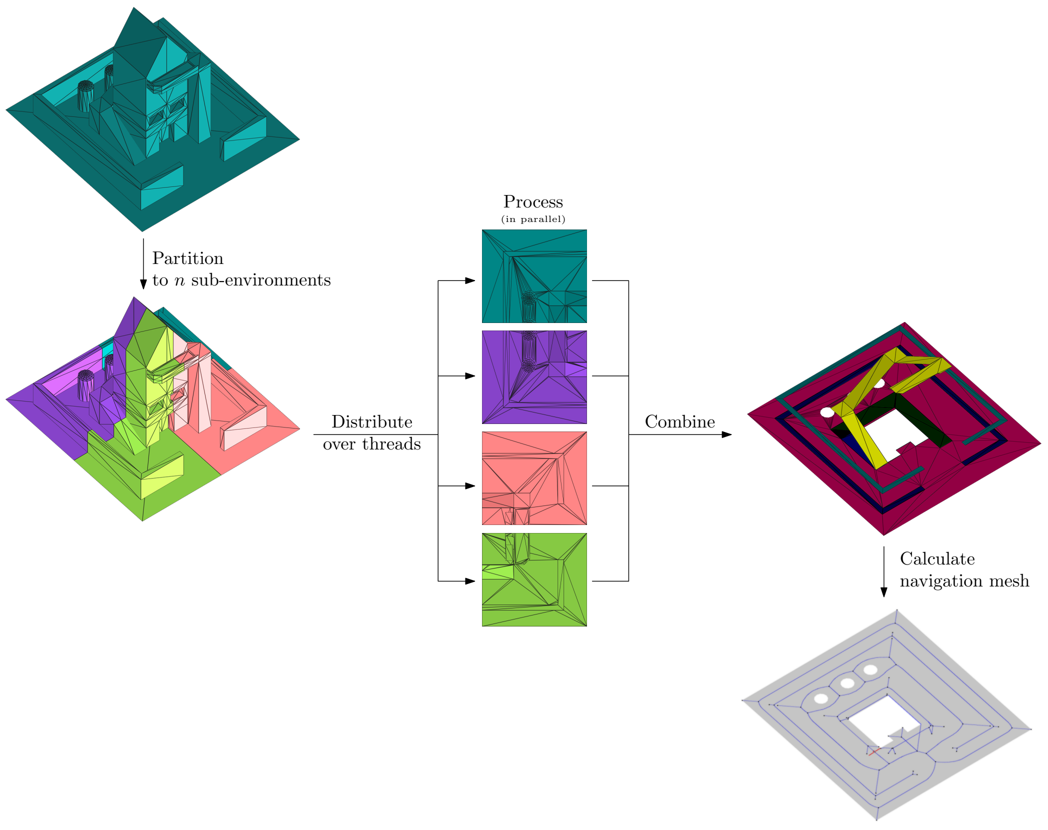
Utrecht University

Master Thesis (ICA-3858103)

Parallelization and optimization of the walkable area mesh generating pipeline



Supervisor
Dr. Roland J. Geraerts

Author
Ricky van den Waardenburg

Date:
March 18, 2019

Abstract

Modern crowd simulation is performed by using a navigation mesh to allow agents to plan paths in the environment. By automatically generating a navigation mesh from the 3D environment the walkable area, i.e. the area agents can navigate on, can be extracted. In the paper by A. Hillebrand [Hil+16] and a thesis by M. Polak [Pol16] a filtering pipeline was proposed that extracts the walkable areas from an input environment by applying filters sequentially. For non-complex environments, this yielded low execution times and exact navigation meshes. However for complex, large environments this approach struggles to perform well in terms of execution times and memory consumption. Consequently, only small 3D environments with low numbers of polygons can be handled in practice.

This thesis proposes an extension on the filtering pipeline [Pol16], which exploits the parallel nature of current computers and networks, to tackle these problems. A parallel filtering pipeline is introduced which subdivides the original environment into smaller partitions and distributes these over available resources (threads and/or computing nodes) to solve the walkable area extraction process in a parallel and/or distributed environment. Several partitioning methods are proposed and explored to find optimal load balancing and low execution times on big and/or complex environments.

Additional methods (and their shortcomings) of gaining low execution times are also explored, such as rounding coordinates. The problems with rounding exact coordinates to non-exact coordinates are explored thoroughly. Additionally, algorithms (or solutions) are proposed to grant compatibility between an exact environment and rounded 3D data.

We explore the scalability, the speed-ups, the efficiency and the correctness of the proposed parallel pipeline. For large environments, the speed-ups are substantial of up to 15× faster than the original sequential pipeline. For small environments the speed-ups are less impressive of up to 2× faster. The proposed pipeline scales consistently well with up to 8 cores and provides a mesh comparable in quality to the original pipeline.

Contents

1 Introduction

In modern, state-of-the-art simulations, it is becoming increasingly important to simulate realistic crowds and plan realistic paths of characters (**agents**). Extracting and processing an environment mesh into a navigation mesh for these agents to be able to traverse is a critical component to get a high-quality simulation. A high amount of research has been put into generating navigation meshes from a 3D polygonal environment, which can be broadly categorized into two groups: voxel-based methods and exact methods. [Van+16]

In voxel-based methods, generating a walkable environment from a polygonal 3D environment is less complex compared to an exact method in terms of computing speed. Due to the nature of the voxelization process, however, its precision and efficiency in computing the navigation mesh depends to a certain degree on the grid resolution. [Van+16] In contrast, calculating a navigation mesh exactly has the highest quality and preserves fine details in the environment. In this technique, constructing the navigation mesh exactly requires pre-processing of the 3D environment [Van+16]. The trade-off for the quality offered by the exact navigation mesh is in the processing time and memory usage. Current implementations of 3D polygon processing pipelines are run sequentially, which do not make optimal use of the resources in a state-of-the-art set-up.

Modern parallel computing comes in several shapes and sizes: from SIMD vector units, GPGPU, multiple processor cores using shared memory, and, on a larger scale, to distributed systems and cloud computing. State-of-the-art personal computers have made strides over the years and nowadays have 2-8 cores. Even more in higher-end computers using all resources available to speed up the existing preprocessing pipeline is a natural step. Distributing the preprocessing pipeline over multiple computing nodes would reduce the memory usage and execution time of the problem. In practice, however, a non-optimal problem division method tends to drag down the parallel performance far below the sequential performance. [DFR93] Additionally, such an implementation highlights many of the issues associated with distributed computing: splitting up and assigning the input to various systems, scheduling and running computation tasks on the available nodes and coordinating the necessary communication between tasks.

1.1 Contributions

In this research the existing single-threaded preprocessing pipeline by Polak [Pol16] and Hillebrand [Hil+16] is extended. Contributing a parallelized (and optimized) framework that reaches an execution time at which the end-user will be able to tweak the navigation mesh in an iterative process (e.g. world design, architectural design or crowd simulation design).
The proposed method is built up of multiple components, described in the following sections. Firstly a parallel software architecture is introduced, which allows the current pipeline to use multiple, available threads in a computing node. Using the same core ideas, a distributed software architecture is introduced to divide an environment over computing nodes in a distributed network. Secondly, a lossless inter-node communication method for exact number types is proposed. The practicality of this technique is explored and compared to existing methods of communication. Thirdly multiple partitioning methods, which subdivide a 3D polygonal environment over threads and/or computing nodes, are proposed. Different partitioning methods are introduced to reach optimal load balancing. Finally, a rounding method for exact number types, which allows for lower latency inter-node communication and faster processing, is introduced.

1.2 Research rationale

As the goal of the research is a parallel geometry pipeline, the central research question this research will answer becomes:

"How can a geometry pipeline (a set of consequently applied geometry filters) be parallelized, given a polygonal soup that represents a 3d environment, to reach low execution times, optimal quality and optimal load balancing?"

To answer the primary research question, multiple sub-questions need to be answered to reach an efficiently implemented, parallelized preprocessing pipeline. Also included in the scope of this research would therefore be the sub-questions:

"How can geometry filters be parallelized and optimized to reach low execution times?"
"When should we distribute the mesh processing problem in terms of execution time?"
"How can exactness be guaranteed when communicating intermediate results between multiple processes of a geometry pipeline?"
"Can non-exact communication be used to speed up the overall computations, while maintaining the quality of the result?"
"How can the given polygonal soup be distributed in such a way that the parallel performance exceeds the sequential performance?"

1.3 Outline

The remainder of this thesis will be structured according to the following template:

Section 2 gives an overview of related work of parallel geometrical algorithms and introduces the *explicit corridor map* (ECM) method.
Section 3 introduces concepts necessary to the parallel pipeline. The concepts explored include definitions of geometry and topology, the exact number type used for geometric operations. Additionally, the software design of the sequential pipeline and its components proposed by M. Polak [Pol16] is explored.
Section 4 defines the contributions: the design of the parallel (and distributed) pipeline (4.1), inter-node communication (4.2), partitioning methods (4.3) and rounding geometry (4.4).
Section 5 describes experiments ran on real world environments and artificially constructed environments. Data produced by these experiments is used to determine the effectiveness of the parallel pipeline compared to the sequential pipeline.
Finally, conclusions will be drawn in Section 6 by using data gathered from testing to select the most optimal version of the parallel pipeline. The pipeline's shortcomings and advantages are explored and potential future work is highlighted.

2 Related work

Performing geometrical algorithms in parallel, or the parallelization of geometrical algorithms, is a subject of wide scale. A modest volume of work has been done in this area, but performing geometrical algorithms in distributed systems or using cloud computing is a rather unexplored area. The focus of the discussion on related work will be put on techniques which can enhance the preprocessing pipeline. Concepts introduced by these works are evaluated in terms of parallelization, load distribution, algorithms, memory optimization or used data-structures.

2.1 Explicit Corridor Map

The environment in which agents move is typically three-dimensional, but agents are constrained to walkable surfaces. These surfaces form the *walkable environment* (WE). It is often useful to subdivide the WE into planar layers. We refer to such a subdivision as a *multi-layered environment* (MLE). The focus in this work is put on a pre-processing pipeline which extracts the walkable environment from a polygonal 3D environment. The obtained walkable environment can then be processed into an *explicit corridor map* (ECM) [Tol+18]. The ECM is a navigation mesh based on the medial axis. [Tol+18] Formal definitions of all these concepts follow in section 3.1.

The previously implemented pipeline [Tol+16] [Pol16] is a sequential set of geometrical operations or filter steps. The pipeline, after successfully completing, ensures the polygonal 3D environment adheres to a set of criteria. After running a pipeline with a set sequence the polygonal 3D environment forms the walkable environment. The set sequence which produces a *walkable environment* is the minimal pipeline. An introduction to the effects of all filter steps and the minimal pipeline is given in section 3.2.

To perform geometrical operations used by filter steps, the sequential pipeline uses the Computational Geometry Algorithms Library [The17], CGAL. Efficient, easily adaptable and robust implementations of core geometrical algorithms are provided by this library. Roughly speaking, the pipeline processes a polygonal 3D environment to an output environment representing a *walkable environment*. This means that the output 3D environment contains no steep polygons, degenerate polygons or overlapping obstacles after the pipeline has completed. [Pol16] In other words, the output environment is an environment fully traversable by agents. The current implementation of the preprocessing pipeline does not make use of the full capabilities of high-end, multi-core computers, because is a sequential set of single-threaded operations.

2.2 General parallel geometric algorithms

Work on theoretical parallel algorithms began decades ago. Even parallel geometric algorithms have received attention in previous literature. In earlier work(s), Chow [Cho81] addressed problems such as intersections of rectangles, convex hulls and Voronoi diagrams. Since then, researchers have studied theoretical parallel solutions in the *parallel random access machine* (PRAM) model, many of which are impractical or inefficient in practice.

The research done by Batista et al. implements a thread-safe compact container in CGAL [Bat+10], which is used to store graphs such as Delaunay triangulations and allows for concurrent addition and removal of elements, with minimal constant-time overhead. [reference needed]

They extended the existing non-thread-safe compact container class to have one independent free list per thread, which completely got rid of the need for synchronization in the removal operation. In addition, while considering the addition operation, if the thread's free list is not empty, then a new element can be taken from its head without need for synchronization either. [Bat+08]

Using this thread-safe compact container they parallelize the geometric algorithms: spatial sorting, *kd*-tree construction, d-dimensional box intersection and bulk insertion into Delaunay triangulations. Their work generates a high-to-excellent speed-up with an increasing amount of cores for all their parallelized geometric algorithms and by using their thread-safe compact container, indicating the importance of a high-quality thread-safe data structure.

Though a good idea, the proposed parallel pipeline is parallelized not at the geometric algorithm level but at a software architecture level. The idea behind this is to provide a general framework for parallelizing every possible type of processing pipeline made up of an arbitrary sequence of filters.

2.3 Parallel mesh simplification

Mesh simplification might not one-to-one relate to the subject at hand, but intelligent partitioning techniques for parallel algorithms are useful for load balancing. The algorithmic structures also have a substantial amount of overlap with the functionality of the preprocessing pipeline.

Other than the importance of a thread-safe data-structure, research by [HG01], [OKK15], [Lin00] and [CL08] all indicate that partitioning is important to load balance. But I/O operations to a hard-disk for intermediary results is highlighted to be the largest time sink in their methods. Cozzi et al. attempt to avoid the problem by not writing data in the same thread as their algorith, but disk IO still dominates their HLOD creation algorithm. [CL08]

Cozzi et al. show similar patterns in their research of parallelized hierarchical level-of-detail generation in a top-down fashion using vertex clustering, which exploits the two orthogonal degrees of parallelism. [CL08] The polygonal simplification process executes on separate nodes in parallel and, additionally, sub-trees are processed and spatially subdivided in parallel. Cozzi et al's work shows also that having each node write to disk generates too much disk IO. A production quality implementation should batch up write commands to drastically improve performance.[CL08]

An alternative strategy for this problem in larger environments is indicated by Cozzi et al is the possibility to combine their research with the research by Lindstrom et al [Lin00] in which an algorithm for out-of-core simplification of large polygonal datasets which do not fit in the main memory is presented. In-core simplification techniques are difficult to adapt to perform out-of-core simplification, because the majority of them are based on performing simple local operations that rely on having direct access to the connectivity of the mesh. [Lin00]

In this research the dominating disk IO described by Cozzi et al. [CL08] are taken as a research topic in Section 5.3.1. In the specific case of the preprocessing pipeline, *CGAL*'s *CORE::Expr* exact number type is used. This number type is of a non-traditional format, which cannot be serialized by standard methods. Multiple types of disk/network IO for this format are explored for the parallel version of the pipeline in this section.

3 Preliminaries

3.1 Geometry and topology

The three-dimensional environments described in this paper are represented by a set of polygonal meshes (or polygonal strips). A three-dimensional environment E contains a set of polygons P stored in an R-tree called T. The r-tree T is used to perform spatial queries on the environment E. Some operations in the preprocessing pipeline mark polygons for deletion, but are not completely removed. Instead these polygons are stored in a separate r-tree $T_{rejected}$. [Pol16] This set of polygons can describe a connected surface, or multiple surfaces [Pol16]. Similar to the paper by M. Polak [Pol16] we work in the three-dimensional Euclidean space $\mathbb{R}^3$, with the XY-plane as the ground plane. Up is described as the positive Z direction. [Pol16]. To preface all descriptions regarding geometry and topology, all concepts relating to these topics will be defined in this section.

A polygon p is defined by a list of n vertices $p_v = \langle v_0, ..., v_{n-1} \rangle$, which are ordered in counter-clockwise order. Internally a polygon p also has a unique identification number $p_{id} \in \mathbb{N}$. For this paper we assume that polygons always have $n = 3$ vertices. This means that all polygons define triangles and all polygonal meshes are represented as triangle meshes.

Sequential vertices v_m and $v_m + 1$ in a polygon, where $m \neq n$, are connected by an edge (v_m, v_{m+1}), with the exception that vertex v_n is connected to the initial vertex v_0. A vertex v is defined as a point in space $v = \{v_x, v_y, v_z\} \in \mathbb{R}^3$. A vertex also owns an identification number $v_{id} \in \mathbb{N}$. To represent connectivity a list of pointers to polygons $v_p = \langle v_{p,1}, ..., v_{p,n} \rangle$ is owned by a vertex. For correct connectivity it is assumed that all polygons referenced to by a vertex v have vertex v in their list of vertices p_v.

Definition 3.1. Correct connectivity A vertex v has correct connectivity when $\forall x \in v_p$, such that $\exists y \in x_v$ where $(y = v)$ holds.

A polygonal mesh (or polygonal strip) is formed by connecting polygons to each other. We call these connections the mesh topology. We consider two polygons to be connected if they share an edge and have correct connectivity. For an edge to be shared, its two vertices must appear consecutively in both polygons' vertex lists. We call connected polygons (topological) neighbours. We define the topological region of P as the set of polygons that P is connected to, directly or indirectly, through this neighbour relation. In this research duplicated geometry is not allowed. Duplicated geometry means that a vertex, edge, polygon or set of polygons is present multiple times. [Pol16]

The notation of vertices in the form $v_1, v_{...}, v_n$ are assumed to be an ordered list of vertices. The notation form v_a, v_b, v_c is assumed to be non-ordered list of vertices. Polygon strips (or partial polygon strips) are assumed to be planar, when all vertices in the polygon strip lie on the same plane, the supporting plane, described by any of the normals of polygons in the polygon strip. For a polygon with $n = 3$, the normal is defined as $\vec{n}/|\vec{n}|$, where $\vec{n} = (v_2 - v_1) \times (v_3 - v_1)$ and v_1, v_2 and v_3 are not collinear.

3.2 Sequential pipeline

The extending of the pipeline as described by Polak et al. [Pol16] into a parallel computing-ready solution forms the subject of the performed research. The concepts, definitions and restrictions necessary to properly define the contributions are explained in this section.

The sequential filtering pipeline in the research by Polak et al. in [Pol16] is defined as a filtering pipeline, Each filter is a relatively simple processing step that solves a specific subproblem for determining the walkable area of an environment [Pol16]. These filters or filter steps are fully customizable in order, can be run multiple times within a pipeline and require individual parametrization by the end user. Each specific type of filter has a set of preconditions requiring the environment to fulfil and the effects it has on the environment. The preconditions and effects of each relevant filter are defined below.

Filter 1. ***Degeneracy filter*** *The degeneracy filter removes all triangles in T and $T_{rejected}$ which are degenerate. Triangles can be removed safely from the environment, since degenerate triangles (refer to definition 4.11) have an area of 0 and do not contribute to the environment.*

- ***Parameters****: -*
- ***Precondition****: -*
- ***Effect****: No triangle in T or $T_{rejected}$ is degenerate.*

Filter 2. ***Duplication filter*** *The duplication filter finds all vertices in polygons in T within a distance dup_{max} of each other and collapses these vertices to a single vertex, so no duplicate vertices remain. The duplication filter can create degenerate triangles using this method. When, for example, the distance between vertices within a single polygon is smaller than dup_{max}, a polygon is created with three references to one vertex.*

- ***Parameters****: $dup_{max} \in \mathbb{R}$*
- ***Precondition****: -*
- ***Effect****: There are no separate vertices closer than dup_{max} to each other.*

Filter 3. ***Slope filter*** *The slope filter finds all polygons in T with a slope lower than α_{max}, removes them from the environment T and adds them to $T_{rejected}$. As it does not modify topology, no degenerate polygons are created.*

- ***Parameters****: $0 < \alpha_{max} < 90\,\text{deg}, \alpha_{max} \in \mathbb{R}$*
- ***Precondition****: -*
- ***Effect****: Every triangle in T has a slope of at most α_{max}.*

Filter 4. ***Vertical clearance filter***

- ***Parameters****: $v_{min} \in \mathbb{R}$*
- ***Precondition****: -*
- ***Effect****: No interior point of any triangle in T has any other interior point of a triangle in T or $T_{rejected}$ above it within distance v_{min}.*

Filter 5. ***Area filter*** *The area filter finds all topological regions in T (polygon strips) and removes them if the sum of the area of their polygons is lower or equal to a_{min}.*

- ***Parameters****: $a_{min} \in \mathbb{R}$*
- ***Precondition****: -*
- ***Effect****: Every topological region in T has a surface area of at least a_{min}.*

Filter 6. ***Simplification filter***

- ***Parameters****: -*
- ***Precondition****: -*
- ***Effect****: The mesh subdivision is changed, while its shape remains the same.*

Filter 7. ***Layer subdivision filter***

- ***Parameters****: -*
- ***Precondition****: $\forall P \in T : area(P) > 0 \wedge slope(P) < 90°$*
- ***Effect****: The polygons in T are subdivided into layers, such that they form a multi-layered environment.*

To obtain the *walkable environment* (WE) from an input environment, the minimal pipeline must be ran.

The minimal pipeline consists of, in order: a *slope filter* and a *vertical clearance filter*, where α_{max} is the steepest angle slope an agent can traverse and v_{min} is the maximal height of an agent. Typically, however, environments include errors and more filters need to be employed, such as the *degeneracy filter*, *duplication filter* and *area filter* to clean up human errors in the environment in the form of extraneous vertices, small polygon strips and degenerate polygons.

To obtain the *multi layered environment* (MLE) the minimal pipeline must be executed with an additional filter, the *layer subdivision filter*.

3.3 Exact number type

The number type used by the pipeline is *CGAL*'s *CORE::Expr* acyclical expression graph number type which provides exact computation over the subset of real numbers that contains integers [Kar+99]. The expression graph is closed by the operations $(+, -, /, *,$ square root$)$. Operations and comparisons between objects of the *CORE::Expr* type are guaranteed to be exact. This number type represents expressions as a *directed acyclical graph* (DAG) allowing it to avoid typical issues encountered in more primitive number types. Where, for example, certain values can not be represented or inaccuracies occur at extreme low and high ends. The expression graph number type, however, requires far more memory storage compared to storing raw values. Even the simplest number in this expression graph format is more expensive to store in memory compared to traditional number types.

Four numerical accuracy levels are described by the CORE-library, namely:

- **Level 1: Machine accuracy**. Identified with the IEEE-standard. Problems with reducing level 3 numerical accuracy to level 1 numerical accuracy are described in chapter 4.4.
- **Level 2: Arbitrary accuracy**. Any user-desired accuracy.
- **Level 3: Guaranteed accuracy**. Guarantees the first n significant bits of a computed quantity are correct. [Kar+99]
- **Level 4: Mixed accuracy**. Accuracy levels 1 through 3 are mixed and local to individual variables for finer accuracy control. [Kar+99]

In this thesis we primarily discuss number types of numerical accuracy level 1 and 3. All further references to the *CORE::Expr* number type are assumed to have accuracy level 3.

Level 3 accuracy is praised to be the "key innovation of *CORE*" [Kar+99]. In level 2 accuracy it is stated that specifying n bits of accuracy doesn't mean that all n bits are significant. Level 2 accuracy can lose accuracy from computational singularities at high bits of accuracy. Level 3 accuracy differs here insofar that it automatically detects these singularities. Another qualitative difference is that level 3 achieves error-free comparisons of any two reals. [Kar+99]

Traditional number types are categorized in these numerical accuracy levels as follows:

- **Level 1 number types**. Primitive number types *int*, *long*, *float* and *double*.
- **Level 2 number types**. *bigInt*, *bigRat*, *bigFloat* and the superclass *Core::Real*.
- **Level 3 number types**. *Core::Expr*

A core feature of the *Core*-library is the automatic promotion and demotion of a number type to a number type with higher accuracy. Promotion or demotion of a number type occurs during assignment or when setting numerical accuracy levels.

At accuracy level 3, primitive number types are promoted to the level 3 *Core::Expr* number type. Inside the graph structure, machine accuracy is kept for as long as possible, without promoting, for speed. The graph structure is populated by nodes which either represent mathematical operations or constant values of different number types. Nodes representing a constant value can be populated by any level of number type, even other *Core::Expr* graph structures. Concretely, the following mathematical operations exist in the implementation used by the *CGAL* library: unary operator nodes, binary operator nodes and constant nodes. Unary operators include the unary minus operation and the square root operation and have a dependence on one child node. Binary operators include the addition, subtraction, division and multiplication operations and have a dependence on two child nodes. Constant nodes (leaf nodes) have a level 1, 2 or 3 number type. In Figure 1, dependencies in an expression directed acyclical graph are highlighted.

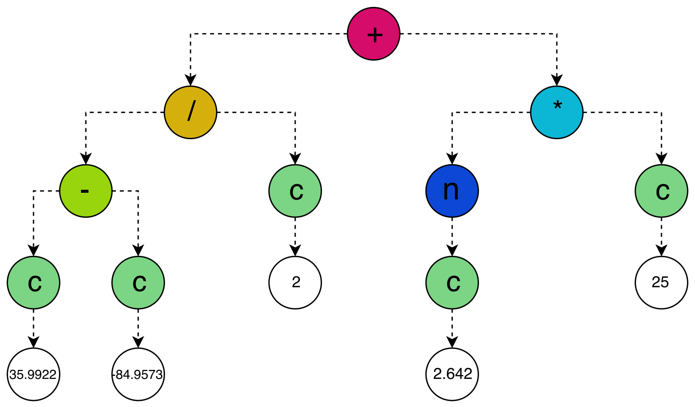


Figure 1: Expression graph of the expression $(35.9922 - (-84.9573))/2 + negate(2.642) * 25$.

The reason for constructing such expressions is that we need to propagate the precision of a parent node into precisions for their children nodes. The *Core::Expr* number type ensures it has a requested precision by downward propagation of precision and upward propagation of errors. This may be iterated until the error is less than the requested precision. [Kar+99]

Naturally, there is a cost with guaranteed exactness for complex number types such as the *Core::Expr* number in the form of overhead. Three sources of overhead are present in the *Core::Expr* number type, namely: function-call overhead, memory management overhead and operation overhead.

Function-call overhead appears due to the object-oriented programming style used in the *Core*-library which increases the relative contribution of function call costs to overall execution times [Kar+99] and prevents compiler optimization in certain places.

Memory-management overhead occurs because the expression trees used by the *Core*-library are dynamic pointer-based structures consisting of simple, small objects. These pointer-based structures incur overheads during allocation and deallocation. Additionally it is not guaranteed that nodes owned by a directed acyclical graph have a high memory locality, due to the non-linear data structure utilized by the number type. Nodes representing operations use the pointer structure to refer to their one to two children, which can be allocated to be anywhere in the memory. Memory access patterns can therefore be sub-optimal during the evaluation of these expression trees. Practically this means that nodes are **not** guaranteed to be near each other in the working memory (nodes can exist in a potentially random location in the working memory).

Even worse, since every node can be in a non-linear place it becomes a non-trivial task to store the acyclical expression graph from memory in order into a file or a memory stream for communication. In section 5.3.1 this is explored in depth.

Operation overhead occurs because level 3 evaluation requires several iterations of the downward propagation of precision and upward propagation of errors. [Kar+99] To preserve encapsulation, both these steps are performed at the granularity of individual operation nodes in the expression tree (e.g., a subtraction operation). [Kar+99] Concretely, this occurs, when, for example, calculating the value of an expression tree into a machine-accurate number type. A relatively large amount of processing time is required to perform this (seemingly) easy task and is one of the main causes of slowdowns when communication is performed between multiple pipelines.

However, even after all these downsides, the property of guaranteed exactness is a big enough upside that the *Core::Expr* number type is used as the core of the pipeline, as it circumvents precision and stability issues encountered in traditional three-dimensional floating point arithmetic.

4 Contributions

In this research the existing single-threaded preprocessing pipeline of the Explicit Corridor Map [Ger10] method is extended with the goal of contributing a parallelized (and optimized) framework which reaches an execution time at which the end-user will be able to tweak the navigation mesh in an iterative process (e.g. game world design, architectural design or crowd simulation design).

The components of the proposed method will allow for (theoretical) lossless inter-node communication of exact number types, a rounding method for exact number types which allows for lower latency inter-node communication, a parallel software architecture which allows for scaling up the current pipeline to use the maximum number of threads in a node and the maximum number of nodes in a distributed network and partitioning methods which ensure optimal load balancing per partition (be it per-node or per-thread).

4.1 Parallel pipeline

In this paper we focus on adapting the software design of the geometry-processing pipeline, as proposed by Geraerts/Vermeulen/Hillebrand/Polak, to allow for use of current advancements in computer hardware, namely the industry standard of having multiple processor cores and advances in distributed computing. A scalable, parallel-ran geometry-processing pipeline is proposed of which the software design allows use as a multi-threaded solution within a single machine, while also serving as a solution for running a geometry-processing pipeline on a distributed network.

The proposed parallel pipeline consists of four main parts:

1. **Pipeline blocks**. Pipeline blocks are defined as a sequence of filters (and their parameters) describing a sequential pipeline or as a sequence of filters (and their parameters) combined with a partitioning method describing a parallel pipeline.
2. **Partitioning method**. A partitioning method subdivides the input environment into n sub-environments.
3. **Thread pool** (node-level) and **scheduling algorithm** (network-level). On a local level, the thread pool ensures partitions owned by the local machine are optimally load balanced over all threads at runtime. On a network level, the scheduling algorithm ensures each of the m nodes have a subset of the n partitions created by the defined partitioning method assigned to them such that the partitions are optimally load balanced over the network and all nodes perform equal work.
4. **Communication method**. On a network level, a communication method is used in order to communicate resulting environments between the master- and worker nodes after a node is done processing the pipeline.

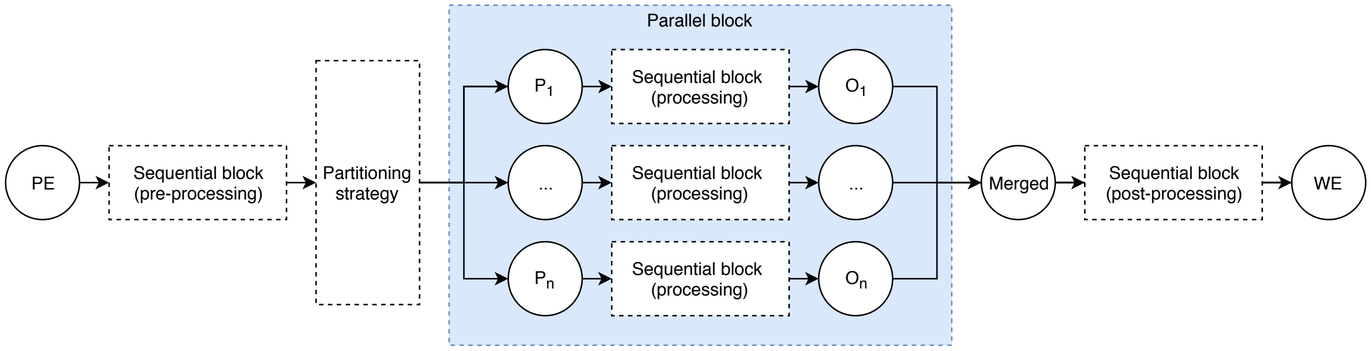

Parallel block
PE
Sequential block (pre-processing)
Partitioning strategy
P1
...
Pn
Sequential block (processing)
Sequential block (processing)
Sequential block (processing)
O1
...
On
Merged
Sequential block (post-processing)
WE

Figure 2: Architecture of our proposed **parallel pipeline**. The partitioning strategy subdivides the *polygonal environment* (PE) into n partitions $(P_0, \cdots, P_n)$. Internally the thread pool divides the n partitions over the available threads in the local machine and applies the processing pipeline block on the partition. After processing the n partitions in parallel the *intermediary environments* $(O_1, \cdots, O_m)$ are combined into one *merged environment*. Finally the post-processing sequential pipeline block is applied to obtain the *walkable environment* (WE).

By running a parallel pipeline on an input environment a sequential list of pipeline blocks, representing sequences of filters, is applied on the environment in order to extract the walkable area and to generate a multi-layered environment. A parallel pipeline is defined as a sequence of these *pipeline blocks*. Pipeline blocks can either be defined as a sequence of filters (and their parameters) describing a sequential block or as a sequence of filters (and their parameters) combined with a partitioning method describing a parallel block. The difference between a sequential block and a parallel block is that a sequential block requires the complete environment to apply their filters, while a parallel block runs on a partial environment (a partition).

Primary focus is put on the most common configuration of these blocks, the common use case. The common configuration is a sequential block followed by a parallel block followed by a sequential block. The first (sequential) block, the

pre-processing phase, contains pre-processing filter steps which clean the environment of geometrical errors which must not be present before a partitioning method partitions the environment. The pre-processing phase contains at least a duplication filter and a degeneracy filter. The second (parallel) block, the **processing** phase, contains crucial filters which process the environment such that a multi-layered environment can be generated in the final step. This block contains at least the steepness filter and the height clearance filter. The final (sequential) block, the **post-processing** phase, contains filters which combine the partitions back into one environment and simplify the environment before extracting the multi-layered environment. This block contains at the very least a degeneracy filter, a duplication filter and the layer subdivision filter.

The sequential blocks are pipelines which are only ran on the master node. Distributed blocks distribute the output of the previous block in the sequence into multiple sub-tasks by using their task distribution method. These sub-tasks are assigned over the network to achieve optimal load balancing using their task scheduling method. Once nodes on the network are assigned tasks by the task scheduling method the sequential block is ran on the nodes locally and the result is communicated back to the master node. The master node then combines the intermediary results into the final result. Blocks are ran in order, which means that when a distributed block is executed and a sequential block is next in the parallel pipeline sequence, the intermediary result of all worker nodes is required in order to generate the input of the following sequential pipeline. This behaviour can result in stalling when load balancing is sub-optimal or communication by nodes is congested. Waiting for results is absolutely necessary, though, to ensure validity of later filter steps.

Three elements contribute the most to lowering the parallel execution time, namely how fast are tasks split into subtasks with the task distribution method (further referred to as a **partitioning method**), how optimal are these subtasks divided over nodes and processors by the task scheduling method (further referred to as a **scheduling method**) and how fast are tasks and results communicated through the network by the inter-process communication method. Note that possible optimization of the filter operations to run optimally in a parallel setting lies outside of the scope of this paper, even though it is a valid strategy to reduce the parallel execution time.

Broadly speaking, the partitioning method divides the main task into an arbitrary number of sub-tasks defined by the parameter of the partitioning method. The scheduling method ensures optimal load balancing of these sub-tasks over all nodes on the network and locally inside nodes on a processor-level. The inter-process communication method ensures rapid delivery of tasks to worker nodes and intermediary output from the worker nodes to the master node.

4.1.1 Distributed pipeline

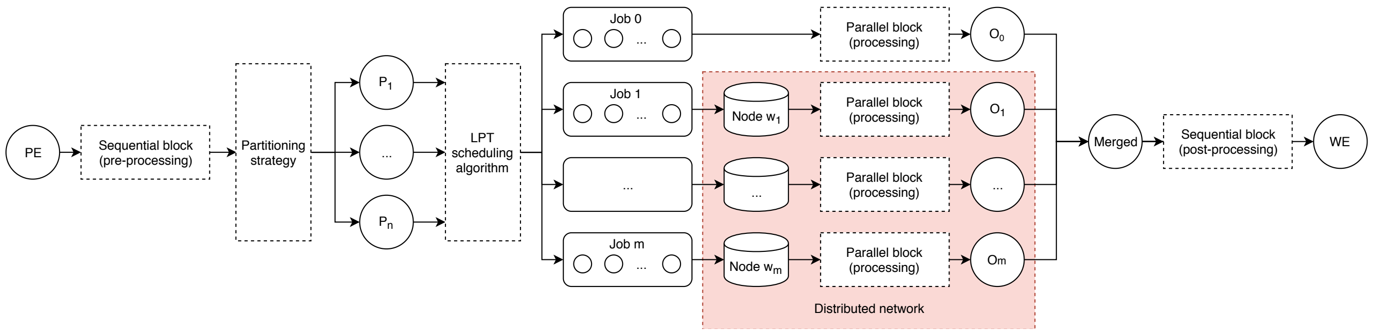


Figure 3: Architecture of a **distributed pipeline** with n partitions distributed over m nodes from the perspective of the master node ($rank = 0$). The partitioning strategy subdivides the *polygonal environment* (PE) into n partitions. The LPT scheduling algorithm finds m tuples of partitions and assigns them to the $m-1$ nodes in the distributed network. Tuple 0 is always assigned to the master node ($rank = 0$). After processing the $m-1$ nodes in the network send their *intermediary environments* $(O_1, \cdots, O_m)$ to the master node ($rank = 0$). The master node combines all intermediary environments into one environment and applies the post-processing sequential pipeline block to get the *walkable environment* (WE). For pseudo-code of this algorithm refer to Algorithm 4.

In a distributed configuration the parallel geometry processing pipeline runs on n_{nodes} nodes, henceforth referred to as the distributed pipeline.

When $m = 1$, the distributed pipeline is ran in parallel on a singular machine, not on a distributed network. Most of this paper refers to a parallel pipeline ran on a singular machine, unless explicitly is referred to running a parallel pipeline over a distributed network. In the case where a parallel pipeline is ran on a singular machine, no communication between other pipelines is necessary, since all data is available locally. That is not to say no communication is necessary, however, since *inter-thread* communication is still necessary when combining intermediary results.

Inter-thread communication is not unique to a network with 1 node, because it happens in all configurations. Worker nodes in a $m > 1$ environment also use threads within the node to make use of all computational resources. Whenever a worker node combines its result to send it to the master node, *inter-thread* communication happens to combine the intermediary results from their threads. Afterwards this combined result is communicated to the master node using *inter-node* communication.

When $m > 1$, the distributed pipeline is ran on a distributed network of machines. A distributed pipeline assumed that one node (of rank 0) is the master node, which distributes tasks to worker nodes (of rank $\neq 0$). The master node also initializes the geometry-processing pipeline on the worker nodes (pipeline initialization). Once the master node has distributed a task to the worker nodes, the worker nodes, in turn, distribute this task further to onto their local resources (processors, threads) to make optimal use of all available computing power. By distributing tasks in this hierarchical fashion an optimal load balance can be attained not only on the network-level, but also on the local node-level. Worker nodes whose tasks have completed communicate their intermediary results back to the master node. Communication in this environment is limited to sending/receiving pipeline initialization data, sending/receiving tasks and sending/receiving intermediary results. **Note:** since the master node also distributes tasks to itself (to circumvent idling and to reduce extraneous communication of intermediary results) it is also considered to be a worker node.

The primary goal of the distributed pipeline is to attain a parallel execution time which is lower than the sequential execution time and to attain a high parallel speed-up. For $m = 1$ it is trivial to define the parallel execution time and the parallel speed-up, because we are able to disregard all costs regarding inter-node communication, since only one node populates the network. We define $t(n, m)$ is the sequential execution time, while $t(n, m)$ to be the parallel execution time, where n is the size of the input and m is the number of nodes or processors. The speed-up in the case of $m = 1$ can then be defined as the ratio between the sequential execution time and the parallel execution time. The parallel speed-up formula for the distributed pipeline then becomes $S(m) = \frac{t(n,1)}{t(n,m)}$, assuming the number of threads is **constant over all nodes**.

The execution time in the distributed pipeline case, however, is composed of not only the time spent running the filter operations, but also of overhead. Overhead is referred to, in the scope of the parallel pipeline, as a sum of multiple factors, namely the time it takes to distribute tasks over the network ($t_{partitioning}$) and the time it takes to communicate tasks and intermediary results ($t_{communication}$). The overhead described here only occurs when $m > 1$.

4.2 Inter-node communication

To perform exact calculations the currently implemented pipeline uses, as described before, an expression graph number type. This number type guarantees exactness by storing all performed operations during the various geometric operations used in processing polygonal environments in an expression tree.

To communicate intermediary results between nodes in the distributed variant of the pipeline an inter-node communication method with low latency is required. Stalling will occur in even non-complex environments if a suboptimal method is used, which will impact the parallel performance significantly compared to the sequential performance. In smaller environments this can lead to computing time being dominated by communication between processes rather than computing the solution to geometric filter steps.

Three methods are proposed and examined with a focus on writing/reading speed, memory usage and preserved exactness:

- Calculate the result of the expression graph number type and store it in an ASCII format,
- Calculate the result of the expression graph number type and store it in a binary format,
- Store the full directed acyclical expression graph to a binary file-format,

The primary goal of these methods is to approach a writing and reading speed which is constrained by the medium's writing and reading speed, rather than extraneous work introduced by the method. Concretely this would mean the method's writing and reading speed would approach either disk writing speed (in a local environment) or the networking speed (in a distributed environment).

The secondary goal would be to have the lowest memory imprint possible to reduce the effect of inter-process communication on the pipeline's performance even further.

4.2.1 Approximated ASCII-format

The existing method of inter-process communication (and file reading in general) uses the Wavefront OBJ standard of describing a three-dimensional mesh. The format represents the three-dimensional geometry by describing the positions of each vertex and the polygons as a list of vertices. Due to vertices being stored in a counter-clockwise ordering, it is not necessary to declare the direction of the polygon's normal.

This format demotes *CORE::Expr* graphs to machine accuracy (into a floating point number type) and then serializes them into a target file by formatting the floating point into string format. This method loses exactness (by demotion) and serializing into string loses precision unless stored with maximum precision of 17 digits, assuming the double floating point number type specified by the used *IEEE 754* standard.

4.2.2 Approximated binary-format

A proposed method of inter-process communication computes and stores the demoted result of the coordinates' expression graphs in binary format. The binary format still stores a number with machine accuracy, but exactly as it appears in the working memory. This bypasses the need to serialize and deserialize the result from and to a text representation. Cutting out the serialization and deserialization process increases the performance significantly, as explored in the benchmarking chapter. In contrast to *ASCII* formats, binary formats have a significantly smaller memory (disk) footprint and can be streamed to disk at extremely high speeds, making them ideal for high-channel-count and real-time applications. [Ins13]

4.2.3 Binary tree-format

In contrast to the other proposed solutions, storing the full expression graph using a binary format is wanted when requiring exactness of the intermediary results. The acyclical expression graph is, in this method, naively walked through and its constant values and mathematical operations stored as an expression tree. The memory structure is effectively replicated. By reading this representation it is possible to guarantee exactness between processes. To reduce the amount of memory it takes to store an expression graph to a minimum, the nodes in the graph are stored using one (1) byte of node type information and a predefined number of bytes used by the specific node's type.

Due to the lack of a need to compute the resulting answer, unlike the approximated methods, no computational cost is attached to writing the expression tree to a file outside of the memory access and writing processes.

4.3 Partitioning methods

To distribute the environment (the full workload) over the available threads optimally it is necessary to split it into (roughly) equally sized sub-environments so that the computational load is evenly balanced. Partitioning is the process by which the set of polygons that comprise the environment is split into sub-environments. Sub-environments are then assigned to a thread belonging to the processor of a parallel machine.

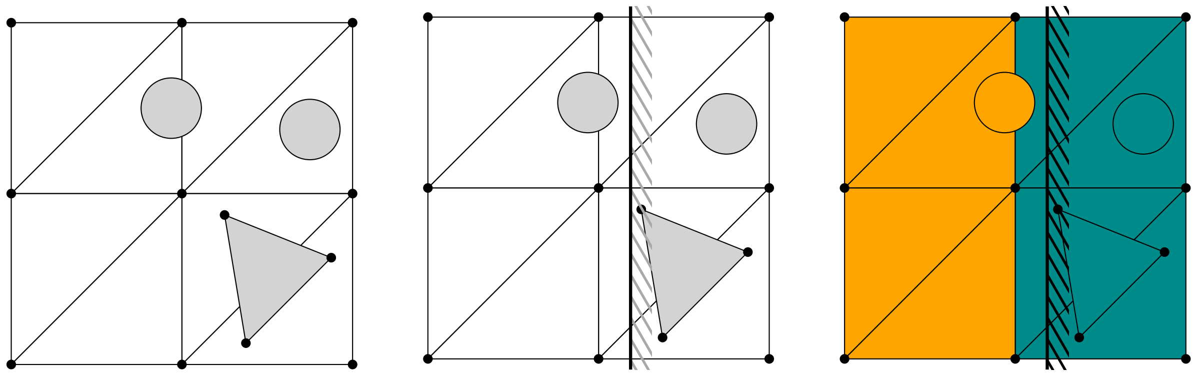
A partitioning method divides an input environment and divides it into set of n (sub-)environments by evaluating the environment in a manner unique to the partitioning method. A grid-based partitioning method, for example, will evaluate the environment based on the environment's bounding box and will divide the full environment into cells.

A partitioning method is a deterministic operation performed on an input environment which splits the input environment into an output set of n environments (cells) based on a set of parameters.

Concretely, two partitioning methods are explored, namely a grid-based partitioning method and a recursive partitioning method. Both methods first evaluate the environment and generate boundaries which describe sub-environments and then split the environment into a set of sub-environments using these generated boundaries. A cell is a convex space described by a set of axis-aligned hyperplanes. Since the environment needs to be split into multiple sub-environments in the **xy**-plane to preserve validity of the constraints imposed by the height clearance filter, hyperplanes are limited to be be aligned to the **xz**-plane and the **yz**-plane. Splitting in the **z**-axis is impossible without breaking these constraints, so only partitioning methods which split the environment in the **xy**-plane are considered worthy of exploration. The methods by which hyperplanes are placed differ per method, but the method of splitting is shared over both partitioning strategies.

Traditional mesh partitioning is unsuited for our problem space, because of the restrictions introduced by the vertical clearance filter step. All polygons regardless of height, contained by or intersecting with the bounded volume defined by the partitioning strategy must be included in the sub-environment. This is because when ran with maximal clearance parameter the vertical clearance filter requires all these polygons.

Since environments are defined as a polygonal soup they can contain any set of polygons. In many cases, the ground plane in an environment has a significantly less coarse mesh compared to the rest of the environment. The differences in coarseness between polygonal groups (or meshes) makes it hard to fulfil the height clearance restriction mentioned earlier and restricts the partitioning which can be done on an environment. Traditional mesh partitioning takes a set of polygons from the original set of polygons as a partition where the union of all partitions equals the original set of polygons. In cases with unbalanced polygonal density this can lead to suboptimal partitions where one partition may contain a set of walkable polygons and a set of obstacle polygons and is processed correctly. On the other hand a partition may not even contain any walkable or obstacle polygons at all.


(a) Environment before partitioning

(b) Partitioning hyperplane is introduced. The dashed region represents the positive side of the plane.

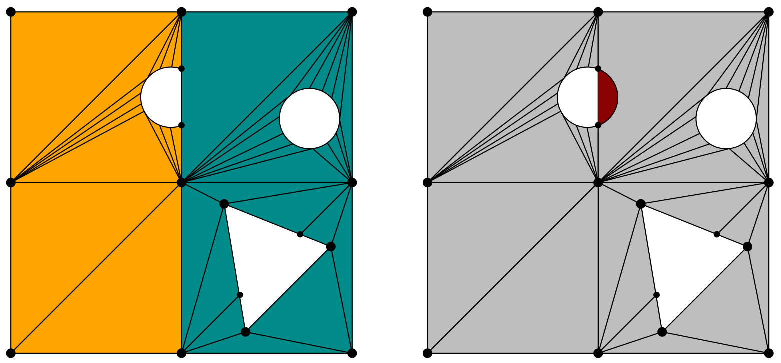
(c) Partitioning is performed where polygons intersecting or on the positive side of the plane form partition P_a, while polygons on the negative side form P_b.

Figure 4: Environment as seen from a topdown-view. Gray polygons represent obstacle polygons, while white polygons represent walkable polygons.

In figure 8 this problem with **traditional** mesh partitioning is displayed. The partition P_a does not contain the obstacle of P_b which partially lies above its ground plane, while P_b does not contain a walkable area below its singular obstacle polygon group. Performing the vertical clearance filter operation with a maximal height clearance parameter on partition P_a and P_b and performing a union on the processed set will result in a violation of the correctness of the vertical clearance filter. This is because the circular obstacle not present in P_a does lie above the ground plane in P_a, but is assigned to partition P_b. In figure 5 we see the area marked in red where this violation occurs.

Not only are the effects of vertical clearance filter violated, correct connectivity between partitions can also not be guaranteed, since the vertices created by the vertical clearance filter that are present on the partition border P_b are not present in the partition P_a because the obstacle polygon group in P_b does not appear in P_a.

For these reasons (or restrictions) and to produce well balanced partitions, the environment is pre-processed by cutting its polygons with a set of axis-aligned planes. These partitioning planes provided by a partitioning strategy. This choice was made because the algorithmic complexity of calculating triangle-plane intersection points is relatively simple. Especially so in the subset of only intersecting **xz**- and **yz**-axis-aligned planes. Due to this low algorithmic complexity it is expected that the time cost of calculating intersection points between partitioning planes and polygons will have a low impact on the overall execution time of a parallel pipeline.


(a) Partitioned environments P_a and P_b after applying vertical clearance operation. Obstacles are not pictured.

(b) Union of P_a and P_b where the area of triangles is marked in red where there exists a point above it within distance v_{min}.

Figure 5: Environment after applying vertical clearance filter with maximal clearance and union operation.

In broad lines, cutting an environment using partitioning planes is done in the following way:

Algorithm 1 Cutting environment using partitioning plane

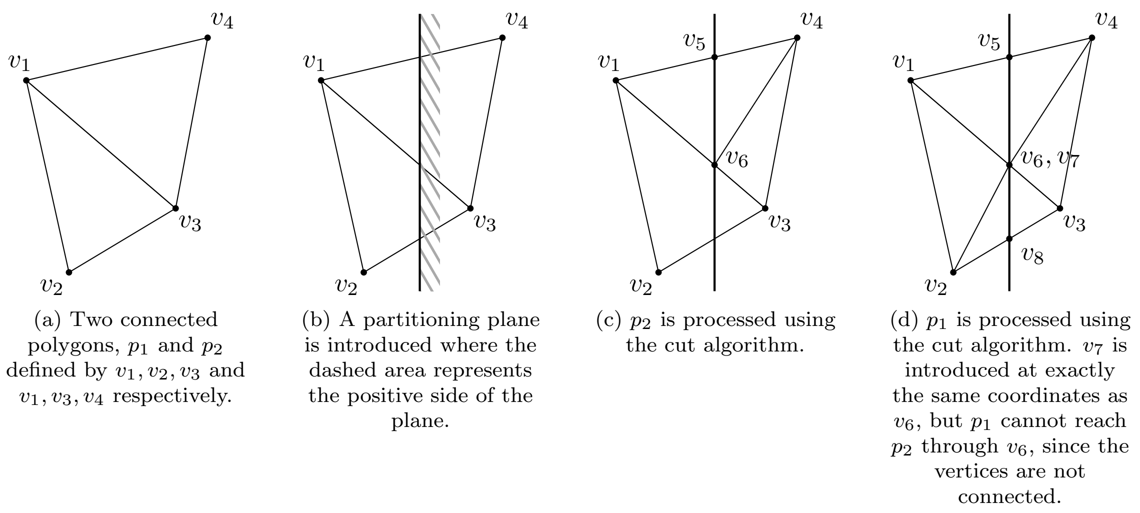
```
1: $polygons \leftarrow$ all polygons in P and $\mathrm{P}_{degenerate}$
2: for each polygon $p_{current}$ in $polygons$ do
3:     Calculate intersection points between $p_{current}$ and $plane_{partitioning}$
4:     Re-triangulate polygon $p_{current}$ into polygon set $P_{new}$
5:     Remove polygon $p_{current}$ from polygon set $P$
6:     for each polygon $p_{new}$ in $P_{new}$ do
7:         if polygon $p_{new}$ is on boundary or positive side of $plane_{partitioning}$ then
8:             Flag $p_{new}$ to be on positive side of partitioning plane
9:         Add polygon $p_{new}$ to polygon set $P$.
```

A naïve method of intersecting a polygon with an axis-aligned plane would be to create a vertex for each intersection point and using these in a triangulation method to create a set of new polygons, such as in the cutting algorithm presented above. The issue with this naive approach, however, is that several edge cases which can occur resulting in degenerate polygons and/or poor connectivity to neighbouring polygons, which are highlighted below:

- In the edge case where the intersection of a polygon and the axis-aligned plane is a point, the resulting point is guaranteed to share its position with one of the polygon's existing vertices. The resulting re-triangulated polygons will result in the original polygon connected to a degenerate polygon.
- In the edge case where the intersection of the polygon and the axis-aligned plane is an edge one situation arises where degenerate polygons arise from re-triangulation. When the resulting edge shares two positions with the original polygon two degenerate polygons are created in the resulting re-triangulation.

To avoid producing degenerate polygons it becomes necessary to take these edge cases into account in the cutting algorithm in order to fulfil the preconditions of filter steps requiring that no polygons may be degenerate. An example of a filter which is built upon this precondition is the layer subdivision filter, which can only handle polygons with non-zero area. Simply walking through the vertices owned by the original polygon would suffice. Here we ensure newly created vertices which share their position with an existing vertex are not introduced. If, at any point, an intersection point shares the position of an existing vertex, the existing vertex can be used in the triangulation, rather than a newly created vertex at the intersection point. Not only does this avoid the creation of degenerate polygons, it also preserves connectivity between the newly created re-triangulated polygons and the original polygon's neighbours correctly, since it re-uses an existing vertex. The triangulation method used, however, must account for duplicate vertices for this to correctly work.

Another issue arises in terms of mesh connectivity when intersecting multiple, connected polygons with an axis-aligned plane. Since polygon adjacency queries are performed by following the pointer structure from the vertex data structure to the polygon data structure, full mesh connectivity cannot be guaranteed from vertices on the boundary of axis-aligned planes. This is, because, using the naïve method, newly introduced vertices on a shared edge between two original polygons might share their position, but do not share connectivity, since they are separately introduced as new vertices in the intersection method by Polak [Pol16]. To ensure correct connectivity, adjacent polygons must be searched for duplicate vertices with coordinates equal to the newly introduced vertices. Duplicate vertices are collapsed into the newly introduced vertex with all polygons changed to refer to the newly introduced vertex.



(a) Two connected polygons, p_1 and p_2 defined by v_1, v_2, v_3 and v_1, v_3, v_4 respectively.

(b) A partitioning plane is introduced where the dashed area represents the positive side of the plane.

(c) p_2 is processed using the cut algorithm.

(d) p_1 is processed using the cut algorithm. v_7 is introduced at exactly the same coordinates as v_6, but p_1 cannot reach p_2 through v_6, since the vertices are not connected.

Figure 6: Minimal case of mesh connectivity errors introduced by the naive cutting method.

In filter steps requiring polygon adjacency operations, this leads to unexpected results. A polygon p_x might not be reachable from a vertex in polygon p_y. The area filter and the simplification filter use polygon adjacency operations to find connected components in an environment's set of polygons in r-tree T. If these filters cannot reach what is originally connected before the cutting algorithm is ran, then the result of the area filter on the processed (cut) environment will differ from the result of the area filter on the unprocessed environment. Since one of the goals of the pipeline is to not alter the existing algorithms, it becomes imperative to fix mesh connectivity in the cutting algorithm.

The environment's polygons in T are pre-processed using the set of partitioning planes provided by an implementation of a partitioning strategy. Afterwards the environment's polygons can be divided into set of partitions without violating the vertical clearance filter's effect after combining the partitions later in the pipeline. The set of polygons belonging to a partition P is defined to be all polygons on the boundary and on the intersection of the positive side of the set of axis-aligned planes assigned to the partition. Due to the height clearance restriction and for simplicity of implementation, axis-aligned planes are aligned to the **xz-** and **yz-** axis, resulting in rectangular-shaped partitions.

4.3.1 Grid strategy

The simplest implementation of a partitioning strategy in terms of algorithmic complexity is the grid strategy. The grid strategy takes two input parameters, namely the number of grid cells in the horizontal direction and the number of grid cells in the vertical direction. The grid strategy subdivides the input environment into $n * m$ partitions with equally sized bounding rectangles.

The grid-based partitioning method uses two parameters to construct partitions, namely n_{rows} and $n_{columns}$. Using linear interpolation between the extents of the bounding box of the input environment, a two-dimensional grid of partitioning planes is created with n rows and m columns. To divide the environment into multiple partitions a bounding rectangle is created based on the size of the original environment, a cell. A cell for the grid strategy with parameters n and m has a width of $|width_{original}|/n$ and a height of $|height_{original}|/m$, where width and height are defined by the following 1-dimensional distance formulas:

$$width_{boundingrectangle} = |x_{min} - x_{max}|$$

$$height_{boundingrectangle} = |y_{min} - y_{max}|$$

To aid in subdividing the environment, using a minimal amount of memory, a set of real numbers representing the x- and y-coordinates of the axis-aligned planes are calculated, from which these planes can be constructed in a later step. We refer to these coordinates in this chapter as x_1 to x_{n-1} and y_1 to y_{m-1}.

A coordinate x_i where $i > 0$ and $i < n$ is defined to be $x_{min} + (|width_{original}|/n) * i$ and a coordinate y_j where $j > 0$ and $j < m$ is defined to be $y_{min} + (|height_{original}|/m) * j$. For the special cases of x_0, y_0 and x_n and y_m the following values are used:

$$x_0 = x_{min}, x_n = x_{max}, y_0 = y_{min}, y_m = y_{max}$$

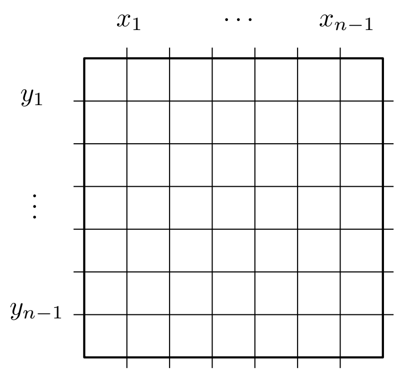


Figure 7: Coordinates $x_1, \cdots, x_{n-1}$ and $y_1, \cdots, y_{m-1}$, which form partitioning planes $pr_0, \cdots, pr_{n-1+m-1}$.

By pre-calculating the coordinates in this manner, we can ensure that only one graph of these x- and y-coordinates is generated by the *Core::Expr* exact number type and, in a later step, we can also ensure that vertices created by cutting which use these pre-calculated coordinates round to the same value. Cutting the environment now becomes trivial, since the coordinates by which the axis-aligned planes are defined are pre-calculated. By sequentially cutting the environment with planes aligned to the yz-axis first, at x-position x_1 to x_{n-1}, followed by the xz-aligned planes second, at y-position y_1 to y_{m-1}, the environment becomes fully pre-processed and can be partitioned. In total, the environment is cut n–1 times in the horizontal direction and m–1 times in the vertical direction, resulting in a total of n–1 + m–1 cuts. With these definitions, a grid cell $P_{i,j}$ can be described by a finite set of half-planes by Boolean operations of set intersection on the input environment. The set of half-planes in the grid strategy has size 4.

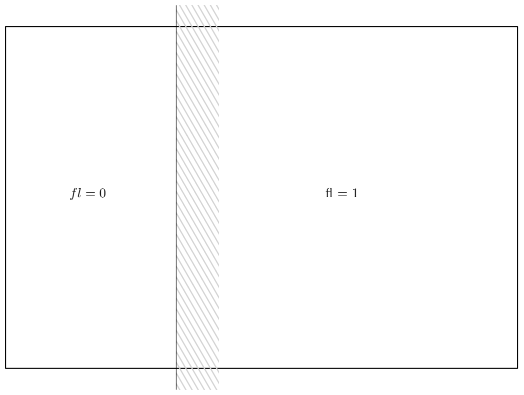
$$P_{i,j} = P \cup \{(x, y, z) : x >= x_i\} \cap \{(x, y, z) : x <= x_{i+1}\} \cap \{(x, y, z) : y >= y_j\} \cap \{(x, y, z) : y <= y_{j+1}\}$$

After cutting the polygon set P in environment $n - 1 + m - 1$ using the partitioning planes $pr_0, \cdots, pr_{n-1+m-1}$, the polygons can be partitioned. The final algorithm for partitioning an environment using a grid strategy becomes:

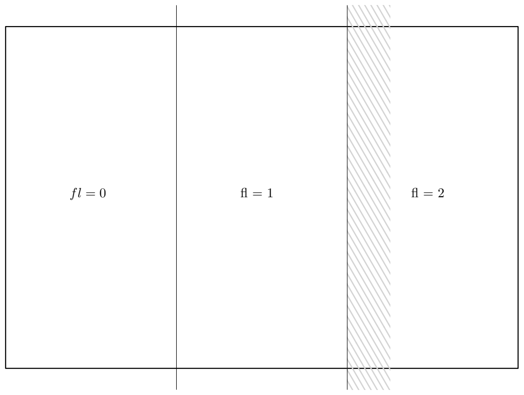
Algorithm 2 Cutting environment using grid strategy

```
1: $polygons$ ← all polygons in P and P$_{degenerate}$
2: $planes$ ← all partitioning planes defined by grid strategy
3: for each plane $pr_{current}$ in $planes$ do
4:     for each polygon $p_{current}$ in $polygons$ intersected by $pr_{current}$ do
5:         Calculate intersection points between $p_{current}$ and $plane_{partitioning}$
6:         Re-triangulate polygon $p_{current}$ into polygon set $P_{new}$
7:         Remove polygon $p_{current}$ from polygon set $P$
8:         for each polygon $p_{new}$ in $P_{new}$ do
9:             if polygon $p_{new}$ is on boundary or positive side of $plane_{partitioning}$ then
10:                Flag $p_{new}$ to be on positive side of partitioning plane
11:            Add polygon $p_{new}$ to polygon set $P$.
12: Sort polygons into partitions based on flags
```

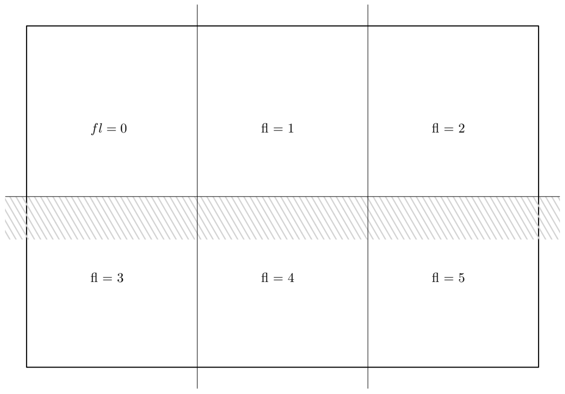
The flag fl of a polygon p is an integer number $fl \in \mathbb{Z}$. A flag fl is assumed to be zero-initialized. Flagging a polygon p to be on the positive side of a partitioning plane is done by incrementing the flag fl by a constant value. The constant value is determined by whether or not the partitioning plane pr_i is an x-axis aligned partitioning plane or a y-axis aligned partitioning plane. Since the list of partitioning planes $pr_0, \cdots, pr_{n-1+m-1}$ is sorted with y-axis aligned partitioning planes first, while y-axis aligned partitioning planes second. For x-axis aligned planes $pr_0, \cdots, pr_{n-1}$ the flag fl is increased by 1, while for y-axis aligned planes the flag fl is increased by n.



(a) Partitioning plane pr_0 increments the flag fl on polygons on its positive side by 1.



(b) Partitioning plane pr_1 increments the flag fl on polygons on its positive side by 1.



(c) Partitioning plane pr_2 increments the flag fl on polygons on its positive side by n.

Figure 8: An environment processed by a grid strategy, where $n = 3$ and $m = 2$.

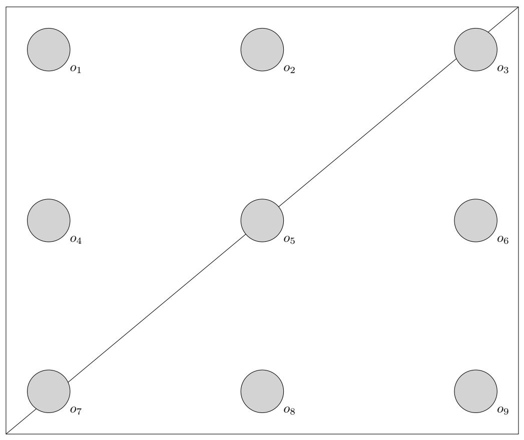
Since the cutting algorithm is applied to polygons intersecting the partitioning plane, we know there are only two types of polygons: polygons fully on the positive or negative side of the plane and polygons with $m < 3$ vertices on the boundary and $3 - m$ vertices on the positive or negative side. Polygons with at least 1 vertex on the positive side of the plane have their flags incremented, which encompasses polygons fully on the positive side and polygons partially on the positive side (polygons with vertices on the boundary and on the positive side). Polygons fully or partially on the negative side are ignored in this process.

After the cutting and flagging process, all polygons in T and $T_{rejected}$ are partitioned into the set of partitions $P_{0,0}, \ldots, P_{n-1,m-1}$. Because polygons are flagged no additional plane checks have to be made. For a partition $P_{i,j}$ polygons flagged with flag $fl = i + j * n$ are part of the partition. Collecting all polygons with a specified flag fl into a partition costs $O(n)$. Additionally, since both the cutting process and the flagging process are deterministic, it is guaranteed that $P(i, j)$ on a parallel machine $node_1$ equals $P(i, j)$ on another parallel machine $node_2$.

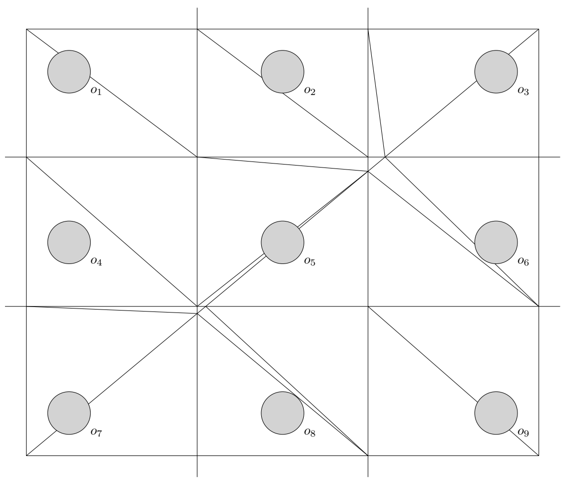
Now that the process by which a grid strategy pre-processes the environment is explored, observations can be made about load balancing. It is expected that in environments with uniform spacing between obstacles and a uniform distribution of polygons this strategy will produce a set of partitions with a good fit (geometrically speaking) and optimal load balancing in terms of the number of polygons per partition. In other environment types where polygons are not uniformly distributed this strategy will produce a poor fit and will offer poor load balancing. Examples of good and poor fits are highlighted in figure 9 and figure 14.

Since an environment is defined to be a polygonal soup, a better strategy would take into account the topology of the environment to produce better placed partitioning planes.

The performance of this on optimal environments and suboptimal environments is further explored in Section 5.

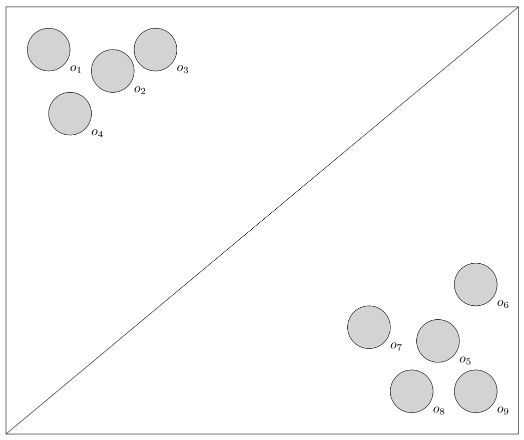


(a) Environment with uniformly distributed obstacle polygons.

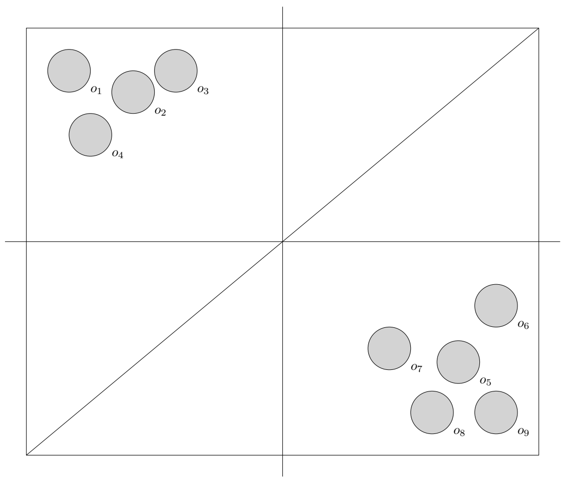


(b) After applying the grid strategy with $n = 3$ and $m = 3$, $n * m = 9$ partitions are created with each having a similar number of polygons.

Figure 9: An environment which produces well-balanced partitions using the grid strategy. It is assumed that obstacle polygons $o_1, \cdots, o_9$ have an equal number of polygons.



(a) Environment with clustered obstacle polygons.



(b) After applying the grid strategy with $n = 2$ and $m = 2$, $n * m = 4$ partitions are created. The top-left partition and the bottom-right partition contain all obstacle polygons, while the other partitions contain none.

Figure 10: An environment which produces poorly balanced partitions using the grid strategy. It is assumed that obstacle polygons $o_1, \cdots, o_9$ have an equal number of polygons.

4.3.2 Recursive strategy

As highlighted in Section 4.3, taking into account the geometry and/or the topology of the environment will produce better partitions with more optimal load balance. The idea behind this is using geometry to guide the placement of partitioning planes in order to reach partitions with equal work, an equal number of polygons per partition. Naively cutting an environment in a uniform grid works only in case where polygons are uniformly distributed over the environment. The grid method fails when the distribution of polygons becomes too far removed from a uniform distribution.

The recursive strategy pre-processes and partitions the environment in a tree-structure which accounts for the number of polygons in a partition cell during runtime. It continues to subdivide a partition cell recursively until a stop condition is reached. In rough lines, the recursive strategy creates a quad tree structure of the environment. The root node is initialized by taking the entire input environment and its bounding box as the node's bounding box. A node in this tree-structure contains four children; north-east, north-west, south-west, south-east. Before the cutting process is applied a node's children are empty. After pre-processing is complete, the tree is pruned in order to generate a set of partitions $P_0, \cdots, P_n$.

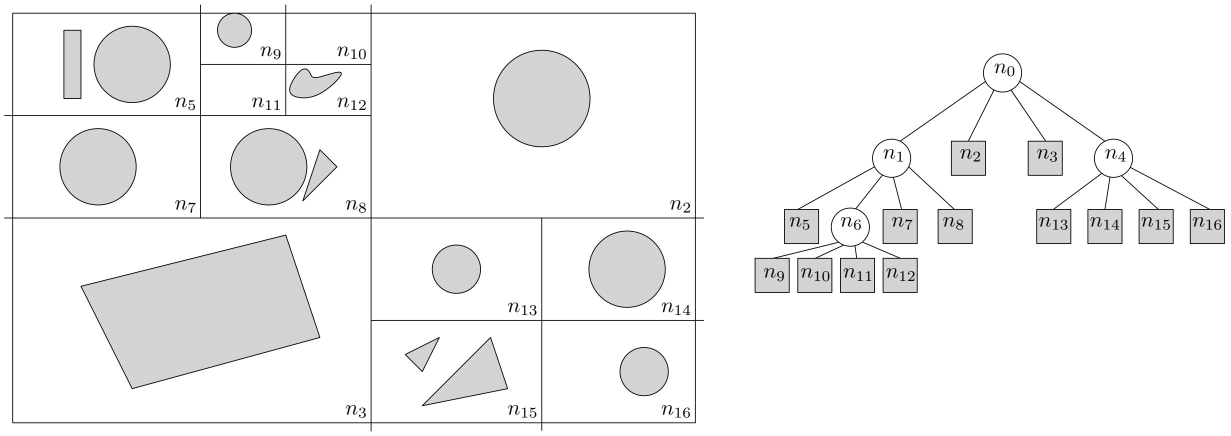


Figure 11: Recursive strategy overview. The left image represents an environment processed by the recursive strategy. The leaf nodes n_2 and n_3 reached their stop condition after only one recursion of the cutting algorithm, while the children of node n_6 reached their stop condition after 3 recursions. The right image represents the corresponding quad tree. Internal nodes are not labelled and triangulation is not shown for clarity.

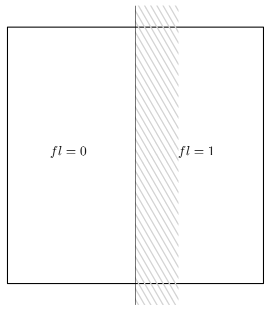
The process described is naturally modelled as a quad tree. A leaf in this quad tree represents a partition, where the set of polygons belonging to partition $P_0, \cdots, P_n$ is stored in the r-tree T of the leaf. Rejected polygons are stored in $T_{rejected}$ of the leaf. An internal node in the quad-tree corresponds to a cut operation performed on the environment and stores the two partitioning planes (horizontal and vertical) used to perform the cut operation.

Similar to the grid strategy, a set of coordinates is pre-calculated in order to prevent multiply defined expression trees to increase the overall memory use. The coordinate x_{half} is defined to be $x_{parent,min} + (width_{parent}/2)$ and a coordinate y_{half} is defined to be $y_{parent,min} + (height_{parent}/2)$. Using these two coordinates a partitioning plane pl_1 aligned to the xz-axis using y_{half} and a plane pl_2 aligned to the yz-axis using x_{half} can be created. The environment in a node is cut first by pl_1 followed by pl_2. Polygons fully or partially on the positive side of a yz-axis aligned plane pl_i have their flag fl incremented by 1. Since the amount of times a node is cut is constant (2) there is no need to account for varying amounts of cuts in the horizontal or vertical direction, like in the grid strategy where the amount of cuts is user-defined. Polygons fully or partially on the positive side of an xz-axis aligned plane pl_i have their flag incremented by 2. Polygons marked with $fl = 0$ are placed in the north-east child, $fl = 1$ in the north-west child, $fl = 2$ in the south-west child, $fl = 3$ in the south-east child. With these definitions, a cell P_i can be described by a finite set of half-planes by Boolean operations of set intersection on the input environment.

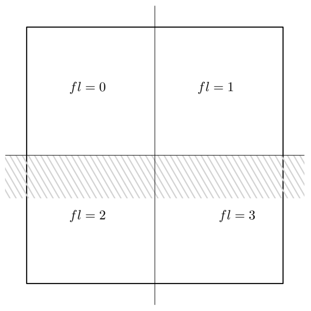
$$P_i = P \cup \{(x,y,z) : x >= x_{i,min}\} \cap \{(x,y,z) : x <= x_{i,max}\} \cap \{(x,y,z) : y >= y_{i,min}\} \cap \{(x,y,z) : y <= y_{i,max}\}$$

A big downside of the recursive strategy is the introduction of *t-joints* when the union of partitions is used to combine partitions back into one environment after being processed by a parallel pipeline. In cases where a node n_a (spatially) neighbours to another node n_b with a higher depth, vertices on the boundary of n_b will not necessarily exist in node n_a. If there exist vertices on the boundary of n_a not present in n_b, or vice versa, correct mesh connectivity is not guaranteed partition between nodes n_a and n_b.

Vertices on the north, south, east and west border in a node n_a which intersect with the partitioning planes p_1 and p_2 need to be introduced in all adjacent nodes with lower depth. By querying the neighbours and inserting the corresponding border vertex into its triangulation, mesh connectivity can be restored post-union. The following vertices need to be introduced for a node if a neighbour is found with lower depth.



(a) Partitioning plane pr_0 increments the flag fl on polygons on its positive side by 1.



(b) Partitioning plane pr_1 increments the flag fl on polygons on its positive side by 2.

Figure 12: A node processed by the recursive strategy.

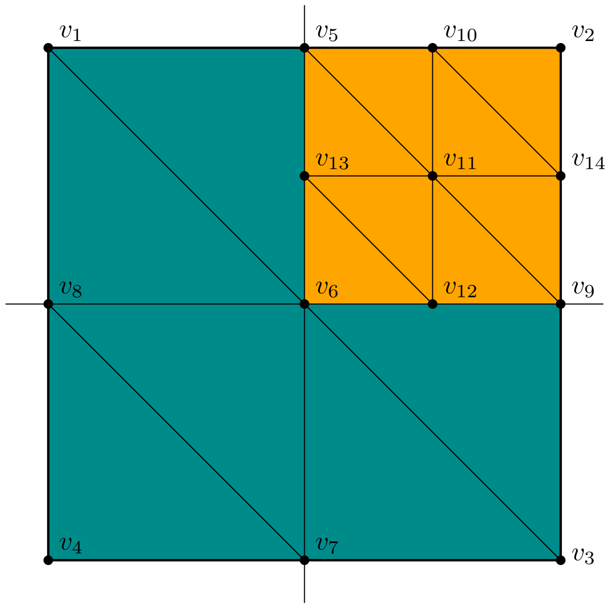


Figure 13: Minimal environment partitioned by recursive strategy. Three of the four orange nodes in the top-right with depth 2 are adjacent to the three cyan nodes with depth 1. Vertices v_{12} and v_{13} exclusively appear in the orange nodes and are not present in the cyan nodes. Because of this mismatch, these nodes cannot guarantee correct mesh connectivity.

- North neighbour of node n_i introduces a vertex $(x_{half}, y_{i,min})$ in every polygon for every intersection with the z-axis.
- South neighbour of node n_i introduces a vertex $(x_{half}, y_{i,max})$ for every intersection in the z-axis.
- East neighbour of node n_i introduces a vertex $(x_{i,min}, y_{half})$ for every intersection in the z-axis.
- West neighbour of node n_i introduces a vertex $(x_{i,max}, y_{i,min})$ for every intersection in the z-axis.

4.3.3 Sorted recursive strategy

Explored by the grid strategy and the recursive strategy is that both strategies fail in edge cases where geometry is sub-optimally partitioned. In cases such as the uniformly distributed environment in Figure 9 the strategies perform well, since the obstacle polygons are contained by partitioning planes by chance, while in cases such as Figure 14 we are left with two near empty partitions. Placing the partitioning plane in the center of the bounding box of a node

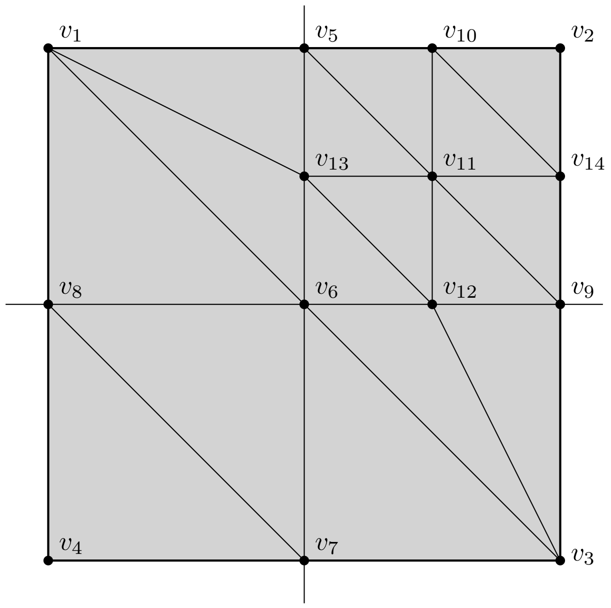


Figure 14: Minimal environment partitioned by recursive strategy after t-joints are fixed.

in the recursive strategy gives better results, such as in figure 15, but still results in edge cases where suboptimal partitions are generated. Furthermore, placing the partitioning plane in the center of the bounding box does not scale the recursive strategy on complex environments. As such, the need arises for a more complex solution.

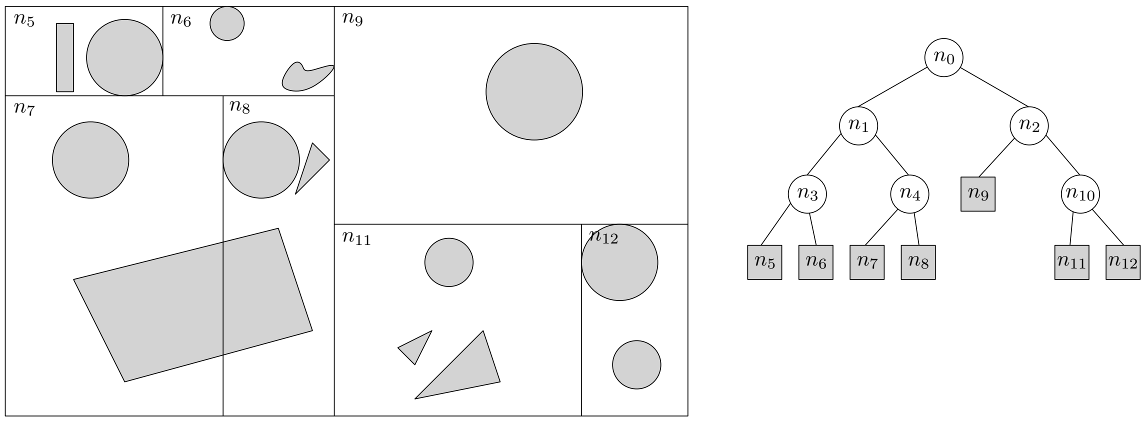
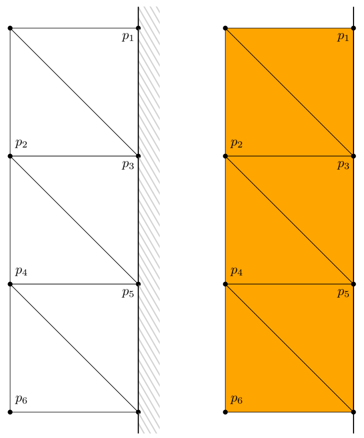


Figure 15: Sorted recursive strategy overview. The left image represents an environment processed by the sorted recursive strategy. The right image represents the corresponding binary tree. Internal nodes are not labelled and triangulation is not shown for clarity.

A simple extension on the recursive strategy that produces more optimal partitions by taking into account the environment's geometry is the sorted recursive strategy. The sorted recursive strategy functions as follows: in short, before every partitioning plane is cut into the environment, the polygons in the environment are sorted in ascending order by their x- or y-coordinate and the partitioning plane is placed on a polygon's x- or y-coordinate in the middle of the sorted list. The x- or y-coordinate used is the maximum x- or y-coordinate of a polygon's vertices. The maximum x- or y-coordinate is chosen to ensure the polygon in the middle of the sorted list (element at position $n/2$, is on the negative side of the partitioning plane.

By sorting the n polygons inside a node by their maximum coordinate in the x- or y-direction $x_0, \cdots, x_{n-1}$ or $y_0, \cdots, y_{n-1}$, a partitioning plane can be placed such that there exist at most $n/2$ polygons on the positive and negative side of the plane. (Note that this assumption only holds when all coordinates are unique.) To achieve this, the partitioning plane is placed at the coordinate $x_{n/2}$ or $y_{n/2}$, depending on the cut direction. By alternating the cut direction based on the depth of the node, the sorted recursive strategy process can be naturally modelled as a binary tree. A leaf in this binary tree represents a partition, where the set of polygons belonging to partition $P_0, \cdots, P_n$ is stored in the leaf's r-tree T. An internal node in the binary-tree corresponds to a cut operation performed on the environment and stores the partitioning plane (horizontal or vertical) used to perform the cut operation.

There are a few obvious edge cases where the sorted recursive strategy does not produce a guaranteed $n/2$ polygons after a cut. Firstly when all polygons have the same maximum coordinate x_{max} or y_{max} all polygons will exist on the boundary of the partitioning plane and partially on the negative side. Performing a cut at x_{max} or y_{max} will then result in one binary node (the node representing the negative side of the partitioning plane) owning all polygons, while the other node owns none. Without a stop condition this phenomena will produce an infinite recursion.



(a) Simple environment cut in the vertical direction where all x_{max} are non-unique.

(b) After performing vertical cut at x_{max} of element $n/2$ (p_3) all polygons are added to a singular partition P_a leaving P_b empty.

Figure 16: Environment where all polygons have non-unique x_{max}.

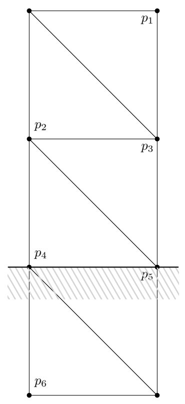
Secondly when a group of polygons of size m have the same coordinate x_{max} or y_{max} and contains element $n/2$ it is not guaranteed the cut will produce partitions of size $n/2$. If the first element of the group of polygons has position $n/2$ the partitioning plane will result in a cut placed at $x_{n/2}$ or $y_{n/2}$. Since in total $m - 1$ other polygons share the maximum coordinate of element $n/2$, polygons beyond $n/2$ will be put in the left node, because they lie on the negative side of the partitioning plane. In this case the left node will own $n/2 + m - 1$ polygons, while the right node will own $n/2 - m$ polygons.

In order to ensure edge case 1 does not occur, two actions are taken to prevent it. Firstly a stop condition is introduced to prevent infinite recursion. The stop condition is fulfilled when the next partitioning plane will result in at least one empty partition. Secondly, the root node will not be processed by a cutting operation if the partitioning plane aligned to element $n/2$ which creates non-empty partitions. In this situation the binary tree will contain only one node owning the original input environment.

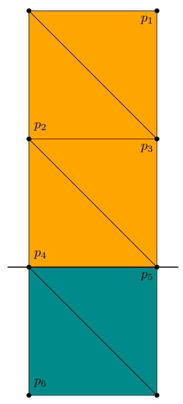
In all other cases it is expected that the sorted recursive strategy will provide a better fit in comparison to the grid- and recursive strategy and handle more types of input better.

4.3.4 Scheduling algorithm

At the distributed level partitioning is used to generate a number of tasks (jobs) which are not necessarily equal to the number of nodes available. A partitioning method creates n jobs, while m nodes are available in a distributed network, where n can be higher, lower or equal to m.



(a) Simple environment cut in the horizontal direction where all x_{max} are non-unique.



(b) After performing horizontal cut at y_{max} of element $n/2$ (p_3) four polygons are added to partition P_a and two polygons are added to partition P_b.

Figure 17: Environment where a group of polygons ($m = 2$) containing element $n/2$ have non-unique y_{max}.

In the case where $n < m$, the observation can be made that either the environment which is split up into n jobs was improperly handled by the partitioning method. Alternatively, for the recursive partitioning strategy, it can be observed that the stop condition was reached too early due to misconfiguration of the parameter controlling the stop condition. For the grid partitioning strategy it can be the case that it was poorly configured as well, when $n_{rows} * n_{columns} < m$.

In the case where $n > m$, a multiprocessor scheduling problem arises, which is an NP-hard optimization problem. The multiprocessor scheduling problem is defined as follows: Given n non-pre-emptive (i.e., cannot be divided) independent jobs and m identical processors (or machines), minimize the total timespan required to process all the jobs. The assumption is made that all parallel machines are identical.

To approximate a solution for this problem, the **longest processing time** (LPT) approximation algorithm is used to divide n partitioned tasks with a cost c over m task bins. In the context of the geometry-processing pipeline, cost c is defined to be the number of polygons, since all filter operations scale on the number of polygons.

The LPT algorithm sorts the tasks $1, \cdots, n$ in descending orders of their cost c_j ($j = 1, \cdots, n$) and assigns one task sequentially to the task bin whose load is smallest so far. Each node then takes ownership of the task bin assigned to them. Ownership of a task bin is determined deterministically using an integer value *rank* assigned by the implementation of MPI, where $1 > id > m$, so the process can be replicated on all nodes of the network without communication.

4.4 Rounding exact number(s)

Due to the memory footprint and pointer structure of the memory structure used by the *CORE::Expr* exact number type, it becomes necessary to reduce the memory used by the vertex coordinates in larger environments. Representing the result of the expression tree in a non-exact number type reduces the size of a coordinate's expression trees to a singular node, namely a node carrying a constant; the simplified, rounded value. In this section the effects of representing an exact number type, such as *CORE::Expr*, in a number type where exactness is not guaranteed, are explored. Geometry errors introduced by this process are explained, as well as their effects on the pipeline. Specifically, what assumptions, definitions and properties of filter operations are violated by representing a vertex' coordinates in a non-exact manner and these geometrical errors. Lastly, algorithms are proposed on how these geometrical errors can be repaired to pass the requirements of filter operations after rounding in order to extracting the environment's correct walkable area, as defined by [Polak].

Without modifications it is not possible to evaluate coordinates of machine precision with high speed. In Section 5.1 necessary modifications to existing geometry filters are discussed.

4.4.1 Floating-point format

Because computer memory is limited, it is not possible to represent real numbers with infinite precision. Floating-point formats, however, can be used to represent a finite subset of real numbers in a limited number of bits. In the current industry standard, IEEE-754, the memory imprint of number in a floating point format is 32 bits (single precision floating-point), 64 bits (double precision floating-point) or 128 bits.

Definition 4.1. Floating point number A floating point number x is a real number $x \in \mathbb{R}$ for which there exists at least one representation (M, e) such that

$$x = M * b^{e-p+1},$$

- b is the **radix**, where $b >= 2$ and $b \in \mathbb{N}$.
- p is the number of digits in the significand (**precision**), where $p >= 2$ and $p \in \mathbb{N}$.
- $e \in \mathbb{N}$ is the **exponent**, where $e_{min} <= e <= e_{max}$.
- e_{max} is the **maximum exponent**, where $e_{max} \in \mathbb{N}$
- e_{min} is the **minimum exponent**, where $e_{min} = 1 - e_{max}$
- $M \in \mathbb{N}$ is the **significand**, where $|M| <= b^p$ and $|M| \in \mathbb{N}$.

The double precision floating point format is used as the non-exact number type, as defined by the IEEE Standard for Floating-Point Arithmetic in [Zur+08], *IEEE 754*. The *IEEE 754* standard specifies formats for binary and decimal floating-point data, for computation and data interchange [Zur+08] and is the current standard for computation with floating-point numbers in a binary environment. The primary reason for the choice being that, other than the IEEE 754 being the industry standard, *CORE::Expr* to double precision floating point format conversion methods are readily available in the used *CGAL* package. For the remainder of this section, binary floating-point numbers are described and handled, which assume a radix b of $b = 2$.

Definition 4.2. Binary floating point number A binary floating point number x is a floating point number which can be represented with radix $b = 2$.

The relevant binary floating-point formats introduced by the *IEEE 754* standard are the single-precision floating point format (referred to as **float** in C-style languages) and the double-precision floating point format (referred to as **double**). These floating-point formats are defined by the following parameters:

parameter	**single-precision format (32 bits)**	**double-precision format (64 bits)**
b	2	2
p	24	53
e_{max}	+127	+1023
e_{min}	-126	-1022

Table 1: Parameters for single- and double precision floating point formats

Since the implementation of the double-precision floating point format, described by the *IEEE 754* standard, is so widely adapted and the implementation of converting the exact number type *CORE::Expr* is readily available and well-documented, it was chosen as the number type to which the exact number type is simplified to. As described earlier, a radix b with value 2 is chosen to create a binary floating-point format, due to the binary number system used by all modern computers.

Mentioned earlier is the fact that a set of floating point numbers is a finite set, limited by the radix, the significand and the minimum and maximum exponents. Since the set of real numbers $\mathbb{R}$ is an infinite set and a floating point set is finite, not every real number $x \in \mathbb{R}$ is representable in a floating point format. The set of real numbers representable by a floating point format will be referred to as a floating point set.

Definition 4.3. Floating point set Consider a floating point system defined by b, p, e_{min} and e_{max}. The set of all floating point numbers representable in floating point system F is defined by

$$F = \{M * b^{e-p+1} || M \leq b^p - 1, e_{min} \leq e \leq e_{max}\}$$

On the other hand there are an infinite number of real numbers in $\mathbb{R}$, and the IEEE 754 format can only represent a limited amount of numbers exactly due to its finiteness. A real number $x \in \mathbb{R}$ which is not representable in the floating point set F is rounded to the nearest floating point number in the set using rounding rules defined by the IEEE 754 format. By rounding a real number x to be representable in a floating point set to the number $\tilde{x}$, it becomes an approximation of x and will have a *rounding error.* Real numbers outside of the range of representable numbers will, however, underflow or overflow. Overflowing numbers, numbers above the largest machine number $1 - 2^{-53} * 2^{e_{max}}$, cannot be represented in a floating point format and are truncated to be $+\infty$, while underflowing numbers are considered 0. Overflowing numbers cannot be handled by filter operations, specified in subsection 3.2. In the context of the pipeline, we assume that all vertex coordinates are bound by the range of the floating point set and cannot overflow when represented in a floating point format.

Definition 4.4. Rounded value The approximated value of a real number x after rounding to the double-precision floating point format is the value $\tilde{x}$. $\tilde{x} \in F$ and $x \in \mathbb{R}$

Due to the nature of rounding a real number to the nearest representable floating point value, *rounding errors* occur. Since floating point numbers are represented by only a finite number of bits, they have a granularity and cannot represent all real numbers. If the floating point number nearest to the real number x is chosen, then the maximum rounding error occurs when $x = \frac{y+z}{2}$, where y and z are representable numbers where x lies inbetween. In this worst case, the relative rounding error $|(\tilde{x} - x)/x|$ is bounded by the machine epsilon $\epsilon_m = 2^{-p}$. For the double precision floating point format the machine epsilon is $2^{-53} \approx 10^{-16}$.

Definition 4.5. Machine epsilon For any binary floating point format, the machine epsilon ϵ_m is the difference between 1 and the next larger number that can be stored in that format.

Bringing this all together, we can define properties for real numbers and vertices relating to their representability in the double-precision floating point format with which we highlight the geometry errors resulting from representing a guaranteed exact, real number in a finite floating point set.

Definition 4.6. Exactly representable real number A real number x is able to be represented exactly in the double-precision floating point format, when $x = \tilde{x}$.
Definition 4.7. Not exactly representable real number A real number x is unable to be represented exactly in the double-precision floating point format, when $x \neq \tilde{x}$.

Using the following vertex properties it becomes possible to define the geometrical errors and problems which can occur when vertex coordinates are represented in a floating point format.

Definition 4.8. Exactly representable vertex A vertex v is able to be represented exactly the double-precision floating point format, when all of the following $v_x = \tilde{v_x}$, $v_y = \tilde{v_y}, v_z = \tilde{v_z}$ hold.
Definition 4.9. Not exactly representable vertex A vertex v is unable to be represented exactly in the double-precision floating point format, when one of the following $v_x \neq \tilde{v_x}$, $v_y \neq \tilde{v_y}, v_z \neq \tilde{v_z}$ hold.
Definition 4.10. Partially exactly representable vertex A vertex v is partially able to be represented exactly in the double-precision floating point format, when one of the following $v_x = \tilde{v_x}$, $v_y = \tilde{v_y}, v_z = \tilde{v_z}$ hold.

4.4.2 Geometry errors and solutions

In this section the geometry errors caused by reducing vertex coordinates' *CORE::Expr* expression trees to constant values of the double precision floating point format is explored. The geometry errors are explored on a per-triangle level and on a per-triangle-strip level. Per-triangle level geometry errors usually reduce the original triangle to a degenerate polygon, while on a per-triangle-strip level more complex problems occur.

Before we define all geometry errors which can occur, the only situation which allows for a polygon which guarantees no geometry errors is an **exactly representable polygon**. An exactly representable polygon occurs when all vertices owned by the polygon are exactly representable vertices. The coordinates represented by the original expression tree number format are able to be represented exactly in the double precision floating point format. No geometrical errors occur in this case, but rounding after subsequent filter operations can still introduce geometrical problems.

Geometry case 1. ***Exactly representable polygon*** *A polygon p is exactly representable if all vertices v_1, ..., v_n are exactly representable vertices.*

In all other cases, errors in the geometry can occur in numerous ways, but to understand them, we need to define the **degenerate polygon** first.

Definition 4.11. Degenerate polygon A polygon p is degenerate when one or more vertices share the same coordinates. A degenerate polygon occurs in four cases, namely $v_1 = v_2 = v_3$ (reduces polygon p to a single point), $v_1 = v_2$, $v_2 = v_3$ or $v_1 = v_3$ (which all reduce polygon p to a single edge).

When a vertex is not exactly representable in the double precision floating point format the following geometry error-cases can occur:

Geometry case 2. ***Not exactly representable degenerate polygon (edge)*** *A polygon p is a not exactly representable degenerate polygon if at least one vertex v_a is (at least partially) not exactly representable and the rounded value of $v_{a,x} = v_{b,x}, v_{a,y} = v_{b,y}$ and $v_{a,z} = v_{b,z}$, where v_b is a vertex different from v_a owned by polygon p.*

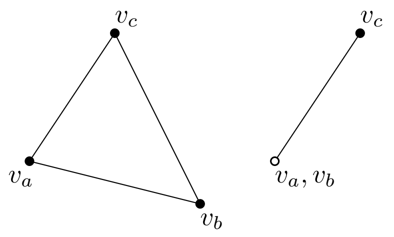


Figure 18: Not exactly representable polygon where two vertices are represented by the same vertex coordinates, after rounding, and create a degenerate polygon.

The geometry case 2 refers to a polygon which degenerates to a polygon with only two unique vertices: an edge. If we assume v_a and v_b to represent the vertices with non-unique coordinates and v_c a vertex with unique coordinates, we can define this case further. If an edge e (v_a, v_c) or (v_b, v_c) is an obstacle edge and edge (v_a, v_c) or edge (v_b, v_c) is a shared edge with a connected polygon $p_{connected}$ in p_2, ..., p_n outside of polygon p_1 and the obstacle edge e does not exist in the connected polygon $p_{connected}$, then the obstacle property of edge (v_a, v_c) or $(v_b, v_c$ needs to be projected to the polygon $p_{connected}$. If the obstacle edge in a degenerate polygon is not projected to polygons connected to the obstacle edge, then whenever degenerate polygons are removed in subsequent filter operations the obstacle edges are lost and subsequently the (at least partially) rounded environment will produce different a different walkable environment compared to unrounded environments.

Algorithm 3 Projecting obstacle edges to neighbors

```
1: $obstacleedges \leftarrow$ obstacle edges in $\mathrm{P}_{degenerate}$
2: for each neighboring polygon $P_{neighbor}$ with vertices in $obstacleedges$ do
3:     for each edge $(v_a, v_b)$ in $obstacleedges$ do
4:         if edge $(v_a, v_b) \in P_{neighbor,edges}$ then
5:             Mark edge $(v_a, v_b)$ as obstacle edge.
```

Geometry case 3. ***Not exactly representable degenerate polygon (point)*** *A polygon p is a not exactly representable degenerate polygon if at least one vertex v_a is (at least partially) not exactly representable and the rounded value of $v_{a,x} = v_{b,x} = v_{c,x}, v_{a,y} = v_{b,y} = v_{c,y}$ and $v_{a,z} = v_{b,z} = v_{c,z}$, where v_b and v_c are unequal to v_a and are owned by polygon p.*

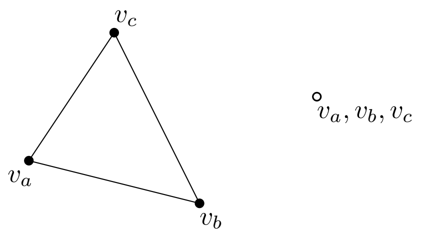


Figure 19: Not exactly representable polygon where all vertices are represented by the same vertex coordinates, after rounding, and create a degenerate polygon.

The geometry case 3 refers to a polygon which degenerates to a polygon with only one unique vertices: a point. Since this case degenerates to a point, the projection of obstacle edges is ignored in this method, since obstacle edges of length 0 are considered to be invalid obstacles.

Both cases, 2 and 3, if a degeneracy filter has been applied to the environment in which a polygon resides of these cases, violate the post-assumption that there are no degenerate polygons in the environment. As such, subsequent filter operations which pre-suppose a degeneracy-free environment, namely the simplification filter and the layer subdivision filter, will produce an unexpected and incorrect output. To correct the newly created degenerate polygons in the rounding process, the degeneracy filter has to be applied after every rounding step to ensure the lack of degenerate polygons in the environment and to ensure obstacle edges are not lost by these degeneracies by projecting them correctly to surrounding polygons.

Geometry case 4. ***Not exactly representable collinear polygon*** *A polygon p is a not exactly representable collinear polygon if at least one vertex v_a is (at least partially) not exactly representable and the rounded coordinates of the vertices v_a, v_b and v_c owned by polygon p fulfil the condition of collinearity.*

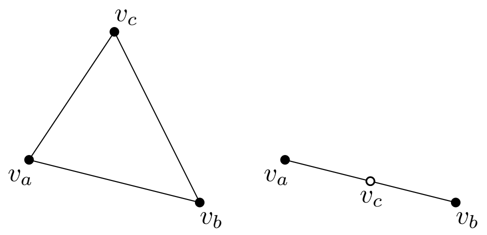


Figure 20: Not exactly representable polygon where all vertices are, after rounding, on the same line and create a degenerate polygon where all vertices are collinear.

Geometry case 5. ***Not exactly representable flipped polygon*** *A polygon p is a not exactly representable flipped polygon if at least one vertex v_a is (at least partially) not exactly representable and the rounded coordinates of the vertex v_a lies on the negative side of the edge described by the vertices v_b and v_c.*

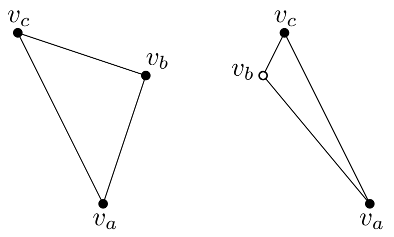


Figure 21: Not exactly representable polygon where a vertex, after rounding, causes the polygon to have a vertex v_b on the negative side of edge (v_a, v_c).

A polygon p with geometry case 5 describes a polygon which, before rounding, is a valid polygon, but after rounding becomes a polygon whose vertices are collinear and exist on a line l. A polygon with collinear vertices violates the presumption of the layer subdivision filter that polygons need to have a non-zero area. To ensure a correct outcome of the area filter and the layer subdivision filter this presumption must be fulfilled. Collinear polygons which form after rounding vertex coordinates are corrected in the exact way geometry case 2 is handled. All collinear polygons are removed from the corresponding environment after their obstacle edges are projected to polygons which share its obstacle edges (if the edges have any).

The geometry error cases presented earlier can occur in polygon strips as well, but can also result in cases unique to polygon strips consisting of ≥ 2 connected triangles. The cases unique to polygon strips can result in broken preconditions and effects for filter operations in different situations than the ones presented in the single triangle geometry cases.

Definition 4.12. Minimal polygon strip The minimal polygon strip is a polygon strip (p_1, p_2) consisting of $n = 2$ polygons p_1 and p_2. The vertex v_1 is exclusively owned by polygon p_1, the vertices v_2 and v_3 describe the shared edge (v_2, v_3) between polygons p_1 and p_2 and the vertex v_4 is exclusively owned by polygon p_2.

A polygon strip consisting of $n > 1$ polygons, where at least 1 polygon is a *not exactly representable polygon*, can, after rounding, break the planarity constraint of the polygon strip the polygon strip fulfilled before rounding. The minimal case in which this can happen is in the minimal polygon strip. Since p_1 is an *exactly representable polygon*, it follows that vertices v_1, v_2 and v_3 are exactly representable vertices. Vertex v_4 is exclusively owned by *not exactly representable polygon* p_2 and is a *not exactly representable vertex*. From these it follows that $v_4 \neq ne(v_4)$ and the planarity of polygon strip (p_1, p_2) described by the plane constructed from vertices (v_1, v_2, v_3) is broken.

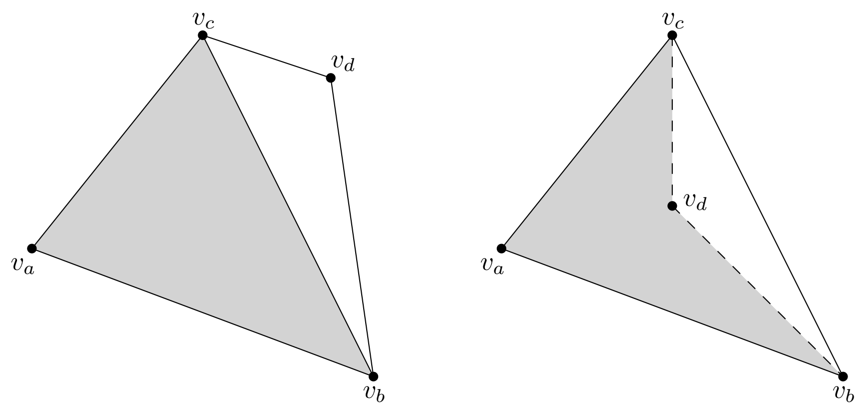


Figure 22: Self-intersecting polygon strip appearing in the minimal polygon strip after rounding. The vertices v_a, v_b and v_d are exactly representable vertices, while v_c is a not exactly representable vertex. After rounding vertex v_d switches sides relative to edge (v_b, v_c).

A polygon strip consisting of $n > 1$ polygons, where at least 1 polygon is a *not exactly representable flipped polygon*, can, after rounding, construct a polygon strip where polygons intersect or overlap other polygons inside the polygon strip, where before rounding no intersections or overlaps were present. Self-intersecting (or self-overlapping) polygon strips, further referred to as self-intersecting polygon strips, violate post-conditions in the height clearance- and degeneracy filter operation and pre-conditions in the layer subdivision filter operations. The minimal case in which self-intersecting polygons occurs in a polygon strip is in the minimal polygon strip. When $ne(v_4)$ lies on the negative side of the shared edge (v_2, v_3), the polygon p_2 will overlap (or intersect) with polygon p_1 self-intersecting polygon is introduced in the minimal polygon strip (p_1, p_2).

By collapsing vertex v_4 to the closest vertex in the shared edge (v_2, v_3) to the polygon p_1, polygon p_2 is reduced to a degenerate polygon and can be removed from the polygon strip. The obstacle edge property from edges (v_2, v_4) or (v_3, v_4) is transferred to shared edge (v_2, v_3) to maintain correct walkability constraints of the polygon strip. In this geometry case, the vertex coordinate $\tilde{v_4}$ is a not exactly representable vertex or a not exactly representable underflowing (or overflowing) vertex. Since the assumption is made that vertex coordinates can be represented within the limits of the double precision floating point format it follows that in all cases the relative maximum error between exact coordinates v_4 and rounded, not exact coordinates $ne(v_4)$ in each of their dimensions is at most half the machine epsilon ϵ_m of the double precision floating point format. The difference between v_4 and $\tilde{v_4}$ is negligible, and, as such, vertex v_4 can be collapsed.

5 Experiments

Four types of experiments are ran in order to evaluate the proposed parallel filtering pipeline. An inter-node communication benchmark is ran in order to determine running the parallel pipeline on a distributed network is worth it in terms of bandwidth use and (potential) stalling. The secondary goal of the communication benchmark is to find the best (in terms of speed) communication method.

In the second experiment we test the three different proposed partitioning methods. A set of polygonal environments is processed using a pre-set pipeline and compared in terms of execution speed and speed-up factor. The goal of this experiment is to find which partitioning method performs best per environment and which partitioning method performs the best overall.

In the third experiment a variable number of threads is taken to measure how well the parallel pipeline scales on different hardware.

The final experiment evaluates the distributed variant of the parallel pipeline to examine its faults in terms of overhead costs induced by communication.

5.1 Implementation

The parallel pipeline was implemented by expanding the filtering pipeline from [12]. Since the original implementation used C++ and the Computational Geometry Algorithms Library (CGAL) the implementation of the parallel pipeline uses this as well. The CORE library's *Core::Expr* number type was used as the primary number type with numerical accuracy level 3. Rounding coordinates by forcefully demoting number types to the machine accuracy level is used to prevent the pipeline from using excessive memory and computation time after partitioning an environment.

5.1.1 Rounding filter & rounding intersections

In order to implement the theory behind demoting a number type of accuracy level 3 to machine accuracy without any resulting degeneracies or mesh errors from Chapter 4.4 a new filter is introduced in the pipeline, i.e. the *RoundingFilterFloatingPoint* filter. There also is a rounding step introduced in each partitioning method, which rounds the expression graphs of the intersection points of the partitioning planes with the environment back to a single node, referred to as the intersection rounding step. For the rest of this section the *RoundingFilterFloatingPoint* filter will be referred to as the rounding filter for brevity.

The rounding filter and the intersection rounding step work using the same principles. A set of vertices have the expression trees of their x-, y- and z-coordinates rounded into a number type of machine accuracy by calculating the result of the x, y and z expressions, in this case the IEEE-754 standard's *double*. By taking the rounded values in double format a new *Core::Expr* is created initialized with the rounded value as the root node of the expression. In essence, each expression tree is demoted to machine accuracy forcibly. Because the *Core::Expr* number type supports automatic promoting and demoting exactness is kept in filter steps *after* the rounding is performed. Whenever higher accuracy is needed the original *double* in the root node can easily be promoted to a number type with higher accuracy.

For the rounding filter this set of vertices is the entire set of vertices within an environment (or sub-environment). The full set of vertices is found by iterating through the polygon set P belonging to the environment. The rounding filter will then calculate the result of the expression tree of the x-, y- and z-coordinates and replace the expression tree with a node with the rounded value. The rounded value is only used when the rounded coordinate is not equal to the not rounded coordinate, since multiple polygons can point to the same vertex. The check is not only performed to check for vertex uniqueness, but it is also performed to not create extraneous nodes in the expression graph structure. If (potentially) a vertex is rounded multiple times in a row, each time it is rounded a new expression graph is created which increases the size and complexity of the dynamic pointer structure used by the *Core::Expr* number type. Allocating memory and deallocating memory for the nodes these multiply rounded coordinates use also substantially increases the time spent performing overhead.

The intersection rounding step the set of vertices is the set of vertices intersecting the partitioning planes and the vertices created while fixing t-joints. For the same reasons, a coordinate is only rounded when it does not equal the rounded coordinate.

5.1.2 Simplification filter

The simplification filter was slightly altered to be compatible with rounded numbers, as described in Chapter 5.1.1. The original implementation by Polak uses the *CGAL* library's *CGAL::coplanar* function in order to determine polygons belong to the same polygon group, but since rounding introduces inaccuracies and geometry errors polygons with rounded vertices which were coplanar before rounding can suddenly lose their coplanarity. To restore correct functionality for these rounded polygons, a dot product is used on the normal of a polygon and the normal of a potential neighbour as a surrogate for the exact coplanarity check of *CGAL*. By comparing the dot product of the normal of a polygon and its neighbour to $1.0 - 2.2e^{-16}$ we can determine if the neighbour belongs to a connected polygon group. Values $> 1.0 - 2.2e^{-16}$ are considered coplanar, due to potential inexactness caused by the rounding factors mentioned before.

5.2 Testing environment

All our experiments were run on Intel Core i7-5960X@3.00GHZ machines with 16 cores (32 cores with hyper-threading) and 32GB of RAM at $3099.3MHz$. The OS installed on the machine is Windows 10, 64bit.

5.3 Communication benchmark

To test the performance of the proposed solutions for inter-process communication a set of random test environments is generated which are written to a consumer-grade hard-disk. To test the binary graph file-format in a realistic manner, the expression trees are grown by applying random binary operations filled with random values. The binary graph format is split into three types: an expression graph with one node, an expression graph with three binary operations applied (a *shallow graph*) and an expression graph with twelve binary operations applied (a *deep graph*).

Tests are ran on a set of n randomly generated expression trees, where n can be: 1, 1000, 10000 and 1000000. $n = 1$ is taken to measure the minimum cost of opening and closing a file, while $n = 1000000$ is chosen to represent an average, large environment.

5.3.1 Implementation

For the ascii-format, results are stored in ASCII-format with the defined maximum of 17 digits precision. Only the memory usage of the coordinates of the vertices in the three-dimensional environment is considered in all comparisons.

The storing of a sequence of n resulting values uses the following amount of memory per value: 1 (possible) byte for a sign 1 byte for a decimal point 1 byte for a white space 17 bytes for digits

Assuming coordinates are represented by three axes, one byte is required to represent a line break. The total amount of memory used in the absolute best case (no sign characters) would then become: $19n + (n/3)$ bytes.

For an environment consisting of 1000000 (one million) vertices the cost to store the environment would then become: $19 * 3000000$ bytes $+$ 1000000 bytes $=$ 58MB.

The storage of a sequence of **n** resulting values in a binary format, using the double floating point number type specified by the *IEEE 754* standard, uses **8** bytes of memory to represent a result in the floating point number type.

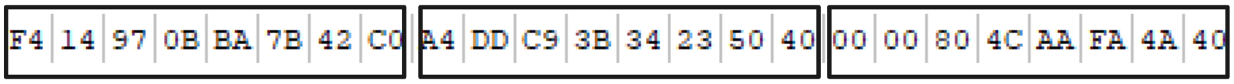
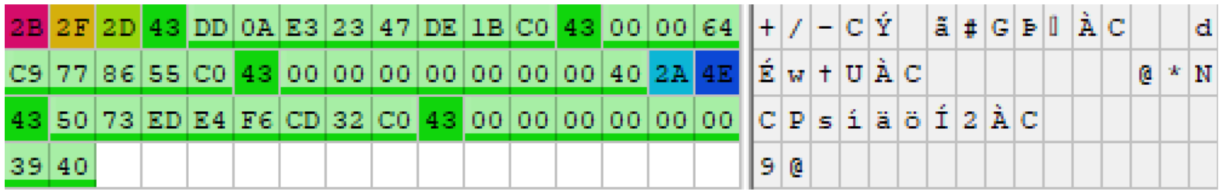


Figure 23: Approximated binary-format of XYZ-coordinate

Assuming coordinates are represented by three axes, the total amount of memory used in all cases would then become: $24n$ bytes.

For an environment consisting of 1000000 (one million) vertices the cost to store the environment would then become: $24 * 1000000$ bytes $= 24$MB.

Lastly, for the expression graph file type, implementation-wise, the parent (top) node of an expression graph is entered and its children are recursively stored into a binary tree format, where splits (binary mathematical operators) in the expression graph have their left nodes take priority in the recursion having priority over their right nodes. Non-leaf nodes are unary/binary mathematical operations, while leaf nodes are constant values. The memory usage per node type is as follows:



```
2B 2F 2D 43 DD 0A E3 23 47 DE 1B C0 43 00 00 64   + / - C Ý   ã # G Þ   À C       d
C9 77 86 55 C0 43 00 00 00 00 00 00 00 40 2A 4E   É w † U À C               @ * N
43 50 73 ED E4 F6 CD 32 C0 43 00 00 00 00 00 00   C P s í ä ö Í 2 À C
39 40                                             9 @
```

Figure 24: Binary tree-format representation of the expression $(35.9922 - (-84.9573))/2 + negate(2.642) * 25$.

Constant node (double)
1 byte representing the node type
8 bytes representing the value

Constant real number node (double)
1 byte representing the node type
1 byte representing the real number type (double)
8 bytes representing the value

Constant real number node (long)
1 byte representing the node type
1 byte representing the real number type (long)
4 bytes representing the value

Constant real number node (GMP BigInt)
1 byte representing the node type
1 byte representing the real number type (big int)
4 bytes of size information m
m bytes of limbs

Constant real number node (GMP BigFloat)
1 byte representing the node type
1 byte representing the real number type (BigFloat)
4 bytes representing the exponent
4 bytes of size information m
m bytes of limbs

Constant real number node (GMP BigRat)
1 byte representing the node type
1 byte representing the real number type (BigRat)
4 bytes of size information m_1
m_1 bytes of limbs
4 bytes of size information m_2
m_2 bytes of limbs

The absolute worst-case of memory usage in this format is when the expression graph forms a fully-completed binary tree. In this case, a maximum number of node type information bytes has to be written and the maximum number of constant values has to be stored.

Depending on the number type of the constant values, the memory used by this method can vastly outscale the memory used by the methods which use approximate values. The expression graph number type has no maximum depth, so each coordinate could potentially be represented by a worst-case expression tree, leading to a vast amount of memory used.

The absolute best-case, however, is when the expression graph owns a singular leaf node with a double floating point constant value, which provides a memory usage of the best case of $9n$ bytes.

5.3.2 Results and discussion

Mean reading time (in ms)	1	1000	10000	1000000
Binary	0.7	2.8	2.2	143.2
Ascii	3.3	6.9	45.3	3446.8
Binary graph (d = 12)	7.0	12.4	40.8	2831.2
Binary graph (d = 3)	6.9	12.5	25.2	814.9
Binary graph (d = 1)	6.5	10.1	12.5	114.3

Table 2: Mean reading time on 1 to 1,000,000 values in binary, ascii and binary graph format. For the binary graph format, d represents the depth of a full binary graph in which every node, other than the leaves, has two children.

Mean writing time (in ms)	1	1000	10000	1000000
Binary	4.2	23.4	111.0	9146.3
Ascii	5.9	46.7	204.7	17042.8
Binary graph (d = 12)	0.0	3.7	26.6	2156.2
Binary graph (d = 3)	0.0	0.7	10.7	606.0
Binary graph (d = 1)	0.0	0.0	1.8	103.0

Table 3: Mean writing time on 1 to 1,000,000 values in binary, ascii and binary graph format. For the binary graph format d represents the depth of a full binary graph, in which every node other than the leaves has two children.

The absolute slowest method, as expected, is the ASCII file-format based method in both writing and reading speeds, the serialization and deserialization from and to text dominates the writing and reading process, making the method an obvious choke point if the parallel pipeline is ran on a distributed pipeline.

Another observation from these experiments is that the raw binary file-format is objectively better than the ASCII-format in all regards, since it requires less memory, represents the resulting floating point number in the working memory exactly and has a negligible reading speed. The writing speed can be massively improved by using the already available precomputed answer to the *Core::Expr* expression graph in the root node, however it must be the case that this available precomputed answer is either the same or within a margin of error from the computed answer.

The graph binary format is faster (during writing primarily) than the ASCII-format and raw binary format, since the memory used per node can be copied into a binary format exactly as it is available in the working memory and as such no resulting values need to be computed. The major downside being the reading speed scaling up when there is a high number of binary operators in the expression in the worst case. This is due to a high number of constant values being present in the leaf nodes.

5.4 Parallel pipeline benchmark

To compare the performance of the parallel pipeline compared to the previously defined sequential pipeline, execution times are measured of a predefined non-parallel pipeline and a parallel pipeline applied to a set of environments. Each partitioning method is tested separately in order to find the most optimal configuration of the parallel pipeline. In total, each environment is tested three times for the parallel pipeline depending on the partitioning strategy used and once for a non-parallel pipeline to obtain the non-parallel execution time.

All aspects of the parallel pipeline are measured; the total running time t_{total}, the time spent preprocessing the environment $t_{preprocessing}$, the time spent partitioning the environment $t_{partitioning}$, the time spent processing the environment in parallel $t_{parallel}$, the time spent merging all parallel results $t_{merging}$, the time spent post-processing the re-merged environment $t_{postprocessing}$ and time spent due to overhead $t_{overhead}$. $t_{overhead}$ is not measured directly, but is instead calculated from existing values $t_{overhead} = t_{total} - t_{partitioning} - t_{parallel} - t_{merging} - t_{preprocessing} - t_{postprocessing}$. For the non-parallel pipeline only the total running time t_{total} is measured, since the non-parallel pipeline does not require the intermediate steps the parallel pipeline does require.

Finally, the measure of speed-up is used to determine whether or not the parallel pipeline offers a significant boost in speed in comparison to the non-parallel pipeline. Since we already have measured the execution time for one processor $T(1)$ as the execution time of the non-parallel pipeline and the execution time for p processors as the execution time of the parallel pipeline as t_{total}, we can calculate the speed-up [EZL89] measure using the following formula:

$$S = \frac{t_{total,sequential}}{t_{total,parallel}}$$

The sequential pipeline is ran with a degeneracy filter to remove degeneracies, a steepness filter, a duplication filter with a low threshold, the vertical clearance filter, an area filter to remove small areas, a simplification filter to reduce the overall mesh complexity and the layer subdivision filter to obtain a multi-layered environment. The sequential pipeline is defined with the following settings:

```
DegeneracyFilter
SlopeFilter 60
DuplicationFilter 0.005
VerticalClearanceFilter 3
DegeneracyFilter
DuplicationFilter 0.005
AreaFilter 500
SimplificationFilter
PEELFilter
```

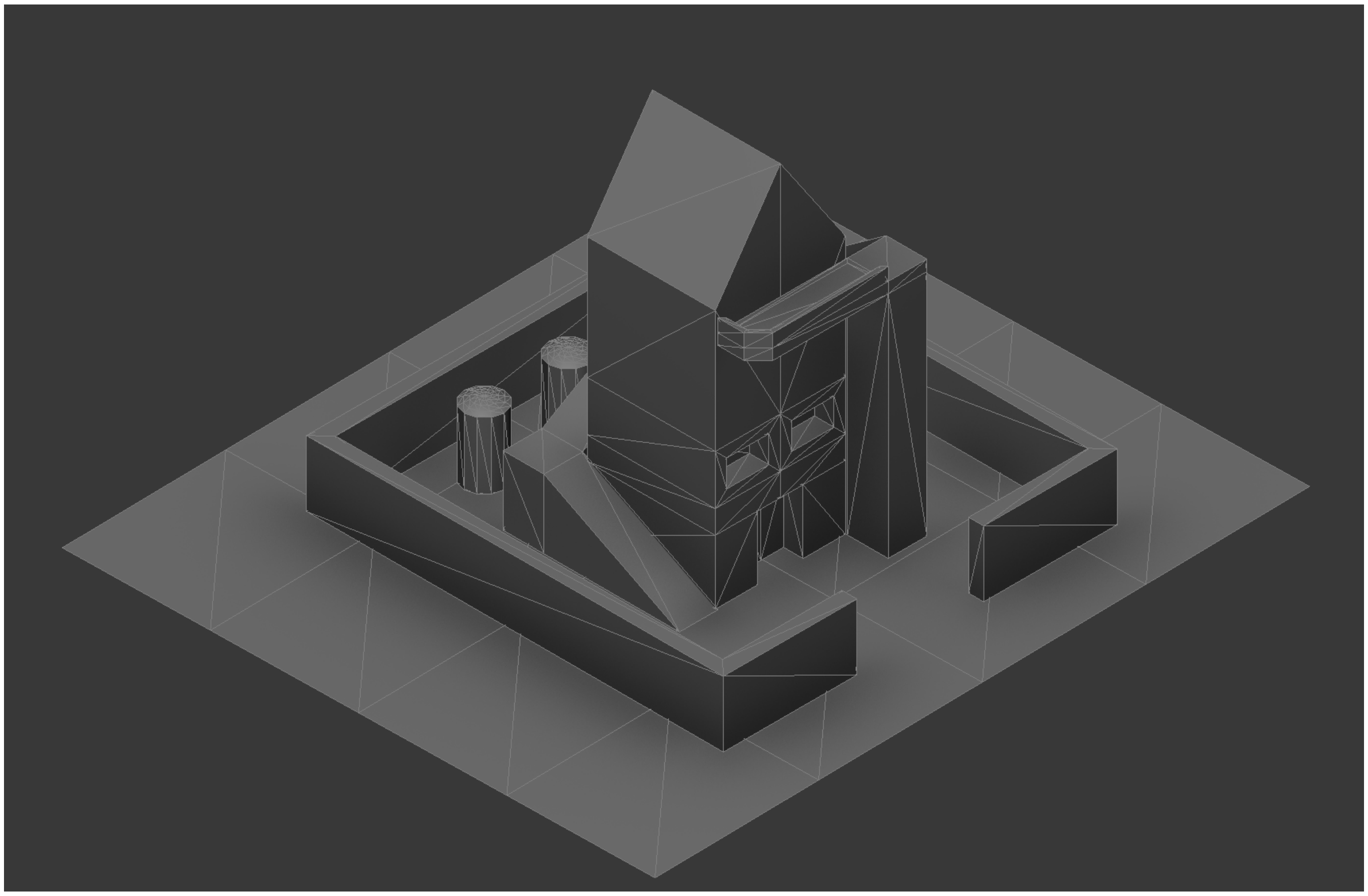
All settings between running the sequential pipeline and the parallel pipeline are kept the same: parameters are identical, the filter order is kept intact. Only filters relating to the parallel processing are different, such as the rounding filter or running multiple duplication filters to restore mesh connectivity. The parallel pipeline is ran with a degeneracy filter, a steepness filter and a duplication filter with a low threshold in the pre-processing block. After pre-processing the environment is partitioned using the grid strategy, the recursive strategy or the sorted recursive strategy. The partitioned environments all are processed using a degeneracy filter to fix degeneracies introduced by rounding intersections, the vertical clearance filter with a low height clearance threshold and a rounding filter which forcibly demotes the numerical accuracy for intersections calculated by the vertical clearance filter to the machine accuracy level. The post-processing block is populated by a degeneracy filter to remove degeneracies introduced by the rounding filter, a duplication filter with a low threshold to restore mesh connectivity, an area filter to remove small areas, a simplification filter to reduce mesh complexity and the layer subdivision filter to construct a multi-layered environment. The definition file of the parallel pipeline using the sorted recursive strategy looks as follows:

```
DegeneracyFilter
SlopeFilter 60
DuplicationFilter 0.005
SmartRecursiveStrategy 50 3 true
DegeneracyFilter
VerticalClearanceFilter 3
RoundingFilter true
Collapse
DegeneracyFilter
DuplicationFilter 0.005
AreaFilter 500
SimplificationFilter true 0.000001
PEELFilter
```

Note: The parallel pipeline experiments are ran using $n = 32$ threads, experiments with other numbers of threads are performed in Section 5.6.

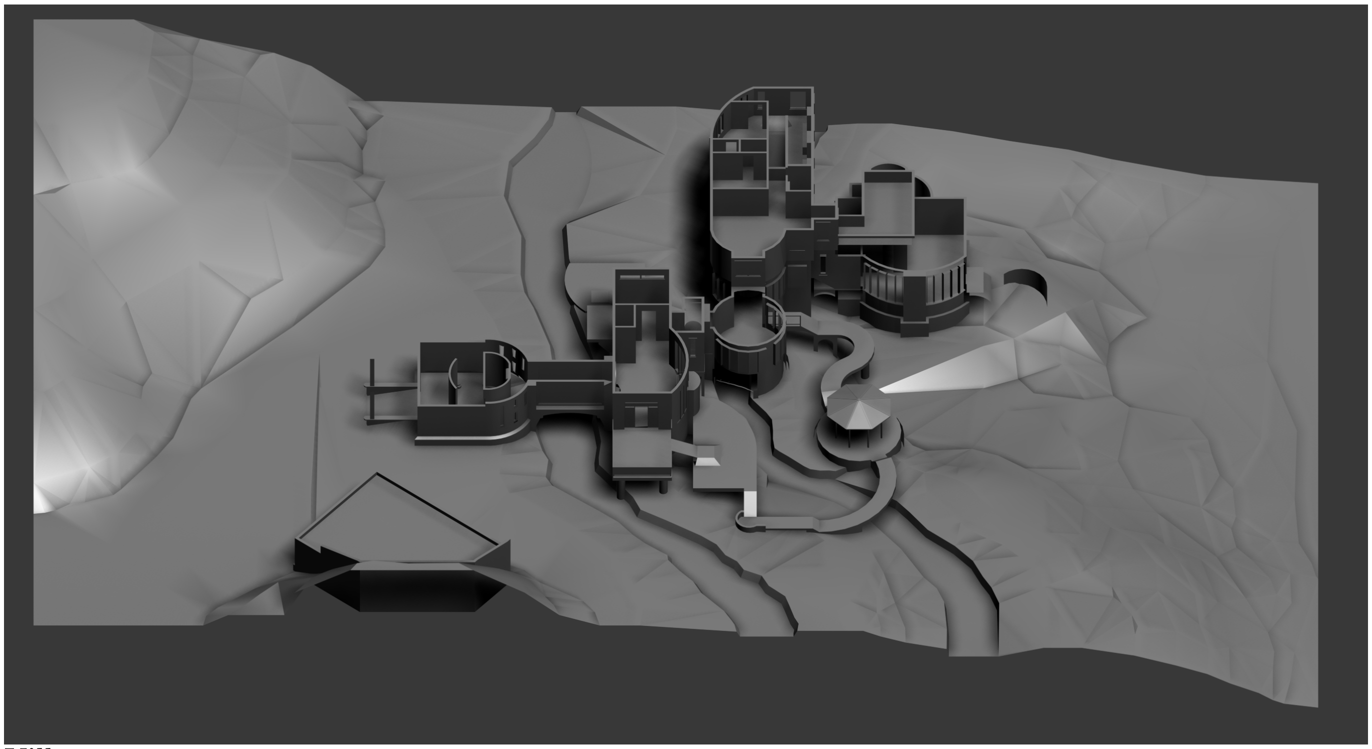
5.4.1 Environments

A set of real-world environments is selected to test the parallel pipeline on. Smaller environments are chosen to highlight the overhead of partitioning and running computations on CORE::Expr graphs in parallel. Large environments are chosen to highlight the strengths of the parallel pipeline and to show better execution times and noticeable speed-ups. An artificial environment is introduced to highlight a structural flaw in the partitioning methods described in Section 4.3.



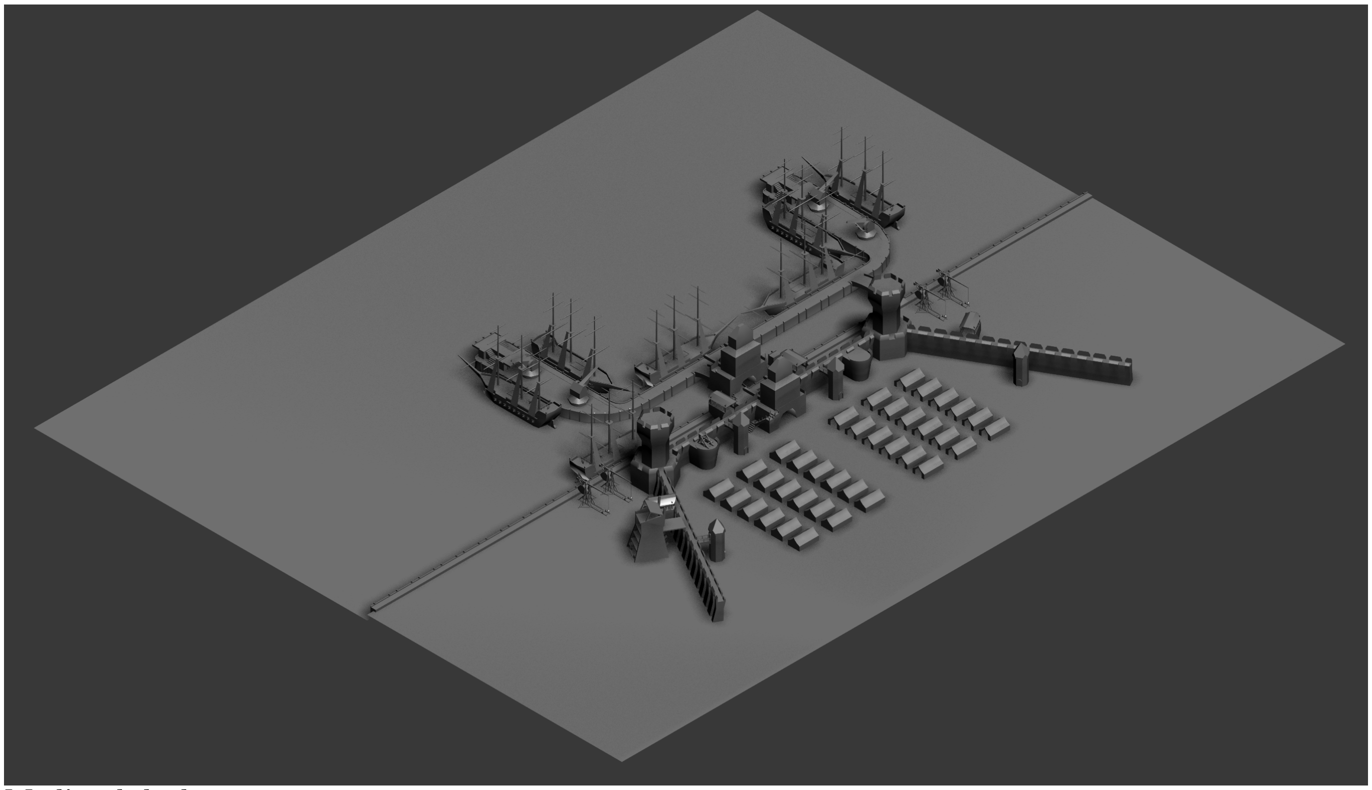
House

Number of polygons	820
Number of vertices	446
Description	Small house containing a number of artifacts relevant to geometrical processing, such as ramps, overhanging polygons and disjoint polygon groups.



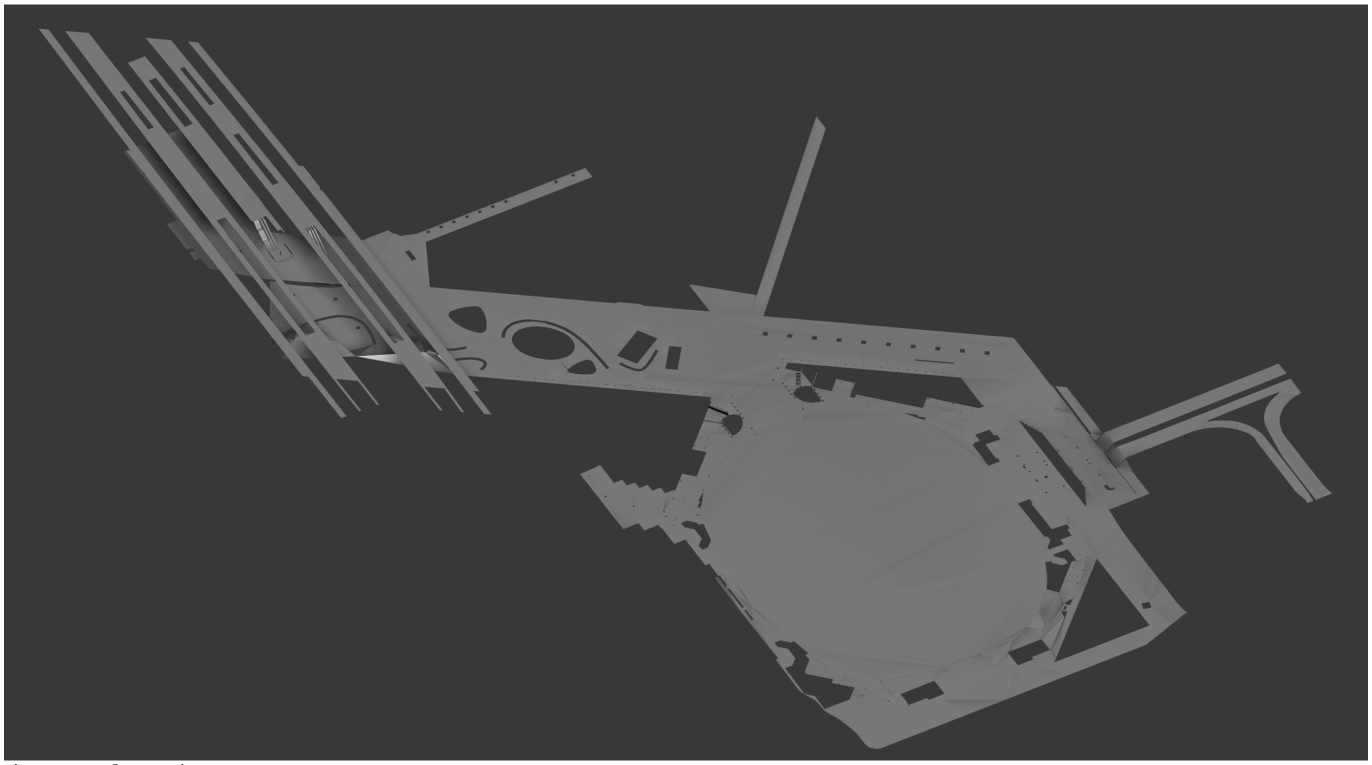
Villa

Number of polygons	30363
Number of vertices	19486
Description	Complex house environment where the ground plane is relatively coarse height-mapped terrain with a large number of mesh errors, such as intersecting polygons, multiply defined polygons, multiply defined vertices and polygons with a high amount of extraneous intersections.

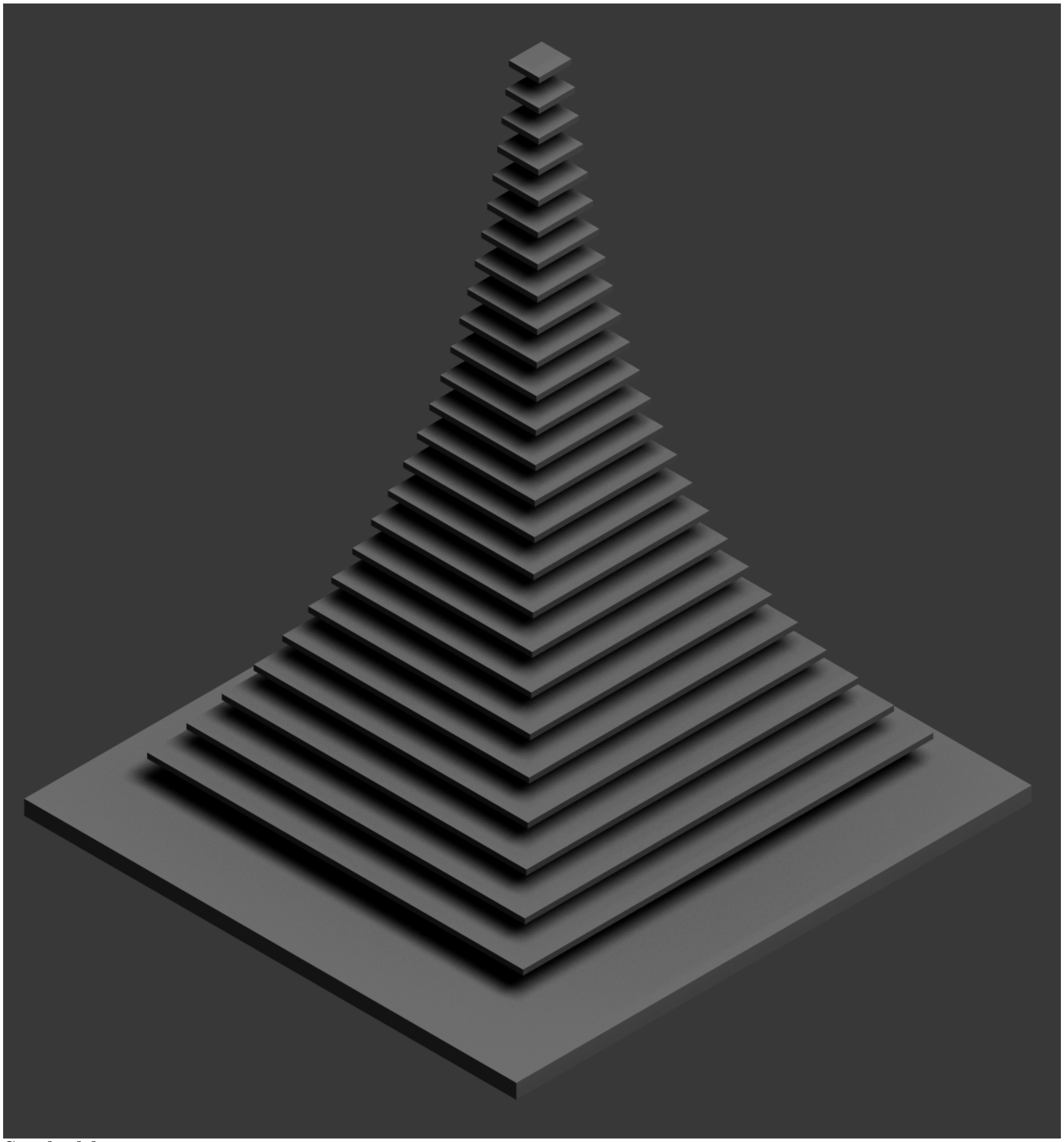


Medieval docks

Number of polygons	59418
Number of vertices	35881
Description	Shipyard environment with two ground planes, one representing water and the other representing walkable ground. Contains a high number of obstacles and polygons which create intersection points using the vertical clearance filter.



Amsterdam Arena	
Number of polygons	19829
Number of vertices	24599
Description	Real-world environment of Amsterdam Arena with a large ground plane, a large number of ramps and areas with a high concentration of obstacles.



Stacked box tower

Number of polygons	440
Number of vertices	272
Description	Artificial environment built to highlight the weaknesses of only considering the environment in the xy-plane during partitioning.

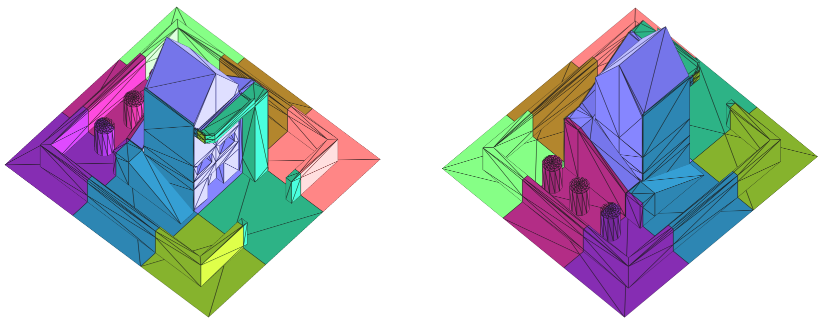
5.4.2 Results and discussion

The first benchmark is performed on the house environment. This environment has been included from previous research because of its feature set (ramps, different layers, obstacles, vertical clearance) and its low polygon count allowing for rapid debugging of the entire pipeline. The point of including a low polygon environment such as this is to highlight the effect the parallel pipeline has on smaller environments.

House ($t_{sequential} = 9,192ms$)	Grid strategy (2x2)	Grid strategy (3x3)	Grid strategy (4x4)	Recursive strategy (max. depth = 2)	Recursive strategy (max. depth = 3)
$t_{total,parallel}$	$3,301ms$	$5,229ms$	$11,225ms$	$6,463ms$	$18,981ms$
s	$2.7846107\times$	$1.757889\times$	$0.818886414\times$	$1.196\times$	$0.484\times$
$t_{preprocessing}$	$43ms$	$47ms$	$116ms$	$37ms$	$21ms$
$t_{partitioning}$	$364ms$	$980ms$	$236ms$	$1,083ms$	$4,338ms$
$t_{parallel}$	$959ms$	$2,217ms$	$9,072ms$	$3,322ms$	$11,755ms$
$t_{merging}$	$15ms$	$14ms$	$8ms$	$12ms$	$34ms$
$t_{postprocessing}$	$320ms$ $450ms$	$306ms$	$318ms$	$1,284ms$	
$t_{overhead}$	$1,600ms$	$1,521ms$	$1,487ms$	$1,691ms$	$1,549ms$

House ($t_{sequential} = 9,192ms$)	Sort. recursive strategy (max. depth = 2)	Sort. recursive strategy (max. depth = 3)	Sort. recursive strategy (max. depth = 4)
$t_{total,parallel}$	$3,498ms$	$4,732ms$	$6,773ms$
s	$2.6277873\times$	$1.942519\times$	$1.357153\times$
$t_{preprocessing}$	$62ms$	$41ms$	$50ms$
$t_{partitioning}$	$565ms$	$1,000ms$	$1,518ms$
$t_{parallel}$	$1,109ms$	$1,880ms$	$3,123ms$
$t_{merging}$	$15ms$	$13ms$	$33ms$
$t_{postprocessing}$	$276ms$	$325ms$	$442ms$
$t_{overhead}$	$1,471ms$	$1,473ms$	$1,607ms$

As clearly visible in the results, partitioning the environment has overhead associated to it that makes the parallel pipeline (in some cases) slower than a sequential pipeline with similar settings. The overhead of partitioning the environment, creating threads, running filters on each thread and merging the partitions back together can be too high. Specifically, in the case of the recursive strategy with depth 3 and the 4×4 grid strategy the performance suffers greatly due to these effects. Creating too many partitions proves to be not worth it here because of the overhead.

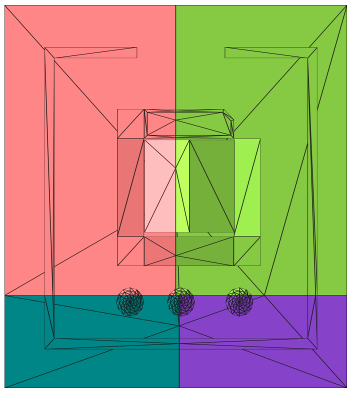

(a) The center partition contains almost the entire house building.

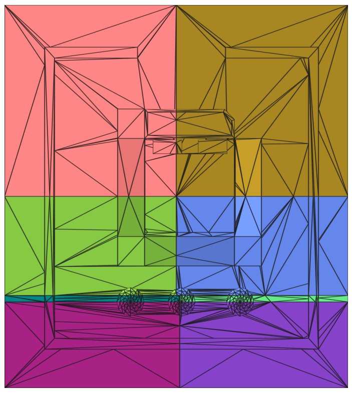
(b) Two partitions contains almost every obstacle polygon in the dense cylindrically-shaped objects.

Figure 25: House environment partitioned using the 3×3 grid partioning strategy.

The only reason for potentially low performance is not overhead, though. In the 3×3 grid strategy most of the obstacle polygons are collected into two partitions, as demonstrated in Figure 25. All of the "work" is concentrated into these two polygons. The other partitions have a significantly lower workload and a poor load balance overall because of this. Due to the poor load balance, the parallel pipeline stalls at the two partitions with the most work, resulting in a longer execution time.

In the 4×4 grid strategy the partitioning method puts a partitioning plane in such a way that one of the coarse cylinders is subdivided into two parts. When the vertical clearance filter is applied on this obstacle object, deep expression graphs will be generated with the (also) subdivided ground plane and the pipeline will stall on this partition. In the recursive strategy, this also occurs at depths 2 and 3, but the sorted recursive strategy avoids this by placing partitioning planes more intelligently, as visible in Figure 26.


(a) Sorted recursive strategy with (maximally) 2^2 partitions. The partitioning planes are placed in such a way that the coarse cylinders in the south are roughly divided equally over all partitions.


(b) Sorted recursive strategy with (maximally) 2^3 partitions. Observe the thin partitions created to partition the cylinder objects even further. Most polygons in this mesh are in that specific location, hence, the placement of the partitioning planes.

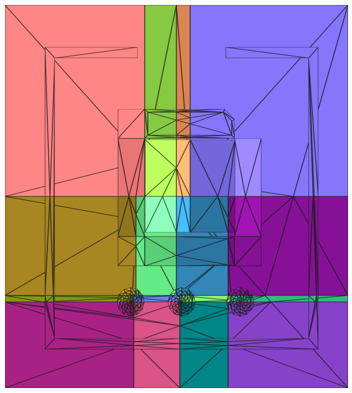
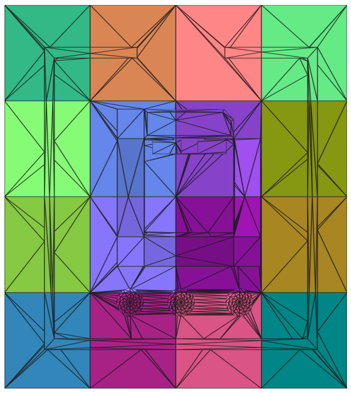
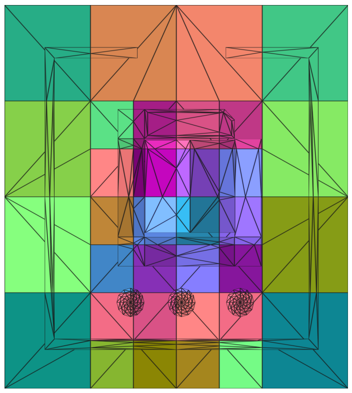

(c) Sorted recursive strategy with (maximally) 2^4 partitions. The stop condition based on the number of polygons in a partition is reached and partitioning is stopped prematurely.

Figure 26: House environment partitioned using the sorted recursive strategy. A top-down view is taken to able to see partitions properly.

The effects of the recursive strategy become clearer in Figure 27. Most of the detail in this environment is not well aligned within the cells created by the recursive strategy's partitioning planes. The recursive strategy then continues to go into recursion until the desired number of polygons (or the maximum depth) is reached. All three variants in the figure contain partitions which are practically empty and do not contain any obstacle polygons to process. Once again, all the detail is focused into the few partitions containing the cylindrical objects and the main building.


(a) Recursive strategy with a quad tree structure of (maximally) depth 2, or 4^2 partitions.


(b) Recursive strategy with a quad tree structure of (maximally) depth 3, or 4^3 partitions.

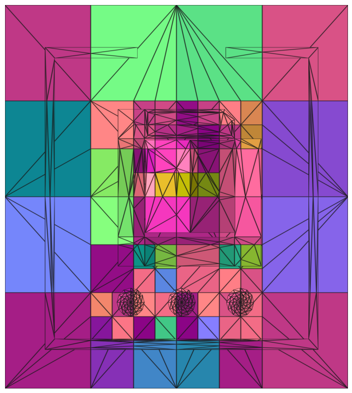

(c) Recursive strategy with a quad tree structure of (maximally) depth 4, or 4^4 partitions.

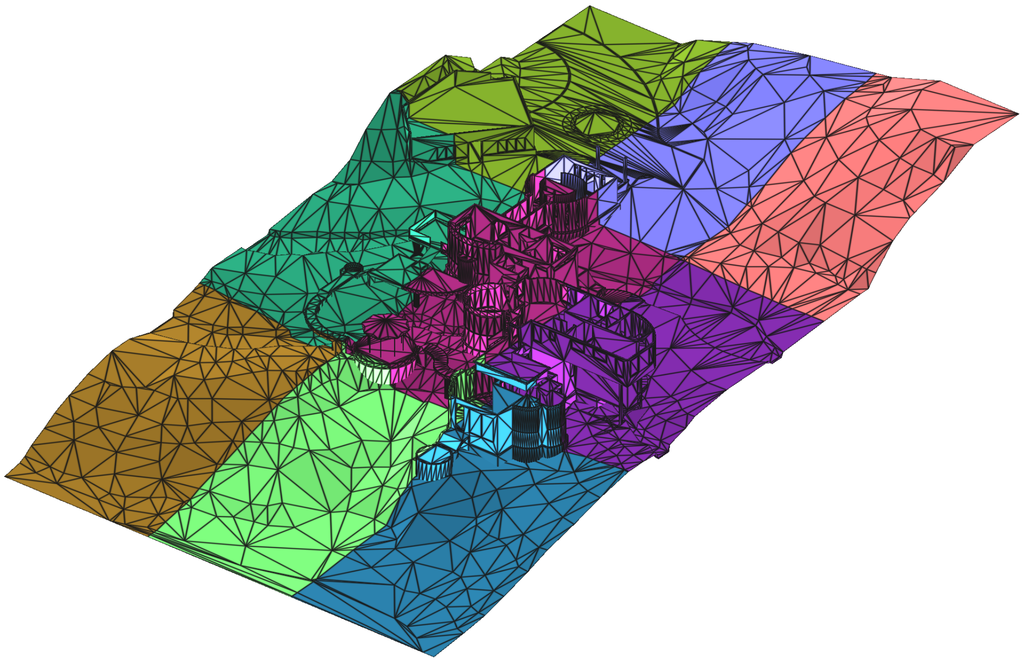
Figure 27: House environment partitioned using the recursive strategy. A top-down view is taken to able to see partitions properly.

The second benchmark is performed on the villa environment. The villa environment has a significantly increased number of polygons compared to the house environment and contains a large number of obstacle polygons which, in turn, will generate a large number of intersection points in the vertical clearance filter and the partitioning strategy. The villa environment also has multiple layers with obstacles, which can produce suboptimal partitions, since not all obstacles lie in one plane aligned to the xy-plane.

Villa ($t_{sequential} = 900,000ms+$)	Grid strategy (2x2)	Grid strategy (3x3)	Grid strategy (4x4)	Recursive strategy (max. depth = 2)	Recursive strategy (max. depth = 3)
$t_{total,parallel}$	$97,237ms$	$124,510ms$	$105,588ms$	$107,109ms$	$129,775ms$
s	$9.255736\times$	$7.228335\times$	$8.523696\times$	$8.402655\times$	$6.93508\times$
$t_{preprocessing}$	$220ms$	$206ms$	$243ms$	$219ms$	$252ms$
$t_{partitioning}$	$1,589ms$	$3,245ms$	$4,302ms$	$4,991ms$	$16,555ms$
$t_{parallel}$	$42,195ms$	$63,010ms$	$40,702ms$	$41,378ms$	$35,253ms$
$t_{merging}$	$455ms$	$797ms$	$806ms$	$1,008ms$	$3,258ms$
$t_{postprocessing}$	$50,522ms$	$55,000ms$	$56,995ms$	$57,236ms$	$72,120ms$
$t_{overhead}$	$2,256ms$	$2,252ms$	$2,540ms$	$2,277ms$	$2,337ms$

Villa ($t_{sequential} = 900,000ms+$)	Sorted recursive strategy (max. depth = 2)	Sorted recursive strategy (max. depth = 3)	Sorted recursive strategy (max. depth = 4)
$t_{total,parallel}$	$99,332ms$	$101,081ms$	$133,770ms$
s	$9.060524302\times$	$8.90375\times$	$6.727966\times$
$t_{preprocessing}$	$243ms$	$212ms$	$240ms$
$t_{partitioning}$	$3,417ms$	$5,074ms$	$8,823ms$
$t_{parallel}$	$38,995ms$	$36,621ms$	$53,334ms$
$t_{merging}$	$658ms$	$2,191ms$	$3,490ms$
$t_{postprocessing}$	$53,772ms$	$54,781ms$	$65,431ms$
$t_{overhead}$	$2,247ms$	$2,202ms$	$2,452ms$

(The villa environment in the sequential, original pipeline caused the test machine to memory overflow and could not continue working beyond the $900,000ms$ mark, so $t_{sequential}$ was taken to be $900,000ms$. In reality, the speed-up factor would be more given more physical memory, but the minimum was taken.)


(a) Orthographic projection.

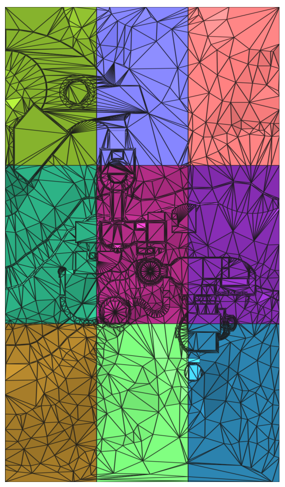
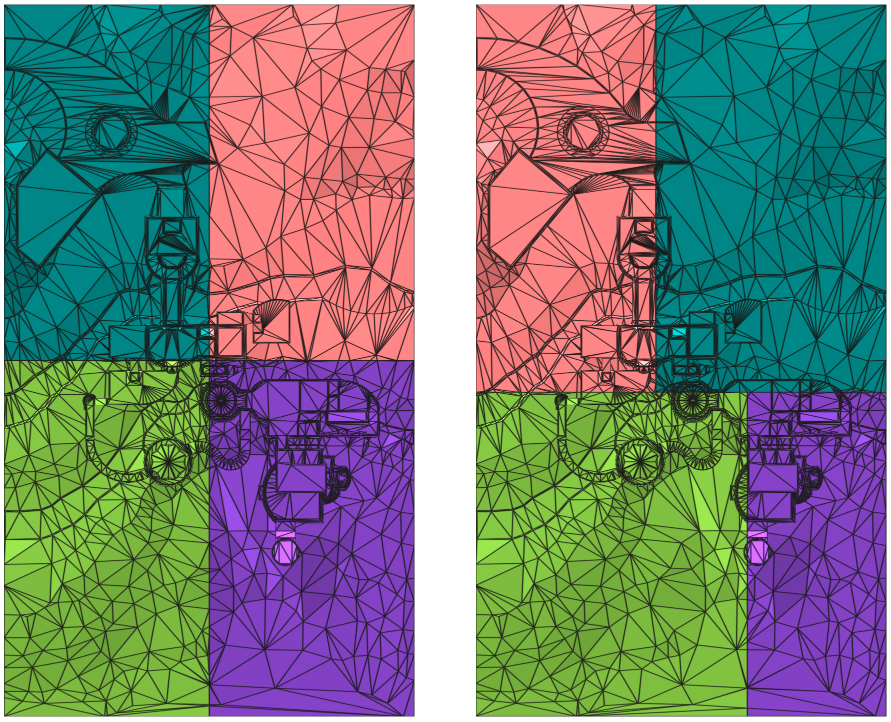

(b) Top-down view.

Figure 28: Villa environment partitioned using the grid strategy into 3×3 partitions. Almost all obstacle polygons reside in the middle partition and the middle-right partition.

The villa environment has similar problems as the house environment but on a larger scale. Once again the grid strategy shows its naivety in the 3×3-configuration, where all detail lies in the center partition, as shown in Figure 28. Because nearly all detail in the environment is in one partition, we get similar performance compared to $t_{sequential}$. The performance in this case is slightly worse because the overhead cost of running the pipeline in parallel adds up. The 2×2-configuration accidentally picks a decent set of partitioning planes because of the layout of the environment similar to the partitions generated by the sorted recursive partitioning strategy. In Figure 29 the similarities between these two configurations are displayed.


(a) Grid strategy with (maximally) 2×2 partitions.

(b) Sorted recursive strategy with (maximally) 2^2 partitions.

Figure 29: Villa environment partitioned using the grid strategy and recursive strategy into 4 partitions each. A top-down view is taken to able to see partitions properly.

In this larger environment the overhead of cutting the environment (during partitioning) and stitching the partitions together (during merging) plays a bigger role. The more partitions are created, the higher $t_{partitioning}$ and $t_{merging}$ become. Because the villa environment has a large number of polygons, the cutting algorithm creates, in turn, a large number of intersections. Because there are more intersection points to process and polygons to retriangulate, $t_{partitioning}$ and $t_{merging}$ become higher.

An additional observation we can make from the villa and house environment is that the sorted recursive strategy roughly offers the same performance as the grid strategy. The recursive strategy generally performs worse due to its higher $t_{partitioning}$ and $t_{merging}$. The poor quality of its partitions in Figure 27 and Figure 30 in combinations with these values prove the approach generates many 'empty' partitions.

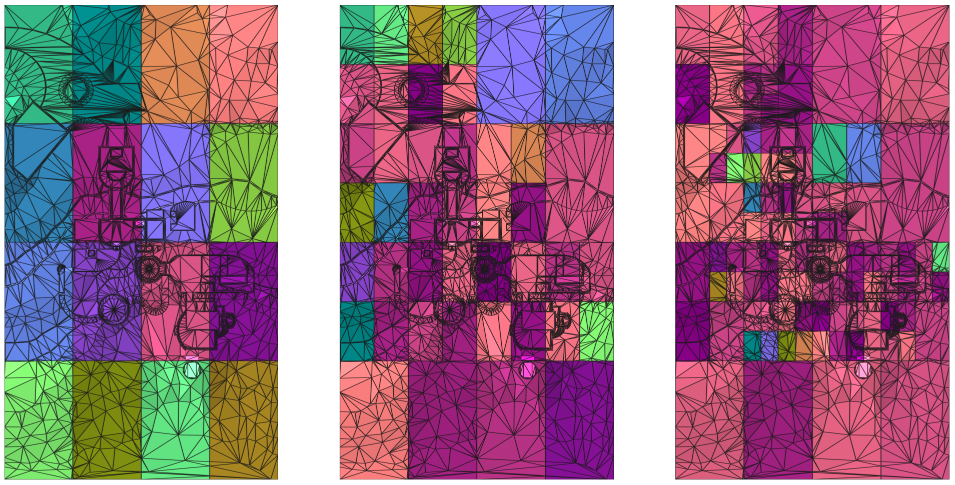

(a) Recursive strategy with (maximally) 4^2 partitions.
(b) Recursive strategy with (maximally) 4^3 partitions.
(c) Recursive strategy with (maximally) 4^4 partitions.

Figure 30: Villa environment partitioned using the recursive strategy. A top-down view is taken to able to see partitions properly.

The Amsterdam Arena environment has a polygon count in between the house and villa environment, but has most of its obstacle polygons focused in one place. Using this environment the adaptability of a partitioning strategy on poorly distributed geometry is tested. Expected is that the grid strategy will cluster most obstacle polygons into one partition, resulting in a very skewed partitioning process where only one partition contains all obstacle polygons. The sorted recursive strategy will scale better on this environment, because it guarantees an equal number of polygons between partitioning splits.

Amsterdam Arena ($t_{sequential} = 1,800,000ms+$)	Grid strategy (3×3)	Grid strategy (4×4)	Recursive strategy (max. depth = 3)	Sort. recursive strategy (max. depth = 3)	Sort. recursive strategy (max. depth = 4)
$t_{total,parallel}$	$1,143,833ms$	$824,745ms$	$190,254ms$	$121,877ms$	$129,752ms$
s	$1.5736563\times$	$2.182493\times$	$9.461036\times$	$14.76899\times$	$13.87261853\times$
$t_{preprocessing}$	$2,479ms$	$2,556ms$	$167ms$	$169ms$	$217ms$
$t_{partitioning}$	$6,476ms$	$10,051ms$	$70,625ms$	$5,002ms$	$9,027ms$
$t_{parallel}$	$1,017,566ms$	$706,710ms$	$58,762ms$	$66,262ms$	$62,496ms$
$t_{merging}$	$67,801ms$	$54,141ms$	$984ms$	$526ms$	$961ms$
$t_{postprocessing}$	$47,921ms$	$49,735ms$	$57,602ms$	$47,640ms$	$54,965ms$
$t_{overhead}$	$1,590ms$	$1,552ms$	$2,114ms$	$2,278ms$	$2,086ms$

Most overlapping polygons which need to be processed by the vertical clearance filter in the processing part of the parallel pipeline ($t_{parallel}$) are in one problem area, shown in Figure 31. The sorted recursive strategy isolates this area and partition it multiple times in order to split this dense area of obstacles into multiple partitions, but the recursive- and grid strategy miss this completely. In Figure 32, the resolved problem area is shown.

(The Amsterdam Arena environment in the sequential, original pipeline caused the test machine to memory overflow and could not continue working beyond the $1,800,000ms$ mark, so $t_{sequential}$ was taken to be $1,800,000ms$. In reality, the speed-up factor would be more given more physical memory, but the minimum was taken.)

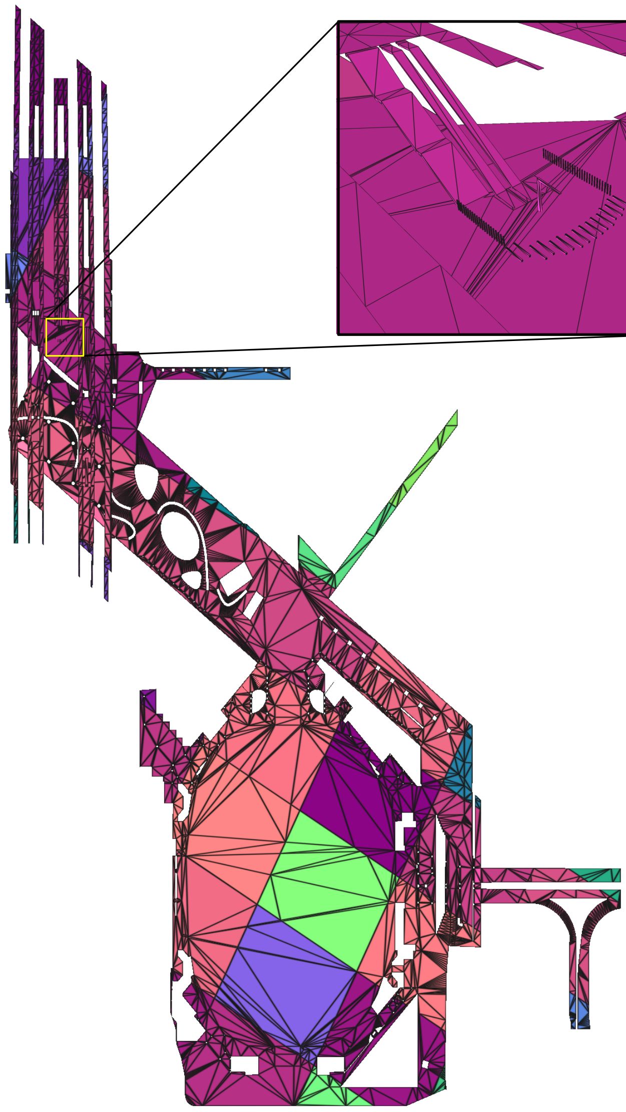


Figure 31: Amsterdam Arena environment partitioned using the recursive method with a maximum of 2^3 partitions. In this configuration, all the obstacle polygons appear in one partition, resulting in a higher execution time compared to the sorted recursive strategy.

A final performance issue arises in the 2^4 sorted recursive configuration. In comparison to the 2^3 sorted recursive configuration, $t_{postprocessing}$ takes roughly $10s$ longer. On the other hand the time spent processing the environment in parallel $t_{parallel}$ is lower. The increase in time spent in the post-processing phase of the pipeline is because of the simplification filter. This filter takes longer to complete, because the polygon density of the environment is artificially increased by the partitioning strategy's cutting algorithm. The more partitioning planes are used to cut into the environment before partitioning, the longer the simplification filter will take to re-triangulate the environment. Ultimately this cost outweighs the performance gained in the parallel segment.

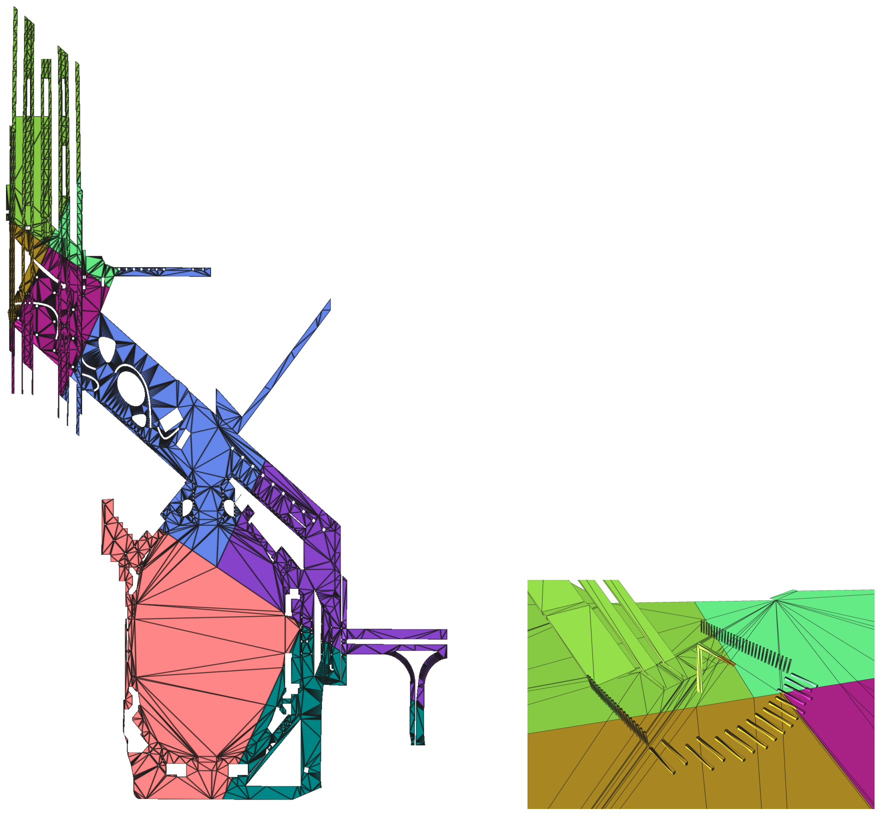

(a) Sorted recursive strategy with a maximum of 2^3 partitions. (b) Resolved problem area.

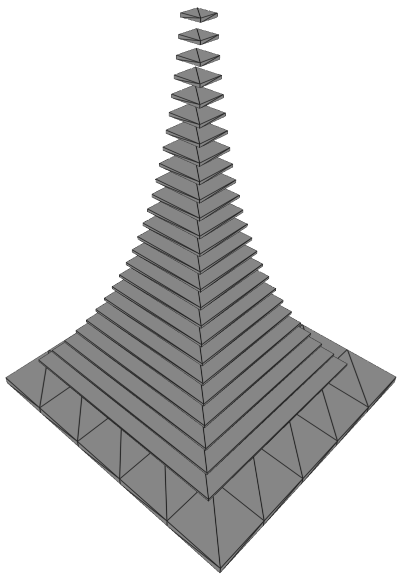
Figure 32: Amsterdam Arena partitioned by the sorted recursive strategy. The problem area, unlike the recursive strategy, is handled by splitting up the obstacles over multiple partitions.

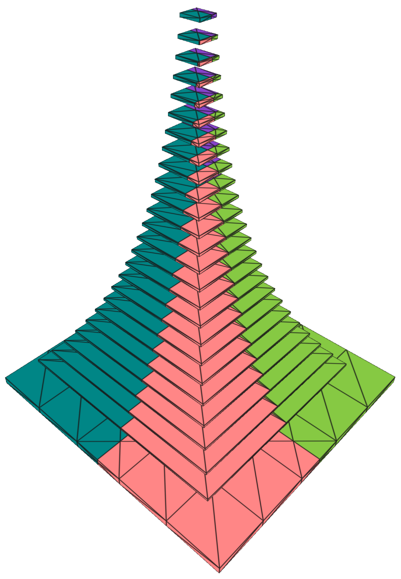
an artificial environment is introduced next. The stacked tower environment is used to highlight an important failure case of the partitioning methods introduced in Section 4.3. When an environment with a large number of stacked walkable areas is introduced with a high height clearance parameter a cascading effect is created. Every walkable area a_1 above another area a_2 has to be intersected with each other. In this environment there are a set of areas $a_0, \cdots, a_n$ where a_0 is the top-most area (box) and a_n is the lowest area (box). Area a_0 has no obstacles above it and does not have to be processed by the vertical clearance filter, but every other box with index i has to process the boxes $a_0, \cdots, a_{i-1}$.

Because the partitioning strategies described in Section 4.3 only place partitioning planes based on geometry from the perspective of the xy-plane, the z-aspect of this tower is lost. The partitions resulting from these partitioning strategies, especially in the grid- and recursive strategies, are so poor that the performance significantly goes down in comparison to the sequential pipeline. After partitioning the sub-environments in the partitions mirror the situation in the original environment; a set of n stacked boxes in the z-axis. By applying the cutting algorithm here, the number of polygons multiplies by a factor of 2 every time a partitioning plane is introduced. Every partition then has to do equal work compared to the sequential pipeline.

Stacked tower ($t_{sequential} = 2,256ms$)	Grid strategy (2x1)	Grid strategy (2x2)
$t_{total,parallel}$	7,962ms	244,056ms
s	0.2833459×	0.00924378×
$t_{preprocessing}$	$38ms$	26ms
$t_{partitioning}$	$316ms$	1081ms
$t_{parallel}$	$5,942ms$	241,153ms
$t_{merging}$	$2ms$	46ms
$t_{postprocessing}$	$142ms$	130ms
$t_{overhead}$	$1522ms$	1620ms

As expected the stacked tower performs abysmally compared to the sequential pipeline, even with a $2x1$ grid partition. The xy-based partitioning methods fail to increase the performance in any particular way and highlight the shortcomings in vertically stacked environments. Crucial real-world environments exist where this partitioning method would fail, such as apartment buildings, houses with multiple stories and skyscrapers. To properly divide these environments over a set of threads, another partitioning method would have to be introduced taking into account geometry in the z-axis and are discussed in Section 6.1.


(a) Original environment.


(b) Environment cut by two partitioning planes. The number of polygons in this environment is 4× higher than the original environment.

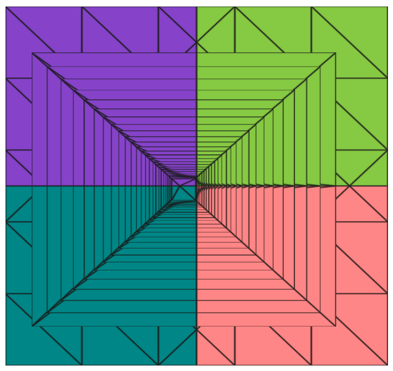
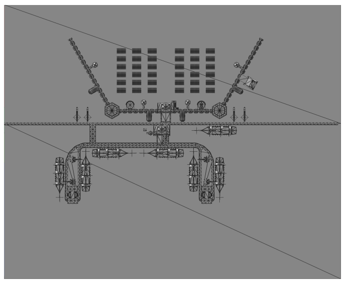

(c) Top-down view.

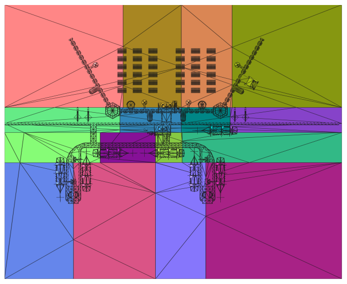
Figure 33: Artificial tower environment partitioned by the 2×2 grid strategy.

Finally, the docks environment is tested. This environment fails to compute in the sequential pipeline. The vertical clearance filter operation stalls in such a matter that no useful running time can be gathered. In the parallel pipeline, however, the multi-layered environment is calculated correctly. Because we do not have any indication of what the sequential timings are, we cannot calculate the speed-up or compare it to the parallel speed. We can, however, say that the parallel pipeline allows for the processing of far more complex and larger environments compared to the sequential pipeline. With the example of the docks environment we see that environments previously unprocessable by the sequential pipeline are now possible to be processed in the parallel pipeline. In the docks environment specifically, $t_{postprocessing}$ is significantly higher than the time spent in parallel $t_{parallel}$. The time spent post-processing is higher because the cost to re-triangulate in the re-triangulation filter exceeds the cost to calculate vertical intersections in the vertical clearance filter. The vertical clearance filter calculates so many new intersection points with the ground plane that the re-triangulation filter stalls on this newly generated complexer (and coarser) ground plane.

Docks ($t_{sequential} = -$)	Sorted recursive strategy (max. depth = 4)
$t_{total,parallel}$	465,772ms
s	$-$
$t_{preprocessing}$	$305ms$
$t_{partitioning}$	$8,654ms$
$t_{parallel}$	$78,175ms$
$t_{merging}$	$5,467ms$
$t_{postprocessing}$	$370,937ms$
$t_{overhead}$	$2,234ms$

Table 4: Resulting timings of the docks environment. **Note:** only one variation of the sorted recursive strategy was tested to indicate that the newly proposed parallel pipeline could process this complex environment.


(a) Original docks environment.


(b) Docks environment partitioned by the sorted recursive strategy with a maximum depth of 4, resulting in maximally 2^4 partitions.

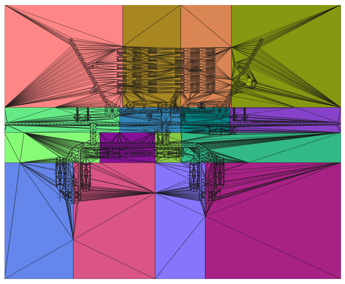

(c) Top-down view.

Figure 34: Docks after vertical clearance filter is applied. The ground plane is extremely coarse, leading to stalling when re-triangulating the environment.

5.5 Mesh correctness testing

In the proposed parallel pipeline, coordinates are rounded in the rounding filter, partitioning strategy and during communication. The sequential pipeline by Polak [Pol16], however, only rounds coordinates when outputting the resulting environment. In this section, we evaluate the correctness and quality of the output walkable environment of our parallel pipeline. By estimating the distance between discrete 3D surfaces represented by triangular 3D meshes, the difference between the output of the sequential pipeline and the output of the parallel pipeline is found. By attaining a low difference, we can show that the output of the parallel pipeline does not significantly differ from the output of the sequential pipeline. One of the simplest approaches in order to provide an *MSE*-like measurement for 3D models is to use the *Hausdorff distance* [ASE02].

We compare the *walkable environment* obtained from the sequential pipeline and the proposed parallel pipeline with similar settings. The goal is to determine the distance between the output of the purely exact sequential pipeline proposed by Polak [Pol16] and our proposed parallel pipeline. In the parallel pipeline, geometric operations are still guaranteed to be exact even after rounding because of *CORE::Expr* automatic promotion. After rounding, coordinates are then of machine accuracy, but subsequent geometric operations can re-promote the coordinates to a higher level of accuracy. With these higher levels of accuracy, geometric operations are guaranteed to be exact until they are forcibly demoted again by a rounding operation. The act of rounding coordinates like this introduces inaccuracies in the output environment (as described in Section 5.1.1), which requires us to test against the original sequential pipeline for validity.

The definition of the Hausdorff distance, as defined by Aspert [ASE02] is as follows. The distance between a point p, belonging to a surface S, and a surface S' is defined to be

$$d(p, S') = \min_{p \in S'} ||p - p'||^2,$$

while the *Hausdorff distance* between two surfaces S and S' is defined to be

$$d(S, S') = \max_{p \in S} d(p, S').$$

The *Hausdorff distance* is calculated using the MeshLab mesh-processing package [Cig+08]. The MeshLab algorithm samples points on the output environment of the sequential pipeline and calculating the *Hausdorff distance* to the output environment of the parallel pipeline. The algorithm then aggregates these per-point distances and calculates the maximal distance, minimal distance, mean distance and the RME of all these samples. By comparing these values we see how large the difference is between the parallel pipeline and the sequential pipeline. For the Hausdorff distance, a lower value indicates a less differences between meshes, with a distance of 0 indicating identical meshes.

Note that it is possible to always gain a Hausdorff distance of $d = 0$ by either only using 1 (one) sampling point or by picking favourable sampling points. This is avoided by using the default number of sampling points in MeshLab, which, in our case, is equal to the amount of vertices on the sequential pipeline's output environment. The sampling points are then placed on the vertices of the sequential pipeline's output environment. When $d = 0$ we then know that the meshes are exactly the same.

A secondary metric of comparison is taken from the comparative study between navigation meshes by Wouter van Toll [Van+16]. The metric used is the *walkable space covered* metric, which measures the area of walkable space that is covered by the navigation mesh divided by the real area of the walkable space [Van+16]. The sequential environment's area is taken as the real area of the walkable space, because the output of the sequential pipeline is the ground truth. The area of walkable space that is covered by the navigation mesh is the area of the parallel environment. The area of the sequential and parallel environment are calculated using MeshLab's *Compute Geometric Measures* filter. For the walkable space covered metric a higher value is better, with 1 being a perfect score [Van+16].

For sake of simplicity we refer to the *sequential environment* as the walkable environment generated by a sequential pipeline. The *parallel environment* is referred to as the walkable environment generated by the proposed parallel pipeline. The parallel pipeline in this section uses the sorted recursive partitioning strategy with a stop condition at 4 iterations.

The experiments are ran in the following way:

1. A sequential pipeline and a parallel pipeline are ran on a polygonal environment and their resulting environments are stored.

2. The sequential environment and parallel environment are loaded into MeshLab and the Hausdorff sampling algorithm is applied using the default number of samples.
3. MeshLab outputs the maximum distance, minimum distance, mean distance and the mean relative error (RME).
4. The area of the sequential and parallel environment are calculated using MeshLab's *Compute Geometric Measures* filter.

5.5.1 Results and discussion

	House	**Amsterdam Arena**	**Villa**
d_{min}	0	0	0
d_{max}	0.000041	0.188383	15.668283
d_{mean}	0.000001	0.002652	0.004275
d_{rms}	0.000005	0.012911	0.214389

Table 5: Hausdorff distance between parallel pipeline's output environment and sequential pipeline's output environment.

In the house environment, the maximum distance between the parallel walkable environment and the sequential walkable environment is negligible at 0.000041. The scale of the house environment is 1 : 1 units to metres. By using this information we can infer that the maximal distance between these environments is $41\mu m$.

In the more realistically sized environments we see a worse trend appear. The mean Hausdorff distance of the sampled points goes up by a substantial amount. Intuitively, this makes sense because the environment has substantially more obstacle polygons compared to the house environment.

The most common cause of high d_{max} values is the rounding of intersection points generated by the vertical clearance filter. The vertical clearance filter cuts obstacle polygons into the ground plane(s), resulting in a large number of new vertices. Because of demoting the cut vertices to machine accuracy larger errors will occur in the accuracy of their coordinates compared to the sequential pipeline. In Figure 35 and Figure 36, these types of outliers are highlighted in a top-down view.

In the Amsterdam Arena environment, the mean error is 0.002652 units, which means the parallel environment, on average, is within 0.002652 units of the sequential environment. The scale of the Amsterdam Arena is 1 : 2 units to metres, which results in this being a mean distance of $1.326mm$.

The villa environment, however, presents us with quite a big outlier with a maximal distance of 15.668283 units. However, this environment has a different unit-to-metre ratio from the other two environments. A door post in this environment (roughly the length of an agent) has a height of about 85 units, as measured in MeshLab. Assuming the length of an agent is 2 metres, we find that the scale of this environment is 1 unit to $\frac{2}{85} = 0.0235$ metres. In Table 6 all data is presented in metres.

	House	**Amsterdam Arena**	**Villa**
d_{min}	0	0	0
d_{max}	0.000041	0.0941915	0.368205
d_{mean}	0.000001	0.001326	0.001574
d_{rms}	0.000005	0.006456	0.000337

Table 6: Hausdorff distance between parallel pipeline's output environment and sequential pipeline's output environment in **metres**.

Next, we explore the high maximal distance in the Amsterdam arena environment and the villa environment. The high maximal distance in the Amsterdam arena environment is an outlier which is produced by a polygon which was re-triangulated to be more flat than its counterpart in the sequential environment. In the parallel pipeline, the approximated simplification filter is used, as proposed in Section 5.1.2, while the sequential pipeline uses the exact implementation, proposed by [Pol16]. The difference in exactness causes the approximated simplification filter to remove edges where the original would not. The result of edge collapsing is that collapsed edges can lead to flat polygons in the parallel environment which were dented or raised in the sequential environment. The high d_{max} values come from the relatively large distance between the dented polygons in the sequential environment and their flattened counterparts in the parallel environment.

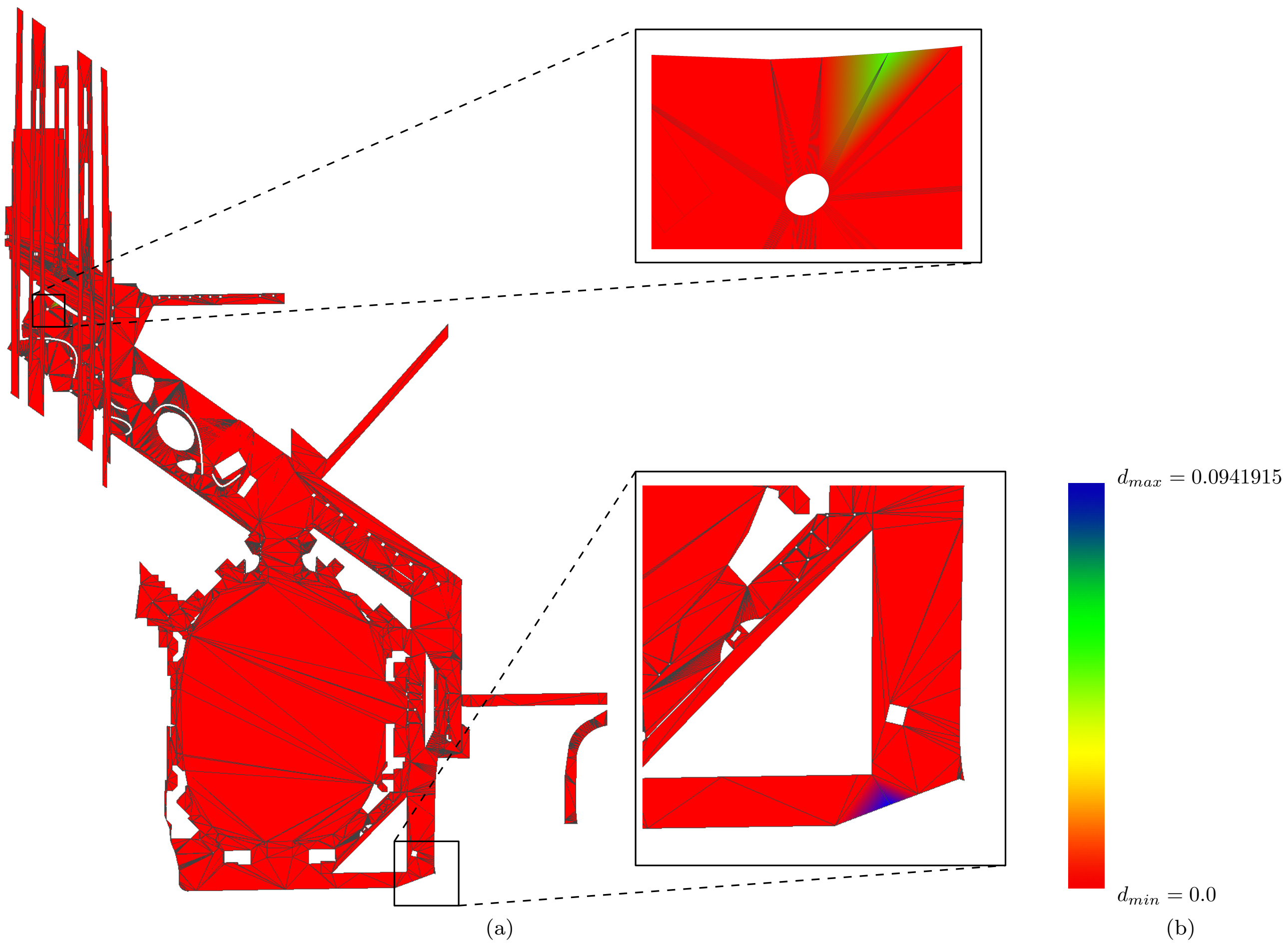


Figure 35: Amsterdam arena Hausdorff distance visualization. The Hausdorff distance is visualized on a red-green-blue scale, where red is 0 distance, blue is d_{max} distance and green lies inbetween.

In Figure 36, a small part of the Amsterdam arena environment is displayed. Aggressive simplification causes the cobweb-like topology in the top-right to be simplified differently in the parallel version leading to a higher maximum Hausdorff distance.

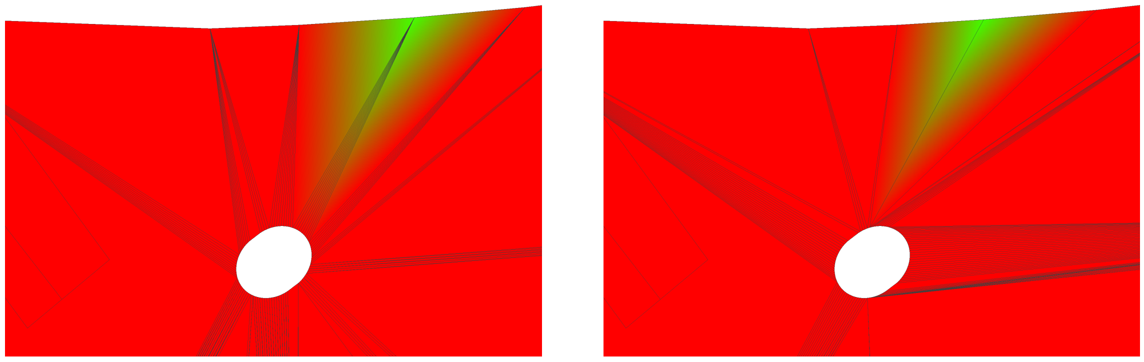

(a) Sequential walkable environment
(b) Parallel walkable environment

Figure 36: Amsterdam arena Hausdorff distance visualization. The Hausdorff distance is visualized on a red-green-blue scale, where red is 0 distance, blue is d_{max} distance and green lies inbetween.

In Figure 37, another part of the Amsterdam arena environment is displayed, this time showing d_{max}. An edge present in the sequential walkable environment is not present in the parallel walkable environment. Again, because of the aggressive approximated simplification filter this edge is simplified away which leads to a high(er) Hausdorff distance.

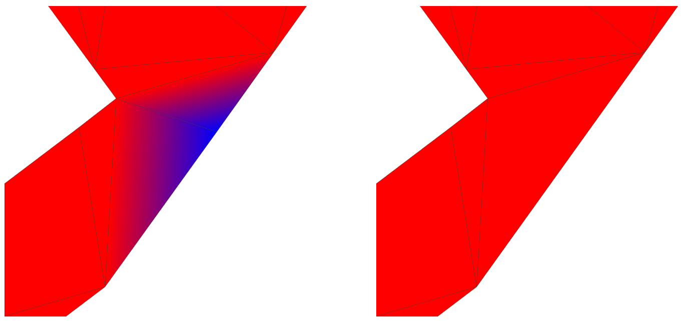

(a) Sequential walkable environment
(b) Parallel walkable environment

Figure 37: Amsterdam arena Hausdorff distance visualization. The Hausdorff distance is visualized on a red-green-blue scale, where red is 0 distance, blue is d_{max} distance and green lies inbetween.

In terms of walkable space covered, the proposed parallel pipeline performs quite well. The pipeline achieves a ratio of nearly 1 in all three non-artificial environments. The minor differences in the environment, such as differences in re-triangulation and vertex positions due to rounding, contribute to the walkable space covered ratio not being perfect. Due to the rounding of intermediate results by forceful demotion of accuracy of the expression trees, however, it is expected that there will always be differences compared to the ground truth.

	Walkable space area (*ground truth*)	Walkable space area (*nav. mesh*)	Walkable space covered
House	179, 997.515625	179, 996.109375	0.99999218739216973567603927866713
Villa	9, 104, 859.000000	9, 104, 523.000000	0.9999630966278555219800767919635
Amsterdam arena	116, 101.421875	116, 101.242188	0.99999845232730919127686178477304

Table 7: Walkable space covered for the three non-artificial environments. The ground truth walkable space area Area is in generic units, since it is not standardized to metres. The walkable space covered metric is a ratio and thus has no unit. The ratio is close to 1 in all three cases, indicating that the proposed parallel pipeline produces an output comparable to the sequential pipeline (ground truth).

5.6 Scaling benchmark

To measure how well the parallel pipeline scales based on the number of threads in a computing node a set of environments is, again, processed. A test in this benchmark consists of a timing test performed in the same way as the parallel pipeline benchmark tests. The variables which change between tests are the number of threads ($2, 4, 8, 16$ and 32). The partitioning method used is the sorted recursive strategy with 2^4 partitions, because it had the lowest execution times in Section 5.4 with 32 threads. A higher number of partitions is not tested due to diminishing returns, as indicated by the experiments in Section 5.4.

Two metrics are used to gauge how well the proposed parallel pipeline scales with thread count, namely the speed-up metric and the *parallel efficiency* metric. The linear (ideal) speed-up

$$S_p = p,$$

is equal to the number of threads, where p represents the number of processors (or threads). The ideal speed-up for 4 threads is $4\times$, the speed-up for 8 threads is $8\times$, and so on. Typically ideal speed-up is not reached due to overhead costs of running a program in parallel. The parallel efficiency metric is

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p},$$

where T_s is the execution time of the sequential algorithm, T_p the execution time of the parallel algorithm and p the number of processors. [EZL89]

5.6.1 Results and discussion

In Figure 38, the speed-up is plotted against the number of threads. Here we can clearly see an expected result in the house environment: it doesn't scale well in a parallel environment. The environment has a low number of polygons, which causes the overhead of running the pipeline in parallel to dominate. Additionally the cutting during the partitioning of the environment also contributes so much to the parallel execution time that the end result is a sub-linear speed-up. Going beyond 8 threads does not offer additional speed-up in this environment due to the number of polygons within each single partition being relatively low at 2^4 partitions.

The villa and the Amsterdam Arena environments show an entirely different phenomena. A super-linear speed-up, for $p = 2, 4$ in the villa environment and $p = 2, 4, 8$ in the arena environment, is attained in both of these environments. Typically a super-linear speed-up cannot be attained, except for cases where the parallel algorithm has large differences in comparison to the sequential algorithm. Another case is when super-linear speed-up results from inefficiency in the serial algorithm [Gus90]. In our proposed parallel pipeline's case there are three large differences between it and the sequential pipeline's speed, namely:

- The environment is partitioned into many smaller sub-environments. Performing the vertical clearance filter on smaller environments results in a shorter execution time.
- Vertex coordinates are demoted to machine accuracy after the vertical clearance filter is applied. The complex expression trees created by the triangle-triangle intersections of the vertical clearance filter are reduced to a single node of machine accuracy. Rounding coordinates in this manner results in a lower execution time for subsequent filter steps (simplification filter, layer subdivision filter) by a large margin.
- An approximated version of the simplification filter is used, as proposed in Section 5.1.2. By not using exact computations here, *CGAL* does not have to evaluate the expression trees. Evaluating the expression trees (possibly extremely deep expression trees at that) is high in cost, so by avoiding these evaluations we gain speed.

Combining these factors together is what allows for the super-linear speed-up. If the original sequential pipeline was modified to also use rounded coordinates and the approximated simplification filter, the speed-up would be less pronounced.

In Figure 39, parallel efficiency is plotted against the number of threads. For the same reasons as the super-linear speed-up, we see an above ideal parallel efficiency. Also, again, in the house environment we see that it suffers in a highly parallel environment due to its low number of polygons. The speed-up for $4, 8$ and 16 threads is roughly the same, corresponding to the gradual approaching of 0 parallel efficiency at 32 threads. Using these data points we conclude that using more than 4 cores for a small environment (< 1000 polygons) is sub-optimal.

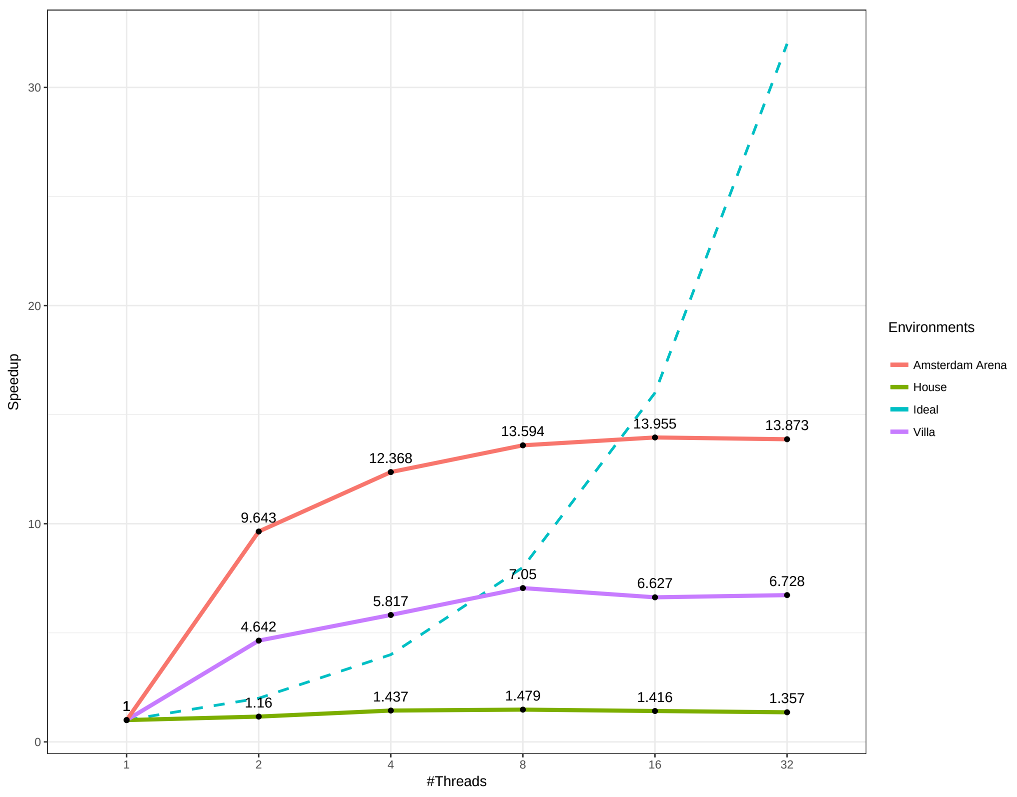


Figure 38: Speed-up vs. number of threads. The ideal speed-up is $s = S$, where S is the number of threads.

For the villa and Amsterdam Arena environment we see the same trend as with the speed-up. The difference between the villa and arena environment is that the villa environment is a highly vertical environment. By even using a small height clearance parameter in the vertical clearance filter, each (relatively) small partition has a high number of intersections to calculate. The arena environment has only a (relatively) small number of obstacle polygons which cause intersections resulting in a higher overall speed-up and, as a result, higher parallel efficiency. The speed-up of both these environments tops out at around 8 threads and beyond these 8 threads the parallel efficiency dips below ideal efficiency. For these reasons *large environments* of $> 10,000$ polygons ideally would be processed by $>= 8$ threads.

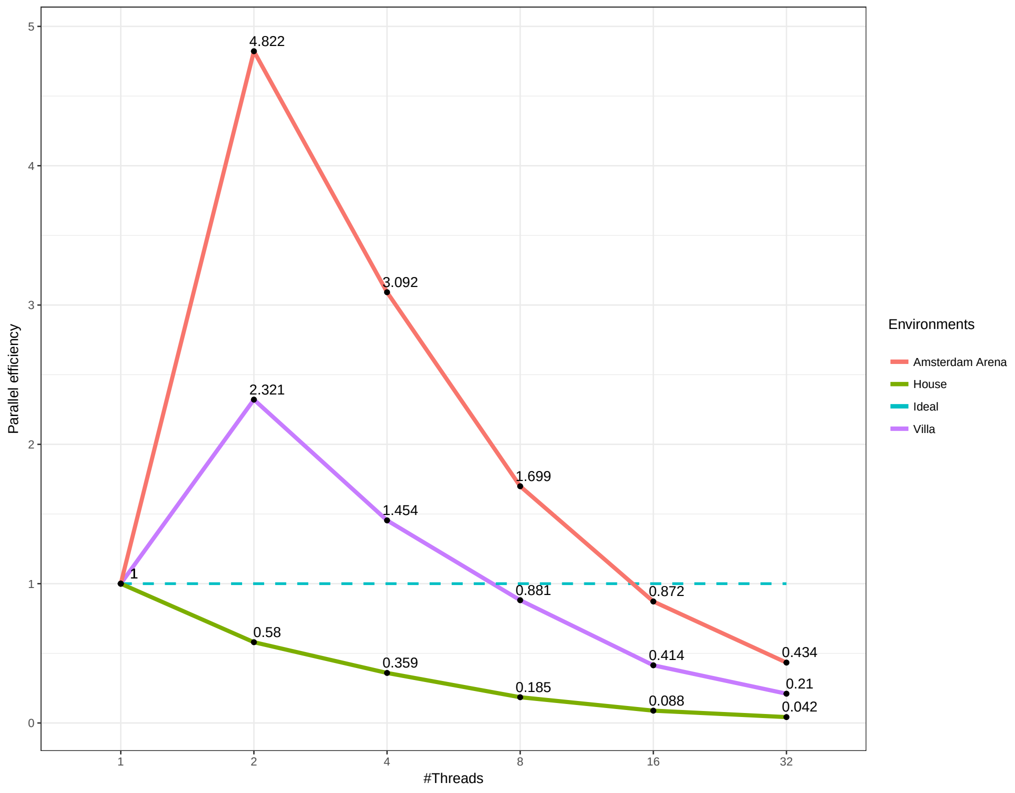


Figure 39: Parallel efficiency vs. number of threads. Ideal parallel efficiency is 1.

5.7 Distributed pipeline benchmark

As a final test, a distributed network is simulated locally on a singular machine. Nodes are simulated using multiple instances of the distributed application built with MPI [Gro+99]. Simulated nodes each have 8 threads assigned to them to run a parallel pipeline on. In total, four nodes are simulated for a total of 32 threads.

The simulated distributed network of $n = 4$ nodes is divided into one master node and three slave nodes. After initializing, the master node sends the slave nodes the polygonal environment and a pipeline definition file. The master node and slave nodes then read the environment and apply the pre-processing pipeline block to it. All nodes then partition the environment using the sorted recursive strategy into $m = 2^4$ partitions. The scheduling algorithm then maps the m partitions to the n nodes by using the LPT-algorithm described in Section 4.3.4. After the scheduler has assigned a node its partitions, the remaining partitions which were not assigned to it are removed from the working memory. The partitions belonging to the node are then processed by the parallel block producing the intermediary environment. By using the inter-process communication methods proposed in Section 4.2, nodes communicate their intermediary environments to node with rank 0, the master node. The master node then merges the intermediary environments into the merged environment. The master node does not communicate its intermediary environment to itself, since it already exists in the working memory. Afterwards the post-processing block is applied to the merged environment to get the walkable environment. A timing is performed from beginning to end on the master node to measure the execution time of the distributed pipeline. In Algorithm 4, the psuedocode of the MPI-application, which implements the distributed pipeline, is given.

Algorithm 4 Distributed pipeline (MPI)

```
1: Initialize MPI environment
2: $size \leftarrow$ Get number of processes in the MPI environment
3: $rank \leftarrow$ Get the rank of the local process in the MPI environment
4: Initialize pipeline blocks $p_{preprocessing}$, $p_{parallel}$, $p_{postprocessing}$ from pipeline definition file
5: Read and initialize polygonal environment $E$
6: Apply $p_{preprocessing}$ to environment $E$
7: Partition environment $E$ into partitions $P_0, \cdots, P_m$
8: $work \leftarrow$ Apply LPT-algorithm to$P_0, \cdots, P_m$to find set of partitions belonging to the local process$rank$
9: if $rank = 0$ then
10:     Apply $p_{parallel}$ to all partitions in $work$
11:     for $i = 1$ to $size$ do
12:         $envsize \leftarrow$ Receive size of intermediary environment of worker node $i$
13:         $E_{intermediary} \leftarrow$ Receive $envsize$ bytes forming the intermediary environment of worker node $i$
14:         Add all polygons from $E_{intermediary}$ to $E$
15:     Merge all intermediary environments into $E$
16:     Apply $p_{postprocessing}$ to merged environment $E$
17:     Write walkable environment to file
18: else
19:     Apply $p_{parallel}$ to all partitions in $work$
20:     $E_{intermediary} \leftarrow$ Merge all partitions into one environment
21:     Convert $E_{intermediary}$ into $n$ bytes using the communication format.
22:     Send size $n$ of intermediary environment to master node 0
23:     Send $n$ bytes comprising the intermediary environment to master node 0
24: Finalize MPI environment
```

The binary-format is used to test the speed of the distributed pipeline because of its high performance (high reading speed) as explored in the communication benchmarks in Section 5.3.1.

5.7.1 Results and discussion

Two environments are processed in the distributed environment to show the cost of communication in a distributed network. The baseline performance $t_{parallel}$ of a parallel pipeline ran on a single node from Section 5.4 is compared to $t_{distributed}$. The difference between $t_{parallel}$ and $t_{distributed}$ is the additional time spent in the distributed pipeline. This distributed overhead ($t_{overhead}$) contains setting up the MPI environment, communicating results between worker nodes and the master node and merging the intermediate results.

$n_{nodes} = 2$	**Villa**	**Amsterdam Arena**
$t_{parallel}$	$133,770ms$	$132,408ms$
$t_{distributed}$	$192,261ms$	$188,359ms$
$t_{overhead}$	$58,491ms$	$55,951ms$
$t_{postprocessing}$	$69,384ms$	$60,082ms$

When using a (simulated) distributed network of two nodes ($n_{nodes} = 2$) a large difference between performance in the non-distributed, parallel pipeline and the distributed pipeline exists. The overhead costs, $t_{overhead}$, makes the final run-time $t_{distributed}$ higher than the non-distributed pipeline. Additionally, the number of polygons per node are high, because the environment is split over only 2 nodes. This causes the communication costs to be high, because a large environment needs to be converted into the communication format and sent to the master node. Only one node needs to do this since the master node already has its own partitions in its working memory.

$n_{nodes} = 4$	**Villa**	**Amsterdam Arena**
$t_{parallel}$	$133,770ms$	$132,408ms$
$t_{distributed}$	$136,790ms$	$133,911ms$
$t_{overhead}$	$3,020ms$	$1,503ms$
$t_{postprocessing}$	$71,484ms$	$58,383ms$

In a distributed network of four nodes (n_{nodes}) performance gets closer to the parallel performance. The environment's partitions are subdivided over four sets with a roughly equal number of polygons. Each node, then, owns roughly 25% of the environment. The overhead in this case is lower, because, instead of one large environment, three smaller environments are communicated. The receiving, reading and merging of environments happens on the main thread. The master thread's own set of partitions are processed regardless of incoming communication. In this the worker nodes were done in order, which allows the distributed pipeline to read the intermediate environments immediately.

6 Conclusion

While the original goal of an execution time of $< 10s$ was not reached, respectable speed-ups were gained for larger, complex environments. In smaller environments, however, the weaknesses of partitioning become apparent. Partitioning a small environment and processing it in parallel brings with it a high number of overhead related to creating threads, cutting the environment and merging the partitions back into on environment. The parallel pipeline is only a semi-effective method for higher speed-ups in these cases and does not seem worth it for very small environments. In the experiments in Section 5.4 with the house environment, we find that the minimum amount of partitions suffices to create a 1.5× to 2.5× speed-up. The grid (by accident) and the sorted recursive partitioning methods seem about equally viable to use.

For large environments, however, the parallel pipeline offers significant improvements in speed. Environments that were previously not processable due to running out of memory or high execution times now successfully have a correct multi-layered environment extracted. In the Amsterdam Arena environment and the villa environment this becomes very apparent, as speed-ups of up to 14× are obtained with optimal partitioning and the usage of 32 cores.

In terms of partitioning complex environments, the sorted recursive strategy is the clear winner. Compared to the grid strategy and the recursive strategy, it produces a better workload balanced over threads in a machine, which naturally leads to lower execution times. The grid strategy and the recursive strategy's non-flexible placement of partitioning planes proves to be their downfall. In cases such as the house environment, a poorly aligned grid can cause extreme slowdowns. The sorted recursive strategy avoids this problem by placing the partitioning plane dynamically, avoiding pre-set coordinates ruining performance.

The sorted recursive strategy does not always provide the best overall performance in every situation though.

Quite unexpected is, however, that more partitions does not necessarily yield a faster parallel execution time $t_{parallel,total}$. In the villa environment, we observe a trend where the more partitions are present, the higher $t_{parallel}$ becomes, indicating that when an environment is cut multiple times, the vertical clearance filter will take longer to complete. This is because when an environment is partitioned, polygons intersecting a partitioning plane are split into two polygons. One of the polygons will remain in the partition in which all polygons below and above have vertices on the border of the partitioning plane. When polygons are projected in the downwards direction in order to compute their height clearance, they intersect a high number of polygons with vertices on the partitioning plane creating complex and expensive intersections for *CGAL* to handle quickly.

Also, partitioning without taking into account verticality was proven to be a major issue in our proposed partitioning methods. In purely vertical environments, the partitioning methods can increase the number of polygons in an environment twofold for each partitioning plane used. The artificial stacked boxes environment (refer to Figure 33) suffered a major performance loss when processed by the grid strategy, for example.

The correctness of the pipeline was evaluated in Section 5.5. Overall the output of the parallel pipeline is comparable to the output of the sequential pipeline by Polak [Pol16]. The distance between meshes generated by these pipeline ranges between a few micrometers (in small environments) and single millimeters (for larger environments). In some cases, however, the difference was significant. The experiments showed that these large differences (the samples with the highest Hausdorff distances) were outliers caused by the simplification filter working with rounded coordinates. The simplification filter aggressively collapsed edges it does not collapse in the purely exact sequential version, but our approximated version does with similar settings. By using less aggressive settings, these outliers can be suppressed, but this requires manual tweaking by the end-user.

In Section 5.6 the scalability of our proposed parallel pipeline was evaluated. The optimal number of computing cores was pinpointed at around 8 cores. Any more increased the overhead by such a significant margin the execution time dipped, rather than increased.

Another trend is the rise of the $t_{postprocessing}$ time against the number of partitions, indicating the rising cost of the simplification filter as the number of partitions rise. Intuitively, this makes sense, since the partitioning process splits all polygons intersecting a partitioning plane into two halves. The more partitioning planes introduced in a partitioning strategy, the more polygons are split and the more polygons will be in the final, merged environment. The more polygons are in the environment, the more polygons need to be processed by the simplification filter.

To summarize, the most important findings are repeated:

- Small environments of < 1000 polygons are worth it to process in parallel, but only into around 4 partitions. Even then, speed gains are at most about 2×.

- Large environments of $> 100,000$ polygons absolutely need to be processed in parallel. Non-complex large environments can be processed by either the grid strategy or the sorted recursive strategy for large speed-ups.
- Complex environments gain the most speed-up by being processed in parallel to a $14\times$ increase in performance. Using the sorted recursive strategy for these types of environments ensures problem areas are split up into multiple partitions.
- Vertical environments should absolutely not be processed in parallel. Purely vertical environments can, when cut by a partitioning method, rapidly increase in total polygon count and actually make the execution time **worse**.
- The proposed parallel pipeline performs best on a maximum of 8 computing cores (or threads). Any more threads will increase the overhead of multi-threading the pipeline by a significant amount and make the use of threads beyond the proposed 8 not worth it.
- The proposed distributed pipeline performs poorly, because the cost of communication is so significant.

All-in-all, a good speed-up of $2\times$ for small environments to up to $14\times$ for complex environments was introduced and extremely large environments, which could not be computed with methods of previous research, now compute in a reasonable time-frame. In the future this research could be extended to aim for an even higher increase in speed.

In Section 6.1 methods by which the geometry processing pipeline could be extended and improved are discussed.

6.1 Future work

In Section 5.4.2, the results show clearly that the major contributors to slowdown in the parallel pipeline are:

1. Performance of the vertical clearance filter in parallel environment $t_{parallel}$,
2. Performance of the simplification filter after partitioning $t_{postprocessing}$,
3. Overhead costs caused by the $CORE :: Expr$ data type $t_{overhead}$,
4. Speed of the partitioning process (cutting) $t_{partitioning}$.

When looking at structural ways to improve the speed of the parallel pipeline, the primary way to improve execution times would be to evaluate the environment in the third dimension z. In the proposed partitioning strategies the environment is evaluated from top-down. Any information pertaining to height is ignored, because of the height clearance restriction. An ideal partition (for the vertical clearance filter, the primary performance hog) would only contain the walkable areas and all potential polygons above these walkable areas that violate the height clearance restriction. These potentially low-hanging polygons can *also* be walkable areas. In the case that an environment is only constructed of potentially walkable polygons, a z-aware partitioning method would still have to include all polygons regardless of height in a single partition. This phenomena can occur if there's a tightly packed (in the z-axis) environment, such as the stacked tower environment in Section 5.4 and in cases where the height clearance restriction is set to a maximum value.

Another dilemma is introduced here because of these factors, since evaluating the environment on a third dimension will increase the time spent analysing the environment to choose optimal locations and orientation of the partitioning planes. By spending a large number of time partitioning the worry that $t_{partitioning}$ will exceed the actual processing time $t_{parallel}$.

For all other environments without a tightly stacked z-axis this would provide a much better fit and better load balancing, since only the minimum number of polygons for a partition to calculate the vertical clearance filter correctly would be available, reducing the number of intersections in the partition needed to be calculated in parallel and thus reducing $t_{parallel}$ significantly.

At the implementation level the speed of the partitioning process can be improved on two fields. Firstly, splitting the polygons in the partitioning process is done by using default $CGAL :: intersection$ methods from a partitioning plane to a polygon. Since the partitioning planes used in the proposed partitioning methods all are purely axis-aligned (to the xz- or yz-plane). Secondly after intersections are calculated polygons are retriangulated using a CGAL Voronoi triangulation. Because the intersection points are known, and it is known on which side of the partitioning plane they lie it is not necessary to use a Voronoi triangulation in order to find out how the polygon will look after a plane intersects and splits it. By finding out which vertices on the original polygon belongs to the negative side of the plane and which vertices belong on the positive side of the plane by using a plane check it becomes possible to create two triangles without ever touching a triangulation. In short, replacing *CGAL* functionality with smarter, tailor-made solutions will increase the speed of the entire pipeline.

Outside the scope of this research completely lies algorithmic restructuring. Some problems stem from choices made in previous research [Pol16], such as the implementation of the vertical clearance filter and simplification filters. The vertical clearance filter takes all obstacle polygons above a walkable polygon and calculates *every single* intersection point. By using a smarter algorithm, such as a space sweep algorithm or a more intelligent data-structure, the costs of the vertical clearance filter can be brought down to a more manageable level. By reducing the execution time of the vertical clearance filter $t_{parallel}$ can be brought down significantly.

The simplification filter is the secondary cause of slowdowns in heavily partitioned environments. Some of the experiments with higher numbers of partitions (and thus, more cuts using partitioning planes) had a high $t_{postprocessing}$. The implication here being that the more the partitioning method cuts an environment, the more time it takes to re-triangulate the merged result to a simplified result.

Additionally the results of the experiments show that the usage of the *CORE*-library's *Expr* number type is not well suited to a multi-threaded environment. The overhead cost of using this number type $t_{overhead}$ is manageable in smaller environments, but larger environments, such as the villa environment show the significant costs behind this number type. The dynamic memory structure, the automatic promotion and demotion in accuracy of number types for internal and the memory management done by the *CORE*-library itself can lead to potentially undefined behaviour. When one thread, for example, removes a node from a graph g_a another graph g_b is dependent on, while g_b is being processed by another thread, undefined behaviour arises.

7 References

[ASE02] Nicolas Aspert, Diego Santa-Cruz, and Touradj Ebrahimi. "Mesh: Measuring errors between surfaces using the hausdorff distance". In: *Proceedings. IEEE International Conference on Multimedia and Expo*. Vol. 1. IEEE. 2002, pp. 705–708.

[Bat+08] Vicente H. F. Batista et al. *Parallel Multi-Core Geometric Algorithms in CGAL*. 2008.

[Bat+10] Vicente HF Batista et al. "Parallel geometric algorithms for multi-core computers". In: *Computational Geometry* 43.8 (2010), pp. 663–677.

[Cho81] Anita L Chow. *Parallel Algorithms for Geometric Problems.* Tech. rep. Illinois University at Urbana Applied Computation Theory Group, 1981.

[Cig+08] Paolo Cignoni et al. "Meshlab: an open-source mesh processing tool." In: *Eurographics Italian chapter conference*. Vol. 2008. 2008, pp. 129–136.

[CL08] Patrick Cozzi and Boon Thau Loo. *Parallel Processing City Models for Real-Time Visualization.* 2008.

[DFR93] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. "Scalable parallel geometric algorithms for coarse grained multicomputers". In: *Proceedings of the ninth annual symposium on Computational geometry.* ACM. 1993, pp. 298–307.

[EZL89] Derek L Eager, John Zahorjan, and Edward D Lazowska. "Speedup versus efficiency in parallel systems". In: *IEEE Transactions on Computers* 38.3 (1989), pp. 408–423.

[Ger10] Roland Geraerts. "Planning short paths with clearance using explicit corridors". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on.* IEEE. 2010, pp. 1997–2004.

[Gro+99] William D Gropp et al. *Using MPI: portable parallel programming with the message-passing interface.* Vol. 1. MIT press, 1999.

[Gus90] John L Gustafson. "Fixed time, tiered memory, and superlinear speedup". In: *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5).* IEEE Press. 1990, pp. 1255–1260.

[HG01] Andreas Hubeli and Markus Gross. "Multiresolution feature extraction for unstructured meshes". In: *Proceedings of the Conference on Visualization'01.* IEEE Computer Society. 2001, pp. 287–294.

[Hil+16] Arne Hillebrand et al. "Separating a walkable environment into layers". In: *Proceedings of the 9th International Conference on Motion in Games.* ACM. 2016, pp. 101–106.

[Ins13] National Instruments. *Comparing Common File I/O and Data Storage Approaches.* Tech. rep. National Instruments, June 2013. URL: http://www.ni.com/white-paper/9630/en/.

[Kar+99] Vijay Karamcheti et al. "A core library for robust numeric and geometric computation". In: *Proceedings of the fifteenth annual symposium on Computational geometry.* ACM. 1999, pp. 351–359.

[Lin00] Peter Lindstrom. "Out-of-core simplification of large polygonal models". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques.* ACM Press/Addison-Wesley Publishing Co. 2000, pp. 259–262.

[OKK15] Hiromu Ozaki, Fumihito Kyota, and Takashi Kanai. "Out-of-Core Framework for QEM-based Mesh Simplification." In: *EGPGV.* Citeseer. 2015, pp. 87–96.

[Pol16] Robertus Mihai Polak. "Extracting walkable areas from 3D environments". MA thesis. 2016.

[The17] The CGAL Project. *CGAL User and Reference Manual.* 4.11. CGAL Editorial Board, 2017. URL: http://doc.cgal.org/4.11/Manual/packages.html.

[Tol+16] Wouter van Toll et al. "The Explicit Corridor Map: Using the Medial Axis for Real-Time Path Planning and Crowd Simulation". In: *LIPIcs-Leibniz International Proceedings in Informatics.* Vol. 51. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[Tol+18] Wouter Van Toll et al. "The medial axis of a multi-layered environment and its application as a navigation mesh". In: *ACM Transactions on Spatial Algorithms and Systems (TSAS)* 4.1 (2018), p. 2.

[Van+16] Wouter Van Toll et al. "A comparative study of navigation meshes". In: *Proceedings of the 9th International Conference on Motion in Games.* ACM. 2016, pp. 91–100.

[Zur+08] Dan Zuras et al. "IEEE standard for floating-point arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70.