Chapter 2

Function Approximation on Discrete Grids

Two main sources of scientific data can be distinguished: the first group consists of measured data, for example acquired by 3D imaging devices like computer tomographs or microscopes. In this case the original, continuous signal is *sampled* at certain positions in space and/or time. The second main source of scientific data are numerical simulations, which for example compute the solution of partial differential equations. The most popular approaches are *finite difference* and *finite element* schemes. Whereas in the first case the continuous solution is approximated at discrete points, lines, area or volume elements, in the latter case it is represented by a finite set of locally defined (polynomial) functions, so called *shape functions*.

So in the vast majority of cases scientific data is not given as an analytical expression $f(\boldsymbol{x})$ that can be evaluated at any position within the data domain Ω, but rather as a finite number of data samples f_i defined at discrete locations $\boldsymbol{x}_i \in \Omega$.

Computational grids are employed to represent the geometrical and topological structure of these discrete approximations of the continuous data. In order to faithfully reconstruct the original signal (*ideal reconstruction*), respectively the continuous function, the discretization has to fulfill certain constraints. In particular the *sampling theorem* states that the spatial/temporal distance of the sampling locations has to correspond to the highest frequency components contained in the Fourier spectrum of the signal.

We will sketch the topics of computational grids and reconstruction, respectively interpolation in the next two sections.

2.1 Computational Grids

It is often advantageous to distinguish the topological structure (*abstract complex*) of a grid from its geometrical embedding (*realization*). Following the discussion in [13], these concepts can be defined as

Definition 1 *(**Abstract complex**): An abstract finite complex $\mathcal{C}$ of dimension d is a finite set of elements e, together with a mapping $dim : \mathcal{C} \mapsto \{0, ..., d\} \subset \mathbb{N}$ and a partial order relation $<$, such that $(e_1 < e_2) \implies (dim(e_1) < dim(e_2))$. $dim(e)$ is called the dimension of e. Elements of dimension 0 are called* vertices*, dimension 1-elements are called* edges*, dimension $(d-1)$-elements are called* faces *and d-dimensional elements are called* cells*.*

Definition 2 *(**Geometric realization**): A geometric realization of an abstract complex $\mathcal{C}$ is a Hausdorff space H together with a mapping*

$$\Phi : \mathcal{C} \mapsto \Phi(\mathcal{C}) = \bigcup_{e \in \mathcal{C}} \Phi(e) \subseteq H$$

that fulfills the following requirements:

$$\begin{array}{ll} (i) & e_1 < e_2 \Leftrightarrow \Phi(e_1) \subset \partial\Phi(e_2) \ \textit{and} \\ (ii) & \partial\Phi(e_2) = \displaystyle\bigcup_{e_1 < e_2} \Phi(e_1), \end{array}$$

for all elements e_1,$e_2 \in \mathcal{C}$.

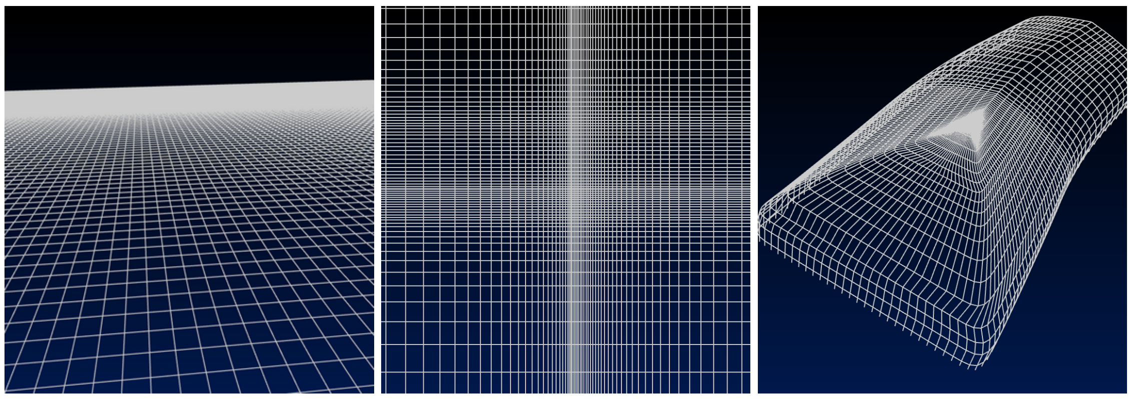
This definition is relatively general and allows for example cells that contain holes. In the special case that each cell in the complex is homeomorphic to open balls in $\mathbb{R}^d$, which is still general enough to cover more or less all types of cells that are employed in numerics and geometrical modeling, the complex is also called a *CW-complex*[1]. A detailed discussion of these topics is for example given in [26, 12].

According to their topological structure computational grids can be classified into two main categories, namely *structured* and *unstructured* grids, as well as mixtures of these two types.

2.1.1 Structured Grids

Structured grids are logically rectangular in the sense that their vertices can be arranged on a rectangular lattice in an appropriate geometric realization. Hence the vertices can be addressed by sets of integer indices, such that sequential indices refer to vertices that are connected by an edge, and so this type of grids does not require the storage of explicit connectivity. Structured grids consist of quadrilateral, respectively hexahedral cells. Their implicit connectivity relation allows for fast and efficient access of adjacent data samples and hence structured grids are popular in finite difference approaches.

[1]CW stands for closure-finite weak topology.



(a) (b) (c)

Figure 2.1: Examples of structured grids with uniform (a), rectilinear (b) and curvilinear coordinates (c).

The simplest but nevertheless very important example, particularly for measured data, are *uniform* grids. In this case the computational domain is discretized by rectangular, axis-aligned cells. Neglecting a potential offset vector, the coordinates of the vertices $\mathbf{x}_{ijk}$ can be computed from the index triples (i, j, k) via

$$\mathbf{x}_{ijk} := (ih_0, jh_1, kh_2). \tag{2.1}$$

Here $\mathbf{h}$ are the edge lengths of the cells, which are constant along each coordinate axis, but might differ between the three main directions. In the following the cell that contains the vertices

$$\{\, \mathbf{x}_{lmn} \,|\, l = i, i+1;\ m = j, j+1;\ n = k, k+1 \,\}$$

will be denoted by Ω_{ijk}

Rectilinear grids are a generalization of uniform grids in the sense that the edge lengths h_i might vary along each coordinate axis, compare Figure 2.1 (b). Although the connectivity between the cells is still defined implicitly, the coordinates of the vertices have to be provided explicitly for each axis, usually in form of three separate lists.

Curvilinear grids have a geometric realization such that the cells are not axis-aligned to an Euclidean coordinate system, as shown in Figure 2.1 (c). Often an analytic mapping from the uniform parameter space to the actual coordinate system is given, which allows to compute the vertex coordinates from their index triples. Otherwise the coordinates have to be stored explicitly. This separation allows to benefit from the memory efficient regular topology, while at the same time curvilinear grids are flexible enough to model a vast range of complex geometries.

2.1.2 Unstructured Grids

The second main category of computational grids are *unstructured* grids. They often consist of triangles or quadrilateral cells in two dimensions, respectively tetrahedral and hexahedral cells in the three dimensional case, but also prisms or pyramid cells are employed. Due to their flexibility, unstructured grids are well suited for modeling highly complex geometries. They further allow for easy grid adaption and local refinement, as discussed below. A disadvantage are their high memory requirements, since vertex coordinates as well as cell connectivity information have to be stored explicitly. Unstructured grids are primarily applied in finite element schemes.

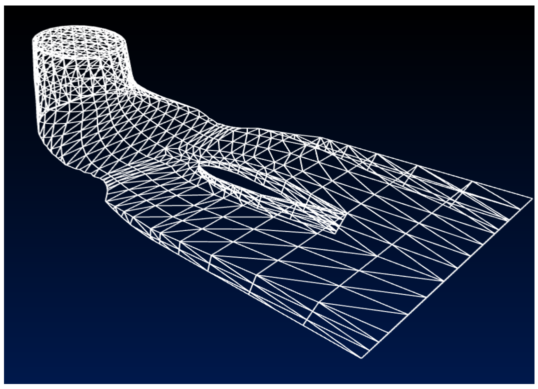


Figure 2.2: Unstructured grid that models the flow field inside a turbine.

2.1.3 Block-Structured, Overlaid and Hybrid Grids

A third category are grid types that combine aspects of both, structured and unstructured grids. *Block-structured* grids, introduced in the 1980s, further increase the flexibility of curvilinear grids. The computational domain is covered by a set of structured grids, which are pieced together at their boundary interfaces, ensuring coinciding vertices in these regions. Figure 2.3 (a) shows an example.

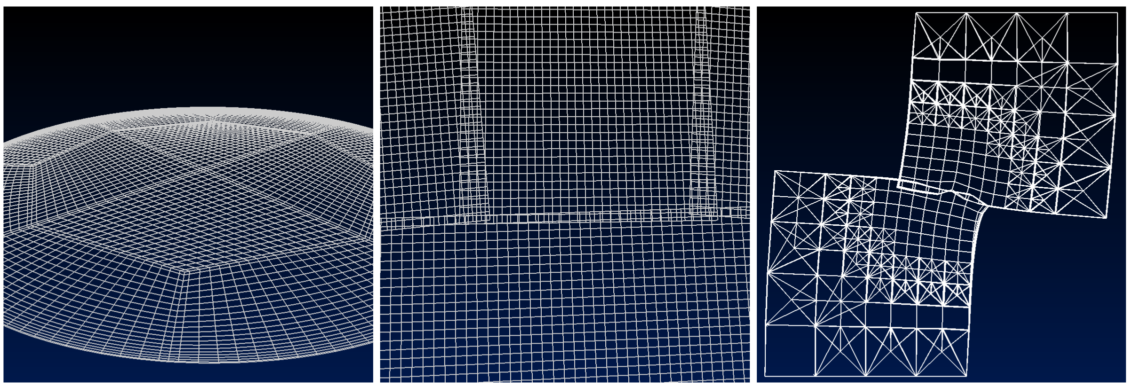

(a) (b) (c)

Figure 2.3: Examples of block-structured (a), overlaid (b) and hybrid (c) grids.

A generalization of block-structured grids are so-called *chimera* or *overlaid* grids, compare Figure 2.3 (b). In this case vertices on the overlapping patch region do not have to coincide, which facilitates the grid generation process substantially. A drawback is the increased complexity for interpolation due to partially overlapping cells in the boundary regions of adjacent patches and communication, especially in case of parallel computing.

Hybrid grids are grids that consist of different types of cells. This allows for example to cover homogenous regions by hexahedral cells, while complex shaped geometries at the boundaries of computational domains might be modeled with tetrahedra. Hybrid grids usually require complex data structures and increase the complexity of the numerical solvers, in order to handle the different types of cells. Figure 2.3 (c) shows an example.

In the remainder of this sections we will deal with the special case, that the discretized data is a numerical solution approximation of some equation, for example a system of partial differential equations.

2.1.4 Grid Adaption and Local Refinement

A powerful strategy to increase the accuracy of numerical solutions is to adapt the underlying grid structure, in order to better adjust to the physical behavior of the given problem.

There exist various grid adaption methods. In the *r-method* the topological structure of the grid remains unchanged. Instead the geometric location of the nodes is altered, based on an analysis of the current solution. Hereby it is crucial to avoid degenerated and overlapping elements. In contrast to this *p-methods* adjust the degree of the approximation by employing higher order shape functions (in the finite element approach). Therefore additional nodes have to be added to existing elements. In *h-methods* the grid adaption is carried out by refining (and coarsening) grid elements. Approaches that follow both of the latter two branches are called *hp-methods*.

In principle h-refinement can be carried out by refining the whole computational grid (*global refinement*), for example by simply replacing every cell by a number of smaller cells. However, for realistic grid sizes this usually results in too high computational efforts, in terms of memory and computational requirements. A much more efficient way is to refine only those cells Ω_i that cover regions where the local error of the solution $e_{loc}(\Omega_i)$ is above a certain threshold, aiming at an equal distribution of the error over the whole computational domain (*local refinement*). This usually involves the application of some kind of error estimator $\bar{e}_{loc}(\Omega_i)$ for the unknown local error $e_{loc}(\Omega_i)$, for example by comparing the solutions obtained by applying shape functions of different order or by different grid spacing.

Local refinement of tetrahedral grids is usually carried out by replacing the cells that need refinement by sets of smaller tetrahedra, generating new vertices in the interior of the cells. Special refinement strategies (for example *red* and *green* refinement rules), are necessary to avoid the creation of degenerated cells that can cause numerical problems, compare for example [71, 7].

Local refinement is more problematic for structured grids, since it interferes with the regular grid topology, and thus requires more sophisticated data structures to store the

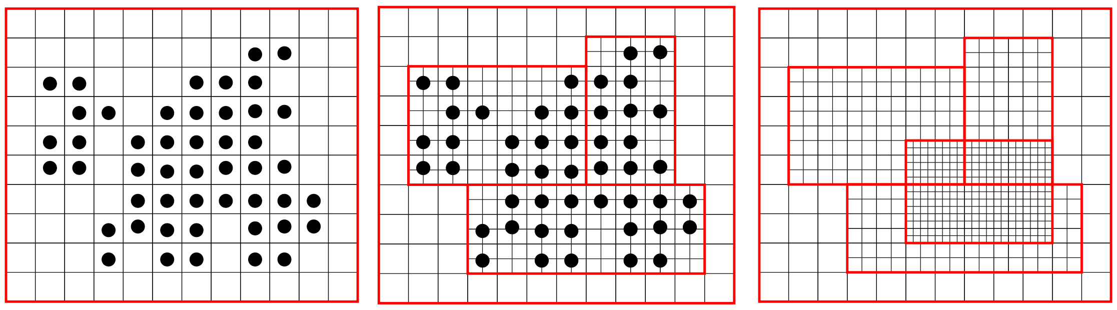


Figure 2.4: Refinement process for AMR schemes: Cells that require refinement are determined using local error criteria (a) and clustered into separate subgrids (b), which cover the regions with higher resolution. This process is recursively continued until each cell fulfills the error criteria (c).

resulting grid structures. A related problem is the introduction of so-called *hanging* or *dangling* nodes. These are nodes in the interior of the domain, that have a smaller number of emerging edges, compare Figure 5.4. These nodes require special treatment by the numerical solver in order to ensure the desired continuity properties of the solutions. One method is to restrict the solution at these nodes to the solution obtained at the location within the adjacent coarse cell (*dependent nodes*). Alternatively a so-called *conforming closure* might be performed, in which cells that contain dangling nodes are replaced by cells of a different types, resulting in a hybrid grid.

2.2 Adaptive Mesh Refinement (AMR)

A special adaptive method for solving hyperbolic partial differential equations was introduced by Berger et al. in 1984 [10]. The basic idea of AMR is to combine the simplicity of structured grids with the advantages of local grid adaption.

In this approach the computational domain is covered by a set of coarse, potentially overlapping structured subgrids. During the computation local error estimators are utilized to detect cells that require higher resolution. These cells are covered by a set of rectangular subgrids which may have arbitrary orientations. Unlike in finite element approaches these subgrids do not replace, but rather overlay the refined regions of the coarse base grid. The equations are advanced on the finer subgrids and this refinement procedure recursively continues until all cells fulfill the considered error criterion, giving rise to a hierarchy of nested refinement levels, as shown in Figure 2.4.

A further advantage of AMR is that each subgrid can be viewed as an separate, independent grid with a separate storage space. This allows to process subgrids almost independently during integration and hence the approach is well suited for parallel processing.

2.2.1 Structured Adaptive Mesh Refinement (SAMR)

In 1989 Berger and Collela [9] proposed a variant of the approach above, called *structured adaptive mesh refinement (SAMR)*, dedicated to simplify the application of the AMR scheme to hyperbolic conservation laws.

The main difference of SAMR is that the subgrids do not have arbitrary orientations anymore, but are rather aligned with the major axes of the Euclidean coordinate system. In particular this facilitates the computation of fluxes of conserved quantities like mass or energy through the cell faces. In principle the base grid and the subgrids can be rectilinear grids, but usually uniform patches are employed, compare Figure 2.5.

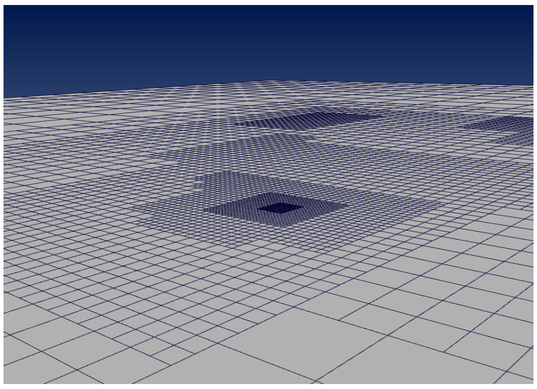


Figure 2.5: 2D example of an unrestricted, structured AMR hierarchy, i.e. adjacent cells can differ by an arbitrary number of refinement levels.

In the last decade SAMR has gained more and more popularity and nowadays it is applied in many domains like computational fluid dynamics [4], meteorology [3], relativistic astrophysics [75, 49] and in particular in cosmology [16, 2]. In the following we will describe the spatial and temporal refinement scheme in more detail and introduce basic notations for SAMR, which are used in the remainder of this thesis.

2.2.2 Notations

Let $\Omega \subset \mathbb{R}^3$ denote the data domain, which is discretized by a hierarchy of axis-aligned, uniform grids $(\Omega^l)_{l=0,1,...,l_{max}}$ with decreasing mesh spacings. The index l numbers the *refinement level*, starting with 0 for the coarsest level. Let the mesh spacing of the coarsest grid be given by $\mathbf{h}^0 = (h_0^0, h_1^0, h_2^0)$. The mesh spacings of the finer grids are recursively defined by $\mathbf{h}^l := (h_0^{l-1}/r, h_1^{l-1}/r, h_2^{l-1}/r)$, where the positive integer r denotes the so-called *refinement factor*. In principle this factor can differ for each direction and each level, but in order to simplify the notation we assume that it is constant.

Further let n_i be the number of vertices along the i-th coordinate axis of the base grid Ω^0. Assuming that the origin of the coordinate system is located at $(0,0,0)$, the coordinates of the vertices of Ω^l are given by

$$\mathbf{x}_{ijk}^l := (ih_0^l, jh_1^l, kh_2^l);\ i = 0, 1, ..., (n_0 - 1)(r)^l,\ j, k = ..., \tag{2.2}$$

and thus the coordinate $\mathbf{x}_{ijk}^l \in \Omega^l$ coincides with $\mathbf{x}_{ri,rj,rk}^{l+1} \in \Omega^{l+1}$.

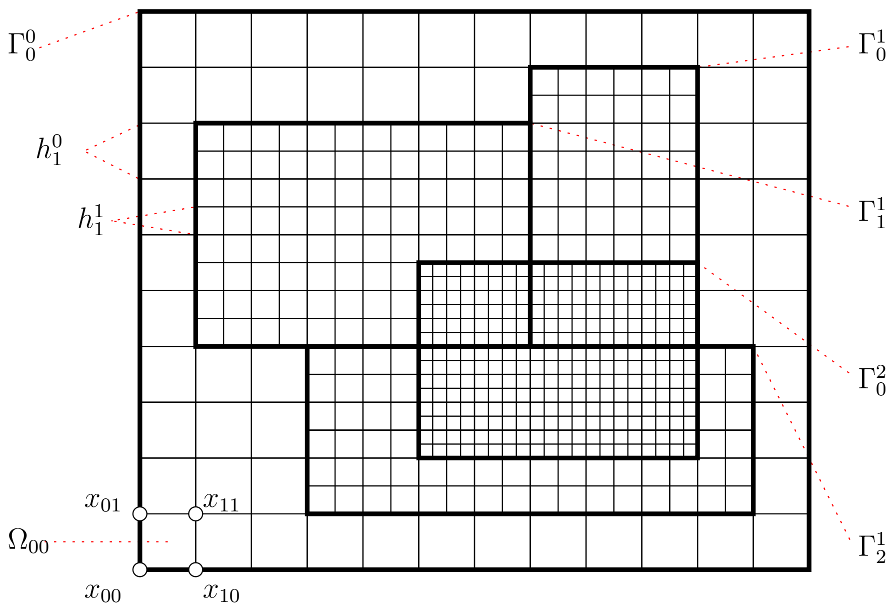


Figure 2.6: Two-dimensional example of a structured AMR grid. The root level Γ_0^0 is refined by three subgrids $\Gamma_0^1, ...\Gamma_2^1$ that generate the refinement level Λ^1. Λ^1 itself is refined by one subgrid Γ_0^2. All three subgrids on Λ^1 are siblings and each is a parent of Γ_0^2.

The grid cell $\Omega_{ijk}^l \subseteq \Omega^l$ is given by

$$\Omega_{ijk}^l := conv\{\ \mathbf{x} \in \Omega^l \mid \mathbf{x} = \mathbf{x}_{ijk}^l + \sum_{m=0}^{2} \alpha_m \mathbf{e}^m,\ \alpha_m \in [0, h_m^l]\ \}, \tag{2.3}$$

where $(\mathbf{e}^0, \mathbf{e}^1, \mathbf{e}^2)$ denote the standard basis in $\mathbb{R}^3$. Each coarse cell can be decomposed into a set of r^3 cells of the next finer discretization

$$\Omega_{ijk}^l = \bigcup_{\hat{\imath},\hat{\jmath},\hat{k}} \Omega_{\hat{\imath}\hat{\jmath}\hat{k}}^{l+1} \text{ with } \hat{\imath} = ri, ri+1, .., ri+r;\ \ \hat{\jmath}, \hat{k} = ..., \tag{2.4}$$

so the cells $\Omega_{\hat{\imath}\hat{\jmath}\hat{k}}^{l+1}$ provide a refinement of the coarse cell Ω_{ijk}^l. Since in AMR grids can be represented as a tree of nested levels, the coarse base grid Ω^0 is also called *root* level in this context. In general it may be composed of a set of non-overlapping, axis-aligned uniform patches, but in order to simplify the notation we assume that Ω^0 consists of just one patch.

As mentioned above, the solution of the equations are initially approximated on this root grid and the coarse solution is inspected utilizing the error estimator that detects cells that require higher resolution. These cells are clustered into disjoint, axis-aligned rectangular regions, which define new *subgrids*, consisting of cells of the next finer discretization Ω^1. We will sketch a clustering algorithm for this purpose in Subsection 2.2.3.

Notice that cells are either completely refined by cells of the next finer grid according to Equation (2.4), or remain completely unrefined. Let the m-th subgrid of Ω^l be denoted by

$$\Gamma^l_m = \left\{\Omega^l_{ijk} \subseteq \Omega^l \mid i = p^m_0, ..., p^m_0 + n^m_0;\ j,k = ...\right\}, \tag{2.5}$$

where $\mathbf{p}^m$ is the integer offset vector of this subgrid, and $n^m_i + 1$ is the number of cells per i-th coordinate axis.

Definition 3 *:*

(i) The union of all level l subgrids $\Lambda^l := \bigcup_{m=0}^{n_l} \Gamma^l_m$ is called **refinement level** *l or just* **level** *l. By construction these levels are nested, i. e. $\Lambda^{l+1} \subseteq \Lambda^l \subseteq \Omega^l$. We will denote the whole grid hierarchy, i. e. the union of all refinement levels by $\mathcal{H} := \bigcup_{l=0}^{l_{max}} \Lambda^l$.*

(ii) The level l subgrid Γ^l_m is called a **child** *or* **descendant** *of $\Gamma^{l-k}_n \in \Lambda^{l-k}$, denoted by $\Gamma^l_m = \mathcal{C}(\Gamma^{l-k}_n)$, if $(\Gamma^l_m \cap \Gamma^{l-k}_n) \neq \emptyset \wedge (\Gamma^l_m \cap \Gamma^{l-k}_n) \nsubseteq (\partial\Gamma^l_m \cup \partial\Gamma^{l-k}_n)$. In this case Γ^{l-k}_n is a* **parent** *of Γ^l_m, denoted by $\Gamma^{l-k}_n = \mathcal{P}(\Gamma^l_m)$.*

(iii) Two subgrids Γ^l_n and Γ^l_m on level l are called **siblings***, denoted by $(\Gamma^l_n = \mathcal{S}(\Gamma^l_m) \Leftrightarrow \Gamma^l_m = \mathcal{S}(\Gamma^l_n))$, if $(\Gamma^l_n \cap \Gamma^l_m) \neq \emptyset \wedge (\Gamma^l_n \cap \Gamma^l_m) \subseteq (\partial\Gamma^l_m \cup \partial\Gamma^l_n)$.*

In the original AMR scheme described in [9] each refinement level had to be enclosed by at least one layer of cells from the next coarser level of resolution, such that adjacent cells differ by at most one level. This constraint was relaxed by others, see for example [2, 65]. In the following we will refer to AMR grids that contain adjacent cells which differ by at most one level as *restricted* AMR grids, and to the more general case, like for example the one depicted in Figure 2.6, as *unrestricted* AMR grids.

2.2.3 A Clustering Algorithm

A crucial part in the AMR algorithm is the generation of the structured subgrids, which cover the cells that require higher resolution. An efficient and fast algorithm for clustering collections of cells into axis-aligned regions was suggested by Berger et al. [11]. It adopts signature based methods used in computer vision and pattern recognition. We will briefly describe the basic ideas in this section.

Assume that the information about which cells of a subgrid Γ^l_m are selected for clustering is encoded by the binary function defined on the index domain of Γ^l_m:

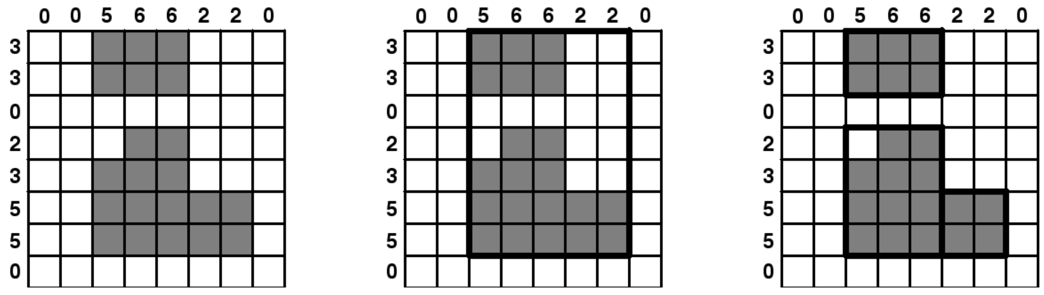
$$S : [p^m_0, .., p^m_0 + n^m_0] \times [p^m_1, ..] \times [p^m_2, ..] \rightarrow \{0, 1\} \tag{2.6}$$

with

$$S(i,j,k) = \begin{cases} 1, & \text{if } \Omega^l_{ijk} \text{ is marked for clustering} \\ 0, & \text{otherwise.} \end{cases} \tag{2.7}$$

In a first step the number of cells that need refinement is computed for each slice perpendicular to the three major coordinate planes and stored in so called signature lists. For example the entry for slice number i parallel to the yz-plane is given by

$$S_{yz}(i) = \sum_{j=p^m_1}^{(p^m_1+n^m_1)} \sum_{k=p^m_2}^{(p^m_2+n^m_2)} S(i,j,k) \tag{2.8}$$



(a) (b) (c)

Figure 2.7: *2D example of the clustering procedure: (a) In a first step the signature lists are computed. (b) Exterior rows and columns with zero entries are pruned off. (c) Interior zero entries and inflection points indicate splitting edges.*

and similarly for the two other orientations.

A two dimensional example is shown in Figure 2.7 (a). In the next step exterior zero-entries in these lists are detected and pruned off in order to place a minimal bounding box around the marked cells, as shown in Figure 2.7 (b). Any interior zero entry in these lists indicates a potential splitting index, i. e. a position at which the given volume is subdivided into two smaller subregions. If all signatures are non-zero, the second derivative

$$\Delta_{yz}(i) = S_{yz}(i+1) - 2S_{yz}(i) - S_{yz}(i-1), \tag{2.9}$$

and similar for $\Delta_{xy}(k)$, $\Delta_{xz}(j)$, of each signature list is computed and the largest inflection point is chosen as the splitting plane, compare Figure 2.7 (c). This procedure is repeated recursively on the newly created subregions until one of the following halting criteria is satisfied:

- The subregion exceeds the *efficiency* ratio, i. e. the ratio of the number of cells tagged for clustering to its total number of cells is greater than a preselected threshold.
- Further subdivision of the region would result in grid dimensions smaller than some *minimal extension*.

Notice that according to these criteria the clusters usually contain a number of cells that are not marked, in order to keep the number of created subgrids low. Usually an efficiency ratio of 85% and a minimal extension of 8 yields good results in terms of the number of grids and the additional memory overhead for the additional cells.

2.2.4 Temporal Refinement Scheme

For numerical solvers for hyperbolic partial differential equations with explicit time-integration, stability conditions demand that the time step size Δt of the scheme corresponds to the mesh size Δx, in the sense that the time step decreases as the mesh spacings decreases. As an example consider the Courant-Friedrichs-Levy (CFL) condition, which for hyperbolic and parabolic systems implies that

$$\frac{\Delta x}{\Delta t} \geq k,$$

respectively

$$\frac{(\Delta x)^2}{\Delta t} \geq k,$$

have to be fulfilled, where k is a system dependent constant. Hence a global time step for all subgrids in an AMR hierarchy would be determined by the cell size of the highest resolved level present in the hierarchy, resulting in a large computational overhead for the coarser levels.

This is the reason for the fact that besides the spatial refinement, AMR schemes for solving partial differential equations additionally perform a refinement in time. That means the spatially refined levels are updated more frequently than the coarser ones. The order in which the levels are advanced in time is demonstrated in the following pseudo code:

```
IntegrateLevel(level, parentTime) {
    dt = getTimeStep(level);
    while (time<parentTime) {
        time = time + dt;
        SetBoundaryValues();
        AdvanceEquations(dt);
        if (level<maxLevel) {
            IntegrateLevel(level+1,time);
            ProjectSolution(level+1,level);
            RegridHierarchy(level+1);
        }
    }
}
```

Figure 2.8: Pseudo code for the recursive AMR time-integration scheme.

First the coarse level Λ^l is advanced for a large time step (*AdvanceEquation*). Boundary values for the subgrids on levels are provided by interpolation on the next coarser level Λ^{l-1} or by copying from sibling subgrids on Λ^l (*SetBoundaryValues*).

Next the integration routine is recursively called for the refined levels $\Lambda^{l+1}, ..., \Lambda^{l_{max}}$, and these subgrids are advanced with a decreasing time step size. The integration of the finer levels is followed by the so-called *restriction step* (*ProjectSolution*), that updates the coarse grid function by the more accurate values of the finer ones.

In the last step the solution is inspected and the grid structure is adapted based on the local error criterion (*RebuildHierarchy*). This implies that the topology of $\Lambda^{l+1}, ..., \Lambda^{l_{max}}$

might change after each integration step on a level l. In general the structure of the whole hierarchy (except for the root level) is modified at time steps at which the root level grid is updated.

The time steps of the refined levels do not necessarily have to be equally distant, but it has to be ensured that after an integer number of updates the times of all levels in the hierarchy match up again.

A temporal refinement factor $r_t \in \mathbb{N}$ between a pair of two consecutive refinement levels $(\Lambda^l, \Lambda^{l+1})$ indicates that Λ^l is evolved one large step Δt_l, and next Λ^{l+1} is evolved r_t times with a step sizes of $\Delta t / r_t$. Figure 2.9 depicts the AMR integration order for a temporal refinement factor of 2. In the following we denote the union of level l subgrids at a certain time by $\Lambda^l(t)$ and the grid hierarchy by $\mathcal{H}(t)$.

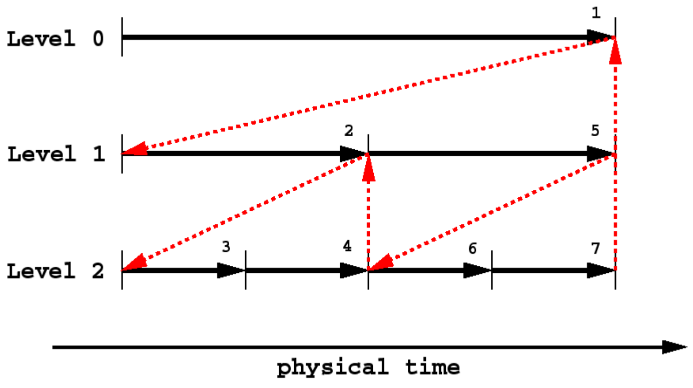


Figure 2.9: Order of temporal integration of a grid hierarchy with an overall temporal refinement factor of $r_t = 2$.

2.3 Interpolation

In the following we will restrict the discussion to interpolation schemes that are employed in the remainder of this thesis. For the sake of simplicity we will assume real-valued, scalar grid functions. For more detailed information the reader might refer to textbooks like [98].

Let the grid function $f : \mathcal{G} \mapsto \mathbb{R}$ and possibly also some of its derivatives $f', f'', ...$ be defined at the n discrete locations $\mathbf{p}_i, i = 0, ..., n-1$ on the computational grid $\mathcal{G}$. Common examples are grid functions sampled at the vertices of the grid (*vertex-centered* grid functions), as well as at cell or face centers (*cell-*, respectively *face-centered* grid functions). In order to obtain data values at arbitrary positions inside the data volume Ω, an interpolant

$$I(f, f', f'', ...|\boldsymbol{p}_0, ..., \boldsymbol{p}_{n-1}) : \Omega \mapsto \mathbb{R} \;\; \text{with } \boldsymbol{p} \mapsto I(\boldsymbol{p}) \tag{2.10}$$

needs to be specified. Interpolation functions are often expressed as linear combinations of a set of *weighting functions* $\omega_i, \tilde{\omega}_i, ... : \mathcal{G} \mapsto \mathbb{R}$, that are also called *shape functions* in the context of finite element analysis

$$I(\mathbf{p}) = \sum_{i=0}^{n-1} \omega_i(\mathbf{p})\; f(\mathbf{p}_i) + \tilde{\omega}_i(\mathbf{p})\; f'(\mathbf{p}_i) + ... \tag{2.11}$$

According to the *Shannon sampling theorem* [64, 77], a continuous signal $\hat{f}$ can be reconstructed from its discrete data samples (*ideal reconstruction*), if the sampling rate was larger than the highest frequency contained in the spectrum of the signal:

Theorem 1 *Let $\hat{f}$ be a function that is band-limited to $[-\sigma, \sigma]$, i. e.*

$$\hat{f}(t) = \frac{1}{\sqrt{2\pi}} \int_{-\sigma}^{\sigma} g(\kappa) e^{-i\kappa t} d\kappa \;\; \textit{for } t \in \mathbb{R},$$

with $g \in L^2(-\sigma, \sigma)$. Then $\hat{f}$ can be reconstructed from its samples $\hat{f}(\frac{k\pi}{\sigma}) = \hat{f}_k$, taken at equally spaced nodes $\frac{k\pi}{\sigma} \in \mathbb{R}$, via convolution with the sinc function

$$\hat{f}(t) = d \sum_{k=-\infty}^{\infty} \frac{sin(\sigma t - k\pi)}{(\sigma t - k\pi)}\; \hat{f}(\frac{k\pi}{\sigma}), \quad \textit{i. e.} \;\; \omega_i := sinc(\sigma t - k\pi).$$

Since the support of the $sinc$ function is infinite, the values of the interpolant I at each location depend on all data samples present in the volume. Thus ideal reconstruction is computationally expensive, especially for highly resolved data. For this reason usually approximations of the $sinc$ function by weighting functions with finite support are used. The simplest example are box functions, that are centered at the data locations

$$\omega_i(x) = \begin{cases} 1, & \text{if } |x - x_i| < \frac{h}{2} \\ 0, & \text{otherwise} \end{cases} \tag{2.12}$$

respectively the products $\omega_i(\boldsymbol{x}) := \omega_i(x_0)\omega_i(x_1)$... for higher dimensions. This results in so-called *constant* or *nearest-neighbor* interpolation

$$I(\mathbf{x}) = f_i \quad \text{for } \mathbf{x} \in supp(\omega_i). \tag{2.13}$$

This interpolation is usually applied to cell-centered grid functions, which for example result from finite volume simulations. In this case the data often represents the average of some (conserved) quantity $q(\mathbf{x})$ over the domain of cell Ω_i, i. e.

$$f_i = \bar{q} := \frac{\int_{\Omega_i} q_i(\mathbf{x}) d\mathbf{x}}{\int_{\Omega_i} d\mathbf{x}}, \tag{2.14}$$

like for example mass density in hydrodynamic simulations. Of course the resulting global interpolant is discontinuous. Higher order interpolation for hexahedral cells is usually realized by shape functions that are (tensor-)products of the *Lagrange polynomials*

$$L_i(x|x_0, x_1, ..., x_m) := \prod_{j=0;\, j\neq i}^{m} \frac{x - x_j}{x_i - x_j}. \tag{2.15}$$

Consider the one dimensional interval $[x_0, x_1]$ with data located at the boundaries, and set $\omega_i(x) := L_i(x|x_0, x_1)$. Introducing the local coordinates $\xi = \frac{x - x_0}{x_1 - x_0}$ with $\xi \in [0, 1]$, one obtains $w_0(\xi) = 1 - \xi$ and $w_1(\xi) = \xi$. Hence Equation (2.11) yields the linear interpolant $I(\xi) = f_0 + \xi(f_1 - f_0)$ for the interval.

For quadrilateral cells, the shape functions in local coordinates are analogously defined by

$$\omega_{ij}(\xi, \eta) := L_i(\xi|0, 1)\ L_j(\eta|0, 1), \tag{2.16}$$

with vertex numbering according to Figure 2.10.

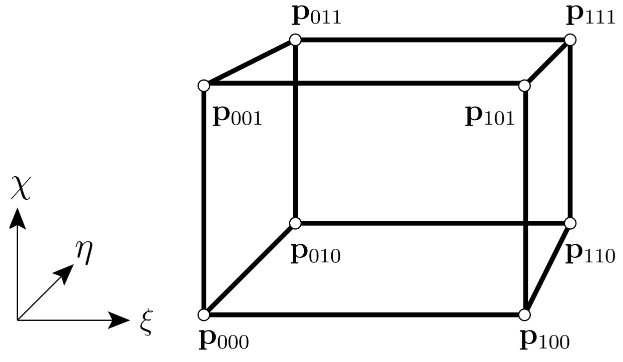


Figure 2.10: Local coordinates and numbering of vertices for hexahedral cells.

Hence the bilinear interpolant reads as follows

$$\begin{aligned} I(\xi,\eta|f_{00},...,f_{11}) &= \sum_{i=0}^{1}\sum_{j=0}^{1}\omega_{ij}(\xi,\eta)f_{ij} \qquad (2.17)\\ &= L_0(\eta)\left(L_0(\xi)f_{00}+L_1(\xi)f_{10}\right)+\\ &\quad L_1(\eta)\left(L_0(\xi)f_{01}+L_1(\xi)f_{11}\right)\\ &= I(\xi|f_{00},f_{10})+\eta(I(\xi|f_{01},f_{11})+I(\xi|f_{00},f_{10})). \qquad (2.18)\end{aligned}$$

This shows that bilinear interpolation is equivalent to two linear interpolations along the x-axis followed by a linear interpolation along the y-axis. This is computationally more efficient than a direct evaluation of (2.17), since the number of multiplications is reduced from 12 to 4.

Completely analogously the shape functions for hexahedral elements and trilinear interpolation are defined by

$$\omega_{ijk}(\xi,\eta,\rho) := L_i(\xi|0,1)\ L_j(\eta|0,1)\ L_k(\rho|0,1), \qquad (2.19)$$

and the resulting trilinear interpolant for hexahedral cells reads

$$\begin{aligned} I(\xi,\eta,\chi) &= \sum_{i,j,k=0}^{1}\omega_{ijk}(\xi,\eta,\chi)f_{ijk} = ... \qquad (2.20)\\ &= I(\xi,\eta|f_{0,0,0}...f_{1,1,0})+ \qquad (2.21)\\ &\quad \chi\left(I(\xi,\eta|f_{0,0,1},...,f_{0,1,1})-I(\xi,\eta|f_{0,0,0},...,f_{1,1,0})\right). \qquad (2.22)\end{aligned}$$

Hence trilinear interpolation is equivalent to two bilinear interpolations parallel to the xy-coordinate plane, followed by a linear interpolation along the z-direction.

If in addition to the function values also information about the first derivative is available, C^1-continuous *Hermite interpolation* might be applied. Let us again consider the case of a one-dimensional interval $[x_0, x_1]$ with the function values f_0, f_1 and first derivatives $f_0^{'}, f_1^{'}$ given at the interval boundaries. Equation 2.11 reads as

$$I(x|f_0,f_1,f_0^{'},f_1^{'}) = f_0\ H_0^3(x)+f_0^{'}\ H_1^3(x)+f_1\ H_2^3(x)+f_1^{'}\ H_3^3(x), \qquad (2.23)$$

where the shape functions $\{H_0^3,...,H_3^3\}$ are given by the *cubic Hermite polynomials*. They can be defined by

$$\begin{aligned} &H_0^3(x_0)=1,\ \tfrac{d}{dt}H_0^3(x_0)=0,\ H_0^3(x_1)=0,\ \tfrac{d}{dt}H_0^3(x_1)=0,\\ &H_1^3(x_0)=0,\ \tfrac{d}{dt}H_1^3(x_0)=1,\ H_1^3(x_1)=0,\ \tfrac{d}{dt}H_1^3(x_1)=0,\\ &H_2^3(x_0)=0,\ \tfrac{d}{dt}H_2^3(x_0)=0,\ H_2^3(x_1)=1,\ \tfrac{d}{dt}H_2^3(x_1)=0,\\ &H_3^3(x_0)=0,\ \tfrac{d}{dt}H_3^3(x_0)=0,\ H_3^3(x_1)=0,\ \tfrac{d}{dt}H_3^3(x_1)=1.\end{aligned}$$

Using local coordinates $\xi \in [0,1]$ they have the explicit form

$$\begin{aligned} H_0^3(\xi) &= 2\xi^3-3\xi^2+1,\\ H_1^3(\xi) &= -2\xi^2+3\xi^3,\\ H_2^3(\xi) &= \xi^3-2\xi^2+\xi,\\ H_3^3(\xi) &= \xi^3-\xi^2. \qquad (2.24)\end{aligned}$$

Piecewisely connected, cubic Hermite polynomials result in globally C_1-continuous interpolation for a one-dimensional grid $[\xi_0, \xi_1, ...\xi_n]$. Often no information about the derivatives of the function is available. In this case the first derivative at a vertex ξ_i might be approximated by the slope of two quadratic polynomials fitted through $(\xi_{i-1}, f_{i-1}), (\xi_i, f_i), (\xi_{i+1}, f_{i+1})$, respectively $(\xi_i, f_i), (\xi_{i+1}, f_{i+1}), (\xi_{i+2}, f_{i+2})$. For equidistant intervals, the resulting *Catmull-Rom spline*, reads as follows

$$\begin{aligned} I(\xi|\xi_0, \xi_1) = \frac{1}{2}(\, 2f_i &+ (f_{i+1} - f_{i-1})\, \xi \\ &+ (2f_{i-1} - 5f_i + 4f_{i+1} - f_{i+2})\, \xi^2 \\ &+ (3f_i - f_{i-1} - 3f_{i+1} + f_{i+2})\, \xi^3 \,). \end{aligned}$$

Similar to the bi-, and trilinear interpolation discussed above, interpolants for quadrilateral and hexahedral cells are generated via products of the one-dimensional cubic Hermite polynomials.

2.4 Spatial Access Methods

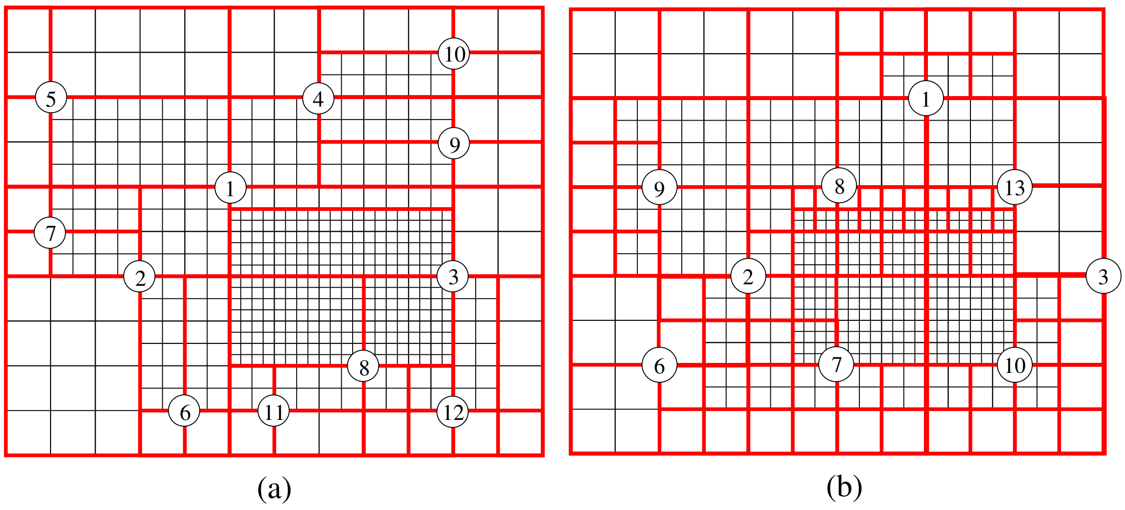


Figure 2.11: Decomposition of the AMR grid example given in Figure 2.6 into disjoint blocks that consist of cells from the same level of resolution, see also Section 5.2. Image (a) shows a result for a point-quadtree, whereas in (b) a region-quadtree is employed. The numbers indicate the order in which the first quadrants are inserted. In the region-octree example quadrants that lie outside the grid domain have been omitted.

For many visualization algorithms it is necessary to locate cells that enclose a particular spatial position, for example during the interpolation operation for volume rendering via raycasting. This *point location* operation has a complexity of $O(1)$ for uniform grids, since according to Equation (2.1), the index triple (i, j, k) of the cell containing the location $\mathbf{x} = (x_0, x_1, x_2)$ can be computed via

$$i = [\frac{(x_0 - k_0)}{h_0}], \quad j = [\frac{(x_1 - k_1)}{h_1}], \quad k = [\frac{(x_2 - k_2)}{h_2}],$$

where $\mathbf{k}$ denotes the grid offset vector and $\mathbf{h}$ is the grid spacing. However, for unstructured grids no such simple relation holds and the simple $O(n)$ approach of inspecting each of the n grid cells is unfeasible even for moderately sized grids.

In order to accelerate the performance of the point location and other spatial operations like intersection-, or adjacency-queries, dedicated spatial data structures are employed. They have in common that they decompose the search domain into a set of smaller, usually polyhedral subregions of simple geometry, which index into the set of spatial objects contained within the search domain. The decomposition is often organized in a hierarchical manner, which results in a logarithmic search complexity.

In the following we will sketch the most popular spatial data structures that are employed for these purposes. For more detailed information the reader might refer to textbooks, like [69, 73].

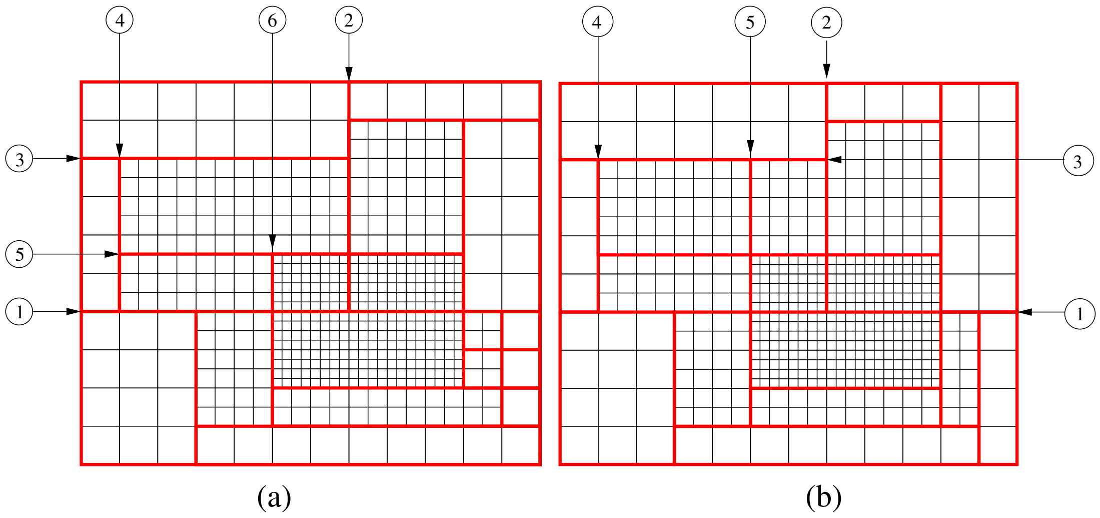


Figure 2.12: Decomposition of the hierarchy of Figure 2.6 using a kD-tree (a) and an adaptive kD-tree (b) data structure. The numbers indicate the order of the first partition axes.

Developed in 1974, quadtrees are among the first data structures investigated for spatial data access. *Quadtrees* and their three dimensional analog, *octrees*, are rooted trees, which recursively decompose the data domain into four axis-aligned rectangles, respectively eight subvolumes. Two types of quadtrees are distinguished in the literature: *region-* and *point-quadtrees*. Point-quadtrees are primarily used to store collections of points. For each point that is inserted in the tree, the leaf node that contains the point is subdivided into four disjoint axis-aligned subnodes, such that the newly inserted point coincides with the intersection of the partition hyperplanes. In contrast to this, nodes of region quadtrees (octrees) are always subdivided into subregions of the same size. Figure 2.11 shows two examples.

Besides to accelerate the spatial operations mentioned above, (region) octrees are often employed to generate multi-resolution representations of volumetric datasets. Here the root node represents the coarsest resolution of the data, which is recursively refined, until the original data resolution is reached. In this context a *full* octree, i. e. an octree where all leaf nodes are located on the same level, is called a *pyramidal* representation of the original data.

Bin-trees are similar to region-quadtrees, with the difference that each internal node is recursively split into two equal sized subregions along one axis-aligned hyperplane. The splitting direction alternates from one level of the tree to the next. *kD-trees*, which operate on k-dimensional domains, are a generalization of bin-trees in the sense that each non-leaf node is subdivided into two subregions of potentially different size. As for bin-trees the (k-1)-dimensional hyperplane is aligned with one of the major coordinate axes, with a cyclic change of orientation from level to level, compare Figure 2.12 (a).

The *adaptive kD-tree* variant relaxes the restriction of the alternating orientations of the dividing planes. Here the direction of the next axis-aligned plane at an internal node is unrelated to the depth of the node and might rather be changed in an flexible way, compare Figure 2.12 (b). *Binary-Space-Partition trees* or just *BSP-trees* are even more general, in the sense that the division planes do not have to be axis-aligned, but may rather have an arbitrary orientation.

R-trees are well suited for representing hierarchies of overlapping d-dimensional intervals. Each internal node stores the minimal axis-aligned rectangle or bounding box that encloses all the object indexed by its children.

The R-tree is a height-balanced tree, with a maximal height of $log_m(n)$ for n objects in the index set. A R-tree has a degree of (m, M), where $m, M \in \mathbf{N}$ and $m \leq \frac{M}{2}$, if each interior node has a number of children that ranges between m and M. The root node has to contain at least two children. In contrast to R-trees, sibling nodes of R$^+$-

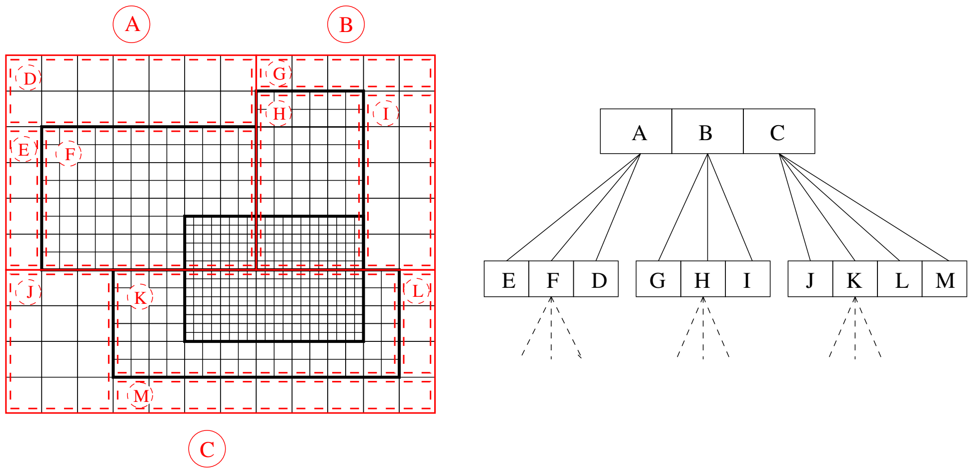


Figure 2.13: Decomposition of the hierarchy of Figure 2.6 using a R^+-tree. To avoid visual clutter only the first two levels are shown.

trees do not overlap. This might be achieved by clipping objects that intersect more than one interval on the same level. This results in an increased search efficiency, since point queries require traversing only one path through the tree, but on the other hand increase the storage requirements due to a potentially higher number of nodes.

The examples given in Figures 2.11 to 2.13 suggest that adaptive kd-trees are well suited for decomposing AMR hierarchies into non-overlapping blocks of constant resolution, due to the small number of resulting regions. Compared to the R^+-tree example that also requires a relatively small number of blocks, they have the further advantage to guarantee that the regions can be traversed in a view-consistent order, which is advantageous for volume rendering approaches, as discussed in Section 4.4.9. In Section 5.2 we will propose an algorithm that utilizes an adaptive kd-tree for such a decomposition.