5.5 Direct Volume Rendering for Structured AMR Data

In this section we present a hardware-accelerated volume rendering algorithm as well as a software-based raycasting approach for structured AMR data, that directly exploit the hierarchical structure in order to achieve fast rendering performance even for highly resolved datasets.

5.5.1 3D Texture-Based Volume Rendering

A possible approach for volume rendering of AMR data via 3D textures is based on the utilization of the stencil buffer, similar to the algorithm for slice extraction discussed in Section 5.4. In this case a separate 3D texture is allocated for each subgrid and a stack of slices $\mathcal{S}_i$, oriented perpendicular to the viewing direction, is extracted and blended back-to-front in the frame buffer:

```
define 3D textures;
enable stencil-buffer test;
enable blending;
/* loop over all slices from back-to-front */
for all (S_i, i=0,...,n) {
  clear stencil-buffer;
  /* loop over all levels from fine to coarse */
  for all (Λ^l, l=l_max,...,0) {
     /* loop over all subgrids */
     for all (Γ^l_k ∈ Λ^l) {
        render (S_i ∩ Γ^l_k) ;
     }
  }
}
```

However, this approach has some drawbacks: Volume rendering is fill-rate limited and the stencil-buffer test is performed in the last stage of the rendering pipeline, compare Section 3.1, so the time consuming interpolation operations are still performed for regions that do not contribute to the final image at all.

Secondly, frequent texture-I/O operations will decrease the rendering performance, since the volume is processed in the order of slices, which increases the number of texture switches. This is especially disadvantageous if the total size of textures required to represent the AMR hierarchy exceeds the amount of available texture memory.

It is more advantageous to avoid multiple processing of regions that are covered on different levels of resolution. We therefore decompose the data domain into axis-aligned blocks $\mathcal{B}^l_m \subset \Lambda^l$, with

$$(\mathcal{B}^l_i \cap \mathcal{B}^l_j) = \emptyset \vee (\mathcal{B}^l_i \cap \mathcal{B}^l_j) \subset (\partial\mathcal{B}^l_i \cup \partial\mathcal{B}^l_j) \quad \text{for } i \neq j,$$

that consists either of cells that are refined by subgrids, or of cells which are not further refined. Each block is processed separately during rendering phase, so it has to be ensured

that no subsets of the blocks build visibility cycles for any viewpoint. In Section 5.2 we proposed a decomposition that fulfills all of these constraints. In particular the resulting blocks are arranged in a kD-tree structure, allowing efficient determination of the view-consistent order for each viewpoint.

To reduce the amount of additional texture memory required by the "power-of-two" restrictions of the graphics-API, respectively graphics-hardware, we apply the texture-packing approach discussed in Section 4.4.

As discussed above we employ nearest-neighbor interpolation for cell-centered AMR data and trilinear interpolation for vertex-centered data. In the first case the texels are aligned with the centers of the cells, while in the second one they are aligned with the vertices of the grid. To avoid artifacts originating from discontinuities between sibling subgrids, adjacent texture-blocks share a row of data samples at their common boundary faces and the data at dangling nodes has to be replaced to the interpolated texel values of the abutting, coarse texture.

5.5.2 Opacity Corrections

If a block is selected for rendering, as discussed in Section 5.5.4, it is processed as in the standard approach for volume rendering via 3D textures. Each texture is sampled on slices perpendicular to the viewing direction, which are blended in the frame buffer.

Since texture-based volume rendering is fill-rate limited, it is advantageous to reduce the number of interpolation operations by adapting the distance of the textured slices according to the resolution of the associated subgrids. If blocks on level 0 are rendered with slice distances Δ_0, the slice distance for blocks on level l is given by $\Delta_l = \frac{\Delta_0}{r^l}$, where r denotes the refinement factor. Slices for the different blocks are aligned as indicated in Figure 5.18.

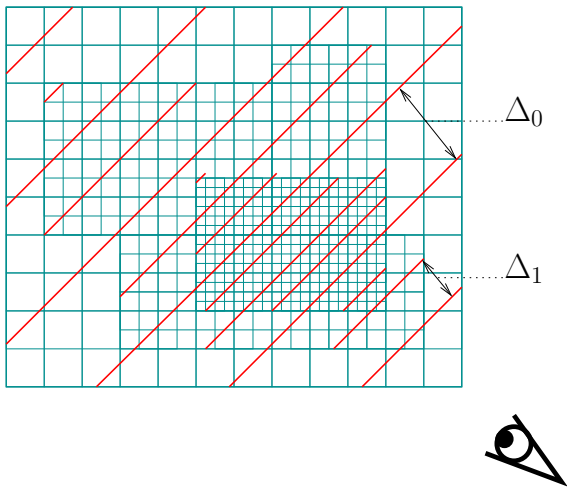

Figure 5.18: The sample distance of slices varies locally and depends on the blocks' cell size.

The texel alignment ensures $\mathcal{C}^0$-continuity for vertex-centered data, if texels that correspond to hanging nodes are obtained by bilinear interpolation between the texels on the coarse face. Nevertheless, since blocks from different levels are rendered with varying sample distances, the opacity entries of the colormap have to be adjusted in order to avoid noticeable differences in the overall transparency between blocks from different resolution levels.

Let $\alpha_i(0)$ denote the i-th colormaps opacity entry used for rendering root level blocks.

According to Equation (3.18), the opacity is related to the absorption coefficient κ_i via

$$\alpha_i(0) = 1 - T_i = 1 - e^{-\kappa_i \Delta_0}.$$

So one yields the following relation for the opacity entries for rendering blocks on refinement level l

$$\begin{aligned} \alpha_i(l) &= 1 - e^{-\kappa_i \Delta_l} \\ &= 1 - (e^{-\kappa_i r^{-l} \Delta_0}) \\ &= 1 - (1 - \alpha_i(0))^{\frac{1}{r^l}} \end{aligned} \tag{5.2}$$

For each level of the hierarchy a separate colormap is precomputed and activated prior to rendering the separate blocks.

Even with these opacity correction artifacts might remain in the resulting renderings. They are due to small regions at level boundaries, where the sample distance does not correspond to the opacity correction according to Equation (5.2). Weiler et al. addressed this problem in detail in [92]. They present an efficient algorithm to detect these problematic regions and render the corresponding slice parts with the correct opacity that corresponds to the actual sample distance in these regions.

5.5.3 Raycasting

For the raycasting approach we also employ the decomposition of the data domain into blocks $\mathcal{B}_i$ of cells from the same resolution level, since it accelerates the point location operation. In contrast to the hardware-accelerated algorithm, the tree is traversed in a front-to-back order, to allow for "early-ray-termination" once the opacity of the ray exceeds a certain threshold. Each block is processed separately. First the bounding box is scan-converted and for each pixel the intensity contribution of its ray-segment, which results from the intersection between the ray and the blocks bounding box, is computed according to (3.11)

$$I(s_n) = I(s_{n-1})T_n + b_n.$$

Here $I(s_{n-1})$ is the accumulated intensity of the pixels ray-segments that have already been processed. We support the standard as well as the adaptive integration scheme described in Chapter 4 for the numerical approximation of T_n and b_n. In the last step the updated intensity values for the pixel are written into the frame-buffer and the next ray-segment of the block is processed.

5.5.4 Adaptive Block Selection

The kD-tree structure is utilized for traversing the separate blocks in a view-consistent order. If a node that is associated with a block is processed, two cases have to be distinguished:

- The node is a leaf node, indicating that the covered cells are not further refined and thus have to be rendered.
- The node is not a leaf node, i.e. it represents a region that is further refined.

In the second case, there are two alternatives: to render the region with the actual level of detail, or to further decent the subtree, since higher visual accuracy is required.

We base this decision upon the projected extend of the subgrid cells in screen space. If the projected size is smaller than the extend of a selectable number of pixels, the block is selected for rendering and the traversal of the subtree is stopped. In order to quickly estimate the maximal screen space extent of the subgrid cell, we project a ball centered at the grids bounding box corner closest to the viewpoint and with a diameter equal to the grid cells diagonal. In addition a maximal level at which the hierarchy traversal is stopped can be specified. Lower resolution can be used during user interaction like rotation or zooming, while a deeper traversal of the hierarchy is performed for still images. A combination of the methods can be used to guarantee a desired lower bound of the frame rate.

5.5.5 Results and Discussion

The measurements were performed on a SGI ONYX2-SYSTEM with a 195 MHz MIPS R10K processor and a single INFINITEREALITY2 graphics-pipeline with two RM7 raster managers and 64 MBytes of texture memory each. The size of the viewport was 800×800 pixels. We applied the algorithms to three different datasets with increasing complexity. See Figures 5.19 to 5.21 for detailed information and resulting renderings.

The results of the measurements are shown in Table 5.1. It lists the number of generated leaf blocks, the preprocessing time for allocating the block hierarchy and texture packing as well as resampling in case of cell-centered data, the percentage of texture memory reduction achieved by texture packing and the size of the resulting texture. An average number of 3 to 4 leaf blocks per subgrid was created, independent of the depth and total number of grids of the hierarchy. The average texture memory reduction achieved by packing was about 45 %.

Table 5.2 shows the associated frame rates for the root level data, the full hierarchy and the close-up view on the refined part for the viewer positions chosen in Figure 5.19

	# leaf blocks	**preprocessing**	**ratio**	**texture memory**
Dataset I	345	0.2 s	45%	1 MB
Dataset II	970	1.2 s	43%	16 MB
Dataset III	3370	5.8 s	46%	16 MB

Table 5.1: This table lists the number of generated leaf blocks, the preprocessing times for allocating and packing the textures as well as resampling, the achieved texture memory reduction and the resulting size of the packed texture.

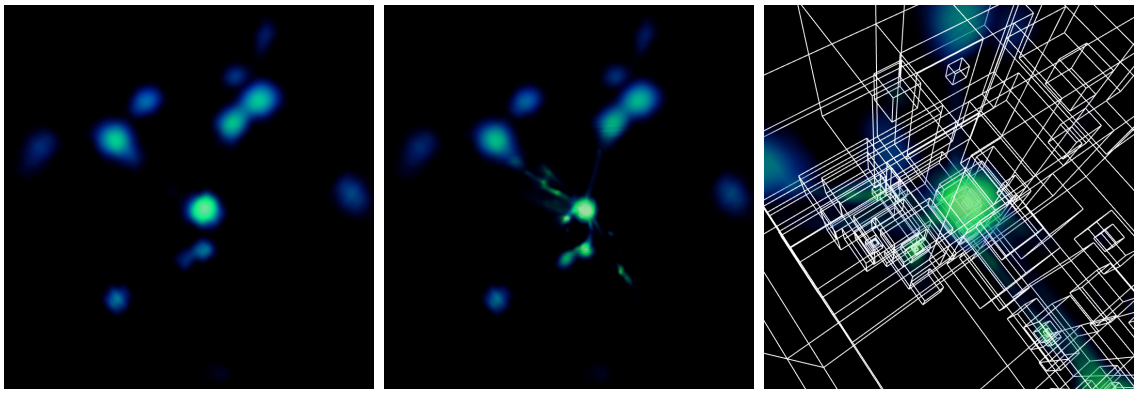

Figure 5.19: Dataset I, resulting from an AMR galaxy cluster simulation, consists of 91 grids on 7 levels of refinement. The (resampled) root level contained 33^3 samples and was rendered with 120 slices, the more refined grids with respectively more, as discussed in Section 5.5.2. If resampled to an uniform grid, the grid would contain more than 4.000^3 data samples, corresponding to about 70 GByte of texture memory. (left) root level, (middle) full hierarchy, (right) associated bounding boxes. *(dataset courtesy of M. Norman, National University of California)*

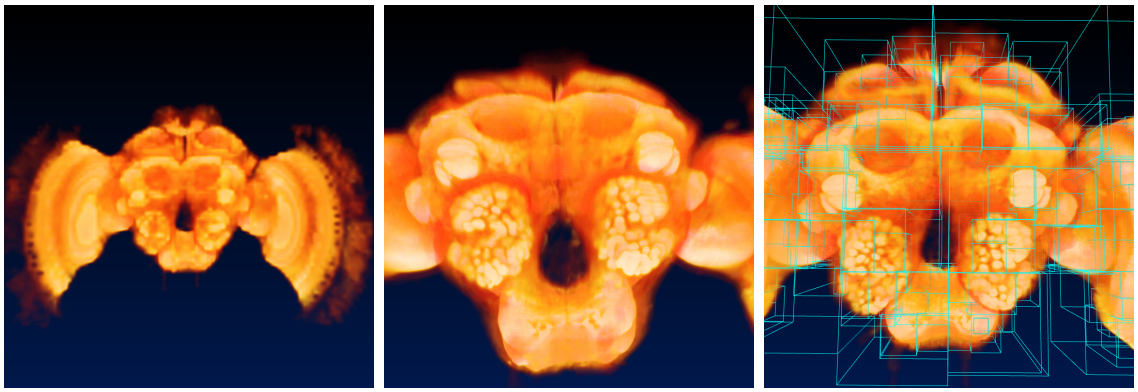

Figure 5.20: Dataset II represents a hierarchy consisting of 359 grids on 4 levels of refinement. The root level contains $95 \times 63 \times 14$ data and was rendered with 200 slices. This AMR hierarchy was generated from an uniform confocal microscopy dataset with $749 \times 495 \times 100$ cells utilizing an opacity based importance criterion. Regions with associated opacity values below a certain threshold are represented at coarser resolution, based on the algorithm proposed in Section 4.4. Rendering the uniform dataset with the standard approach for texture-based volume rendering resulted in frame rates below 2 fps. The amount of texture memory in this case was 64 MBytes. (left) root level, (middle) full hierarchy, (right) associated bounding boxes. (*uniform dataset courtesy of R. Brandt and R. Menzel, Freie Universität Berlin)*

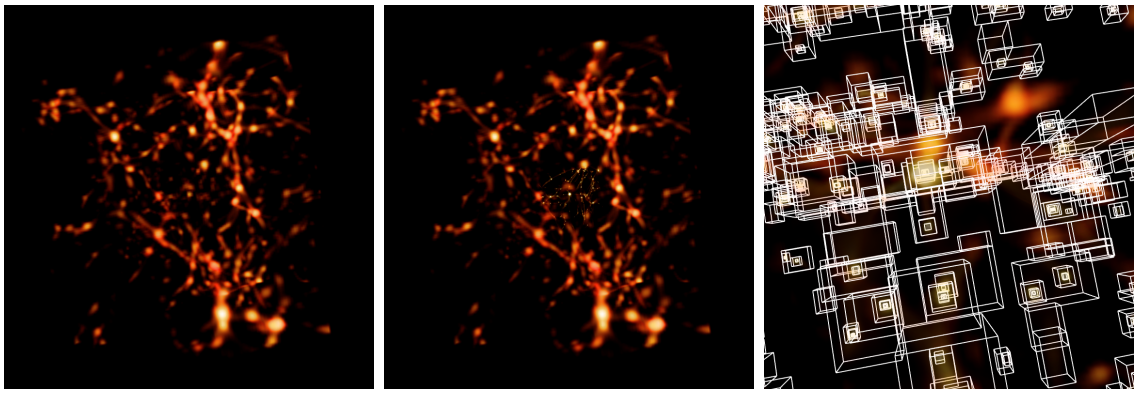

Figure 5.21: Dataset III is another AMR hierarchy resulting from a cosmological simulation that consists of 813 grids distributed on 9 levels of refinement. The (resampled) root grid contains 129^3 data samples and was rendered with 250 slices. If resampled to an uniform grid, the grid would contain about 66.000^3 data samples, resulting in an amount of 2.7×10^8 MBytes of texture memory. (left) root level, (middle) full hierarchy, (right) associated bounding boxes. *(dataset courtesy of G. Bryan, Princeton University)*

to 5.21. The last entry represents the frame rate achieved by rendering the full hierarchy in the mode described in Subsection 5.5.4, i. e. the subtree traversal is stopped, once the cells of the grid associated to the subtree root node have a screen space extension that is smaller than a pixel.

For all datasets (almost) interactive frame rates were achieved [1]. The frame rates were minimal for the close-up views, since the covered screen space is maximal for these viewpoints. As the performance results for the third dataset show, the subpixel criterion for block selection can result in significant performance gains. In general the effect is more pronounced for deep hierarchies with a large number of subgrids on the more refined levels. Rendering the datasets with the stencil buffer approach as discussed in Section 5.5.1 was about three times slower than the approach that utilizes the domain decomposition.

	root	**full**	**close-up**	**full adap.**
Dataset I (hardware)	10.4	6.7	2.0	7.2
Dataset II (hardware)	10.1	3.2	2.0	3.2
Dataset III (hardware)	6.5	1.4	1.1	2.4

Table 5.2: This table shows the frame rates for the root level data, the full hierarchy, the close-up view on the refined part and the view-dependent rendering.

[1] Notice that the frame rate are more than two times higher on actual graphics hardware.

5.6 Visualization of Time-Dependent AMR Data

For analysis purposes of scientific data, like animation, feature identification, or feature tracking, the underlying non-discrete time-dependent function has to be faithfully reconstructed from the grid function. This is done by spatial and temporal interpolation.

Appropriate interpolation methods may improve the results of data analysis greatly. Furthermore, they help to reduce the amount of grid data to be saved and handled. For instance in large numerical simulations this allows to store fewer time steps and nevertheless create smooth animations that display the underlying process without discontinuities or cracks. Even if numerical methods that provide *dense* output by maintaining and internally evaluating some interpolants are avaliable, it is usually unfeasible to store all the data due to the immense storage requirements. Only information that can not be regenerated by spatio-temporal interpolation methods should be saved.

In this section we describe an approach that allows the generation of dense output for time-dependent AMR data. In principle temporal interpolation approaches for unstructured grids like proposed by Polthier et al. [68], Happe et. al. [31] or Schmidt et al. [74], compare Subsection 5.1, could be applied also in the case of AMR data. These approaches require the storage of the grid structures and grid functions before and after grid adaption for each time step. In this case the temporal interpolation can be performed on pairs of identical grids.

But for realistic AMR simulations, which often contain dozens of refinement levels, and typically evolve several scalar and vector quantities, this would require to store huge amounts of data, because the temporal step size usually doubles between two consecutive levels of refinement, i.e. increases exponentially. AMR simulation codes therefore often store data only for time steps that correspond to root level updates.

We therefore propose a different approach in which intermediate grid hierarchies are generated in order to connect the given key-frames hierarchies. An imaginable approach for this would be to identify corresponding subgrids present in the key-frames, and to interpolate their position and sizes for the intermediate steps. But this would in general result in overlapping grids on the same level of refinement and it is also not clear how to proceed in case a subgrid has no corresponding 'partner' on the next frame. Instead we propose an approach that

- generates an intermediate grid hierarchy by merging the cells on each refinement level that is present in the key-frames,
- uses a clustering algorithm to induce a nested grid structure on the resulting collection a cells,
- projects the grid functions of each key-frame to such an 'merged' grid hierarchy and finally
- generates the intermediate grid functions by interpolating between each set of corresponding data samples on these 'merged' hierarchies.

5.6.1 Generation of Intermediate Grids

The temporal refinement scheme for AMR data was described in Subsection 2.2.4. The resulting change of the underlying grid structure complicates the interpolation of intermediate time steps during the visualization phase. We can propose the problem as follows

> Given a set of grid hierarchies and associated grid functions $\left(\mathcal{H}(t), f^l(t)_{l=0,...,n}\right)$ at discrete time steps $t_0, t_1, ..., t_m$, with potentially different topology, we want to generate intermediate grid hierarchies $\mathcal{H}(t)$, as well as interpolated grid functions $f^l(t)$ for $t \in]t_i, t_{i+1}[$.[2]

Let us first address the construction of the intermediate grid hierarchies. We make the following assumptions, which are usually fulfilled by the numerical schemes:

- The root-grid structure $\Lambda^0(t)$ remains constant for all time steps [3], and
- the spatial refinement factors between two consecutive levels $\left(\Lambda^l(t), \Lambda^{l+1}(t)\right)$ does not change in time.

In a first step, the refinement levels of the intermediate grid are generated. This is done by merging the subgrids for each level of the key-frame hierarchies:

$$\mathcal{H}(t) = \mathcal{H}(t_0) \cup \mathcal{H}(t_1) \cup ... \cup \mathcal{H}(t_m) \tag{5.3}$$

$$= \bigcup_{l=0}^{\bar{l}_{max}} \left(\Lambda^l(t_0) \cup \Lambda^l(t_1) \cup ... \cup \Lambda^l(t_m)\right). \tag{5.4}$$

Here $\bar{l}_{max}$ denotes the maximum number of refinement levels present in the set of considered time steps. By this merging of corresponding levels, in general we loose the subgrid structure present in the keyframe hierarchies, as illustrated in Figure 5.22. The resulting collection of cells on each level could be stored as an unstructured hexahedral grid with explicit connectivity information. But in terms of memory efficiency and performance it is more advantageous to reintroduce a structure of disjoint subgrids on these unions of cells, since the rendering algorithms work faster on blocks with implicitly given (trivial) connectivity.

So for each $\Lambda^l(t)$ of the intermediate hierarchy, we require a partition into axis-aligned, non-overlapping rectangular subgrids $\Gamma^l(t_i)$, such that $\Lambda_l(t) \subseteq \bigcup_i \Gamma_i^l(t)$. We achieve this by utilizing the clustering algorithm described in Subsection 2.2.3. In principle the clustering could be applied at once to all subgrids on the same level, but this would result in too high computational efforts for computing the signature lists, due to the large number of cells contained in the minimal bounding box enclosing the higher resolved levels. Thus we perform the clustering procedure per subgrid rather than per level.

[2] The number of key-frames required for this depends on the order of the interpolation function that is applied to obtain the associated grid function, compare Subsection 5.6.2.

[3] This assumption can be weakened. It just has to guaranteed that the cells on the coarsest level of refinement are not shifted or rotated against each other.

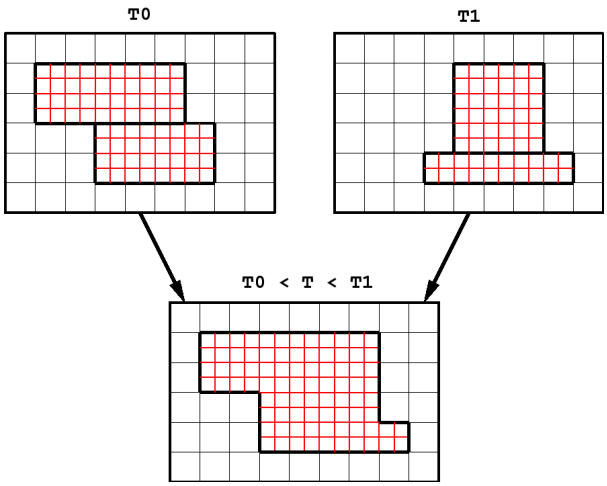

Figure 5.22: Top: Coarse grid and level 1 subgrids of two keyframes. Bottom: Coarse grid and level 1 union of subgrids for intermediate time steps (created by merging corresponding level l subgrids of each keyframe).

Per assumption Γ^0 remains unchanged, i. e. $\Gamma_0^0(t) := \Lambda_0^0(t) = ... = \Lambda_m^0(t)$. Suppose we want to generate the subgrids of a grid $\Gamma_q^l(t) \subset \Lambda^l(t)$. The signature list for its index field is initialized as follows:

$$S(i,j,k) = \begin{cases} 1, & \text{if } \exists\, t_s \in (t_0,...,t_m) \text{ with } \Omega_{ri,rj,rk}^{l+1} \subset \bigcup_{s=0}^{m} \Lambda^{l+1}(t_s) \\ 0, & \text{otherwise,} \end{cases} \tag{5.5}$$

that is, a cell in $\Gamma_q^l(t)$ is marked for clustering, if it is refined by the next finer level in at least one of the key-frames. This signature list is passed to the clustering algorithm, which generates a set of subgrids $\Gamma_0^{l+1}(t), \Gamma_1^{l+1}(t), ...$ that belong to the next level $\Lambda^{l+1}(t)$. This procedure is recursively repeated for each of the newly created subgrids, until the maximal level $\tilde{l}_{max}$ is reached.

5.6.2 Temporal Interpolation of Grid Functions

Next the vertex-, respectively cell-centered grid functions $f^l(t), \overline{f}^l(t) \colon \Lambda^l(t) \mapsto \mathbb{R}$ associated with the intermediate grid hierarchy $\mathcal{H}(t)$, are constructed. Two cases have to be distinguished for each cell $\Omega_{ijk}^l \in \Lambda^l(t)$:

- $\Omega_{ijk}^l \in \Lambda^l(t_s)\, \forall\, t_s \in (t_0,...,t_m)$,
- $\exists\, t_s \in (t_0,...,t_m)$ with $\Omega_{ijk}^l \notin \Lambda^l(t_s)$.

In the first case $f_{ijk}^l(t)$ can simply be computed from the set of given functions values $f_{ijk}^l(t_s)$ at the given keyframes. In the second case, in at least one of the given hierarchies no corresponding cell on the refinement level l exists, so we have to apply some

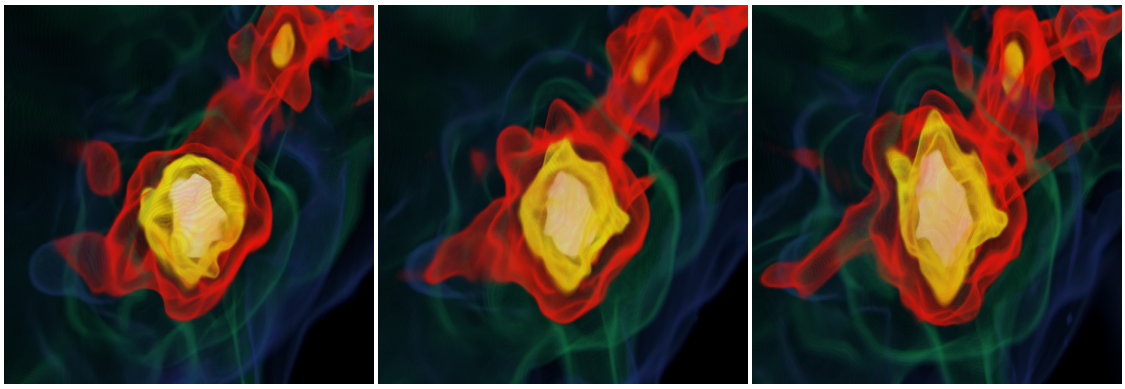

Figure 5.23: Two keyframes and an interpolated intermediate frame generated from dataset I, a galaxy-formation simulation, via texture-based volume rendering. *(dataset courtesy of T. Abel, Stanford University)*

form of interpolation on the grid function on the coarser levels of resolution to obtain it. Let us assume that this is the case for $\Lambda^l(t_s)$ in the following. Because of the nesting property of the levels, the cell is covered by at least one coarser cell. Let $\Omega^{\tilde{l}}_{\tilde{i}\tilde{j}\tilde{k}}$ denote the cell on the finest level $\tilde{l} \leq l$ that contains Ω^l_{ijk}. In the *cell-centered* case we perform a nearest-neighbor interpolation, i. e. $\overline{f}^l_{ijk}(t_s) := \overline{f}^{\tilde{l}}_{\tilde{i}\tilde{j}\tilde{k}}(t_s)$. For *vertex-centered* grid functions, $f^l_{ijk}(t_s)$ is obtained by trilinear interpolation within $\Omega^{\tilde{l}}_{\tilde{i}\tilde{j}\tilde{k}}$.

For larger hierarchies determining which of the two cases holds for each cell and collecting the associated data samples can be an expensive operation. In order to accelerate this procedure we resample each grid function $f^l(t_s)$ onto a grid with the topology of the intermediate hierarchy in a first step. This involves some interpolation in order to obtain data values for fine cells or vertices that are not present in the given set of hierarchies. After this preprocessing the grid function can be computed much more efficient, since now for each subgrid $\Gamma^l(t)$ there exist corresponding subgrids $\Gamma^l(t_s), \Gamma^l(t_{s+1}), ...$ This is especially advantageous if more than one intermediate time step for the same subset of key-frames has to be be generated.

We employ three different temporal interpolation schemes. The first one is $\mathcal{C}^0$-continuous piecewise linear interpolation

$$f^l_{ijk}(t) = \frac{t_{s+1} - t}{t_{s+1} - t_s} f^l_{ijk}(t_s) + \frac{t_s - t}{t_s - t_{s+1}} f^l_{ijk}(t_{s+1}) \tag{5.6}$$

The second one is $\mathcal{C}^1$-continuous cubic Hermite interpolation. So besides the function values for t_s and t_{s+1}, also the first derivative at these time steps has to be taken into account

$$f^l_{ijk}(t) := f^l_{ijk}(t_s)\, H^3_0(t) \quad + \quad \left(\frac{d}{dt} f^l_{ijk}(t_s)\right) H^3_1(t) + \tag{5.7}$$

$$f^l_{ijk}(t_{s+1})\, H^3_2(t) \quad + \quad \left(\frac{d}{dt} f^l_{ijk}(t_{s+1})\right) H^3_3(t). \tag{5.8}$$

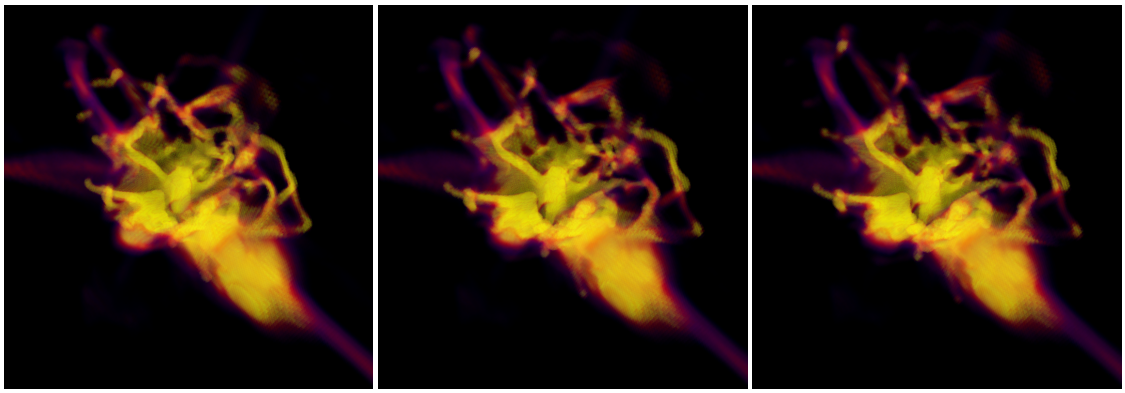

Figure 5.24: Three interpolated frames from dataset II, a simulation describing a supernova explosion visualized by texture-based volume rendering. *(data set courtesy of T. Abel, Stanford University)*

In case the first derivative of the grid functions is not available during the visualization phase, we generate a Catmull-Rom spline, as discussed in Section 2.3. This implies that we require the grid function values at four successive time steps in order to obtain an approximation of the first derivatives at t_s and t_{s+1}.

It is possible to obtain more precise values for the first derivative of the grid functions in the case that the grid function represents some conserved quantity that fulfills a conservation law of the form

$$\frac{\partial}{\partial t}\rho(\vec{x},t) = div\ \vec{j}(\vec{x},t). \tag{5.9}$$

Here $\rho(\vec{x},t)$ denotes the density of the conserved quantity and $\vec{j}(\vec{x},t)$ is the associated current. An common example is the case of mass conservation in hydrodynamic simulations. Here $\rho(\vec{x},t)$ denotes the mass density and the current is computed from the density and the velocity field $\vec{v}\,(\vec{x},t)$ via $\vec{j} = \rho\,\vec{v}$. Since usually in hydrodynamic simulations besides the mass, respectively density fields, also the associated velocity vector fields are stored, we can compute the derivative of the scalar field according to equation (5.9). The divergence of the current is approximated by the flux of the mass through each of the cells faces

$$div\ \vec{j} = div\ (\rho\,\vec{v}) = \frac{1}{V_{cell}} \sum_{i=0}^{5} \rho_i\, A_i\, \vec{n}_i\, \vec{v}_i. \tag{5.10}$$

$\rho_i, \vec{v}$ denote the density and velocity fields evaluated at the i-th face and $A_i, V_{cell}, \vec{n}_i$ are the face area, its volume and the outward-oriented face normals.

5.6.3 Results and Discussion

The performance was tested on a SGI ONYX3 system on a single 500 MHz MIPS R14000 processor. Dataset I is a result from a cosmological simulation of the formation of stars in the early universe with a root grid resolution of 128^3 cells, 8 levels of

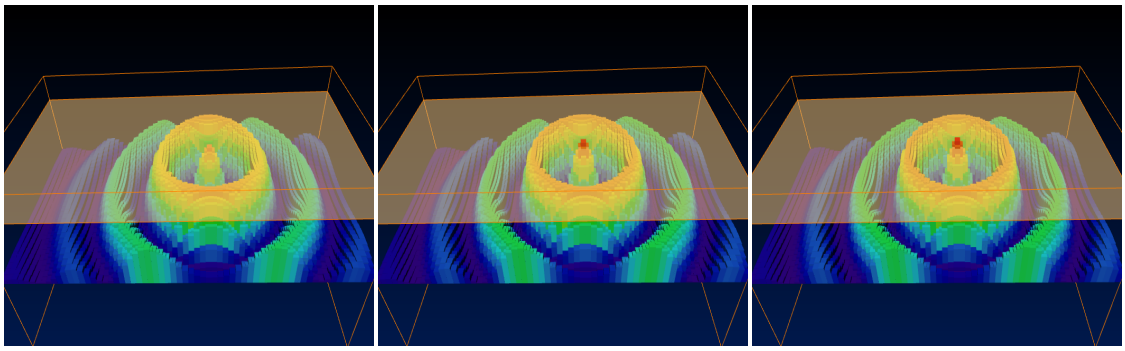

Figure 5.25: Comparison of three different time interpolation schemes for the analytical dataset III, a damped cosine wave traveling along the x-axis with constant velocity. For linear interpolation oscillations occur, whose minimal peak amplitude during the considered time interval is depicted by the semi-transparent plane. The oscillations decrease for the Hermite interpolation (middle image) and vanish for the flux-based approach (right image). The cell-centered data was rendered using constant spatial interpolation.

refinement and about 2000 grid per time step, compare Figure 5.23. Dataset II depicts a supernova explosion with 8 levels of refinement and about 1600 grids per time step. We took 10 time steps and generated 8 intermediate frames for each pair using linear and Hermite interpolation, with estimated first derivatives. Figure 5.24 shows some volume rendered images of the sequences. The resulting animations show slight oscillations in some parts for linear interpolation, which decrease for Hermite interpolation. Besides that the resulting animations are smooth.

For illustrating the differences of animation quality of the different interpolation schemes we choose an analytical example as dataset III. It shows a damped cosine oscillation that moves along the x-axis with constant velocity. The $64 \times 32 \times 32$ root grid is refined two times and contains about 1100 sub grids at each of the 20 keyframes. We compared the animation quality of linear, Hermite and flux-based interpolation by generating 8 intermediate frames per pair of time steps. Figure 5.25 shows volume rendered images of the resulting sequences. The animation shows disturbing oscillations for linear interpolation were visible. They decreased for Hermite interpolation with estimated derivatives and vanished for the flux-based interpolation, where the knowledge of the velocity vector fields are taken into account.

Information about performance and memory requirements is given in Figure 5.26. The number of subgrids in the merged hierarchies is decreased by about 20% compared to the number of subgrids present in the stored hierarchies. As can be seen in the table the amount of additional memory requirements and the times for grid generation where highest for the Hermite interpolation, since 4 keyframes had to be merged in this case. But the space increase was still less than 30% in all examples. The middle row depict the times for grid generation and keyframe projection, which has to be carried out only if the subset of keyframes used for the interpolation is changed. Again it was highest for Hermite interpolation, but with less than 7 seconds even for the 2000 grid dataset it

	increase of cells	grid generation	interpolation
Dataset I (linear)	7%	3.4 sec	0.2 sec
Dataset I (Hermite)	12%	6.6 sec	0.8 sec
Dataset II (linear)	11%	2.2 sec	0.1 sec
Dataset II (Hermite)	15%	3.5 sec	0.3 sec
Dataset III (linear)	20%	1.8 sec	0.1 sec
Dataset III (Hermite)	30%	4.2 sec	0.3 sec
Dataset III (flux-based)	20%	3.5 sec	2.0 sec

Figure 5.26: The first two columns denote the increase in the number of cells for the intermediate time steps relative to the given keyframes. The third column states the times for generating the intermediate grid and projecting the given grid functions. This has to be carried out only if the keyframes change. The last column gives the time for interpolation of the intermediate grid function.

still admits a on-the-fly generation during the visualization phase. Due to the keyframe projection step the times for the interpolation (right row) are short, which is advantageous if more than one intermediate frame is generated for a constant set of keyframes.

5.6.4 Future Work

There are several ways to extend the presented algorithm. Higher order interpolation schemes could be implemented for spatial interpolation during the prolongation step. Further it would be interesting to combine the presented approach with feature tracking algorithms. Also it seems promising to adapt the order of temporal interpolation to the rate of change of the underlying data. It might be beneficial to use lower order temporal interpolation for subgrids with slowly varying data and higher order interpolation for subgrids with rapidly changing data (which is usually the case for the higher resolved levels).