Efficient visualization and analysis of large-scale, tree-based, adaptive mesh refinement simulations with rectilinear geometry

Guénolé Harel, Jacques-Bernard Lekien, Philippe P Pébaÿ, Jérôme Dubois

Research report

Efficient Visualization and Analysis of Large-Scale, Tree-Based, Adaptive Mesh Refinement Simulations with Rectilinear Geometry

Guénolé Harel[1], **Jacques-Bernard Lekien**[1], **Philippe P. Pébaÿ**[2], **Jérôme Dubois**[1]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
e-mail: {guenole.harel,jacques-bernard.lekien,jerome.dubois}@cea.fr
[2] NexGen Analytics, 412 N. Main St, 82834 Buffalo WY, U.S.A.
e-mail: philippe.pebay@ng-analytics.com

Paper written in July 2019

Abstract We present the first systematic treatment of the problems posed by the visualization and analysis of large-scale, parallel tree-based adaptive mesh refinement (AMR) simulations on an Eulerian grid.

When compared to those obtained by constructing an intermediate unstructured mesh with fully described connectivity, our primary results indicate a gain of at least 80% in terms of memory footprint, with a better rendering while retaining similar execution speed.

We thus describe herein the key concepts that allowed us to obtain these results, together with the methodology that facilitates the design, implementation, and optimization of algorithms operating directly on such refined meshes. In 2019, this native support for AMR meshes has been contributed to the open source Visualization Toolkit (VTK).

We note that this work pertains to a broader long-term vision, with the dual goal to both improve interactivity when exploring such data sets in 2 and 3 dimensions, and optimize resource utilization.

Key words scientific visualization, meshing, AMR, mesh refinement, tree-based, octree, VTK, parallel visualization, large scale visualization, HPC, isocoutour, isosurface

Contents

1 Introduction

1.1 Preamble

Massive numerical simulations are nowadays routinely run on petascale supercomputers such as Tera100 and the pre-exascale Tera1000 [1]. Among simulation codes, those using adaptive mesh refinement (AMR) are especially efficient at tracking fine details within very large domains of interest. AMR enables a trade-off between numerical accuracy, memory footprint, and computational cost, by allowing for mesh refinement (and coarsening) in sub-regions of the simulation. Recent large scale simulations have reached ten trillion cells on a regular Eulerian grid [26]. This pioneering work, representative of what will be "everyday" tomorrow exascale computing, showed that it might however be either impossible to store due to a too large number of elements, or would be computationally too expensive to post-process.

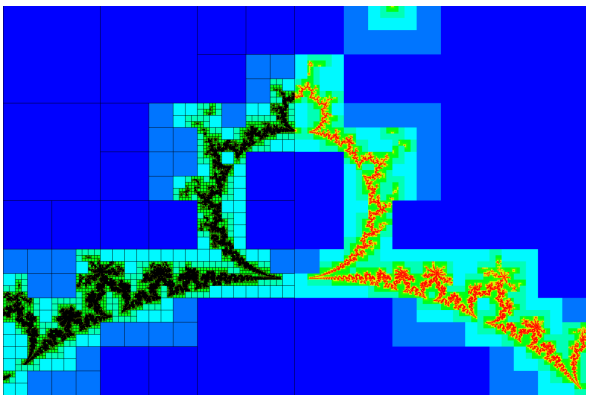

Fig. 1.1 Simplified simulation of a Mandelbrot set based on a tree-based AMR mesh, highlighting the underlying grid as wireframe in the leftmost half of the image.

These difficulties can be alleviated by refining the original mesh only where needed, while retaining coarser elements wherever local feature scales permit. Of course,

this approach is limited to those specific physical problems where the meaningful phenomena are spatially localized and where appropriate refinement and/or coarsening criteria can be defined. This is the case e.g. in shock wave computation [7,8] (illustrated in Figure 1.1), astrophysics [25], or transient wave propagation [3].
Since the first description of an AMR methodology with the Berger-Oliger [9] type, several implementations have been proposed and developed. It is beyond the scope of this article to provide an in-depth comparison of *block structured* (also known as *patch-based*) versus *tree-based* (also known as *point-wise structured*) AMR methodologies. Nonetheless, in order to fully understand the motivations and constraints of the work presented hereafter, one must be aware that the fundamental difference between the two approaches is, essentially, a trade-off between memory footprint, and complexity of processing algorithms. Specifically, *ceteris paribus*, a typical tree-based AMR grid will occupy much less memory estate than its block structured equivalent, at the cost of higher processing time as a result of more complicated algorithms. In fact, this dichotomy between methods arises from the more general opposition between implicit and explicit representations, with the ensuing consequences when storing, as opposed to processing, the resultant data objects. It is worth noticing here that the FLASH framework [17] offers both options, although this is actually done with two different underlying codes: Chombo (block structured) and Paramesh (tree-based).
Block structured AMR will not be discussed in the rest of this article; the interested reader can refer in particular to the Chombo pages for more details [2]. Our interest instead focused on the analysis of data sets produced by tree-based AMR codes. Several codes pertain to this group, for instance starting with successive refinement in octants (therefore producing *octrees*) of an initial root cell as done in RAMSES [25], or using a uniform, structured grid of root cells as done in RAGE/SAGE [18] or HERA [20], the tree-based AMR hydrodynamics simulation code developed at CEA.

1.2 Scope

We begin by providing in §2 **[Context]** the background and context for this work, analyzing the challenges posed by tree-based AMR simulations to scientific visualization. As a result, we propose our global vision for addressing these challenges in an exascale perspective. However, the scope of this article is limited to the foundational aspects of this vision, laying out the necessary data structures as well as the methodology to optimally process these.
The fundamentals of our novel data structures are provided in §3 **[Foundations]**, and additional design choices are proposed in §4 **[Properties and Operations]**, which we implemented in VTK [6]. Wherever necessary, based in particular on acquired experience with large-scale data sets, we mention changes to claims or hypotheses that we had made in earlier work [12].
We then study in §5 **[Method]** the method we designed to operate on these data objects, with a particular emphasis on execution speed, in order to maintain interactivity even while processing the largest data sets that can be stored on currently available hardware. We illustrate this methodology in §5.7 with isocontour (resp. isosurface), which is arguably one of the most widely used visualization techniques albeit being difficult to implement in a way that is both correct and efficient.
In §6 **[Results]**, we examine the validity of our claims relative to performance with a set of tests that are representative of the scientific simulation data sets we wish to address.
Finally, we conclude this article in §7 **[Conclusion]** by examining to what extent the work presented in these pages covers what we initially intended to do. We subsequently discuss how future work will be articulated with what has been achieved so far, in order to achieve our long-term vision.

2 Context

2.1 Problem Statement

In order to exploit the massive data sets produced by the various numerical simulation codes of CEA, our visualization team developed the Large Object Visualization Environment (LOVE) [4,5], a dedicated parallel visualization tool. It is based on VTK/ParaView, an open-source, C++ set of libraries and an application for scientific data visualization and analysis supporting many data types and featuring hundreds of algorithms, with thousands of users in the global scientific community.
One approach for the visualization and analysis of AMR data sets with VTK is to use its native unstructured grid data objects. One obvious advantage of this method is to make available the wealth of existing filters already in VTK for such data sets (e.g., cutting, clipping, isocontour, etc.). However, the additional memory requirements that arise from converting a mostly implicit data object into a fully explicit one rapidly become prohibitive as the size of the grid grows. Furthermore, when the cells of an AMR mesh are directly used as unstructured element inputs (quadrilaterals or hexahedra) of an algorithm such as isocontour, topological irregularities resulting in strong visual artifacts such as gaps may appear. These are caused by the topology of AMR meshes, which have partly connected vertices ("T-junctions") when they contain neighboring cells at different refinement levels. Linear interpolation, commonly used by visualization algorithms, produces discontinuities across T-junctions, ultimately resulting in incorrect visualizations. The latter problem is further explicated in dimension 2 by Figure 2.1, where the top row represents an AMR grid

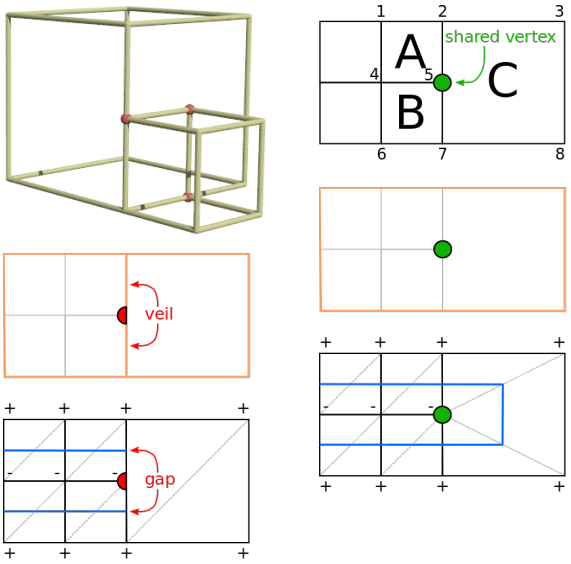

Fig. 2.1 Top: AMR grid converted into: (left) a quadrangle mesh with a T-junction at point 5, which is shared by A and B but not by C, and (right) a generic unstructured mesh where vertex 5 is shared by pentagon C as well as quadrangles A and B. Middle: outside boundary (orange) computed for both meshes; a topological artifact (*veil*) is caused by the T-junction (left), but not in the conforming, generic mesh (right). Bottom: linear isocontour (blue) computed for both meshes, between vertex values above (+) or below (−) a given value; dashed gray lines represent possible triangulations used by the contouring algorithm.

with 5 cells. On the left, the cells are considered as the elements of an unstructured quadrilateral mesh: by construction, quadrangle C does not have any reference to vertex 5, creating a T-junction along edge 2–7. On the right, the cells are now viewed as arbitrary polygons, with pentagon C sharing vertex 5 with quadrilaterals A and B, hence eliminating the T-junction. Attempting to extract the outside boundary of the quadrilateral mesh results in a topological artifact, called a *veil*, whereas the outside boundary is correctly extracted on the polygonal mesh. Similarly, the effect of linear isocontour on both constructions, when cell values are above or below a given isovalue are shown in the bottom row. The T-junction on the left causes a gap in the isocontour, because the algorithm cannot detect a contour intercept along edge 2–7 of cell C. Meanwhile, the same contouring algorithm is able to correctly process the generic cells, and produces a correct isocontour without false gaps.

The Hercule I/O library developed at CEA [11] supports such conversion from AMR grids into unstructured, conforming meshes. Prior to 2012, this was the only option available to visualize the tree-based AMR data sets produced at CEA. In addition to the already discussed performance limitations, using this approach also comes at the price of reduced interactivity because of I/O latency. Furthermore, at the time of writing, VTK does not support well the mixing of hexahedral elements with generic

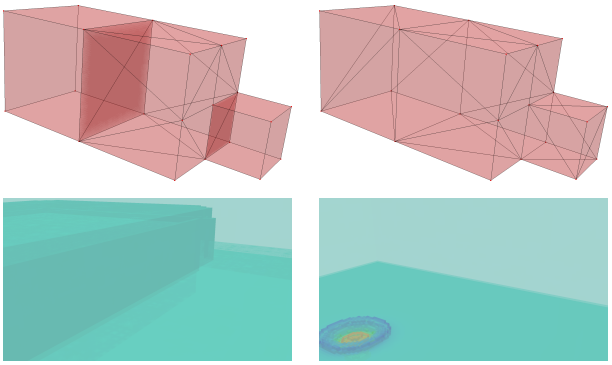

Fig. 2.2 Top row, left: one generic cell located between 2 hexahedra, resulting in the appearance of extraneous veils being generated by the VTK geometry filter; right: all cells are generic and the boundary is correctly extracted by the filter. Bottom row: outside boundary of a bi-material 3D AMR simulation; left: extraneous veils appear when the filter is applied to a mixed-cell conforming unstructured mesh; right, the boundary is correctly extracted with all generic cells.

cells as illustrated in Figure 2.2, left. Specifically, when attempting to extract the outside surface of the unstructured mesh with mixed cells, the subdivision of the generic cells by VTK results in incompatible tessellations across neighboring element faces. Although it is possible to resolve this problem by using only generic cells, as shown in Figure 2.2, right, but the computational and memory costs quickly become prohibitive for realistically-sized meshes.

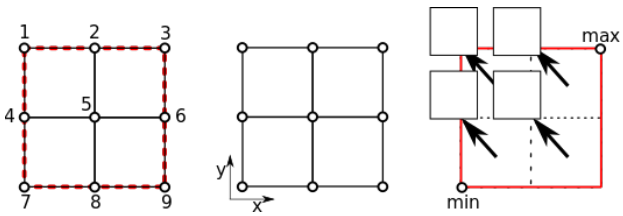

Fig. 2.3 Two different representations of the same mesh: explicit unstructured representation (left), versus AMR description (right). The red color coding indicates the root cell, which is stored in the AMR representation, but not in the unstructured mesh.

It is easy to illustrate, for instance with the simple example depicted in Figure 2.3, the dramatic inefficiency of using explicit unstructured meshes to represent AMR grids. Considering a quadrilateral in dimension 2, decomposed into four sub-elements, it is straightforward to devise a corresponding tree-based AMR representation using four floats for the extremum coordinates of the grid and a single Boolean value to indicate that the quadrangle is subdivided.

Meanwhile, an explicit unstructured representation of the same requires $9 \times 2 = 18$ floats for the vertex coordinates, as well as $4 \times 4 = 16$ integers to describe the connectivity of the 4 cells. Therefore, the AMR descrip-

tion reduces the memory footprint of almost a full order of magnitude for this simple case alone.
VTK has long provided some support for block-structured AMR data sets. Prior to 2012, it also offered very limited support for a particular case of tree-based AMR with a single-root octree object [27].

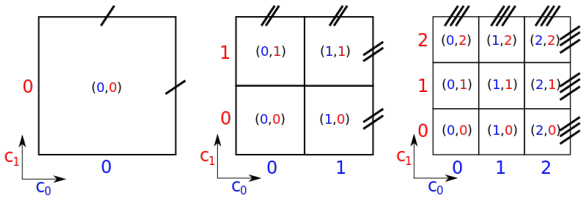

Fig. 2.4 The 3 allowed AMR subdivision patterns in dimension 2: without refinement ($f = 1$, left), binary subdivision ($f = 2$, center), and ternary subdivision ($f = 3$, right), with respective numbers of children equal to 1, 4, and 9. In dimension 3 these translate respectively into 1, 8, and 27 children.

Furthermore, simulation codes such as HERA use either binary or ternary subdivision schemes when refining meshes, i.e., with *branching factor* $f \in \{2; 3\}$ along each dimension of the grid. This is illustrated by Figure 2.4 in dimension 2. In general, in dimension d, refining a cell results in obtaining f^d sub*children* (sub-cells). Any post-processing methodology designed to handle the results of such simulations must therefore be able to accommodate, not only the usual binary trees, quadtrees and octrees, but also more exotic ternary trees. Finally, we must also take into account the constraint that AMR simulation codes used at CEA are run in parallel, with the corresponding data sets being distributed over many thousands of compute nodes. These codes balance computational sub-domains by allocating the root cells in the grid of trees, resulting in individual AMR trees that are never shared between different compute processes. Traversal objects for such grids of trees must therefore be carefully designed in order to *a priori* allow for extremely unbalanced trees structures between various areas of the overall domain.

2.2 Vision

Our global, long-term vision for tree-based AMR visualization and analysis can be articulated as follows:

[a] Propose a novel VTK data object to support all requested tree-based features, that is memory-efficient.
[b] To allow for the direct utilization of the wealth of existing unstructured mesh algorithms, propose a filter to convert VTK data to conforming meshes (dual mesh).
[c] Design and implement visualization and analysis algorithms that are specific to the primary tree structure, with a strong emphasis on performance. In our vision, this optimization of execution speed is best achieved by using specialized constructs called *cursors* and *supercursors*.
[d] Optimize rendering speed, in order to maintain interactivity when visualizing the largest possible tree grids that can be contained in memory. A possible approach could be to take advantage of the tree structure of the grids, to allow for level-of-detail culling relative to the size of the rendering window, screen resolution, view and camera position, etc.
[e] Qualitatively improve the final rendering with, e.g., mapping, texture splatting or ray tracing techniques specifically tailored for the tree-based AMR objects.
[f] Design and implement a way to pass object information, so that a reader specific to tree-based AMR grids will be able to limit actual reading and storing of those parts of the entire grid that are explicitly needed by filters and rendering (such as maximum depth of refinement and bounding box).
[g] Define a serialization specification for these structures, and develop I/O classes implementing it. Such a serialization protocol will also improve current parallel load balancing schemes by allowing for communication of large sub-grids in small messages.
[h] Expand the range of efficiently supported tree-based AMR data sets; envisioned objects include grids that have many root cells but a small number of refinement levels or, conversely, that only have a very small number of root cells with many refinement levels. Such extensions would have to be achieved while maintaining the same goal of memory footprint minimization and execution speed maximization.
[i] Expand the current post-processing paradigm to include concurrent approaches based on *in situ* and *in transit* processing. Such a data-centric approach would allow for increased spatial and temporal resolutions for post-processing purposes, reduced I/O costs, and significant decrease of time from data to insight. This would therefore alleviate increasing difficulties encountered by AMR simulation analysts caused by the current need to save a sufficient amount of raw solution data to persistent storage.

We acknowledge that some of these items are mutually independent, and thus do not have to be executed in the order of the list. However, [a] and [c] constitute the necessary foundation of the whole; this article is therefore focused on these two steps. Furthermore, neither of these features for tree-based AMR grids were supported by VTK prior to 2012, when we briefly described some governing principles of our preliminary work in [12]. In addition, several years have passed since this first approach to the problem, and our ideas and implementations have since matured and solidified.

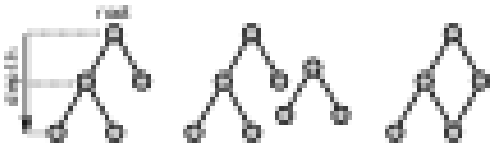

Fig. 3.1 Three different types of graphs: tree (left); not connected graph (center), not a tree; and a graph containing a cycle (right), therefore not a tree. We decide to always show the root at the top.

3 Foundations

As a result of the context described in § 2, we have designed the first native support for such data sets, that is also independent regardless of the choice of implementation. This includes both data and execution models, whose foundations are now fully described in this section.

3.1 Vertices, Graphs, and Trees

It is beyond the scope of this article to provide an extensive picture of graph nomenclature and classification; the interested reader can refer, to [10] for a systematic treatment of the theory of graphs. The fundamental building blocks of our trees are *vertices*, which can also be implemented as data objects containing various quantities of interest, such as simulation data, and mesh topology or geometry attributes. Given a set of vertices V, we then define an *undirected edge* as a set of pairs of vertices, with the following requirements for the set of all undirected edges:

(i) be connected, i.e., any two vertices are connected by a path of adjacent edges, and
(ii) not contain any cycle, i.e., a set of edges forming a closed polygon.

In addition, one (and only one) vertex is chosen in V to be the *root*. In this setting, the directed edges are immediately deduced from the undirected ones with the unambiguous implicit ordering based on distance from the root, in the sense of number of edges needed to transitively connect to it; one can then define a unique *depth* as being the number of edges in this path. Finally, at least one vertex does not have any directed edge leaving it, and any vertex that has this property is called a *leaf*; all non-leaf vertices are called *strict nodes*.
For a good understanding of this article, it is important to acknowledge that the typical usage of the term *tree* in computer science refers to a *directed, rooted, acyclical graph*, see for instance [21], whereas in mathematics it is more broadly understood as an *undirected, transitive, acyclical graph*. The double (V, E), where E is the set of all directed edges, is the definition of a *tree* to be used hereafter. A handful of examples and counter-examples are provided in Figure 3.1; note that, for concision, we never represent the directionality of the edges, for it is implicit as we use the convention to always represent a tree with its root at the top. We also decide to always horizontally align vertices that have the same depth.

3.2 Hypertree Object

We now introduce the concepts specific to our work.

3.2.1 Fundamentals We define what a hypertree object.

Definition 3.1 *A* Hypertree object *(shorthand* Hypertree*) in dimension* $d \in \mathbb{N}^*$ *with* branching factor $f \in \mathbb{N}^*$*, is a type of data set that can be represented as a tree, and where each strict node has exactly* f^d *children. In addition, primary attributes of this data set are attached to the vertices of the tree.*

Remark 3.1 The types of AMR meshes required by our target applications are limited to the possible combinations of $d \in \{1; 2; 3\}$ and $f \in \{2; 3\}$. The corresponding objects are called *quadtrees* in $(d = 2; f = 2)$ and *octrees* in $(d = 3; f = 2)$.

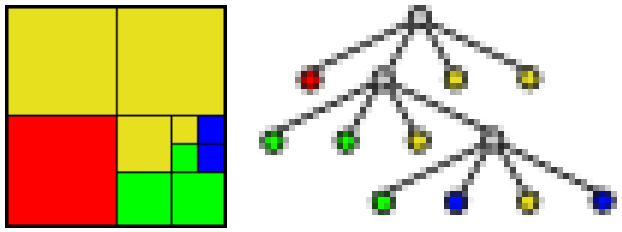

Fig. 3.2 Left: a 2-dimensional AMR mesh obtained with 4 levels of successive binary refinements of one quadrilateral; right: the corresponding hypertree representation. Here, different colors are used to represent the attribute values attached to mesh cells. It is possible to assign attribute values to strict nodes, here colored in gray.

There is a trivial bijection between hypertree objects and tree-based AMR meshes descending from a unique root cell: for instance, each leaf of a hypertree object $\mathcal{H}$ represents exactly one mesh cell that is not refined, whereas strict nodes in $\mathcal{H}$ are bijectively associated with all *coarse cells* (i.e., cells in the mesh that are subdivided). This bijective construction is illustrated with the case of a quadtree in Figure 3.2. Some CEA simulations codes take advantage of this feature and compute coarse cells attribute values. Last, as will be seen later, we exploit this capability in the aim of optimizing tree traversals for some classes of filters, e.g. for level-of-detail (LOD) purposes.
We make the choice to embed any meshes in the 3-dimensional Euclidean space $\mathbb{R}^3$ and to support only axis-aligned tree-based AMR meshes. This allows us,

when algorithms are applied, to have all input and output meshes in the same dimensional Euclidean space, even if the output has lower topological dimensionality. On the other hand this conceptual choice does not allow us to define meshes natively in 1D or 2D when needed; rather, they are always viewed as a 3D object with one or two fixed coordinates.
Because the considered AMR meshes are always rectilinear, the geometry of a hypertree object is implicitly given by $(\overrightarrow{x}; \overrightarrow{s})$:

1. the origin $\overrightarrow{x} = (x_0; x_1; x_2) \in \mathbb{R}^3$ of the root node;
2. the size $\overrightarrow{s} = (s_0; s_1; s_2) \in \mathbb{R}^3$ of the root node.

From the size can be deduced the actual axis used by the hypertree. For instance, if s_1 is equal to 0 then the normal of the plan including the hypertree will be $(0, 1, 0)$.

3.2.2 Hypertree Child Implicit Index Map An hypertree is constructed by successively refining cells when needed, starting from its root cell. By definition, we consider there are always f^d children cells to any coarse cells. By design choice, the traversal order of the children of one coarse cell is imposed, thus unambiguously defining an implicit ordering. We decide to use the following convention:

Definition 3.2 *The* Hypertree Child Index Map $\Phi_{d,f}$, *with* $(d, f) \in \mathbb{N}^{*2}$ *is the lexicographic order over* $[\![0; f[\![^d$.

It is beyond the scope of this article to discuss the lexicographic order over Cartesian products in a detailed way; suffices to know that it is the analog to the lexicographic order over finite words in a finite alphabet (the dictionary order) and that it indeed provides a total order. In addition, one has the following property, whose proof is left to the interested reader by recurrence over d:

Proposition 3.1 *Given child coordinates* $(c_0, \ldots, c_{d-1}) \in \mathbb{N}^d$ *in dimension* d*:*

$$\Phi_{d,f}(c_0, \ldots, c_{d-1}) = \sum_{k=0}^{d-1} c_k f^k.$$

The index maps for $d = 1$ are simply the identities over $[\![0; f[\![$. For the values of d and f that are of practical interest to us, the (c_0, c_1, c_2) tables are given in Figures 3.3 and 3.4, which illustrate respectively the following hypertree configurations: $(d = 3,\ f = 2)$ and $(d = 3,\ f = 3)$. When considering only the 2 tables with $c_2 = 0$ amongst the above, one obtains the corresponding maps for $d = 2$, shown in Figure 2.4.

3.3 Hypertree Grid Object

In order to account for a broad category of tree-based AMR grids, including those that do not have uniform geometry along each axis, or whose initial refinement pattern is not that of hypertree, we introduced a broader-scoped object.

c_1	1	6	7
	0	4	5
$c_2 = 1$		0	1
		c_0	

c_1	1	2	3
	0	0	1
$c_2 = 0$		0	1
		c_0	

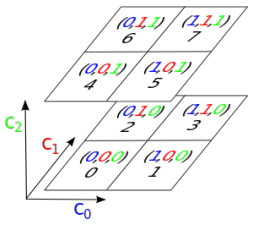

Fig. 3.3 The hypertree child index map in the 3-dimensional binary case $(d = 3,\ f = 2)$.

c_1	2	6	7	8
	1	3	4	5
	0	0	1	2
$c_2 = 0$		0	1	2
		c_0		

c_1	2	15	16	17
	1	12	13	14
	0	9	10	11
$c_2 = 1$		0	1	2
		c_0		

c_1	2	24	25	26
	1	21	22	23
	0	18	19	20
$c_2 = 2$		0	1	2
		c_0		

Fig. 3.4 The hypertree child index map in the 3-dimensional ternary case $(d = 3,\ f = 3)$.

Fig. 3.5 A 2-dimensional AMR mesh obtained with 5 levels of successive binary refinements of 4×2 rectilinearly aligned hypertree objects with different sizes along each axis. Here, different colors are used to represent the level of each cell, i.e. red for unrefined hypertrees and blue for the finest meshes.

Definition 3.3 *Let* $\mathcal{H}$ *and* $\mathcal{H}'$ *be two hypertree objects, with same dimension* $d \in \{1; 2; 3\}$ *and same* branching factor $f \in \mathbb{N}^*$, *with respective* 3*-dimensional embeddings* $(\overrightarrow{x}; \overrightarrow{s})$ *and* $(\overrightarrow{x}'; \overrightarrow{s}')$. *If*

$$\exists k \in \{0; 1; 2\} \quad \begin{cases} x'_k = x_k + s_k \\ \forall l \in \{0; 1; 2\} \setminus \{k\}\ (x'_l, s'_l) = (x_l, s_l) \end{cases}$$

and, when $d \neq 3$, *we say that* $\mathcal{H}'$ *is* rectilinearly consecutive to $\mathcal{H}$ for component k, *denoted* $\mathcal{H} \underset{k}{\prec} \mathcal{H}'$.

Intuitively, what this means is that the outside boundary of $\mathcal{H} \cup \mathcal{H}'$ is a line segment in dimension 1, a rectangle in dimension 2, and a rectangular prism in dimension 3, with origin and orientation equal to those of $\mathcal{H}$, and size vector as well, except for its k component which is equal to $\overrightarrow{s_k} + \overrightarrow{s_k}'$. For example, on Figure 3.5, are shown eight binary hypertree objects in dimension 2, arranged in order to have rectilinear consecutiveness for components 0 and 1.
Given any triple $t \in \mathbb{N}^3$, we denote Πt the product of its components, and $[\![t[\![$ the set of triples $t' \in \mathbb{N}^3$ such that

$t' < t$ in the lexicographic sense. For example,

$$[\![(3;2;2)]\!] = \{\{0;0;0\};\{0;0;1\};\{0;1;0\};\ldots;\{2;1;1\}\}.$$

We now introduce our main object:

Definition 3.4 *A* Hypertree Grid object *(shorthand* Hypertree Grid*) in dimension* $d \in \{1;2;3\}$ *with* branching factor $f \in \mathbb{N}^*$ *and* extent $E \in \mathbb{N}^{*d} \times \{1\}^{3-d}$*, denoted* $\mathcal{G}_E^{d,f}$*, is a type of data set comprising* ΠE *hypertree objects in dimension* d *and with branching factor* f*, denoted* $\mathcal{H}_{i,j,k}$ *where* $(i;j;k) \in [\![E]\!]$*, such that*

$$\forall (i;j;k) \in [\![E]\!] \begin{cases} i+1 < E_0 \Rightarrow \mathcal{H}_{i,j,k} \underset{0}{\prec} \mathcal{H}_{i+1,j,k} \\ j+1 < E_1 \Rightarrow \mathcal{H}_{i,j,k} \underset{1}{\prec} \mathcal{H}_{i,j+1,k} \\ k+1 < E_2 \Rightarrow \mathcal{H}_{i,j,k} \underset{2}{\prec} \mathcal{H}_{i,j,k+1} \end{cases}$$

In addition, primary attributes of this data set are attached to the individual hypertrees.

Given an arbitrary hypertree grid object $\mathcal{G}_E^{d,f}$, we denote $\mathcal{H}_{i,j,k}^{d,f}$ the hypertree object with discrete coordinates $(i;j;k) \in [\![E]\!]$ and call it the *constituting hypertree of* $\mathcal{H}_E^{d,f}$ *at position* $(i;j;k)$. Under the assumptions of Definition 3.4 regarding d, f, and E, we have:

Proposition 3.2 *The outside boundary of a hypertree grid object* $\mathcal{G}_E^{d,f}$ *is a* d*-dimensional rectangular prism, uniquely determined by the* 3*-dimensional embeddings of its constituting hypertree objects.*

Remark 3.2 Applying the same argument to the origin vectors of the constituting hypertrees shows that these are exactly the vertex coordinates of a rectilinear grid, whose elements are exactly the bounding boxes of said hypertrees. Because of this structure, it is sufficient to describe it *implicitly* by storing one coordinate array per dimension and a single orientation for the entire grid, at a much smaller total cost between $d(\sqrt[d]{\Pi E}+1)$, where ΠE is the number needed for a full explicit store, (best case: $\mathcal{G}_{(a,a,a)}^{3,f}$) and $\Pi E + 5$ (worst case: $\mathcal{G}_{(a,1,1)}^{d,f}$) all hypertrees consecutive along a single direction) floats, plus a single integer.
From *Hypertree Grid*, one can derive a more constrained and specialized structure called the *Uniform Hypertree Grid* based on a *regular* rectilinear grid. This allows us to describe the vertex coordinates in an even more *implicitly* and compact manner based on coordinates of bounding box for a constant cost of $2 * d$ floats plus d integers.

4 Properties and Operations

This design includes choices like defining optional properties, hypertree grid operation for the traversals, connection with existing non-AMR algorithms and finally, we introduce a hypertree grid-aware algorithm.

4.1 Properties

We made the choice to store some optional properties at the hypertree grid.

4.1.1 Property: Depth-limiter Filters might want to apply depth-limiting strategy to reduce execution times at the expense of mesh resolution. To do so, we introduce a logical depth-limiter value at the hypertree grid level.

4.1.2 Property: Global Index Maps In scientific computations, multiple properties are generally defined, all of equal length and allow for random access per cell index. For AMR meshes, each cell has an associated indexed value in each of these properties and is also mapped to an hypertree node. To achieve this, a first index map identifies each hypertree (Definition 4.1) and a second index map associate a tag for each node in a chosen hypertree (Definition 4.2).

Definition 4.1 *The* Hypertree Global Index Map $n_{HT} \in \mathbb{N}$ *of a hypertree* $(i,j,k) \in \mathbb{N}^3$ *is an injective function returning values in* $[O, \Pi E]$.

Definition 4.2 *The* Global Index Map $n_g \in \mathbb{N}$ *of one vertex in a hypertree grid* $\mathcal{G}_E^{d,f}$*, locally identified by* $n_l \in \mathbb{N}$ *inside the constituting hypertree* $n_{HT} \in \mathbb{N}$*, is an injective function:*

$$\begin{aligned} \Gamma_{d,f,E} : \quad \mathbb{N}^2 &\longrightarrow \mathbb{N} \\ (n_{HT}; n_l) &\longmapsto n_g \end{aligned}$$

with $n_l \in [0\,; C_{n_{HT}}[$ *where* $C_{n_{HT}}$ *is the number of cells of a hypertree identified by* n_{HT}*, and*
$n_g \in [0\,; C[$ *where* C *is the number of nodes (strict or not) in the whole hypertree grid.*

One might want to define the *Global Index Map* implicitly. For instance, an implicit ordering of the cells might be defined as $n_g = n_l + \sum_{i=0}^{n_{HT}-1} C_i$, with the requirement of the simulation properties following the same indexing. In practice, simulation properties are not ordered the same way as the visualization needs, and so creating an explicit *Global Index Map* will be mandatory. In any case the indexing scheme will be applied for all properties and consequently all properties must be ordered the same way.

4.1.3 Property: Bitmask As stated in Definition 3.1, a non-strict node has always f^d children. The notion of *bitmask* is introduced to allow the masking of each hypertree grid node. It is a bit array, sized equal to the number of vertices (both non-strict and strict nodes). It follows the global indexing scheme of the hypertree grid. This bitmask is an attribute with negligible relative memory footprint, for it only consumes one bit per tree vertex.
AMR simulations results we wish to support can distinguish between different *materials* participating in the

simulation. A first visualization approach is to create a single hypertree grid and define a simulation field value on a per-cell basis for the whole mesh. This means that for instance if one material with an associated simulation property is present in a small portion of the whole simulation domain, then a value must be defined for all nodes of the AMR mesh, even those not participating in this material. This can introduce unnecessary higher memory usage. Also with such approach, it is not possible to visualize a single material directly without some filtering process such as Threshold. The bitmask will be required to virtually prune the trees for proper visualization of the filtered hypertree grid. A second approach to handle materials is to create one hypertree grid per material. Depending on the materials presence in the simulation results, such approach can greatly reduce the memory footprint. The bitmask will still be of use to stick to the actual material presence in the trees. Thanks to this second approach, the associated simulation properties size will be reduced depending on materials definition.
Furthermore, the bitmask offer opportunities to optimize filters outputting hypertree grid memory-wise, when coupled with some *shallow copy* operations. When doing so, simulation properties size are kept identical, and so any new added property array will be as large as the original mesh. When the filtering produces small meshes, it is better for filters to generate a new hypertree grid of smaller size, with smaller attached property arrays.

4.2 Hypertree Grid Operation

We are now at a point where we can represent and store all tree-based, rectilinear axis-aligned AMR meshes we want to support. The question that immediately follows is that of operating efficiently on these. Hypertree grids have properties that must be respected during traversal: depth-limiter 4.1.1 and bitmask 4.1.3. Because a hypertree grid is inherently a collection of hypertrees, it is only natural to iterate over these as a way to traverse it in its entirety. To traverse a hypertree grid, it is more than encouraged to use the following abstractions: *hypertree accessor and iterator*, *cursor* and *supercursor*.

4.2.1 Hypertree accessor and iterator To allow direct access or iteration through the hypertrees we introduce the *hypertree accessor* and the *hypertree iterator*, which are illustrated in Figure 4.1:

`GetTree(`n_{HT}`)`: From a defined hypertree grid, one can access directly the n_{HT} hypertree.
`GetNextTree()`: An hypertree iterator can be created from a hypertree grid, and GetNextTree() is then called to iterate through all the defined hypertrees, thus avoiding undefined hypertrees.

4.2.2 Cursors Thanks to the hypertree obtained from either *hypertree accessor or iterator*, it is possible to ask the hypertree grid object to create what we call a cursor. This cursor will only operate the hypertree it has been created on.
We thus introduce the following object:

Definition 4.3 *A* cursor *is a structure pointing to a vertex of a hypertree of a hypertree grid, that can both access its vertex attributes and move from its current vertex to another connected vertex.*

Any cursor will hold at least the following information: a reference to the hypertree it is traversing and the identifier of the vertex it is pointing to. From this information it is possible to retrieve the associated Global Index n_g (cf. Definition 4.2). Also it is to be noted that by design cursors are richer than *iterators* in the common sense as they do not impose a predetermined traversal scheme.
Displacement operators might be available in each cursor, illustrated in Figure 4.1, such as:

`ToChild(`*ichild*`)`: descend into the vertex with child index *ichild* (cf. 3.2), respecting depth-limiter (cf. 4.1.1) and bitmask 4.1.3, relative to the vertex currently pointed at, except if already at a leaf.
`ToParent()`: move one vertex up in the tree, except if already at the root.
`ToRoot()`: move to the root of the tree.

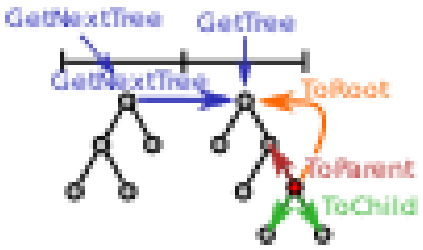

Fig. 4.1 Illustrating the different ways of operating on a hypertree grid. GetTree(n_{HT}) or GetNextTree() operations allow to get a tree from the hypertree grid, and `ToRoot()`, `ToParent()` or `ToChild(`*ichild*`)` operates on a single hypertree.

We subsequently define several frgrances to meet different needs and allow for efficient implementations. Like perfumes, concrete cursors types are created from a mixture of fragrances. For instance, the *Orientation* fragrance, i.e. the `Oriented` or `NonOriented` values, expose the following displacement operators:

`Oriented`: `ToChild(`*ichild*`)`.
`NonOriented`: `ToChild(`*ichild*`)`, `ToParent()` and `ToRoot()`.

Other fragrance might be introduced when needed to provide the set of operations to implement algorithms. One might want to take into account the geometry, i.e. coordinates of cell center and size, as well as orientation or other fragrances. From the displacement operations described previously, one can easily implement usual tree

traversal such as Depth-First-Search (DFS) or Breadth First Search (BFS).

Remark 4.1 Due to the different displacement operations they will implement, the Oriented cursor will be more efficient than Non Oriented version for simple traversal since it naturally makes fewer operations. Thus one should really pay attention to choose the appropriate cursor answering needs. Picking a more complex cursor than needed will lead to inefficiency.

Remark 4.2 The AMR mesh and hypertree grid object do not provide mechanisms to directly access a cell. Cells are identified by the Global Index n_g, and we could get a direct access by building a costly mapping between n_g and (n_{HT}, n_l), hypertree identifier and local identifier within the hypertree. From this mapping a Oriented cursor might be created, allowing one to get local information on the cell, such as the simulation property value, and also descend in the desired children of this cell. However it would not allow to retrieve more complex information such as ascendance to get the parent or the neighborhood.

4.2.3 Supercursors Many visualization algorithms require neighborhood information to perform their computations. For instance the outside boundary extraction algorithm generates a boundary face if and only if it is not shared by two cells. In order to provide neighborhood information we devised and implemented the following compound structures:

Definition 4.4 *A* supercursor *is a cursor (cf. Definition 4.3) that keeps track of a neighborhood information based on the hypertree grid.*

Implementing a supercursor thus entails providing the logic necessary when operating on the hypertree to update the neighborhood information which might be spread across several hypertrees of the hypertree grid. When a supercursor is created, its central cursor references the cell root of the considered hypertree and neighborhood information corresponds to the different root cells of the neighbor hypertrees.
When a supercursor descends into a tree, the neighbors of a cell can be a mix of cells on the same level (coarse or leaf cells) or from previous levels because consequently being solely leaf cells. This is illustrated in the upper part of Figure 4.2: the yellow cell marked by the cross-shaped structure has four neighbor across its edges. The top and left neighbors of the cell are cells of lower level and consequently are leaves. The right and bottom neighbor are on the same level, and are respectively coarse and leaf cells. We note in particular that a neighbor might be referenced several times by the same supercursor. In this case, this neighbor is necessarily a leaf cell at the level above that of the supercursor. This is shown in the lower part of Figure 4.2, where the supercursor references the smaller yellow cell, while both top center and right neighbors are indeed the same referenced top right yellow leaf cell from the level above.

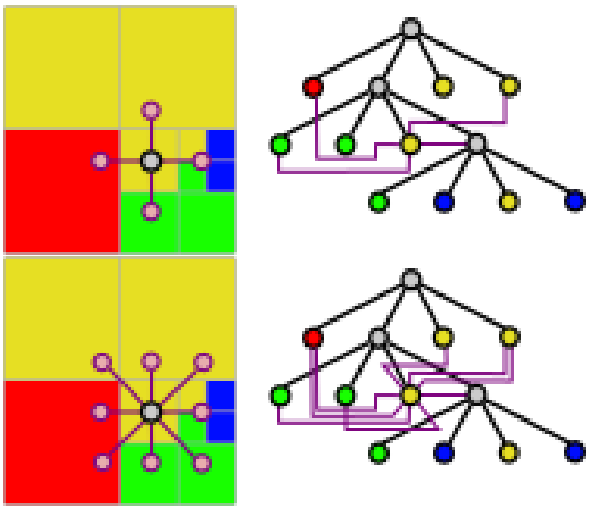

Fig. 4.2 Von Neumann (top) and Moore (bottom) neighborhood of a cell; to the left, in a 2-dimensional binary AMR mesh; to the right, corresponding neighborhood on the mapped hypertree object.

4.3 Connection with existing non-AMR algorithms

Filters have been developed to exploit optimally hypertree grid capabilities, and at the time of writing hypertree grid-aware filters coverage is not as wide as filters coverage for unstructured meshes which can be seen as the historical reference.
Therefore, to help make hypertree grid exploitable for the wealth of visualization scenarios, one should be capable of producing an unstructured view of the hypertree grid. Depending on the targeted algorithms, different unstructured views can be generated from the strict nodes, see in Figure 4.3, and such views can be a complete representation of the mesh or a partial one.

4.3.1 Non-conforming unstructured primal view This view is trivially constructed by iterating with a cursor, thus generating quad cells in 2D or hexahedral cells in 3D. A cursor with the *Geometry* fragrance must be used, providing access to origin coordinates and size of each node. As stated in 2.1, AMR has T-Junctions and so the resulting view will be non-conforming.

4.3.2 Conforming unstructured primal view
In order to resolve the difficulties posed by AMR T-junctions it is possible to generate a conforming primal view by involving polygonal or polyhedral cells (i.e. generic), based on quad or hexahedron surface with added points on edges or faces for each AMR T-Junction. To be noted, it is not trivial to identify which points to add. The more differences in level there are between cells, the

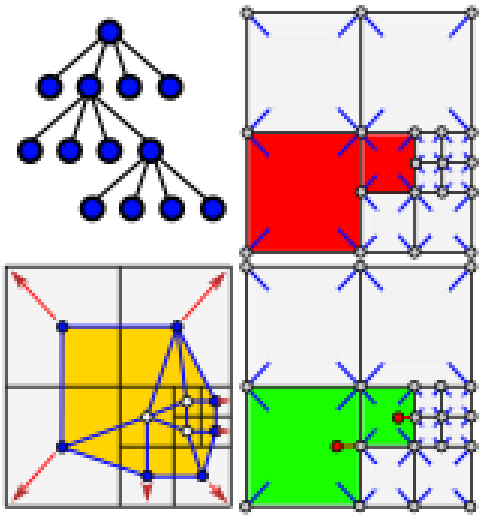

Fig. 4.3 Four representations of the same AMR mesh ($d = 2, f = 2$): top-left: a tree view describing all AMR cells, both coarse and leaf ones; top-right: a non-conforming unstructured primal view using either quadrilaterals or hexahedra (in green) to describe leaf cells; bottom-right: a conforming unstructured primal view using generic cells (in red) for describing leaves with the red dots that indicate the additional nodes; bottom-left: a conforming unstructured dual view (in yellow) with red arrows indicate how the dual vertices may be moved to produce an adjusted dual. Nodes having a blue line from then to a mesh all participates in the description of this cell. The non conformity of a mesh is seen as soon as a cell is not described by all the nodes lying on its circumference.

more T-Junctions are to be encountered. Moreover, manipulation and treatment of generic-cell based meshes are more costly than their non-generic counterpart.

4.3.3 Conforming unstructured dual view Another way to manage T-junctions is to generate a conforming unstructured dual view, defined as follows.

Definition 4.5 *Given $d \in \{1; 2; 3\}$ and a d-dimensional mesh $\mathcal{M}$, referred to as the* primary mesh*, we define its* dual mesh $\mathcal{M}^*$ *as follows:*

(i) to every d-dimensional cell $e \in \mathcal{M}$ is associated a dual vertex $v^* \in \mathcal{M}^*$*, located at the center of mass of e;*
(ii) to every vertex $v \in \mathcal{M}$ is associated a dual cell $e^* \in \mathcal{M}^*$*, whose vertices are exactly the dual vertices $v_i^* \in \mathcal{M}^*$ such that v is a vertex of e_i.*

Depending on the value of d, dual edges *and* dual faces *are defined as the* 1 *and* 2 *dimensional elements of the dual cells, respectively.*

When replacing the primal AMR mesh with its dual or its adjusted dual (which does not modify the topology), then the problem of T-junction vanishes.
Original AMR mesh has its value cell-centered and dual projection put the corresponding values on dual vertices. Thus, filters designed for vertex-centered attributes (i.e., isocontour) can natively operate on the original simulation values. Compared to primal approach, there is no projection of the cell-centered attributes to vertices and the visualization results are less smoothed.

4.4 Hypertree Grid-aware Algorithms

When designing hypertree grid-aware filters one should pay attention to instantiate the proper (super)cursor(s) to avoid the use of an unnecessary costly iterators. For some filters it might even be very interesting to adopt a multi-traversal strategy, where cursors of increased complexity might be used, and maybe some pruning involved in the preliminary traversals. To prune the trees it is advised to use the bitmask. By relying on (super)cursors, filters development is greatly simplified, and hypertree grid properties such as dept-limiter are propagated throughout the filters.
Filters should try to take advantage of the hierarchical properties of hypertree grid. For instance, non-strict nodes have an associated value in the simulation properties, and so a filter could modify these values depending on the goal to achieve.
Also, one should take advantage of the global index to optimize filters memory-wise. This is especially useful for sources and readers, where reordering the simulation properties can be avoided, or strict nodes associated with the same index in the simulation properties thus compressing the data.
We note that some hypertree grid filters may rely on already available algorithms taking non-AMR input. In this case and for the sake of efficiency, instead of creating the whole non-AMR input, i.e. an unstructured mesh, these filters should *stream* [24] it, one unstructured cell at a time, in order to lower memory consumption.

5 Method

After having established the necessary foundations for our work, we now discuss the methodology that we used to turn this theoretical framework into an actual implementation. While §3 can be understood as a frame of reference that shall not evolve much in the future, the concrete methods discussed below are, by nature, subject to further improvements or revisions. In particular, we have completely revised our approach to utilizing the dual, as well as the design of supercursors, with respect to our our earlier presentation [12]. We begin by describing our methodological choices for efficient representation and indexing of hypertrees and hypertree grids.

5.1 The Compact Representation

Using strict nodes entails a relative memory overhead whose ratio is within $[\frac{1}{f^d}; \frac{1}{f^d - 1}]$, tending towards the

upper bound of this interval as the number of nodes increases. It thus follows that the number of leaves dominates that of strict nodes. Which is why we sought to implicitly define the leaves, while explicitly storing in memory only the strict nodes. Such a *compact representation* still allows for traversal, at the cost of minimal additional processing when visiting the leaves due to their implicitness compensated by fewer cache missing.

In order to fully describe a hypertree, it is therefore sufficient that each strict node store one index to refer the first amongst its children, called the *eldest* child node. It is indeed sufficient to only store a reference to the eldest when all children of a given cell are created as once as a block of contiguous indices, at the end of the node array, instead of allocating memory for each child which might be non-leaf cell itself. Furthermore, the size of this block is constant and equal to f^d, by definition of a hypertree. Using short integers with a size of four bytes (B), in order to refine nodes, and denoting $m \in \mathbb{N}^*$ the number of its strict nodes, describing the topology of a hypertree thus costs at a minimum $4m$ B for the indices of the eldest children. The overall efficiency of this approach is very sensitive to the topological structure of the tree and the order in which it is traversed at construction time. This is illustrated in Figure 5.1: when the topological structure of the tree is created in DFS order, some unnecessary allocations (namely, for nodes 4 and 5) occur; in contrast, an optimal traversal only allocates space for strict tree nodes. This worst case occurs when the last-refined cell is also the last entry during the penultimate refinement stage, and the cost for eldest children indices can be as high as $4[1 + (m-1)f^d]$ B.

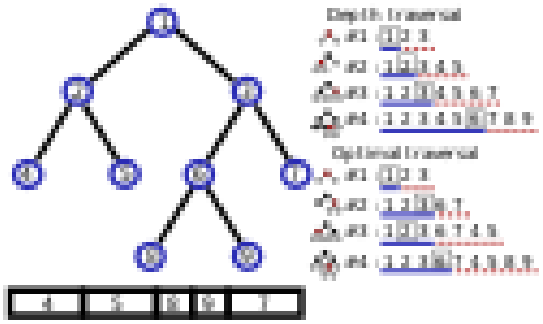

Fig. 5.1 Construction of a binary hypertree in dimension 1 with five leaves: the order in which it is performed impacts the total memory footprint. At each step (#), the index in a square is that of the nodes being refined; indices underlined with a solid (resp. dashed) line represent allocated (resp. implicit) nodes.

One thus obtains the following bounds for the memory cost $C(m)$:

$$4mB \leq C(m) \leq 4\left[1 + (m-1)f^d)B\right].$$

Note that these theoretical values cost are rarely attained. This lower bound is a theoretical memory footprint, with an ideal topology where all children, except one, of a strict node have the same type (either all strict nodes, or all leaves) and ideal implementation (the traversal strategy refines last all strict nodes than only have leaf children). Unfortunately, there is not a way to devise a traversal strategy that is optimal for all possible topological structure of trees.

When using $n \in \mathbb{N}^*$ hypertrees embedded inside a hypertree grid, we obtain this new for the memory cost $C(m, n)$

$$4mB \leq C(m, n) \leq 4\left[n + (m-1)f^d)B\right].$$

On the other hand, the memory footprint relative to the description of the spatial grid, using double precision floats to store coordinates, is least equal to $8\,(3 + d\sqrt[d]{n})$ B for a square or cubic grid, and at most $8(d + n + 2)$ B for a linear grid.

However, all aforementioned theoretical lower bounds are difficult to attain. But as the lowest possible bound for a given topology may be attained with an optimal implementation, it is thus the developer's responsibility to decide whether additional CPU processing is acceptable in order to achieve a better memory footprint, with potentially considerable gains. For instance, in the octree case ($d = 3, f = 2$), the memory gain factor between this AMR description and its explicit, unstructured all-hexahedral equivalent ranges within $[14; 112]$.

5.2 The Global Index Map

One possible choice to build a concrete $\Gamma_{d,f,E}$ is to combine a 0-*level indexing* of the constituting hypertree roots with the child index maps in each of these hypertrees. For instance, the 0-level indexing can be the lexicographic order applied to $[\![E]\!]$. One can then set

$$\Gamma_{d,f,E}(n_1; i; j; k) = n_1 + S_{i,j,k}$$

where $S_{i,j,k}$ is the *global index start* of the hypertree object at position i, j, k in the Cartesian grid of hypertree objects. By construction, the restriction of $\Gamma_{d,f,E}$ to any particular hypertree, being piece-wise affine with unit slope, is strictly increasing over $\mathbb{N}$ and therefore injective. Therefore, if the $S_{i,j,k}$ are chosen so that there be no overlap across the image spaces of these per-hypertree restrictions, then $\Gamma_{d,f,E}$ as a whole is injective and thus satisfies the specification of Definition 4.2.

In this setting, the global index start of each constituting hypertree only needs to be stored as an integer offset, at the cost of 8 B per hypertree. In practice, this can be achieved by constructing the hypertree grid one hypertree object at a time, and incrementing the global index start when moving to the next hypertree with the number of vertices in the last constructed hypertree.

It is important to note, however, that this method to build the global index map by means of assigning a global index start per constituting hypertree is in no way mandatory. Rather, our implementation provides the ability to specify an arbitrary version of $\Gamma_{d,f,E}$; it is the responsibility of the developer to ensure that this map complies with the requirements of Definition 4.2. In addition, such an explicit definition increases the total memory footprint by the cost of representing as many integers as there are vertices in the hypertree grid.

5.3 A Hierarchical Approach to Cursors

We now justify why and how the concept of cursors, introduced in §4.2, naturally lends itself to efficient tree traversal. We propose to illustrate with following algorithm: one may request cursor creation by using the index of a hypertree, *indHT*, as done, in Algorithm 5.1. Subsequently, the created cursors may be used to perform a hypertree grid traversal such as DFS-type (Depth-First-Search) as described in the Algorithm 5.2. In order

Algorithm 5.1 `InitializeDFS`$(htg, indHT)$

```
1: cursor ← NewNonOrientedCursor(htg, indHT)
2: ToProcessDFS(cursor)
```

Algorithm 5.2 `ToProcessDFS`$(cursor)$

```
1: if ¬IsLeaf(cursor) then
2:    for all iChild ∈ [[f^d[ do
3:       ToChild(cursor, iChild)
4:       ToProcessDFS(cursor)
5:       ToParent(cursor)
6:    end for
7: end if
```

to meet different needs and to allow effective implementations, several fragrances, introduced in §4.2.2, are already defined with the methods necessary for accessing to this enriched information built on the fly:

1. *Movement (no default)*
   `Oriented`: *ToChild(ichild)*.
   `Non Oriented`: *ToChild(ichild)*, *ToParent()* and *ToRoot()*.
2. *Geometric (default not explicit expressed)*
   `Geometric`: *GetOrigin()*, *GetSize()*, *GetBounds()* (bounds box including cell), *GetPoint()* (center cell).
3. *Topological (non default)*
   `Cursor`: No neighbors descriptions.
   `VonNeumannSuperCursor`: Neighbors in $1-d$ by points, $2-d$ by edges, in $3-d$ by faces
   `MooreSuperCursor`: Neighbors by points, edges and faces
4. *Neighbors (default not explicit expressed with all cursors contain a Geometric fragrance)*
   `Light`: Only the center cursor contains a *Geometric* fragrance

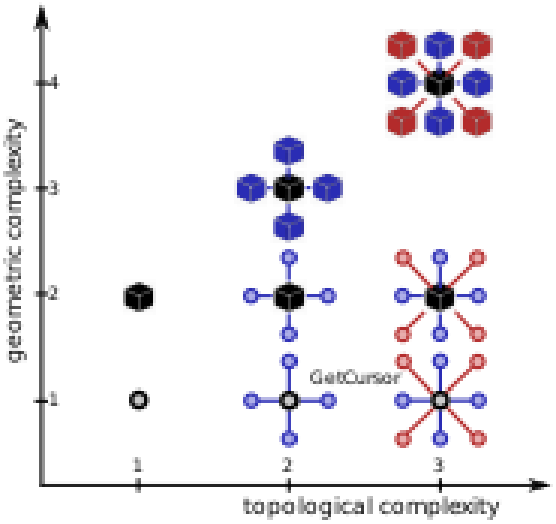

Fig. 5.2 This figure describes the subsets, where the horizontal axis distinguishes, left to right, between 3 increasing levels of complex topological information (single, Von Neumann and Moore); meanwhile, the vertical axis separates, from bottom to top, between the 4 different levels of geometric information (without, just center cursor, Von Neumann neighbors cursor and, for finish, Moore neighbors cursor) that could conceivably be needed by hypertree grid filters.

Apart from the specifics of the cursors linked to their fragrances, the latter have some interesting common abilities:

(i) To suggest to create a new hypertree from an index when creating or initializing a cursor;
(ii) To choose to refine anytime, just the center cursor for supercursors.

Like perfumes, each proposed concrete cursor type is created from a mixture of fragrances and named by concatening the fragrances' name used. Only the types of cursors that are required for our needs are currently implemented, as follows:

```
OrientedCursor
NonOrientedCursor
OrientedGeometricCursor
NonOrientedGeometricCursor
NonOrientedVonNeumannSuperCursorLight
NonOrientedMooreSuperCursorLight
NonOrientedVonNeumannSuperCursor
NonOrientedMooreSuperCursor
```

The geometric and topological properties of these concrete cursor types are summarized from a qualitative vantage point in Figure 5.2.

It is of course permissible to define new fragrances. Similarly, should other types of connectivities arise in the future, these would readily find their place along the topological complexity scale. This ability to extend the functionality of the cursors allows us to refine the algorithms to optimize the execution speed.

5.4 Supercursor Traversals

In the case of cursors that are not supercursors, both methods are relatively easy to design, and are therefore not discussed in detail here. Furthermore, the `ToRoot()` method for *Movement* fragrance is also rather simple to implement for supercursors, given the Cartesian layout of a hypertree grid at the root level.
The matter is more complicated for `ToChild(`*ichild*`)`. However, because all neighborhood cursors must be updated upon descent of the supercursor into a child node. This descent on each of these cursors must take into account the depth-limiter and bitmask properties of hypertree grid that could have been positioned. This update cannot be done *a priori*, because neighborhoods no longer have a Cartesian grid structure as soon as depth is non-zero. Instead, the neighborhood of a child must be explicitly computed from that of its parent. This task may seem daunting at first glance, but we devised an approach based on pre-computed *traversal tables* that greatly facilitates these updates.
Given a supercursor s pointing at a cell C, each of the children of this cell are uniquely identified by their respective child index $i \in [\![f^d]\!]$, as explained in Definition 3.2. Now, given f and d, there exists a unique mapping from the entries of the supercursor of child C_i into those of its parent C.

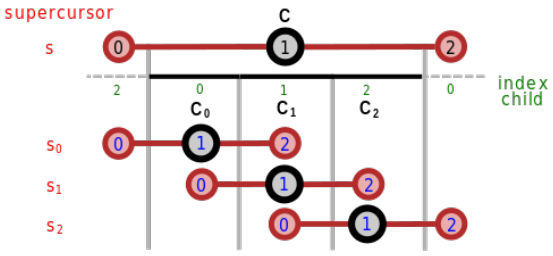

Fig. 5.3 Supercursor transformation through to child when $d = 1$ and $f = 3$: parent cursor indices (black), child indices (green), the three result for each transformation through to child for a child index 0, 1 and 2 with child cursor indices (blue). s is a supercursor pointing at a cell C, s_i is a supercursor pointing at a cell C_i, child i of C.

5.4.1 One-Dimensional Case Consider for instance the 1-dimensional case, where the Von Neumann and Moore supercursors are identical, with both having 3 cursors. Figure 5.3 illustrates this case, with a solid black line representing a coarse cell C divided with 3 children cells (C_0, C_1, C_2); child indices are indicated in green. Potential neighbor cells are shown on both sides with dashed lines; child indices adjacent to the cell of interest are labeled as well. In the same figure are also pictured the supercursors centered at C (above the line) and those centered at each of its children (below).
Child Cursor to Parent Cursor Table: In this case, the cursor with index 0 (i.e., pointing to the left) of the supercursor s_0 centered at child C_0 will point towards to either the same cell as cursor with index 0 of the supercursor s centered at parent cell C, or to one of its children. Meanwhile, the two other cursors of s_0 will point to either the same cell as cursor with index 1 of s, *or to one of its children*. This logic thus yields the following map, between child and parent cursors, for child 0: $0 \mapsto 0$, $1 \mapsto 1$, and $2 \mapsto 1$, denoted $(0; 1; 1)$ in compact form. One can easily deduce the corresponding maps for the children of C with indices 1 and 2, by reading the blue indices of Figure 5.3 from left to right, mapping them to the cursor indices in black for the corresponding parent supercursor. When concatenated in child index order, these 3 maps provide the *child cursor to parent cursor table* for the case where $d = 1$ and $f = 3$, i.e. $(0; 1; 1; 1; 1; 1; 1; 1; 2)$. Note that the *or to one of its children* clause above may occur only when the cell towards which a cursor c of the supercursor is pointing is not a leaf.
Child Cursor to Child Index Table: As explained in §4.2.3, c cannot point to a cell of depth greater than that of C, but a supercursor centered at child of C can point to cell exactly at most one level deeper. When this situation occurs, c must be descended into the adequate child of the parent cell neighbor in order to retrieve the corresponding vicinity cursor of the child supercursor. For example, in Figure 5.3, if cell C_i, with $i \in 1, 2, 3$, towards which the cursor with index 0 in the supercursor of C is pointing is not a leaf, then the cursor with index 0 in the supercursor of child cell C_0 must point towards the child with index 2 of C_i. Another type of map is therefore required to perform the descent into the relevant children of coarse cells whenever necessary. In the current example, C is coarse, hence cursor with index 1 in s_0, (i.e., the center cursor) will point to C_0 itself, i.e. to the child cell with index 0. Similarly, C being coarse, cursor 2 of s_0 will point at child with index 1 of C. We hence obtain the following map in compact form: $(2; 0; 1)$. The corresponding maps for children C_1 and C_2 are obtained accordingly, reading the green indices of Figure 5.3 from left to right, mapping them to the child cursor indices in blue for the corresponding child supercursor. When concatenated in child index order, these maps yield the child cursor to child index table for the case where $d = 1$ and $f = 3$, i.e. $(2; 0; 1; 0; 1; 2; 1; 2; 0)$.

5.4.2 Two-Dimensional Case For the sake of additional illustration, we provide the corresponding diagrams for

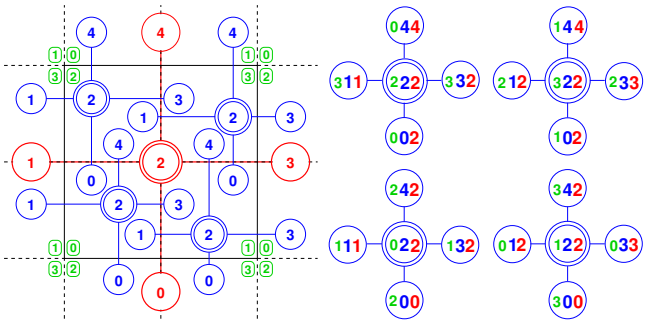

Fig. 5.4 Von Neumann supercursor $d = 2$ and $f = 2$: parent/child and child/child relationships (left), corresponding traversal table (right) with vicinity cursor indices in blue, child indices in green, and parent cursor indices in red.

the Von Neumann supercursor when $d = 2$ and $f = 2$ in Figure 5.4.

5.4.3 General Case Such schematics can be used to derive all traversal tables, for all types of supercursors and all possible values of d and f. Drawing on all possible cases would however be a tedious as well as error-prone task, so we implemented a Python script in order to generate the 2804 entries filling the 24 ($2 \times 2 \times 2 \times 3$) possible tables. Traversal table initialization can thus be performed only once at supercursor construction time, based on template parameter value and type of supercursor. This methodology thus ensures code correctness, as well as optimal execution speed for table entry retrieval is only a matter of random access in a small static arrays.

When endowed with these pre-computed tables, updating supercursors when performing a traversal becomes easy: given a supercursor s centered at a given coarse cell, all of its cursors are copied in temporary storage to avoid memory stomping as cross-permutations will occur. Then, given a child index i, for each cursor index j the corresponding cursor index k in the parent supercursor is retrieved from the child cursor to parent cursor table. The cursor with index k of the parent supercursor, previously copied as $c[k]$, is assigned to the child cursor and if $c[k]$ points at a leaf, then the update is complete. However, if $c[k]$ points to a coarse cell, it must be descended into, using the appropriate child index retrieved from the child cursor to child index table.

This scheme is summarized in Algorithm 5.3. We explicitly distinguish between the `SuperCursorToChild`(c, i) and `CursorToChild`(s, i) methods in order to emphasize that this method is *not* recursive: when descent into a child is required, it is only performed on a cursor of the supercursor, not on the supercursor itself. Note that this formulation of the algorithm ignores, for the sake of legibility, everything that regards the geometric updates that must also be performed.

Algorithm 5.3 `SuperCursorToChild`(s, i)

```
1:  n ← GetNumberOfCursors(s)
2:  for all j ∈ [[n[[ do
3:      c[j] ← GetCursor(s, j)
4:  end for
5:  C ← GetChildCursorToChildTable(s, i)
6:  P ← GetChildCursorToParentCursorTable(s, i)
7:  for all j ∈ [[n[[ do
8:      k ← P[j]
9:      GetCursor(s, j) ← c[k]
10:     if ¬IsLeaf(c[k]) then
11:         CursorToChild(GetCursor(s, j), C[j])
12:     end if
13: end for
```

5.5 The Virtual Dual

Our approach is to use duality as a natural means to process conforming cells when necessary for the considered visualization technique, while adding the two following design requirements:

(i) ready access to individual dual cells when required,
(ii) storage of the entire dual mesh is prohibited.

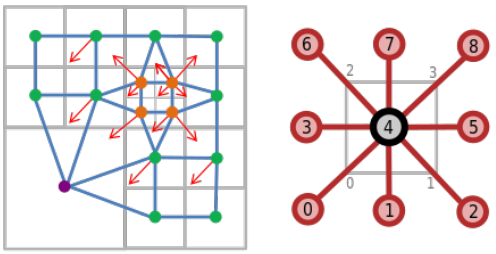

Fig. 5.5 Left: a 2-dimensional tree-based AMR mesh $\mathcal{M}$ (gray), overlaid with $\mathcal{M}^*$ (blue), showing dual cell ownership by primal vertices with orange arrows; right: cursor indices in a 2-dimensional Moore supercursor, used as tie-breakers for dual cell ownership amongst the deepest primal cells.

In order to avoid cell replication in the dual grid, our method assigns ownership of dual vertices to a single leaf, amongst the 2^d that may be neighbors of a primal vertex in dimension d: specifically, ownership of the dual cell is assigned to the deepest of these leaves, breaking ties in favor of the one that has the greatest *cursor index* relative to the others. Specifically, the function that determines ownership of the dual cell at any corner $i \in [\![2^d[\![$ of an arbitrary leaf cell, at which a Moore supercursor s is centered, is explained in Algorithm 5.4 and illustrated in Figure 5.5, left. The 3-dimensional integer array called `CornerNeighborsCursorTable` is a table that provides, given a d-dimension Moore supercursor and a corner index, the indices of the cursors that surround said corner. that surround centered at a cell is a corner-to-leaf traversal table to retrieve the 2^d indices of all the cell cleaves

Algorithm 5.4 `IsOwner`(s, i)

```
δ ← GetDepth(s)
ω ← (3^d − 1)/2
for all j ∈ ⟦2^d⟦ do
    k ← CornerNeighborCursorsTable[d][i][j]
    c ← GetNeighbor(s, k)
    if Masked(c) ∨ ¬IsLeaf(c) ∨ (k > ω ∧ GetLevel(c) = δ) then
        return False
    end if
end for
return True
```

touching a given corner of a given cell. For example in dimension 2, as illustrated in Figure 5.5, right, we have:

$$\texttt{CornerNeighborCursorsTable}[2][0] = \{0; 1; 3; 4\},$$
$$\texttt{CornerNeighborCursorsTable}[2][1] = \{1; 2; 4; 5\},$$
$$\texttt{CornerNeighborCursorsTable}[2][2] = \{3; 4; 6; 7\},$$
$$\texttt{CornerNeighborCursorsTable}[2][3] = \{4; 5; 7; 8\}.$$

Although this method keeps the additional memory footprint at the strict minimum, by avoiding dual cell duplication. Our methodology consists of utilizing a *virtual dual*, of which only one cell can be stored at any point in time. Provided an efficient way to generate, on demand, such individual cells from the virtual dual can be devised, then all memory footprint problems will vanish. Meanwhile, and by the same token, it will remain possible to apply visualization techniques that must, by design, operate on the cells of a conforming mesh. In this goal, we retained from our earlier approach the notion of dual item ownership, with the subtle yet important difference that it is expressed in terms of primal cell (and hence dual vertex) ownership of dual cells. This trade-off comes obviously at the price of added computational cost for the benefit of memory footprint, as the dual is not computed and stored once and for all.

5.6 Filters

We now discuss our methodology to *filtering*, i.e. applying visualization and data analysis algorithms, to hypertree grid objects. We begin with the case of geometric transformations, which can be especially efficiently addressed thanks to the notion of geometric embedding. We then explain our two-pass approach, based on a preselection stage, used to improve execution speed for those algorithms that rely on heavyweight supercursors. This section closes with a high-level description of the currently implemented filters, whose choice was dictated by actual analysis needs rather than for the sake of academic interest, and how they relate to the previously discussed cursors and supercursors.

5.6.1 Geometric Transformations We recall that, as defined in §3.2, we can represent the geometry of any arbitrary rectilinear, tree-based AMR by means of the 3-dimensional embedding $(\overrightarrow{x}; \overrightarrow{s}) \in (\mathbb{R}^3)^2$ of its hypertree grid equivalent. Because we restricted ourselves to the case of axis-aligned geometries, not all geometric transformations can be represented with this model: for example, a projective transformation will not transform, in general, a rectilinear hypertree grid into another. In fact, not even all affine transformations are suitable: as a result of our choice to only support axis-aligned grids, arbitrary rotations cannot be supported within our current framework either. Nonetheless, restricting possible transformations to that preserve alignment with the coordinate axes entails no loss of generality because, the AMR grids we aim to support are assumed to be axis-aligned by design (cf. §3.2).
For example, it is easy to see that all axis-aligned reflections, i.e., symmetries across a hyperplane that is normal to one coordinate axis, comply with the requirements above, being affine and preserving parallelism with all coordinate axes. We call `AxisReflection` such a transformation filter in our nomenclature. Furthermore, the reflection across a hyperplane in dimension $d \leq 3$, that is normal to axis $i \in [\![0; d[\![$ and has coordinate $\omega \in \mathbb{R}$ can be embedded in dimension 3 as follows:

$$\begin{array}{rccc} r_{i,\omega} : & \mathbb{R}^3 & \longrightarrow & \mathbb{R}^3 \\ & (x_0; x_1; x_2) & \longmapsto & (x'_0; x'_1; x'_2) \end{array}$$

where

$$\forall k \in \{0; 1; 2\} \quad \begin{cases} k = i \Rightarrow x'_k = 2\omega - x_k, \\ k \neq i \Rightarrow x'_k = x_k. \end{cases}$$

It thus follows that the image by $r_{i,\omega}$ of the 3-dimension geometric embedding of an hypertree object is

$$r_{i,\omega}(\overrightarrow{x}; \overrightarrow{s}) = (r_{i,\omega}(\overrightarrow{x}); \overrightarrow{s}_{-i}),$$

where $\overrightarrow{s}_{-i}$ denotes the vector equal to $\overrightarrow{s}$, save for its i-th coordinate which is opposed to that of $\overrightarrow{s}$. Therefore, the geometric embedding of the image by $r_{i,\omega}$ of an hypertree grid is exactly the collection of all image geometric embeddings of its constituting hypertrees. Axis-aligned reflection of hypertree grid objects can thus be implemented in a way that only operates upon the geometric embeddings using the very simple formula above. As a result, such an implementation is both extremely fast and memory efficient, for all it needs to do is create a new array of transformed coordinates along a single axis for the geometric embeddings of its constituting hypertrees. Meanwhile, the topological structures of said hypertrees only have to be shallowly copied.

5.6.2 Dual-Based Filters As explained in 5.5, we devised the concept of virtual dual, in order to extend the range of applicability of our original dual-based approach

to include large-scale meshes. The elements of this virtual dual are thus to be generated, processed and discarded at once as the filter traverses the input grid. In order to generate the dual cell associated with an arbitrary primal vertex (*corner*) as illustrated in Figure 5.5, left, a filter must be able to iterate over all primal cells sharing that corner.

Traversal of the input AMR mesh is performed over the vertices of the corresponding hypertree grid using cursor objects discussed in 4.2.2. Therefore, on-the-fly dual cell creation occurs by iterating over all corners of all input primal cells and, for each such corner, iterating over all primal cells having it as a corner. In dimension 2 for instance, there can be 2 across-edge neighbors and 1 across-corner neighbor to a primal cell that shares a given corner thereof. In dimension 3, three across-face neighbors can also exist, as well as an one additional across-edge neighbor. The cursor must thus provide Moore neighborhoods so that all those types of neighbors of a cell are made available when iterating around one of its corners.

In addition, when a dual cell must actually be generated, based on the ownership rules introduced in 5.5, its vertices are, by definition, located at primal cell centers (and possibly moved the primal boundary when dual adjustment is be performed). As a result, the cursor must also provide access to the geometric information of all neighbors. Both features, topological and geometric, are provided by the Moore supercursor which is thus required by all dual-based filters. This super-cursor is the most complex in our hierarchy of cursors, and every traversal operation onto it requires many operations, with a computational cost that becomes quickly prohibitive as input mesh size increases. This can result in losses in interactivity detrimental to the analysis process, or even in unacceptable execution times.

Fig. 5.6 Stages of a two-stage filter applied to one constituting hypertree within a binary hypertree grid. Left: pre-processing stage with post-order DFS traversal, using a lightweight cursor, selecting vertices check-marked in green. Right: main stage with pre-order DFS traversal with a heavier cursor, only across pre-selected vertices. The indices reflect the order in which vertices are processed by each stage.

In order to circumvent this difficulty, we devised a two-pass approach where a more lightweight cursor is used to traverse the entire mesh in a pre-processing stage, selecting only those cells that are concerned by the algorithm. The dual-based computation is thus only performed in the subsequent processing stage, where only those pre-selected parts of the grid are actually traversed by the most expensive Moore supercursor. Specifically and as illustrated in Figure 5.6, the pre-selection stage uses *post-order* DFS traversal, in order to propagate upwards per-branch selection (and possibly aggregated attribute information as well), whereas the main stage is performed with *pre-order* DFS fashion, immediately processing the pre-selected cells in the order in which they are reached when skipping non-selected branches. Albeit more complex in appearance, this two-stage approach can in fact be dramatically more efficient than a direct traversal of the input grid with the most complex cursor, provided a clever pre-selection criterion not requiring neighborhood information be contrived. The key success factor to this approach thus rests on devising a criterion that is easy to compute with minimal information and yet is discriminatory enough so as to avoid as many false positives as possible (while false negatives will result in an incomplete output).

5.6.3 Concrete Filters We now provide a brief overview of the filters we have developed so far, as concrete instances of our cursor-based general methodology. This list can, and most likely will, be extended as dictated by tree-based AMR post-processing needs.

`AxisClip`: clip, i.e., mask out all input cells that do not fulfill a geometric condition that can take three forms: hyperplane, rectangular prism (shorthand *box*), or quadratic function. In hyperplane mode, only those leaf cells that are either intersected by said hyperplane or wholly within a prescribed half-space that it defines are retained. A similar selection process occurs in box mode, based on whether cells are intersected by said box or located entirely in its interior. In quadratic mode, a leaf cell is retained if and only if said function takes on positive values at all corners of this cell. The hypertree grid output always has a bitmask even when the input does not.

`AxisCut`: produces a 2-dimensional hypertree grid output from a 3-dimensional input, comprising the intersection of all cells in the latter that are intercepted by an axis-aligned plane. The output has an associated bitmask only when the input has one.

`AxisReflection`: already presented as a geometric transformation filter exemplar in §5.6.1.

`CellCenters`: generates the set of points consisting of the centers of the leaf cells in a hypertree grid, with the option to make it also a polygonal data set containing only vertex elements.

`Contour`: computes polygonal data sets representing isocontours corresponding to a set of given values for the cell-wise attribute, using a dual-based approach with a pre-selection criterion discussed in detail in §5.7.

`DepthLimiter`: stops the descent into each of the constituting hypertrees whenever either a leaf or the requested maximum depth are reached; in the latter case, a leaf is issued to replace the reached node, and

Table 5.1 Cursors vs. filters: unless otherwise mentioned, check-marks correspond to the cursors used to iterate over the input grid, when needed (which is not always the case). d denotes the dimensionality of the input grid. Cursors are arranged left to right in increasing order of complexity. Here, they are all *NonOriented*.

		`Cursor`	`GeometricCursor`	`VonNeumannSuperCursorLight`	`MooreSuperCursor`
`AxisClip`		✓(output)	✓		
`AxisCut`		✓(output)	✓		
`AxisReflection`					
`CellCenters`			✓		
`Contour`		✓(pre-processing)			✓
`DepthLimiter`					
`EvaluateCoarse`		✓			
`Geometry`			✓($d < 3$)	✓($d = 3$)	
`ImageDataToHyperTreeGrid`		✓(output)			
`PlaneCutter`	primal		✓		
	dual	✓(pre-processing)			✓
`Threshold`		✓			
`ToDual`					✓
`ToUnstructuredGrid`			✓		

therefore all its descendants too. The output is a hypertree grid that has a bitmask only if the input does as well. The aim of this filter is to reduce the restitution time by limiting the traversal depth and by this way the number of treated cells. A counterpart is to make some coarse cells visible.

`EvaluateCoarse`: produces a shallow copied hypertree grid output with a new field obtained by copying the values of a field input, each coarse value having been calculated from children values by applying an operator. e.g., first child, minimum, maximum, sum, average.

`Geometry`: generates the outside surface of a hypertree grid as a polygonal data set, in particular for rendering purposes. Note that, already memory costly in dimension 3, this conversion into an unstructured mesh can, in dimension 2, create an output whose footprint may be several orders of magnitude larger than that of the hypertree grid input.

`ImageDataToHyperTreeGrid`: generates a hypertree grid output with an associated bitmask from a $2-d$ PNG image.

`PlaneCutter`: similar to the `AxisCut`, except that it can take an arbitrary plane as cut function, to produce a polygonal data set output. This filter has two modes of operation: primal or dual. In primal mode, both topology and geometry of the original leaf cells are preserved, hereby ensuring that no interpolation error may occur and that the cut planes extend to the primal boundary, at the topological cost of producing T-junctions wherever the cut plane intercepts an interface between cells at different depths.

`Threshold`: produces a hypertree grid output with an associated bitmask, even when the input does not have any, in order to mark out all cells whose attribute value is not within a specified range.

`ToDual`: generates the entire dual mesh, possibly adjusted. For the reasons developed in §5.5, this filter should never be used with sizable hypertree grid inputs, but only for prototyping or illustration purposes.

`ToUnstructured`: generates a fully explicit unstructured grid data set whose elements are exactly the leaf cells of the input hypertree grid, represented as rectangular prisms (i.e., lines, *quads*, or *voxels* depending on the dimensionality of the input). The output thus has exactly the same geometric support as the input; it is not a conforming mesh due to the presence of T-junctions. In addition, it is prohibitively expensive for sizable AMR meshes and shall thus only be used for prototyping or illustration purposes.

These filters are implemented using their respective minimal cursors within the set described in §4.2.2. The correspondence between filters and cursors is provided in Table 5.1; it is left to the reader to examine why these relationships are indeed both correct and minimal.

5.7 Isocontour

We conclude this methodological discussion by emphasizing the case of isocontour, because it is arguably one of the most widely used amongst all existing visualization techniques, while being especially difficult to perform on AMR grids – in practice, impossible when dealing with large grids if they must be converted to an explicit grid prior to isocontour. It is important to mention that we isocontour hypertree grid attribute fields by considering only their values at leaf nodes. This design choice is

made in order to simplify a complex problem. Note however that subsequent implementations could be allowed to take into account field values at strict tree nodes as well. That said, there is no known, efficient isocontour algorithm for general polyhedral meshes.

Instead, the canonical approach is to subdivide polyhedra into simplices which are subsequently isocontoured[1]. As discussed in §3, this approach is prohibitive in terms of memory footprint and execution time. It is therefore natural to consider a dual-based approach to tackle isocontour of hypertree grids. Therefore, as explained in §5.6.2, the computational efficiency of the algorithm rests upon a pre-selection criterion, deciding whether a cell may be intercepted by an isocontour without any information retrieval concerning its neighbors.

Given a hypertree grid $\mathcal{H}$ with n_v vertices and an array C of isovalues, our selection criterion defines $|C|$ Boolean arrays with length n_v called *sign arrays*. For every value in C with index $j \in [\![|C|]\!]$, the corresponding signed array is denoted S_j. The goal of each S_i is to capture the relative position of the field of interest at all tree vertices, with `True` (resp. `False`) when the cell-centered[2] value is greater (resp. smaller) than $C[j]$.

We also define another Boolean array, T, called the *truth array*, with length n_v as well. We note that T is global to the entire set of isocontours and is used to pre-select tree cells that will be immediately isocontoured by the main processing phase. Only one such T is used across all isocontours because a dual cell must be generated when required by at least one isovalue. Algorithm 5.5 summarizes the pre-processing stage, for every cursor position c inside the input hypertree grid $\mathcal{H}$. The goal of this function is two-fold: first, store the position of the attribute value of c relative to each of the isovalues in each of the S_i arrays; second, store the truth value at $T[c]$ to indicate whether c is intercepted by at least one isocontour. When c is coarse, $T[c]$ can be `True` only when c has in its descent at least two leaf cells with opposed signs; in this case, the $S_i[c]$ values are irrelevant. In contrast, when c is coarse and $T[c]$ is `False`, then its entire descent has the same sign, defining the value stored in $S_i[c]$; in this case, $T[c]$ as well as the $S_i[c]$ values are relevant. When c is a leaf, $T[c]$ is not meaningful and is assigned `False` by default; in this case, the $S_i[c]$ values are relevant. As required, Algorithm 5.5 needs neither geometric nor topological information, hereby allowing for the use of the lightweight `TreeGridCursor` for the pre-processing stage. We note that a single function T is used for all isocontours because a dual cell must be generated when required by at least one isovalue.

Subsequently, the contouring stage executes the function `RecursivelyProcessTree()`, described in Algorithm 5.6,

Algorithm 5.5 `RecursivelyPreProcessTree`(c)

```
1: if IsLeaf(s) then
2:   if ¬Masked(s) then
3:     for all i ∈ [[2^d]] do
4:       if IsOwner(s, i) then
5:         D ← GenerateDualCell(s, i)
6:         for all j ∈ [[|C|]] do
7:           I ←+ MarchingCube(D, j)
8:         end for
9:       end if
10:    end for
11:  end if
12: else
13:   i ← GetGlobalIndex(s)
14:   for all j ∈ [[|C|]] do
15:     for all k ∈ [[2^d]] do
16:       l ← GetNeighborGlobalIndex(s, k)
17:       if T[i] ∨ T[l] ∨ S_j[i] ≠ S_j[l] then
18:         for all k ∈ [[f^d]] do
19:           RecursivelyProcessTree(GetChild(s, k))
20:         end for
21:         return
22:       end if
23:     end for
24:   end for
25: end if
```

upon every cell of $\mathcal{H}$, using a Moore supercursor s. For each generated dual cell D and each contour value $\mathcal{C}[j]$, the call to `MarchingCube`$(D, \mathcal{C}[j])$ returns set of polygons (possibly empty) that is appended to the isocontour mesh $\mathcal{I}$. The MarchingCube [22] function is nothing more than that which exists and can be used by various contour filters for generate one or more isocontours that produce a unstructured mesh. Here, cell by cell, this function is applied after building an unstructured cell.

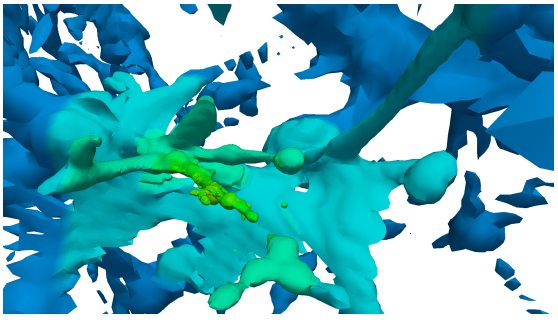

Fig. 5.7 Close-up view of an iso-surface generated by the native `isocontour` filter with a large hypertree grid input.

Figure 5.7 illustrates the results of the main isocontour phase, following the pre-processing stage, in the case of a large AMR simulation.

[1] provided the interpolation scheme be linear, an axiom which we make for the type of elements we want to support, and which therefore we will not discuss further here.

[2] or, for the sake of isocontour, considered as such.

Algorithm 5.6 `RecursivelyProcessTree`(s)

```
1: if IsLeaf(s) then
2:   if ¬Masked(s) then
3:     for all i ∈ ⟦2^d⟦ do
4:       if IsOwner(s, i) then
5:         D ← GenerateDualCell(s, i)
6:         for all j ∈ ⟦|C|⟦ do
7:           I ←+ MarchingCube(D, j)
8:         end for
9:       end if
10:    end for
11:  end if
12: else
13:   i ← GetGlobalIndex(s)
14:   for all j ∈ ⟦|C|⟦ do
15:     for all k ∈ ⟦2^d⟦ do
16:       l ← GetNeighborGlobalIndex(s, k)
17:       if T[i] ∨ T[l] ∨ S_j[i] ≠ S_j[l] then
18:         for all k ∈ ⟦f^d⟦ do
19:           RecursivelyProcessTree(GetChild(s, k))
20:         end for
21:         return
22:       end if
23:     end for
24:   end for
25: end if
```

5.8 Parallel Implementation

Even with this memory and computation efficiency, visualization of data from HPC simulation may require more than a compute node as large as it is, moving from sequential execution to parallel execution. The parallel word refers to autonomous processes that work on the same physical computer and interact with each other through message passing. The mesh is then distributed over these autonomous processes. A priori, the use of several processes must also improve the restitution time for a given problem, this can also be a solution when interactivity is not as smooth as desirable. However, efficiency being directly related to the quality of the mesh distribution, or more broadly to the balancing of computational loads. In fact, depending on the execution parameters or simply the choice of the algorithm, the cost per cell is very variable. The application of a filter will likely cause and load imbalance that would impact the following filters. Applying a rebalancing filter necessarily causes a memory overhead since the input mesh must be retained to potentially power other filters.
Some filters require information that goes beyond what is available locally. Often it is because the processing requires the data from neighboring meshes to the local domain, as for isocontour or gradient computation. A mechanism must be developed in order to locally copy to a process this information that is available to others. This approach, called the *ghosts cells* method is described in [13]. For a given mesh, this has the effect of growing the overall memory needed when increasing the number of processes. Currently, the distribution is the one given at the beginning and is not being challenged while running the processing pipeline which can be complex.
We acknowledge that other approaches to parallelization exist, such as the use of multiple threads, but are not discussed here because those can be considered as implementation details.

6 Results

We now discuss the main results obtained with the hypertree grid object, beginning with a study of its performance in terms of memory footprint. We continue with an overview of the filters that we have developed so far for this object with a focus on the massive memory savings allowed for by our approach based on separating geometry from topology through the analysis of axis-aligned reflection filter. Finally, we expose performance on various datasets of real-world data ranging from moderate to large-scale. These results are those obtained with our concrete implementation in VTK version 8.

6.1 Hypertree Grid Object

We begin with the case of a hypertree grid with 150 constituting hypertrees, used to represent a variable number of cells in an AMR mesh.

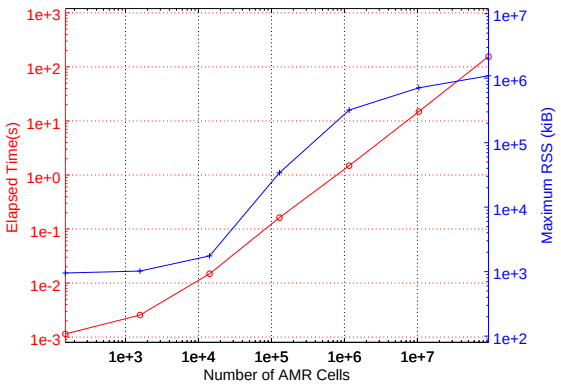

Fig. 6.1 Execution time (red) and memory footprint (blue) *versus* number of cells in a synthetic hypertree grid.

Figure 6.1 illustrates this case, when a varying number of cells is obtained by increasing the tree depth $\delta \in [1; 6]$. When $\delta = 1$, only root-level cells are present in the constituting hypertrees, resulting in relatively high memory fixed costs per hypertree; as δ increase, these costs are progressively diluted by the ensuing greater number of hypertree cells. In addition, we observe a linear speedup in terms of execution time, hereby demonstrating the scalability of our approach.

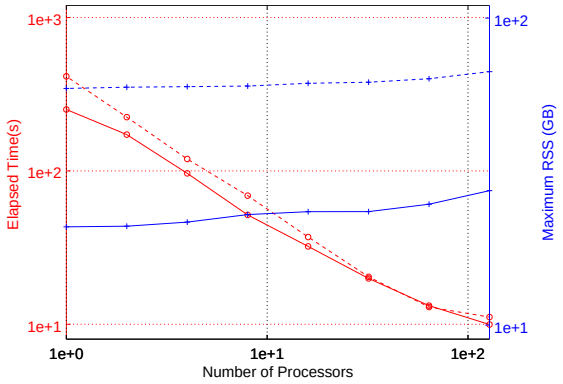

Fig. 6.2 Execution time (red) and memory footprint (blue) *versus* number of processors used to represent a 2D AMR mesh with $\mathcal{O}(10^8)$ leaves; solid (resp. dashed) lines correspond to a hypertree (resp. unstructured) grid.

Figure 6.2 demonstrates the strong scalability (i.e, with fixed total workload) of our approach, which scales almost optimally for memory footprint, and super-optimally for execution time, until maximum speedup is achieved for this problem size. Moreover, when comparing the performance in terms of memory footprint of our hypertree grid object with respect to that of using an unstructured grid representation, we note almost a full order of magnitude improvement (approximately a factor of 7). Furthermore, we have observed that this massive decrease in memory usage remains constant across a wide range of workload distribution schemes for highly refined meshes.

6.2 Hypertree Grid Filters

We now illustrate some results of the native hypertree grid filters presented in §5.6.3 and implemented in VTK, exploring the 2 and 3-dimensional cases as well as the two possible branch factor values, beginning with a 2-dimensional binary hypertree grid input, with a 2×3 layout of root cells to which is attached a single attribute field filled with the cell depths. Figure 6.3 applies the native hypertree grid `Geometry` filter (note that shrinkage of the output geometry is sometimes used in order to facilitate the interpretation of the results) to render hypertree grid outputs. In addition to it, these images illustrate the following filters:

(a) `Geometry`.
(b&c) `AxisReflection`, where hyperplanes are lines, respectively parallel to the vertical and horizontal axes, passing through the center of the hypertree grid.
(d) `DepthLimiter` with depth limit is set to 2.
(e) `CellCenters`, hooked to a glyphing filter to produce the black crosses shown at cell centers.
(f) `Contour` with attribute isovalues 1.25, 2.5, and 3.75. Note that, as explained in §5.5, the isocontours are topologically correct but do not intercept the primal boundary, because a non-adjusted dual is used.

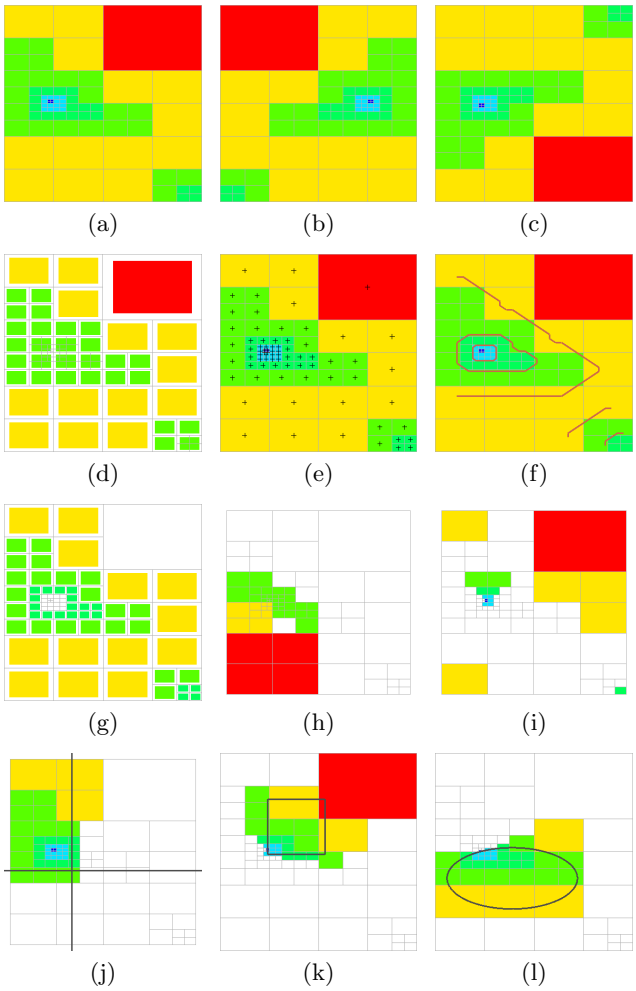

Fig. 6.3 Visualizations obtained by applying the `Geometry` (a), `AxisReflection` (b&c), `DepthLimiter` (d) `CellCenters` (e), `Contour` (f), `Threshold` (g), `ExtractSelected{Ids,Locations}` (h&i), and `AxisClip` (j–l) native filters to a 2-dimensional, binary hypertree grid.

(g) `Threshold` for attribute values within $[1; 3]$.
(h&i) `ExtractSelected{Ids,Locations}`.
(j–l) `AxisClip`, illustrated for each of its three modes of operation, respectively: hyperplane (here, with 2 consecutive appelications), box, and a quadratic corresponding to an axis-aligned ellipse; note that alignment with the grid axes is not required by the filter as any arbitrary quadratic can be specified.

The results computed by the same filters, but when a non-empty bitmask is attached to the hypertree grid input, are shown in Figure 6.4. We are not showing here the results obtained with the `AxisClip` filters in order to save space, but suffices to say that corresponding images are obtained as expected.
Of particular interest is the isocontour case (f): because the current implementation of the filter uses a non-adjusted dual, the computed isocontours exhibit additional geometric oddities in the vicinity of the non-convexities resulting from the presence of masked cells. In our typical,

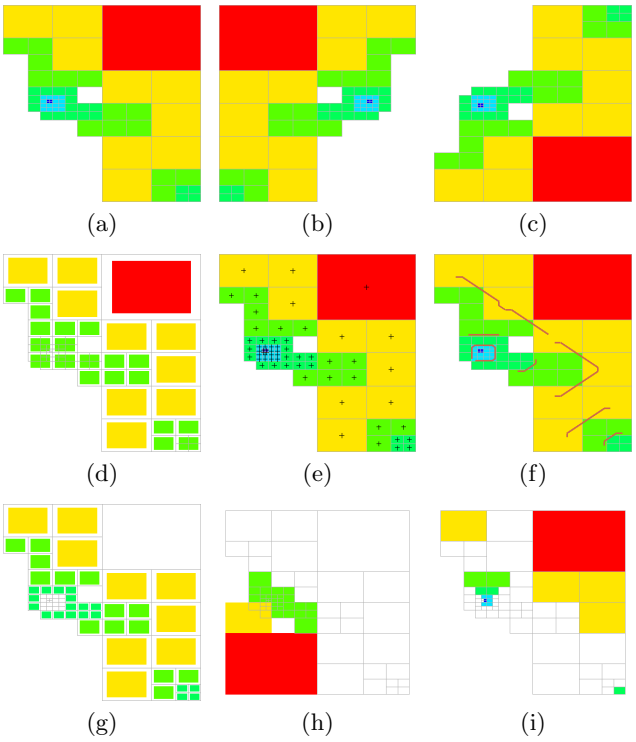

Fig. 6.4 Results of the same tests as in Figure 6.3 (a–i), when the input grid has a non-empty bitmask.

large-scale applications, the phenomena of interest which are searched for in the post-processing stage tend to be removed from object boundaries (external or internal); therefore, geometric error in computed isocontours are generally not encountered. It however remains our goal to provide the option to adjust the dual in future implementations.

The case of the `DepthLimiter` filter (d) also reveals an interesting feature, that can only present itself when non-convexities are present – and therefore, only when a non-empty mask is attached to the input hypertree grid. Specifically, the hypertree grid output by the filter can have a larger geometric extent that the input. This results from the fact that an input coarse cell at the depth limit is retained to create an output leaf as soon as *at least* one of its descendants is not masked. Indeed, this behavior can be observed in the figure, with the green cell at depth 2 located at the middle-left of the grid.

A 3-dimensional, ternary set of test cases is now used, with a $3 \times 3 \times 2$ layout of roots to further illustrate our point. In Figure 6.5, we show visualizations obtained with the following filters (note that `Geometry` is used to visualize all hypertree grid outputs):

(a) `Geometry`.

(b&c) `AxisReflection`, respectively with 1 and 2 successive reflections about planes passing through the center of the hypertree grid.

(d) `AxisCut` with two axis-aligned cut planes, whose 2-dimensional AMR outputs are shrunk for legibility.

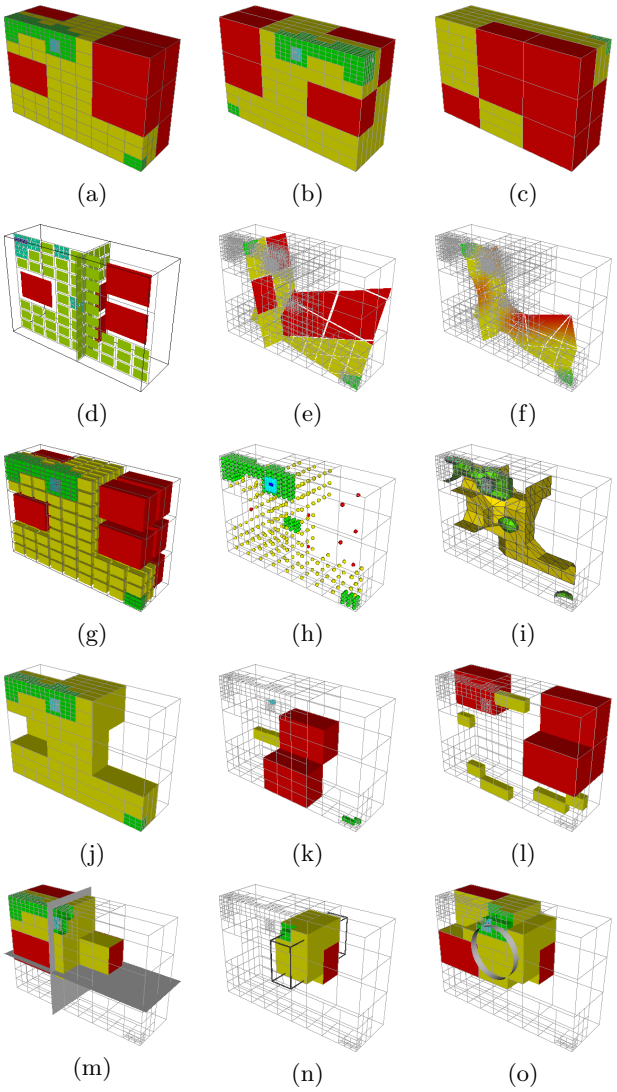

Fig. 6.5 Visualizations obtained by applying the `Geometry` (a), `AxisReflection` (b&c), respectively across one and two axis-aligned planes, `AxisCut` (d), `PlaneCutter` (e&f), respectively in primal and dual mode, `ToUnstructured` (g), `CellCenters` (h), `Contour` (i), `Threshold` (j), `ExtractSelected{Ids,Locations}` (k&l), and `AxisClip` (m–o) native filters to a 3-dimensional, ternary hypertree grid.

(e&f) `PlaneCutter` which, in contrast, produces polygonal data sets, respectively in primal and dual modes. The main benefit of the latter is its conforming mesh output suitable for subsequent post-processing requiring perfect connectivity, at the cost of being considerably slower than the former.

(g) `ToUnstructured`, whose all-hexahedral unstructured grid output is connected downstream to a shrink filter. As discussed in §2.1, the resulting unstructured mesh is not conforming.

(h) `CellCenters`, hooked to a glyphing filter to produce the spheres shown at cell centers, colored by depth.

(i) `Contour`, again with three isovalues.

(j) `Threshold`, for depth values within $[1; 3]$.
(k&l) `ExtractSelected{Ids,Locations}`.
(m–o) `AxisClip` with its three modes of operation: respectively, two successive clips with planes parallel to the grid axes, an axis-aligned box, and a quadratic associated with a cylinder.

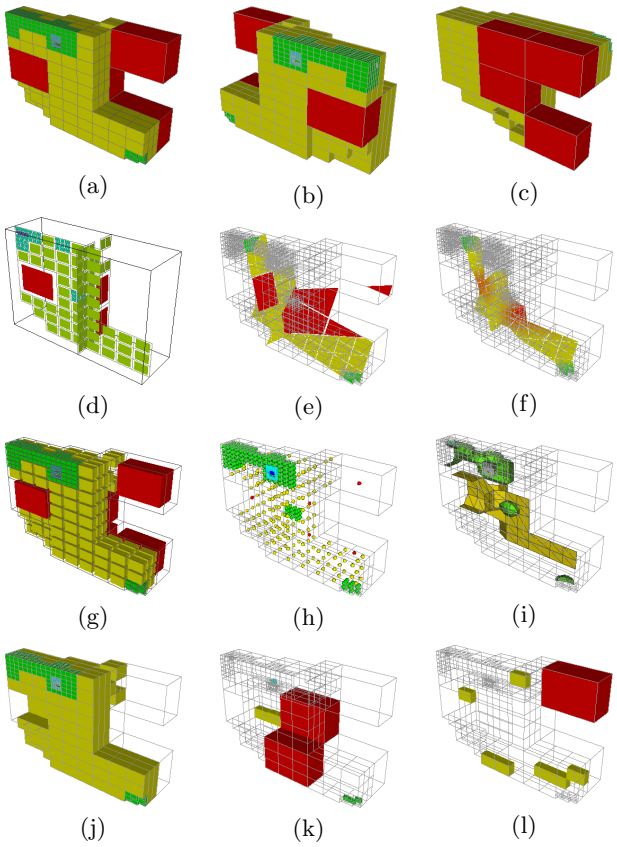

Fig. 6.6 Results of the same tests as in Figure 6.5 (a–l) when the input has a non-empty bitmask.

As previously done with the 2-dimensional cases, Figure 6.6 presents a subset of these cases, but obtained with a non-empty mask attached to the input hypertree grids. Comments similar to those made in the 2-dimensional, binary case can be made and we will not therefore repeat ourselves. The interested reader is invited to draw parallels between corresponding 2 and 3 dimensional sub-figures, and to inspect the contents of the test harness we implemented for all existing hypertree grid filters, across different dimensions, branching factors, and other modalities: to date, 58 individual tests are available and can be either executed as they are, or modified and experimented with at will.
We close this discussion with the particular case of the `AxisReflection` filter. In Figure 6.7, we illustrate the use of this filter by applying it to the case of a ternary tree-based AMR grid with $5 \times 5 \times 6$ root cells, where a bitmask is defined using a quadratic function retaining only those cells that are within or intersect a truncated octant of the unit ball. The experiment thus consisted in creating this object, hereafter referred to as the `octant`, then performing seven reflections adequately. Those were defined in order to produce outputs whose union, together with the initial `octant`, produces the unit ball truncated by the original plane and its symmetrical about the sphere center. This whose output is called `reflections`. Octant creation, geometry extraction and rendering times were excluded from this experiment in order to assess the performance of the `AxisReflection` filter in isolation.

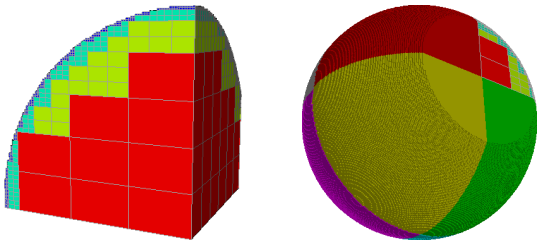

Fig. 6.7 Left: the first octant of a truncated unit ball, approximated with 5 levels of a ternary tree-based AMR grid with $5 \times 5 \times 6$ root cells. Right: a rendering showing the same truncated octant (upper right corner), together with its successive images (in solid colors) by axis-aligned reflections, yielding a truncated unit ball.

Table 6.1 Main characteristics, and memory footprints in terms of maximum resident set size, of a ternary hypertree grid object (`octant`), its 8-time replication (`octant*8`) and of its union (`reflections`) with seven images thereof by the `AxisReflection` filter.

	Number of cells	Number of leaves	Number of trees	**RSS (kiB)**
`octant`	128724	123962	150	**44924**
`octant*8`	1029792	991696	1200	**359392**
`reflections`	1029792	991696	1200	**45172**

The main results of this experiment performed on a single core are summarized in Table 6.1; in particular, the `reflections` represents an AMR mesh 8 times larger, with over one million cells (96.3% of which are leaf cells), than the original `octant`. Executing the 7 reflections took a negligible time, compared to the octant creation or its rendering, hereby confirming the theoretical prediction that, if correctly implemented, the reflection filter should have negligible execution time. Another key finding of this test was to measure a negligible increase in memory readings[3], as compared to the real replication of the object requiring a commensurate increase in memory footprint (which might not be available to

[3] We assess memory footprint in terms of maximum resident set size (RSS), indicating the amount of memory that belongs to a process and resides in RAM.

the target platform). These results demonstrate that our implementation fully delivers the promises of the theoretical analysis, in terms of execution speed as well as of memory footprint. As a result, all future hypertree grid structure-preserving geometry transformation filters shall be implemented following the same paradigm.

6.3 Performance results

In previous subsections we presented synthetic results and now we are expanding to real-world data from moderate to large scale, 2D and 3D cases. We will first focus on 2D use cases. First is a 100 million cells shock wave simulation data, generated with internal CEA simulation codes. Second is a 6 billion cells Mandelbrot calculation [19], [14] generated using well-known formula, and saved as an HyperTree Grid. Figures 6.8 and 6.9 illustrate the rendering of those. We have developed 2D optimized rendering which offers a smoother experience and also greatly reduced video memory usage compared to naive, full scene rendering. Table 6.2 summarizes the interactivity we get for this moderate to large scale 2D data sets. For more details on the optimizations we introduced thanks to the HyperTree Grid data structure, we invite the reader to turn to [16]. We used either the Tera-1000-1 CEA supercomputer nodes [1] or a 32 GB RAM workstation in order to render these images. The limiting factor was generally memory, and so as the shock wave data size is about 2 GB, rendering can be done on the workstation or one 32 cores, 128 GB of Ram Tera-1000-1 node. Regarding Mandelbrot calculations, it is a larger 200 Gbytes test case, and rendering could be achieved using one to two Tera-1000-1 supercomputer nodes. We now discuss some 3-dimensional

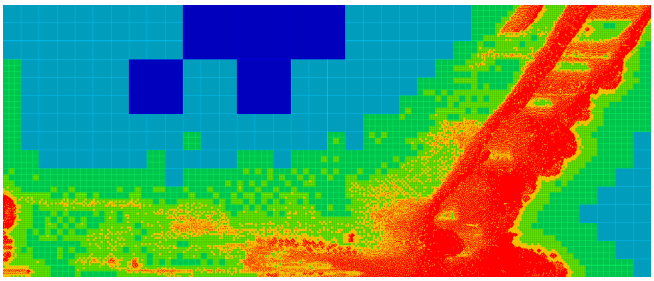

Fig. 6.8 Rendering 100 million cells 2D hypertree grid shockwave simulation data.

test cases results, for which we chose in particular the 3×10^8-cell "asteroid fall" simulation [23] provided by Los Alamos National Laboratory/NASA, a 1 billion cell structured "bubbles" simulation data set and a 72 billion cells astrophysics simulation of the universe provided by CEA-IRFU. Aside from the specifics of these cases, these datasets substiantally differ from each other, because:

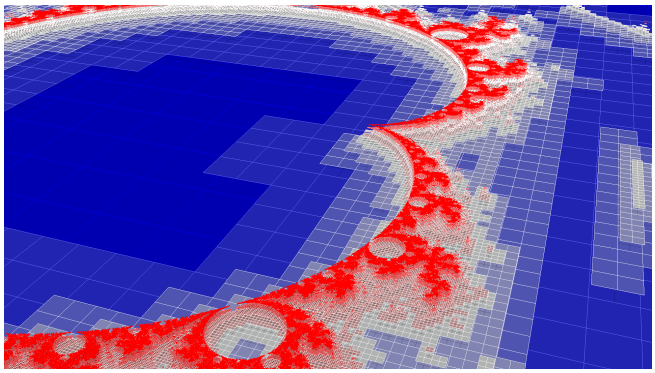

Fig. 6.9 Rendering 6 billion cells 2D hypertree grid Mandelbrot calculation data.

Table 6.2 Frame rates and video memory usage of 2-dimensional optimized vs. naive rendering. Screen resolution is 3440x1440.

	Optimized Rendering		Standard Rendering	
Shock100	8-20 fps	1 GB	2 fps	4 GB
Mandel6	1-5 fps	3 GB	0.05 fps	40 GB

- the "asteroid fall" simulation uses an unstructured mesh;
- the "bubbles" case is structured data;
- the "universe" case is a native hypertree grid data set.

Therefore, we had to convert offline the unstructured asteroid fall to an hypertree grid for further exploitation. In order to process the structured bubble expansion data, we added a hypertree grid streaming capability to our reader, as it is easy to add coarser cells on top of the final leaves representing the actual Cartesian grid. Some resulting visualizations are presented in Figures 6.10, 6.11 and 6.12.
In order to properly render the data sets, we respectively used one, two and sixteen "thin" 128 GB nodes of Tera-1000-1 supercomputer. We could also have used "fat" one TB nodes, hereby reducing the required hardware to respectively 1, 1 and 2 fat nodes. We noticed while conducting these experiments that data exploration remained interactive and smooth. Furthermore, converting the "asteroid" data to a native hypertree grid dataset lead to the following paper [15] where for the first time this dataset was exploited on a 16 GB laptop instead of several supercomputer nodes, at the full resolution of the simulation.

7 Conclusion

There are many more details to this story than we could possibly fit within the frame of a journal article. What are we, then, to make of this already long *exposé*, which encompassed general motivations, theoretical foundations, application methods, and experimental results?

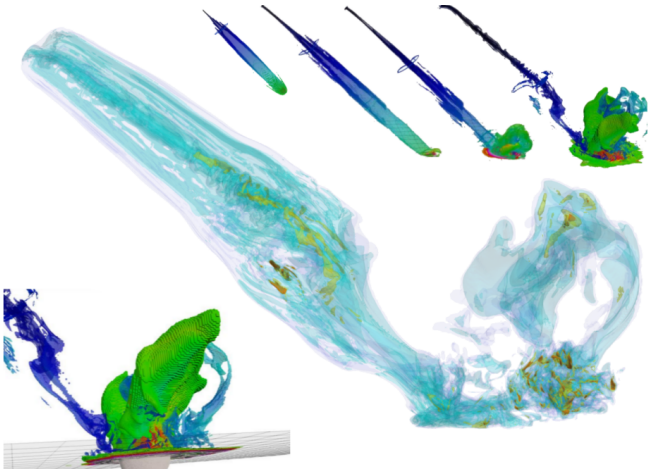

Fig. 6.10 Rendering 300 million cells 3D unstructured mesh asteroid fall simulation data, re-sampled to HyperTree Grid offline. Multiple isocontours are applied for the center mainly blue view, and outer views are generated using thresholding

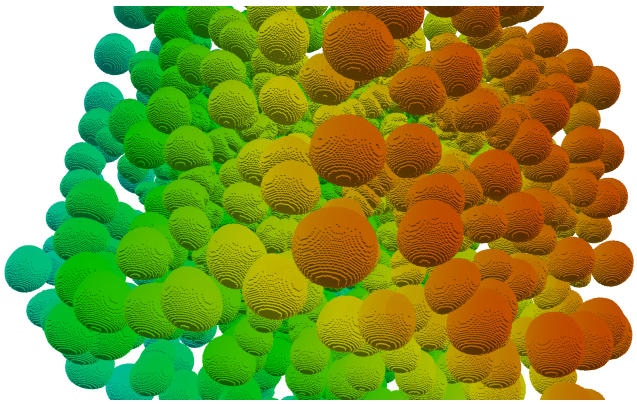

Fig. 6.11 Rendering 1 billion cells 3D structured bubble expansion calculation data, dynamically re-sampled to hypertree grid. No filtering applied, we simply select the "bubble" material to render it.

In the next few lines, we will first look back on what has been achieved so far, as compared to what we were initially envisioning. This will allow us to conclude with a set of remarks as to our subsequent projects and goals, articulating them within our general vision together with what we have discovered and done during the course of the work described in this article.

7.1 Main Findings

We set out in §2.2 ([a]) with the goal to propose a novel VTK data object that would be able to support all conceivable types of rectilinear, tree-based AMR data sets, based not only on today's software but also on what we can foresee of tomorrow's extreme-scale simulations. We can confidently claim that we have accomplished this first goal, based on the `vtkHyperTreeGrid` object and

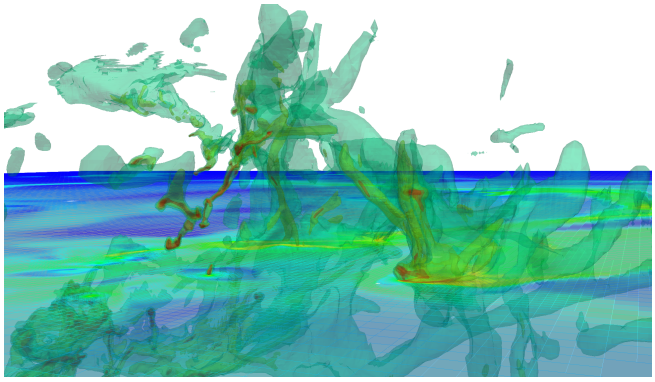

Fig. 6.12 Rendering 72 billion cells 3D hypertree grid universe simulation data. Multiple isocontours are applied to get the 3D shapes, and an AxisCut provides the reference plane.

its family of lesser objects presented throughout this article. In particular, the key design constraint to drastically reduce memory usage, as compared to either earlier implementations of this object or to other existing VTK data objects, was fully attained. This was amply demonstrated by our numerical results in in §6, consistent with what the theory laid out in §3 was allowing to hope for. This achievement dramatically reduces hardware requirements by several orders of magnitude, as compared to the alternatives currently used in AMR visualization.

Moreover, we propounded in §2.2 ([b]) to design and implement visualization filters that could natively operate on this novel data object, with the added stated goal of measurable performance in terms of execution speed. We have also entirely fulfilled our objectives in this regard, with demonstrated performance improvements with respect to our earlier design and implementation (not to mention the alternatives which are plainly useless for large-scale meshes).

Meanwhile, in the process of designing such filters, we entirely revised our earlier notion of tree cursors and, more importantly, supercursors. This resulted, in particular, in the introduction of a hierarchy of such objects in order to allow for the selection of the cursor that is the most tightly adapted to the particular algorithm being considered. Thanks in particular to the use of templates, and to the careful nesting of geometric and topological cursor properties, we achive in the added benefits of enhanced code maintainability and legibility. This new incarnation of the hypertree grid object allows us to confidently assert that it can be used even by an application developer not intimately familiar with the implementation details.

An other important aspect of this work is the availability of a full set of visualization filters able to natively operate over the novel data object. As explained in this article, these filters have all been written with performance in mind, in terms of both memory footprint and execution speed. Furthermore, our design based on the hierarchy of

templated cursors and supercursors, combined with the pre-selection paradigm for enhanced performance, gives developers the opportunity to easily create new filters tailored to their particular needs.
After everything is said, both in terms of theoretical soundness as of experimental results, what is ultimately our main claim to success is that our HERA code users at CEA have been able to begin routine post-processing of their AMR simulations, within a setting similar to that which already exists for the visualization of simulations based on other types of meshes. We therefore decided to contribute our development to the VTK code base so it can benefit the tree-based AMR community at large.

7.2 Perspectives

As formulated in §2.2, we have considered many potential avenues for further advances in the field of tree-based AMR visualization and analysis. These are not only of academic interest; in fact, they appear as strictly necessary when considering the post-processing options that are commonly available for other types of simulations, such as the finite element method using fully unstructured, conforming meshes.
First, we expect that our original goal [b] will be further achieved, as the community of users of our contributed code will increase and expand to connected yet different application domains.
Meanwhile, 2-dimensional AMR visualization can be especially challenging, as it requires that all leaf cells be rendered. In consequence, the interactivity of the visualization process decreases as input data object size increases. This problem is further compounded by the enhanced efficiency, in terms of memory footprint, of our hypertree grid model which elicits a new situation where rendering has become the bottleneck for our target platforms. As a result, the next goal ([c]) is indeed an urgent need, for which the lack of existing solution is currently hindering the AMR visualization and analysis workflow. Our preliminary developments in this regard should be finalized, validated and contributed shortly. Those focus on rendering speed, in particular in dimension 2, by exploiting level-of-detail properties, which we also plan to carefully study and explain in a sequel to this article.
Besides, the 3-dimensional visualization technique known as volume rendering, which has now been broadly used for almost two decades, for different types of data objects, remains mostly unchartered territory when it comes to tree-based AMR data and would come in direct support of our stated goal ([d]). Isocontour is often derided as being the "poor man's volume rendering". Albeit excessive, as in many cases an iso-surface is exactly what is required by the nature of the analysis being performed, this statement nonetheless usefully conveys the general idea that "true" 3-dimensional visualization is a capability that most if not all users want to have in a visualization tool set before they deem it sufficient. Considerable theoretical and experimental effort will be required in order to support this need, for almost no prior work exists in this area. However, such a major endeavor could potentially be amortized by 2-dimension specializations in addition to the overarching 3-dimensional goal.
The work done so far does not address *per se* any of items [e]-[g] in our initial vision. However, we believe that the theoretical groundwork which we have already conducted will allow for an easier pursuit of these goals in the future.
Last, we would like to close this panorama by mentioning an ongoing reflection regarding *in situ* and *in transit* visualization and analysis. This contemplates the possibilities that exist to directly couple an existing production-level AMR simulation code with a visualization tool set adapted to it, in a fashion that would entirely eliminate intermediate storage to disk. We are confident that this will allow us to address the last vision item ([h]) in the near future, which will be discussed in subsequent work.

Acknowledgments

This work was supported by the Department Military Applications (DAM) of French Alternative Energies and Energy Atomic Commission (CEA). We are grateful for advice from C. Guilbaud, F. Ledoux and N. Lardjane (CEA), and D. Chapon at the Institute of Research into the Fundamental Laws of the Universe (CEA IRFU) for providing and helping us exploit RAMSES large-scale simulation results. We also thank helpful discussions with J.-C. Frament and F. Meunier for syntactic corrections. This article could not have been written without the support and patience of ours families.

References

1. CEA's Tera supercomputer. http://www-hpc.cea.fr/en/complexe/tera.htm.
2. Overview of block structured AMR. Online: https://commons.lbl.gov/display/chombo/Overview+of+Block+Structured+AMR.
3. Patch-based adaptive mesh refinement for multimaterial hydrodynamics. In *Joint Russian-American Five-Laboratory Conference on Computational Mathematics/Physics*, Vienna, Austria, June 2005.
4. D. Aguilera, T. Carrard, G. Colin de Verdière, J.-P. Nominé, and V. Tabourin. Parallel software and hardware for capability visualization of HPC results. *Numerical Modeling of Space Plasma Flows: Astronum*, 2007.
5. D. Aguilera, T. Carrard, C. Guilbaud, J. Schneider, and S. Sorbet. Visualization and post-processing for high performance computing. *CHOCS*, 41:57–67, 2012.
6. L. Avila, U. Ayachit, S. Barré, J. Baumes, F. Bertel, R. Blue, D. Cole, D. DeMarle, B. Geveci, W. Hoffman, B. King, K. Krishnan, C. Law, K. Martin, W. McLendon, P. Pébay, N. Russell, W. Schroeder, T Shead, J. Shepherd, A. Wilson, and B. Wylie. *The VTK User's Guide*. Kitware, Inc., eleventh edition, 2010.

7. D. Bale, R. J. LeVeque, S. Mitran, and J. A. Rossmanith. A wave-propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM J. Sci. Comput.*, 24:955–978, 2002.
8. M. J. Berger, D. L. George, R. J. LeVeque, and K. T. Mandli. The GeoClaw software for depth-averaged flows with adaptive refinement. *Adv. Water Res.*, 34:1195–1206, 2011.
9. M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53(3):484–512, 1984.
10. A. Bondy and U.S.R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer London, 2011.
11. O. Bressand, L. Colombet, A. Fontaine, G. Harel, and J.-B. Lekien. Hercule: A library of scientific data management for numerical simulation. *CHOCS*, 41:29–37, 2012.
12. T. Carrard, C. Law, and P. Pébaÿ. A generic hyper tree grid implementation for AMR mesh manipulation and visualization in VTK. In *Proc. 21st International Meshing Roundtable*, San Jose, CA, U.S.A., October 2012.
13. D. D. Cline. Cfd on the 1-k node ncube/2. In *Proceedings of the Conference on Parallel Computational Fluid Dynamics '92 : Implementations and Results Using Parallel Computers: Implementations and Results Using Parallel Computers*, CFD '92, pages 99–109, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.
14. Adrien Douady and John Hamal Hubbard. On the dynamics of polynomial-like mappings. *Annales scientifiques de l'École Normale Supérieure*, Ser. 4, 18(2):287–343, 1985.
15. Jerome Dubois, Guenole Harel, and Jacques-Bernard Lekien. Interactive visualization and analysis of high resolution hpc simulation data on a laptop with vtk. IEEE VIS'18 SciVis Contest, 2018.
16. Jerome Dubois and Jacques-Bernard Lekien. Highly efficient controlled hierarchical data reduction techniques for interactive visualization of massive simulation data. 21st EG/VGTC Conference on visualization (eurovis19), 2019.
17. B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, 2000.
18. M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, D. Ranta, and R. Stefan. The RAGE radiation-hydrodynamic code. *Computational Science & Discovery*, 1(1), 2008.
19. P. Holmes. An introduction to chaotic dynamical systems (robert l. devaney); nonlinear dynamics and chaos: Geometrical methods for engineers and scientists (j. m. t. thompson and h. b. stewart) (with the assistance of r. ghaffari and c. franciosi and a contribution by h. l. swinney). *SIAM Review*, 29(4):654–658, 1987.
20. H. Jourdren. HERA: A hydrodynamic AMR platform for multi-physics simulations. In *Adaptive Mesh Refinement Theory and Application*, volume 41 of *LNCSE*, pages 283–294. Springer, 2005.
21. D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, 3 edition, 1997.
22. William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.
23. John M.Gisler Patchett and Galen Ross. Deep water impact ensemble data set. 2017.
24. P. P. Pébaÿ and D. C. Thompson. Communication-free streaming mesh refinement. *ASME Transactions, J. of Computing & Information Science in Engineering, Special Issue on Mesh-Based Geometry*, 5(4):309–316, 2005.
25. R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement. a new high resolution code called RAMSES. *Astronomy and Astrophysics*, 385:337–364, 2002.
26. P. Woodward, J. Jayayaraj, P.H. Lin, Mike M. Knox, D. Porter, C. Fryer, G. Dimonte, C. Joggerst, G. Rockefeller, W. Dai, R. Kares, and V. Thomas. Simulating turbulent mixing from Richtmyer-Meshkov and Rayleigh-Taylor instabilities in converging geometries using moving cartesian grids. Technical Report LA-UR-13-20949, Los Alamos National Laboratory, 2012.
27. M.-M. Yau and S. N. Srihari. A hierarchical data structure for multidimensional digital images. *Communications of the ACM*, 26(7):504–515, July 1983.