposed for efficient hardware implementation [6]. However, these algorithms are specifically aimed at optimizing voxelization for visualization purposes. For example, given a triangle, these algorithms may decide that the triangle is far enough from the view point to only be roughly voxelized; or a low-pass filter may be applied to smooth the cubic look of an object after voxelization [50]. This is not relevant to our work, since we require *all* voxels of the polygon and not only those viewable from a specific point.

Voxelization on the GPU

Some methods have recently been proposed for voxelization on the GPU. [51] proposes a GPU-based triangle mesh voxelization algorithm which exploits the GPU's 2D rasterization hardware by rasterizing the triangle along an axis where the triangle has the greatest area, and then encoding the resulting 3D coordinates in a set of 2D GPU textures. [52] suggests a similar approach, but used for a different application. [53, 54] also do voxelization on the GPU, but consider solid voxelization and not surface-based voxelization.

These methods are designed for visualization. For our purposes, they can not be used for two reasons. First, we require that we can somehow download the voxel coordinates to the CPU subsequently. Since the methods are aimed at visualization, the 3D coordinates of the rasterized voxels are only kept implicitly. For example, the method in [51] writes voxels to textures by applying blending operations and never stores the voxel coordinates explicitly. Second, the methods are too limited for our use: The maximum triangle size is limited by the voxel space. For instance, by using GPU textures of size 2048×2048 to write data, we should be able to handle triangles whose area is 2048×2048. However, with this texture size, the triangle may only extend 512 units in any direction using the method in [51], since the method emulates 3D space by using many 2D textures. This can be particularly disadvantageous if we want to download data to the CPU, because in general, it is more efficient to download large amounts of data than small amounts [27].

We now introduce a polygon voxelization method which does not have these two problems and is more suitable for our purposes. As previous methods, we also encode 3D voxels in 2D textures, but our encoding scheme is different in that it allows for easy download to the CPU, and is not optimized for visualization.

3.6.2 A New Approach for Voxelizing Triangles

We now propose a new approach for voxelizing triangles. Given three 3D vertices v_1, v_2 and v_3, our key idea is to apply *transformations* to the vertices so that all vertices lie in the plane $z = 0$. If we only use shape- and length-preserving transformations such as translations (moving a point to a new location) and rotation (rotating a point about an axis a specified number of degrees), we may proceed to extract all *2D pixels* of the corresponding 2D triangle. Next, we can apply an inverse transformation to all the pixels found to produce the 3D voxels. Figure 3.10 on the next page illustrates our idea. Note that the set of voxels in the left part of the figure are not necessarily in a plane parallel to the xz plane.

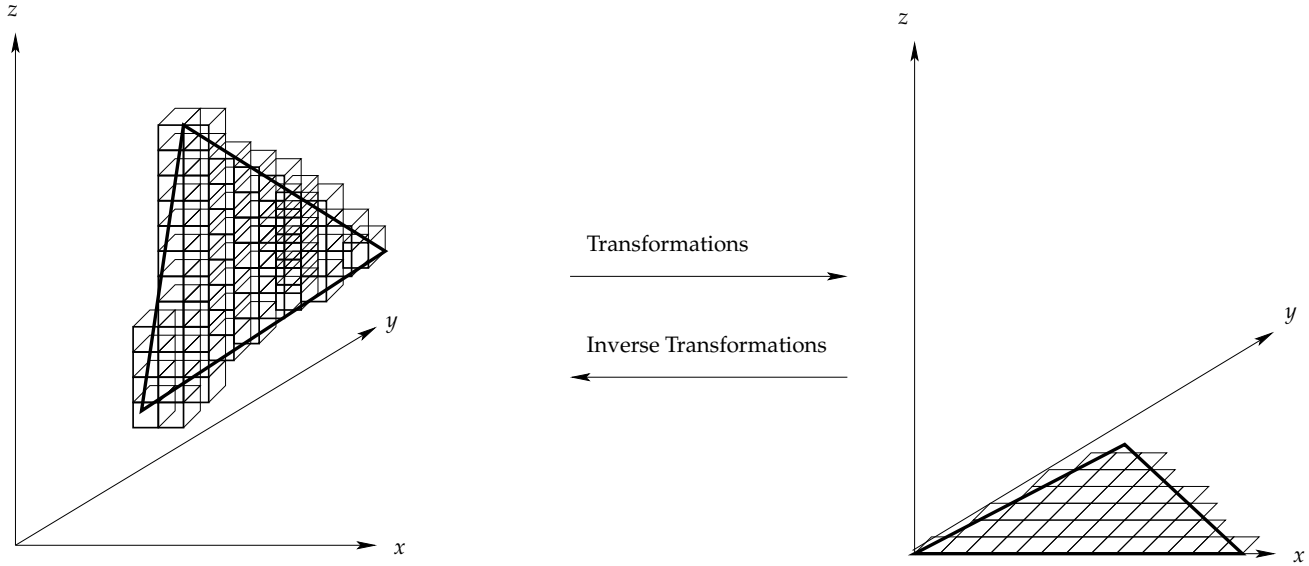


Figure 3.10: Converting the triangle voxelization problem to a rasterization problem by using shape- and length-preserving transformations.

Putting a triangle into the $z = 0$ plane requires a sequence of a translation and three rotations and translations. When we have found all the 2D pixels using a 2D rasterization algorithm, we need to apply the inverse rotations and translations to each pixel to obtain the 3D voxel. However, fortunately, *any* combination of translation and rotation transformations applied to a coordinate in 3D can be written using a single matrix-vector multiplication, where the matrix is of size 4×4 (see e.g. [15] for a proof). If we therefore merge all translations and rotations to a single matrix, any number of rotations and translations will computationally cost precisely the same as one rotation and one translation.

Let us now describe how to do the transformations of Figure 3.10. Figure 3.11 on the facing page illustrates the entire process.

Step 1. Translation

Let the triangle's vertices be v_1, v_2 and v_3. It is convenient to start with moving one of the vertices to the origin; that is, we seek to move or *translate* the triangle such that one of the vertices is at coordinate $(0, 0, 0)$:

$$v_i \mapsto (0, 0, 0) \tag{3.31}$$

This can be obtained by simply subtracting the coordinates of a vertex from the coordinates of all the vertices. Suppose that we choose v_1 to be the vertex located at the origin. Then, we apply the translation transformation and obtain new, translated vertices $v_1^{(1)}, v_2^{(1)}, v_3^{(1)}$ by computing:

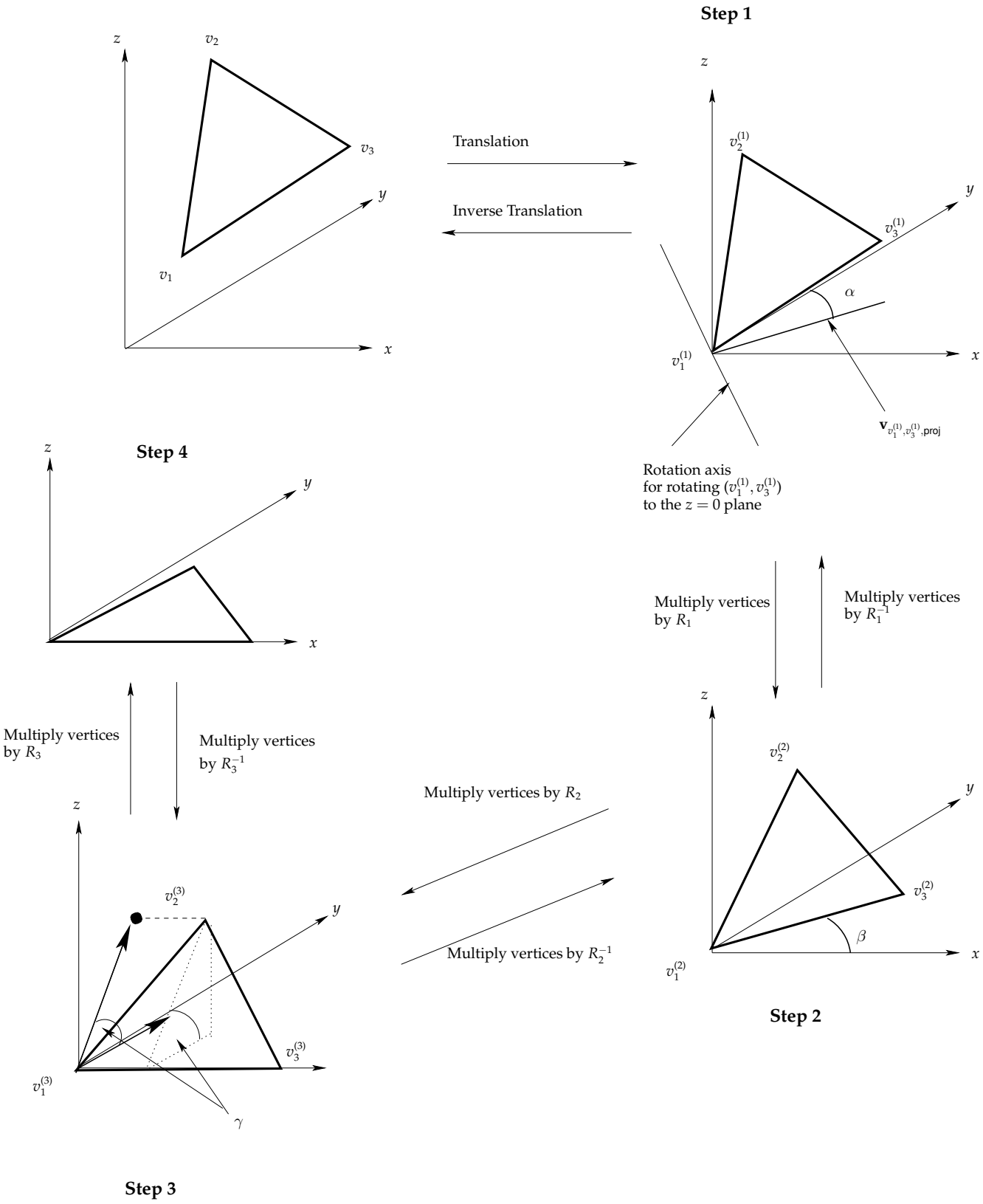


Figure 3.11: Rotating a triangle into the $z = 0$ plane, such that the longest edge of the triangle is aligned with the x-axis.

$$v_1^{(1)} = (0,0,0) \tag{3.32}$$
$$v_2^{(1)} = v_2 - v_1 \tag{3.33}$$
$$v_3^{(1)} = v_3 - v_1 \tag{3.34}$$

Optionally, we may write the above equations as a matrix-vector product, assuming that the x, y and z-components of v_i are $v_{i,x}$:

$$\begin{bmatrix} v_{i,x}^{(1)} \\ v_{i,y}^{(1)} \\ v_{i,y}^{(1)} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -v_{1,x} \\ 0 & 1 & 0 & -v_{1,y} \\ 0 & 0 & 1 & -v_{1,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{i,x} \\ v_{i,y} \\ v_{i,z} \\ 1 \end{bmatrix} \tag{3.35}$$

for $i = 1, 2, 3$.

Step 2. Rotating $(v_1^{(1)}, v_3^{(1)})$ to the $z = 0$ plane

Now, we rotate the triangle so that the orientation becomes as the triangle after Step 2 in Figure 3.11. *Rotation* is the operation of rotating coordinates about an axis by a specific number of degrees. The first rotation we may do is rotating the edge $(v_1^{(1)}, v_3^{(1)})$ to the $z = 0$ plane. To do this, we can rotate $v_2^{(1)}$ and $v_3^{(1)}$ about a line *perpendicular* to $(v_1^{(1)}, v_3^{(1)})$ which passes through the origin *and* is in the $z = 0$ plane. If we rotate the vertices of the triangle α degrees (see Figure 3.11) about this line, we have obtained a triangle whose $(v_1^{(1)}, v_3^{(1)})$ edge lies in the $z = 0$ plane.

First, we must find the angle α. Second, we must find a line that is (1) perpendicular to $(v_1^{(1)}, v_3^{(1)})$, (2) has a z-coordinate equal to zero, and (3) passes through the origin. First, we find a vector $\mathbf{v}_{v_1^{(1)}, v_3^{(1)}}$ from $v_1^{(1)}$ to $v_3^{(1)}$. Since $v_1^{(1)}$ is at the origin, we simply have

$$\mathbf{v}_{v_1^{(1)}, v_3^{(1)}} = v_3^{(1)} \tag{3.36}$$

Now, if the components of $v_3^{(1)}$ are $(v_{3,x}^{(1)}, v_{3,y}^{(1)}, v_{3,z}^{(1)})$, the projection of the line between $v_1^{(1)}$ and $v_3^{(1)}$ in the xy-plane is given by $\mathbf{v}_{v_1^{(1)}, v_3^{(1)}, \text{proj}} = (v_{3,x}^{(1)}, v_{3,y}^{(1)}, 0)$. The angle α can now be found by using the definition of the vector dot-product:

$$\cos\alpha = \frac{\mathbf{v}_{v_1^{(1)}, v_3^{(1)}} \bullet \mathbf{v}_{v_1^{(1)}, v_3^{(1)}, \text{proj}}}{|\mathbf{v}_{v_1^{(1)}, v_3^{(1)}}||\mathbf{v}_{v_1^{(1)}, v_3^{(1)}, \text{proj}}|} \tag{3.37}$$

Taking the arc cosine of Equation 3.37, we have now obtained α. Note that if the z-component of the vector from $v_1^{(1)}$ to $v_3^{(1)}$ is negative, we must negate α since we must do a clockwise

(instead of counterclockwise[5]) rotation of α degrees about the rotation axis. That is,

$$\alpha' = \begin{cases} \alpha & (v_{3,z}^{(1)} \geq 0) \\ -\alpha & (v_{3,z}^{(1)} < 0) \end{cases} \tag{3.38}$$

The rotation axis is simply a line perpendicular to $\mathbf{v}_{v_1^{(1)},v_3^{(1)},\text{proj}} = (v_{3,x}^{(1)}, v_{3,y}^{(1)}, 0)$. One perpendicular line is $(-v_{3,y}^{(1)}, v_{3,x}^{(1)}, 0)$. The rotation axis is displayed in the Step 1 part of Figure 3.11[6].

Thus, we require rotating $v_1^{(1)}, v_2^{(1)}$ and $v_3^{(1)}$ about the rotation axis α' degrees. How do we do the actual rotation? It turns out that using a mathematical construct known as a *quaternion* to represent the rotation makes it simple to do rotation about an arbitrary axis. Combining two quaternions into one quaternion is also easily done. Finally, straightforward formulas exist for obtaining the 4×4 rotation matrix from the corresponding quaternion.

We will not describe quaternions in detail presently, but we include a brief description of how they work in Appendix E.2 on page 130. In this context, it suffices to know that a quaternion q is defined using an axis, represented by a vector $\mathbf{v}$, which is the rotation axis, and a scalar value, which corresponds to the counterclockwise angle about the axis we are going to rotate:

$$q = (\mathbf{v}, s) \tag{3.39}$$

From the quaternion, we can easily obtain the rotation matrix, and we can also easily combine several quaternions into one quaternion. Note that the alternative to using quaternions is to do decompose a rotation along the x, y and z axes, which is tedious. Using quaternions is more elegant. For details, consult the appendix.

Thus, to rotate the triangle such that $(v_1^{(1)}, v_3^{(1)})$ is in the $z = 0$ plane, we construct a new quaternion $q_1 = (\mathbf{v}, s)$ where $\mathbf{v} = (-v_{3,y}^{(1)}, v_{3,x}^{(1)}, 0)$ and $s = \alpha'$ and obtain the rotation matrix R_1 from the quaternion. Then we multiply R_1 with $v_1^{(1)}, v_2^{(1)}$ and $v_3^{(1)}$ to obtain the vertices $v_1^{(2)}, v_2^{(2)}$ and $v_3^{(2)}$ of the new, rotated triangle

$$v_2^{(2)} = R_1 v_2^{(1)} \tag{3.40}$$
$$v_3^{(2)} = R_1 v_3^{(1)} \tag{3.41}$$

(Note that $v_1^{(2)} = (0, 0, 0)$ prior to the rotation, so it is unnecessary to multiply this vertex by R_1).

Subsequent to the rotation, the triangle will be oriented as shown in Step 2 of Figure 3.11 on page 55.

[5]We use the convention that positive angles about an axis correspond to counterclockwise rotations when looking in the direction of the negative axis. See [15].

[6]A special case occurs when $v_{3,x}^{(1)} = v_{3,y}^{(1)} = 0$. In that case, $\alpha = \pi/2$ and the rotation axis is a unit vector along the y axis.

Step 3. Aligning $(v_1^{(2)}, v_3^{(2)})$ with the x-axis

Next, we need to align the vertex $(v_1^{(2)}, v_3^{(2)})$ with the x-axis — that is, the transformation from Step 2 to Step 3 in Figure 3.11. In this case, we rotate all triangle vertices β degrees about the z-axis. β can be found by using the dot product definition on the unit x-vector with the vector $\mathbf{v}_{v_1^{(2)}, v_3^{(2)}} = v_3^{(2)}$[7]:

$$\beta = \arccos\left(\frac{\mathbf{v}_{v_1^{(2)}, v_3^{(2)}} \bullet (1,0,0)}{|\mathbf{v}_{v_1^{(2)}, v_3^{(2)}}|}\right) \tag{3.42}$$

$$= \arccos\left(\frac{v_{3,x}^{(2)}}{|\mathbf{v}_{v_1^{(2)}, v_3^{(2)}}|}\right) \tag{3.43}$$

Analogously to α, if $v_{3,y}^{(2)}$ is positive we must rotate with $-\beta$ (since, in that case, we do a clockwise rotation of β degrees). The opposite is the case when $v_{3,y}^{(2)}$ is negative. Hence, the rotation angle β' is given by

$$\beta' = \begin{cases} -\beta & (v_{3,y}^{(2)} \geq 0) \\ \beta & (v_{3,y}^{(2)} < 0) \end{cases} \tag{3.44}$$

We can then construct the quaternion $q_2 = ((0,0,1), \beta')$ and obtain a second rotation matrix R_2 from q_2. We then multiply $v_2^{(2)}$ and $v_3^{(2)}$ by R_2 to obtain new coordinates $v_1^{(3)}, v_2^{(3)}$ and $v_3^{(3)}$, as shown in Step 3 of Figure 3.11. (Note that $v_1^{(2)} = v_1^{(3)}$).

Step 4. Rotate $v_2^{(3)}$ to the $z = 0$ plane

We need one more rotation before we are done. The final step is to rotate $v_2^{(3)}$ to the $z = 0$ plane, that is, the transformation between the Step 3 and Step 4 triangles in Figure 3.11. First, we find the angle γ between the projection of $v_2^{(3)} = (v_{2,x}^{(3)}, v_{2,y}^{(3)}, v_{2,z}^{(3)})$ in the xy-plane, $v_{2,\text{proj}}^{(3)} = (v_{2,x}^{(3)}, v_{2,y}^{(3)}, 0)$, and $v_2^{(3)}$.

From the Step 3 part of of Figure 3.11, it is evident that we may rotate $v_2^{(3)}$ to the $z = 0$ plane by a rotation about the x-axis. To find the rotation angle, we project $v_2^{(3)}$ and $v_{2,\text{proj}}^{(3)}$ to the yz-plane by setting the x-components of both vectors to zero. The angle γ between these lines is thus the same as the angle between a vector from the x-axis to $v_2^{(3)}$, perpendicular to the x-axis, and the projection of this vector to the xy-plane, which is the precisely the rotation angle about the x-axis with to project $v_2^{(3)}$ to the $z = 0$ plane. This is shown in the Step 3 part of Figure 3.11. γ can thus be found by:

[7] If $|\mathbf{v}_{v_1^{(2)}, v_3^{(2)}}| = 0$, the triangle is degenerate. In our implementations, we check for this case before doing the rotations. In the rest of the derivation, we assume that the triangle is *not* degenerate.

$$\gamma = \arccos\left(\frac{(0, v_{2,y}^{(3)}, v_{2,z}^{(3)}) \bullet (0, v_{2,y}^{(3)}, 0)}{|(0, v_{2,y}^{(3)}, v_{2,z}^{(3)})||(0, v_{2,y}^{(3)}, 0)|}\right) \tag{3.45}$$

$$= \arccos\left(\frac{(v_{2,y}^{(3)})^2 + v_{2,y}^{(3)} v_{2,z}^{(3)}}{v_{2,y}^{(3)}\sqrt{(v_{2,y}^{(3)})^2 + (v_{2,z}^{(3)})^2}}\right) \tag{3.46}$$

Note that we assume that the triangle is not degenerate, so we are guaranteed that $v_{2,y}^{(3)} \neq 0$.

Similarly to α' and β', the rotation angle γ' we must use depends on the orientation of the vertices. To get the correct angle, we let

$$\gamma' = \begin{cases} -\gamma & (v_{2,y}^{(3)} \geq 0 \wedge v_{2,z}^{(3)} \geq 0) \vee (v_{2,y}^{(3)} < 0 \wedge v_{2,z}^{(3)} < 0) \\ \gamma & \text{else} \end{cases} \tag{3.47}$$

This can easily be seen by examining each case. Next, we construct a new quaternion $q_3 = ((1,0,0), \gamma')$ and obtain the corresponding rotation matrix R_3. After multiplying R_3 and $v_3^{(3)}$, the triangle is now in the $z = 0$ plane as shown in Step 4 of Figure 3.11[8].

Summary

Let us summarize our steps:

1. First, we translated to the origin the vertex of the longest edge of the triangle which was closest to the origin.
2. Second, we rotated the longest edge of the triangle into the $z = 0$ plane.
3. Third, we rotated the longest edge to align with the x-axis.
4. Fourth, we rotated the final vertex to the $z = 0$ plane.

Pseudocode for this algorithm is listed in Appendix E.3.1.

Figure 3.12 on the next page shows how we can proceed to do the actual voxelization. After the triangle has been rotated to the $z = 0$ plane, the following steps are executed:

1. First, we rasterize the two triangle lines using a line voxelization algorithm, such as Bresenham's algorithm, described in Appendix E.1. Let the sets rasterized coordinates of these lines be L_1 and L_2.
2. Next, we iterate over all the x-values until the rightmost vertex on the x-axis is reached. For each x-coordinate, we iterate over y-coordinates until an element in L_1 or L_2 is reached, in which case we proceed to the next x-coordinate.
3. Then comes the main trick: Consider Figure 3.12. For each coordinate visited in Step 2, we apply the *inverse* transformation to the coordinate we are visiting. With this method, we essentially transform the 2D pixels of the transformed triangle to 3D voxels of the original triangle.

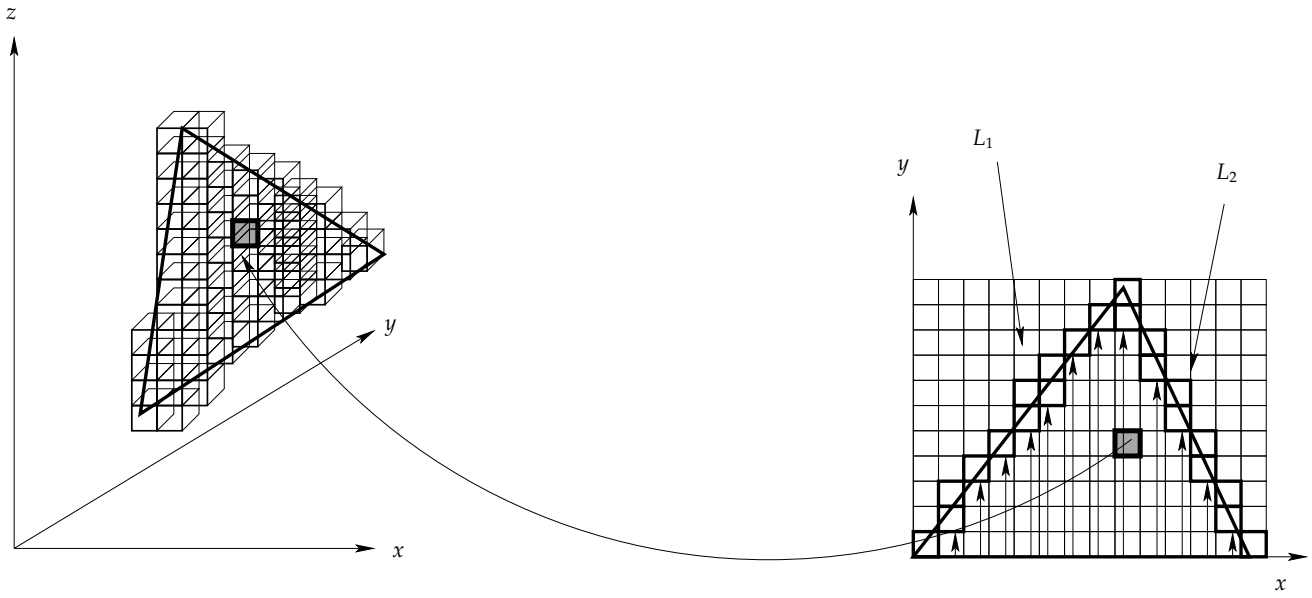


Figure 3.12: The new voxelization algorithm

Pseudocode for this algorithm is shown in Appendix E.3.2.

Now, let us consider efficient implementation of this algorithm on CPUs and GPUs.

3.6.3 CPU Implementation

We now discuss two things that can be done to speed up the algorithm described in the previous section on CPUs.

Multi-Core Parallelization

The voxelization algorithm we described is easily parallelizable. The domain decomposition is illustrated in Figure 3.13 on the facing page.

As the figure shows, each processing core is assigned a subset of the x values. As described in Chapter 2, the OpenMP programming interface provides support for such programming constructs.

One problem with this approach is that in general, the cores which are assigned x-values near the middle of the triangle will have more work than processors assigned x-values close to the edges. For example, core 2 will have more work than core 1 and 3 in Figure 3.13. Furthermore, the entire problem may be bounded by memory speed and not computational speed — which will give us little speedup if using cores which share a memory bus. We return to benchmarking the performance of this parallel version in Chapter 5.

[8]Note that v_2's y-component might be negative after the rotation. In that case, we apply a mirroring transformation about the x-axis — that is, negate the y-component. To avoid too many special cases in our discussion, we assume that the y-component is positive, however, our implementations also deal with this case.

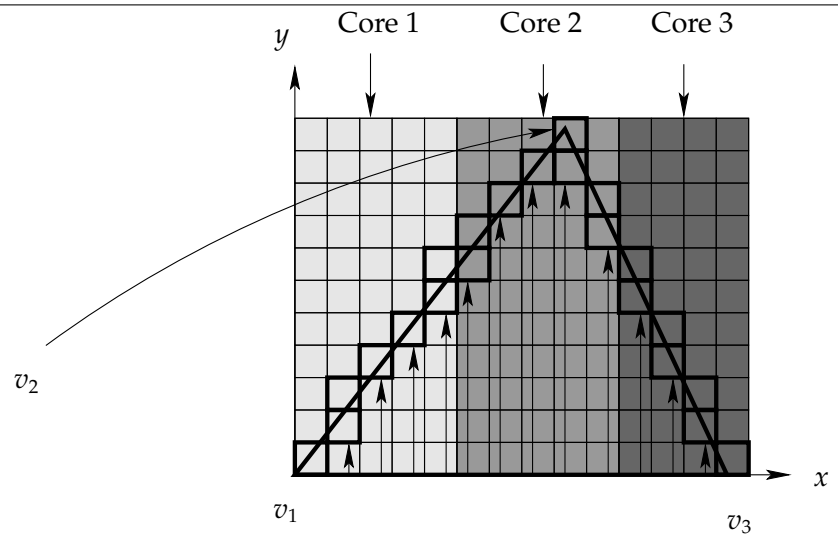


Figure 3.13: The multi-core VOXELIZE-FACE algorithm, using 3 cores

Removing the Inner Loop Branch

Recall that in our description of the voxelization algorithm, we constructed the L_1 and L_2 arrays in order for the voxelization algorithm to know when to stop increasing the y value and move on to the next x-value. This implies that every time a coordinate c is visited, we must check whether or not $c \in L_1$ or $c \in L_2$. This can also be seen in the pseudocode in Appendix E.3.2.

This gives a branch instruction in the inner loop. With this particular branch, guaranteed is at least one branch misprediction per iteration of the loop, since there is no clear pattern as to when a coordinate is in L_1 or L_2. This depends entirely on the triangle which is being voxelized.

In general, branches in inner loops should be avoided. One possible strategy is doing extra work [55], which is what we can do here.

We will preprocess the L_1 and L_2 lists. Consider Figure 3.13 again, where entries in L_1 and L_2 are marked with bold squares. Notice that for some x-values, there are multiple entries in L_1 or L_2. For example, for $x = 0$, there is only one entry in L_1, but not for $x = 2$.

Let us construct the array L_c. The entry $L_c[k]$ denotes the *minimum y-coordinate of the pixels in either L_1 or L_2 with x-value k*. This is illustrated in Figure 3.14 on the next page.

If we construct this array, which can be done in $O(|L_1| + |L_2|)$ time, we can change the loop iterating over y-values to only iterate up to $L_c[x]$. This also means that the branch can be safely removed. The program has still precisely the same output as before.

The optimized program using the L_c array is shown in Appendix E.3.3.

3.6.4 GPU Implementation

We now show how our voxelization algorithm can be implemented on the Graphics Processing Unit (GPU) (an introduction to GPUs was given in Section 2.5.2 on page 21). The GPU

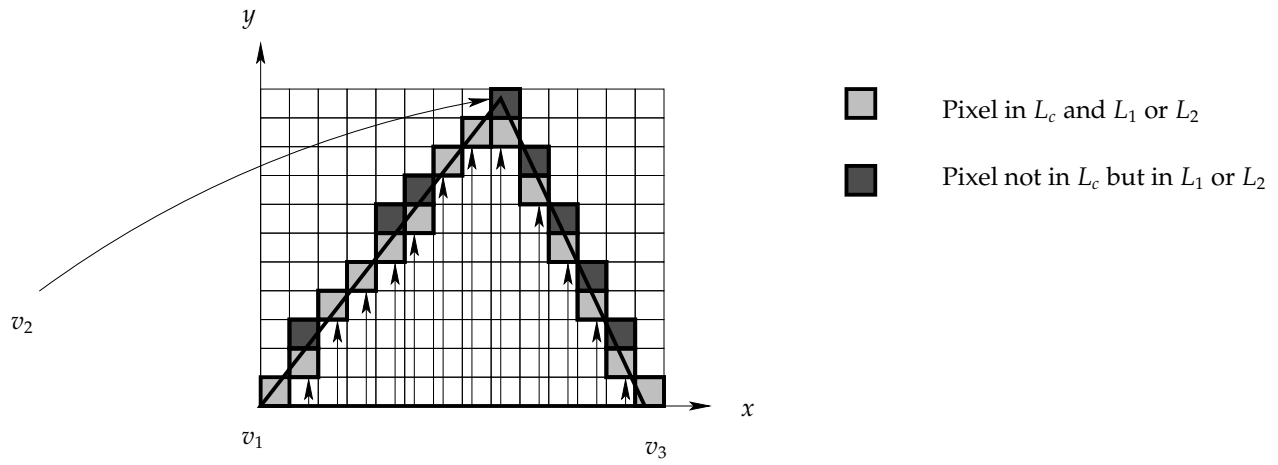


Figure 3.14: The L_c array

rasterization unit essentially does the loops iterating over the coordinates of the triangle in hardware, in a massively parallel fashion. Each coordinate of each rasterized pixel is then passed to the fragment processors.

To exploit the GPU hardware to do voxelization, we do the following three steps:

1. First, we transform the triangle to be voxelized to the $z = 0$ plane, as described before. These vertices are then passed to the GPU. (The triangle is transformed to the $z = 0$ plane by the CPU, since this is a relatively cheap operation.)
2. Next, a fragment processor program is run for each pixel belonging to the triangle. *This fragment processor program applies the inverse rotations and the inverse translation to each coordinate, and encodes the x, y, z coordinates inside the resulting 4-component RGBA 2D texture.*
3. Finally, we download and iterate over the resulting 2D texture on the CPU, and for each 4-component entry, record the coordinates of the corresponding 3D voxel by reading the encoded coordinates.

The fourth component of the 4-component texture is a flag, which is always set to 1 by our fragment processor program. This way, the CPU knows that the other values in the downloaded texture are actual 3D coordinates. The texture is initially cleared so that all other entries are marked with zero. The process is shown in Figure 3.15 on the next page.

This computation is efficient on the GPU. The entire rotation and translation takes at most *four* clock cycles on the GPU for *both* the x, y and z-components, since it can be implemented as a 4×4 matrix multiplication operation, which can be implemented with four 4-component dot products — which take at most four cycles on the GPU [27][9]. No branches are necessary in the fragment programs, which is very important on the GPU since it has limited branch-prediction capabilities.

To ensure cross-platform compatibility we have used the OpenGL graphics library and

[9]Our program does one additional translation to fit the triangle into the GPU texture memory. Therefore our GPU fragment processor program must apply an inverse transformation prior to the matrix multiplication. This may require a few extra cycles, but the program is still very compact.

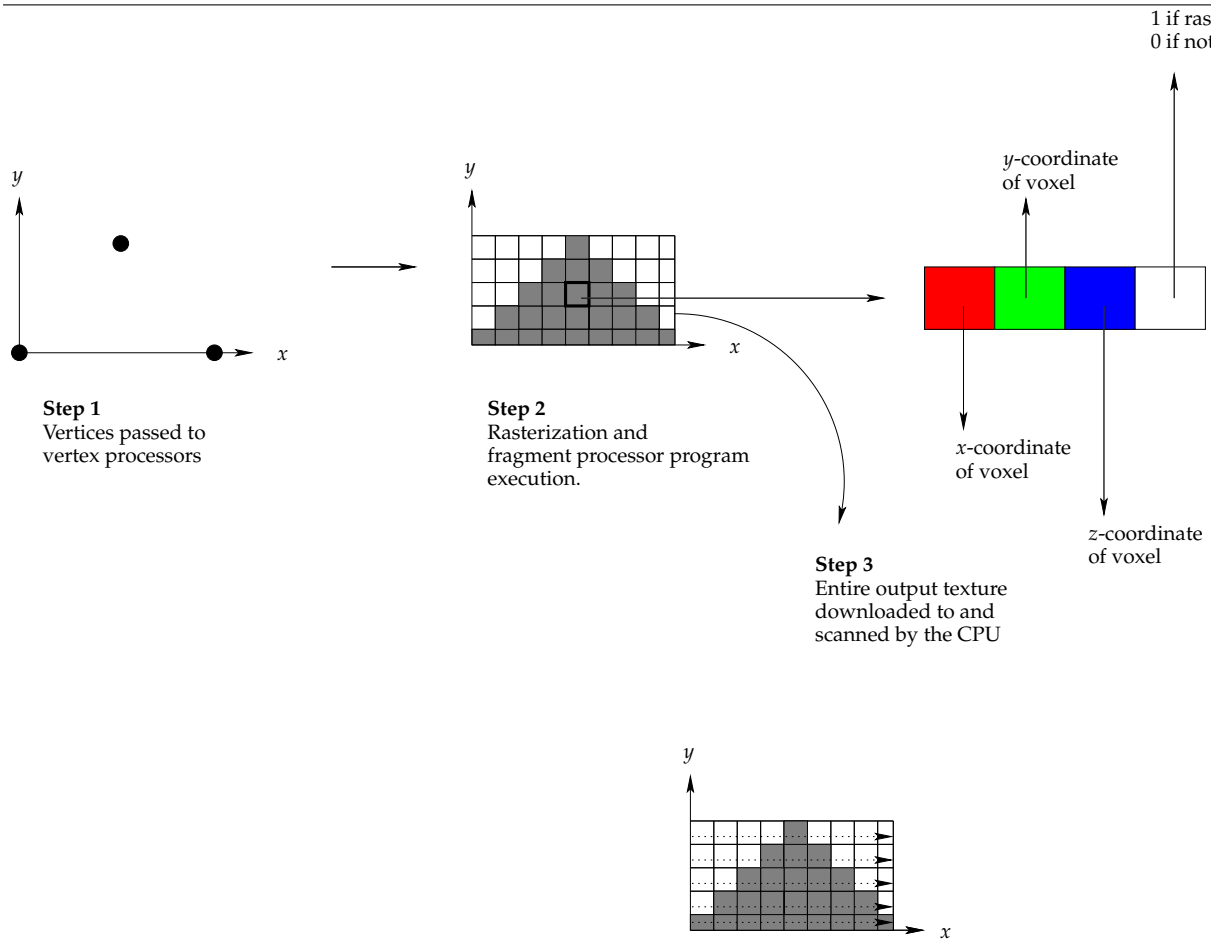


Figure 3.15: Voxelization of Triangles on the GPU

NVIDIA's Cg toolkit for interfacing and programming the GPU[10]. When doing the matrix multiplication, we use 32-bit floating point numbers, as 16-bit numbers do not offer nearly enough precision. This is unfortunate, since 16-bit arithmetic is quicker on the GPU than 32-bit arithmetic [26].

The GPU and CPU versions do not in all cases produce precisely the same voxels, due to differences in the floating-point implementations, but the deviations are small. Listing E.1 below shows the actual GPU fragment processor program. The vertex processors are not utilized — the vertex programs we use only pass vertices further down the pipeline. The uniform parameters in the program are set by the CPU and are constant for each program execution. The rotation matrix is obtained from the inverse combined quaternion, as described above. Consult the source code documentation in Appendix C on page 119 and the source code for further information.

Notice that using the GPU does *not* reduce the asymptotic running time of the algorithm. In Step 3, the CPU still has to iterate over the bounding box of the triangle and put each found voxel into an array which contains all the coordinates of the triangle. The GPU only helps with rasterizing the 2D triangle and multiplying each coordinate vector with the inverse

[10]Recent libraries for general purpose GPU programming, such as NVIDIA's CUDA library, are unnecessary since our problem fits the GPU hardware *very* well. In fact, this method would likely be more difficult to create with CUDA.

rotation matrix. Therefore, CPU time must still be spent when using GPU voxelization.

The GPU fragment program is shown in Appendix E.3.4.

3.6.5 Comparison to Kaufman's Algorithm

Our algorithm has two advantages over Kaufman's algorithm:

1. No branches are used in the inner loops.
2. The algorithm can utilize the rasterization hardware on GPUs.

The disadvantages are:

1. Our algorithm uses more floating-point operations than Kaufman's algorithm, since we have to do the inverse rotation for each pixel.
2. Kaufman's algorithm is more general since it also applies to polygons which are not triangles. However, the approach we presented can be extended to other polygon types by triangulating the polygon to be voxelized and treating each triangle independently.
3. Doing inverse rotations on the GPU may lead to rounding errors. The GPU does not implement full IEEE floating-point support. Therefore, certain computed coordinates may be rounded improperly. This is, however, not critical for our purposes, since for pillar gridding purposes, it is unimportant if some voxels are slightly off the actual polygon.

It is difficult to speculate which algorithm will be quickest. Having branch instructions in inner loops is generally not a good idea, but neither is having more computations than necessary — however, when using our algorithm, these computations can be done on the GPU, which is specifically designed to execute the 4×4 matrix-vector multiplications we require. We will benchmark our voxelization algorithm in Chapter 5.

Chapter 4

Framework Design Part II: Load-Balancing Strategies

So far, our discussion has assumed that the responsible node for a polygon is always equal to the centroid node. This is not necessarily an optimal strategy. Consider Figure 4.1, which shows how load can be unbalanced if we do so.

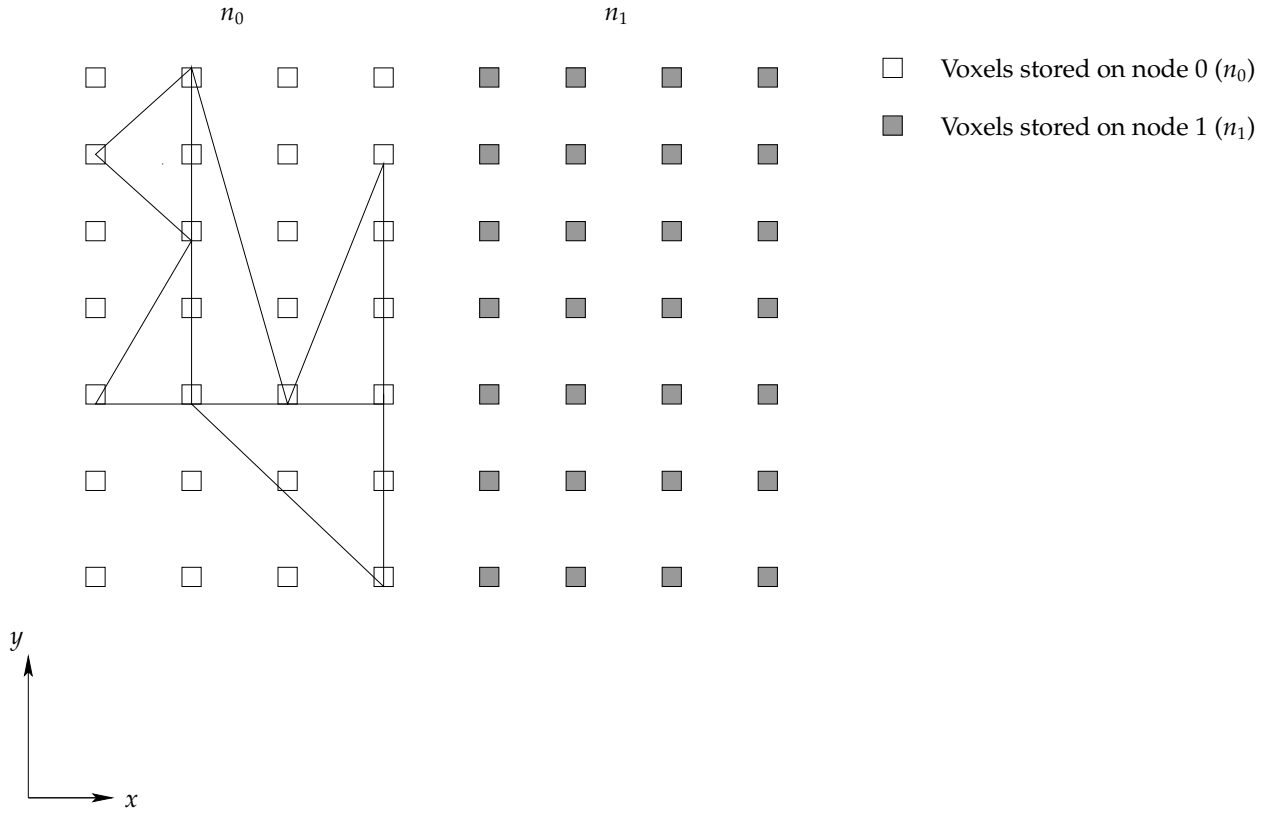


Figure 4.1: Unbalanced polygon mesh. Node n_0 will have to do much work, while node n_1 has no work.

Consider what happens when EXTRACT-ALL is run with the situation above, and suppose that we want to do some computation over the voxels found by this routine. If the responsible node is always the centroid node, node n_0 will have to do all the work while node n_1

has no work to do!

Therefore, we need *load-balancing strategies*. Since our polygon model is global, the load-balancing algorithms we present here *do not require any communication between the nodes*. The reason for this choice is that in general, computation is cheaper than communication. Furthermore, with our strategy, each node knows precisely which other nodes are responsible for all the faces in all meshes.

We introduce notation used in discussing load-balancing in Section 4.1. Next, we discuss when load-balancing should be done in Section 4.2. Subsequently, we present three strategies for load-balancing in Section 4.3, which are compared in Section 4.4.

4.1 Notation

We have previously used $A(F^i)$ to denote the area of the set of all faces whose responsible node is i. Let

$$\overline{A} = \frac{1}{|N|} A\left(\bigcup_{i \in N} F^i\right) \tag{4.1}$$

that is, $\overline{A}$ is the total area of all faces on all nodes (in all meshes), divided by the number of nodes. Now let

$$L(i) = A(F^i) - \overline{A} \tag{4.2}$$

for $i \in N$. Then:

- If $L(i) < 0$ for some node i, node i has available capacity since it has assigned a total area that is less than the average area of all nodes. The optimal distribution is that all nodes deal with precisely the same area $\overline{A}$. This is optimal since the voxelization algorithms are asymptotic in the total area of the faces stored on a node, and so are any subsequent computations. We say that node i is *underloaded*.
- If $L(i) = 0$, node i is *perfectly balanced*.
- If $L(i) > 0$, node i is *overloaded*.

The goal for our load-balancing algorithms is to make $L(i)$ as close to zero as possible for all $i \in N$. This might not be possible, since we can only move entire faces. Additionally, this requirement must be balanced with the fact that if we move a polygon face from the centroid node to another node, more voxels need to be transferred when executing EXTRACT-ALL since the other node does not store any voxels of the polygon. Preferably, faces should be sent to a node whose coordinate space is close to the centroid node of the face, in order to minimize the number of voxels that need to be transferred.

Additionally, it is not certain that load-balancing will pay off at all. It is a general principle in load-balancing that if a node is only slightly overloaded, the cost of data migration to other nodes may excess any savings we may obtain from the migration. The threshold which must be used is application-specific [36]. We therefore adopt the convention to only consider a node $i \in N$ overloaded if

$$L(i) > \delta \tag{4.3}$$

where $\delta \geq 0$ is a threshold parameter. The optimal choice for δ will generally depend on the number of polygons, average polygon sizes, and which (if any) computations we want to do over the polygon's face after executing the EXTRACT-ALL operation.

We now consider when load-balancing should be done.

4.2 When to Load-Balance

One question is when the load-balancing algorithms should be done. For example, we could load-balance whenever adding a polygon to a mesh; that is, in the ADD-FACE operation. However, load-balancing itself adds overhead.

By choosing polygon models to be global, we deferred all communication to the EXTRACT-ALL operation in order to keep the other operations quick. The communication that takes place at the EXTRACT-ALL operation will likely take more time than the load-balancing operation. Consequently, doing load-balancing when doing extraction of all faces in all meshes will not severely affect the performance of that operation. But if we rather do load-balancing in other operations, the operations may suddenly take several times longer to execute. We have therefore chosen to defer load-balancing to the EXTRACT-ALL operation.

Any of the caching and transfer strategies of the previous chapter is compatible with any of the load-balancing strategies. The total time T_{Comp} for an EXTRACT-ALL computation operation is thus

$$T_{\mathsf{Comp}} = T_{\mathsf{LoadBalance}} + T_{\mathsf{CacheTransfer}} \tag{4.4}$$

where $T_{\mathsf{LoadBalance}}$ is the time of the load-balancing strategy and $T_{\mathsf{CacheTransfer}}$ is the time of the caching and transfer strategies, in addition to the computation done over the voxelized surfaces (recall that we included the expression T_{other} in the expressions for $T_{\mathsf{CacheTransfer}}$ in the previous chapter to represent the computation done after the data transfer has taken place).

Expressions for $T_{\mathsf{CacheTransfer}}$ for the Minimum Communication, Block Transfer or Bounding Volume strategies were thus given in the previous chapter. We give expressions for $T_{\mathsf{LoadBalance}}$ for different load-balancing strategies below.

Note that load-balancing algorithms affect $T_{\mathsf{CacheTransfer}}$, both in terms of the computation time (T_{other} in the previous chapter) and in terms of the communication terms in the expressions.

4.3 Three Load-Balancing Strategies

We introduce three load-balancing strategies in Sections 4.3.1, 4.3.2 and 4.3.3 below.

4.3.1 Strategy 1: Global Load-Balancing Strategy

Description

The first strategy exclusively tries to get $L(i)$ as close to zero as possible. The algorithm is shown below in the procedure GLOBAL-LOADBALANCE. Throughout this chapter, we assume that each face F has an attribute $area[F]$ which contains the face area (computed by a simple cross product when the triangle is added).

GLOBAL-LOADBALANCE

▷ We assume that:
• $totalarea[i]$ contains the
total area of all polygons whose responsible node is i.
• $\overline{A}$ and F^i are computed during add/remove/move
operations. F^i is the set of faces with responsible node i
(which, initially, is the centroid node).

```
1  for i ∈ N
2      do if totalarea[i] ≤ Ā + δ
3           then continue
4         targetarea ← totalarea[i] − Ā
5           do sortednodes ← sorted array of nodes j ∈ N − {i}
               in order of increasing totalarea[j]
6              for j = 0 to |N| − 1
7                  do k ← sortednodes[j]
8                     maxtarget ← Ā − totalarea[k]
9                     if maxtarget ≤ 0 or targetarea ≤ 0
10                      then break
11                    Find subset S ⊂ F^i with A(S) ≤ min(targetarea, maxtarget)
12                    Assign responsible node of faces in S to be k
13                    targetarea ← targetarea − A(S)
14                    totalarea[i] ← totalarea[i] − A(S)
15                    totalarea[k] ← totalarea[k] + A(S)
16                    F^i ← F^i − S
17                    F^k ← F^k ∪ S
```

The algorithm is executed on all nodes simultaneously, and works as follows. Line 1 iterates through all nodes. Line 2 checks if the current node investigated is overloaded — that is, if $L(i) > \delta$. If not, there is nothing to do for this node, so we continue to the next node.

If the node is overloaded, line 4 computes the *target area*, which is the surplus capacity the node has. Thus, in this case, $L(i) > \delta$ and the *targetarea* variable is assigned the value of $L(i)$. Line 5 sorts the nodes in order of increasing load. Thus the first entry in the *sortednodes* array has the least area stored on it, the second entry has the second least area stored, and so on.

Next, in line 6 and the rest of the code, we iterate over nodes $k \in sortednodes$ in an order of increasing load, and assign as many as possible faces to each node with available capacity. Nodes with the least load are assigned to k first, next the node with the second least load is assigned, and so on. *maxtarget* is the available capacity on node k. Line 9 checks if we are done with node i. This happens when *targetarea*, which is decremented every time faces are moved from node i to some underloaded node, reaches zero, or when the current node in

sortedarray is overloaded, in which case all remaining nodes in *sortedarray* are overloaded. In that case, we continue to the next node.

Lines 11–17 choose a subset of the nodes stored on i which are the minimum of the target area and the surplus capacity on node k. This is done since we must make sure that node k does not get overloaded, and at the same time, we do not wish to assign to node k more polygons than necessary in order for the current node to become balanced. In our implementation, we always try to find as large polygons as possible, so that the voxel cache of the target node does not become fragmented by storing many voxels of different polygons at different locations. The set of faces is assigned to node k in lines 16–17, while the *totalarea* and *targetarea* variables are updated according to the assignment.

The global load-balancing approach is illustrated in Figure 4.2. A polygon whose centroid node is an overloaded node moves a triangle to the most underloaded node that can be found. Notice that the coordinates of the triangle are not moved. The triangle still intersects voxels in the original node's coordinate space. The only thing changed is the responsible node. If the red node is still overloaded, more faces are transferred to the green node. If the green node does not have capacity available, we look for the next node with the most available capacity.

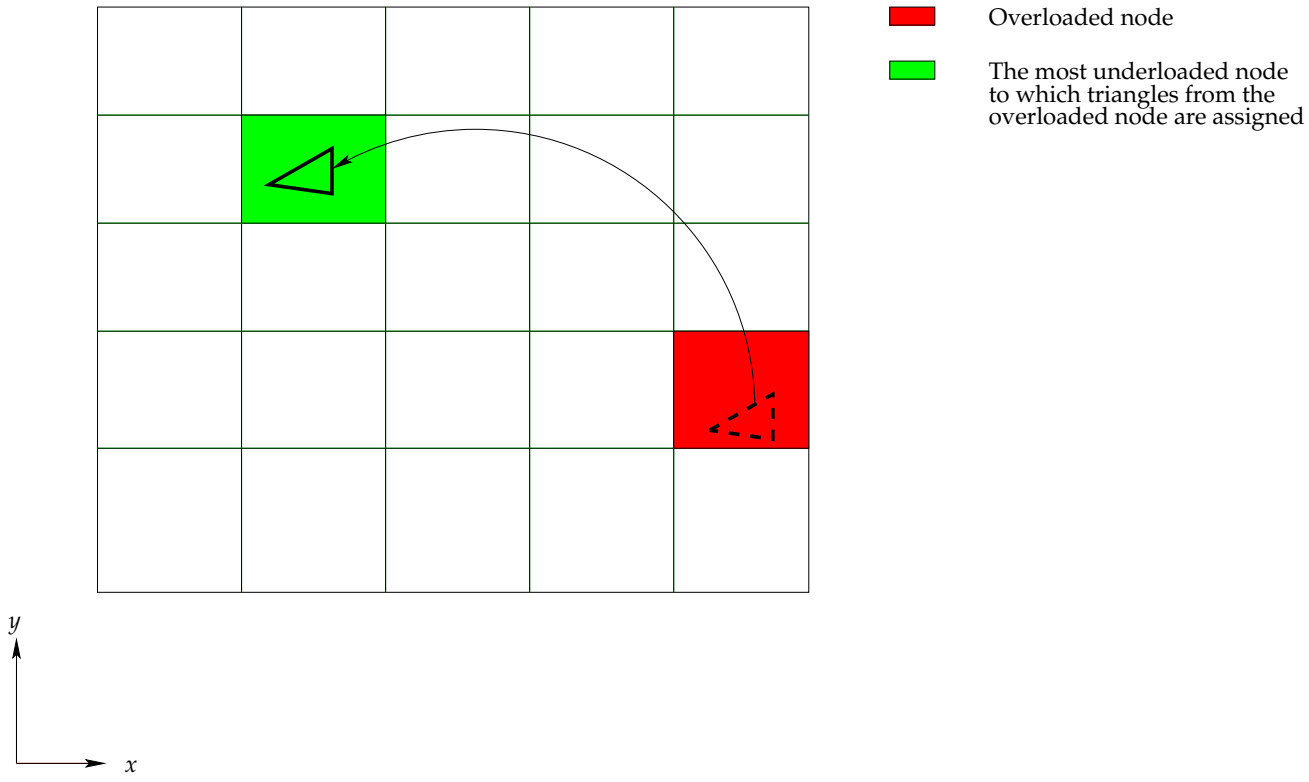


Figure 4.2: Global load-balancing strategy.

Analysis

For now, assume that $\delta = 0$. The best-case running time occurs when all nodes are perfectly balanced. In that case, the loop in line 1 takes $O(1)$ time, so the time complexity is given by

$$T_{\mathsf{GlobalLB,Best}} = k_1|N| \tag{4.5}$$

where $|N|$ is the total number of faces in all meshes and k_1 and k_1' are constants.

In the worst case, $|N| - 1$ nodes are overloaded while one node is heavily underloaded. This leads to the sorting in line line 5 — taking $O(|N| \log |N|)$ time in general — being done for $|N| - 1$ nodes. Furthermore, line 11 will take $O(|F|)$ time in total over all iterations (since each node i only iterates through $|F^i|$ faces, and not $|F|$ faces — summed up over all nodes, however, this totals to $|F|$ faces in the worst case). Therefore, the worst-case running time is:

$$T_{\mathsf{GlobalLB,Worst}} = k_3(|N| - 1)(|N| \log |N| + k_3') + k_3''|F| + k_3''' \tag{4.6}$$

where k_3, k_3', k_3'', k_3''' are constants. The last constant represents the time the loop in lines 1–3 take for the single underloaded node. Note that the loop in line 6 will in this case be executed only once.

Due to the $O(|N|^2 \log |N|)$ term in Equation 4.6, this load-balancing algorithm might not scale well if we use a very large number of nodes (say, $|N|$ is in the thousands). However, the running time can reduced by using a heap [22] instead of sorting the array each time. A heap structure is suitable for priority queues. The heap structure can be constructed in linear time, and the node with the minimum area can be found also in $O(\log |N|)$ time. When the areas are updated in line 13–14, the heap structure can also be updated in $O(\log |N|)$ time. Thus we can reduce the $|N| \log |N|$ factor to $|N| + \log |N|$. However, in our case, $|N|$ is fairly small, so the heap structure adds unnecessary overhead. We therefore use the sorting technique in our implementations.

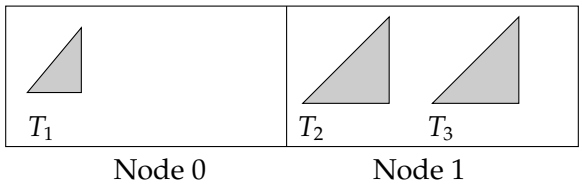
Note that as δ increases, the running time will decrease, since fewer nodes will be considered to be overloaded. Thus, with increasing δ, the probability of hitting the worst-case running time is reduced.

Discussion

Before discussing this algorithm in particular, note that the general load-balancing problem is NP-complete. Fundamentally, our problem is to choose $|N|$ subsets from the set of all faces such that the sum of the areas of the faces in all subsets is as close as possible to $\overline{A}$. For $|N| = 1$, this reduces to the subset-sum problem, which is NP-complete [22]. Given an algorithm to solve the subset-sum problem, this problem can be solved for $|N| > 1$. Therefore our problem is NP-complete.

The global load-balancing algorithm is thus only a greedy approach to the NP-complete problem: In each step, we only choose what seems to be the optimal solution at that specific point, but that may not be the globally optimal solution. A simple example is shown in Figure 4.3 on the facing page. In the figure, triangle T_1 and has node 0, but T_2 and T_3 have centroid node 1. Assume that $A(T_1) = 1$, $A(T_2) = A(T_3) = 2$. Then, when executing the global load-balancing algorithm, no polygons are transferred from node 1 to 0 since neither T_2 nor T_3 can be migrated, since node 0 has only 1.5 in available capacity. This is not optimal. Better work distribution would be obtained if one of T_2 and T_3 was migrated to node 0.

However, note that *work distribution is not everything*. The time it takes to migrate polygons to other nodes also comes into play, which is why the next two strategies may outperform the



$$A(T_1) = 1, A(T_2) = A(T_3) = 2, \overline{A} = 2.5$$

Figure 4.3: Non-optimality of the global load-balancing strategy.

global strategy even though it is a better approximation to the solution of the NP-complete problem.

Advantages of the global load-balancing approach are:

1. *Work distribution will be good.* We always assign polygons to nodes having the most available capacity. Therefore, in general, $L(i)$ should approach 0 for all nodes $i \in N$, even though the work distribution is not optimal.
2. *The best-fitting polygon can be chosen so that the cache on each node does not become severely fragmented*. In line 11 of the pseudo code, we can choose the set of faces to be the largest polygons not exceeding the capacity of the destination node or the *targetarea* variable. For voxel caching reasons, it is generally better to assign large polygons to other nodes since the fragmentation of the voxel cache on the destination node becomes less than if assigning many small faces.

Disadvantages are:

1. *Large data transfers may be required.* When the centroid node of a triangle is overloaded, we assign the triangle to a node whose coordinate space might be far away from the centroid node. Thus, when EXTRACT-ALL is executed, the caching and transfer strategies in the previous section may need to transfer *many* voxels, since the triangle may intersect *no* voxels in the new node's coordinate space.
2. *Slow running time if using very many processors or many faces*. As we mentioned, if $|N|$ is large, $|N|^2 \log |N|$ becomes very large. Therefore, when using thousands of processors, this algorithm may run slowly. Additionally, if $|F|$ is large, the running time will also be slow in the worst-case.

We now consider another strategy which can be viewed as the opposite extreme to the global strategy.

4.3.2 Strategy 2: Local Load-Balancing Strategy

Description

The main problem with the global load-balancing strategy was that triangles may be moved far away from the centroid node. The global strategy does not consider whether any of the voxels of the polygon are stored on the node the polygon is migrated to. In return,

we get relatively good work distribution between the nodes. We now introduce the *local* load-balancing strategy, which is shown in the LOCAL-LOADBALANCE procedure below.

LOCAL-LOADBALANCE

```
1  areas[0..|N|−1] ← 0
2  Ā_temp ← 0
3  for all faces F
4       do if areas[centroidnode[F]] ≤ Ā + δ
5            then responsiblenode[F] ← centroidnode[F]
6            else minnode ← centroidnode[F]
7                 minarea ← areas[minnode]
8                 for all nodes i where C_i intersects the bounding volume of F
9                      do if areas[i] < minarea
10                          then minnode ← i, minarea ← areas[i]
11                responsiblenode[F] ← minnode
12           areas[responsiblenode[F]] ← areas[responsiblenode[F]] + area[F]
13           Ā_temp ← Ā_temp + area[F]/|N|
```

This algorithm works as follows. The *areas* and $\overline{A}_{\text{temp}}$ are temporary variables which hold $\overline{A}$ and the areas stored on each node. Initially, we pretend that each node has 0 area stored and the average area is 0. Next, in line 3, we iterate over all faces. Line 4 checks if the centroid node of the face is underloaded — and if so, assigns the centroid node to be the responsible node for the face. In the opposite case, we detect *all nodes i whose coordinate spaces C_i intersect the bounding volume of the face*, and then choose the node which has the lowest load to be responsible for the face in lines 6–11.

Note that this method uses a slightly different approach than the global load-balancing strategy: In that strategy, we consider each node at a time, while in this strategy, we consider each face at a time.

Assuming $\delta = 0$, in the best case, all faces intersect the coordinate spaces of the centroid node. Thus the best case is

$$T_{\text{LocalLB,Best}} = k_1|F| + k_1' \tag{4.7}$$

for constants k_1, k_1', which is slightly worse than the global load-balancing algorithm since $|F|$ is usually larger than $|N|$.

In the worst case, all faces intersect all coordinate spaces, so the running time is

$$T_{\text{LocalLB,Worst}} = k_3|F||N| + k_3' \tag{4.8}$$

Note that the worst-case is *very* improbable. On average, the running time is likely closer to that of Equation 4.7.

As with the global load-balancing strategy, notice that as δ increases, the running time will decrease since the **if** statement on line 4 will more often become true.

The local load-balancing algorithm is illustrated in Figure 4.4 on the next page. In the figure, a polygon whose centroid node is overloaded is considered. Of those nodes that have a coordinate space which intersects the face, the node with the lowest load (marked with

green) is chosen as the responsible node for the face. Note that the node which is assigned to be the responsible node is not necessarily underloaded.

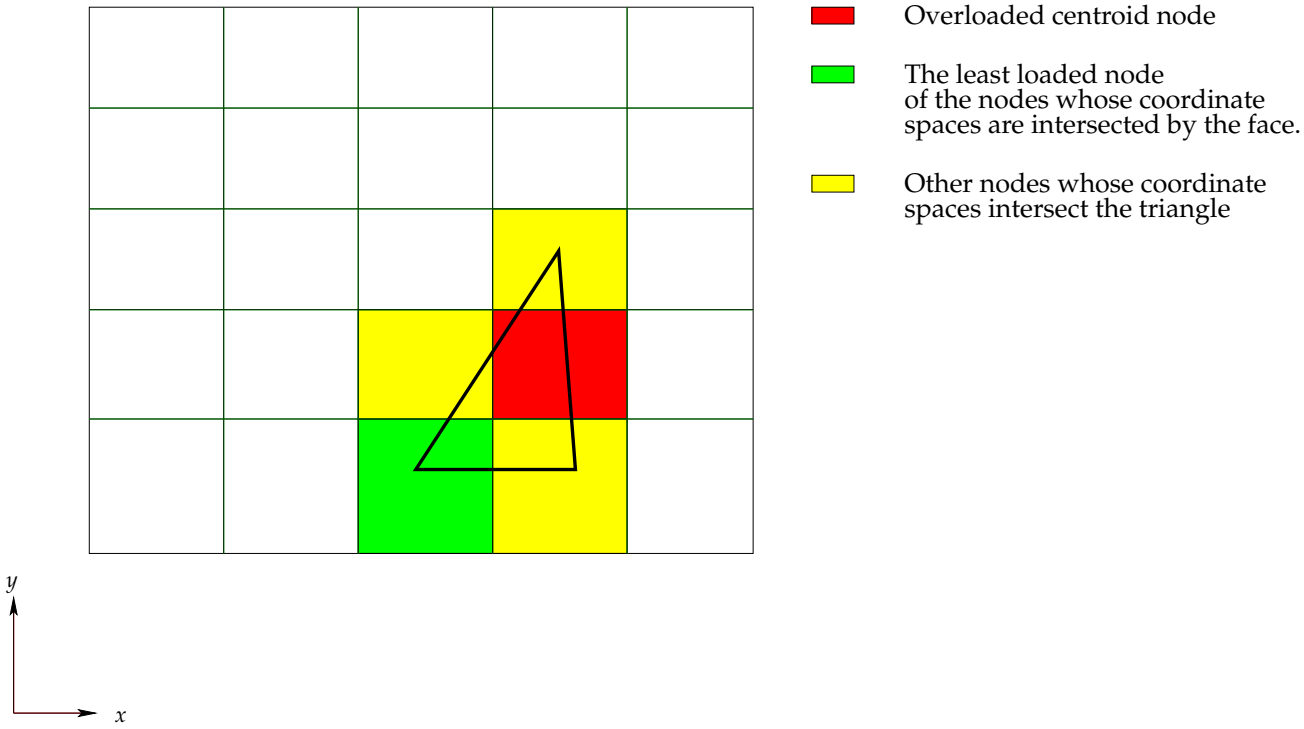


Figure 4.4: The local load-balancing strategy.

Advantages of this strategy are:

1. *The responsible node to which a triangle is assigned is always guaranteed to have some voxels intersecting the triangle.* This will lead to less communication between the nodes than in the other strategies, where no such guarantee is present.
2. *Scales better than the global strategy with respect to an increasing number of nodes*. The algorithm is in the worst case linear in $|N|$, unlike the global strategy, which is $O(|N|^2 \log |N|)$ in the worst case. Therefore, increasing the number of nodes should have less impact on the running time of the algorithm.

Disadvantages are:

1. *Work distribution can be far from optimal.* Consider Figure 4.4 again. The local strategy is restricted in the nodes it can choose. Of the nodes marked with color in the figure, all may have $L(i) > 0$; that is, all nodes may be overloaded. The local strategy chooses only the node that is the *least* loaded of a set of nodes, but that node might be overloaded compared to other nodes whose coordinate spaces do not intersect the triangle.
2. *Depends on many polygons crossing borders*. If all polygons are small, if there are few polygons and if the coordinate space of each node is very large, almost no polygons will cross any borders and we are left with no load-balancing at all.
3. *Could potentially fragment the voxel cache*. Since we consider each face by itself, there is no guarantee in this approach that we move only fairly large polygons. There is a risk

that the voxel cache of neighboring nodes becomes fragmented if the local strategy assigns many small polygons to neighboring nodes instead of large polygons. This happens to a lesser extent in the global strategy.

4. *Does not scale well when increasing the number of faces in the worst case.* In the worst case, the running time is $O(|N||F|)$. If increasing the number of faces significantly, the algorithm will become slow. The global strategy scales better as the number of faces increase.

4.3.3 Strategy 3: Manhattan Distance Load-Balancing Strategy

Description

The global and local load-balancing strategies may be viewed as two extremes. The global strategy tries to get the work distribution to be optimal, and largely ignores the extra communication added. The local strategy does the opposite. Can we find a compromise?

One possibility, we suggest, is based on the Manhattan distance between coordinate spaces, viewed along the z direction. The *Manhattan distance* $M(p,q)$ between two discrete points $p = (x_0, y_0)$ and $q = (x_1, y_1)$ is [18]

$$M(p,q) = |x_1 - x_0| + |y_1 - y_0| \tag{4.9}$$

Figure 4.5 shows the Manhattan distances between a node and the other nodes, viewed along the z-axis.

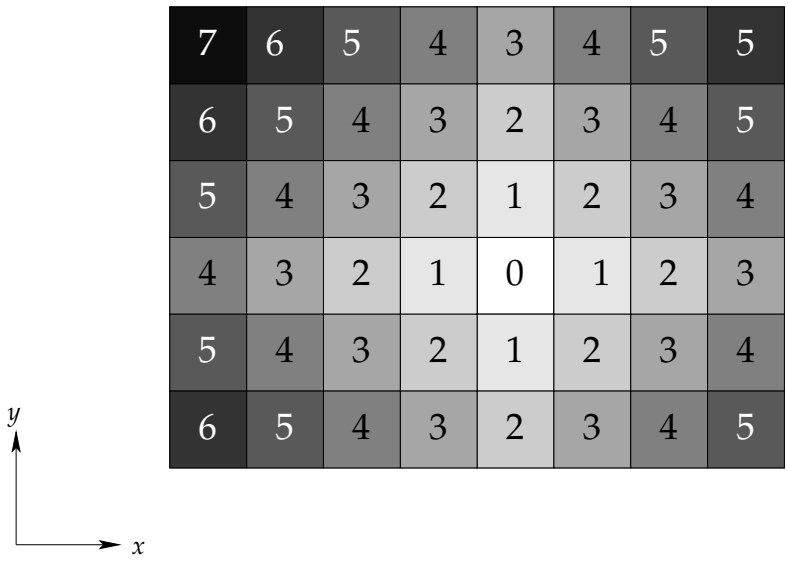


Figure 4.5: The Manhattan distance load-balancing strategy.

Suppose that we initially compute the manhattan distances from each node to all other nodes[1]. Then, when executing the EXTRACT-ALL operation, if a node is overloaded, we

[1]This can be done trivially in $\Theta(|N|^2)$ time. How we do this is not significant, since it is only done once when the program is started.

search for underloaded nodes *in the order of increasing Manhattan distance*. However, in order to keep the algorithm from moving polygons to nodes with a relatively large Manhattan distance from the current node, we define a threshold τ such that, when considering an overloaded node i, only nodes j such that $M(i,j) \leq \tau$ are considered. The algorithm is shown in the MANHATTAN-LOADBALANCE procedure below.

MANHATTAN-LOADBALANCE

▷ Assume that:
• A pre-computed table *manhattanmaps*[i][j]
contains all nodes with a Manhattan distance j from i up to and including τ
• *totalarea*[i] contains the
total area of all polygons whose centroid node is i.
• $\overline{A}$ and F^i are computed during add/remove/move
operations.

```
1  for i ∈ N
2      do if totalarea[i] ≤ Ā + δ
3           then continue
4           else targetarea ← totalarea[i] − (Ā + δ)
5         for j ← 1 to size[manhattanmaps[i]]
6             do for k ∈ manhattanmaps[i][j] until targetarea ≤ 0
7                    do if totalarea[k] < Ā
8                         then maxtarget ← Ā + δ − totalarea[k]
9                              Find subset S ⊂ F^i with A(S) ≤ min(targetarea, maxtarget)
10                             Assign responsible node of faces in S to be k
11                             targetarea ← targetarea − A(S)
12                             totalarea[i] ← totalarea[i] − A(S)
13                             totalarea[k] ← totalarea[k] + A(S)
14                             F^i ← F^i − S
15                             F^k ← F^k ∪ S
```

This algorithm is very similar to the global load-balancing algorithm. The difference is primarily that nodes in the inner loop are evaluated in the order of increasing Manhattan distance, instead of in the order of increasing total area. Additionally, notice that the *maxtarget* and *targetarea* variables are computed differently from the global load-balancing algorithm: If the underutilized capacity of the underloaded node i is less than the excess capacity stored on the overloaded node j, i.e. $|L(i)| < L(j)$, then, in the global load-balancing algorithm, polygons will be migrated to the underloaded node up until $L(i)$ reaches 0. However, the Manhattan strategy fills nodes up until they reach $L(i) = \delta$. This is done since it is easier to find nodes with load $L(i) < \delta$ than nodes with load $L(i) < 0$ in the neighborhood of the overloaded node (if $\delta > 0$).

We briefly explain the algorithm line by line. In line 1, we iterate over all nodes. Lines 2–4 check if the current node is underloaded. If it is, continue. If not, compute a target area which gives how large area must be moved to other nodes.

Lines 5–6 iterate over nodes k in the order of increasing Manhattan distance, up to and including the threshold τ. First, k is assigned nodes with Manhattan distance 1, then nodes with distance 2, and so on, until *targetarea* has become ≤ 0, at which case node i is not overloaded any more and we can continue with the next node in line 1.

For each node k, we check if the node has available capacity in line 7. If it does, we choose a

subset from node i's faces which have an area which is the minimum of *targetarea* — the area i needs to offload — ad *maxtarget* — which is the capacity node k has available. As with the global load-balancing strategy, we choose as large as possible faces so that the voxel cache on the destination node does not become too fragmented. These faces are then assigned to node k and the *targetarea* and *totalarea* variables are updated accordingly in lines 9–15.

Suppose, for example, that the node with Manhattan distance zero in Figure 4.5 is overloaded. The node will attempt to offload faces to nodes in the order of increasing Manhattan distance, but only evaluates nodes with a Manhattan distance up to the threshold τ. Since nodes close to a node i are more likely to either (1) have a coordinate space which intersects faces whose centroid node is i, or (2) already have some voxels from i which have been cached, this should be a good idea.

Analysis

Assuming $\tau = \infty$ and $\delta = 0$, the best-case running time of this algorithm occurs when all nodes are perfectly (or approximately perfectly) balanced. Thus

$$T_{\mathsf{ManhattanLB,Best}} = k_1|N| + k_1' \tag{4.10}$$

for constants k_1, k_1'.

In the worst case, one node is highly underloaded while all other nodes are overloaded. We therefore risk iterating in line 5–6 $|N|$ times to find the underloaded node, and line 9 will in total take approximately $|F|$ times since it is executed for almost all nodes. An upper bound is therefore

$$T_{\mathsf{ManhattanLB,Average}} = k_3|N|^2 + k_3'|F| + k_3'' \tag{4.11}$$

for constants k_3, k_3', k_3''.

By decreasing τ, the running time of the algorithm will generally become smaller since the loop in line 5 will execute fewer times. Also, as δ increases, the running time decreases since it is easier to find underloaded nodes.

Discussion

The advantages of the Manhattan load-balancing method is

1. *Compromise between the advantages of the global and local strategies.* The work distribution is balanced with the requirement to keep polygons close to the centroid node. Therefore, this strategy has a combination of the advantages of the global and local strategies, with the disadvantages to a lesser extent: It does not have the disadvantage of the global strategy of potentially moving faces to nodes which have coordinate spaces far away from the faces' centroid nodes, and does not have the disadvantage of the local strategy that potentially produces a less optimal work distribution. Also, the advantage of the global strategy of choosing to assign large polygons instead of many small polygons is retained.

Disadvantages include

1. *Scales poorly with an increasing number of nodes in the worst case.* In the worst case, the algorithm time is $\Theta(|N|^2 + |F|)$. This is, however, still better than the global load-balancing strategy, even if implemented using heaps.
2. *It is not certain that polygons will be sent to nodes close to the centroid node.* In the worst case, polygons are sent just as far away as with the global strategy, if τ is large.
3. *Works best when there are a few severely overloaded nodes.* The strategy works best when only a few nodes are overloaded, especially if τ is large. If many nodes are overloaded, the first nodes considered by the algorithm might assign polygons with a large Manhattan distance but close to other overloaded nodes considered later in the algorithm. When these overloaded nodes are considered, they will be unable to transfer polygons to the underloaded nodes with a low Manhattan distance from the nodes.
4. *Large amounts of data transfer may still be required.* Even if we manage to solve the problem mentioned above, it is not certain that assigning polygons to neighboring nodes is any better than the global strategy, since it is not certain that these nodes have voxels cached from neighboring nodes. As we mentioned above, this algorithm does not — as the local strategy does — send polygons exclusively to nodes which already have some voxels of the polygon.

4.4 Comparison of Strategies

If the computation done over the voxels is sufficiently complicated, computation time will dominate the communication time, which always is linear in the number of voxels sent (although the coefficients to this linear equation may be relatively large). Therefore the global and Manhattan strategies with a large threshold, which give better work distribution than the local strategy, will at some point win over the local strategy as computation time increases. The question, however, is whether or not the computation must be *very* complex in order to make up for the increased communication time used by these strategies compared to the local strategy. The local strategy may be superior in most cases since less voxels must be transferred.

In the next chapter, we benchmark the different strategies.

Chapter 5

Benchmarks and Discussion

In this chapter, we consider the performance of our implementations of the methods described in the preceding chapters. First, we give an introduction to the terms speedup and efficiency, which are commonly used metrics for benchmarking parallel programs, in Section 5.1. We will then benchmark the major component of the framework, which is the the voxel caching and load-balancing algorithms, in Section 5.2. The results of the benchmark are then analyzed in Section 5.3. Finally, in Section 5.4, we also include benchmarks of our voxelization algorithm suggested in Section 3.6.

The programs we use to do benchmarking are described in the User's Guide in Appendix B. The source code implementation is in the electronic attachment accompanying this thesis. An overview of the source code of the framework implementation is given in Appendix C.

5.1 Performance Measurements

The principal metric for measuring the parallel performance is the *speedup*. If the quickest serial version of a program takes time T_s and the parallel program takes time T_p, we define the speedup S as [23]

$$S = \frac{T_s}{T_p} \tag{5.1}$$

If $S \geq 1$, the parallel version is quicker than the serial version of the program. With p processes, the ideal goal of a parallel program is to have $T_p = T_s/p$. This happens rarely in practice since parallel programs usually require some sort of communication or synchronization between the processes. In some cases, particularly in machines with large caches, *superlinear* speedup, where $T_p < T_s/p$, can be seen due to the data sets fitting entirely in cache.

Another metric is *parallel efficiency* E, which is defined as [23]

$$E = \frac{S}{p} \tag{5.2}$$

where p is the number of processes. Efficiency can be used to measure the *scalability* of the program. If the efficiency decreases as p increases, the program does not scale well since

the speedup per processor — the efficiency — becomes less. Therefore, less time is spent on doing actual work and more time is spent on communication or synchronization among the processes. An efficiency of one corresponds to linear speedup.

5.2 Caching and Load-Balancing Benchmarks

We now describe the results of the caching and load-balancing benchmarks. This section consists of two parts: Section 5.2.1 describes the test equipment and methodology, Section 5.2.2 shows the results.

5.2.1 Test Equipment and Methodology

Machine

For benchmarks, we will use the Njord supercomputer at NTNU, which has the following specifications:

- 992 IBM Power5+ p575+ CPUs, spread over 62 nodes, linked with proprietary interconnects.
- 1984 GiB total memory
- IBM AIX 5.3 operating system

The Njord machine is far superior to most clusters, particularly with regard to the interconnects. Nodes on Njord are grouped into sets of 16 CPUs with shared memory. Thus, when using the MPI library, communication within a single set of shared-memory nodes — which involves a simple memory-copy operation — is extremely fast, and high-speed interconnects between nodes ensure that communication with other nodes is very also quick.

Communication on standard commodity clusters, however, usually use gigabit Ethernet between all nodes, which is inferior to Njord both in terms of bandwidth and latency. Additionally, Njord has a price tag of 30 million NOK, while the price tag of commodity clusters is much lower. As we mentioned in Chapter 2, one of the incentives behind using clusters is precisely their low price tag. One question is therefore whether or not it would be fair to benchmark using Njord instead of lower-powered clusters.

Benchmarking on Njord, however, has several advantages. First, it allows us to easily and quickly experiment with large data sets and *very* many processors. Second, if a caching or load-balancing strategy performs badly on Njord, the probability of other caching and load-balancing strategies being superior on other clusters is decreased: Although the Njord interconnects are very fast, the arithmetic complexity of our computations is also high. This should to a certain degree reduce the significance of the interconnects. However, it is difficult to generalize our findings to hold for all types of clusters, which vary greatly in processing power and the type of interconnect used (Gigabit Ethernet versus quicker interconnects such as InfiniBand). Therefore, when using the framework in an application, we recommend doing benchmarking on the cluster type where the application will be used. A user's guide for the benchmarking programs is given in Appendix B.

Benchmark Parameters

In testing the framework, there are many parameters that can be varied:

1. The data set size,
2. average polygon size,
3. the number of polygons,
4. the number of meshes,
5. the number of move-extract cycles,
6. how far each vertex is moved in each cycle,
7. caching strategy,
8. load-balancing strategy,
9. the δ value used in the load-balancing strategy,
10. the τ value used in the Manhattan load-balancing strategy,
11. which computation to do,
12. the number of nodes used,
13. whether we use single- or multi-core or GPU voxelization algorithms, and
14. the machines used for testing.

Even when only using two choices for each item on the list, we get 16,384 combinations to try! We have therefore chosen to fix some of the values on the above list, so that the numbers of benchmarks is kept at a reasonable number. We will use the following three benchmarks as a basis for our discussions:

1. **Benchmark 1** consists of a data set of size $100 \times 100 \times 50$, with each point containing a randomly generated 32-bit floating-point number, with average polygon size 1000, 5 move-extract cycles, 5,000 polygons spread over 16 meshes and $\delta = 0$. In each move-extract cycle, vertices are moved an average distance of about 10 voxels. Single-core CPU voxelization algorithms are used[1].
2. **Benchmark 2** is the same as **Benchmark 1**, except the data set size is $200 \times 200 \times 100$ and 10,000 polygons are used.
3. **Benchmark 3** is the same as **Benchmark 1**, except the data set size is $400 \times 400 \times 200$ and 15,000 polygons are used.

The data sets were generated with the `genscript` program, which is described in the User's Guide in Appendix B. Thus, in each test run:

- 16 random meshes with the specified amount of polygons with the specified average polygon size are generated and the CREATE-MESH and ADD-FACE operations are executed.

[1]Even though it might be possible to increase performance by decreasing the number of coarse-grained tasks in exchange for more voxelization threads, as we will see in Section 5.4, only Njord gets a speedup by using multiple voxelization threads. There is no speedup when using Intel architectures, which is the computer architecture where the framework will likely be mostly used. We will therefore focus on other parameters and keep this parameter fixed.

- 5 move-extract cycles on each vertex are done. That is, for all vertices of all polygons, the MOVE-VERTEX operation is done. Next, EXTRACT-ALL is executed, and a computation (described below) is done over the voxelized surfaces of all the polygons. The voxel cache is not flushed between each cycle.
- After the 5 cycles, all polygons are removed using the REMOVE-FACES operation and all meshes are destroyed using the DESTROY-MESH operation.

This entire process is again repeated 2 times with a set of 16 different meshes each time, so that 16 random meshes are generated *twice*. That is, each of the two runs operates on 16 different, random meshes. We have also repeated the entire test twice, so that we do a total of 4 runs, 2 of which are unique.

The numbers we chose are aimed to be typical for the seismological applications we described in Chapter 2.1 on page 7.

We do *two* computations after each EXTRACT-ALL operation (i.e. in each move-extract cycle). The numbers we do the computations on (i.e. the $V(p)$ values) are generated randomly. One computation is for geophysical interpretation purposes, and the other is for efficient compression of the voxels for transmission over a network or storage:

1. The first computation we do is a simple, but commonly used operation: For all faces F, we compute the variance of the voxels intersected by F. In our formal notation, for each face F, we compute

$$\mathsf{Var}(F) = \frac{1}{|V^*(F)| - 1} \sum_{p \in V^*(F)} \left(V(p) - \frac{1}{|V^*(F)|} \sum_{q \in V^*(F)} V(q) \right)^2 \tag{5.3}$$

   ($\mathsf{Var}(F) = 0$ if $|V^*(F)| \leq 1$). The variance is commonly used in seismological applications. In our case, the variance gives an indication as to how the seismological rock structure varies within a polygon. Different variance levels can then be color encoded, and each polygon, along with their color level, can be sent to the client for visualization.
2. We also compute the one-dimensional Lapped Orthogonal Transform (LOT) [56] of all the points of all the faces. Methods based on the LOT are the most efficient (in terms of quality versus data size) for compressing seismic data [57]. In an application, the LOT can be used to compress all voxels intersecting a face, so that transmitting the voxel data to a client, which could proceed to visualize the polygon surface, is efficient. The LOT and algorithms for computing the LOT is discussed extensively in [56], but we have included a brief description in Appendix E.4.

5.2.2 Results

Speedups relative to the single-core CPU algorithm obtained for **Benchmark 1** are shown in Table 5.1 on the facing page. For **Benchmark 2**, speedups are shown in Table 5.2. Finally, **Benchmark 3** speedups are shown in Table 5.3. In the tables, the best caching strategy for a fixed load-balancing strategy is typeset in boldface, while the best combination of caching and load-balancing strategy is typeset in boldface and italics.

Wallclock times for all the benchmarks can be found in Appendix F.1.

CPUs / **Algorithm**	1	2	4	8	16	32	64	128	256
Serial, 1-cpu	1.00	—	—	—	—	—	—	—	—
MC-nolb	—	0.33	1.03	**1.11**	1.51	2.39	**4.20**	**6.23**	**11.06**
BT-nolb	—	0.67	0.81	0.68	0.95	1.36	2.23	3.07	5.29
BV-nolb	—	**0.85**	**1.03**	1.01	**1.54**	**2.48**	1.13	0.61	1.45
MC-globallb	—	0.72	**1.62**	***2.68***	**5.47**	**8.66**	**14.44**	**22.05**	**33.69**
BT-globallb	—	0.69	0.80	1.18	2.49	3.71	5.91	8.29	14.49
BV-globallb	—	**0.85**	1.36	1.92	3.06	5.49	2.74	2.24	4.77
MC-locallb	—	0.82	***1.75***	**2.57**	***5.81***	***10.80***	***19.89***	***28.43***	***36.39***
BT-locallb	—	0.44	0.87	1.14	2.69	4.69	8.14	11.03	15.75
BV-locallb	—	**0.85**	1.36	1.77	3.12	6.28	2.35	3.52	5.52
MC-mhlb-∞	—	0.84	**1.73**	**2.67**	**5.42**	**8.54**	**14.44**	**22.19**	**34.33**
BT-mhlb-∞	—	0.69	0.85	1.16	2.82	4.57	7.06	9.53	15.04
BV-mhlb-∞	—	**0.85**	1.39	1.95	3.06	5.46	2.68	2.21	4.65
MC-mhlb-2	—	0.84	**1.71**	**2.53**	**2.50**	2.76	**4.05**	**6.31**	**11.06**
BT-mhlb-2	—	0.68	0.85	1.17	1.55	1.55	2.15	3.09	5.25
BV-mhlb-2	—	***0.87***	1.46	1.69	2.42	**2.78**	1.45	0.63	1.46

Table 5.1: Speedups obtained in Benchmark 1 relative to single-core CPU algorithm on Njord.

In the tables, the following abbreviations are used:

1. For the caching strategy, we use the following abbreviations:
   - **MC**: The Minimum Communication Strategy
   - **BT**: The Block Transfer Strategy, which block size 2
   - **BV**: The Bounding Volume Strategy
2. For the load-balancing strategy, we use the following abbreviations:
   - **nolb**: No load-balancing
   - **globallb**: Global load-balancing
   - **locallb**: Local load-balancing
   - **mhlb**: Manhattan load-balancing. We will try two variants of the Manhattan load-balancing strategy:
     (a) **mhlb-∞**: Manhattan load-balancing with infinite threshold, i.e. $\tau = \infty$
     (b) **mhlb-2**: Manhattan load-balancing with a threshold of 2, i.e. $\tau = 2$

CPUs / Algorithm	1	2	4	8	16	32	64	128	256
Serial, 1-cpu	1.00	—	—	—	—	—	—	—	—
MC-nolb	—	**0.71**	0.73	**1.04**	**1.36**	**1.43**	**1.94**	**3.32**	**5.21**
BT-nolb	—	0.65	0.68	0.84	1.05	0.87	1.19	1.92	2.88
BV-nolb	—	0.69	**0.76**	0.95	1.20	1.12	1.71	0.66	0.46
MC-globallb	—	**0.30**	**0.84**	***2.18***	***4.63***	**5.38**	***7.17***	**11.69**	**18.09**
BT-globallb	—	0.14	0.40	1.01	1.84	2.48	4.00	6.25	9.08
BV-globallb	—	0.19	0.55	1.09	1.59	2.09	0.80	1.98	1.64
MC-locallb	—	***0.79***	0.87	**0.89**	**1.96**	**2.95**	**6.67**	***14.26***	***24.44***
BT-locallb	—	0.77	***0.87***	0.49	0.99	1.36	3.11	6.17	10.26
BV-locallb	—	0.70	0.72	0.67	1.32	1.79	1.14	0.88	2.16
MC-mhlb-∞	—	**0.30**	**0.80**	**2.18**	**4.53**	***5.53***	**7.11**	**11.72**	**18.24**
BT-mhlb-∞	—	0.14	0.42	1.01	2.23	3.09	4.13	6.31	9.04
BV-mhlb-∞	—	0.19	0.55	1.09	1.66	2.01	0.86	1.93	1.65
MC-mhlb-2	—	**0.31**	**0.85**	**2.00**	**1.77**	**1.53**	**1.94**	**3.30**	**5.21**
BT-mhlb-2	—	0.14	0.41	1.09	1.34	0.92	1.18	1.91	2.82
BV-mhlb-2	—	0.20	0.62	1.06	1.38	1.18	1.27	0.75	2.94

Table 5.2: Speedups obtained in Benchmark 2 relative to single-core CPU algorithm on Njord.

CPUs / Algorithm	1	2	4	8	16	32	64	128	256
Serial, 1-cpu	1.00	—	—	—	—	—	—	—	—
MC-nolb	—	**0.70**	**0.70**	**0.74**	**0.79**	**1.07**	**1.51**	**1.56**	**1.99**
BT-nolb	—	0.67	0.67	0.68	0.70	0.87	1.15	0.97	1.20
BV-nolb	—	0.59	0.59	0.60	0.65	0.76	1.06	0.27	<0.2
MC-globallb	—	**0.27**	**0.62**	***1.53***	**2.43**	**3.56**	**5.14**	***5.93***	**7.21**
BT-globallb	—	0.12	0.25	0.64	1.02	2.54	3.88	3.62	4.30
BV-globallb	—	0.09	0.15	0.35	0.45	0.76	0.28	0.26	0.73
MC-locallb	—	***0.70***	0.71	**0.83**	**0.92**	**0.87**	**2.08**	**3.37**	**7.04**
BT-locallb	—	0.70	***0.71***	0.81	0.91	0.48	1.08	1.56	3.32
BV-locallb	—	0.47	0.48	0.44	0.54	0.50	0.97	0.23	0.29
MC-mhlb-∞	—	**0.27**	**0.60**	**1.50**	***2.45***	***3.58***	***5.18***	**5.82**	***7.24***
BT-mhlb-∞	—	0.12	0.25	0.66	1.25	2.58	3.83	3.63	4.33
BV-mhlb-∞	—	0.09	0.16	0.34	0.44	0.79	0.28	0.26	0.73
MC-mhlb-2	—	**0.27**	**0.61**	**1.31**	**1.01**	**1.19**	**1.52**	**1.60**	**2.01**
MC-mhlb-2	—	0.12	0.25	0.68	0.87	0.95	1.15	0.98	1.21
MH-mhlb-2	—	0.08	0.16	0.36	0.59	0.76	0.71	0.24	<0.2

Table 5.3: Speedups obtained in Benchmark 3 relative to single-core CPU algorithm on Njord.

5.3 Discussion of the Caching and Load-Balancing Benchmarks

We now discuss and analyze the results of the previous section. We will look at four main issues:

1. First, we will discuss some characteristics of the problem our framework solves in Section 5.3.1.
2. Second, we consider the caching strategies in Section 5.3.2.
3. Third, we consider the load-balancing strategies in Section 5.3.3

5.3.1 Discussion 1. Problem Characteristics

To gain an understanding of the numbers obtained, we will first briefly describe one inherent characteristic of our problem which limits the scalability of our caching and load-balancing strategies.

Consider Figure 5.1, which shows the situation in the problem we are considering. Voxels of a polygon not cached previously are transferred to the responsible node. (With the Block Transfer strategy, surrounding blocks are exchanged in addition to the voxels of the polygon, while with Bounding Volume strategy, bounding volumes are exchanged.)

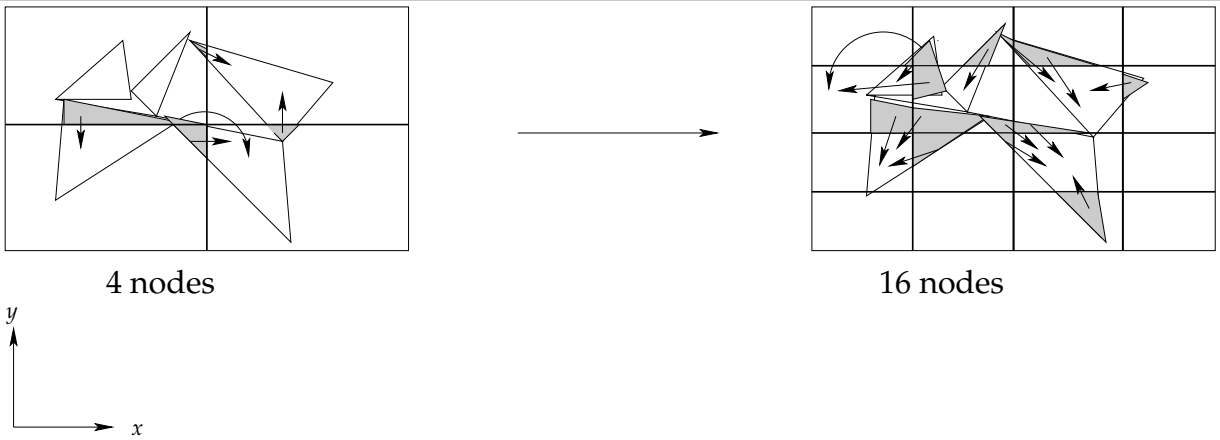


Figure 5.1: Voxel exchanges in the framework with the Minimum Communication strategy (with no load-balancing strategy enabled).

The exchanges we use are not border exchanges, since border values are not necessarily transferred. Dubbing the exchange a *voxel exchange* is more appropriate. And this property limits the scalability of the problem *with respect to increasing the number of polygons*. Observe from Figure 5.1 that:

1. In general, each node, in total, transfers *more* data as the number of nodes increase. This happens since the probability of a polygon intersecting a border increases as the number of nodes increase, and if a polygon intersects a border, voxels must be transferred to the polygon's responsible node.

2. The voxel exchanges may not necessarily be done in parallel. If there is a large polygon intersecting the coordinate spaces of many nodes, all nodes transfer voxels to the responsible node of the polygon. Presumably, the responsible node has only one physical communication channel, and will have receive data serially from all the other nodes containing voxels of the large polygon.

If the number of nodes increase with an increasing number of polygons *and* an increasing the grid size, these problems are reduced, since the average number of borders crossing each polygon will remain constant, and large polygons will intersected a constant number of coordinate spaces. However, for seismic applications, we not interested in increasing the size of the grid, since the grid size is decided by the size of the seismic survey in advance. Instead, we *do* require using many polygons in cases where a data set contains many faults. In fact, it is in such situations that fault surface detection becomes particularly useful. Thus, the number of polygons increases more rapidly than the data set size, so increasing the number of nodes while also increasing the number of polygons is more important than increasing the grid size. However, for other applications, number of nodes is increased at the same rate as the if the grid size can be increased at the same rate as the number of polygons, the problem scales better.

Load-balancing does not solve the above problems, but only reduces its effect since the amount of data each node has to transfer in a voxel exchange is to a greater degree normalized between the nodes. These problems are key to understanding that obtaining linear speedups of the problem we are considering is very difficult unless the grid size is increased at the same rate as the number of polygons. This explains why the efficiency, which for Benchmark 2 is shown in Figure 5.2 on the next page, shows a falling (but gradually flattening) trend as the number of nodes increase for all the caching strategies.

5.3.2 Discussion 2. Performance of the Caching Strategies

We now consider the performance of the caching strategies.

Overall Winner: The Minimum Communication Strategy

In all three benchmarks, the winning strategy (with a few exceptions) is the Minimum Communication strategy. The larger caches in the other strategies apparently do not make up for the increased amount of data that must be transferred. For Benchmark 2, this can clearly be seen from Figure 5.2. Furthermore, although the Bounding Volume strategy does voxelization and communication in parallel, this is not enough to make up for the less communication time used in the Minimum Communication strategy.

The caching strategy of the Block Transfer and Bounding Volume strategies seem to fail for our workload. Considering the theoretical expressions from Chapter 3 again, the $C'_{i,j}$ and $B_{i,j}$ sets of the Block Transfer and Bounding Volume strategies simply become too big compared to the $C_{i,j}$ sets of the Minimum Communication strategy. When vertices are moved with the distance we used in the benchmarks, not enough voxels *used in subsequent cycles* are cached to make up for the increased communication time, and therefore, the voxels transferred in the previous move-extract cycle are wasted. This explains why the Minimum Communication strategy wins.

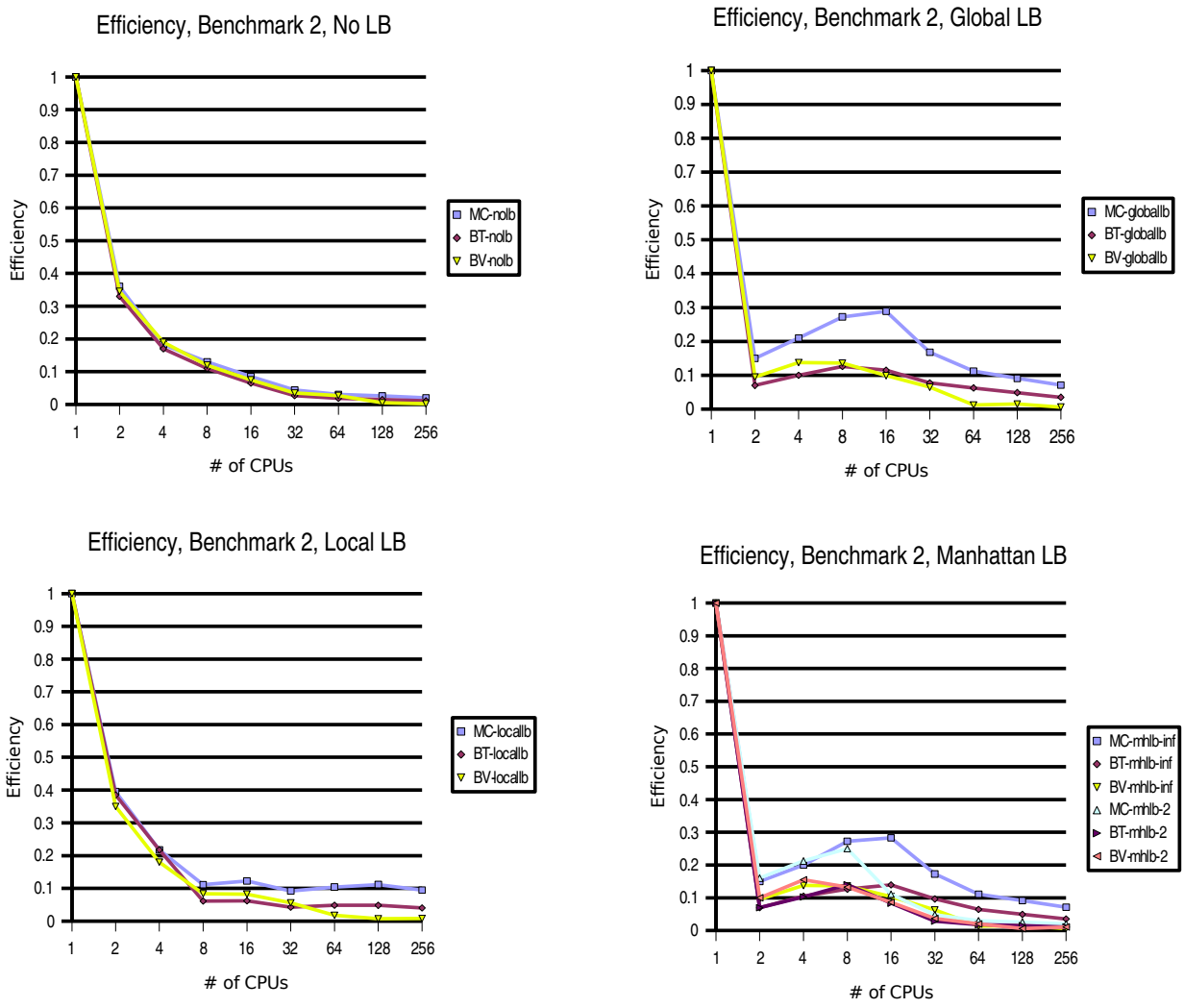


Figure 5.2: Efficiency in Benchmark 2. The horizontal axes show the number of nodes, while the vertical axes show the efficiency as computed in Equation 5.2. (*Top left*) Efficiency when load-balancing is disabled. (*Top right*) Efficiency when using the global load-balancing strategy. (*Bottom left*) Efficiency when using the local load-balancing strategy. (*Bottom right*) Efficiency when using the Manhattan load-balancing strategy.

Further Analysis of the Bounding Volume Strategy

The other strategies do win in certain situations. In Benchmark 1 (Table 5.1 on page 83), when load-balancing is disabled, the Bounding Volume strategy performs better than the other strategies when using 2, 4, 16 and 32 nodes. The reason for the Bounding Volume strategy not winning when using more nodes is related to our discussion in the previous section (see Figure 5.1 on page 85): With more nodes, more communication occurs. This increased communication affects the Bounding Volume strategy more significantly than the

other strategies: The Bounding Volume strategy transfers *volumes* which are usually[2] much larger than the *planes* the other strategies transfer[3].

Without load-balancing, there are no migration costs. Therefore, less communication occurs regardless of the caching strategy used, but *in particular* with the Bounding Volume strategy. Without load-balancing, fewer Bounding Volume are transferred between nodes (since no polygons are migrated to other nodes), and the communication time in this strategy is reduced significantly. Combined with the fact that voxelization and communication are done in parallel, it is no surprise that it wins in certain cases *when load-balancing is disabled*. As can be seen from Table 5.1, when load-balancing is enabled, the Bounding Volume strategy has more problems keeping up with the other strategies.

Further Analysis of the Block Transfer Strategy

The Block Transfer fails to beat the Minimum Communication strategy since it does more communication and the fails to cache enough voxels so that time is saved in subsequent move-extract cycles. However, this strategy is in many cases quicker than the Bounding Volume strategy. This is a consequence of the Block Transfer strategy transferring less data than the Bounding Volume strategy. For example, in Benchmarks 1 and 2, Bounding Volume is superior when using few nodes and either no load-balancing or the Manhattan load-balancing with threshold 2. Otherwise, the Block Transfer strategy is generally quicker.

Larger block sizes could be used in the Block Transfer strategy to try to increase the cache efficiency. However, it is unlikely that larger block sizes will have any significance, since the communication requirements increase very rapidly — in fact, if the block size is k, the communication requirements grow as k^3. For a block size of 2, $2^3 - 1 = 7$ extra voxels are transferred for each voxel requested from another node. For a block size of 3, $3^3 - 1 = 26$ additional voxels are transferred, and so on. We tried using greater block sizes for a selection of the data sets in Table F.2, and discovered that the Block Transfer strategy rapidly becomes worse than the Bounding Volume strategy as the block size increases.

Summary

The Minimum Communication strategy wins by more than a margin. For 128 nodes and global load-balancing in Benchmark 1, it beats the Block Transfer strategy by a factor of 2.66. For 256 nodes in Benchmark 3 with local load-balancing, it is 24.28 times as fast as the Bounding Volume strategy. On average, it is about twice as fast as Block Transfer and five times as fast as the Bounding Volume strategy, when load-balancing is enabled. The increased caching from the other strategies is not enough to compensate for the increase in communication time, even on Njord. Also, when increasing the number of polygons in Benchmarks 2 and 3, the Bounding Volume strategy does not win at all.

That is not to say that the other strategies would not have worked better in other workloads. For example:

[2] If the polygon is parallel to one of the principal planes, the bounding volume degenerates to the bounding plane.

[3] With the Block Transfer strategy, for small block sizes (as we used in our benchmark), the data transferred is slightly larger than a bounding plane. With larger block sizes, the data transferred is more like a bounding volume.

- If the number of move-extract cycles increase, or
- if the average distance each vertex is moved in a move-extract cycle is decreased or increased,

the other strategies may be better. Therefore, we recommend testing the framework under the workload that reflects the application in use. The benchmarking suite (see Appendix B) makes such testing easy.

5.3.3 Discussion 3. Performance of Load-Balancing Algorithms

Standard Deviation of Load

In order to discuss the load-balancing algorithms, we will consider one metric in addition to the wallclock time. Consider Figure 5.3 on the following page, which shows the logarithm of the *standard deviation* of the load on each node in Benchmark 2 under different load-balancing strategies. The average standard deviation of the load with a set of nodes N is computed by

$$\overline{S} = \frac{1}{\#\text{cycles}} \sum_{j=1}^{\#\text{cycles}} \sqrt{\frac{1}{|N|-1} \sum_{i \in N} L_j(i)^2} \tag{5.4}$$

($\overline{S} = 0$ when $|N| = 1$), where $L_j(i)$ denotes $L(i)$ as defined in Section 4 on page 65 for move-extract cycle j, and #cycles denotes the number of move-extract cycles, which in our case is 5. Thus, S measures the standard deviation of the load, averaged over all move-extract cycles in a single run. Note that this quantity is automatically computed by the benchmarking program described in Appendix B.

Note that the y-axis in Figure 5.3 is *logarithmic*, due to the large differences in the load when using different strategies. Actual standard deviations are included in Appendix F.2.

For work distribution to be as efficient as possible, the standard deviation of the load should also be as low as possible. Therefore, this value measures how efficient the load-balancing algorithms were at distributing work between nodes. However, note that the standard deviation measure does not take into account the additional communication time required to actually *make* the standard deviation become lower.

Notice the spike occurring when going from 4 to 8 nodes with the global and Manhattan strategies. Although these load-balancing algorithms generally reduce the standard deviation of the load as the number of nodes increases, they are only greedy approximations to an NP-complete problem. For certain numbers of nodes, the strategies may make particularly bad choices. This is clearly the case when going from four to eight nodes.

Load-Balancing Pays Off

Our initial observation is that *load-balancing almost pays off*. In all benchmarks, at least one of our load-balancing algorithms outperform the algorithms not using load-balancing. With very few exceptions does one of our algorithms give lower performance than disabling algorithms — in particular, when using the Bounding Volume strategy with a certain number of

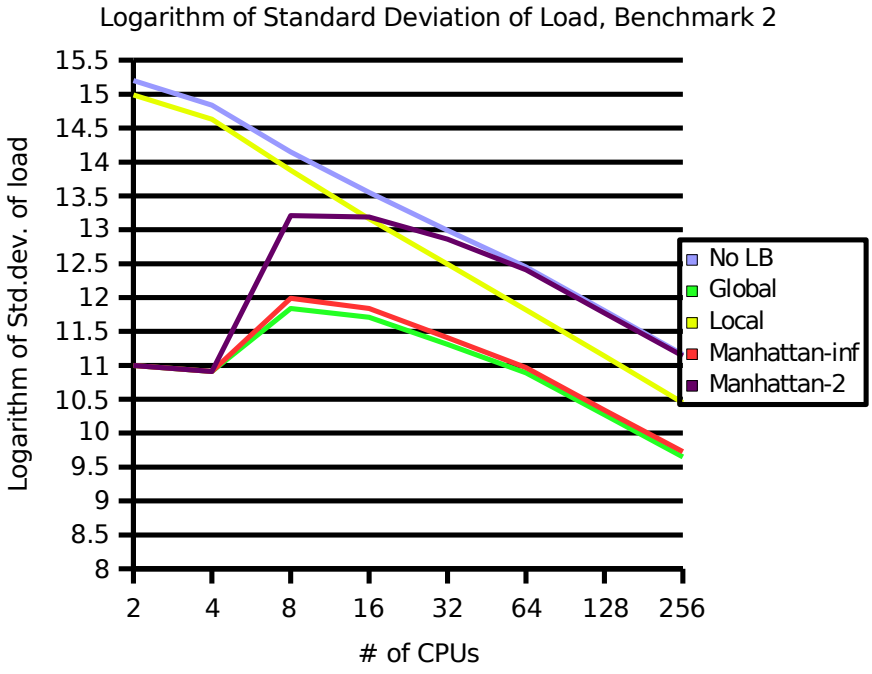


Figure 5.3: Standard deviation of the load in Benchmark 2

nodes, where the migration costs are very high. In all benchmarks, the best load-balancing algorithm outperforms using no load-balancing (for the same caching strategy) by a factor of about 3.5 on average and at most 5.77 in the best case (with the Bounding Volume strategy in Benchmark 1, using 128 CPUs).

The Global And Manhattan-∞ Strategies: Winners With Increasing Number of Polygons

From the Figure 5.3, it can be seen that *the global load-balancing strategy provides the best work distribution and smallest $L(i)$ values.* The goal of the global load-balancing strategy was precisely to produce a good work distribution. However, considering Figure 5.2 and Tables 5.1, 5.2 and 5.3 again, the local load-balancing strategy seems to produce the lowest running time when the number of nodes increase, *but only when using a small amount of polygons*. In Benchmark 1, which used 5000 polygons, the local load-balancing strategy wins when the number of nodes is equal to or exceeds 16 and when using 4 nodes. In Benchmark 2, with 10000 polygons, it wins when the number of nodes is above or equal to 128 or less than or equal to 4, and in Benchmark 3, with 15000 polygons, it does not win at all except when using 2 and 4 nodes.

As the number of polygons increase, the computational complexity increases, so the computation time increases. Then, the increased parallelism of the computations provided by the global strategy (and Manhattan strategy with an infinite threshold) — which can be seen that the standard deviation in Table F.4 of the load is lower with these strategies — will lead to savings which *catch up* with the increase in communication time when using these strategies. Seemingly, the decrease in communication time provided by the local load-balancing strategy does not make up for the loss of parallelizing the computations *as the data sets gets*

larger.

Further Analysis of The Local Strategy

The local load-balancing strategy is, however, superior to the other strategies, even with many polygons, when using very few (2 and 4) nodes. This is explained by the standard deviation when using no load-balancing with few nodes being higher than using no load-balancing with more nodes (Figure 5.3, note the logarithmic scale). Therefore, when using few nodes, the global and Manhattan-∞ strategies must migrate many polygons between few nodes to other nodes in order to reduce the standard deviation. Since there are few nodes, the communication can not happen in parallel. The local strategy thus wins in these situations, since less data is transferred.

Further Analysis of the Manhattan Strategies

The Manhattan strategy with infinite threshold and the global strategy produce mostly equal work distributions. However, the difference may be greater if the polygons are not randomly spread among all nodes. As we mentioned in the previous chapter, the Manhattan strategy with a large threshold will be most effective when a few nodes are severely overloaded. With our random meshes, about half of the nodes are overloaded and half of the nodes are underloaded. Therefore, it is not surprising that the Manhattan strategy with infinite threshold produces similar results to the global load-balancing strategy. However, for some situations, the Manhattan strategy is consistently superior to the global strategy — for instance, when using the Block Transfer strategy in Benchmark 1.

The Manhattan strategy with a threshold of 2 performance is disappointingly weak. Although the strategy manages to keep up with the other strategies when the number of nodes are small, it quickly converges to have the same performance as using no load-balancing at all. Only when there are very few polygons and only two nodes in Benchmark 1 does the Manhattan strategy with $\tau = 2$ outperform the other strategies, but only slightly. In general, it seems that in situations where work distribution matters (large data sets), the strategy fails to provide a sufficiently good distribution to outperform the global and Manhattan-∞ strategies, while in situations where communication time matters, the strategy fails to reduce the communication requirements more than the local load-balancing strategy.

Summary

As for the caching and transfer strategies, general guidelines for the load-balancing strategies are hard to give, since the superior strategy depends on the number of polygons, polygon size, and the computation to be done. However, we can say that:

- Use the local load-balancing strategy when there are relatively few polygons, and always with very few (2 to 4) nodes.
- With increasing number of polygons, the global or Manhattan-strategies with an infinite (or relatively large) threshold should be increasingly used. Due to the increasing amount of communication that occurs with more nodes, the local load-balancing strategy may still be superior when using many CPUs — but will eventually lose also in this case as the data set sizes increase.

5.4 Voxelization Benchmarks

We start by describing our test methodology in Section 5.4.1. Next, results are shown in Section 5.4.2, and finally, the findings are summarized in Section 5.4.3.

5.4.1 Test Equipment and Methodology

We have used two machines for benchmarking of the voxelization algorithms:

- **P4** has the following equipment:
  - Intel Pentium 4 3.2 GHz CPU with HyperThreading (SMT)
  - 1 GiB RAM
  - NVIDIA GeForce 8800 GTX GPU, 768 MiB RAM, connected via PCI Express x16 bus
  - Ubuntu Linux 7.1, with Linux kernel 2.6.20
  - NVIDIA GPU drivers version 1.0.9755
  - NVIDIA Cg library version 1.5
- **Xeon** has the following equipment:
  - 2 Intel Xeon 5160 3.0 GHz, each of which is a dual-core processor
  - 4 GiB RAM
  - NVIDIA Quadro 3500 GPU with 512 MiB RAM, connected via PCI Express x16 bus
  - Red Hat Enterprise Linux 4, with Linux kernel 2.6.9-EL4
  - NVIDIA GPU drivers version 1.0.9755
  - NVIDIA Cg library version 1.5
- **Njord**, which is the Njord supercomputer at NTNU, described in the caching and load-balancing benchmarks.

P4 has the quickest GPU, but a slower, single-core CPU compared to Xeon (despite the GHz being higher). Xeon has a GPU which is one generation behind P4, but has two CPUs which are a generation *ahead* of P4. Speedups should therefore be larger on P4 than Xeon. Njord has no GPU, but is massively parallel. In contrast to the quad-core Xeon, each CPU on Njord has a dedicated memory bus, which should increase performance if the voxelization is bounded by memory access time instead of computation time.

We have benchmarked the voxelization algorithms for different triangles of different sizes. Each test result given in the following sections is the average of 20 runs. In each run, we generate 512 different triangles from three vertices passed to the program: If the vertices are v_1, v_2 and v_3, we generate $2^9 = 512$ triangles by assigning all sign combination to each component of the vertices. Thus, for instance, run 0 will have all components as positive, while run 1 will set the x-component of v_1 to be negative, and so on.

Since we generate triangles in this manner, not all triangles in a single run will have the same area. But we are not interested in testing the voxelization algorithms for only a specific area

Machine/Algorithm \ Avg. area	10	100	1000	10000	10^5	10^6	10^7
P4: gpu-voxelize	0.25	0.22	0.27	0.71	**1.34**	**2.14**	**1.68**
P4: (base) cpu-voxelize-1cpu	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Xeon: gpu-voxelize	0.01	0.01	0.08	0.26	0.77	**1.32**	0.37
Xeon: (base) cpu-voxelize-1cpu	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Xeon: cpu-voxelize-2cpu	0.70	0.50	0.67	0.95	0.84	0.77	0.24
Xeon: cpu-voxelize-4cpu	0.18	0.17	0.55	0.91	0.83	0.80	0.26
Njord: (base) cpu-voxelize-1cpu	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Njord: cpu-voxelize-2cpu	1.00	1.00	**1.27**	**1.26**	**1.40**	**1.20**	**1.35**
Njord: cpu-voxelize-4cpu	0.67	1.00	**1.47**	**1.54**	**1.69**	**1.61**	**1.53**
Njord: cpu-voxelize-8cpu	0.50	0.83	**1.65**	**1.94**	**2.21**	**1.94**	**1.93**
Njord: cpu-voxelize-16cpu	0.33	0.63	**1.40**	**1.92**	**2.12**	**1.79**	**1.76**
Njord: cpu-voxelize-32cpu	0.25	0.45	0.93	**1.34**	**1.38**	**1.29**	**1.18**

Table 5.4: Speedups of voxelization algorithms compared to single-core CPU version for each machine. Boldface values mark entries where the speedup is greater than 1.

size. Rather, we want to find out how the voxelization performs with triangles with areas that *on average* are equal to a specific area. In applications, if the average triangle size is 10, it is very unlikely that all triangles are of area 10. More likely is it that there is some variance in the area of the triangles. This is what we will be benchmarking by generating many different triangles in the manner we described above.

All systems have been idle during benchmarking.

The benchmarks measure the time it takes to compute precisely the same results on the CPU and GPU. That is, the GPU measurements show the time it takes to pass three vertices to the application until a list of voxels intersecting the triangle spanned by the vertices is available in RAM.

5.4.2 Results

The speedups of each algorithm relative to the single-core CPU algorithm for each machine is shown in Table 5.4. We have also included the absolute timings in Table F.5 in Appendix F.3.

On Njord, when executing with 32 threads, we used the Symmetrical Multi-Threading (SMT) feature available on the Power5+ CPUs. SMT enables instructions from several threads to be in the CPU pipeline at one time, thereby potentially increasing performance. Although P4 has a limited SMC-feature called HyperThreading available, we only used one thread on this machine, since HyperThreading has limited advantages for computationally intensive tasks with few cache-misses and branch mispredictions. HyperThreading exploits idle time during such stalls. No actual execution units are duplicated in the processor. Therefore, HyperThreading is mainly used for latency reduction in interactive applications and is of limited importance in our application.

Since Xeon is quad-core, we also tried using four threads on this machine.

CPU times relative to the single-core version for each machine of the runs in Table F.5 are shown in Table 5.5. Absolute timings are provided in Appendix F.3.

Avg. area / Machine/Algorithm	10	100	1000	10000	10^5	10^6	10^7
P4: gpu-voxelize	**0.30**	**0.31**	**0.37**	**0.83**	**1.46**	**2.15**	**1.72**
P4: cpu-voxelize-1cpu	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Xeon: gpu-voxelize	**0.02**	**0.02**	**0.12**	**0.38**	**0.93**	**1.51**	**1.09**
Xeon: cpu-voxelize-1cpu	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 5.5: CPU-time speedups of voxelization algorithms compared to single-core CPU version for each machine. Boldface values mark entries where the speedup is greater than 1.

5.4.3 Discussion and Findings

By considering Table 5.4 on the preceding page, we draw the conclusions discussed below.

GPU voxelization algorithm only gives a speedup for large triangles

As previous work suggests [27], large amounts of time goes into downloading and uploading data to the GPU, in addition to setup time. The time taken on the GPU, T_{GPU}, for the voxelization algorithm, can be written as

$$T_{\mathsf{GPU-Voxelize}} = T_{\mathsf{Setup}} + T_{\mathsf{Upload}} + T_{\mathsf{Rasterize}} + T_{\mathsf{FP}} + T_{\mathsf{Download}} + T_{\mathsf{Postprocess}} \tag{5.5}$$

where T_{Setup} is the GPU initialization time, T_{Upload} is the time to upload data to the GPU, which is a small quantity in our case since we only pass three vertices to the GPU, $T_{\mathsf{Rasterize}}$ is the time the GPU hardware rasterization takes, T_{FP} is the time our fragment processor program applying the inverse rotations takes to execute, T_{Download} is the time to download the resulting texture to the CPU, and finally, $T_{\mathsf{Postprocess}}$ is the time the postprocessing step on the CPU takes (see Figure 3.15 on page 63).

In general, the GPU version only showed speedups for very large triangle sizes. There are two primary reasons as to why we do not obtain a greater speedup, and why the speedup only comes for large triangle sizes:

- *The arithmetic complexity of the rotation is low*. An inverse rotation is essentially a 4×4 vector product, but the fourth row and column of the matrix are zero except for the rightmost lower element (see Equation E.18 on page 131). A 3×3 matrix-vector product takes 9 multiplications and 6 additions. If we incorporate the initial translation into the rotation matrix, three additional floating-point additions are needed, giving a total number of floating-point operations of 18. Efficient Discrete Cosine Transform (DCT) algorithms, which are commonly used in data compression, require 80 floating-point operations per transform value [27]. When executing DCT on the GPU on Xeon, we barely got a speedup when using a single core, and no speedup when using more cores [27]. Thus, since our current problem is only 22.5% of the DCT complexity, our small speedups on the GPU are easily explained. The gains we obtain from not having to upload large amounts of input data (which is the case with applying the DCT), which we hoped would save some time, are not enough to offset the decrease in arithmetic complexity until the triangles get very large.
- *Postprocessing is required*. Postprocessing on the CPU is required subsequent to the GPU fragment program execution. The texture received from the GPU is the bounding

plane of the triangle. Therefore, the CPU must scan the texture in order to know which values in the downloaded texture actually represent valid voxels. But this process actually iterate over *more* values than the corresponding CPU voxelization algorithms (see Section 3.6.2 on page 53), which only iterate over the coordinates to which the rotation matrix is multiplied. Hence, the GPU only helps to offload the inverse rotation computations from the CPU, but the CPU still has to scan over all the coordinates of the triangle to which the rotation computations are applied. Since the rotation computations are *very* efficient on a modern CPU, the speedup only comes as the triangles get very large. This is also supported by the fact that Table F.6 shows the CPU times for the GPU algorithms to be very close to the wallclock times of Table F.5.

Nevertheless, as we concluded in [27], as the data set size increases, the computation time dominates. Therefore, we do get a speedup from using the GPU for large triangles — albeit not quite the speedup we initially hoped for. The savings we obtained in not having to upload data were not enough to compensate for the decreased arithmetic complexity.

Notice that the speedup decreases when going from an area of 10^6 to 10^7, both on the GPU and on Njord. This is due to different triangles being generated for each benchmark, so the triangles in the 10^6 test may be quicker to voxelize than the triangles generated in the 10^7 test. Furthermore, the GPU drivers seem to use larger amounts of time when the triangle size is 10^7, perhaps due to increased memory usage.

Multi-Core Algorithms

P4 has a slower CPU but a quicker GPU compared to Xeon. Thus, the speedups obtained on P4 are greater than Xeon. However, although P4's GPU is quicker, this is not enough to offset the slowdown caused by the post-processing step. Since this step is CPU-bound, the GPU algorithm on Xeon outperforms the GPU algorithm on P4.

Table F.5 shows that Xeon gives a *slowdown* when using multiple cores. This is a likely indication of the problem being *memory-bound*. Due to the low arithmetic complexity, the computation time is dominated by memory accesses. The Xeon architecture only has one shared memory bus between all cores. Therefore, as more cores are used, contention occurs on the bus, and slowdown occurs due to the contention preventing each core of filling up its execution pipeline, in addition to the added overhead induced by OpenMP when using more cores.

This situation is different for the Njord, which is the Njord supercomputer. On Njord, each of the 8 dual-core CPUs on each node have separate memory buses connected to local memory banks. Thus, there is no contention — and indeed, as Table F.5 shows, speedup occurs when using up to 8 CPUs. However, when using *more* than 8 CPUs, memory-bus contention also occurs on Njord, and the same problem as on Xeon can be seen.

It is also interesting that both P4 and Xeon outperform Njord on this operation. For example, for triangles of size 10^4, P4 uses 1.28 seconds, Xeon uses 0.58 seconds and Njord uses 1.18 seconds *when using 8 CPUs* and 2.29 seconds when using one CPU! Similar results hold for other triangle sizes. This either suggests that the Power5+ architecture of Njord is not as well suited to our problem as the Intel architectures, *or* — perhaps more likely — that the compiler on the Intel systems produces more efficient code[4].

[4]On Njord, we used IBM's C compiler. On the Intel machines, we used Intel's C compiler.

Chapter 6

Conclusions and Future Work

In this chapter, our findings are summarized and directions for future work are provided.

6.1 Caching and Transfer Algorithms

Our conclusion from our caching benchmarks were that the Minimum Communication strategy, which is the strategy that uses the least communication, was the quickest in nearly all cases. This strategy outperforms the Block Transfer and Bounding Volume strategies by factors of up to 2.66 and 24.28, respectively. On average, it is about twice as fast as Block Transfer and five times as fast as the Bounding Volume strategy, when load-balancing is enabled.

The other caching strategies ran slower in most cases. It should be emphasized, however, that *the other strategies only failed for our specific workloads*. If, for example

1. The distance each vertex is moved is decreased,
2. not *all*, but only a few, vertices are moved in each move-extract cycle,
3. the number of dynamic operations increase,
4. the number of polygons changes,
5. the average polygon size changes, or
6. the computation done over the voxelized polygon surfaces is different,

the other strategies may be more efficient. Additionally, the optimal strategy also to a certain degree depends on the actual computer the framework runs on. Other systems than Njord have different CPU and network interconnect characteristics. Thus, in a practical application, benchmarks on actual application workloads should be run in order to determine the optimal strategy.

6.2 Load-Balancing Algorithms

We found that for all benchmarks, at least one of our load-balancing always outperforms using no load-balancing for our workloads, and in almost all cases, all load-balancing algorithms outperform when running without load-balancing. Considerable savings are obtainable when using load-balancing: On average, the best load-balancing algorithms obtain

a speedup of approximately 3.5 over using no load-balancing (for the same caching algorithm). In the best situation, the speedup was 5.77.

We found that the local load-balancing strategy is the superior strategy when there are:

1. few number of nodes (2 to 4), since, with the other strategies, there will be large amounts of communication between few nodes (which is not done in parallel), and
2. few numbers of polygons, since, in that case, the total arithmetic complexity of the problem is low and communication time is therefore more important than parallelizing the computations.

As the number of polygons increase, we saw that the global and Manhattan-∞ strategies gradually perform better than the local load-balancing strategy, since as the total arithmetic complexity increases, the savings gained from better work distribution catch up with the savings gained from decreased communication in the local load-balancing strategy. However, we also saw that the total amount of communication done when increasing the number of nodes also increases, so that the local load-balancing strategy may still be the optimal strategy in situations with many polygons if we use many nodes.

As with the caching and transfer strategies, we should also emphasize that the conclusions drawn about whether or not to use load-balancing is somewhat tied to our workload. If, for example, the average polygon size increases, the local load-balancing strategy may not lose to the other strategies as quickly as it did in our cases, since in that case, the other strategies will need to transfer more voxels between the nodes. Also, with an increasing average size, the number of coordinate spaces a polygon intersects increases, so the local load-balancing strategy can consider more nodes when migrating a polygon.

Similarly, the optimal strategy also depends on the computation to be done, in addition to machine and interconnect characteristics. Thus, as with the caching and transfer strategies, in a practical application, benchmarks on actual application workloads should be run in order to determine the optimal load-balancing strategy.

6.3 Voxelization Algorithms

While our GPU voxelization algorithm did provide a speedup over single-core CPUs of at most 2.14, speedup was obtained only for triangle sizes with an area in the order of 10^5. On Njord, our multi-core algorithm obtained a speedup of 2.21 when using 8 CPUs over the single-core version. No speedups were obtained on multi-core machines with a single shared memory bus.

The bottlenecks of this algorithm were found to be:

1. The algorithm has low arithmetic complexity. Therefore, the decreased computation time when running the voxelization on the GPU is overshadowed by the uploading and downloading data from/to the GPU.
2. The CPU must post-process the data from the GPU, which is downloaded to main memory. Since the problem is of such low arithmetic complexity, communication between the CPU and RAM is the bottleneck, and not the actual computation. Therefore, the GPU algorithm gives limited speedup.

The voxelization problem is *memory-bound*. This is why we only get significant speedups on the GPU when voxelizing very large triangles, since in that case, the time the arithmetic operations take eventually dominate the memory access and post-processing time. This is also the reason why we do not obtain significant speedups when using dual- or quad-core CPUs with shared memory buses, since the shared memory bus becomes the bottleneck. However, on systems with multiple independent memory buses, such as the Njord supercomputer, we *do* get a speedup when using multiple cores.

Since most triangles in typical applications where the framework will be used will have an area in the order of 10^5, and since the GPU algorithms save neither wallclock nor CPU time for smaller triangles, the GPU voxelization algorithm will unfortunately be of limited use by itself. However, in the next section, we suggest future work that, if explored, might give better speedups.

6.4 Future Work

There are many aspects of the framework which can be explored further. In this section, we provide some pointers to areas where our framework can be extended.

6.4.1 Caching and Transfer and Load-Balancing Algorithms

Dynamic Selection of Algorithms and Parameters from Workload and Machine Characteristics

We only benchmarked a very specific workload. As we mentioned in Section 5.2.1, many parameters which influence the efficiency of the caching and load-balancing algorithms can be varied. Since the optimal algorithms and algorithm parameters depends on the workload, a fully automatic system which maps from a set of workload characteristics:

1. The data set size,
2. average polygon size,
3. the number of polygons,
4. the number of meshes,
5. the number of move-extract cycles,
6. approximately how far each vertex is moved in each cycle,
7. which computation to do,
8. the number of nodes used, and
9. the characteristics of the machine and interconnect (α and β values) used,

to the optimal algorithm parameters:

1. The caching and transfer strategy used,
2. the load-balancing strategy used,
3. the δ value used in the load-balancing strategy,

4. the τ value used in the Manhattan load-balancing strategy (if the Manhattan strategy is used), and
5. the number of fine-grained, multi-core voxelization threads versus the number of coarse-grained processes on machines with several independent memory buses,

would be desirable.

We have given some general guidelines as to when to use which strategies, but only for our specific workloads. Further investigations will have to be done in order to create a fully automatic system for determining the best parameters.

Also, in making such a system, computations that are done *outside* the polygonal model and on a per-voxel basis must also be considered. What happens if we require doing some computation that is independent of the polygonal model and only works on a per-voxel basis? Some of our caching strategies may be more efficient than others if they are not only used for computing functions over the voxelized polygon faces, but also combined with such computations. Therefore, the caching and transfer algorithms should not be considered only with regard to the computations being done over the voxelized polygon surfaces, and may need to be adjusted in order to be efficient for other types of computations as well.

The ultimate goal is thus to produce a system which can dynamically chooses which algorithms to use based on the workload and underlying hardware. This will, however, be a large task to do in full generality, and also require unifying the load-balancing algorithms.

Unify Load-Balancing Algorithms Into a Fully Dynamic Load-Balancing Algorithm

Our current load-balancing algorithms are not fully dynamic.

Our current load evaluation step — which determines whether there is any imbalance in the load — depends on a tuned, pre-supplied δ value. The optimal δ value should be detected automatically, so that this step relies on no tuned or pre-supplied parameters. Since δ is not computed automatically in our algorithms, the profitability determination step — which determines whether load-balancing should be done — in our algorithms is not optimal. We currently only speculate when load-balancing is advantageous on the basis of the pre-supplied δ value. To compute δ automatically, a further understanding of how the previously mentioned workload and machine parameters are related is required, since the computation will likely depend on many of these parameters.

Furthermore, the task selection step — selecting the destination node to which excess load is transferred — is what separates the global, local and Manhattan load-balancing strategies. Currently, we leave it up to the user to choose what to do in this step. In a fully dynamic load-balancing system, the task selection should choose one of these strategies based on workload characteristics. We saw in Chapter 5 that the optimal strategy depends on both the number of polygons and the number of CPUs used. Likely, many of the other parameters listed on the previous page will also have an effect on which load-balancing strategy is optimal. Thus, before a fully dynamic load-balancing scheme can be developed, finding the precise relationships between the workload characteristics and the optimal strategy for task selection is also necessary.

Investigate the τ Parameter Further

One task, which may be necessary to do in order to create fully dynamic load-balancing, is to investigate the τ parameter in the Manhattan load-balancing strategy further. In our benchmarks, we tried two values for τ in the Manhattan load-balancing strategy. With $\tau = \infty$, the performance was on par, and in some cases better, than the global strategy. With $\tau = 2$, performance got gradually worse when we increased the number of CPUs.

The optimal τ parameter is thus likely a function of the number of CPUs used. τ should thus be increased as the number of CPUs increase. Finding precisely the optimal value of τ, given a number of CPUs used, should therefore be investigated. It should also be investigate whether or not different τ values will boost the performance of the Manhattan load-balancing strategy, particularly when using many CPUs.

Improved Load-Balancing Algorithms?

Our load-balancing algorithms may have some room for improvement. In particular, our local load-balancing algorithm only moves polygons to nodes whose coordinate spaces intersect the bounding volume of the polygon. However, some voxels of the polygon's bounding volume have might previously been sent to other nodes whose coordinate spaces do *not* necessarily intersect the bounding volume of the polygon.

These nodes could be considered when choosing which node to migrate the polygon to in the local load-balancing strategy. If these nodes are considered, the local load-balancing strategy may be able to further reduce the standard deviation of the load, since more nodes are considered, and at the same time, not significantly increase the amount of voxels that need to be transferred to the node the polygon is migrated to.

This would, however, require efficient algorithms for keeping track of which nodes voxels inside the bounding volume of a polygon have been previously sent to, and perhaps also how many voxels of the bounding volume that have been sent to each node. This may be more complicated than it initially seems: The bounding volume of the polygon considered may intersect the coordinate spaces of other nodes than the centroid node. These other nodes could also have previously transferred parts of the bounding volume intersecting the polygon and their coordinate spaces to other nodes, which should also be considered when choosing the node to migrate the polygon to.

6.4.2 Voxelization Algorithms

Compare to Kaufman's Algorithm

Our voxelization algorithm was not compared to Kaufman's algorithm [46]. As we mentioned in Chapter 3.6, our algorithm uses more floating-point instructions than Kaufman's algorithm, but eliminates the two branch instructions in the inner loop of Kaufman's algorithm. Investigations should be made as to which algorithm is quickest for different polygon sizes.

Combine Computations and Voxelization on the GPU

We only used the GPU for voxelization. However, we could try to combine the voxelization part with doing computations on the GPU. For example, we have previously shown [27] that computing the LOT on the GPU gives very good speedups on the GPU. Therefore, if the voxelization and computation steps could be done on the GPU in one pass, with a data transfer between the CPU and GPU occurring only twice (once for upload and once for download), it may be possible to obtain greater speedups than first doing voxelization on the GPU, then downloading the voxel coordinates to the CPU, and then doing computations. If computations are done on the GPU, the post-processing on the CPU could also perhaps be avoided. Such an approach, however, would require uploading the voxel values to the GPU (which is not the case with our present algorithm, which only passes vertices to the GPU.)

Bibliography

[1] M. Aurnhammer and K. Tönnies. The Application of Genetic Algorithms in Structural Seismic Image Interpretation. In *Lecture Notes in Computer Science*, volume 2449. Springer, 2002.

[2] L. F. Kemp, J. R. Threet and J. Veezhinathan. A neural net branch and bound seismic horizon tracker. In *62nd Annual International SEG Meeting, Expanded Abstracts, Houston, USA*, 1992.

[3] D. Gibson, M. Spann and J. Turner. Automatic Fault Detection for 3D Seismic Data. In C. Sun, H. Talbot, S. Ourselin and T. Adriaansen, editors, *Proceedings of the 7th Conference on Digital Image Computing: Techniques and Applications, 10–12. Dec. 2003, Sydney, Australia*, 2003.

[4] R. Pepper and G. Bejarano. Advances in Seismic Fault Interpretation Automation. In *Poster presentation at AAPG Annual Convention, 19–22. Jun. 2005*, 2005.

[5] K. M. Tingdahl and M. de Rooij. Semi-automatic detection of faults in 3D seismic data. *Geophysical Processing*, 53(4), 2005.

[6] F. Dachille IX and A. Kaufman. Incremental Triangle Voxelization. In *Proceedings of Graphics Interface 2000*, pages 205–212, 2000.

[7] E. Camahort and I. Chakravarty. Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures. *Proceedings of the 1993 symposium on Parallel rendering (PRS 1993)*, pages 89–96, 1993.

[8] J. C. Cavendish, D. A. Field and W. H. Frey. An apporach to automatic three-dimensional finite element mesh generation. *International Journal for Numerical Methods in Engineering*, 21(2):329–347, 1984.

[9] T. Möller and E. Haines. *Real-Time Rendering*. AK Peters, 2nd edition, 2002.

[10] M. Henne and H. Hickel. The Making of "Toy Story". *Digest of Papers, Compcon '96, 'Technologies for the Information Superhighway'*, pages 463–468, 1996.

[11] R. Samanta, T. Funkhouser, K. Li and J. P. Singh. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108, 2000.

[12] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[13] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, 1994.

[14] M. F. Worboys. *GIS: A Computing Perspective*. Taylor & Francis, 1st edition, 1995.

[15] D. Hearn and M. P. Baker. *Computer Graphics with OpenGL*. Prentice Hall, 3rd edition, 2004.

[16] Wikipedia. Polygon — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Polygon&oldid=103527323`, 2007. (Last visited 17-06-2007).

[17] B. Chazelle. Triangulating a simple polygon in linear time. *Proceedings of the 31st Annual Symposium on Foundations of Computer Science, 22–24 Oct 1990*, 1:220–230, 1990.

[18] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf. *Computational Geometry — Algorithms and Applications*. Springer, 2nd edition, 2000.

[19] A. Gueziec, G. Taubin, F. Lazarus and B. Hom. Cutting and stitching: converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):136–151, 2001.

[20] B. G. Baumgart. Winged edge polyhedron representation. *Proceedings of 1975 AFIPS National Computer Conference*, pages 589–596, 1975.

[21] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley, 3rd edition, 2004.

[22] T. H. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[23] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1st edition, 1997.

[24] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/mpi-20.ps`, 1997. (Last visited 17-06-2007).

[25] The OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`, 2005. (Last visited 17-06-2007).

[26] E. Kilgariff and R. Fernando. The GeForce 6 Series GPU Architecture. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 30, pages 471–491. Addison-Wesley, 1st edition, 2005.

[27] L. C. Larsen. Utilizing GPUs on Computer Clusters. *Pre-Master's project report, HPC Group, Dept. of Computer and Information Science, Norwegian University of Science and Technology*, 2006.

[28] P. Micikevicius. GPU Computing for Protein Structure Prediction. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 43, pages 695–702. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[29] J. Bolz, I. Farmer, E. Grinspun and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[30] J. Krüger and R. Westermann. GPU Computing for Protein Structure Prediction. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 44, pages 703–718. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[31] C. Kolb and M. Pharr. Options Pricing on the GPU. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 45, pages 719–731. Addison-Wesley, Upper Saddle River, NJ, USA, 1st edition, 2005.

[32] X. Wei W. Li, Z. Fan and A. Kaufman. Flow Simulation with Complex Boundaries. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 47, pages 747–764. Addison-Wesley, 1st edition, 2005.

[33] T. Sumanaweera and D. Liu. Medical Image Reconstruction with the FFT. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 48, pages 765–784. Addison-Wesley, 1st edition, 2005.

[34] P. Kipfer and R. Westermann. Improved GPU Sorting. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 46, pages 733–746. Addison-Wesley, 1st edition, 2005.

[35] NVIDIA Corporation. NVIDIA CUDA — Compute Unified Device Architecture — Programming Guide. `http://developer.download.nvidia.com/compute/cuda/0_81/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf`, 2007. (Last visited 17-06-2007).

[36] J. C. Meyer. Load Balancing Visualization Servers. Master's thesis, Dept. of Computer and Information Science, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, June 2004.

[37] V. Kumar, A. Grama, A. Gupta and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings, NY, USA, 1994.

[38] S. Piersall and S. Elfayoumy. DYLAPSI: A Dynamic Load-Balancing Architecture for Image Processing. In *Proceedings of the 15th International Conference on Parallel and Distributed Computing (ISCA)*, 2002.

[39] A. C. Elster, M. Ü. Uyar and A. P. Reeves. Fault-Tolerant Matrix Operations on Hypercube Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, 1989.

[40] C. Hui and S. T. Chanson. Improved Strategies for Dynamic Load-Balancing. *IEEE Concurrency*, 7(3), 1999.

[41] J. Watts and S. Taylor. A Practical Approach to Dynamic Load-Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3), 1998.

[42] M. E. Mortenson. *Mathematics for Computer Graphics Applications*. Industrial Press, 2nd edition, 1999.

[43] A. J. Rueda, R. J. Segura, F. R. Feito and J. R. de Miras. Voxelization of solids using simplical converings. In *SHORT Communication Papers Proceedings, WSCG 2004, Feb. 2–6, 2003, Czech Republic*, 2003.

[44] B. G. Blundell and A. J. Schwarz. The Classification of Volumetric Display Systems: Characteristics and Predictability of the Image Space. *IEEE Transactions on Visualization and Computer Graphics*, 8(1), 2002.

[45] S. Pastoor and M. Wopking. 3-D displays: A review of current technologies. *Displays*, 17(2):100–110, 1997.

[46] A. Kaufman. Efficient Algorithms for Scan-Converting 3D Polygons. *Computer and Graphics*, 12(2):213–219, 1988.

[47] M. Sramek and A. Kaufman. Alias-free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics*, 05(3):251–267, 1999.

[48] J. Huang, R. Yagel, V. Filippov and Y. Kurzion. An accurate method for voxelizing polygon meshes. In *Proceedings of the IEEE Symposium on Volume Visualization, Oct. 19–20, 1998*, pages 119–126, 1998.

[49] D. Haumont and N. Warze. Complete polygonal scene voxelization. *Journal of Graphics Tools*, 7(3):27–41, 2002.

[50] S. Thon, G. Gesquiére and R. Raffin. A Low Cost Anitaliased Space Filled Voxelization Of Polygonal Objects. In *Proceedings of GraphiCon 2004*. 2004.

[51] Z. Dong, W. Chen, H. Bao, H. Zang and Q. Peng. Real-time voxelization for complex polygonal models. In *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications, Oct. 6–8, 2004*, 2004.

[52] E. Eisemann and X. Décoret. Fast Scene Voxelization and Applications. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 8, New York, NY, USA, 2006. ACM Press.

[53] X. Wei W. Li, Z. Fan and A. Kaufman. Flow Simulation with Complex Boundaries. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 47, pages 747–764. Addison-Wesley, 1st edition, 2005.

[54] Hsien-Hsi Hsieh, Yueh-Yi Lai, Wen-Kai Tai, and Sheng-Yi Chang. A flexible 3d slicer for voxelization using graphics hardware. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 285–288, New York, NY, USA, 2005. ACM Press.

[55] R. Gerber. *The Software Optimization Cookbook: High-Performance Recipes for the Intel®Architecture*. Intel Press, 1st edition, 2002.

[56] H. S. Malvar and D. H. Staelin. The LOT: transform coding without blocking effects. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(4):553–559, 1989.

[57] L. C. Duval and T. Q. Nguyen. Seismic data compression: a comparative study between GenLOT and wavelet compression. *Proceedings of SPIE conference on Wavelet Applications in Signal and Image Processing VII*, 3813(1):802–810, 1999.

[58] W. T. Corrêa, J. T. Klosowski and C. T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96, Aire-la-Ville, Switzerland, Switzerland, 2002.

[59] S. Bachthaler, M. Strengert, D. Weiskopf and T. Ertl. Parallel Texture-Based Vector Field Visualization on Curved Surfaces Using GPU Cluster Computers. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06)*, pages 75–82. Eurographics Association, 2006.

[60] J. Zhang, J. Sun, Y. Zhang, Q. Han and Z. Jin. A Parallel Volume Splatting Algorithm Based on PC-Clusters. *Computational Science and Its Applications — ICCSA 2004 (Lecture Notes in Computer Science)*, 3044:272–279, 2004.

[61] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[62] T. Sellis, N. Roussopoulos and C. Faloutsos. The R^+-tree: A Dynamic Index for Multi-Dimensional Data. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, 1987.

[63] N. Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[64] R. Schneider and H.-P. Kriegel. The TR^*-tree: A new representation of polygonal objects supporting spatial queries and operations. *Proceedings of the 7th Workshop on Computational Geometry (Lecture Notes in Computer Science)*, 553:249–264.

[65] Y.-J. Lee and C.-W. Chung. The DR-tree: A Main Memory Data Structure for Complex Multi-dimensional Objects. *GeoInformatica*, 5(2), 2001.

[66] E. G. Hoel and H. Samet. Data-parallel R-tree algorithms. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume III — Algorithms and Applications, pages 47–50. CRC Press, 1993.

[67] I. Kamel and C. Faloutsos. Parallel R-trees. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 195–204. ACM Press, 1992.

[68] H. Tan, Z. Yang, Y. Li and C. Shen. Hardware-Based Voxelization For True 3-D Display. *Transactions of Nanjing University of Aeronautics & Astronautics*, 23(1):59–63, 2006.

[69] N. R. Adam and A. Gangopadhyay. *Database Issues in Geographic Information Systems (Advances in Database Systems)*. Springer, 1st edition, 2003.

[70] T. K. Puecker and N. R. Chrisman. Cartographic Data Structures. *The American Cartographer*, 2(1):55–69, 1975.

[71] F. P. Preparata and M. I. Shamos. *Computational Geometry — An Introduction (Monographs in Computer Science)*. Springer, 1st edition, 1990.

[72] M. Mantyla. *Introduction to Solid Modeling*. W. H. Freeman & Co, 1st edition, 1988.

[73] L. Guibas and J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transactions on Graphics (TOG)*, 4(2), 1985.

[74] A. Khodakovsky, P. Alliez, M. Desbrun and P. Schröder. Near-optimal connectivity encoding of 2-manifold polygon meshes. *Graphical Models*, 64(3/4):147–168, 2002.

[75] S. R. Ala. Design methodology of boundary data structures. *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*, pages 13–23, 1991.

[76] J. B. Kuipers. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace and Virtual Reality*. Princeton University Press, 1st edition, 2002.

[77] Wikipedia. Quaternions and spatial rotation — wikipedia, the free encyclopedia, 2007. [Online; accessed 21-April-2007].

[78] D. H. Eberly. *Game Physics*. Morgan Kaufmann, 1st edition, 2004.

[79] Y. Arai, T. Agui and M. Nakajima. A fast DCT-SQ scheme for images. *Transactions of the Institute of Electronics, Information and Communication Engineers*, E71(11):1095–1097, 1988.

[80] E. Feig and E. Linzer. Scaled DCT Algorithms for JPEG and MPEG Implementations on fused multiply/add architectures. *Proceedings of the SPIE Conference on Image Processing Algorithms and Techniques; San Jose, CA; (USA); 25 Feb.-1 Mar. 1991*, pages 458–467, 1991.

[81] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[82] H. S. Malvar. *Signal Processing with Lapped Transforms*. Artech House, 1st edition, 1992.

Appendix A

Related Papers

A.1 3D Rendering Applications

- [10] gives an overview of the making of "Toy Story", which was the first full movie rendered using clusters. The movie was rendered using 300 CPUs; and the paper describes how the movie was made (many nontechnical aspects).
- [11] describes a rendering scheme suitable for clusters. Traditionally, rendering systems for animations and movies have used *frame-based parallelization* (parallelizing each frame). This scheme instead assigns to each node a different set of polygons to render. This paper does not apply particularly to our situation, since it only considers polygons with no voxels.
- [58] describes a similar approach as [11], which is particularly suitable for rendering large data sets with many triangles. However, the method does not apply directly to our problem, for the same reasons as the previous method.
- [13] gives a taxonomy for classifying parallel *volume* rendering algorithms. Roughly, we can separate between image-space and object-space partitioning. In image-space partitioning methods, the screen is split into several sections, and each node renders one section. Data is distributed either in a static or dynamic (varying between each frame) fashion on nodes. This method does less rendering work per node, but communication costs are greater than for object-space partitioning methods. Object-space partitioning methods spread voxels of an object to several nodes. Each node renders an image, and then, images are composed to a single, fully-rendered image. This approach gives less communication costs but redundant rendering work may be done.
- [59, 60] describe volume rendering schemes which are partly image-space and object-space based. This is not relevant to our case, since we are not optimizing rendering time, but extraction of voxels from a *given* polygon surface.
- [12] describes the marching cubes algorithm, which converts voxel representations to a polygonal isosurface representation. This algorithm is an example of an algorithm requiring quick access to logically adjacent voxels when processing.
- [7] describes an object-based rendering scheme used in (an early) seismological applications. It suffers from the problem we are trying to solve: No explicit polygonal model exists, so modification of an imposed structure is not user-friendly.

A.2 Geographic Information Systems

- [14] gives a general introduction to GISs, from a computing perspective. Discussed are both models and algorithms for creating GIS systems. Also discusses different operators on polygon meshes and data structures for such meshes.
- *R*-trees were first introduced in [61], and are commonly used in GISs to support *point queries* (find the polygon containing a point) and *range queries* (find all polygons within a specified square).
- Many variants of R-trees exist. The R^+ tree, introduced in [62], and the R^* tree, introduced in [63], have proven to be quicker than regular R-trees. These trees employ heuristics to reduce overlapping regions, thereby reducing the number of branches that must be visited during point queries.
- Other R-tree structures such as the TR^*-tree [64] and DR-tree [65] have been proposed. These trees are optimized for CPU-time and not disk access time, sacrificing memory usage for performance.
- Many other variants of R-trees exist. Consult [65] for pointers.
- Some work has been done on parallelizing R-trees. Parallel R-tree algorithms for construction and certain operations can be found in [66].
- Parallel algorithms for range queries using R-trees can be found in [67].

A.3 Voxelization Algorithms

- Kaufman's original algorithm [46] is the most commonly used algorithm for voxelizing simple polygons. The algorithm is an extension of similar 2D algorithms.
- Recent research has focused on voxelizing nonplanar objects [43, 47], or complex polygons [49], or guaranteeing a specific thickness [48].
- Other algorithms which apply low-pass filtering for visualization and antialiasing [6, 50] have also been proposed.
- [51] proposes a GPU-based triangle mesh voxelization algorithm which exploits the GPU's 2D rasterization hardware by rendering each rasterizing the triangle along an axis where the triangle has the greatest area, and then encoding the 3D coordinates in a set of 2D GPU textures.
- [52, 68] suggest a similar approach, but used for different applications.
- [53, 54] also do voxelization on the GPU, but consider solid voxelization and not surface-based voxelization.

Appendix B

Software User's Guide

In this appendix, we provide a user's guide to the software created in this thesis. While the primary purpose of this thesis is to create an algorithmic framework for dealing with polygonal structures on clusters, several benchmark programs have been developed. (We have also included a source code overview in Appendix C).

Two benchmarking programs exist: one for caching and load-balancing and one for voxelization, described in Sections B.1 and B.2, respectively. All programs are compiled using `Makefile`s, and although precompiled versions for Linux and IBM AIX (Njord) are included in `bin/` directory of the electronic attachment — we recommend that all programs are compiled from scratch before running. We describe how the programs can be compiled in the next sections.

The requirements for compiling the programs are:

- Intel x86 or IBM Power5+ series machine
- Linux kernel version 2.6 or IBM AIX 5.3 operating systems
- Intel C Compiler version 9.1 or the IBM C Compiler included with AIX 5.3
- OpenMPI MPI Library version 1.1.2 or IBM's MPI library included with AIX
- (For GPU voxelization): NVIDIA Cg library 1.5, NVIDIA OpenGL drivers 1.0.9755, NVIDIA GeForce 7 or more recent GPU, in addition to a GL Utility Toolkit (GLUT) library and a running X11 server configured to use the NVIDIA drivers.

For both programs, GPU voxelization support is by default enabled, but can be disabled by setting the `-DGPU_DISABLE` preprocessor flag when compiling. Examples are shown in the included `Makefile`s.

B.1 Cluster Polygon Framework program — `cpfw`

To test the caching and load-balancing strategies, the `cpfw` (Cluster Polygon Framework) program can be used. To install the program, unzip the attached file, and `cd` into the `cpfw` directory. The program is then compiled as follows:

To compile the program on Intel/Linux systems using OpenMPI, the OpenMPI C compiler must first be set to Intel's C compiler `icc`. This can be done by issuing the command

```
export OMPI_MPICC=/opt/intel/cce/9.1.045/bin/icc
```

where the path to `icc` must be modified according to where `icc` is installed. Next, unzip the electronic attachment, and execute the commands

```
make clean
make
```

to compile the program. An executable named `cpfw` will be generated.

Note that if the host machine has an older architecture than Intel's Core architecture, the compiler flags may have to be modified in order to create program that is able to run on the machine. In that case, remove the `-xT` compiler flag from `Makefile`.

If running on IBM AIX, instead of typing `make`, use the command

```
make -f Makefile.njord
```

to compile. The `make` program will then use `Makefile.njord` is used instead of `Makefile`. After the program has been compiled, the executable `cpfw` may be invoked by the command

```
./cpfw
```

1. If the target machine does not have GPU or the required libraries available, add `-DGPU_DISABLE` to the `CFLAGS` in the `Makefile`. This flag is disabled by default.
2. If the program is to be run on more than 32 CPUs or in more than 32 processes, add `-DSCALABLE_CACHE` to the `CFLAGS`. If less than 32 CPUs are used, `cpfw` will store a simple 32-bit bitstring with each voxel to keep track of which nodes a voxel has been sent to. If using more than 32 processes, more such bitstrings are required and thus, each voxel will use more memory. `-DSCALABLE_CACHE` enables this feature, supporting up to 256 processors. This flag is enabled by default all supplied makefiles (and all benchmarks were run with this flag enabled). Disabling the flag, however, reduces program memory usage.

The `cpfw` program reads a *script*, which consists of a sequence of commands and metadata such as 3D space size, number of repetitions, and so on[1]. Thus, to use `cpfw`, one must first generate scripts. This can be done automatically by the `genscript` program included in the `scripts/` folder of the accompanying source code. We therefore first describe how the `genscript` program works.

B.1.1 Generating Scripts with `genscript`

The `genscript` program can be used for generating scripts. A `Makefile` for Linux/Intel systems is included inside the `scripts/` folder[2]. To compile `genscript`, first, in the `cpfw/` directory of the electronic attachment, type

```
make clean
```

`cd` into the `scripts/` directory, and type

```
make
```

[1] Scripts do not contain any voxel data. This data is generated by `cpfw`, and contains random values. This is sufficient since the time the program takes to execute is fully independent of the actual voxel data.

[2] `genscript` is not supported on Njord, since there is little point in utilizing the supercomputer for simple script generation.

`OMPI_MPICC` must be set as described in the previous section. An executable called `genscript` will be generated.

What does `genscript` do? It generates a sequence of operations which:

- first creates a 16 meshes[3] and adds a set of polygons with a user-defined *approximate* average area,
- then performs the EXTRACT-ALL operation followed by variance and LOT computation over the voxelized polygon surface,
- then does a sequence of move-extract cycles, where *each vertex of all polygons* are moved by a random length, which on average is approximately equal to a user-defined parameter,
- then removes all polygons and destroys the mesh

The above operations are repeated n times, where n is a user-defined value. A sample script has been provided in the `test.cpfws` file in the `scripts/` subdirectory[4].

After successful compilation, the `genscript` program may be run with the command

```
./genscript
```

This command will print the syntax for running the program:

```
genscript - generate script files for cpfw
usage: ./genscript <NxMxK> <approx. average polygon size> <#polygons>
    <move-extract cycles> <move distance> <repeats> <output file>
```

The arguments are:

- `NxMxK`: Specify the size of the 3D coordinate space. For example, `100x200x300` specifies a coordinate space with 100 coordinates in the x-direction, 200 in the y-direction and 300 in the z-direction.
- `approx. average polygon size`: Specify the average polygon size of the polygons initially created. This is only a *rough estimate* of the average polygon size. Actual polygons created by the program may have an area greater or lower than what is specified by this parameter[5].
- `#polygons`: Specify the number of polygons to be created.
- `move-extract cycles`: Specify the number of move-extract cycles to perform.
- `move distance`: Specify the average distance each vertex is moved in a move-extract cycle.
- `repeats`: Specify the number of times the operations described above are repeated. For example, a value of 100 specifies that the adding, move-extract cycles and destruction operations will be done 100 times, sequentially.
- `output file`: Specifies the output file to which the script is written.

[3]The number 16 is coded into the program, and may be changed by changing a preprocessor `#define`. See `genscript.c`.

[4]The files we used for benchmarking have not been included, since these files are relatively large. However, the files in our benchmarks can be easily reconstructed using `genscript`

[5]In our benchmarks, we ensured that the actual polygon size created by `genscript` was close to the average polygon size we specified. This can be done easily since `genscript` prints the actual area of the polygons generated.

After the script has been created, the `cpfw` program can read it and execute benchmarks based on the instructions in the script. The program will print the actual average area of the triangles generated. This area may deviate from the area specified. Follow the instructions on-screen if this is the case. (In our benchmarks, we always used an actual area of 1000.)

B.1.2 Running Benchmarks with `cpfw`

The syntax for running the `cpfw` program is:

```
mpirun -np <procs> ./cpfw <transfer mode> <load-balancing strategy>
  <gpu-extract|cpu-extract> <threads> <script file>
```

Note that on an actual cluster, `mpiexec` must be used instead of `mpirun`, and the `-np` parameter may need to be replaced by `-n` or dropped entirely. The arguments to the program are:

- `transfer mode`: The caching strategy. Available options are:
  - `mc`: The Minimum Communication strategy
  - `bt=`K: The Block cache strategy, with block size K. For example, to specify a block size of 2, write `bt=2`.
  - `bv`: The Bounding Volume cache strategy
- `load-balancing strategy` specifies the load-balancing strategy, and must be one of:
  - `none`: No load-balancing
  - `global=`δ: The global load-balancing strategy, with threshold δ.
  - `local=`δ: The local load-balancing strategy, with threshold δ.
  - `manhattan=`δ: The Manhattan load-balancing strategy, with threshold δ and infinite Manhattan distance threshold.
  - `manhattan=`δ,th=τ: The Manhattan load-balancing strategy, with threshold δ and Manhattan distance threshold τ.
- `gpu-extract` or `cpu-extract`: State whether GPU or CPU voxelization algorithms will be used. Using GPU voxelization algorithms will only work if the `-DGPU_DISABLE` compiler flag was *not* set during compilation.
- `threads`: Specifies the number of OpenMP threads to use when voxelizing.
- `script file`: The script file to execute, which has been generated by the `genscript` program.

After the program has run, various benchmarking information is printed. An example output is shown below.

```
chrisl@linux # mpirun -np 16 ./cpfw mc global=0 cpu-extract 1 scripts/test.cpfws

Framework for Polygonal Structures on Clusters
Leif Christian Larsen <leifchl@idi.ntnu.no>

[cpfw] Configuration: 16 processes, 2D topology 4x4, 1 thread(s) per node
[cpfw] Transfer Mode: Minimum Communication
[cpfw] Load-balancing on each extract operation, global mode (delta = 0.00).
[cpfw] using CPU voxelization algorithm
```

```
[cpfw/script] Reading script
[cpfw/script] Script info: meshes(16) # of polygons(96) repeats(10) cycles(2)
[cpfw/voxel_space_create] Allocating 67600 voxels per node.
     Padded voxel space extends from (0, 0, 0) -> (104, 104, 100) (total size: 1081600).
     Wait while allocating...
[cpfw/voxel_space_create] Allocation done. Initializing MPI data types
[cpfw/voxel_space_create] MPI data types initialized.
[cpfw/script] Executing script...
[cpfw/script] ------------------------------------------------ repeat 1 done.
[cpfw/script] ------------------------------------------------ repeat 2 done.
[cpfw/script] ------------------------------------------------ repeat 3 done.
[cpfw/script] ------------------------------------------------ repeat 4 done.
[cpfw/script] ------------------------------------------------ repeat 5 done.
[cpfw/script] ------------------------------------------------ repeat 6 done.
[cpfw/script] ------------------------------------------------ repeat 7 done.
[cpfw/script] ------------------------------------------------ repeat 8 done.
[cpfw/script] ------------------------------------------------ repeat 9 done.
[cpfw/script] ------------------------------------------------ repeat 10 done.
[cpfw/script] Script execution done. Gathering benchmarks.

Benchmark results
-----------------
--- Add opertions.............................: 1120
--- Remove opertions..........................: 1440
--- Move opertions............................: 1440
--- Extract-All/compute opertions.............: 20
--- Slowest node took.........................: 2.661520  Per Repeat:  0.266152
--- Quickest node took........................: 2.515026  Per Repeat:  0.251503
--- Average load StDev over all cycles..........: 789.028954
[cpfw] Finalizing...
[cpfw] Terminate
```

The output can be interpreted as follows. The script we executed had two cycles and 10 repeats. 16 meshes are generated and two move-extract cycles are executed in each repeat.

The initial lines give the number of ADD-FACE, REMOVE-FACES, MOVE-VERTEX and EXTRACT-ALL/computation operations, respectively. Note that there will always be fewer ADD-FACE operations than REMOVE-FACES operations since REMOVE-FACES operate on vertices instead of faces.

The next two lines show how much time the slowest and quickest nodes took to complete all the operation. Only the time the actual operations take is included in this timing, and not the time for reading the script. Per-repeat times are also displayed.

Finally, the last line

```
--- Average load StDev over all cycles..........: 789.028954
```

gives that the standard deviation of the load, averaged over all the move-extract cycles — as computed in Equation 5.4 on page 89.

B.2 Voxelization Benchmarking Program

To compile the voxelization benchmarking program on Intel systems using OpenMPI, the OpenMPI C compiler must first be set to Intel's C compiler `icc`, as described in the previous section. Next, execute the commands

```
make clean
make -f Makefile.voxeltest
```

to compile the program. An executable named `voxeltest` will be generated.

Note that if the host machine has an older architecture than Intel's Core architecture, the compiler flags may have to be modified in order to create program that is able to run on the machine. In that case, remove the `-xT` compiler flag from `Makefile.voxeltest`.

As with `cpfw`, if running on Njord, use `Makefile.voxeltest.njord` instead of `Makefile.voxeltest`.

After the program has been compiled, the executable `voxeltest` may be used for voxelization benchmarks. The program is invoked by the command

```
./voxeltest
```

When run with no arguments, usage information is presented. The syntax for running the program is

```
./voxeltest <gpu-extract|cpu-extract> <threads>
    <v1x,v1y,v1z> <v2x,v2y,v2z> <v3x,v3y,v3z> [onetriangle]
```

The arguments are:

- `gpu-extract` or `cpu-extract`: State whether GPU or CPU voxelization algorithms will be used. Using GPU voxelization algorithms will only work if the `-DGPU_DISABLE` compiler flag was *not* set during compilation.
- `threads`: Specifies the number of OpenMP threads to use when voxelizing.
- `v1x,v1y,v1z`, `v2x,v2y,v2z` and `v3x,v3y,v3z`: Specify the x,y,z coordinates of the triangle from which 512 other triangles will be generated. The 512 triangles will be generated by varying the signs of the 9 specified values, giving $2^9 = 512$ different triangles.
- `onetriangle` specifies that instead of generating 512 different triangles, voxelization of one triangle is tested 512 times. This is useful in situations where triangles become larger than the GPU's texture size, in which case some of the 512 triangles generated will exceed the maximum texture extents.

An example run and example output[6], showing the wallclock, CPU and average, minimum and maximum triangle areas of the generated triangles, is shown below.

```
chrisl@linuxws ~ # ./voxeltest gpu-extract 1 15,12,5 10,5,2 4,3,6

Framework for Polygonal Structures on Clusters
Leif Christian Larsen <leifchl@idi.ntnu.no>

VOXELIZATION BENCHMARKING PROGRAM.

Statistics
==========
Used 1 threads, algorithm = gpu-extract
Total/per triangle wallclock time.........: 0.84956/0.00166
Total/per triangle CPU user time..........: 0.17201/0.00034
Total/per triangle CPU system time........: 0.38802/0.00076
Total/per triangle CPU user+sys time......: 0.56004/0.00109
Average/max/min triangle area ............:
```

[6]Parts of the output has been removed due to space constraints.

```
100.67522/192.60841/17.83255
```

Appendix C

Source Code Overview

In this appendix, we give an introduction to how the source code for the framework is structured. The purpose of this section is to be a starting point for further development of the framework or integrating the framework in an actual application.

C.1 Overview of Files

The framework is written in C99. Requirements and instructions for compiling the program are described in Appendix B. The individual source code files are thoroughly commented, so in this appendix, we provide only an overview of how the program works across the individual source code files. For technical details on each file, we refer to the source code.

The framework consists of the following files:

- `fw_main.c`: Handles the command-line interface of the benchmarking program and initialization. Also provides wrapper functions for allocating, reallocating and freeing memory.
- `fw_script.c`: Reads and executes scripts generated by `genscript`, and also does timing.
- `fw_splitpolygons.c`: Implements the three caching strategies for the EXTRACT-ALL operation and also implements the ADD-FACE, REMOVE-FACES and MOVE-VERTEX operations.
- `fw_loadbalancer.c`: Contains load-balancing methods. Functions in this file are called from `fw_splitpolygons.c` when EXTRACT-ALL operations are executed. This file also computes the standard deviation statistics.
- `fw_extract_cpu.c`: Contains the CPU implementation of the voxelization algorithm of Section 3.6. Also provides other functions used in the GPU algorithm; for example, the method of rotating triangles to the $z = 0$ plane of Section 3.6. Additionally, this file contains the `main` function of the voxelization benchmark program.
- `fw_extract_gpu.c`: Contains the GPU implementation of the voxelization algorithm of Section 3.6.
- `fw_lot.c`: Contains implementation of the Lapped Orthogonal Transform, described in Section E.4 on page 135.

- `fw_voxel.c`: Contains implementation for storing voxel values, both *native* voxels, which are voxels whose coordinates are in a node's coordinate space, and *cached* voxels, which are voxels copied from other nodes.
- `fw_voxelcachetransfer.c`: Implements MPI communication routines used in the Bounding Volume strategy of Chapter 3. Functions in this file are called from `fw_splitpolygons.c` when executing an EXTRACT-ALL operation.
- `fw_voxeltransfer.c`: Implements MPI communication routines used in the Minimum Communication and Block Transfer strategies of Chapter 3. Functions in this file are called from `fw_splitpolygons.c` when executing an EXTRACT-ALL operation.
- `fw_coord.c`: Implements *coordinates* and *coordinate lists*. Coordinate lists are used several places in the program. For example, the voxelization algorithms return the set of coordinates intersecting a triangle as a coordinate list. Coordinate lists are implemented using an array in addition to a hash table, which makes it efficient to check whether a coordinate is inside a list or not.
- `fw_vector.h`: Implements vector structures and vector operations. To facilitate inlining, all vector functions are marked as static and inlined[1], and are therefore only inside the header file. Operations such as vector addition, subtraction, dot product, cross products and rotating a point/vector about a quaternion are provided.
- `fw_quaternion.c`: Implements quaternions as described in Section E.2. Provides methods for creating and combining quaternions.
- `fw_wingededge.c`: Implements the Winged-Edge data structure as described in Chapter 2. Provides methods for adding, removing and moving polygons to the winged-edge structure.

In addition, the source code of the `genscript` is located in the file `genscript.c` file in the `scripts` folder in the source code directory.

C.2 Basic Execution Flow

Figure C.1 on the facing page illustrates the basic execution flow of the benchmarking program.

[1]Some compilers have problems with inlining nonstatic functions.

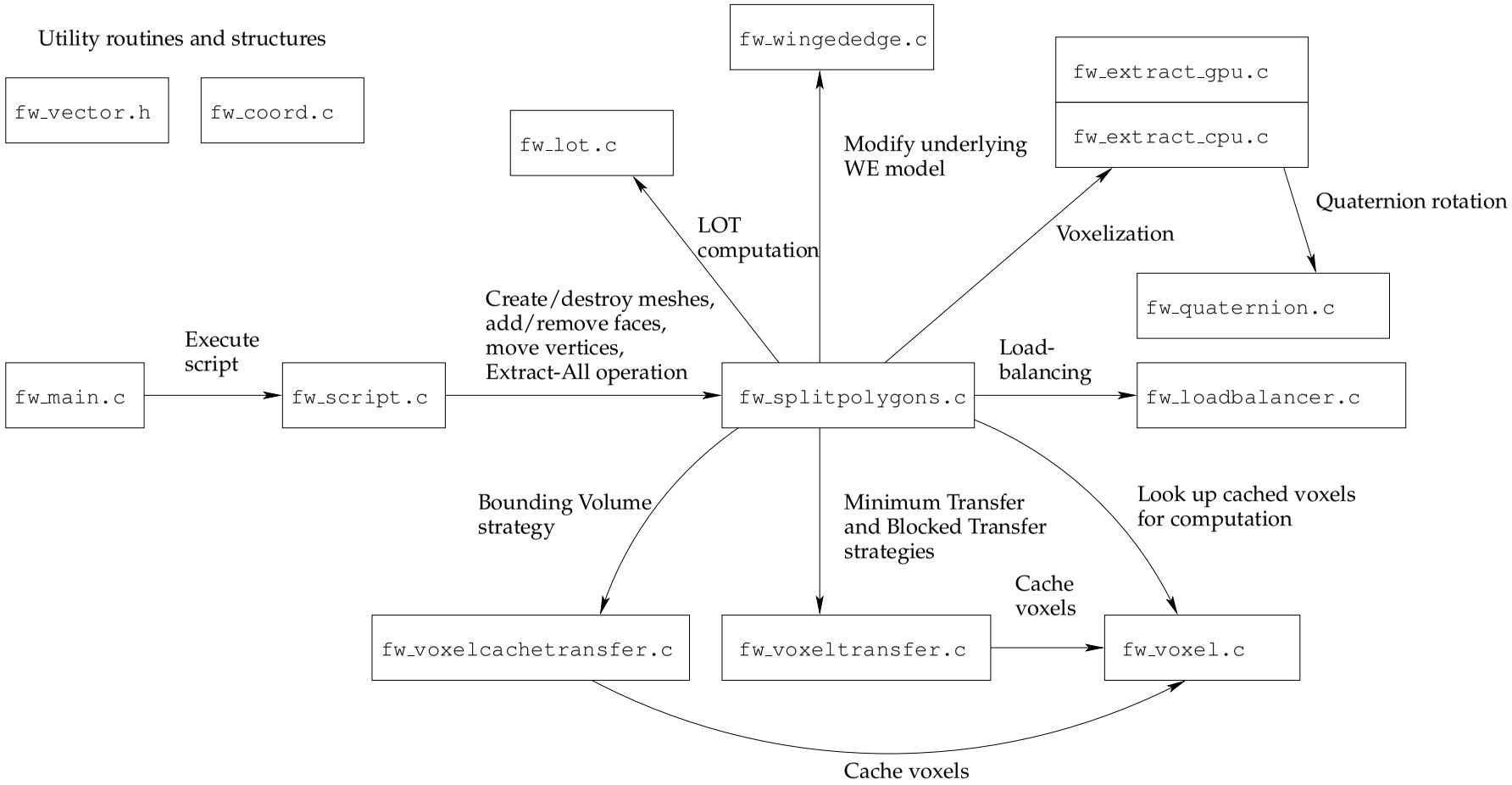


Figure C.1: Execution flow of `cpfw`

The program starts in `fw_main.c`. In this file, the program is initialized. Next, `fw_main.c` calls the `script_execute` function in `fw_script.c`, which executes the script file generated by `genscript`. The format of the script file is described in the source code of `genscript`.

A script is a sequence of instructions: CREATE-MESH, DESTROY-MESH, ADD-FACE, REMOVE-FACES, MOVE-VERTEX and EXTRACT-ALL. These operations are implemented in the file `fw_splitpolygons.c`, and are called from `fw_script.c` as the script file is read. `fw_splitpolygons.c`, in turn, calls routines in `fw_wingededge.c` to update the winged-edge polygon model, for example, if a polygon is added to a mesh, removed, or if a vertex is moved. When executing the EXTRACT-ALL operation, `fw_splitpolygons.c` calls routines in `fw_voxeltransfer.c` if using the Minimum Communication of Blocked Transfer strategies, or in `fw_voxelcachetransfer.c` if using the Bounding Volume strategy for actual MPI communication. Received voxels are inserted into a voxel cache, which is maintained by functions in `fw_voxel.c`. This file also stores the voxels in the node's native coordinate space (i.e. for all nodes $i \in N$, this file stores and generates the voxels in the coordinate space C_i).

If load-balancing is enabled, `fw_splitpolygons.c` calls routines in `fw_loadbalancer.c` when an EXTRACT-ALL operation is performed, before calling any MPI routines.

Voxelization is done by calling the voxelization routines in `fw_extract_cpu.c` and `fw_extract_gpu.c`. These files, again, use `fw_quaternion.c` to do rotations and compute rotation matrices.

LOT computation is done by calling routines in `fw_lot.c`.

Furthermore, the files `fw_vector.h` and `fw_coord.c` contain utility functions throughout the program. The file `fw_vector.h` contains vector arithmetic routines, and is used, for example, for vector arithmetic in the voxelization routines. `fw_coord.c` implements hashed coordinate lists, allowing quick lookups, in addition to routines for mapping coordinates to nodes in the Cartesian communicator. The routines of this file is used in connection with both voxelization, computation, caching and data transfer.

C.3 Futher Details

The source code is in the electronic attachment to this thesis and has been extensively commented. For further details, we therefore refer to the source code.

Appendix D

Other Data Structures for Polygon Meshes

We now briefly describe some alternative data structures to motivate why we chose the winged-edge structure.

Minimal/Spaghetti Representation

The *minimal/spaghetti representation* [69] stores polygon meshes as a set of faces, where each face consists of a set of edges, and each edge consists of two vertices. This structure has very low memory requirements, but there is no association between vertices and faces. Furthermore, the associations between faces and edges and edges and vertices are unidirectional. Finally, with each edge, no pointers to other edges are stored. This makes operations such as

- finding whether an edge or vertex belongs to a face or not — the time complexity is $O(|E|)$ and $O(|V|)$, respectively, where $|E|$ is the number edges and $|V|$ is the number of vertices in the mesh,
- traversing the entire mesh in such a way that each succeeding edge is connected to the previous edge, and
- modifying the mesh

difficult and time-consuming compared to the winged-edge structure.

Arc Node Representation

The *arc node representation*, introduced in 1975 by Puecker and Chrisman [70], is a simple extension to the minimal representation which store pointers to the faces of which an edge is a part of with each edge object. This makes it quicker ($O(1)$) to detect which faces an edge is a part of, but the rest of the operations we mentioned above are still as slow as with the minimal representation.

Doubly Connected Edge List (DCEL)

The *Doubly Connected Edge List (DCEL)* representation, introduced by Muller and Preparata in 1978 [71], is an extension of the arc node representation. Now, with each edge, we also

store pointers to the *next* and *previous* edges[1] of the edge. But the notion of next and previous is not unique: Consider Figure 2.7. When storing (d, b) in DCEL, are we referring to the next/previous clockwise or counterclockwise edges? To solve this ambiguity, in DCEL, we would store *two* edges: (d, b) and (b, d). With the first edge, the clockwise next/previous edges are stored, while with the second edge, counterclockwise next/previous edges are stored.

DCEL therefore uses more memory than the winged-edge structure, since two edges must be stored in the edge table, but in return for this sacrifice, it has the possibility of storing unidirectional edges — which is more interesting when using the structure for storing general graphs, and not important when storing polygon meshes.

The Half-Edge Representation

The *Half-Edge Representation*, introduced by Mantyla in 1988 [72], is a slight extension of DCEL. With each edge object in DCEL — known as a *half-edge* since the edge objects are unidirectional — a pointer to the edge in the opposite direction is also stored in the half-edge structure. Thus, the half-edge is advantageous if we require storing unidirectional edges, since the edge in the opposite direction can be obtained in $O(1)$ time versus possibly $O(|E|)$ for the DCEL structure, since finding the edge in the opposite direction might involve a scan of all the edge objects.

As for the DCEL, storing the directions of edges is of less importance to applications dealing with graphs to represent polygon meshes.

The Quad-Edge Structure

The *Quad-Edge Structure*, introduced by Guibas and Stolfi in 1985 [73], is based on the winged-edge structure. This structure has certain advantages over the winged-edge structure. In particular, the quad-edge structure allows for quick computation of the *dual* and mirror-imaged graphs[2] of the mesh. One problem where these properties will be useful is if we want to compute the minimum spanning tree of n points in the structure. Also, certain complex polyhedra (figures created from plane figures) may be constructed efficiently using dual structures [73]. The dual graph also has applications in polygon mesh compression [74].

The quad-edge representation replaces the two vertex and two face references of the edges in the winged-edge structure with four fields containing topological information to construct dual and mirror-imaged structures. Thus, the structure does not explicitly store polygon faces and vertices, but faces and vertices can be easily deduced from the other four fields [73].

For our purposes, it is more convenient to deal with faces and vertices. Additionally, the advantages of the quad-edge structure are irrelevant for our use, except for perhaps if we

[1] Note that there are many variants of the DCEL structure. Some authors use DCEL as a winged-edge structure without "previous" pointers.

[2] A *dual* of a planar graph (a graph with no intersections) $G = (V, E)$ is a graph $G_d = (V_d, E_d)$ where for each planar region (polygon) in G, there is a vertex $v \in V_d$, and an edge $e \in E_d$ between each two neighboring regions in G. The dual is a symmetric property, and thus, the dual of G_d is a dual G. The dual need not be unique.

require compressing the polygon mesh. However, in this case, we can compute the dual graph on the fly. This operation is unlikely to be done very often (only when, for instance, storing the mesh to disk).

Other Structures

Far more structures than we described above have been proposed; for example, the hybrid edge, bridge edge, split edge, vertex edge and face edge structures (see [75] for an overview). A grand data structure which generalizes all other representations, known as the Universal Data Structure (UDS)[3] has also been proposed [75]. The winged-edge structure is a suitable compromise for storage requirements, redundancy and complexity versus computational speed, which is likely why it retains its popularity even today, despite being over 30 years old. Although we use the winged-edge structure, our framework is, however, easily be adapted to use other representations, if required by an application. (Only the file `fw_wingededge.c` needs to be replaced).

[3]The name *universal data structure* has also been used in database research; however, the database UDS is something entirely different.

Appendix E

Additional Algorithms, Methods and Code

E.1 Line Drawing Algorithms

Our proposed solution for implementing the voxelization algorithm relies on being able to determine which pixels and voxels are intersected by a line. This process is called *rasterization*. Consider Figure E.1, which shows the idea in 2D.

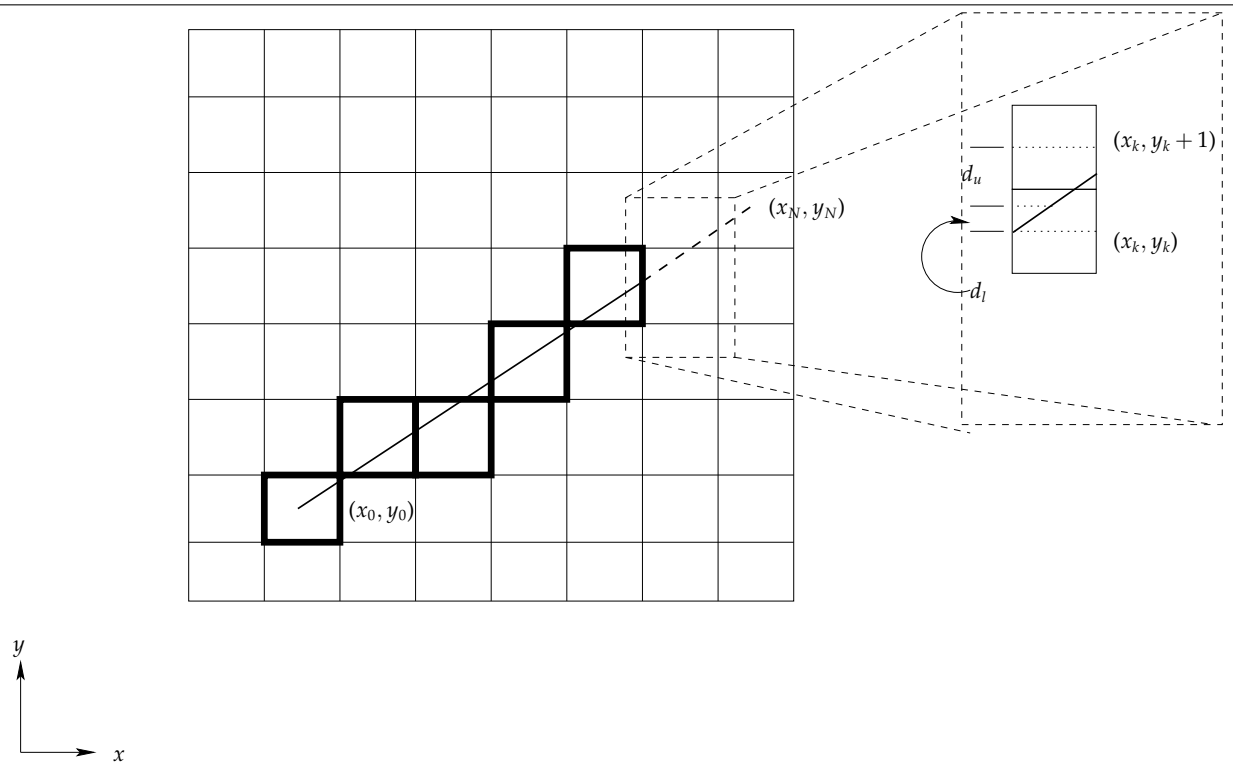


Figure E.1: Bresenham's algorithm

The most popular method for rasterizing lines is *Bresenham's line-drawing algorithm* [15]. Graphics Processing Units (GPUs) usually implement parallel variants of Bresenham's algorithm in hardware. Originally invented by IBM researcher Jack Bresenham in the 1970s,

one major advantage of Bresenham's algorithm is that no floating-point numbers are used. Also, the number of multiplications used is small compared to the number of additions, however, this argument is less significant today as addition and multiplication usually take the same number of clock cycles.

We now describe briefly how Bresenham's algorithm works. Our description is based on the one given in [15], with a few added details.

We are given a line from point (x_0, y_0) to point (x_N, y_N). For now, we assume that the line's slope coefficient m is such that $0 \leq m \leq 1$.

At step k of the algorithm, we can either plot a pixel at point (x_k, y_k) or at point $(x_k, y_k + 1)$. The decision on which pixel to plot depends on the computation of the two quantities

$$d_{u,k} = y_k + 1 - y \tag{E.1}$$
$$d_{l,k} = y - y_k \tag{E.2}$$

These quantities measure, respectively, the distance from pixels $(x_k, y_k + 1)$ and (x_k, y_k) to the mathematical line. y is computed by the well-known formula $y = mx + b$, which yields

$$d_{u,k} = y_k + 1 - (m(x_k + 1) + b) \tag{E.3}$$
$$d_{l,k} = (m(x_k + 1) + b) - y_k \tag{E.4}$$

Writing $\rho'_k = d_{l,k} - d_{u,k}$, we should thus plot pixel $(x_k + 1, y_k)$ if $\rho' < 0$ and $(x_k + 1, y_k + 1)$ otherwise (see Figure E.1). The expression for ρ' is

$$\rho'_k = 2m(x_k + 1) + 2b - 2y_k - 1 \tag{E.5}$$

Since $m = \Delta y / \Delta x = (y_N - y_0)/(x_N - x_0)$, we have

$$\rho'_k = \frac{2\Delta y}{\Delta x}(x_k + 1) + b - 2y_k - 1 \tag{E.6}$$

Letting $\rho_k = \rho'_k \Delta x$, we may still use the same decision criterion as before. By using ρ_k instead of ρ'_k, however, we avoid floating-point computations since all the components of the expression for ρ_k are integers:

$$\rho_k = 2x_k\Delta y - 2y_k\Delta x + (2\Delta y - \Delta x + 2b\Delta x) \tag{E.7}$$

Now comes the entire trick: We can obtain ρ_{k+1} from ρ_k:

$$\rho_{k+1} - \rho_k = 2\Delta y(x_k + 1 - x_k) - 2\Delta x(y_k + 1 - y_k) \tag{E.8}$$

Using $x_{k+1} - x_k = 1$ we have:

$$\rho_{k+1} = \rho_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \tag{E.9}$$

The expression $y_{k+1} - y_k$ is either zero or one, depending on which pixel was chosen in step k.

Initially, we have

$$\begin{aligned}
\rho_0 &= \Delta x(d_{l,0} - d_{u,0}) && \text{(E.10)}\\
&= \Delta x\left[((m(x_0+1)+b) - y_0) - (y_0 + 1 - (m(x_0+1)+b))\right] && \text{(E.11)}\\
&= \Delta x\left[((m(x_0+1)+b) - 2y_0 - 1 + (m(x_0+1)+b))\right] && \text{(E.12)}\\
&= \Delta x\left[2m(x_0+1) + 2b - 2y_0 - 1\right] && \text{(E.13)}\\
&= \Delta x\left[2\left(\frac{\Delta y}{\Delta x}(x_0+1) + b\right) - 2y_0 - 1\right] && \text{(E.14)}\\
&= 2\Delta y(x_0+1) + 2b\Delta x - 2y_0\Delta x - \Delta x && \text{(E.15)}
\end{aligned}$$

Inserting $b = y_0 - \frac{\Delta y}{\Delta x}x_0$, this reduces to

$$\rho_0 = 2\Delta y - \Delta x \tag{E.16}$$

This and Equation E.9 yields the following pseudocode for Bresenham's algorithm in 2D. Note that the pseudocode assumes $|\Delta x| \geq |\Delta y|$. If not, we can simply exchange x and y. Also, note that if $\Delta y < 0$, we must *decrement* y instead of incrementing it as we did above. The same goes for x.

BRESENHAM-2D(x_0, y_0, x_1, y_1)

```
 1  L ← ∅
 2  Δx ← x_1 − x_0
 3  Δy ← y_1 − y_0
 4  if Δx < 0
 5     then xstep ← −1
 6     else xstep ← 1
 7  if Δy < 0
 8     then ystep ← −1
 9     else ystep ← 1
10  ε ← 2|Δy| − |Δx|
11  x ← x_0, y ← y_0
12  for i ← 0 to |Δx|
13        do L ← L ∪ {(x, y)}
14           if ε > 0
15              then y ← y + ystep
16                   ε ← ε − 2|Δx|
17           x ← x + xstep
18           ε ← ε + 2|Δy|
19  return L
```

Lines 1–3 initializes variables used in the algorithm. Lines 4–7 check whether we should move in positive or negative directions in each dimension. Next, line 8 initializes the ϵ variable, which, in iteration i in the for-loop of line 10, corresponds to ρ_i. Line 9 initializes temporary x and y variables. Finally, the for-loop iterates $|\Delta x|$ times, adding one pixel on each iteration. If $\epsilon > 0$, we must step in the y-direction and update ϵ accordingly (see Equation E.9), which is done in lines 13–14. Finally, the x-value is updated. Note that ϵ is always computed according to Equation E.9, whether or not $\epsilon > 0$ or not. Finally, we return the resulting list of line pixels on line 19.

The algorithm is easily extended to do 3D line voxelization. In that case, we add another parameter $\rho_{z,k}$, which is computed precisely as ρ_k except using z instead of y-values, for deciding the z-value of each voxel. See the accompanying source code for an implementation.

E.2 Quaternions

We now briefly describe how to use *quaternions* to do 3D rotation. For further details, consult e.g. [76]. Quaternions are frequently used to do 3D rotation in computer graphics, since using quaternions makes it easy *and* arithmetically efficient to:

1. Rotate a point about an arbitrary line.
2. Combine several rotation operations into one rotation operation

In addition, quaternions also require less storage space than alternative methods and are numerically more stable than other approaches [1] [76, 77]. Also, as we will see, inverting a rotation using quaternions is efficient.

[1] An alternative approach is the *axis-angle approach*, where a rotation about an arbitrary axis is represented as a rotation about the x,y and z-axes.

A quaternion q is defined as a vector $\mathbf{v} = (v_x, v_y, v_z)$ and a scalar s:

$$q = (\mathbf{v}, s) \tag{E.17}$$

q represents a counter-clockwise rotation s radians about $\mathbf{v}$. Quaternions can be viewed as extensions of complex numbers: We could write $q = s + v_x i + v_y j + v_z k$. See Figure E.2.

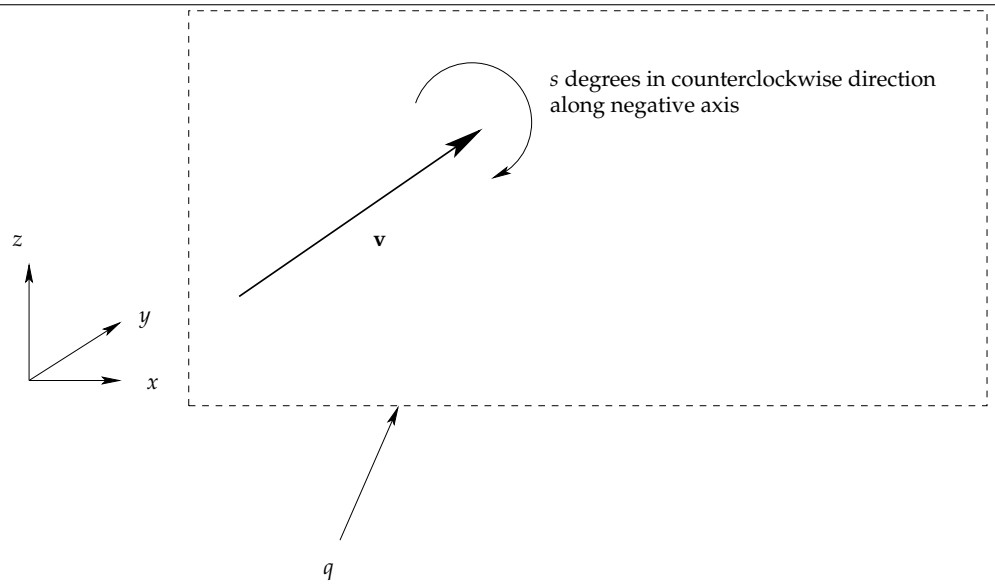


Figure E.2: A quaternion $q = (\mathbf{v}, s)$

From q, we may obtain the corresponding rotation matrix by the following equation (see e.g. [78] for a derivation).

$$\begin{bmatrix} s^2 + v_x^2 - v_y^2 - v_z^2 & 2v_xv_y - 2sv_z & 2sv_y + 2v_xv_z & 0 \\ 2sv_z + 2v_xv_y & s^2 - v_x^2 + v_y^2 - v_z^2 & 2v_yv_z - 2sv_x & 0 \\ 2v_xv_z - 2sv_y & 2sv_x + 2v_yv_z & s^2 - v_x^2 - v_y^2 + v_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{E.18}$$

Multiplying the rotation matrix by any point $\mathbf{x} = (x, y, z, 1)$ will yield rotated point coordinates[2].

To obtain the inverse rotation, we could invert the matrix in Equation E.18. However, note that the inverse quaternion of q is simply

$$q = (-\mathbf{v}, s) \tag{E.19}$$

In terms of arithmetic complexity, it will therefore be quicker to obtain the inverse quaternion by Equation E.19 and obtain a new rotation matrix by Equation E.18, rather than inverting the rotation matrix of q directly.

Finally, we may *combine* several quaternions into one quaternion. Let q_1 and q_2 be quaternions. Then, the combined quaternion q_C is given by

[2]We use a 4×4 matrix so that translations may be incorporated into the same matrix (see Equation 3.35)

$$
\begin{aligned}
q_C &= q_1 * q_2 && \text{(E.20)}\\
&= (\mathbf{v}_1, s_1) * (\mathbf{v}_2, s_2) && \text{(E.21)}\\
&= (s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2, s_1 s_2 - \mathbf{v}_1 \bullet \mathbf{v}_2) && \text{(E.22)}
\end{aligned}
$$

Here, $\times$ denotes the vector cross-product and $\bullet$ denotes the vector scalar product. Note that had we used regular rotation matrices to combine two rotations, we would have to do computations equivalent of a 3×3 matrix-matrix multiply — which is a computationally more expensive operation than computing Equation E.22.

E.3 Additional Voxelization Pseudocode

E.3.1 Transforming A Triangle To $z = 0$

A routine which computes a quaternion for doing the initial translation and the rotations represented by q_1, q_2 and q_3 (or, equivalently, the rotation matrices R_1, R_2 and R_3) is shown below in the procedure TRANSFORM-TRIANGLE-TO-XY-PLANE. The routine returns transformed vertices, along with a quaternion which represents the rotation and a translation which represents the initial translation to the origin (Equation 3.31).

TRANSFORM-TRIANGLE-TO-XY-PLANE(v_1, v_2, v_3)

1 $translation \leftarrow v_1$
2 $v_1^{(1)} \leftarrow v_1 - v_1$
3 $v_2^{(1)} \leftarrow v_2 - v_1$
4 $v_3^{(1)} \leftarrow v_3 - v_1$
5 $\triangleright$ q_1 is the first quaternion, α' is computed as in Equation 3.38
  $\triangleright$ The ROTATE operation is obtained by multiplying a vertex with a
  $\triangleright$ quaternion's rotation matrix (see Appendix E.2)
6 $q_1 \leftarrow quaternion((-v_{3,y}, v_{3,x}, 0), \alpha')$
7 $v_2^{(2)} \leftarrow \text{ROTATE}(q_1, v_2^{(1)}), v_3^{(2)} \leftarrow \text{ROTATE}(q_1, v_3^{(1)})$
8 $\triangleright$ q_2 is the second quaternion, β' is computed as in Equation 3.44
9 $q_2 \leftarrow quaternion((0, 0, 1), \beta')$
10 $v_2^{(3)} \leftarrow \text{ROTATE}(q_2, v_2^{(2)}), v_3^{(3)} \leftarrow \text{ROTATE}(q_2, v_3^{(2)})$
11 $\triangleright$ q_3 is the third quaternion, γ' is computed as in Equation 3.47
12 $q_2 \leftarrow quaternion((1, 0, 0), \gamma')$
13 $v_2' \leftarrow \text{ROTATE}(q_3, v_2^{(3)})$
14 $v_1' \leftarrow v_3^{(1)}, v_3' \leftarrow v_3^{(3)}$
15 $\triangleright$ q is the combined quaternion of q_1, q_2, q_3 (see Appendix E.2)
16 $q \leftarrow \text{COMBINE-QUATERNIONS}(q_1, q_2, q_3)$
17 **return** $\{translation, q, v_1', v_2', v_3'\}$

E.3.2 The Unoptimized VOXELIZE Algorithm

VOXELIZE-FACE(v_1, v_2, v_3)

```
1  ▷ We assume that (v1, v3) is the longest edge of the triangle
2  translation, q, v1, v2, v3 ← TRANSFORM-TRIANGLE-TO-XY-PLANE(v1, v2, v3)
3  q^-1 ← INVERT-QUATERNION(q)
4  ▷ The following two lines call a line drawing algorithm, such as Bresenham's algorithm
   ▷ See Appendix E.1 for further details.
5  L1 ← VOXELIZE-LINE(v1, v2)
6  L2 ← VOXELIZE-LINE(v2, v3)
7  ▷ Now, we can extract the voxels.
8  L ← ∅
9  for x ← 0 to v3,x
10       do for y ← 0 to v2,y
11            do if (x, y) ∈ L1 ∨ (x, y) ∈ L2
12                 then break
13          c ← ROTATE(q^-1, (x, y, 0)) + translation
14          L ← L ∪ {c}
15 return L
```

Line 2 transforms the triangle to the $z = 0$ plane, and obtains the quaternion and translation vector. Line 3 inverts the combined quaternion representing the rotations, which is equivalent to inverting all the rotations done to rotate the triangle to the $z = 0$ plane. Next, we extract the points of the edges (v_1, v_2) and (v_2, v_3). We can use any algorithm for converting a mathematical line to a set of pixels, such as Bresenham's line-drawing algorithm, to do this operation. We described Bresenham's algorithm in Appendix E.1.

After this, we iterate over all the x-coordinates of the triangle. For each x-coordinate, we add coordinates to the final coordinate list L until we have reached a pixel of either edge (v_1, v_2) or (v_2, v_3). The coordinate lists L_1 and L_2 may be implemented using a hash table, giving $O(1)$ expected running time for the search [22]. The entire branch in the inner loop can, however, be optimized away, as discussed in Section 3.6.3 and as implemented in Section E.3.3.

E.3.3 The Optimized VOXELIZE Algorithm Without Inner-Loop Branches

VOXELIZE-CPU-OPTIMIZED(v_1, v_2, v_3)

1 ▷ We assume that (v_1, v_3) is the longest edge of the triangle
2 *translation*, $q, v_1, v_2, v_3 \leftarrow$ TRANSFORM-TRIANGLE-TO-XY-PLANE(v_1, v_2, v_3)
3 $q^{-1} \leftarrow$ INVERT-QUATERNION(q)
4 $L_1 \leftarrow$ VOXELIZE-LINE(v_1, v_2)
5 $L_2 \leftarrow$ VOXELIZE-LINE(v_2, v_3)
6 $L_c \leftarrow$ CONSTRUCT-LC-ARRAY(L_1, L_2)
7 $L \leftarrow \emptyset$
8 **for** $x \leftarrow 0$ **to** $v_{3,x}$ **in parallel**
9 **do for** $y \leftarrow 0$ **to** $L_c[x] - 1$
10 $c \leftarrow$ ROTATE$(q^{-1}, (x, y, 0)) +$ *translation*
11 $L \leftarrow L \cup \{c\}$
12 **return** L

E.3.4 The GPU Fragment Voxelization Program

```
/* Fragment program does inverse rotations by 4x4 matrix-vector multiplication
 */

void extract_fp(
      in   float2     tc0 : TEXCOORD0,
      in   float4     pos : POSITION,

      out float4      col0  : COLOR,

      uniform float4x4  rotmtx,
      uniform float2    minXY)
{
  float4 tmpvec;

  /* On the GPU, we do one additional translation to ensure that all coordinates
   * are (1) positive and (2) most likely fit the GPU texture size.
   * Therefore we translate by the negative of the minimum x- and y-values of
   * all triangle vertices. Therefore, we must translate back prior
   * to doing the matrix multiplication.
   */

  tmpvec.xy = floor(tc0) + minXY;
  tmpvec.zw = float2(0, 1);

  /* 4x4 matrix-vector multiplication, rounded to integers  */

  col0 = round(mul(rotmtx, tmpvec));
}
```

Listing E.1: Cg fragment processor program for voxelization

E.4 The Lapped Orthogonal Transform

In this section, we give a brief introduction to the *lapped orthogonal transform*, or LOT, which we have used as one of the computations done after executing the EXTRACT-ALL operation. For more details about the LOT, consult [56, 27].

LOT, which was invented in 1988 by H. S. Malvar, is used for a compression technique called *transform coding compression*. This technique has the following main steps:

1. First, a mathematical transform, such as the LOT or the Discrete Cosine Transform (DCT) is applied on blocks of size k. In the JPEG standard for image compression, $k = 8$.
2. Next, the transformed values are *quantized*. For example, the DCT transforms coefficients into frequency space. High-frequency coefficients correspond to details in the data, while low-frequency coefficients correspond to slowly varying information. By setting the DCT coefficients which represent higher frequencies to zero, some detail is lost, but long sequences of zeros or very small values in the stream of coefficients are produced.
3. Since the previous step produces long ranges of zeros, techniques such as Huffman and run-length encoding can be applied in order to compress the transformed coefficients.

To decompress the data, the steps are reversed, and the inverse mathematical transform is computed instead of the forward transform.

The amount of details lost depends on how much data is removed by the quantization step during the forward transform.

One problem with the DCT is that blocks are transformed individually. This leads to so-called *blocking effects* in the data after decompression, which leads to noticeable boundaries between the blocks. The *lapped orthogonal transform* attempts to reduce the blocking effects by combining the transform coefficients of each block with the previous and next blocks. This has advantages for seismic data, where LOT-based methods are considered to be superior [57].

The LOT is computed as follows: First, the Discrete Cosine Transform (DCT) is applied:

$$Y_f = \sqrt{\frac{2}{n}} C_f \sum_{t=0}^{n-1} x_t \cos\left[\frac{(2t+1)f\pi}{2n}\right] \tag{E.23}$$

where

$$C_f = \begin{cases} 1/\sqrt{2} & \text{for } f = 0 \\ 1 & \text{for } f > 0 \end{cases} \tag{E.24}$$

In the case of blocked transform, $n = 8$ and Equation E.23 is applied to each block. Efficient algorithms, such as the AAN and Feig-Linzer multiply-add algorithms [79, 80], exist for computing the DCT with block size equal to 8. These methods are derived from the Fast Fourer Transform (FFT) by Cooley and Tukey [81].

Subsequent to computing the DCT, neighboring blocks are merged. Consider Figure E.3, which shows how the LOT algorithm for block size 8 works. First, DCT is computed for each block. Next, two *butterflies* merge neighboring blocks in order to reduce blocking effects. In the graph, black nodes indicate addition and arrows indicate negation. After the second butterfly, all coefficient are multiplied by 0.5 and then multiplied by the $\widetilde{\mathbf{Z}}$-matrix, which is a series of three 2D rotation transformations. The optimal angles to use in the rotations, as well as more details on the $\widetilde{\mathbf{Z}}$ matrix, can be found in [56] or [27].

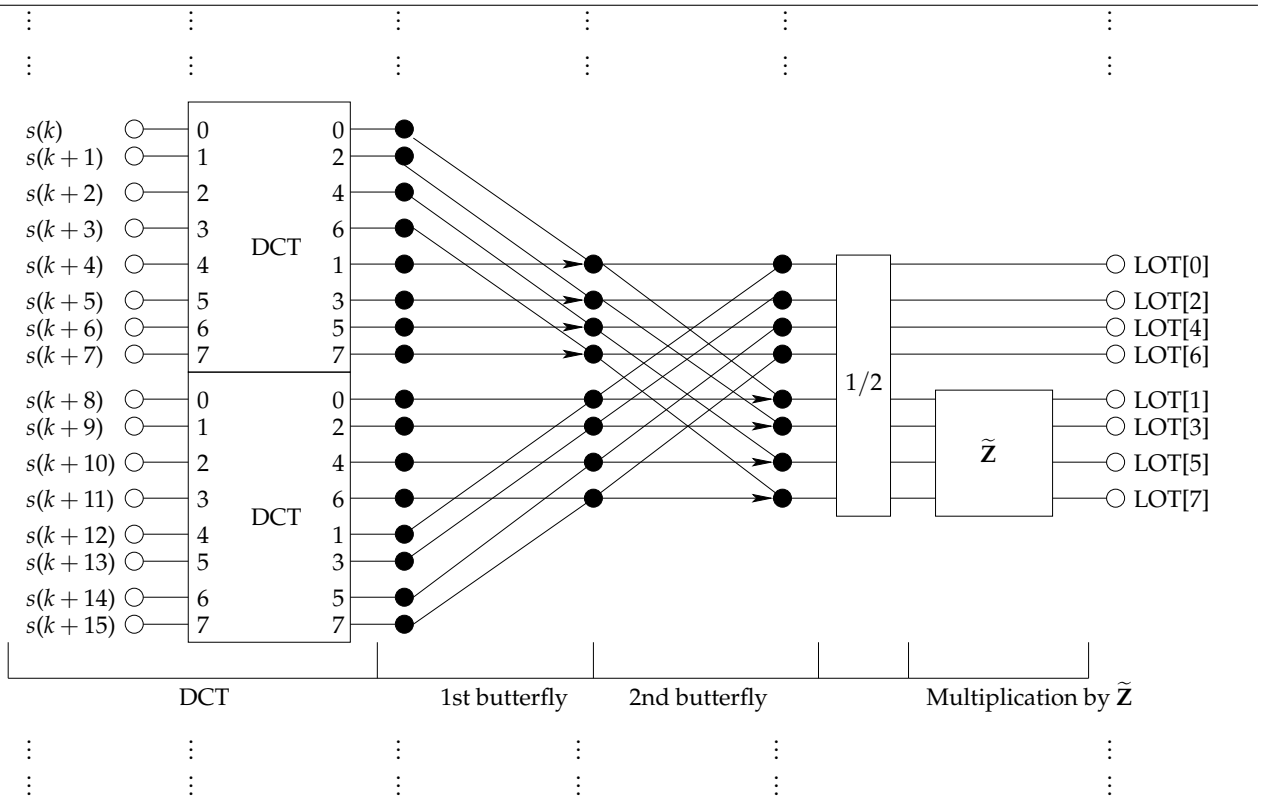


Figure E.3: Flowgraph for the fast type-I LOT algorithm, with block size 8, adapted from Malvar's original paper on the LOT [56].

In order to compute the first and last blocks, we may add padding blocks filled with zeros, which requires storing one additional block for reconstructing the data with no artifacts. Alternatively, we may add blocks which mirror the first and last blocks. This requires no additional storage, since the coefficients of the mirror blocks are given implicitly. See [82] for further details.

Appendix F

Additional Benchmarks

In this Appendix, we provide additional benchmarks. Section F.3 describe additional voxelization benchmarks, while Section F.1 describe additional framework benchmarks.

F.1 Additional Framework Benchmarks

Tables F.1, F.2 and F.3 show the wallclock times of Benchmark 1, 2 and 3, respectively, of Section 5.2.

F.2 Additional Load-Balancing Benchmarks

Table F.4 shows the standard deviation obtained in the different load-balancing strategies.

F.3 Additional Voxelization Benchmarks

Table F.5 shows the wallclock obtained in the voxelization benchmark. Table F.6 shows the CPU times.

CPUs / **Algorithm**	1	2	4	8	16	32	64	128	256
Serial, 1-cpu	36.39	—	—	—	—	—	—	—	—
MC-nolb	—	43.83	35.49	**32.75**	24.14	15.24	**8.67**	**5.84**	**3.29**
BT-nolb	—	53.54	45.19	53.87	38.22	26.76	16.29	11.84	6.88
BV-nolb	—	**42.74**	**35.29**	35.75	**23.68**	**14.67**	32.09	59.55	25.33
MC-globallb	—	42.92	**22.46**	***13.58***	**6.65**	**4.20**	**2.52**	**1.65**	**1.08**
BT-globallb	—	53.07	45.37	30.95	14.61	9.80	6.16	4.39	2.51
BV-globallb	—	**42.58**	26.81	18.94	11.90	6.62	13.30	16.23	7.63
MC-locallb	—	44.38	***20.81***	**14.16**	***6.26***	***3.37***	***1.83***	***1.28***	***1.00***
BT-locallb	—	83.61	41.82	31.81	13.53	7.76	4.47	3.30	2.31
BV-locallb	—	**42.85**	26.71	20.56	11.68	5.79	15.46	10.34	6.59
MC-mhlb-∞	—	43.10	**21.08**	**13.62**	**6.71**	**4.26**	**2.52**	**1.64**	**1.06**
BT-mhlb-∞	—	53.10	42.73	31.44	12.90	7.97	5.15	3.82	2.42
BV-mhlb-∞	—	**42.65**	26.23	18.62	11.89	6.67	13.59	16.48	7.83
MC-mhlb-2	—	43.34	**21.32**	**14.37**	**14.54**	13.17	**8.99**	**5.77**	**3.29**
BT-mhlb-2	—	53.31	42.97	31.07	23.50	23.54	16.99	11.79	6.93
BV-mhlb-2	—	***41.98***	24.87	21.56	15.05	**13.10**	25.07	57.73	25.01

Table F.1: Benchmark 1 on Njord. Boldface entries indicate the best load-balancing algorithm within a single caching strategy for a specific number of CPUs, while boldface italic values denote the best time for a specific number of CPUs.

Algorithm \ CPUs	1	2	4	8	16	32	64	128	256
Serial, 1-cpu	86.99	—	—	—	—	—	—	—	—
MC-nolb	—	**122.20**	119.63	**83.27**	**63.82**	**60.92**	**44.77**	**26.18**	**16.71**
BT-nolb	—	133.40	128.34	104.17	83.21	100.12	73.29	45.28	30.20
BV-nolb	—	126.00	**114.96**	91.71	72.22	77.45	50.99	130.09	190.45
MC-globallb	—	**287.70**	**103.25**	***39.84***	***18.80***	**16.16**	***12.13***	**7.44**	**4.81**
BT-globallb	—	622.31	217.24	86.52	47.21	35.05	21.77	13.91	9.58
BV-globallb	—	450.39	156.84	79.86	54.87	42.59	108.20	43.74	52.98
MC-locallb	—	***109.87***	100.05	**97.88**	**44.30**	**29.48**	**13.03**	***6.10***	***3.56***
BT-locallb	—	112.99	***99.46***	177.12	87.90	63.94	27.91	14.11	8.48
BV-locallb	—	124.95	121.27	130.54	65.85	48.57	76.36	98.93	40.08
MC-mhlb-∞	—	**286.56**	**109.26**	**39.92**	**19.20**	***15.74***	**12.24**	**7.42**	**4.77**
BT-mhlb-∞	—	618.33	208.58	86.30	38.96	28.19	21.08	13.79	9.62
BV-mhlb-∞	—	450.41	157.70	79.92	52.56	43.26	100.73	44.85	52.67
MC-mhlb-2	—	**280.49**	**102.25**	**43.43**	**49.15**	**57.04**	**44.79**	**26.41**	**16.70**
BT-mhlb-2	—	627.25	211.31	79.95	64.73	94.44	73.74	45.62	30.82
BV-mhlb-2	—	445.73	140.89	82.27	62.85	73.67	68.68	115.92	29.63

Table F.2: Benchmark 2 on Njord. Boldface entries indicate the best load-balancing algorithm within a single caching strategy for a specific number of CPUs, while boldface italic values denote the best time for a specific number of CPUs.

Algorithm	CPUs	1	2	4	8	16	32	64	128	256
Serial, 1-cpu		133.57	—	—	—	—	—	—	—	—
MC-nolb		—	**191.90**	**190.64**	**181.05**	**169.92**	**125.31**	**88.21**	**85.75**	**67.04**
BT-nolb		—	199.87	198.03	196.39	191.37	153.97	116.12	138.28	110.95
BV-nolb		—	227.01	226.87	223.31	204.89	176.81	126.49	499.26	>750
MC-globallb		—	**493.34**	**215.86**	***87.55***	**54.91**	**37.54**	**25.98**	**22.53**	**18.52**
BT-globallb		—	1154.89	534.35	208.59	131.52	52.71	34.46	36.85	31.03
BV-globallb		—	1599.07	864.49	381.64	297.55	176.43	481.23	509.44	183.32
MC-locallb		—	***190.25***	188.14	**160.80**	**145.04**	**153.54**	**64.14**	**39.62**	**18.95**
BT-locallb		—	190.99	***187.46***	163.97	146.32	281.08	123.25	85.64	40.29
BV-locallb		—	285.72	277.87	302.69	246.95	266.33	138.18	581.77	455.56
MC-mhlb-∞		—	**498.49**	**220.99**	**89.34**	***54.34***	***37.30***	**25.79**	**22.90**	***18.46***
BT-mhlb-∞		—	1155.67	524.71	203.60	106.46	51.87	34.92	36.82	30.84
BV-mhlb-∞		—	1613.38	825.17	393.35	301.93	169.23	476.98	512.50	183.72
MC-mhlb-2		—	**499.85**	**219.66**	**102.25**	**132.83**	**112.56**	**87.98**	**83.49**	**66.35**
MC-mhlb-2		—	1141.40	527.74	197.18	153.84	140.28	115.83	136.65	110.05
MH-mhlb-2		—	1585.46	828.54	368.57	225.41	176.42	188.80	547.48	>750

Table F.3: Benchmark 3 on Njord. Boldface entries indicate the best load-balancing algorithm within a single caching strategy for a specific number of CPUs, while boldface italic values denote the best time for a specific number of CPUs.

CPUs / LB Algorithm	1	2	4	8	16	32	64	128	256
None	0	4,035,741	2,795,743	1,409,932	764,145	440,201	255,620	134,062	70,636
Global	0	**59,952**	**54,850**	**138,740**	**121,864**	**81,925**	**53,437**	**28,854**	**15,577**
Local	0	3,221,577	2,259,919	1,062,365	520,218	264,423	136,158	68,542	34,525
Manhattan, ∞	0	**59,952**	54,951	161,597	139,445	90,575	58,135	31,003	16,763
Manhattan, 2	0	**59,952**	54,951	549,288	536,090	383,139	245,467	129,715	68,884

Table F.4: Average standard deviation of the load in Benchmark 2 with different load-balancing algorithms. Boldface entries indicate the lowest standard deviation for a specific number of CPUs.

Machine/Algorithm \ Avg. area	10	100	1000	10000	10^5	10^6	$0.5 \cdot 10^7$
P4: gpu-voxelize	1.30	1.55	1.62	1.81	**7.80**	**75.55**	**346.81**
P4: cpu-voxelize-1cpu	**0.33**	**0.34**	**0.44**	**1.28**	10.44	161.32	582.42
Xeon: gpu-voxelize	0.66	0.68	0.80	2.26	8.25	**64.51**	**290.10**
Xeon: cpu-voxelize-1cpu	**<0.01**	**0.01**	**0.06**	**0.58**	**6.38**	84.98	316.79
Xeon: cpu-voxelize-2cpu	0.01	0.02	0.09	0.61	7.57	108.01	457.19
Xeon: cpu-voxelize-4cpu	0.04	0.06	0.11	0.64	7.72	106.56	411.68
Njord: cpu-voxelize-1cpu	**0.02**	**0.05**	0.28	2.29	24.74	269.24	972.51
Njord: cpu-voxelize-2cpu	0.02	0.05	**0.22**	1.82	17.71	225.05	720.10
Njord: cpu-voxelize-4cpu	0.03	0.05	0.19	1.49	14.67	167.41	635.16
Njord: cpu-voxelize-8cpu	0.04	0.06	0.17	**1.18**	**11.17**	**139.12**	**503.32**
Njord: cpu-voxelize-16cpu	0.06	0.08	0.20	1.19	11.67	150.66	553.74
Njord: cpu-voxelize-32cpu	0.08	0.11	0.30	1.71	17.94	208.21	824.46

Table F.5: Wallclock times (in seconds) for voxelization of 512 triangles. Boldface values represent the quickest algorithm within a machine. Note that the average area represents *approximately* the average area of the 512 voxelized triangles. Each run is the average of 20 runs.

Machine/Algorithm \ Avg. area	10	100	1000	10000	10^5	10^6	10^7
P4: gpu-voxelize	**1.10**	**1.10**	**1.11**	**1.54**	**7.09**	**73.54**	**338.18**
P4: cpu-voxelize-1cpu	0.33	0.34	0.44	1.28	10.36	158.40	582.42
Xeon: gpu-voxelize	**0.42**	**0.42**	**0.49**	**1.51**	**6.86**	**56.11**	**289.54**
Xeon: cpu-voxelize-1cpu	<0.01	0.01	0.06	0.58	6.37	84.97	316.78

Table F.6: CPU times (user and system time, in seconds) for voxelization of 512 triangles in the runs in Table F.5. Boldface entries represent values substantially different from the wallclock times.