# 3D cache-oblivious multi-scale traversals of meshes using 8-reptile polyhedra

van der Plas, G.A.J.

*Award date:*
2016

# 3D cache-oblivious multi-scale traversals of meshes using 8-reptile polyhedra

BSc. G.A.J. van der Plas

Supervisor: Dr. H. Haverkort

May 18, 2016

this page is intentionally left blank

**Abstract**

In the field of numerical simulation, a multi-scale grid and its traversal can have a huge impact on the cache-efficiency of the calculations performed. We present the Haverkort element traversal and refinement scheme which is based on the bisection of 8-reptile tetrahedra. We developed a stack-assignment scheme that allows the Haverkort traversal to use stacks for storing the input data, the output data, and the temporary data of vertices and elements. Our parallel plane approach to assign stacks to vertices gave solution that requires at most 8 stacks to store temporary vertex data for uniform and multi-scale refined grids independent of the refinement level. Furthermore 8 is also the lower bound for the number of stacks for our parallel plane approach. Additionally we show that a general lower bound exists of 5 temporary stacks for storing the vertex data during a traversal.

The combination of the Haverkort traversal with our Constant Stack algorithm is cache-oblivious, suitable for multi-level cache hierarchies and maintains its performance for multi-level refined grids and allows for fast and efficient space-filling-curve-based (re)partitioning. The Constant Stack algorithm has a time complexity of $O(1)$ for stack-assignment and -access and can achieve a cache-miss ratio of less than 5.2% .For all of these reasons we expect that a constant-number-of-stacks solution can compete with and outperform numerical simulations using cache-optimization techniques based on loop blocking when running on CPUs or GPUs. We apply the approach found for obtaining a constant-number-of-stacks solution for the Haverkort element traversal and a refinement scheme was applied to two other suitable 8-reptile polyhedra. Traversal and refinement schemes are given for an 8-reptile bisected cube and an 8-reptile bisected triangular prism needing 9 and 7 stacks respectively.

this page is intentionally left blank

# Contents

# Chapter 1

# Introduction

In the field of fluid dynamics, numerical simulation is a common approach for finding an approximate solution of a partial differential equation problem. One would like to quicken the simulations by optimizing the use of available computational resources and in some cases distribute the required computational effort and resources over multiple computing units. A recent approach to do so is basing the partitioning and traversing of a grid on a space-filling curve [1]. A space-filling curve can be used for the recursive subdivision of a grid cell into smaller grid cells of similar shape.

A space-filing-curve-based traversal can keep the communication volume between partitions distributed over different computing units small. A space-filling-curve-based traversal can efficiently use the cache of a computing unit by storing all temporary grid data in a stack. Therefore, it is a very fast and low-memory-requiring algorithm. From data reported by Schamberger and Wierum [2], Dennis [3], and Harlacher et al. [4] it is expected that such approaches can challenge and possibly outperform more traditional grid traversals and partitioning techniques.

At least three space-filling-curve-based grid traversals exist for two-dimensional grids that require only a constant number of stacks [1]. The only three-dimensional grid solution found in literature is the one by Weinzierl and Mehl [5] which dissects a cube into 27 subcubes and requires 6 temporary stacks. In chapter 2 and 4 we present three new tree-dimensional traversals with grid cells that have the shape of tetrahedra, cubes and triangular prisms. These new solutions are based on bisection, allowing a more fine-grained control over the grid refinement than the solution by Weinzierl and Mehl. The solutions require 8, 9, and 7 temporary stacks respectively.

## 1.1 Numerical simulations

Let's consider a numerical simulation where we want to determine the speed and pressure of water flowing through a pipe. Here the partial differential equations model the behaviour of the water. The problem itself is defined by its boundary conditions, like the pipe-shape and the pressure at the input and output of the pipe. Together the partial differential equations and the boundary conditions determine the solution: the

speed and volume of water flowing through the pipe at every position and moment in time.

The calculation of the flow of water through a pipe is a simple case at low velocities. The simple pipe-case can easily be solved analytically by hand. It becomes rapidly more complex at higher speeds when turbulence occurs. Some problems are even too complex to model analytically. For example, the problem of solving the dynamic airflow around a flying air-plane or predicting the weather patterns on earth. In such cases, a researcher is forced to be satisfied with an approximate solution obtained through numerical simulation.

An analytical solution to a partial differential equations problem yields a solution that is continuous in the spatial and temporal domain. In numerical simulation one discretizes the spatial and temporal domain. The spatial domain is divided into a grid or mesh of small elements and the temporal domain is sliced into small discrete time steps. Each element has a state: a number of properties that describe the flow in the element at some time step. One can calculate the state of each element in the next time step from the state of the elements in earlier time steps.

In the example of the pipe flow, one can approximate the flow in each of the tiny elements individually for a single small time step. In its simplest form the approximation in an element is done using linearised equations derived from the partial differential equations. If we use linear equations for approximating the flow field, water following a curved path is modelled in a grid cell with a straight path. The smaller the grid cell, the smaller the discrepancy between an analytical curved path and a path constructed of small linear segments. If the approximation of each small element is sufficiently accurate, the simulation of the whole pipe flow can be sufficiently accurate for a sequence of time steps.

In order to keep the approximation sufficiently accurate, while keeping the number of computations over the full simulation period as small as possible, an adaptive grid refinement technique can be used. When the local flow becomes more detailed, one replaces one or more grid elements with a larger number of smaller elements. In other words, one locally refines the grid. If the local flow becomes of a more uniform nature, one merges several smaller grid elements to one: the grid is coarsened locally. If the changes between time steps become too large and the accuracy of the simulation degrades, one would make the time step smaller. If the changes become small enough one can increase the time step again.

## 1.2 Partitioning of the numerical domain

For simple numerical problems such as the pipe flow a small desktop computer suffices. However with growing complexity of the partial differential equations and the boundary conditions the computation time required for a flow simulation can increase substantially. A climate simulation can easily require enormous computational resources and take decades or centuries to complete on a desktop computer from 2015. One needs an efficient way to distribute these computational and resource requirements over mul-

tiple computing units to quicken such simulations. The speed of data-access inside a computational unit is orders of magnitude faster than the speed of data-exchange between computational units. Therefore, the goal of partitioning is to divide the numerical domain into equal amounts of computational work while keeping the required data-exchange between the partitions as small as possible. Such a distribution can be achieved by partitioning the grid or mesh containing the elements wisely. Each partition is assigned to a computational unit in a network.

Multi-level graph partitioning is a well-known known approach for the partitioning and repartitioning of graphs, finite element meshes, and grids, see Karypis and Kumar [6]. Several software packages like METIS, Jostle, Chaco, Party and Scotch exist for such a task. Multi-level graph partitioning operates by first coarsening a grid in steps, next partitioning the coarsened grid, and finally de-coarsening the reduced grid into the full grid again. This approach results in a partitioning of the full grid which takes less time than partitioning the full grid directly.

A more recent, but very fast, partitioning method is named Inverse Space-filling Partitioning, see Pilkington and Baden [7]. The basic approach is to map a space-filling curve to a computational grid and partition the space-filling curve into segments with a more or less equal number of elements. The grid partitions are obtained by inverse mapping of the partitioned space-filling curve. The algorithm requires a low amount of memory for the (re)partitioning and load-balancing of numerical simulations. A disadvantage is that the partitioning is non-optimal if the grid of elements does not match the grid structure of the space-filling curve well.

Several articles by Schamberger and Wierum [2], Dennis [3] and Harlacher et al. [4] report about the performance of Inverse Space-filling Partitioning and multi-level graph partitioning for numerical grids and meshes. In the articles of Dennis [3] and Harlacher et al. [4] there is evidence that in some cases Inverse Space-filling Partitioning outperforms multi-level graph partitioning algorithms. Specifically, in those cases where the space-filling curve, that is used to partition, matches the grid and traversal used for simulations well.

## 1.3 Grid traversal and IO-efficiency

Computers can use different media as a data-store for numerical element data: RAM, SSD, disks, network drives etcetera. Ideally, it would be preferred to use the fastest data-store available. However, for technical and financial reasons this is rarely possible.

Computations/programs often reuse recently accessed data. In order to speed-up a computation data-stores use a cache to store recently used data. The cache is much faster than the storage medium, but also much smaller. When wisely used, it reduces the average time for data-access and accelerates the calculation of an element.

For two-dimensional numerical simulations the classical approach was to traverse a regular grid in row/column or column/row order. Data from neighbouring grid cells, elements, are accessed and updated for computing a cell's new state. If the row or column contains more elements than the cache can store, processing will slow down as

data-access and -retrieval takes significantly more time. It would be more efficient to process those elements first who need data that is mostly present in the cache.

One approach to do so, is to use a space-filling curve to traverses a grid that matches a discretized curve exactly. Its recursive properties ensure that the spatial distance between two elements in a grid is bound by an polynomial function of the distance between those two elements on the traversal of a space-filling curve , see chapter 11 in [1] on the Hölder Continuity.

## 1.4 Multi-scale stack-based space-filing traversals

A stack is a data-structure in which data-objects are inserted and removed following the last-in first-out (LIFO) principle. A stack allows only those two operations: push the data-object the stack and pop the data-object from of the stack. Both stack-operations work well together with caches. Together with a space-filling curve for grid traversal one has the building blocks for a cache-efficient traversal.

The elements in a grid based on a space-filling curve are visited in order of the space-filling-curve-based traversal. Each element is visited once to calculate its new state for the next time step in a simulation. Elements are visited in the traversal order and each element's data is used only once. This element data can be popped from an input stack when an element is processed. It is pushed on an output stack when the processing is finished with the element and traverses to the next. Once the full grid is processed one can swap the input and output stacks and visit the elements in a reversed order by reversing the traversal.

Each element also shares data with its neighbouring elements, data stored at vertex locations, that is reused and updated during a traversal. This vertex-data is read at first need from an input stack and stored temporarily on vertex stacks between usage in one element and the next. Once the vertex data is no longer needed it is written to the output stack. In some cases only a small and constant number of stacks is required. Some vertex data may be on a face or an edge and updated by one refined element and that updated vertex data is later used by another equally refined element visited later in the traversal. If the set of updated vertex data can be stored on a single stack for storage at the moment of traversing to the next element and retrieved at the first need for other elements we define the traversal over such an set to be palindromic.

The stack-based traversal using the Sierpiński-Knopp curve, described in Chapter 14 by Bader in [1], nedds only 2 vertex stacks. A three-dimensional traversal based on a cubic 27-reptile was found by Weinzierl and Mehl [5] and used only 6 stacks for vertex-data. The solutions for stack-assignment above use traversals in which subsequent elements in the traversal have two congruent faces that are coinciding. Such a traversal is called face-continuous. When a traversal is not face-continuous it is considered to be face-discontinuous. Face-continuity, being face-continuous, is not necessarily a required condition for a cache-efficient solution as is demonstrated in this report.

## 1.5 three-dimensional $k$-reptile

Haverkort [8] defines an $r$-gentiling as the dissection of a shape into $r \geq 2$ parts which are all similar to the original shape. An $r$-reptiling is an $r$-gentiling of which all parts are mutually congruent. A geometric figure defined by a set of points in Euclidean space is a $r$-reptile if we can tile it with equally large smaller copies of itself. Some of these reptiles are excellent building blocks for a space-filling curve.

An $r$-reptiling can be used to generate a multi-level refinement grid. Take as the domain of a numerical simulator the shape of such an $r$-reptile and repeatedly dissect this shape, effectively refines the original element into a grid. The grid elements form the leaves of a refinement tree. A tree where the root node is the original element and every dissection of a node produces $r$ nodes for which an exact order of visitation exists during a traversal.

After a finite number of $i$ recursive dissections of all the nodes in the tree one obtains an $r$-ary tree where the root node is the original element and every node branches into $r$ new nodes. The nodes who are not dissected are leaves of the $r$-ary tree. An in-order traversal of all the leaves of a tree where all leaves are the product of the same number of dissections gives us a traversal, whose geometric shape is similar to that of a space-filling curve. In this report we allow a space-filling curve to be face-discontinuous.

If one chooses not to refine some nodes while refining others, leaves created by a different number of dissections occur. The leaves of such a tree form a multi-scale grid. Each dissection is considered a refinement of the grid. Leaves or nodes produced by $i$ dissections are defined to have a refinement level of $i$. A three-dimensional $k$-reptile can yield a grid and grid traversal based on a space-filling curve. It is an excellent building block for a cache-efficient multi-scale three-dimensional traversal.

## 1.6 SHL 8-reptile tetrahedra

Kynčl and Safernová [9], showed that any $r$-reptile simplex in three-dimensional requires $r$ to be $k^3$ with $k \in \mathbb{N}^+$. Liu and Joe [10], found a particular valuable 8-reptile simplex whose repeated bisection produced cyclically 3 distinct 8-reptile tetrahedra. Two of the shapes were tetrahedra mentioned by Hill [11] in 1895. One shape he called tetrahedra of the first type. The other shape he designated as a 'special' tetrahedron, which bisects into tetrahedra of the first type. The 'special' tetrahedron shape was demonstrated by Sommerville [12] in 1922 to be a 'true' space-filler; no mirrored copies are needed to fill space. To distinguish the three classes of similar tetrahedra during our written discourse, they are called the Sommerville (type S), Hill (type H) and Liujoedron (type L) tetrahedra: the SHL 8-reptile tetrahedra. Herman Haverkort found an element traversal and refinement scheme that was expected to be palindromic. The Haverkort element traversal and refinement scheme introduces only one discontinuity per refinement cycle. Figure 1.1 depicts the tetrahedra, the traversal-order and the refinement steps. We will show that the Haverkort element traversal it is palindromic for the elements visited on the bisection faces. For a cache-efficient traversal we would

like to store the temporary vertex information also on stacks. This requires a stack-assignment algorithm that ensures that the proper temporary vertex information occurs timely on top of a stack when visiting an element during a traversal.

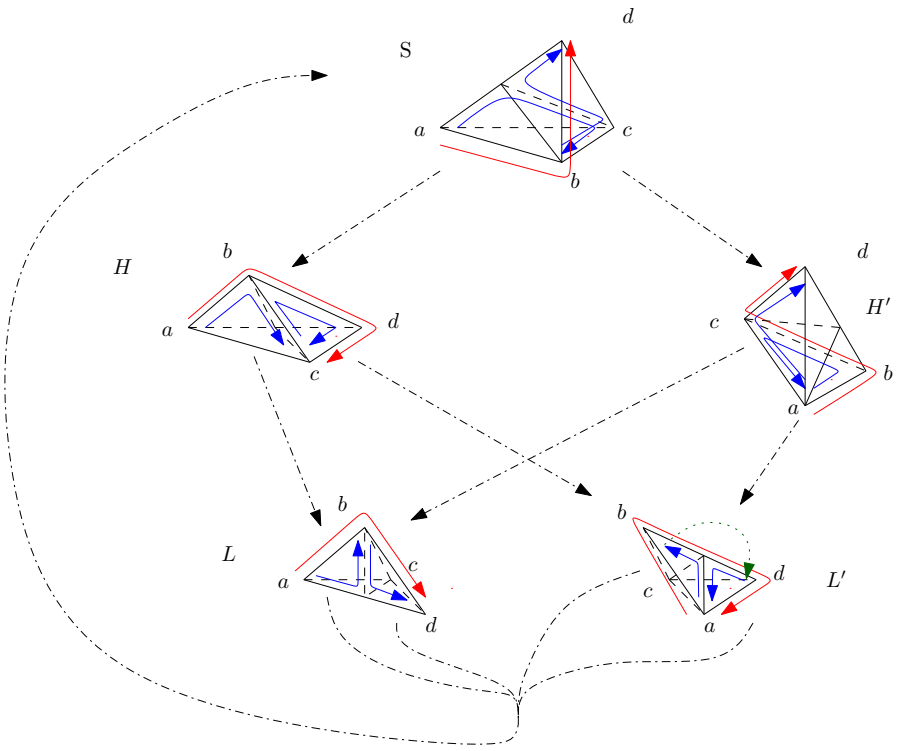


Figure 1.1: Haverkort refinement steps: From Sommerville ($S$), to Hill ($H$ and $H'$), to Luijoedron ($L$ and $L'$) and back to Sommerville ($S$) shaped tetrahedrons again

These tetrahedra and their traversal are investigated in chapter 2. First the pop and push orders in which vertices are needed during a traversal are investigated. These orders are fixed per type of tetrahedron involved and will be shown to be independent of the refinement level and their position in a traversal. As a consequence this ordering also minimizes the number of vertex stacks involved while traversing the elements; hence, we call it the natural order.

The vertex sequence during a traversal defines when a vertex needs to be pushed and when it needs to be popped. A vertex $a$, that needs to be stored before vertex $b$ is stored and is needed before $b$ is retrieved, cannot be on the same stack as $b$. In such a case, when vertex push/pop events cross, we need two stacks. When more vertex events cross, one might need even more stacks. However, less vertex stacks means a more efficient use of a cache's size. Therefore, one would like to find the lowest number of vertex stacks needed. The lower and upper bounds for the maximum value of the minimal number of vertex stacks required during a traversal were investigated using the natural vertex order. The upper bound we found can be used for a stack-assignment algorithm that requires only a constant number of stacks.

**Lower bound**

We investigate the lower bound theoretically using the palindromicity of the bisection faces in chapter 2. When tetrahedra are sufficiently refined all the faces of a tetrahedron and its bisection face can have vertex events that cross with each other. Hence, at least 5 stacks are required for any stack-assignment solution. Each outer face of such a tetrahedron is an ancestor in the binary bisection tree of the root tetrahedron, which implies that up to 4 of its ancestors have bisection faces whose stacks are different.

Experimental verification of this result is possible. One can determine a lower bound for the minimum number of stacks required to store all the vertex events for every moment during a traversal numerically, see chapter 3. We did so until refinement level 20. The known lower bound rises tot five stacks at level refinement level 11 and remains constant from there on.

**Upper bound**

A stack-assignment solution for a traversal is the assignment of vertices and their data to such a stack for storage so that these vertices and their data are available on the top of the stack the next time they are needed during the traversal. One stack-assignment solution is to assign a unique stack per refinement level, every bisection face created at a level is assigned the same unique stack, which gives an upper bound that is equal to the refinement level. As each bisection face on a given level is in a different subtree no vertex events will cross between those bisection faces. This scheme ensures that all ancestors of a subtetrahedron have a different stack for their bisection face.

We tested several adaptive stack algorithms. Our best performing algorithm was based on the optimization scheme used for determining the lower-bound algorithm. It performed slightly better than the afore-mentioned solution with one stack per refinement level. However, further investigations showed there is a stack-assignment solution that requires only a constant number of stacks.

This, so-called, Constant Stack solution assigns vertex stacks to the faces of a tetrahedron depending on the orientation of the face. All faces with the same orientation are assigned the same stack. This strategy holds for all faces of any tetrahedron at any scale and at any refinement level. It allows for an initial upper bound of 9 stacks, while still being compatible with the requirements of a multi-level stack algorithm. Further more experiments show that two of these orientations may be assigned to the same stack. Doing so allows for a final upper bound of 8 stacks. Due to results for the lower bound and tests done on combinations of the stacks planes it can be concluded that for the stack plane approach this is the lowest upper bound possible.

## 1.7 Bisection-based refinement

Bisection is the dissection of an element into the smallest number of parts possible: two. When a bisection creates two parts that are each other's reflection over the bisec-

tion face and whose traversals are each other's reversed reflection, then palindromicity is ensured for all vertices, edges, and faces on the bisection face. Such a traversal and refinement scheme will also support a stack-based, multi-scale traversal due to the palindromicity of the bisection faces. Besides the SHL-tetrahedra, two other reptile convex polyhedra were investigated. For both, an 8-reptile bisected cube and an 8-reptile bisected triangular prism, a traversal and refinement scheme are given; needing 9 respectively 7 stacks, see chapter 4 for the details.

## 1.8 Report structure

First we discuss the Haverkort element traversal for the 8-reptile SHL-tetrahedra and its theoretical properties, lower and upper stack bounds, and the parallel plane stack approach in Chapter 2. Chapter 3 discusses the experiments verification of the lower bound and the Constant Stack solution for the Haverkort element traversal. In Chapter 4 we discuss the traversal and refinement schemes for the two other reptile convex polyhedra mentioned in the previous section. In Chapter 5 we discuss the solution; its expected performance and practical applicability. Chapter 6 summarizes our conclusions.

# Chapter 2

# Traversing 8-reptile SHL-tetrahedra

In this chapter we will first give a general description of the Haverkort element traversal. Next, we will introduce the used notation and discuss stack properties for vertices in the traversal. Subsequent, we will define the Haverkort element traversal and prove some palindromic properties. Building on this, we deduce the natural pop and push orders for the vertex information during a traversal. With these tools we will investigate the upper and lower bounds for the number of temporary vertex stacks; which will be verified experimentally in the next chapter.

## 2.1 General description of the Haverkort element traversal

The Haverkort element traversal and refinement scheme is based on the 8-reptile SHL-tetrahedra. Every tetrahedron in the scheme is refined using the bisection plane through the midpoint of its longest edge and the two vertices that are not part of the longest edge. As a result there are no hanging vertices in a uniform refined traversal, see section 2.8. All vertices are located on the midpoint of an edge. A graphical representation of the traversal and refinement scheme in forward direction is depicted in Figure 1.1.

The dash-dotted arrows indicate in which cyclical order the refinement operations occur. The large arrows (red) near a larger tetrahedron indicate the traversal patterns for the two subtetrahedra that are generated by bisection. The smaller arrows (blue) indicate the traversal patterns for the two subsubtetrahedra that are generated by bisecting a larger tetrahedra twice. The refinement cycle exists out of 3 stages: the Sommerville stage which contains $S$-type tetrahedra, the Hill stage which contains $H$- and $H'$-type tetrahedra and the Luijoedron stage which contains $L$- and $L'$-type tetrahedra. All the tetrahedra at the same refinement level have the same volume as they are mirrored or rotated versions of each other. Tetrahedra are classified into types, $\{S, H, H', L, L'\}$, depending on the bisection-operation required when refining them and their relative place within the traversal.

The refinement scheme produces two child tetrahedra whenever a tetrahedron is refined. The family relationships between tetrahedra form a binary tree structure. Fol-

lowing the path from the root node down to a descendent defines the sequence of refinement-operations on the root node that produce that descendent; only the leaves are actual elements of a grid visited by the traversal. Every tree node has a refinement level equal to its depth in the tree. The root node has a refinement level of 0.

In this report all grids are generated by refining a single 'root' $S$-type tetrahedron, hence the domain has the shape of a $S$-type tetrahedron. Unless stated otherwise, grids in this report are also generated by uniform refinement: whenever we refine a tetrahedral element we refine also all the other tetrahedral elements in the traversal.

Whenever a tetrahedron is bisected into two subtetrahedra $h_1$ and $h_2$ a bisection face is created. If a traversal visits them in their subindex order, their vertex data will be updated. Vertex data located in the bisection face will need to be passed from the subtetrahedron processed first in the traversal, $h_1$, to the subtetrahedron processed next in the traversal, $h_2$. If the subtetrahedra are refined, they will have subtetrahedra themselves, recursively so. These subtetrahedra may need to pass vertex data between them that needs to be updated before it can be passed to tetrahedron $h_2$, recursively so.

Every tetrahedron in a traversal receives vertex data sourced from either a temporary vertex stack or its input stack and passes information on to either a temporary vertex stack or its output stack. Only vertex data located inside the root tetrahedron is passed over the bisection faces of its subtetrahedra. At first need, vertex data is read from the input stack. After its last use, it is written to the output stack. If we can use a single stack to pass data between elements of a traversal over the surface of a geometric shape, that is, a face or an edge, we call the traversal palindromic over that shape.

## 2.2 Basic notation

A right-handed orthonormal Cartesian coordinate system with axes $X$, $Y$ and $Z$ is used. A position vector $\mathbf{p}$ is defined as:

$$\mathbf{p} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = (p_1, p_2, p_3)^T$$

with $T$ the transpose function. A tetrahedron $h$ is defined by 4 vertices to which we assign vertex labels $a$, $b$, $c$ and $d$ respectively as shown in Figure 1.1. We define $\mathbb{T}$ the set of tetrahedron types: $\mathbb{T} := \{H, H', L, L', S\}$. A tetrahedron $h$ of type $\theta$, with $\theta \in \mathbb{T}$, is then denoted as $h = \theta(a, b, c, d)$ using the vertex labels as depicted in Figure 1.1. Edges are written as $ab$ and faces as $abc$. A sequence of tetrahedra, $h_0$ through $h_k$, visited during a traversal $\tau$ is written as

$$h_0 \triangleright h_1 \triangleright h_2 \triangleright ... \triangleright h_k$$

Labels are used to indicate the refinement operations in the Haverkort element traversal. They are named after their tetrahedron types: $\{\mathbf{H}, \mathbf{H}', \mathbf{L}, \mathbf{L}', \mathbf{S}_1, \mathbf{S}_2\}$. The two $S$-type sibling tetrahedra produced when refining an $L$-type or $L'$-type tetrahedron are

given a subscript to indicate their order in a forward traversal. A sequence of refinement operations that produces an $L'$-type tetrahedron $h'$ out of an $S$-type tetrahedron $h$ is therefore written as:

$$\mathbf{H}.\mathbf{L}'$$

and produces $h'$ from $h$:

$$h' = h.\mathbf{H}.\mathbf{L}'$$

.

## 2.3 Traversal events

We define two types of events that occur during a traversal: tetrahedron events and vertex events. The first types of events define the processing of a tetrahedron-shaped element. The second type of events describes when a vertex and its data are read or popped from a vertex stack and when it is written or pushed to an vertex stack.

Let $\tau$ be a tetrahedron event expressed as tuple $\langle b, v_0, v_1, v_2, v_3 \rangle$, in which $b$ is the time index when the tetrahedron is being processed and $v_0$ through $v_3$ are the 4 vertices of the tetrahedron. Tetrahedron time indices are positive integer values: starting at 0 with the first tetrahedron in the traversal and increase with 1 for every tetrahedron in the order of the traversal. Every tetrahedron has 4 stack actions for reading/popping a vertex before processing the tetrahedron element and 4 stack actions for pushing/writing the vertices after it was processed.

We define a vertex (stack) event $\epsilon$ as being a tuple $\langle b, e, p \rangle$, where $b$ is the begin time when it is pushed on stack, $e$ the end time when it is popped from stack, and $p$ the vertex position. All $b$ and $e$ are integer values and unique in the whole vertex time range of the traversal in respect to each other and among themselves. Furthermore time $b - 1$ occurs before time $b$. Writing $\epsilon.b$ means the push time of $\epsilon$, $\epsilon.e$ means the pop time of $\epsilon$ and $\epsilon.p$ extracts the vertex itself.

## 2.4 Some vertex event properties

The first vertex property follows directly from the LIFO nature of a stack:

**Theorem 2.4.1.** *For any stack-assignment solution the push times of the vertices on a stack should decrease when visiting the stack from top to bottom while the pop times should increase from top to bottom during a grid traversal. We call this the* ***vertex event stack order****.*

A second property follows from the fact that we are only processing one element at a time and while processing we only need one copy of a vertex updated or passed on. It is borne out of our desire for efficiency. We do not want duplicate copies of a vertex or its data as it increases our data-size and reduces our cache-efficiency. Fortunately, a single copy suffices as elements are visited sequentially:

**Definition 2.4.2.** *The **single instance property** states that at any time $t$ a vertex exists at most once on any stack be it a temporary vertex stack or input and output stacks.*

Additionally we define for the ease of discussion several predicates that define some properties between two vertex events. The first is that of vertex event spanning:

**Definition 2.4.3.** *A vertex event $\epsilon$ with tuple $\langle b, e, p \rangle$ **spans** vertex time $t$ if and only if*

$$b < t < e \tag{2.1}$$

Events can also cross:

**Definition 2.4.4.** *We define that vertex event $\epsilon_i$ with tuple $\langle b_i, e_i, p_i \rangle$ **crosses** vertex event $\epsilon_j$ with tuple $\langle b_j, e_j, p_j \rangle$ for $i \neq j$ if and only if*

$$(b_i < b_j < e_i < e_j) \vee (b_j < b_i < e_j < e_i) \tag{2.2}$$

In such a case, it is impossible to have the two vertices of two crossing events during the overlap in events on the same stack as this violates the vertex stack order property.

However in some cases vertex events overlap such that the vertices can both be stored on the same stack:

**Definition 2.4.5.** *We define that vertex event $\epsilon_i$ with tuple $\langle b_i, e_i, p_i \rangle$ **covers** vertex event $\epsilon_j$ with tuple $\langle b_j, e_j, p_j \rangle$ for $i \neq j$ if and only if*

$$b_i < b_j \wedge e_j < e_i \tag{2.3}$$

## 2.5 Symmetries and palindromicity

For a $H$-type tetrahedron, with vertices $a$, $b$, $c$, and $d$ we require in this report an angle between all three edges $ab$, $bc$, and $cd$ equal to $\pi/2$ radials and a equal length $l$ for the three edges. Hence, the coordinates for a tetrahedron $h_H = H(a, b, c, d)$ can be picked as follows: $a = (0, 0, 0)^T$, $b = (0, 0, l)$, $c = (l, 0, l)$, and $d = (l, l, l)$ with $l \in \mathbb{R}^+$. A $H$-type tetrahedron is notated by writing $H(a, b, c, d)$.

The shape of a $H'$-type tetrahedron is defined by a reflection of an $H$-type tetrahedron over mirror plane $M$ covering face $bcd$ of $h_H$. Tetrahedron $h_H$ and its reflection $h_{H'}$ are siblings created by bisecting a $S$-type tetrahedron. Vertices $c$, $d$ and $b$ of $H$ lie in the mirror plane $M$ and they are reflected into their own location. The normal vector of face $bcd$ of tetrahedron $h_H$ can be calculated from any two of its edges or from any two edges of face $abc$ of tetrahedron $h_{H'}$. The normal vector of the mirror plane $M$ is $(0, 0, 1)$ and can be described by the equation $z = l$. The vertex labels $a$, $b$, $c$, and $d$ of $H'$, as depicted in Figure 1.1, are assigned to the mirrored vertices of $c$, $d$, $b$, and $a$ of $h_H$.

A tetrahedron $h_{H'}$ has a coinciding face in the mirror plane $M$ with a face of $h_H$. The values for the vertices of $h_{H'}$ are therefore: $a = (l, 0, l)^T$, $b = (l, l, l, )^T$, $c = (0, 0, l)^T$, and $d = (0, 0, 2l)^T$. Edges $ab$, $bc$, and $cd$ of $h_{H'}$ all have length $l$ and their angles are

equal to $\pi/2$ radials. Both the $H$-type tetrahedron $h_H$ and its symmetric sibling $H'$-type tetrahedron $h_{H'}$ are called Hill-type tetrahedra, see [13].

Any traversal $\tau$ visiting elements in a refined $h_H$ in some order will, when reflected, visit their mirror elements in the same order in a refined $h_{H'}$. If we reflect and reverse the traversal $\tau$ in $h_H$ these mirror locations are visited in a reverse order in $h_{H'}$. This also holds true for the order in which the subset of elements which are adjacent to their parents bisection face are visited. Therefore, the bisection face of $h_S$ is palindromic for the order in which the traversal visits its elements. Similarly, when rotating and translating $h_H$ and $h_{H'}$ such that face $abc$ of $h_H$ coincides with face $acd$ of $h'_H$, these face pairs will be traversed palindromic too. In fact, any combination of a face of $h_H$ with a face of $h_H$ will be traversed palindromic with its mirrored companion face of $h_{H'}$.

We define a tetrahedron $h_S$ such that it is the tetrahedron whose bisection produces $h_H$ and $h'_H$. In that case, the longest edge in $h_S$ is $ad$ with length $2l$. For the case of $h_H$ and $h'_H$ above, we find that the vertices of $h_S$ are as shown in Tabel 2.1. Looking at the lengths of the edges of $h_S$, we find that there are only 4 viable combinations of coinciding faces between $h_S$ type tetrahedra, section 2.7.3 states them. Each of which, when refined once to the Hill-type level, is traversed palindromic due to the traversal palindromicity between $h_H$ and $h'_H$. Similarly, if we construct an $L$-type or an $L'$-type tetrahedra from two $S$-type then the elements' faces on the bisection face between them will be traversed in palindromic order too. So all $S$, $H$, $H'$, $L$ and $L'$ tetrahedron types that can be constructed by using the Haverkort refinement scheme have bisection faces whose elements' faces are visited in a palindromic order.

## 2.6 The Haverkort element traversal and refinement scheme

We can use the information above to write a more formal definition of the traversal and refinement scheme. We use the vertex labels as depicted in Figure 1.1. Now the refinement scheme can be written as follows:

$$S(a,b,c,d) \overset{\triangle\triangle}{\rightarrow} H(a,a\bar{\circ}d,b,c) \triangleright H'(b,c,a\bar{\circ}d,d) \tag{2.4}$$

$$H(a,b,c,d) \overset{\triangle\triangle}{\rightarrow} L(a,b,a\bar{\circ}d,c) \triangleright L'(c,b,a\bar{\circ}d,d) \tag{2.5}$$

$$H'(a,b,c,d) \overset{\triangle\triangle}{\rightarrow} L'(a,b,b\bar{\circ}d,c) \triangleright L(a,c,b\bar{\circ}d,d) \tag{2.6}$$

$$L(a,b,c,d) \overset{\triangle\triangle}{\rightarrow} S(a,a\bar{\circ}d,c,b) \triangleright S(b,a\bar{\circ}d,c,d) \tag{2.7}$$

$$L'(a,b,c,d) \overset{\triangle\triangle}{\rightarrow} S(a,b\bar{\circ}d,c,b) \triangleright S(d,b\bar{\circ}d,c,a) \tag{2.8}$$

in where '$\overset{\triangle\triangle}{\rightarrow}$' means bisection refinement and '$\triangleright$' determines the forward traversal order from left to right. In the above notation $a\bar{\circ}b$ is the vector label representing the vector operations $\frac{1}{2}(a+b)$ on the corresponding vertices: The point exactly between vertices $a$ and $b$. If $b = c\bar{\circ}d$ then our notation implies $a\bar{\circ}b = a\bar{\circ}(c\bar{\circ}d)$: the vector label for the vector defined by $\frac{1}{2}(a+\frac{1}{2}(c+d))$.

The formulas show that the splitting of a tetrahedron depends solely on the tetrahedron type of the corresponding parent node. A reversed traversal is found by simply reversing the order of the tetrahedron visited. Hence, when reversed, a traversal sequence $h_i \triangleright h_{i+1} \triangleright h_{i+2}$ becomes $h_{i+2} \triangleright h_{i+1} \triangleright h_i$. From the formulas it can be deduced that on every uniform refined level only tetrahedron types of its stage occurs. Tetrahedron type $S$ occurs on Half-Liujoedron levels, types $H$ and $H'$ occur on Hill levels, and types $L$ and $L'$ occur on Liujoedron levels.

We calculated the vertex positions and edge lengths for each of the basic tetrahedron types based on the vertex positions given above for an $H$-type tetrahedron, see Tabel 2.1.

| $\mathbb{T}$ | $a$ | $b$ | $c$ | $d$ | $\lvert ab\rvert$ | $\lvert ac\rvert$ | $\lvert ad\rvert$ | $\lvert bc\rvert$ | $\lvert bd\rvert$ | $\lvert cd\rvert$ |
|---|---|---|---|---|---|---|---|---|---|---|
| S | $(0,0,0)^T$ | $(l,0,l)^T$ | $(l,l,l)^T$ | $(0,0,2l)^T$ | $\sqrt{2}l$ | $\sqrt{3}l$ | $2l$ | $l$ | $\sqrt{2}l$ | $\sqrt{3}l$ |
| H | $(0,0,0)^T$ | $(0,0,l)^T$ | $(l,0,l)^T$ | $(l,l,l)^T$ | $l$ | $\sqrt{2}l$ | $\sqrt{3}l$ | $l$ | $\sqrt{2}l$ | $l$ |
| H′ | $(l,0,l)^T$ | $(l,l,l)^T$ | $(0,0,l)^T$ | $(0,0,2l)^T$ | $l$ | $l$ | $\sqrt{2}l$ | $\sqrt{2}l$ | $\sqrt{3}l$ | $l$ |
| L | $(0,0,0)^T$ | $(0,0,l)^T$ | $(\frac{l}{2},\frac{l}{2},\frac{l}{2})^T$ | $(l,0,l)^T$ | $l$ | $\sqrt{\frac{3}{4}}l$ | $\sqrt{2}l$ | $\sqrt{\frac{3}{4}}l$ | $\sqrt{\frac{3}{4}}l$ | $l$ |
| L′ | $(l,0,l)^T$ | $(0,0,l)^T$ | $(\frac{l}{2},\frac{l}{2},\frac{l}{2})^T$ | $(l,l,l)^T$ | $l$ | $\sqrt{\frac{3}{4}}l$ | $l$ | $\sqrt{\frac{3}{4}}l$ | $\sqrt{\frac{3}{4}}l$ | $\sqrt{2}l$ |

Table 2.1: Vertex coordinates for some of the tetrahedra in Figure 1.1

## 2.7 Vertex data flow

While the Haverkort element traversal defines a total order for the tetrahedral elements to be traversed, it does not so for the vertices in those tetrahedral elements. For each face of a tetrahedron we are required to temporarily store sets of vertices and their information on the stack of a bisection face. The order in which the tetrahedral elements are traversed and the fact that the vertex event stack property must hold, reduces the set of possible orders in which vertices can be pushed and popped. Each face of an tetrahedron lies either in a bisection face or in the outer face of the root tetrahedron. The order in which each of these faces are visited reduces the set of possible orders for pushing and popping the vertices of an element even further. However, pushing and popping these vertices should occur such that vertex event crossings are minimized. For this, we will analyse the vertex data flow and construct a vertex order for pushing and popping vertices of an element that does not create additional crossed vertex events, even when refining the element in question. Doing so we find and give proof for a traversal order for the vertices such that the vertex- and element-data of a bisection face are visited in palindromic order.

### 2.7.1 The vertex traversal order when refining an element

We take a root tetrahedron $S_r = S(a,b,c,d)$. Next, we apply substitution of equation 2.4 through equation 2.8:

$$S_r(a,b,c,d) \overset{\triangle\triangle}{\rightarrow} H_1(a, a\overline{\circ}d, b, c) \triangleright H_2'(b, c, a\overline{\circ}d, d) \tag{2.9}$$

$$\overset{\triangle\triangle}{\rightarrow} L_1(a, a\overline{\circ}d, a\overline{\circ}c, b) \triangleright L_2'(b, a\overline{\circ}d, a\overline{\circ}c, c) \triangleright L_3'(b, c, c\overline{\circ}d, a\overline{\circ}d) \triangleright L_4(b, a\overline{\circ}d, c\overline{\circ}d, d) \tag{2.10}$$

$$\begin{aligned}\overset{\triangle\triangle}{\rightarrow} S_1(a, a\overline{\circ}b, a\overline{\circ}c, a\overline{\circ}d) \triangleright S_2(a\overline{\circ}d, a\overline{\circ}b, a\overline{\circ}c, b) \triangleright \\ S_3(b, c\overline{\circ}(a\overline{\circ}d), a\overline{\circ}c, a\overline{\circ}d) \triangleright S_4(c, c\overline{\circ}(a\overline{\circ}d), a\overline{\circ}c, b) \\ S_5(b, c\overline{\circ}(a\overline{\circ}d), c\overline{\circ}d, c) \triangleright S_6(a\overline{\circ}d, c\overline{\circ}(a\overline{\circ}d), c\overline{\circ}d, b) \triangleright \\ S_7(b, b\overline{\circ}d, c\overline{\circ}d, a\overline{\circ}d) \triangleright S_8(a\overline{\circ}d, b\overline{\circ}d, c\overline{\circ}d, d)\end{aligned} \tag{2.11}$$

After the 3 refinement steps, we obtain 8 $S$-type tetrahedra, each of which can be refined further.

As can be seen from the formulation a vertex push/pop order starts to appear for the vertices $a, b, c, d$ of the initial $S$ for their input and output. Formula 2.9 shows that when $S_r$ is refined to $H \triangleright H'$ vertices $a, b, c, a\overline{\circ}d$ are required in the first element $H$ for processing and before that $d$ is required in the second element $H'$. Furthermore, $a$ is no longer required after the first element and it can be purged from memory to the output stream before $b, c, d$ and $a\overline{\circ}c$ can be purged to the output stream. This order for the vertices of a $S$-type element becomes more strict with further refinement.

In fact, we find that after several refinements formulas 2.9 through 2.11 restrict the input/output order on the original tetrahedron $S$ further. This is only a stronger version of the input/output orders of lesser refinement levels: a vertex input order of $a, b, c,$ and $d$ and a vertex output order of $a, c, b,$ and $d$. Further refinement only increases the number of tetrahedra but not the ordering, see equation 2.4 through equation 2.8. The input/output order are valid for any $S$ refined from a root tetrahedron $S$.

| | $S$ | $H$ | $H'$ | $L$ | $L'$ |
|---|---|---|---|---|---|
| input pop | $a<b<c<d$ | $a<b<c<d$ | $a<b<c<d$ | $a<c<b<d$ | $a<c<b<d$ |
| output push | $a<c<b<d$ | $a<b<d<c$ | $b<a<c<d$ | $a<b<c<d$ | $b<d<c<a$ |

Table 2.2: Input pop and output push orders for the different tetrahedra types.

We can reuse formulas 2.9 through 2.11 to obtain the input/output orders for $H$, $H'$, $L$ and $L'$ when refining in a similar fashion, see Table 2.2. The input order is defined as the order in which they are required when processing the input element stream, the output order is defined as the order in which they are to be written to the output element stream. We define the ordering in Table 2.2 as the natural input/output order for vertices. This ordering is independent of the refinement level.

The above input/output order of any vertex is determined by the refinement scheme. This fact allows us to determine the push/pop times of any vertex in a traversal in the following way. We assign each of the vertices of a root tetrahedron a vertex time such that it respects the input/output order. Each push and pop event is given an unique time index. Every tetrahedron event has therefore 8 vertex events. Using the refinement equations together with the input/output order of Table 2.2 we can reorder the vertices of the tetrahedron being refined, insert the newly created bisection vertex, and assign indices for each of the push and pop actions for every new vertex event. Each of those new vertex events will lie between the time indices of the events of every predecessor and successor, of the current tetrahedron that is being refined, in a traversal. As a result, we know the required push and pop order of every vertex beforehand doing a mesh traversal.

When refining an $S$-type tetrahedron once, we find that the output of the common vertices between a tetrahedron of type $H$ and the input order of its successor in the traversal, sibling tetrahedron of type $H'$, match. None of the vertex events between them cross and no additional stacks are required. This is also the case between any of the siblings of the other refinement level types, proof of this will be given in the following sections below.

In fact, we will give proof that the input/output vertex orders from Table 2.2 are palindromic for all adjacent faces of the tetrahedra in a refinement level. In order to do this, we investigate the vertex input/output orders between adjacent tetrahedra in a uniform-refined root tetrahedron.

### 2.7.2 Vertex traversal orders for tetrahedron faces

Each tetrahedron exchanges data with adjacent tetrahedra. Tetrahedra can have a face, edge or vertex which coincide with another tetrahedra. Therefore at most 3 vertices of a tetrahedron are stored temporarily on a bisection stack to be exchanged to a tetrahedron on the other side of the shared bisection face.

First we find the possible combinations of adjacent faces. Then we derive the constraints on the vertices' order for each combination as follows from the tetrahedron shape and vertex labelling. Finally, we try to find a solution, preferable independent of the adjacent face combination. Ideally is a solution that matches with the orders of Table 2.1 as it reduces the complexity of a traversal scheme for the vertices on a bisection face. Our analysis is done for each refinement level type independently in the sections below.

### 2.7.3 Sommerville-type refinement level

For two similar sized $S$-type tetrahedra, $S_1$ and $S_2$, we find three fully face connected possibilities using the edge lengths in Table 2.1. If the order and length match, the adjacent faces may coincide. This means that every face of an $S$-type tetrahedra has only one face of any other $S$-type tetrahedra it can coincide with. Figure 2.1 shows a possi-

ble palindromic order in which the coinciding vertices between face $abc$ of tetrahedra $S_1$ and face $cbd$ of tetrahedron $S_2$ may be traversed.

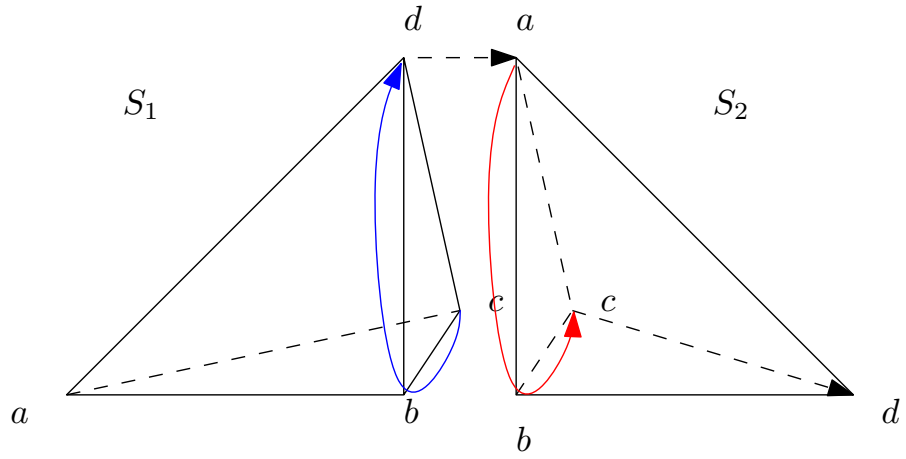


Figure 2.1: A possible order in which the coinciding vertices of $S_1$ and $S_2$ can be visited in palindromic order. This order is represented by the expression: $a \rhd b \rhd c \stackrel{m}{=} c \rhd b \rhd d$.

Equation 2.12 expresses the order depicted in Figure 2.1. On the left we write the sequence of vertices traversed in $S_1$ and on the right the matching sequence of $S_2$:

$$a \rhd b \rhd c \stackrel{m}{=} c \rhd b \rhd d \tag{2.12}$$
$$a \rhd c \rhd d \stackrel{m}{=} a \rhd c \rhd d \tag{2.13}$$
$$a \rhd d \rhd b \stackrel{m}{=} a \rhd d \rhd b \tag{2.14}$$

here $\stackrel{m}{=}$ is used to express the matching palindromic traversal of the vertices. We may reverse and shift the cyclical sequence $a \rhd d \rhd b$ for both the left and the right side without violating the stack properties for these vertices. Doing so, we obtain:

$$a \rhd b \rhd c \stackrel{m}{=} c \rhd b \rhd d \tag{2.15}$$
$$a \rhd c \rhd d \stackrel{m}{=} a \rhd c \rhd d \tag{2.16}$$
$$a \rhd b \rhd d \stackrel{m}{=} a \rhd b \rhd d \tag{2.17}$$

If we use the left side for input order for any $S$-type tetrahedron and the right side for output order of any $S$-type tetrahedron. we obtain from them the following rules for input,

$$(a < b < c) \wedge (a < c < d) \wedge (a < b < d) \Leftrightarrow a < b < c < d \tag{2.18}$$

,

and output,

$$(c < b < d) \wedge (a < c < d) \wedge (a < b < d) \Leftrightarrow a < c < b < d \tag{2.19}$$

.

This yields a total stack order for any intersection of faces or edges that may occur in Sommerville-type levels. We have obtained an input/output order which is equal to the input/output orders in Table 2.2 for $S$-type tetrahedra.

### 2.7.4 Hill-type refinement level

From the combinations allowed for $S$-type tetrahedron it follows that only $H$ and $H'$-type tetrahedra have coinciding faces. We know that some faces that coincide are traversed differently when refined two more levels. For example face $abc$ is split into two $S$-type faces of an $L$-type tetrahedron and face $abc$ of type $H'$ is refined into two $S$-type faces of an $L'$-type tetrahedron. The visiting order of these subfaces and the tetrahedron would not be palindromic if they would occur. Hence we obtain the following cyclical orders for edge lengths of an $H$- or $H'$-type tetrahedra and its edge lengths for possible coinciding face combinations. We write on the left side of the matching palindromic equivalence $\stackrel{m}{=}$ the $H$-type faces and on the right side the $H'$-type faces.

$$a \rhd b \rhd c \stackrel{m}{=} a \rhd c \rhd d \tag{2.20}$$
$$b \rhd d \rhd c \stackrel{m}{=} a \rhd b \rhd c \tag{2.21}$$
$$a \rhd d \rhd b \stackrel{m}{=} c \rhd b \rhd d \tag{2.22}$$
$$a \rhd c \rhd d \stackrel{m}{=} b \rhd a \rhd d \tag{2.23}$$

We may shift and reverse the matching palindromic equivalent cyclical sequences $c \rhd b \rhd d$ and $b \rhd a \rhd d$ without violating the stack properties for these vertices. For similar reasons, we can shift $b \rhd d \rhd c$. Doing so, we obtain:

$$a \rhd b \rhd c \stackrel{m}{=} a \rhd c \rhd d \tag{2.24}$$
$$b \rhd d \rhd c \stackrel{m}{=} a \rhd b \rhd c \tag{2.25}$$
$$a \rhd b \rhd d \stackrel{m}{=} b \rhd c \rhd d \tag{2.26}$$
$$a \rhd d \rhd c \stackrel{m}{=} a \rhd b \rhd d \tag{2.27}$$

Using the right side for input order for any $H'$-type tetrahedron and the left side for output order of any $H$-type tetrahedron we obtain the following rules for input,

$$(a < c < d) \wedge (a < b < c) \wedge (b < c < d) \wedge (a < b < d) \Leftrightarrow a < b < c < d \tag{2.28}$$

,

and output,

$$(a < b < c) \wedge (b < d < c) \wedge (a < b < d) \wedge (a < d < c) \Leftrightarrow a < b < d < c \tag{2.29}$$

.

Which is equal to the input/output orders in Table 2.2 for $H'/H$.

We may reverse and shift the cyclical sequence $b \rhd d \rhd c$ and $a \rhd d \rhd c$ for both the left and the right side without violating the stack properties for these vertices. For similar reasons, we can shift $b \rhd d \rhd c$. Doing so, we obtain:

$$a \rhd b \rhd c \stackrel{m}{=} a \rhd c \rhd d \tag{2.30}$$
$$b \rhd c \rhd d \stackrel{m}{=} b \rhd a \rhd c \tag{2.31}$$
$$a \rhd b \rhd d \stackrel{m}{=} b \rhd c \rhd d \tag{2.32}$$
$$a \rhd c \rhd d \stackrel{m}{=} b \rhd a \rhd d \tag{2.33}$$

Using the right side for input order for any $H$-type tetrahedron and the left side for output order of any $H'$-type tetrahedron we obtain the following rules for input,

$$(a < b < c) \wedge (b < c < d) \wedge (a < b < d) \wedge (a < c < d) \Leftrightarrow a < b < c < d \tag{2.34}$$

,

and output,

$$(a < c < d) \wedge (b < a < c) \wedge (b < c < d) \wedge (b < a < d) \Leftrightarrow b < a < c < d \tag{2.35}$$

.

Which is equal to the input/output orders in Table 2.2 for $H/H'$.

Not all face combinations may occur. However if they do, the input/output order in Table 2.2 will ensure that the vertices of a face or edge will be written in a palindromic order on any bisection face stack on a Hill-type refinement level. It can be observed that any $H$-type edge coinciding with an $H$-type edge is part of a face which coincides with $H'$, hence the order will be proper.

### 2.7.5 Liujoedron-type refinement level

Again making use of the the symmetry between a refined $H$-type tetrahedron and $H'$-type tetrahedron we can determine that face-intersections between an $L$-type tetrahedron and an $L'$-type tetrahedron are limited to the following sets:

$L \stackrel{m}{=} L$ :

$$a \rhd b \rhd c \stackrel{m}{=} c \rhd b \rhd d \tag{2.36}$$
$$a \rhd b \rhd d \stackrel{m}{=} a \rhd d \rhd a \tag{2.37}$$
$$a \rhd c \rhd d \stackrel{m}{=} d \rhd c \rhd a \tag{2.38}$$

$L \stackrel{m}{=} L' :$

$$b \rhd c \rhd a \stackrel{m}{=} a \rhd c \rhd b \tag{2.39}$$

$L' = L' :$

$$a \rhd b \rhd d \stackrel{m}{=} b \rhd d \rhd a \tag{2.40}$$
$$b \rhd c \rhd d = b \rhd c \rhd d \tag{2.41}$$
$$a \rhd b \rhd c \stackrel{m}{=} c \rhd d \rhd a \tag{2.42}$$

$L' = L :$

$$a \rhd c \rhd d \stackrel{m}{=} b \rhd c \rhd a \tag{2.43}$$

We shift an reverse some in order to get the input order of $L$ an $L'$ equal to that of Table 2.2.

$L \stackrel{m}{=} L :$

$$a \rhd c \rhd b \stackrel{m}{=} b \rhd c \rhd d \tag{2.44}$$
$$a \rhd b \rhd d \stackrel{m}{=} b \rhd d \rhd a \tag{2.45}$$
$$a \rhd c \rhd d \stackrel{m}{=} d \rhd c \rhd a \tag{2.46}$$

$L \stackrel{m}{=} L' :$

$$a \rhd c \rhd b \stackrel{m}{=} b \rhd c \rhd a \tag{2.47}$$

$L' \stackrel{m}{=} L' :$

$$a \rhd b \rhd c \stackrel{m}{=} c \rhd d \rhd a \tag{2.48}$$
$$a \rhd b \rhd d \stackrel{m}{=} b \rhd d \rhd a \tag{2.49}$$
$$b \rhd c \rhd d \stackrel{m}{=} b \rhd c \rhd d \tag{2.50}$$

$L' \overset{m}{=} L$ :

$$a \rhd c \rhd d \quad \overset{m}{=} \quad b \rhd c \rhd a \tag{2.51}$$

Using the left side for input order for any L and the right side for output order of any $L$ or $L'$ we obtain the following rules for input for $L$

$$(a < b < d) \wedge (a < c < b) \wedge (a < c < d) \wedge (a < c < b) \Leftrightarrow a < c < b < d \tag{2.52}$$

and output for $L$,

$$(b < c < d) \wedge (b < d < a) \wedge (d < c < a) \wedge (b < c < a) \Leftrightarrow b < d < c < a \tag{2.53}$$

.

and for $L'$

$$(a < b < d) \wedge (a < c < d) \wedge (c < b < d) \wedge (a < c < d) \Leftrightarrow a < c < b < d \tag{2.54}$$

,

and output for $L'$,

$$(b < c < a) \wedge (b < d < c) \wedge (d < c < a) \wedge (b < d < a) \Leftrightarrow b < d < c < a \tag{2.55}$$

.

Not all face combinations may occur. However if they do, the input/output order in Table 2.2 will ensure that no vertex events passing information over a face will cross. The vertices of a face or edge will be written in a palindromic order on any bisection face stack on a Hill-type refinement level. It can be observed that any $L$-type edge coinciding with an $H$-type edge is part of a face which coincides with $L'$, hence the order will be proper. Giving us the final set of input/output orders needed.

### 2.7.6 Discussion

Not all face combinations discussed above may occur. However if they do, the input/output order in Table 2.2 will ensure that the vertices of a face or edge will be written and read in a palindromic order for each element face. The vertex input/output orders remain constant even when uniformly refining the tetrahedra, see 2.7.1. Furthermore, section 2.5 shows that any traversal visiting points on a bisection face from one side in some order will visit these points in the reversed order on the other side.

Hence we can conclude that supplementing the element traversal with the input/output orders for vertices in Table 2.2 ,we obtain an element and vertex traversal which is palindromic over any bisection face in any Haverkort element traversal.

As both the vertex and the element data of a bisection face are visited in palindromic order, we can store this information using a single stack per bisection face. We call such a single stack a bisection face stack. Remember, the bisection face of a tetrahedron $h$ is the face that bisects the tetrahedron $h$. Therefore, only during the traversal of a refined tetrahedron, data is pushed on the bisection face stack of that tetrahedron during traversal of the first child and subsequently popped during traversal of its second child. Therefore we can state:

**Theorem 2.7.1.** *The state of the bisection face stack of a tetrahedron just before and straight after traversing its tetrahedron is identical. We call this property the* ***bisection face stack state property****.*

If we always pass the vertex information over the bisection face to the next tetrahedron that needs it, we will only have one copy of that information and theorem 2.4.2 the single instance property will hold true.

## 2.8 Some properties of the SHL-tetrahedra

The SHL-tetrahedra have some properties that have shown themselves to be of interest for determining any lower or upper bounds of a stack-assignment solution. We can explain these more briefly by discussing them separately in this section.

Sommerville [12] showed that a tetrahedron of type $S$ is a space-filler. It is a true space-filler as termed by Sommerville: it requires no reflection to fill space. Four copies of an $S$-type tetrahedron rotated in steps of $\pi/2$ over $bd$ builds a pyramid and 6 of those pyramids build a cube which is a space-filler. Each tetrahedron has no hanging edges or vertices when filling space.

As will be proven in the next section, each face of a type $S$ tetrahedron has only one candidate congruent face of its own tetrahedron type without using reflection. The tetrahedron configuration at a Sommerville refinement level occurs within the space-filling configuration as described by Sommerville. The shape of a great grandparent node of an $S$-type tetrahedron leaf would span two such Sommerville cubes, see Figure 2.2. An $S$-type tetrahedron is replicating. Eight such Sommerville cubes can form a new Sommerville cube where the edges of the $S$-type tetrahedra are twice as large.

The cube configuration allows us to determine that the vertices in a Sommerville refinement level are a member of 48, 24 or 8 tetrahedra. Within a space-filling configuration, Two S-type tetrahedron form the shape of a Liujoedron tetrahedron. Therefore, in a Liujoedron refinement level vertices are a member of 48, 24 or 12 tetrahedra. Similarly reasoned, in a Hill refinement level vertices are a member of 48 or 16 tetrahedra. Similar as for the Sommerville tetrahedra, neither Hill or Liujoedron tetrahedra have hanging edges or vertices when filling space.

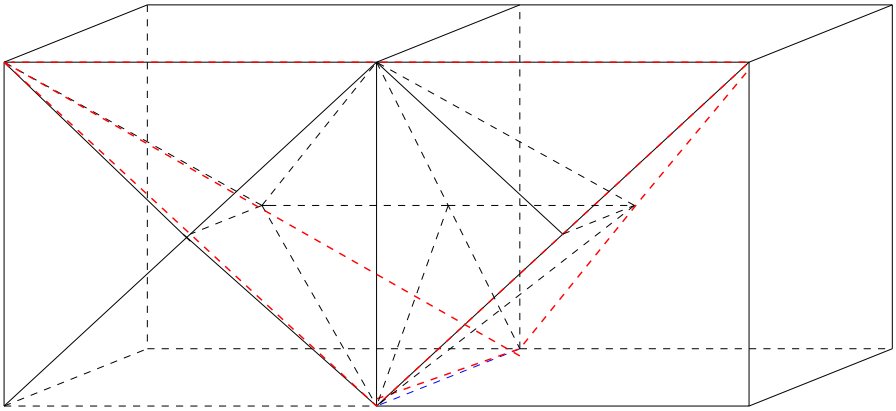

Figure 2.2: The SHL-reptile configuration drawn in Sommerville cubes. A single great-grantparent of type $S$ spans two subcubes of its own Sommerville-cube.

A Sommerville cube is dissected by a fixed number of planes: nine. A cube shape is a space filler and requires no rotation or reflection for filling space. Therefore, even when refining a cube into smaller Sommerville cubes , the set of plane orientations remain the same. Hence, at most 9 unique plane orientations can occur at any Sommerville-type refinement level. If we bisect all tetrahedra in a Sommerville cube into an $H$ and $H'$-type tetrahedron, no new plane orientations are created. Hence, at most 9 unique plane orientations can occur at any Hill-type refinement level. Similarly so, we find the same nine unique plane orientations for Liujoedron-type refinement levels.

## 2.9 A lower bound for the number of vertex stacks

We can determine a lower bound for the number of stacks needed for the 4+1 faces of a bisection tetrahedron. For this we assume processing a sufficiently refined tetrahedron $h$, each of the faces has as many subfaces as we need. In each of the traversal intervals related to a descendant of $h$ there are push and pop operations, vertex events, for each of the vertices on a face of $h$. All faces of a tetrahedron $h$ are part of a bisection face in the domain or part of an outer face of the root tetrahedron. No information is passed over the outer face the root tetrahedron. All bisection faces are palindromic, so no vertex events of vertices on the same bisection face will ever cross using a stack. Tetrahedron $h$ is defined to have a bisection face $bisect$. We investigate whether vertex events associated with $bisect$ cross with events associated with the outer faces of $h$. We do this for each of the different tetrahedron types that exist for $h$. The results will give us a lower bound when using the same vertex stack for passing vertex information over a bisection face.

### 2.9.1 Type $S$

For the following cases, we apply the refinement scheme together with the palindromicity of the bisection faces.

**Vertex events of $bisect$ cross vertex events of $abc$:** There are two cases.

In the case events, stored on face stack $abc$, are pushed before processing $S$ and popped when processing $S.H$. Then events popped from face stack $abc$ in $S.H.L'.S_2.H'$ cross with events pushed during $S.H.L'.S_2.H$ the bisect stack.

In the case events, stored on face stack $abc$, are popped after processing $S$ and pushed when processing $S.H$. Then events pushed on face stack $abc$ in $S.H.L'.S_2$ cross with events pushed at $S.H.L.S_1$ and popped in $S.H'.L'.S_2$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $abd$:** There are two cases.

In the case events, stored on face stack $abd$, are pushed before processing $S$ and popped when processing $S.H.L$ and $S.H'.L$. Then events popped from face stack $abd$ in $S.H'.L.S_1.H$ cross with events pushed during $S.H.L'.S_1.H$ and popped during $S.H'.L.S_1.H'$ from the bisect stack.

In the case events, stored on face stack $abd$, are popped after processing $S$ and pushed when processing $S.H.L$ and $S.H'.L$. Then events pushed on face stack $abd$ in $S.H'.L.S_1.H$ cross with events pushed at $S.H.L.S_1$ and popped in $S.H'.L.S_1.H'$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $acd$:** There are two cases.

In the case events, stored on face stack $acd$, are pushed before processing $S$ and popped when processing $S.H$ and $S.H'$. Then events popped from face stack $acd$ in $S.H.L'.S_2$ cross with events pushed during $S.H.L'.S_1$ and popped during $S.H'.L'.S_2$ from the bisect stack.

In the case events, stored on face stack $acd$, are popped after processing $S$ and pushed when processing $S.H$ and $S.H'$. Then events pushed on face stack $acd$ in $S.H.L'.S_2$ cross with events pushed at $S.H.L'.S_1$ and popped in $S.H'.L'.S_2$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $bcd$:** There are two cases.

In the case events, stored on face stack $bcd$, are pushed before processing $S$ and popped when processing $S.H'.L'.S_1$ and $S.H'.L$. Then events popped from face stack $bcd$ in $S.H'.L'.S_1$ cross with events pushed during $S.H.L'.S_1$ and popped during $S.H'.L'.S_2$ from the bisect stack.

In the case events, stored on face stack $bcd$, are popped after processing $S$ and pushed when processing $S.H'.L'.S_1$ and $S.H'.L$. Then events pushed on face stack $bcd$ in $S.H'.L'.S_1$ cross with events pushed at $S.H.L'.S_1$ and popped in $S.H'.L'.S_2$ from the bisect stack.

The above case analysis proves that the bisection face crosses with all outer faces of a tetrahedron of type $S$. Reversing the traversal reverses all vertex events (switching pushes and pops), hence it hold for both traversal directions.

### 2.9.2 Type $H$

**Vertex events of $bisect$ cross vertex events of $abc$:** There are two cases.

In the case events, stored on face stack $abc$, are pushed before processing $H$ and popped when processing $H.L$. Then events popped from face stack $abc$ in $H.L.S_2.H'$ cross with events pushed during $H.L.S_2.H$ and popped during $H.L'.S_1.H'$ the bisect stack.

In the case events, stored on face stack $abc$, are popped after processing $H$ and pushed when processing $H.L$. Then events pushed on face stack $abc$ in $H.L.S_2.H'$ cross with events pushed at $H.L.S_2.H$ and popped during $H.L'.S_1.H'$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $abd$:** There are two cases.

In the case events, stored on face stack $abd$, are pushed before processing $H$ and popped when processing $H.L.S_1$ and $H.L'$. Then events popped from face stack $abd$ in $H.L'.S_1.H'.L'$ cross with events pushed during $H.L.S_2.H$ and popped during $H.L'.S_1.H'.L$ the bisect stack.

In the case events, stored on face stack $abd$, are popped after processing $H$ and pushed when processing $H.L.S_1$ and $H.L'$. Then events pushed on face stack $abd$ in $H.L'.S_1.H'.L'$ cross with events pushed during $H.L.S_2.H$ and popped during $H.L'.S_1.H'.L$ the bisect stack.

**Vertex events of $bisect$ cross vertex events of $acd$:** There are two cases.

In the case events, stored on face stack $acd$, are pushed before processing $H$ and popped when processing $H$. Then events popped from face stack $acd$ in $H.L.S_2.H'$ cross with events pushed during $H.L.S_2.H$ and popped during $H.L'.S_1.H'$ from the bisect stack.

In the case events stored on face stack $acd$, are popped after processing $H$ and pushed when processing $H$. Then events pushed on face stack $acd$ in $H.L.S_2.H'$ cross with events pushed at $H.L.S_2.H$ and popped in $H.L'.S_1.H'$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $bcd$:** There are two cases.

In the case events, stored on face stack $bcd$, are pushed before processing $H$ and popped when processing $H.L'$. Then events popped from face stack $bcd$ in $H.L'.S_1.H$ cross with events pushed during $H.L.S_2.H$ and popped during $H.L'.S_1.H'$ from the bisect stack.

In the case events, stored on face stack $bcd$, are popped after processing $H$ and pushed when processing $H$. Then vents pushed on face stack $bcd$ in $H.L'.S_1.H$ cross with events pushed at $H.L.S_2.H$ and popped in $H.L'.S_1.H'$ from the bisect stack.

The above case analysis proves that the bisection face crosses with all outer faces of a tetrahedron of type $H$. Reversing the traversal reverses all vertex events (switching pushes and pops), hence it hold for both traversal directions.

### 2.9.3 $H'$

As $H'$ is a mirrored tetrahedron of type $H$ with a reversed traversal, the proof of $H$ holds too. Pops become pushes and pushes become pops. Hence, in $H'$ the $bisect$ face events cross with all 4 tetrahedron faces.

### 2.9.4 $L$

**Vertex events of $bisect$ cross vertex events of $abc$:** There are two cases.

In the case events, stored on face stack $abc$, are pushed before processing $L$ and popped when processing $L.S_1$. Then events popped from face stack $abc$ in $L.S_1.H'.L$ cross with events pushed at $L.S_1.H'.L'$ and popped in $L.S_2.H.L'$ from the bisect stack.

In the case events, stored on face stack $abc$, are popped after processing $L$ and pushed when processing $L.S_1$. Then events pushed on face stack $abc$ in $L.S_1.H'.L$ cross with events pushed at $L.S_1.H'.L'$ and popped in $L.S_2.H.L'$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $abd$:** There are two cases.

In the case events, stored on face stack $abd$, are pushed before processing $L$ and popped when processing $L$. Then events popped from face stack $abd$ in $L.S_1.H'.L$ cross with events pushed at $L.S_1.H'.L'$ and popped in $L.S_2.H.L'$ from the bisect stack.

In the case events, stored on face stack $abd$, are popped after processing $L$ and pushed when processing $L$. Then events pushed on face stack $abd$ in $L.S_1.H'.L$ cross with events pushed at $L.S_1.H'.L'$ and popped in $L.S_2.H.L'$ from the bisect stack.

**Vertex events of $bisect$ <u>do not</u> cross vertex events of $acd$:** There are two cases.

In the case events are stored on face stack $acd$ and they are pushed before processing $L$. Then all elements on the bisection face are pushed and popped between $L.S_1.H'$ and $L.S_2.H$, while events are popped from $acd$ during $L.S_1.H$ and $L.S_2.H'$. No vertex events cross.

In the case events are stored on face stack $acd$ and popped after processing $L$. Then all elements on the bisection face are pushed and popped between $L.S_1.H$ and $L'.S_2.H'$, , while events are pushed onto $acd$ during $L.S_1.H$ and $L.S_2.H'$. No vertex events cross.

**Vertex events of $bisect$ cross vertex events of $bcd$:** There are two cases.

In the case events, stored on face stack $bcd$, are pushed before processing $L$ and popped when processing $L.S_2$. Then events popped from face stack $bcd$ in $L.S_2.H.L$ cross with events pushed at $L.S_1.H'.L'$ and popped in $L.S_2.H.L'$ from the bisect stack.

In the case events, stored on face stack $bcd$, are popped after processing $L$ and pushed when processing $L.S_2$. Then events pushed on face stack $bcd$ in $L.S_2.H.L$ cross with events pushed at $L.S_1.H'.L'$ and popped in $L.S_2.H.L'$ from the bisect stack.

The above case analysis proves that the bisection face crosses with 3 of the outer faces of a tetrahedron of type $L$. Reversing the traversal reverses all vertex events (switching pushes and pops), hence it hold for both traversal directions.

### 2.9.5 $L'$

**Vertex events of $bisect$ cross vertex events of $abc$:** There are two cases.

In the case events, stored on face stack $abc$, are pushed before processing $L'$ and popped when processing $L.S_1$. Then events popped from face stack $abc$ in $L'.S_1.H'$ cross with events pushed during $L'.S_1.H$ and popped during $L'.S_2.H'$ from the bisect stack.

In the case events, stored on face stack $abc$, are popped after processing $L'$ and pushed when processing $L'.S_1$. Then events pushed on face stack $abc$ in $L'.S_1.H'$ cross with events pushed during $L'.S_1.H$ and popped during $L'.S_2.H'$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $abd$:** There are two cases.

In the case events, stored on face stack $abd$, are pushed before processing $L'$ and popped when processing $L'$. Then events popped from face stack $abd$ in $L'.S_1.H'$ cross with events pushed during $L'.S_1.H$and popped during $L'.S_2.H'$ from the bisect stack.

In the case events, stored on face stack $abd$, are popped after processing $L'$ and pushed when processing $L'$. Then events pushed on face stack $abd$ in $L'.S_1.H'$ cross with events pushed during $L'.S_1.H$ and popped during $L'.S_2.H'$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $acd$:** There are two cases.

In the case events are stored on face stack $acd$ are pushed before processing $L'$and popped when processing $L'.S_2$. Events popped from face stack $acd$ in $L'.S_2.H$ cross with events pushed during $L'.S_1.H$ and popped during $L'.S_2.H'$ from the bisect stack.

In the case events, stored on face stack $acd$, are popped after processing $L'$ and pushed when processing $L'.S_2$. Then events pushed from face stack $acd$ in $L'.S_2.H$ cross with events pushed during $L'.S_1.H$ and popped during $L'.S_2.H'$ from the bisect stack.

**Vertex events of $bisect$ cross vertex events of $bcd$:** There are two cases.

In the case events, stored on face stack $bcd$ ,are pushed before processing $L'$.and popped when processing $L'$.. Then events popped from face stack $bcd$ in $L'.S_1.H$ cross with events pushed during $L'.S_1.H$ and popped during $L'.S_2.H'$ from the bisect stack.

In the case events,stored on face stack $bcd$, are popped after processing $L'$ and pushed when processing $L'$. Then events pushed from face stack $bcd$ in $L'.S_1.H$ cross with events pushed during $L'.S_1.H$ and popped during $L'.S_2.H'$ from the bisect stack.

The above case analysis proves that the bisection face crosses with all outer faces of a tetrahedron of type $L'$. Reversing the traversal reverses all vertex events (switching pushes and pops), hence it hold for both traversal directions.

### 2.9.6 Combining the lower bounds

For $S$, $H$, $H'$, and $L'$ type tetrahedra we know that any bisection face independent of the tetrahedron type has vertex events that cross with all the outer tetrahedron faces. For $L$ there are only 3 faces whose vertex events cross with the bisection face. Traversing a sufficiently refined and detached tetrahedron, other than of type $L$, requires at least 5 different vertex stacks. At a sufficiently high uniform refinement level, we can find tetrahedra whose outer faces are all bisection faces of larger tetrahedra-shaped nodes themselves and all cross with the inner bisection face and each other. Hence, a lower bound of 5 temporary vertex stacks will exist for every refinement level type as $L$- and $L'$-type tetrahedra occur on the same refinement levels.

**Theorem 2.9.1.** *The lower bound property: There is a constant $c$ such that at every refinement level $i$ with $i \geq c$ there are at least 5 different bisection face stacks required for storing the temporary vertex events.*

This also means that nearly all the bisection faces of the tetrahedra-shaped nodes in the refinement tree that intersect are required to have different stacks. Finding a way to reuse bisection face stacks such that their number remains small enough to be cache-efficient will not be trivial. More so, solution based on the approach of Weinzierl and Mehl [5] for a 27 cube subdivision is therefore not feasible. Neither is a solution based on the solutions for 2D-traversals as mentioned by Bader in [1].

## 2.10 Constant upper bound

Using the palindromicity of the bisection faces we can construct several upper bounds for the maximum number of stacks needed for a traversal. One of them has a constant number of temporary vertex stacks independent of the refinement level. Each of them should follow the constraints we have discussed earlier in this chapter for the refinement and the traversal:

1. **Theorem 2.4.1: The vertex event stack order.**
2. **Definition 2.4.2: The single instance property.**
3. **Theorem 2.9.1: The lower bound property.**
4. **Theorem 2.7.1: The bisection stack state property.**

A simple approach would be to use one stack per coinciding face. Each tetrahedron has 4 faces. Each face coincides with at most one other tetrahedron. Non-coinciding faces do not require a temporary vertex stack. This would require at most $2^{i+1}$ stacks where $i$ equals the refinement level. The cache-efficiency of using stacks would be quickly dampened at higher refinement levels by the large number of vertex stacks involved, when the tops of the stacks take a significant portion of the cache its effective cache-size is reduced.

The bisection face stack state property, theorem 2.7.1, allows us to use one stack per bisection face without getting vertex events that cross. The state of a bisection face stack is identical before and after processing its tetrahedron. Hence, we need at most one extra temporary vertex stack for each new refinement level. This gives us an upper bound linear with the refinement level. Therefore, we know that for the number of required temporary stacks $n_t$ it holds that:

$$n_t \leq i_{\text{max}} \tag{2.56}$$

.

While this is reasonable for a limited number of levels, we can do better. In order to get a constant number of stacks, we need to find some way to reuse the stacks assigned to bisection faces. And we need to do it in such a way that any vertex events of a bisection face do not cross with the events already on the stack. However, tetrahedra in the traversal use bisection faces borne at different refinement levels, simply reusing a stack after some finite number of uniform level refinements will not work: at some deep level of refinement, these bisection faces will intersect in some tetrahedra and vertex events may cross. From section 2.9.6 we know that nearly all the bisection faces of the tetrahedra-shaped nodes in the refinement tree that intersect are required to have different stacks. We can strengthen this constraint slightly by defining the following requirement:

**Definition 2.10.1.** *The unique bisection stack requirement: All bisection faces that intersect must have a different stack assigned.*

We do this by requiring that all (bisection) planes in the traversal grid that intersect all have different stacks. Tetrahedra have no parallel faces, nor is the bisection face parallel with one of its faces. The number of unique plane orientations in a traversal grid is at most 9, see section 2.4. Hence, our upper bound for $n_t$ is now reduced to a constant number of stacks:

$$n_t \leq 9 \tag{2.57}$$

.

An even lower upper bound may exist if the vertex events of two plane-orientation do not cross at any refinement level of the same type: Sommerville, Hill or Liujoedron. In that case, we may combine them without creating crossing vertex events and use less than 9 stacks: using the same stack for two or more planes. In the experiments performed, we found exactly such a case, see section 3.3.2. An upper bound of 8 temporary vertex stacks exists indeed. It is a lower bound for this approach. No other options for combining plane stacks were found to be viable. We obtain:

$$n_t \leq 8 \tag{2.58}$$

.

## 2.11 Required depth of verification

We will show that verifying the upper and lower bounds for the number of required stacks during a traversal for a sufficiently high level of refinement proves it for any refinement level. This allows us to experimentally verify our theoretical upper and lower bounds for any refinement level.

Imagine a sufficiently refined root tetrahedron. The tetrahedra inside its volume will occur in different configurations around every vertex in the root volume. We say that a tetrahedron is detached from an ancestor if none of its faces lie in a face of that ancestor. We start with $S$ as a root tetrahedron with refinement level 0. After 4 bisections, at refinement level $4$, the first two tetrahedrons occur who are detached from the root tetrahedron. The two detached tetrahedra are Hill-type tetrahedra: $S.H.L'.S_1.H$ and $S.H'.L'.S_2.H'$. Two refinement levels further the first $S$ type tetrahedra occur whose faces are detached from the root tetrahedra. Refining 3 times more we find the first $S$ type tetrahedra whose vectors are also detached from the root tetrahedra's outer faces. At refinement level 9 we find all possible configurations of $S$-type tetrahedra around a vertex of a tetrahedron of type $S$. Two more refinements gives us all the configurations of Hill and Liujoedron tetrahedra around a vertex at refinement level 11. Therefore if our lower bound holds at at a refinement level of 11, it should hold at higher refinement levels too.

For our upper bound we need take the different plane orientations in account. However, we are now only interested in all the possible tetrahedron configurations that occur for all the bisection face stacks at every possible orientation. If we refine a root

tetrahedron of type $S$ 9 times, the first Sommerville cubes occurs inside the root volume. All the configurations that exist for all of the tetrahedra faces now occur on the sides of the Sommerville cube. These configurations cover all the different cases for each of the plane orientations as the Sommerville cube only occurs in our SHL-mesh without reflection and rotation. When uniform refining a Sommerville cube once, we split it into 8 Hill cubes containing only Hill-type tetrahedra and when refining once more we obtain Liujoedron cubes. Therefore if our upper bounds holds at refinement level 11 it should hold at any higher level too.

**Theorem 2.11.1.** *If the lower bound property, theorem 2.9.1 and upper bound in equation 2.57 can be proven until refinement level 11 it will hold for all higher refinement levels.*

## 2.12 Multi-scale traversals

Bisection face stacks are well suited for multi-scale refined traversals. Consider a face-adjacent tetrahedra that is less or more refined that its neighbour in a multi-scale refined traversal. Each bisection face stack involved is only read from or written to when processing a tetrahedron. When writing vertex data we simply write the data for the size of the current tetrahedra. In the case data is read from an adjacent face a different approach is used. For the faces of the adjacent tetrahedra which are less refined we can read the three vertices from the temporary vertex stacks and apply a bisection of the face and push the refined vertices on the proper input and vertex stacks. If the faces of the adjacent tetrahedra are more refined we can simply read the all face data that lies in a face of the current tetrahedra being processed, coarsen the vertex data and use them. The extra refined information is not required to be written to the output streams on behalf of the lesser refined tetrahedra. Only data for the tetrahedron at their originally refinement level is required to be written to the output. The above approach assumes we can interpolate the points over a face without issue. Such is the case when using linear approximation for the integration an element.

If this is not the case, one could consider allowing no hanging vertices between elements and allow at most one refinement step during a traversal per element. After a traversal, one then removes any hanging vertices by iteratively refining any neighbouring cells. This can be done efficiently using the technique described by Bader [1]. First one calculates the next time step and adaptively refines the grid during a traversal. Then one traverses and whenever a hanging vertex is found we refining its adjacent elements. We repeat this until no hanging vertices are found. As discussed in the first paragraph of section 2.1 there are no hanging vertices in a uniform refined traversal so a solution exists. Due to the fact that the bisection scheme of any tetrahedron only bisects two of the four faces of a tetrahedra a multi-scale solution will exist.

## 2.13 Traversal code

A traversal and refinement framework was written and tested for the tetrahedra using the tetrahedra refinement scheme following Figure 2.4 through 2.8. The framework creates a traversals for up to a given $i_{max}$ number of levels. The software stores the refinement output for every refinement level less or equal to $i_{max}$ and a discrete time index indicating the order in which vertices are used and reused. Due to the discrete timings we assume a pop occur always at the start of timing $b$ before any other actions occur while a push always occurs at the end of timing $b$ after any other actions occurring at timing $b$. This requires forethought when implementing the predicates and properties in section 2.4. First usage of a vertex is considered to require a read-operation. The last usage of a vertex is considered to require a write operation. Any usage of a vertex in between requires either a push to a stack for later reuse or a pop for retrieval from a stack when needed. The full sequence of vertex operations during a traversal is thus created. This list of vertex operations is then converted to a list of vertex events which are then traversed in push order. Any stack assignment approach can then be implemented and subsequently tested. For each push operation we can test if the vertex stack order property holds. For each pop operation we can test if a vertex is indeed present on the right stack as indicated by the stack-assignment algorithm before popping it. This allows us to verify and analyse the working of any stack algorithm during runtime. We can log relevant state information during runtime and when exceptions are thrown.

Verification of the implemented refinement algorithm and the produced list of vertex events was done by comparing the output for the first 4 levels against a manual computed list of vertex events. The framework was then used to confirm the lower bound of 5 stacks from theorem 2.9.1. After which it was used to prove the upper bound of 9 bisection face stacks in equation 2.57 and show that it could be reduced experimentally to an upper bound of 8 bisection face stacks. The following chapters will discuss these experiments using the framework.

# Chapter 3

# Upper and lower bounds for the number of stacks required

The traversal and refinement framework described in section 2.13 is used in this chapter to investigate the Haverkort element traversal and refinement scheme and to gain an understanding of its structure and properties. We experimentally verify the natural vertex push and pop orders and the theoretical lower and upper bounds for the number of temporary vertex stacks required during a traversal.

In this chapter we will first discuss the verification of the lower bound experimentally, without assuming the palindromicity of the bisection faces nor assigning stacks to bisection faces as we did for the theoretical lower bound. It is a general lower bound for the minimal number of stacks requiring only that the stack event order property and the single instance property hold true. The lower bound algorithm was adapted to a stack assignement algorithm: the Compacting Greedy Algorithm. This algorithm was used to verify our upper bound of 9 stacks and prove that it can be reduced to 8 stacks. Once proven, the Constant Stack solution was implemented.

## 3.1 A numerical lower bound for the number of stacks required

For a lower bound, we would like to know the lowest number of stacks needed to store all the temporary vertex data during a uniformed refined traversal with a refinement level of $i$. We call this the traversal's maximum lower bound for the number of stacks at refinement level $i$: $s(i)$.

There is a set of vertex events that span a vertex time $t$. The smallest number of stacks required to store these spanning vertex events while the vertex event stack order holds for all the stacks denote as: $l(i,t)$. The largest value for $l(i,t)$ for all possible $t$ during a uniformed refined traversal with a refinement level of $i$ we denote as $l(i)$. It is the maximum of all values for $l(i,t)$ for any value of $t$. When progressing from time $t$ to $t+1$ vertex data is pushed or popped from a stack. As a result, the set of stacks at vertex time $t$ may require a radical different configuration of the vertices on the set of stacks at

$t+1$ to be minimal. Even if the smallest number of stacks required is equal. The result is that during the traversal the vertices on a stack would need to be reordered to transition from a stack configuration for $l(i,t)$ to a valid stack configuration for $l(i,t+1)$. We find that $l(i)$ is a weaker bound than $s(i)$ and a lower bound for $s(i)$.

$$s(i) \geq l(i)$$

A brute-force analysis of $l(i)$ becomes quickly very time consuming at higher refinement level, it running time increases exponentially with the refinement level. However using topological sorting on the push time of a vertex event we can significantly reduce the running time complexity to $O(\log n)$ for calculating $l(i,t)$ and thus also for $l(i)$, with $n$ the number of vertex events spanning a vertex time $t$.

In a brute-force approach we would test all stack permutations to determine the smallest number of stacks whose vertex configuration respects vertex event stack order. By sorting the spanning vertex events in the order of ascending vertex time for the push-operation, we can greatly reduce the permutations needed to be investigated. Using a sorted list we can directly generate a vertex event stack ordered minimal configuration for a vertex time $t$ in which a set of vertex events exist that contains one vertex event from each stack and each vertex event in that set crosses with every other vertex event in that set. We will construct a LOWERBOUND-algorithm that does so and afterwards give proof that the constructed configuration is indeed minimal.

For this the algorithm we first sort all the spanning vertex events $\epsilon_0$ to $\epsilon_n$ in the order of ascending vertex time for the push-operation and store it into a list $L$. The first vertex event $\epsilon_0$ on the list is the one with the lowest vertex time for its push action. It is placed on stack 0. We visit the remaining vertex events on the list $L$ in order. If the vertex stack order property allows us to push a vertex event on stack 0 we do so otherwise we try the next stacks in their index order $(0..m-1)$. If we run out of stacks to push a vertex event on, we add an empty stack which will store the vertex event. We add as many stacks as are needed. After we have processed the list, the number of stacks created is equal to $l(i,t)$.

**Lemma 3.1.1.** *When the* LOWERBOUND*-algorithm terminates, $m$, the number of stacks, is equal to $li,t)$.*

The pairing of push and pop actions of a vertex into a vertex event occurs always such that the single instance property, definition 2.4.2, is fulfilled as any vertex is popped before a new copy is pushed. The algorithm itself ensures the vertex event stack order, theorem 2.4.1, holds true while the sorting of the vertex events ensures they are pushed in push-order on to the stacks. As a result, each stack is a set of covering vertex events. Let's define a vertex event $\epsilon_{i,j}$ that was pushed on vertex time $i$ with $i < t$, that lies on a stack with index $j$ and $0 < j < m$. From our algorithm we know there exists a vertex event $\epsilon_{k,j-1}$ with $k < i$ such that i crosses with $\epsilon_{i,j}$ as $k < i$. This implies that the pop time of $\epsilon_{k,,j-1}$ is before that of $\epsilon_{i,j}$, otherwise they would cover each other and lie on the same stack. Remember, each vertex stack operation has its own unique time index.

We now define a function $p_j(\epsilon_{i,j})$ such that $\epsilon_{k,j-1} = p_{j-1}(\epsilon_{i,j})$ and $p_{j-2}(\epsilon_{i,j}) = p_{j-2}(\epsilon_{k,j-1})$. We also know that push times of any vertex event on the stacks occur before or at the beginning of $t$ and any pop times of any vertex event on the stacks occur after $t$. Hence all $p_k(\epsilon_{i,j})$ with $k$ is $0 \leq k < j$ cross with $\epsilon_{i,j}$ for any $0 < j < m$. Therefore, a set of vertex events exists that contains one vertex event from each stack and each vertex event in that set crosses with every vertex event in that set. No smaller stack configuration will exists for the lower spanning bound $l(i,t)$ while theorem 2.4.1, the vertex event stack order, holds true. The number of stacks created by the algorithm is the minimal number for a given vertex time $l(i,t)$. If no vertex events exists, there are no stacks required and $m = 0$.

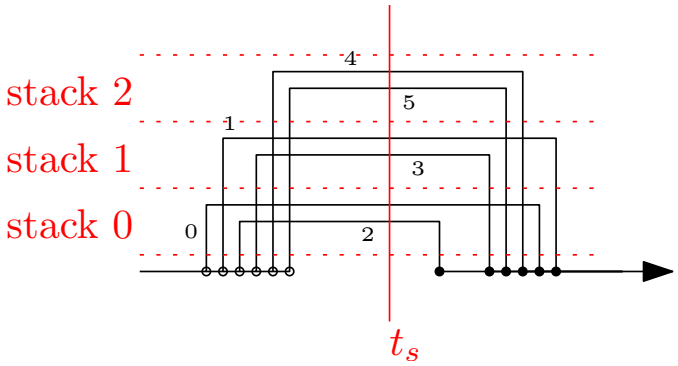


Figure 3.1: A graphical representation of the lower bound packed stacks. $t_s$ is the vertex time being spanned.

The number of stacks created by the algorithm is a numerical solution for lower bound $l(i,t)$. The running time complexity for calculating $l(i,t)$ is $O(mn + n \log n)$ with $m$ the number of stacks and $n$ the number of vertex events spanning $t$. The core part of the algorithm takes $O(mn)$ time complexity per vertex event time $t$. Updating the sorted data when moving from $t$ to $t+1$ would take at most $O(\log n)$.

The vertex events are stacked in a unique configuration by the LOWERBOUND algorithm. However, more than one possible minimal configuration can exists for a given vertex time $t$. No vertex event can be moved from a stack with a higher index onto a stack with a lower index without violating theorem 2.4.1, the vertex event stack order. So the number of stacks for a minimal configuration will never become less. However, vertex events from a stack with a lower index can in some cases be moved to a stack with a higher index. Therefore more than one possible minimal configuration may exist for $l(i,t)$. Several of these possible minimal configurations may or may not allow a transition from a spanned vertex time $t$ to a vertex time $t+1$ while doing the one push or pop required, maintaining a LOWERBOUND calculated number of stacks, and adhering to the vertex event stack properties.

The LOWERBOUND algorithm was implemented using the traversal and refinement framework and we determined the numerical lower bound up to refinement level 20. The results showed that the lower bound grows to 5 when reaching refinement level

**Algorithm 1** LOWERBOUND($\epsilon[]$, $n$, $m$).

Calculates $m$, the minimum number of stacks needed to store the vertex events stored in array $\epsilon[]$ with array size $n$ spanning a vertex time $t$. Function PEEK() looks at the top element of a stack, function SIZE() returns the number of elements on a stack.

**Require:** $n \geq 0 \wedge (\forall i : 0 \leq i < n : \epsilon[i]$ spans vertex time $t) \wedge (\forall i, j : 0 \leq i < j < n : \epsilon[i].b < \epsilon[j].b)$ {Vertex events are sorted in push times and span vertex event time $t$.}
  **if** $n < 2$ **then**
    $m \leftarrow n$ {Empty array requires 0 stacks, 1 vertex event requires 1 stack.}
    **return**
  $m \leftarrow 1$ {Set $m$ to 1 stack, stack 0 is in use.}
  $s[0] \leftarrow \epsilon[0]$ {Put first vertex event in stack 0.}
  $j \leftarrow 1$
  **while** $j < n$ **do**
    flag $= false$
    $j \leftarrow 0$
    **for** $i \leftarrow 0, i < m$ **do**
      **if** PEEK$(s[i]).e > \epsilon[j].e$ **then**
        PUSH$(s[i], \epsilon[j])$ {Vertex event is allowed on stack $i$.}
        flag $= true$
        break {Break for loop.}
    **if** flag $= true$ **then**
      $j \leftarrow j + 1$
    **else**
      PUSH$(s[i], \epsilon[j])$
      $m \leftarrow m + 1$
      $j \leftarrow j + 1$
  **return** $m$ {$m$ contains the minimum number of stacks required.}

| refinement level | LOWERBOUND-value |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 6 | 3 |
| 7 | 3 |
| 8 | 4 |
| 9 | 4 |
| 10 | 4 |
| 11 | 5 |
| 12 | 5 |
| 13 | 5 |
| 14 | 5 |
| 15 | 5 |
| 16 | 5 |
| 17 | 5 |
| 18 | 5 |
| 19 | 5 |
| 20 | 5 |

Table 3.1: Lower bound calculated using the LOWERBOUND-algorithm.

11 and it remains constant beyond, see Table 3.1. Thus, our experimental lower bound confirms the lower bound of 5 stacks from theorem 2.9.1 up to level 20, while refinement level 11 would suffice following theorem 2.11.1. However, the LB-algorithm does not assume using the same stack for a bisection face, while theorem 2.9.1 does.

An adaptation of the LOWERBOUND-algorithm allows us to calculate a value for the minimal number of vertices that needs to be popped to transition our configuration at vertex time $t$ to a minimal configuration at $t + 1$. We call this the reshuffle count. A lower bound for the reshuffle count will assist us in our effort to lower the upper bound from 9 to 8.

When vertex event $\epsilon$ is popped from a stack $k$, rerunning algorithm LOWERBOUND would place some vertex events from higher stacks on stack $k$. We find the vertex event with the largest depth on stack $k + 1$ that would be placed on stack $k$ at $t + 1$. Its depth we define as $r(t)$ at time $t$. $2r(t)$ is a lower bound for the number stack-operations for reshuffling the vertices on the stacks into the configuration at $t + 1$. While $2r(t)$ is not very close to the highest lower bound for reshuffles it does tell us that when $r(t) = 0$ we can transition from time $t$ to time $t + 1$ without reshuffling the vertex-information on the stacks.

## 3.2 The Compacting Greedy algorithm

Measuring the number of spanning events as function of the refinement level showed they increased exponentially. Any brute force approach to determine a lower or upper bound would therefore also increase exponentially or worse as a function of the refinement level. However, our framework used a greedy stack-assignment algorithm during development which required approximately one stack per refinement level. The greedy algorithm first tries to push a new vertex event on the stack with the lowest stack index. If that violates the vertex event stack property, it tries the stack with the next index. If no suitable stacks exist its adds an empty stack and pushes the vertex information on it. First we tried several ad-hoc adaptations of it, removing empty stacks, sorting the stacks by pop time and more. The Compacting Greedy algorithm, which removes any empty stacks after a pop, was the most effective of our adaptations.

The Compacting Greedy algorithm required at most number of stacks linear with the refinement level. At level 19 it required only 11 stacks, see Table 3.2. However, none of the above adaptations yielded a constant-stack-assignment solution or gave a sub-linear solution. Also, all of the adaptations had a higher time complexity than assigning one stack per refinement level which takes $O(1)$. On the other hand, the analysis of these experiments did yield valuable insights into the complex way in which vertex events cross.

## 3.3 Upper bounds and solvers

For implementing any stack assignment scheme that uses one stack per bisection face we need to determine which vertices need to be stored on the stack of which bisection plane.

### 3.3.1 Bisection stack assignments during refinement

For every tetrahedron we need to read vertex data from a set of bisection face stacks, update it, and write it to another set of bisection face stacks. The bisection face stack sets are defined by order the traversal visits the tetrahedra. When we refine a tetrahedron into two new subtetrahedra, a new vertex and a new bisection face is created. The information of this new vertex will be read from the input stream when we start processing the first subtetrahedron in the traversal. It is written to the output stream stack when we finished processing the second subtetrahedron. The new bisection face is assigned a stack. A new bisection face bisects faces into new faces, creates and bisects edges and creates a single new vertex. Of the new faces only the one that are part of a bisection face have vertices that are stored on temporary vertex stacks. As it is the bisection faces that are assigned a temporary vertex stack, the temporary vertex stack is passed down from a bisection face down to new faces, down to their edges, and finally to a new vertex. Information of vertices from non-bisection faces, that lie in the outer

| refinement level | LOWERBOUND-value |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 6 | 4 |
| 7 | 5 |
| 8 | 5 |
| 9 | 7 |
| 10 | 8 |
| 11 | 8 |
| 12 | 9 |
| 13 | 10 |
| 14 | 10 |
| 15 | 11 |
| 16 | 13 |
| 17 | 13 |
| 18 | 14 |
| 19 | 16 |
| 20 | 17 |

Table 3.2: Stacks used by the Compacting Greedy algorithm as a function of the refinement level

faces of the root tetrahedron, are read from the input stream or written to the output stream.

Using the refinement scheme we can determine the way in which stacks are passed down from the bisection face to the vertices. We use the following notation to specify the assignment of the bisection stacks to each vertex of a tetrahedron when refining. We define $x$,$y$, and $z$ being vertices and $y.r$ is the pop stack of $y$ and $y.w$ the push stack of $y$. Then $(x.r, x.w) \leftarrow (y.r, z.w)$ means $x.r$ gets the stack of $y.r$ and $x.w$ gets the stack of $z.w$. Note, that to the right of the $\leftarrow$ only vertices, edges, and faces of the refined child are allowed. To the left of the $\leftarrow$ only vertices, edges, and faces of the tetrahedron (parent) being refined are allowed. SF is a stack function which assigns a stack for the input/output case of a newly created bisection face. The general stack assignment scheme for bisection faces is detailed below for each of the refinement operations.

**Assignments for $S \overset{\triangle\triangle}{\rightarrow} H \triangleright H'$**

Assignments to stacks of H from stacks of S:

$$(a.r, a.w) \leftarrow (a.r, a.w) \tag{3.1}$$
$$(b.r, b.w) \leftarrow (ad.r, \text{SF}) \tag{3.2}$$
$$(c.r, c.w) \leftarrow (b.r, \text{SF}) \tag{3.3}$$
$$(d.r, d.w) \leftarrow (c.r, \text{SF}) \tag{3.4}$$
$$(ab.r, ab.w) \leftarrow (ad.r, ad.w) \tag{3.5}$$
$$(ac.r, ac.w) \leftarrow (ab.r, ab.w) \tag{3.6}$$
$$(ad.r, ad.w) \leftarrow (ac.r, ac.w) \tag{3.7}$$
$$(bc.r, bc.w) \leftarrow (abd.r, \text{SF}) \tag{3.8}$$
$$(bd.r, bd.w) \leftarrow (acd.r, \text{SF}) \tag{3.9}$$
$$(cd.r, cd.w) \leftarrow (bc.r, \text{SF}) \tag{3.10}$$
$$(abc.r, abc.w) \leftarrow (abd.r, abd.w) \tag{3.11}$$
$$(abd.r, abd.w) \leftarrow (acd.r, acd.w) \tag{3.12}$$
$$(acd.r, acd.w) \leftarrow (abc.r, abc.w) \tag{3.13}$$
$$(bcd.r, bcd.w) \leftarrow (\text{IN}, \text{SF}) \tag{3.14}$$

Assignments to stacks of H′ from stacks of S:

$$(a.r, a.w) \leftarrow (\text{SF}, b.w) \tag{3.15}$$
$$(b.r, b.w) \leftarrow (\text{SF}, c.w) \tag{3.16}$$
$$(c.r, c.w) \leftarrow (\text{SF}, ad.w) \tag{3.17}$$
$$(d.r, d.w) \leftarrow (d.r, d.w) \tag{3.18}$$
$$(ab.r, ab.w) \leftarrow (\text{SF}, bc.w) \tag{3.19}$$
$$(ac.r, ac.w) \leftarrow (\text{SF}, abd.w) \tag{3.20}$$
$$(ad.r, ad.w) \leftarrow (bd.r, bd.w) \tag{3.21}$$
$$(bc.r, bc.w) \leftarrow (\text{SF}, acd.w) \tag{3.22}$$
$$(bd.r, bd.w) \leftarrow (cd.r, cd.w) \tag{3.23}$$
$$(cd.r, cd.w) \leftarrow (ad.r, ad.w) \tag{3.24}$$
$$(abc.r, abc.w) \leftarrow (\text{SF}, \text{OUT}) \tag{3.25}$$
$$(abd.r, abd.w) \leftarrow (bcd.r, bcd.w) \tag{3.26}$$
$$(acd.r, acd.w) \leftarrow (abd.r, abd.w) \tag{3.27}$$
$$(bcd.r, bcd.w) \leftarrow (acd.r, acd.w) \tag{3.28}$$

**Assignments for $H \overset{\triangle\triangle}{\rightarrow} L \rhd L'$**

Assignments to stacks of $L$ from stacks of $H$:

$$(a.r, a.w) \leftarrow (a.r, a.w) \tag{3.29}$$
$$(b.r, b.w) \leftarrow (b.r, \text{SF}) \tag{3.30}$$
$$(c.r, c.w) \leftarrow (ad.r, \text{SF}) \tag{3.31}$$
$$(d.r, d.w) \leftarrow (c.r, \text{SF}) \tag{3.32}$$
$$(ab.r, ab.w) \leftarrow (ab.r, ab.w) \tag{3.33}$$
$$(ac.r, ac.w) \leftarrow (ad.r, ad.w) \tag{3.34}$$
$$(ad.r, ad.w) \leftarrow (ac.r, ac.w) \tag{3.35}$$
$$(bc.r, bc.w) \leftarrow (abd.r, ) \tag{3.36}$$
$$(bd.r, bd.w) \leftarrow (bc.r, ) \tag{3.37}$$
$$(cd.r, cd.w) \leftarrow (acd.r, ) \tag{3.38}$$
$$(abc.r, abc.w) \leftarrow (abd.r, abd.w) \tag{3.39}$$
$$(abd.r, abd.w) \leftarrow (abc.r, abc.w) \tag{3.40}$$
$$(acd.r, acd.w) \leftarrow (acd.r, acd.w) \tag{3.41}$$
$$(bcd.r, bcd.w) \leftarrow (\text{IN}, \text{SF}) \tag{3.42}$$

Assignments to stacks of $L'$ from stacks of $H$:

$$
\begin{aligned}
(a.r, a.w) &\leftarrow (\text{SF}, c.w) && (3.43)\\
(b.r, b.w) &\leftarrow (\text{SF}, b, c), b.w) && (3.44)\\
(c.r, c.w) &\leftarrow (\text{SF}, b, c), ad.w) && (3.45)\\
(d.r, d.w) &\leftarrow (d.r, d.w) && (3.46)\\
(ab.r, ab.w) &\leftarrow (\text{SF}, b, c), bc.w) && (3.47)\\
(ac.r, ac.w) &\leftarrow (\text{SF}, b, c), acd.w) && (3.48)\\
(ad.r, ad.w) &\leftarrow (cd.r, cd.w) && (3.49)\\
(bc.r, bc.w) &\leftarrow (\text{SF}, b, c), abd.w) && (3.50)\\
(bd.r, bd.w) &\leftarrow (bd.r, bd.w) && (3.51)\\
(cd.r, cd.w) &\leftarrow (ad.r, ad.w) && (3.52)\\
(abc.r, abc.w) &\leftarrow (\text{SF}, b, c), \text{OUT}) && (3.53)\\
(abd.r, abd.w) &\leftarrow (bcd.r, bcd.w) && (3.54)\\
(acd.r, acd.w) &\leftarrow (acd.r, acd.w) && (3.55)\\
(bcd.r, bcd.w) &\leftarrow (abd.r, abd.w) && (3.56)
\end{aligned}
$$

**Assignments for $H' \overset{\triangle\triangle}{\rightarrow} L' \rhd L$**

Assignments to stacks of $L'$ from stacks of $H'$:

$$
\begin{aligned}
(a.r, a.w) &\leftarrow (a.r, \text{SF}, a, c) && (3.57)\\
(b.r, b.w) &\leftarrow (b.r, b.w) && (3.58)\\
(c.r, c.w) &\leftarrow (bd.r, \text{SF}, a, c) && (3.59)\\
(d.r, d.w) &\leftarrow (c.r, \text{SF}, a, c) && (3.60)\\
(ab.r, ab.w) &\leftarrow (ab.r, ab.w) && (3.61)\\
(ac.r, ac.w) &\leftarrow (abd.r, \text{SF}, a, c) && (3.62)\\
(ad.r, ad.w) &\leftarrow (ac.r, \text{SF}, a, c) && (3.63)\\
(bc.r, bc.w) &\leftarrow (bd.r, bd.w) && (3.64)\\
(bd.r, bd.w) &\leftarrow (bc.r, bc.w) && (3.65)\\
(cd.r, cd.w) &\leftarrow (bcd.r, \text{SF}, a, c) && (3.66)\\
(abc.r, abc.w) &\leftarrow (abd.r, abd.w) && (3.67)\\
(abd.r, abd.w) &\leftarrow (abc.r, abc.w) && (3.68)\\
(acd.r, acd.w) &\leftarrow (\text{IN}, \text{SF}, a, c) && (3.69)\\
(bcd.r, bcd.w) &\leftarrow (bcd.r, bcd.w) && (3.70)
\end{aligned}
$$

Assignments to stacks of $L$ from stacks of $H'$:

$$(a.r, a.w) \leftarrow (\text{SF}, a.w) \tag{3.71}$$
$$(b.r, b.w) \leftarrow (\text{SF}, c.w) \tag{3.72}$$
$$(c.r, c.w) \leftarrow (\text{SF}, bd.w) \tag{3.73}$$
$$(d.r, d.w) \leftarrow (d.r, d.w) \tag{3.74}$$
$$(ab.r, ab.w) \leftarrow (\text{SF}, ac.w) \tag{3.75}$$
$$(ac.r, ac.w) \leftarrow (\text{SF}, abd.w) \tag{3.76}$$
$$(ad.r, ad.w) \leftarrow (ad.r, ad.w) \tag{3.77}$$
$$(bc.r, bc.w) \leftarrow (\text{SF}, bcd.w) \tag{3.78}$$
$$(bd.r, bd.w) \leftarrow (cd.r, cd.w) \tag{3.79}$$
$$(cd.r, cd.w) \leftarrow (bd.r, bd.w) \tag{3.80}$$
$$(abc.r, abc.w) \leftarrow (\text{SF}, \text{OUT}) \tag{3.81}$$
$$(abd.r, abd.w) \leftarrow (acd.r, acd.w) \tag{3.82}$$
$$(acd.r, acd.w) \leftarrow (abd.r, abd.w) \tag{3.83}$$
$$(bcd.r, bcd.w) \leftarrow (bcd.r, bcd.w) \tag{3.84}$$

**Assignments for $L \overset{\triangle\triangle}{\rightarrow} S_1 \rhd S_2$**

Assignments to stacks of $S_1$ from stacks of $L$:

$$(a.r, a.w) \leftarrow (a.r, a.w) \tag{3.85}$$
$$(b.r, b.w) \leftarrow (ad.r, \text{SF}) \tag{3.86}$$
$$(c.r, c.w) \leftarrow (c.r, \text{SF}) \tag{3.87}$$
$$(d.r, d.w) \leftarrow (b.r, \text{SF}) \tag{3.88}$$
$$(ab.r, ab.w) \leftarrow (ad.r, ad.w) \tag{3.89}$$
$$(ac.r, ac.w) \leftarrow (ac.r, ac.w) \tag{3.90}$$
$$(ad.r, ad.w) \leftarrow (ab.r, ab.w) \tag{3.91}$$
$$(bc.r, bc.w) \leftarrow (acd.r, \text{SF}) \tag{3.92}$$
$$(bd.r, bd.w) \leftarrow (abd.r, \text{SF}) \tag{3.93}$$
$$(cd.r, cd.w) \leftarrow (bc.r, \text{SF}) \tag{3.94}$$
$$(abc.r, abc.w) \leftarrow (acd.r, acd.w) \tag{3.95}$$
$$(abd.r, abd.w) \leftarrow (abd.r, abd.w) \tag{3.96}$$
$$(acd.r, acd.w) \leftarrow (abc.r, abc.w) \tag{3.97}$$
$$(bcd.r, bcd.w) \leftarrow (\text{IN}, \text{SF}) \tag{3.98}$$

Assignments to stacks of $S_2$ from stacks of $L$:

$$(a.r, a.w) \leftarrow (\text{SF}, b.w) \tag{3.99}$$
$$(b.r, b.w) \leftarrow (\text{SF}, ad.w) \tag{3.100}$$
$$(c.r, c.w) \leftarrow (\text{SF}, c.w) \tag{3.101}$$
$$(d.r, d.w) \leftarrow (d.r, d.w) \tag{3.102}$$
$$(ab.r, ab.w) \leftarrow (\text{SF}, abd.w) \tag{3.103}$$
$$(ac.r, ac.w) \leftarrow (\text{SF}, bc.w) \tag{3.104}$$
$$(ad.r, ad.w) \leftarrow (bd.r, bd.w) \tag{3.105}$$
$$(bc.r, bc.w) \leftarrow (\text{SF}, acd.w) \tag{3.106}$$
$$(bd.r, bd.w) \leftarrow (ad.r, ad.w) \tag{3.107}$$
$$(cd.r, cd.w) \leftarrow (cd.r, cd.w) \tag{3.108}$$
$$(abc.r, abc.w) \leftarrow (\text{SF}, \text{OUT}) \tag{3.109}$$
$$(abd.r, abd.w) \leftarrow (abd.r, abd.w) \tag{3.110}$$
$$(acd.r, acd.w) \leftarrow (bcd.r, bcd.w) \tag{3.111}$$
$$(bcd.r, bcd.w) \leftarrow (acd.r, acd.w) \tag{3.112}$$

**Assignments for $L' \overset{\triangle\triangle}{\rightarrow} S_1 \rhd S_2$**

Assignments to stacks of $S_1$ from stacks of $L'$:

$$(a.r, a.w) \leftarrow (a.r, \text{SF}) \tag{3.113}$$
$$(b.r, b.w) \leftarrow (bd.r, \text{SF}) \tag{3.114}$$
$$(c.r, c.w) \leftarrow (c.r, \text{SF}) \tag{3.115}$$
$$(d.r, d.w) \leftarrow (b.r, b.w) \tag{3.116}$$
$$(ab.r, ab.w) \leftarrow (abd.r, \text{SF}) \tag{3.117}$$
$$(ac.r, ac.w) \leftarrow (ac.r, \text{SF}) \tag{3.118}$$
$$(ad.r, ad.w) \leftarrow (ab.r, ab.w) \tag{3.119}$$
$$(bc.r, bc.w) \leftarrow (bcd.r, \text{SF}) \tag{3.120}$$
$$(bd.r, bd.w) \leftarrow (bd.r, bd.w) \tag{3.121}$$
$$(cd.r, cd.w) \leftarrow (bc.r, bc.w) \tag{3.122}$$
$$(abc.r, abc.w) \leftarrow (\text{IN}, \text{SF}) \tag{3.123}$$
$$(abd.r, abd.w) \leftarrow (abd.r, abd.w) \tag{3.124}$$
$$(acd.r, acd.w) \leftarrow (abc.r, abc.w) \tag{3.125}$$
$$(bcd.r, bcd.w) \leftarrow (bcd.r, bcd.w) \tag{3.126}$$

Assignments to stacks of $S_2$ from stacks of $L'$:

$$
\begin{aligned}
(a.r, a.w) &\leftarrow (d.r, d.w) && (3.127)\\
(b.r, b.w) &\leftarrow (\text{SF}, bd.w) && (3.128)\\
(c.r, c.w) &\leftarrow (\text{SF}, c.w) && (3.129)\\
(d.r, d.w) &\leftarrow (\text{SF}, a.w) && (3.130)\\
(ab.r, ab.w) &\leftarrow (bd.r, bd.w) && (3.131)\\
(ac.r, ac.w) &\leftarrow (cd.r, cd.w) && (3.132)\\
(ad.r, ad.w) &\leftarrow (ad.r, ad.w) && (3.133)\\
(bc.r, bc.w) &\leftarrow (\text{SF}, bcd.w) && (3.134)\\
(bd.r, bd.w) &\leftarrow (\text{SF}, abd.w) && (3.135)\\
(cd.r, cd.w) &\leftarrow (\text{SF}, ac.w) && (3.136)\\
(abc.r, abc.w) &\leftarrow (bcd.r, bcd.w) && (3.137)\\
(abd.r, abd.w) &\leftarrow (abd.r, abd.w) && (3.138)\\
(acd.r, acd.w) &\leftarrow (acd.r, acd.w) && (3.139)\\
(bcd.r, bcd.w) &\leftarrow (\text{SF}, \text{OUT}) && (3.140)
\end{aligned}
$$

### 3.3.2 Parallel-plane stacks

For every unique angle a bisecting plane can have, we create a temporary vertex stack. As discussed in the section 2.8, there are 9 unique plane orientations in total. We assign them indices running from 0 to 8. Following the approach discussed above in the previous section 3.3.1, we assign pop and push stack references for each vertex, edge, and face of a tetrahedron. Every new bisecting face is assigned the parallel-plane stack that matches the plane angle of the face. We define a parallel-plane stack assignment function SF for this.

#### Stack assignments to new bisection faces

The SF function for the parallel-plane approach takes the three face vertices of the bisection face as input. We assign a stack based on the face normal vector calculated using the cross product on two of its edges, $\mathbf{u} = \mathbf{a} - \mathbf{c}$ and $\mathbf{v} = \mathbf{b} - \mathbf{c}$:

$$\mathbf{u} \times \mathbf{v} = (s_1, s_2, s_3)^T \tag{3.141}$$

with

$$
\begin{aligned}
s_1 &= u_2 v_3 - u_3 v_2\\
s_2 &= u_3 v_1 - u_1 v_3\\
s_3 &= u_1 v_2 - u_2 v_1
\end{aligned}
$$

| plane normal vector | mapped stack index |
|---|---|
| $(1,0,-1)^T$ | 0 |
| $(0,1,0)^T$ | 1 |
| $(1,-1,0)^T$ | 2 |
| $(1,0,1)^T$ | 3 |
| $(0,0,1)^T$ | 4 |
| $(0,1,-1)^T$ | 5 |
| $(1,1,0)^T$ | 6 |
| $(0,1,1)^T$ | 7 |
| $(1,0,0)^T$ | 8 |

Table 3.3: Plane normal vector to stack index mapping table.

The dot products of $\mathbf{u}$ and $\mathbf{v}$ with the stack plane's normal vector $\mathbf{u} \times \mathbf{v}$ should both yield 0 when matched. We assign indices to the 9 normal vectors for the unique plane orientations as stated in Table 3.3.

**Testing**

For our tests we adapted the lower bound algorithm LOWERBOUND and the Compacting Greedy algorithm. Both algorithms were adapted for testing to work per plane orientation into a parallel-plane lower bound algorithm and a parallel-plane Compacting Greedy algorithm. First the parallel-plane lower bound algorithm showed that each stack plane had a lower bound of one stack and required no reshuffling between vertex times. Next, traversal tests with the parallel-plane Compacting Greedy algorithm up to refinement level 16 confirmed that indeed one stack per plane orientation sufficed, see Table 3.4. This demonstrates that a constant upper bound exists for only 9 stacks, confirming theorem 2.57.

**Lowering the upper bound**

In an attempt to see if we could reduce the upper bound, we tried to combine the vertex events of two plane orientations onto one stack. Combinations of stack indices 2-6, 2-7, 2-8, 4-8, 5-7, 5-8, 6-8, and 7-8 combine to a parallel-plane lower bound of 1 stack below refinement level 14, see the graph in Figure 3.2.

The possible candidates for triple parallel plane combinations 2-7-8, 2-6-8 and 5-7-8, were tested using the parallel-plane Compacting Greedy algorithm but they did not yield any additional stack reduction. The lower bound for the triple plane stack combinations rose to 3 stacks or more for any combination at refinement level 9 and above. We also tested each of the individual stack combinations for a uniform refined traversals using the parallel Compacting Greedy algorithm. Combination 4-8 was a success, it needs only one stack. This demonstrates that a constant upper bound exists for just 8 stacks. We can explain why vertex events of plane orientations with stack

| | plane stack | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| refinement level | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 4 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 11 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 12 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 13 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 14 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 15 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 16 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 3.4: The lower bound values per stack plane. The lower bound for reshuffling is 0 for all those cases.

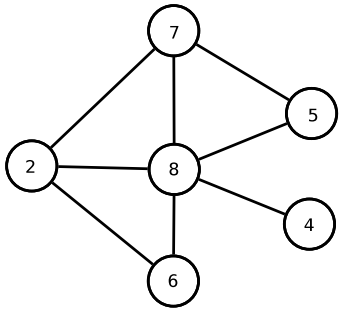


Figure 3.2: The stack-reduction graph. Vertices are parallel-plane stacks identified by their indices. Edges indicate a lower bound exists for the combined vertex events of the connected parallel-plane stacks.

indices 4 and 8 can be placed on the same stack. Faces using stack indices 4 and 8 do not occur in the same tetrahedra in a Sommerville cube. When refining some of the $S$-type tetrahedra in a Sommerville cube into Hill-type tetrahedra, some faces do occur in the same Hill-type tetrahedra. In those cases, the combination of the unrefined siblings $H \triangleright H'$ requires only one vertex to be carried over the bisection face. As a result, we can use the stack we use for popping or pushing that vertex also as the bisection face stack. When refining these Hill-type tetrahedra into Liujoedron tetrahedra, faces using stack indices 4 and 8 again do not occur in the same tetrahedra. Hence, 8 stacks are possible at any refinement level.

Other non-trivial solutions 2-6, 2-7, 5-7 and trivial solutions 2-8, 5-8, 6-8, 7-8 required additional stacks in order to not violate the vertex stack order property. For the parallel-plane stack approach we can conclude no smaller Constant Stack solution than 8 stacks exists.

# Chapter 4

# Stack-based traversals for some bisection-based traversals over convex polyhedra

The approach used to find a Constant Stack solution for the traversal of the SHL-tetrahedra can be generalised and applied to some other bisection-based traversals. Vertex events containing vertex information that are passed over a bisection face in one child polyhedron of $P$ can never cross with vertex events in the other child polyhedron of $P$. These children of $P$ are processed sequentially and vertex information between them is only passed over the bisection face of $P$. Therefore, crossings between the vertex events of one tetrahedra with the vertex events of another tetrahedra may only occur if any bisection faces of their ancestors intersect. Using this, we can find a general upper bound for a palindromic bisection-based traversal over convex polyhedra. For any orientation of a face that occurs only once in any refined element we require one stack. For any face orientation that occurs twice in any refined element, we require three stacks for a multi-scale traversal. We require three stacks for parallel faces because we want to be able to refine our elements. No face orientation will occur more than twice in the faces of any convex polyhedra. Therefore, we can state that:

**Theorem 4.0.1.** *Let $n$ be the size of the set of face orientations occurring in the domain which are unique in every element and $m$ be the size of the set of face orientations occurring in the domain which occur twice in every element. Any multi-scale bisection-based traversal that is palindromic over convex polyhedral elements and their vertices requires at most $n + 3m$ stacks. With $n$ the size of the set of face orientations occurring in the domain which are unique in every element and $m$ is the size of the set of face orientations occurring in the domain which occur twice in every element.*

We will discuss two cases for which this applies other than the Haverkort traversal. The triple bisection of a cube which contains three pairs of parallel cube faces and the triple bisection of a triangular bar who has a mix of parallel and unique face orientations. For

both of these cases, we show a palindromic traversal for their elements and vertices as well as give a receipt for a constant stack algorithm.

## 4.1 Cubes

We take an 8-reptile simple cube. Triple bisection is used to produce its 8 self-similar shapes, see Figure 4.1. Bisection is done for every stage by splitting it on the long side. We start with a cube. We obtain a bar $B$ by mirroring a cube $C$ over one of its faces and reversing its traversal. We obtain a plate $P$ by mirroring the bar over one of its long faces and reversing its traversal. The plate is mirrored again over its largest face and once more the traversal is reversed returning to the cube shape $C$ again.

$$C \to P \to B \to C$$

This approach ensures we have palindromicity for the element traversal for each of the three shapes in a similar fashion as in section 2.5. The vertices of an element are visited in the order in which its children are to be visited after refinement. This yields a solution in line with our solution for the SHL-tetrahedra. Both vertex- and element-information of a bisection face can now be visited in palindromic order during a traversal and we can store this information using a single stack per bisection face.

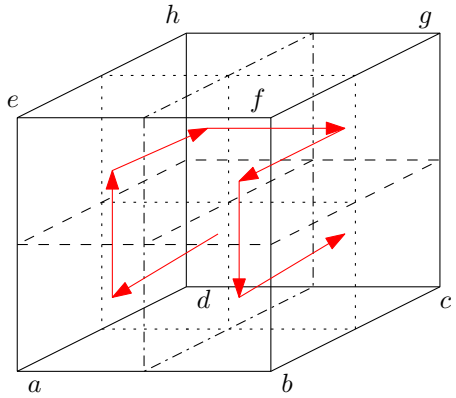


Figure 4.1: A 8-reptile cube by way of triple bisection. The order of the bisections is in first by the dashed-dotted, then the dashed, and then the dotted planes.

The described bisection scheme results into the following refinement and traversal scheme:

$$
\begin{aligned}
C(a,b,c,d,e,f,g,h) &\overset{\Delta\Delta}{\rightarrow} P(a,a\overline{\circ}b,d\overline{\circ}c,d,e,e\overline{\circ}f,g\overline{\circ}h,h) \rhd P'(a\overline{\circ}b,b,c,c\overline{\circ}d,e\overline{\circ}f,f,g,g\overline{\circ}h)\\
P(a,b,c,d,e,f,g,h) &\overset{\Delta\Delta}{\rightarrow} B(a,b,c,d,a\overline{\circ}e,b\overline{\circ}f,c\overline{\circ}g,d\overline{\circ}h) \rhd B'(a\overline{\circ}e,b\overline{\circ}f,c\overline{\circ}g,d\overline{\circ}h,e,f,g,h)\\
P'(a,b,c,d,e,f,g,h) &\overset{\Delta\Delta}{\rightarrow} B(a\overline{\circ}e,b\overline{\circ}f,c\overline{\circ}g,d\overline{\circ}h,e,f,g,h) \rhd B'(a,b,c,d,a\overline{\circ}e,b\overline{\circ}f,c\overline{\circ}g,d\overline{\circ}h)\\
B(a,b,c,d,e,f,g,h) &\overset{\Delta\Delta}{\rightarrow} C'(a\overline{\circ}d,b\overline{\circ}c,c,d,e\overline{\circ}h,f\overline{\circ}g,g,h) \rhd C(a,b,b\overline{\circ}c,a\overline{\circ}d,e,f,f\overline{\circ}g,e\overline{\circ}h)\\
B'(a,b,c,d,e,f,g,h) &\overset{\Delta\Delta}{\rightarrow} C(a,b,b\overline{\circ}c,a\overline{\circ}d,e,f,f\overline{\circ}g,e\overline{\circ}h) \rhd C'(a\overline{\circ}d,b\overline{\circ}c,c,d,e\overline{\circ}h,f\overline{\circ}g,g,h)
\end{aligned}
$$

We consider the bars and the plates as scaled per axis, but not rotated, versions of the cube $C$ with the labels in the same corners as for cube $C$. Furthermore we know $C'$ is a reverse mirrored over-any-of-its-cube-faces version: it is a rotated version of $C$ such that: $C'(a,b,c,d,e,f,g,h) = C(c,d,a,b,g,h,e,f)$. Which gives us a second set of refinement rules. Together they define the full traversal and refinement scheme in such terms that we do not need to calculate any face angles to determine the stacks. An element's type and and whether its refinement level is odd suffices to determine the bisection face stack for vertex. The basic traversal signature is that of the Hilbert space-filling curve, see the arrows in Figure 4.1. Proof of palindromicity can be found by constructing larger $C$'s and $C''$s from smaller $C$'s and $C''$s using cubes, bars and plates, see section 2.5 for an example.

As every face of a cube is parallel with another face of that cube we can't use one stack per face orientation. However, we can assign 3 stacks per face orientation. Doing so ensures that we can give every pair of parallel faces of a cube a different stack at any refinement level. We define a cube with its faces perpendicular to the axes. We assign one face pair stacks 1 and 3. The stack value for the face orientation which is not being used by the cube is assigned to the bisection face whenever we bisect. For the root cube it means its bisection face gets assigned stack 2. An example of the stack pattern for parallel faces is found in Table 4.1.

| refinement level | stacks | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | | 3 |
| 2 | 1 | | | | 2 | | | | 3 |
| 3 | 1 | | 3 | | 2 | | 1 | | 3 |
| 4 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |

Table 4.1: Stack pattern for parallel faces.

This approach ensures that each set of parallel faces perpendicular to the x,y, and z planes have different stack sets and we can suffice with a constant number of temporary vertex stacks of 9.

## 4.2 Sierpínski bar

We take a 8-reptile triangular prism. Its triangular faces are isosceles. Triple bisection is used to produce its 8 self-similar shapes. Bisection along the bar is first done by splitting it on the long side. Then two stages are following the rules for the Sierpínski curve after which we obtained 8 self similar shapes of the original bar shape. The triple bisection cycle goes from a slim $S$ to a fat $M$ to a medium $M$ thick bar.

$$S \to F \to M \to S$$

We apply the same approach as for the cube. We mirror and reverse the traversal in a Sierpinsky pentahedron to construct a larger one. This approach ensures we have palindromicity for the element traversal for each of the three shapes in a similar fashion as in section 2.5. Palindromicity for the vertices of a face is easily achieved by visiting the vertices in the order they would be visited when further refining the elements until a non-ambiguous solution is found. As a result both vertex- and element-information of a bisection face are visited in palindromic order during a traversal and again we can store this information using a single stack per bisection face.

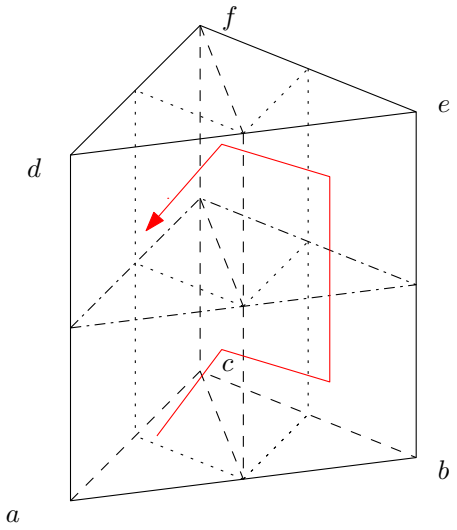


Figure 4.2: The Sierpínski bar: A 8-reptile triangular prisma by way of three bisections. The order the bisections is in first by the dashed-dotted, then the dashed, and then the dotted planes.

Our traversal and refinement scheme is defined as follows:

$$N(a,b,c,d,e,f) \overset{\triangle\triangle}{\rightarrow} F(a,b,c,a\overline{\circ}d,b\overline{\circ}e,c\overline{\circ}f) \triangleright F'(a\overline{\circ}d,b\overline{\circ}e,c\overline{\circ}f,d,e,f) \quad (4.1)$$

$$F(a,b,c,d,e,f) \overset{\triangle\triangle}{\rightarrow} M(c,a,a\overline{\circ}b,f,d,d\overline{\circ}e) \triangleright M(b,c,a\overline{\circ}b,e,f,d\overline{\circ}e) \quad (4.2)$$

$$F'(a,b,c,d,e,f) \overset{\triangle\triangle}{\rightarrow} M'(b,c,a\overline{\circ}b,e,f,d\overline{\circ}e) \triangleright M'(c,a,a\overline{\circ}b,f,d,d\overline{\circ}e) \quad (4.3)$$

$$M(a,b,c,d,e,f) \overset{\triangle\triangle}{\rightarrow} N(b,c,a\overline{\circ}b,e,f,d\overline{\circ}e) \triangleright N'(c,a,a\overline{\circ}b,f,d,d\overline{\circ}e) \quad (4.4)$$

$$M'(a,b,c,d,e,f) \overset{\triangle\triangle}{\rightarrow} N'(b,c,a\overline{\circ}b,e,f,d\overline{\circ}e) \triangleright N(c,a,a\overline{\circ}b,f,d,d\overline{\circ}e) \quad (4.5)$$

Furthermore we know $N'$ is a reversed-traversal, mirrored over-any-of-its-faces version of $N$: it is a rotated version of $N$ such that: $N'(a,b,c,d,e,f,g,h) = N(e,d,f,b,a,c)$. Which gives us a second set of refinement rules. The generated Sierpínski bars are pentahedra with two parallel faces. The remaining non-parallel faces all have different face orientation. This allows us to apply the one stack per face orientation approach. For the pair of parallel faces we can not assign unique stacks per face orientation. However, we can use the approach for the 8-reptile cube: three stacks per pair of parallel faces. This ensures that any parallel face of the bar at any refinement level has a different stack. Showing that we can suffice with a constant number of temporary vertex stacks of 7 stacks in total. If so desired, the traversal and refinement scheme can be written in a similar fashion as for the cubes. Doing so, we can determine the bisection face stack for each vertex from its element's type and whether the refinement level is odd.

# Chapter 5

# Discussion

Current generation CPUs are capable to process data much faster than the main memory can supply it. CPUs use caches to increase their performance. When a CPU needs to access data, it checks first if it is available in the cache before retrieving it from the main memory. Caches are designed to be accessed faster and retain previously used data to increase data-access. Modern CPUs can have several dedicated caches. Usually one may expect separate caches for data, instructions, and virtual-to-physical address translations. In order to keep a CPU as good as is possible supplied with the required data for processing a traversed element our stack-based traversals should work well together with the data-caches.

In some cases a data-access hierarchy may exists with multiple cache-levels with different sizes and latencies. Some of those cases may even include external disk- or networked-based data-stores. Generally such an hierarchy goes from small and fast data-storage capacity to large and slow capacity. At some point in that hierarchy, cache or data-store latencies and bandwidth limit the processing speed of the CPU. It is for this issue that our traversals are designed; to improve the temporal and spatial locality of our data and its access, increasing the computational performance. Temporal locality implies that access to a data-element is clustered around a moment in time and reduces the number of times a data-element needs to be retrieved from the main memory. Spatial locality implies that subsequent needed data is stored in nearby memory addresses which in many computer hardware configurations allows for lower latencies. For example, main memory types like DDR have better latencies when accessing data with a high spatial locality due to a feature called fast paging.

We first discuss CPU caches and their memory hierarchy. Followed by their impact on our traversals and how we expect to do in comparison to loop blocking. Loop blocking is another cache-use optimization strategy, see Kowarschik and Weiss [14]. Afterwards, we briefly discuss partitioning and the applicability of our traversals for GPUs.

## 5.1 Caches and modelling cache-efficiency

Sen [15] and Frigo et al. [16] discuss cache-efficiency. In their models they assume a two-level memory hierarchy: the upper level memory is fast and small and can't contain all the data, the lower level memory is slow and large. Data is exchanged between the two memory levels using a fixed block size. Their models are very similar to the external memory model discussed by Aggerwal et al. [17]. Some differences however do exist. In the external memory model the time complexities of the program are considered negligible compared to the latencies for block-IO, while this is rarely justified for the cache-efficient model. Furthermore, full control usually exists over the IO-operations accessing external memory while CPU-caches have a hardware-dictated and implemented cache-eviction policy.

CPUs usually have a small on-chip cache, often called a level one (L1) cache, to provide data within one or two clock cycles of the CPU. This is to prevent it from idling until needed data is fetched from the memory. Due to the die size of a CPU the L1 cache size is bound by its maximum signal delays. The typical size of a L1-cache for modern CPUs in 2015 is 64 kilobyte or less. CPUs usually have an additional cache level, level 2 (L2), and sometimes even an L3 cache. On-chip L2 caches usually have a latency of 5 to 10 clock cycles and off-chip L2/L3 caches usually have a latency of 10 to 20 CPU cycles. Caches are smaller and faster the nearer they are to the CPU in the memory hierarchy. Data between a cache and the main memory is exchanged in blocks called a cache line.

A fully associative cache can map any cacheline to any word-aligned main memory address block. However, many caches are not fully associative. A $n$-associative cache can store data from the main memory only in $n$ different cache lines. This limits the full-use of the available cache memory for some use cases. Another feature of modern CPUs is the existence of software and hardware pre-fetching to fill their data-cache. Even though pre-fetching requires overhead CPU time and bandwidth wise, it may improve the overall execution time of a program by retrieving data from the main memory while the CPU is processing data, see van der Wiel [18].

Typically L1-cacheline are 64 bytes large. An L1-cache of 64 kilobytes is therefore typically a 1000 times larger than a cacheline. When randomly accessing data from the main memory the change exists that useless data bytes appear in the cache if the cachelines do not align with the word boundaries of the computational data required by the CPU. This wastes cache-space and reduces the effective cache size and ultimately the cache hit-ratio.

Pure row/column traversals can be replaced by loop blocking to increase the spatial and temporal locality. Instead of processing a grid in row/column order, a grid is divided into smaller blocks that align with the cacheline sizes and fit well into the cache, see Kowarschik and Weiss [14]. Loop blocking however is highly cache-aware. The algorithms need to be tuned for the cache size and the cacheline size. One cache-optimization technique mentioned by Kowarschik and Weiss uses a tiling approach based on a space-filling curve. However there is no mention of using stacks for the storage of temporary data. Tiling is then still cache-aware.

We define the size of a cacheline as $L$ words and the size of a cache as $C$ words. For each continuous data block that does not align with a cacheline we waste between 1 and $L-1$ words. For a cube-shaped block we waste more than $(L-1)n$ cachelines with $n$ the number of continuous data blocks. In a multi-level refined grid we may assume that even more data blocks are misaligned compared to a uniform refined grid. In contrast, a constant stack configuration of $m$ stacks mostly wastes only $m$ cachelines, one per stack when none of the stacks fit completely into the cache. Effectively, a constant number independent of the stack and cache size. Therefore we expect a stack-based traversal to be cache-oblivious.

Using their ideal-cache model, Frigo et al. [16] proof that an optimal cache-oblivious algorithm designed for two levels of memory is also optimal for multiple levels. Our stack algorithms are expected to perform well in a multi-level memory-hierarchy. However knowledge of the cache implementation is still needed if the cache is not fully associative. Selecting the proper memory address ranges used for a stack can in such a case be of great influence on the performance. This holds both for stack-based algorithms and loop-blocking-based algorithms. The cache policy is also relevant, but due to our traversal we know at what time which data-elements will be popped from which stacks. We can timely execute software pre-fetching while processing data without wasting cache memory and try to use the cache as efficient as possible.

## 5.2 The performance of the Constant Stack solution for large CPU caches

The maximum processing speed of a traversal is limited by the speed a cache can fetch needed data from the main memory: its bandwidth. Let's assume the following conditions:

- Optimal caching.
- CPU time to process an element is negligible compared to the latencies for fetching data from the main memory.
- Mentioned latencies and bandwidth assume a simplified form of write-back caching; no extra latency is incurred for cacheline writes to the main memory.
- The amount of data for processing an element is much less than the cache size.

We assume elements and vertices are $D$ words large and the traversal is sufficiently refined. At an uniform refinement level $i$ we have $n_e$ elements with $n_e = 2^i$. At that refinement level we also have $n_b$ bisection faces with $n_b = 2^i - 1$: for every bisection of a tetrahedron we add one tetrahedron to the count. A vertex in traversal can be part of up to 48 tetrahedra, see section 2.8. In a space-filling configuration of Sommerville cubes we have on average 30 tetrahedral elements sharing a vertex. For Hill cubes in a space-filling configuration we have on average slightly more than 20 tetrahedral

elements sharing a vertex and for Liujoedron we have on average 16.8 tetrahedral elements sharing a vertex. Each tetrahedron has 4 vertices which gives us fractions of $\frac{5}{21}$ and $\frac{2}{15}$ vertices per element. Then, if we assume a perfect cache with infinite size and access speed we can calculate the average processing internal speed for elements that do not have a vertex in an outer face of the root tetrahedron. Our average (internal) processing speed $\Phi_{\text{avg}}$ in elements per second for an infinite cache is then bounded by

$$\frac{B}{(1+\frac{5}{21})D} \leq \Phi_{\text{avg}} \leq \frac{B}{(1+\frac{2}{15})D} \tag{5.1}$$

for a uniform refined traversal. $\Phi_{\text{avg}}$ depends on the refinement level type. Sommerville levels have the fastest processing speed and Liujoedron levels the slowest. When all vertices needed are present in the cache we need to read only the element data. If the cache is much smaller than our traversal data and the cache is full whenever we read an element or vertex, we must fetch at most 4 vertices and 1 element worth of data: $5D$. Our processing speed for an individual element, $\Phi$, is:

$$\frac{B}{5D} \leq \Phi \leq \frac{B}{D} \tag{5.2}$$

If the computation time complexity does become important, the processing speed will become less than $\Phi$. However when pre-fetching data in the background, we will only incur a time penalty for the pre-fetching instructions. Once the time for processing an element is much larger than the latencies involved for fetching data from the main memory, the unused bandwidth can be used to run multiple cores. Hence, if the calculation of an element takes 4 times longer than fetching missing data from the main memory we could run more cores, sharing the bandwidth to the main memory. While L1-caches are usually dedicated per core, L2 and L3-caches usually are not.

A lower bound for the average performance can be found also. When using a finite cache we know that we need only to read stack data again that does not fit into the cache. Furthermore, the cache stores only one copy of every vertex data and the tops of the stacks always contain the next temporary vertex data needed. $D$ words of cache space are used for each of the 4 streams that needed for the input/output-streams for elements and vertices. We can therefore store $\lfloor C/D \rfloor - 4$ vertex data for reuse. Table 5.1 gives the number of vertices and elements for the first 12 refinement levels.

| refinement level | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n_v$ | 4 | 5 | 7 | 10 | 14 | 22 | 37 | 55 | 95 | 185 | 285 | 525 | 1137 |
| $n_e$ | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 |

Table 5.1: Number of vertices ($n_v$) and elements ($n_e$) as function of refinement level for an uniform refined $S$-type root tetrahedron.

Starting refinement level 4 there are more elements than vertices. This is expected as an internal vertex, a vertex not part of any outer face of the root tetrahedron, is part

of 8 to 48 tetrahedra. Hence, for $j \geq 4$ we can store the vertices for at least a tetrahedral subtree of size $j = \lfloor \log_2(\frac{C}{D} - 4) \rfloor$ levels large inside the cache. In a traversal we have $2^{i-j}$ subtrees. With $n_v(j) \leq 2^j$ and $n_e(j) = 2^j$ being the number of vertices and elements respectively in a subtree of refinement level $j$ we need to fetch at most $(n_v(j) + n_e(j))D$ data per subtree, an average processing speed of

$$\frac{B}{(1 + \frac{n_e(j)}{2^j})D}$$

elements per second for the subtree with $2^j$ elements. Looking at the numbers in Table 5.1 for $j = 11$, an $L$-type level, we find that we are within 1.4% of the processing speed for $L$-type levels following equation 5.2 for an infinite cache. For $j = 12$, an $S$-type level, we find that we are within 12% of the processing speed for $S$-type levels following equation 5.2 for an infinite cache.

The cache-hit rate for traversing a subtree with $j = 11$ is 93.6%: 2048 elements are traversed, each with 4 vertices, so the 525 vertices in the subtree will be accessed 8192 times. For $j = 12$, the cach-hit ratio is 93%. At refinement level 14 the cache-hit ratio is 94.8% as there are 3417 vertices. As most successive subtrees in the traversal have a coinciding face, the average processing speed and cache-hit ratio will be even higher for a cache capable of storing a subtree of height $j$. Our analysis here does not take the amount of cache that is lost when the data blocks do not align with the cacheline size. However when using a small and constant number of stacks only a tiny and known amount of cache is lost: our analysis holds well enough for in the case of the Constant Stack approach for the Haverkort traversal if $8D \ll C$ .

In comparison, it is expected that loop blocking performs similar in some cases. However, once the time complexity of the CPU becomes a relevant factor, we find that for loop blocking we need to use half of the cache for fetching a new loop block in the background while processing another loop block in the other half of the cache; this to optimize the performance. Our stacks require far less blocks to be fetched in the background. A smaller cache requires smaller loop blocks and increases the number of vertices on the border of a loop block which will decrease the performance. More vertices must be read multiple times into the cache and more cachelines will not be aligned.

Our stack-based solution is cache-oblivious, suitable for multi-level cache hierarchies and maintains its performance for multi-level refined grids. It also allows for fast and efficient space-filling curve based (re)partitioning. For all of these reasons mention above, we expect that a constant stack-based solution in most cases can compete and outperform a numerical simulation using loop blocking under real world circumstances due to its more efficient use of the cache size and cache-obliviousness.

## 5.3 Alternative stack-assignement options

Snel [19] researched using stacks in arbitrary discretizations in two and three dimensions. Arbitrary discretization would allow a grid that is conforming with inner and

outer domain boundaries. This could increase the accuracy of a numerical simulation while using less elements. However our multi-scale grids when using the Constant Stack approach may be warped to achieve conforming boundaries. When local warping would distort the elements to strong for accurate simulation, we can refine the grid locally and warp afterwards. Which of the two is more efficient, is difficult to say without further analysis and testing.

Snel builds palindromic traversals by visiting adjacent elements multiple times. He calculates an element's the next time step state only once. The additional visits to the elements reshuffle the vertex data to maintain palindromicity. For 2D, he reports that calculating a palindromic traversal solution for arbitrary discretizations requires a time complexity of $O(n)$ with $n$ the number of elements. However, he uses doubling of the traversal length in 2D: he visits every element twice. For 3D, he reports two approaches. One is based on the overlap graph of the vertex events and the other is based on interval graph colouring of the vertex event. The interval graph colouring, when using a breadth first traversal, performs best. A disadvantage is the number of stacks needed, they are unbounded.

Basically his approach needs to calculate a graph describing the vertex events and determine a stack compatible traversal. For a set of vertices $E$ and elements $V$ that are involved calculating it requires a time complexity of $O(V \log V + E)$. Afterwards, assigning the vertex events to $m$ stacks is done using an iterated greedy approach. Which, after sorting the vertex events, requires only an amount of time that is linear with the number of elements. He reports to have found a highly efficient solution – while not being optimal – that can achieve a cache-miss ratio as low as 5%. All this is not needed for our Constant Stack solution, assignments are done on-the-fly and take $O(1)$ per vertex with a similar or better cache-miss ratio.

Snel does not discuss the applicability of his approach for a dynamic refinement. It is unclear whether this can be done efficiently. In the case where data fetches to the main memory are the dominant factor in the processing speed, we expect our Constant Stack approach to outperform his 3D solution. While we may require smaller elements, we visit elements only once. Our solution needs less fetches from the main memory. Furthermore, due to the fact that palindromicity is built-in our refinement scheme we can apply dynamical refinement with a time complexity of $O(1)$ per refinement operation: for the stack assignment and the refined traversal construction. Any refinement- and stack-assignment that does not scale linear with the number of elements and requires more than local data will not scale well on a distributed computing cluster.

## 5.4 Partitioning

A space-filling-curve-based traversal is well suited to be partitioned using Inverse Space-filling Partitioning, see Pilkington and Baden [7]. The basic approach of ISP is to map a space-filling curve to a computational grid and partition the space-filling curve into equal segments. The grid partitions are then obtained by inverse mapping of the partitioned space-filling curve. The algorithm requires a low amount of memory and time

complexity for the (re)partitioning and load-balancing of numerical simulations. In case elements require different computational loads, weights can be used to improve the load balancing, see Harlacher et al. [4].

There is evidence, in the articles of Dennis [3] and Harlacher et al. [4], to assume that inverse space-filling partitioning (ISP) can outperform the currently popular multi-level graph partitioning algorithms for numerical simulations if a good match between the numerical grid and the used space-filling curve approximation exists, see Schamberger [2], Dennis [3] and Harlacher et al. [4]. Our space-filling-curve-based grid refinement and traversals allow for a perfect match.

Multi-level graph partitioning algorithms operate by first coarsens a graph in steps, next partitioning the coarsened graph and finally de-coarsens the reduced graph again. This then results in a partitioning of the full graph. Algorithms using this approach are called multilevel graph partitioning schemes and can be used to minimize the communication volume between partitions, see Karypis and Kumar [6].

Compared to METIS, a software package for multi-level graph partitioning, ISP however was stated to be a lot faster and requiring a lot less memory, see Schamberger and Wierum [2]. On the other hand, ISP does no minimization of the communication volume while multi-level graph partitioning can do so. However this minimization effect is inherent in the shape and localization properties of a space-filling curve approximation and it does not seem to create a huge disadvantage for ISP as shown in the results of Dennis [3] and Harlacher et al. [4]. This even despite the fact that METIS produces a smaller communication volume unless a high number of partitions is created, see Schamberger and Wierum [2].

One not only needs to exchange the shared boundary data between partitions, it needs to be stored cache- or IO-efficiently. For a traversal that needs a constant number of $m$ stacks, one can store the boundary data of each neighbouring partition boundary data for the next traversal in at most $m$ stacks. Due to the bisection-based refinement scheme boundary data that needs to be shared is generated at a much lower pace than that of the pace a traversal visits its elements. Transporting this boundary data requires therefore a much slower bandwidth than that required for the traversal itself. With the current size of internal memory available to computing nodes in a computing grid we expect the boundary data to be transmitted to be orders less than that of a partition itself and not to be a limiting factor for running time of a simulation. We expect our to traversals work well with ISP and outperform METIS and most alternative partitioning algorithms. We base this primarily on the fact that ISP can be done faster and with less memory than multi-level graph partitioning.

### 5.4.1 GPU-based traversals

Current graphics processing units (CGUs) can have thousands of cores that can process data in parallel. Similar as for CPUs, temporal and spatial locality play an important role. A well-known GPU-architecture is that of FERMI by NVidia. In it, the cores are organized in a streaming multi-processor (SM) existing out of 16 or 32 cores. Each core

has its own L1-cache coherent per SM and a unified L2-cache that is coherent over the GPU-chip. Each SM has 64 KB of on-chip memory. This memory can be configured as 48 KB of shared memory with 16 KB of L1-cache or as 16 KB of shared memory with 48 KB of L1-cache. Threads in a thread block use the shared memory for communication, data sharing, and result sharing between threads within a thread block. A collection or grid of thread blocks can share results using the global memory space after a kernel-wide global synchronization is called. In the more recent Kepler-architecture even up to 1024 threads can be run within in a thread block.

For our stack-based traversals the total sum of data from all the input, output, and temporary stacks is constant during a traversal; assuming one is not dynamically refining it. When one does want to refine dynamically, one would need to reserve memory space for storing additional elements in the shared memory. Were we to refine any element during a traversal at most once, approximately double the memory space is needed. In any case, we can run one micro-traversal per core. Each micro-traversal being a tiny partition of the full traversal. We can apply our analysis of section 5.2 also for GPUs: it is possible to achieve a performance close to the maximum average performance for each micro traversal in a core with its own cache.

Partitions running inside a thread block that are sequential in the traversal can exchange some of their boundary data using shared memory access. Boundary data exchanged between thread blocks should be shared using global memory space. A form of multi-level partitioning might be needed to reduce the latencies involved for data exchange between thread blocks and optimize use of the available bandwidth between components in the memory hierarchy. Our stack-based traversals are therefore well-suited to use with CPUs and GPUs.

# Chapter 6

# Conclusion

The Haverkort element traversal and refinement scheme presented here is based on the bisectioning of 8-reptile SHL-tetrahedra. We proved the palindromicity of the element traversal over its bisection faces and determined a palindromic vertex traversal orders for storing temporary vertex information on stacks. For this traversal we initially found a stack-assignment solution that required only 9 stacks to store temporary vertex data during a traversal. The, so-called, parallel plane approach uses the same stack for all bisection faces with the same plane orientation, keeping it suitable for traversing multi-scale refined grids. We determined experimentally that two stacks can be merged into one, yielding a 8-stack-assignment solution for which we gave a theoretical explanation. Eight is also the greatest lower bound for the parallel plane approach. Theoretically, we determined that a general lower bound of 5 stacks exists when using the same stack for a single bisection face and confirmed this experimentally. A greater general lower upper bound less than the greatest lower bound for the parallel plane approach may exists however.

Our theoretical lower and upper bounds are independent of the refinement level value, but not of its level-type. We show that experimental validation of the lower and upper bounds up to refinement level 12 implies also that the bounds hold for any higher refinement level. The lower bound of 5 stacks was verified experimentally up to level 20 without the requirement of using the same stack for a single bisection face. The upper bounds were verified experimentally up to level 16.

The parallel plane approach was used to implement a stack-assignment algorithm, the Constant Stack algorithm, which has a time complexity of $O(1)$ and is expected to achieve a cache-miss ratio of less than 5.2% if 525 ore more vertices fit into the cache.

The approach for obtaining a constant-number-of-stacks solution was discussed for two other suitable 8-reptile polyhedra. Traversal and refinement schemes are given for an 8-reptile bisected cube and an 8-reptile bisected triangular prism. These are expected to need 9 and 7 stacks, respectively.

Our traversal uses stacks for storing the input, output, and temporary data of vertices and elements. As a result our stack-based traversal solution is cache-oblivious, suitable for multi-level cache hierarchies, and maintains its performance for multi-level

refined grids. It also allows for fast and efficient space-filling-curve-based (re)partitioning. We expect that a constant-number-of-stacks solution can compete with and outperform a numerical simulation using other cache-optimization techniques based on loop blocking when running on CPUs or GPUs.

# Bibliography


[1] M. Bader, *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, vol. 9 of *Texts in Computational Science and Engineering*. Springer-Verlag, 2013.

[2] S. Schamberger and J.-M. Wierum, "Partitioning finite element meshes using space-filling curves," *Future Gener. Comput. Syst.*, vol. 21, pp. 759–766, May 2005.

[3] J. M. Dennis, "Partitioning with space-filling curves on the cubed-sphere.," in *IPDPS*, p. 269, IEEE Computer Society, 2003.

[4] D. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf, "Dynamic load balancing for unstructured meshes on space-filling curves," in *Proc. of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS) Workshops & PhD Forum, Shanghai, China*, pp. 1655–1663, IEEE Computer Society, May 2012. Workshop on Large-Scale Parallel Processing.

[5] T. Weinzierl and M. Mehl, "Peano; a traversal and storage scheme for octree-like adaptive cartesian multiscale grids," *SIAM J. Sci. Comput.*, vol. 33, pp. 2732–2760, Oct. 2011.

[6] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, Dec. 1998.

[7] J. Pilkington and S. Baden, "Partitioning with spacefilling curves," Tech. Rep. CS94-349, Univ. of Calif. at San Diego Dept. of Computer Science and Engineering, 1994.

[8] H. J. Haverkort, "No acute tetrahedron is an 8-reptile," *CoRR*, vol. abs/1508.03773, 2015.

[9] J. Kynčl and Z. Safernová, "On the nonexistence of k-reptile simplices in $\mathbb{R}^3$ and $\mathbb{R}^4$," in *The Seventh European Conference on Combinatorics, Graph Theory and Applications* (J. Nešetřil and M. Pellegrini, eds.), vol. 16 of *CRM Series*, pp. 191–196, Scuola Normale Superiore, 2013.

[10] A. Liu and B. Joe, "On the shape of tetrahedra from bisection," tech. rep., Math. Comp, 1994.

[11] M. J. M. Hill, "Determination of the volumes of certain species of tetrahedra without employment of the method of limits," *Proceedings of the London Mathematical Society*, vol. s1-27, no. 1, pp. 39–53, 1895.

[12] D. M. Y. Sommerville, "Space-filling tetrahedra in euclidean space," *Proceedings of the Edinburgh Mathematical Society*, vol. 41, pp. 49–57, 2 1922.

[13] H. Maehara, "Some simplices other than hill-simplices," *Beiträge zur Algebra und Geometrie / Contributions to Algebra and Geometry*, vol. 55, no. 2, pp. 429–432, 2014.

[14] M. Kowarschik and C. Wei, "An overview of cache optimization techniques and cache-aware numerical algorithms," in *Algorithms for Memory Hierarchies Advanced Lectures, volume 2625 of Lecture Notes in Computer Science*, pp. 213–232, Springer, 2003.

[15] S. Sen, S. Chatterjee, and N. Dumir, "Towards a theory of cache-efficient algorithms," *J. ACM*, vol. 49, pp. 828–858, Nov. 2002.

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, (Washington, DC, USA), pp. 285–, IEEE Computer Society, 1999.

[17] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, Sept. 1988.

[18] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, pp. 174–199, June 2000.

[19] H. Snel, "Cache-efficient algorithms for finite element methods on arbitrary discretizations," master thesis, 2015.

# Appendix A

# Paper tetrahedra

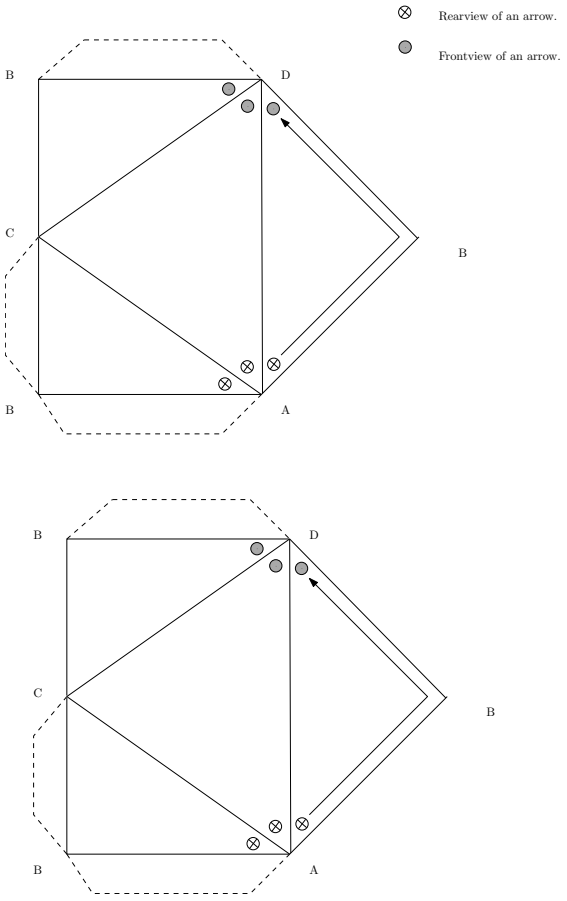


Figure A.1: Print, cut and glue for an S-type paper tetrahedra.