Visualizing Multidimensional Raster Data with rView †

Andreas Dehmel
FORWISS
Orleansstr. 34
D-81667 Munich, Germany
(dehmel@forwiss.tu-muenchen.de)

Peter Baumann
Active Knowledge GmbH
Kirchenstr. 88
D-81675 Munich, Germany
(baumann@active-knowledge.de)

Abstract

rView is a visual frontend to the RasDaMan DBMS, providing raster data visualization functionality and a graphical user interface to the database system. RasDaMan, a commercial array DBMS whose prototype has been developed by FORWISS, is designed for raster data of arbitrary dimensionality and base type, which makes it a powerful storage system for all types of rastered data. This generality calls for a similarly flexible visualization tool for data stored in the database. rView has been designed as a generic visualization tool with very few restrictions regarding dimensionality and base (i.e., array cell) type; among other things, it provides several visualization techniques like texture mapping, voxel- and heightfield rendering, which may be combined with user-configurable mappings to the RGB colour space.

Keywords: multidimensional raster data, information visualization, visual database frontend.

1. Introduction

Raster data of arbitrary dimensionality and base type, which we call Multidimensional Discrete Data or MDD for short, appears in many application areas, typically in the form of sampled analogue data. Examples for this kind of data are time series like sound samples, images of varying number and depth of spectral components, movies or volumetric data like tomograms, spatio-temporal simulation data or datacubes in data warehousing and OLAP. The traditional approach of managing this kind of data in databases is one of two extremes: either mapping each single array cell to a relational tuple for “sparse” data or writing the array in its entirety into BLOBs for “dense” data. In practice, however, in many areas like scientific computing the file system still is the preferred way of storage.

The goal of the RasDaMan DBMS [4] is to provide uniform, efficient MDD management across all application domains and independent from data-inherent properties such as sparsity of an array. RasDaMan clients use RasQL, an extension to SQL-92 [10], to execute operations on the server, examples of which are range queries, aggregate functions (condensers), induced operations or histograms and filtering. Internally, MDD objects1 are stored as sets of non-overlapping tiles, i.e., subarrays, [8] to improve retrieval performance when only parts of the object need to be accessed. RasDaMan stores meta information about the spatial extent and the base type of the MDD objects which makes it possible to preprocess data on the server and send only the results to the client, rather than shipping the entire object which might well be in excess of several Gigabytes. For further information see [3, 4, 8], which cover various aspects of RasDaMan in more depth.

rView is an easy-to-use graphical front-end to the RasDaMan DBMS which allows to phrase queries against a database and display the resulting MDD sets. The tool offers a set of raster data visualization techniques, naturally restricted to the 1D to 3D data sets displayable on screen. The visualization techniques were designed to handle a large variety of MDD structures and pixel types efficiently and to allow the user to get a good first view on his data. Naturally the generality of rView also implies that specialized solutions may deliver better results in their resp. fields, e.g. a voxel renderer written specifically for preprocessed 3D medical volume data [16] will almost certainly produce higher-quality images; in the case on hand, however, the goal was to develop a tool as universal as possible, similar to the database system itself which supports a large number

1The term “object” is used in a straightforward sense here, i.e. a data item bearing a persistent identifier.
of different applications ranging from geo and neuro sciences to supercomputing. The natural restriction of rView in the number of MDD dimensions does not mean that higher dimensions can not be addressed. For example, while the current implementation of the voxel renderer can not handle more than three dimensions, the RasDaMan server can well project higher-dimensional data to 3D results for display. The same applies to complex base types where RasQL queries allow to extract components such as the RGB channels from a 7-band Landsat image. Notably all this server-based information extraction contributes significantly to reducing server/client traffic.

We will use the following terminology:

**Spatial Domain:** a multidimensional interval spanned by two vectors \( l, h \). We use the syntax \([l_1 : h_1, \ldots, l_d : h_d]\), \( l_i, h_i \in \mathbb{Z} \) to represent a spatial domain in \( d \) dimensions where \( l_i \leq h_i \), \( i = 1, \ldots, d \). The asterisk * has special meaning when defining a spatial domain, in case \( l_i = * \) it means \(-\infty\), in case \( h_i = * \) it means \(+\infty\).

**Base Type:** corresponds to types in high level languages. There are atomic types (char, short, int, float, ...) and structured types which may contain atomic and structured types. The only restriction on base types is that they must have constant length, e.g. variable character strings are not allowed.

**MDD:** Multidimensional Discrete Data. An MDD object is instrumented with a spatial domain and a base type.

**Cell:** A cell is located at every position within the spatial domain of an MDD object.

**Collection:** a set of MDD objects with similar properties\(^2\).

2. RasDaMan Data Model and Retrieval

The data model of RasDaMan extends some general data model - be it relational or object-oriented - with array capabilities based on an array algebra [3]. To this end, the model provides typed multidimensional arrays and collections (i.e., sets) thereof. Array dimension, boundaries (fixed or variable), and cell type are laid down in the type definition. This type information is not only used for semantic query checking at runtime, but it also is useful for optimisation purposes.

Each array instance carries a unique persistent OID. This way, arrays smoothly embed into both relational and object-oriented data models: in relations, OIDs can be used as foreign keys in tuples, whereas in object-oriented models object references to arrays can be expressed conveniently. The RasDaMan query language, RasQL, supports retrieval on array sets based on the classical select/from/where paradigm of SQL. To this end, declarative operators have been added to SQL which allow for array post-processing as well as array search predicates. For details on the expressiveness of RasQL the interested reader is referred to [3]. rView allows to create, retrieve, and delete MDD collections. Retrieval of MDD object sets for visualization can be done either by loading entire collections or by interactively formulating a RasQL query which allows specifying arbitrary areas of interest and base type projections (e.g. retrieve only the green component of an RGB image). MDD objects loaded from the database are displayed in a results window which is the starting point for all transformation and visualization functionality offered by rView. Special support is provided for atomic base types and the particular structured type struct {char red, char green, char blue} RGBPixel. More complex operations, like for instance calculating a vector field of the absolute velocity out of the vector fields of the velocities in the three spatial directions in a Computational Fluid Dynamics (CFR) simulation, can be formulated interactively as RasQL queries.

3. Visualization Techniques

rView offers a range of raster visualization techniques suitable for a large number of MDD types. These include the more abstract, tabular view on the data, several chart formats and sound playback, but the main focus is on creating images using various techniques such as orthogonal projections, texture mapping, voxel- and heightfield rendering. Only texture mapping and voxel rendering have strict limits on the dimensionality\(^3\), all other modes require a minimum number of dimensions but can cope with any number of dimensions larger than this minimum. The treatment of base types is not as generic for most visualization techniques, as only table mode supports totally arbitrary base types whereas the others can only handle atomic types and RGBPixel. This is no real limitation, however, since type projection (which can be performed by both RasDaMan and rView) makes it possible to transform any type of MDD object into an MDD object that is understood by all of rView’s visualization techniques.

There are always physical limitations on the number of dimensions that can be visualized simultaneously, like for instance the display hardware which limits the dimensionality to 2D or simulated 3D. Therefore all visualization techniques have to limit the number of dimensions visualized at the same time by projecting coordinates of those dimensions of the MDD object exceeding the display mode’s capacity to fixed values. In addition it’s often desirable to clip

\(^3\)This limitation will be removed in the near future if time permits.
the region to visualize, e.g. display a visualization of the area [100:200, 100:200] of a [0:1000, 0:1000] object. Both projection and clipping is done using a projection string, which is the one element shared by all visualization modes. A projection string is a list of comma-separated specifiers for each dimension, where each specifier can have the form l/h for a range selection with l and h being either numbers or the asterisk for the smallest / highest possible value, or the form p for a projection to coordinate p. Using this system, arbitrary projections and clippings of MDD objects can be performed interactively by the user. In the following sections we will only present some image visualization modes due to space limitations.

### 3.1. Height Field

The height field renderer interprets MDD values over a 2D carrier as height information, which makes the resulting object 3D. The positions of the support cells form a regular 2D grid \((g_x, g_y) = (\delta_y(x_i - l_i), \delta_y(x_j - l_j))\) where \(\delta_y\) is the grid width and \(i, j\) are the two support dimensions, i.e. \(x_k = \text{const}\) for \(k \neq i \land k \neq j\). With the height scaling factor \(\delta_h\) the height field consists of the vertices \(V = \{ (\delta_y(x_i - l_i), \delta_y(x_j - l_j), \delta_h M(x) ) | l_i \leq x_i \leq h_i, l_j \leq x_j \leq h_j \}\). This set of vertices is used to create a triangle mesh which is then fed into a polygon rendering engine. Each set of 4 neighbouring vertices is used to span 2 triangles, with 2 possible split diagonals. In order to choose the best split diagonals for the triangulation and to render the height field using shaded polygons we also need approximations for the normals at the grid vertices \((g_x, g_y)\). A tangent vector in direction \(i\) at \(g_x + \frac{g_x}{2}\) can be approximated with the standard finite difference \((\delta_y(0, \delta_h(M(x + e_i) - M(x)))\), with \(e_i\) the \(i\)th unit vector; this leads to a normal vector \((-\delta_h(M + e_i) - M(x)), 0, \delta_h\) or any multiple thereof. Since this is the normal approximation halfway between grid vertices, but we need approximations at the grid vertices themselves, we calculate \(n_x(x)\) as the arithmetic average of the normal approximation at \(g_x + \frac{g_x}{2}\) and \(g_x + \frac{g_x}{2}\) except at the grid boundaries where only one approximation is defined; \(n_y(x)\) is treated analogously. The total normal at \(x\) is then calculated as the normalized vector sum, \(n(x) = \frac{n_x(x) + n_y(x)}{\|n_x(x) + n_y(x)\|}\). The resulting normal vector field can now be employed to create a smooth triangulation by choosing that splitting diagonal whose scalar product with the normals at its endpoints is minimal.

In the rendering process the normals are also used to calculate the light intensities at the surface vertices by computing the angle \(\alpha\) between the normal in vertex \(v(x) \in V\) and the vector from \(v(x)\) to the light source \(L\) for each vertex. There are two lighting models available, a fast and simple but unphysical variant and a more complex model described in Section 3.4. Finally, the rendering engine plots the triangles using Gouraud shading for smooth colour gradients between vertices. An example of heightfield rendering can be seen in figure 1. For objects with more than 2 dimensions the heightfield can also be animated by automatically iterating over a third, user-specified dimension.

![Figure 1. The height field renderer used on a digital elevation map](image)

To the top is the original data, a DEM of the Rocky Mountains with the domain \([0:456, 0:446]\), visualized as greyscale image where brighter colours correspond to higher points. To the bottom the same data is visualized as a height field; in order to cut down the number of polygons, the MDD object was first downsampled by a factor of 0.1, reducing the number of polygons to 3960.

### 3.2. Texture Mapping

The texture mapper can operate on 3D data only; in contrast to conventional texture mapping algorithms which map 2D textures onto 3D surfaces [20], this variant uses a 3D data cube as texture. The MDD object is represented by a 3D cube; the cells that are intersected by the cube’s faces are used as textures for those faces, including faces created by depth clipping which are not necessarily rectangular or orthogonal to any coordinate axis anymore. Extracting the texture on the cube’s unclipped surface for use in a standard texture mapping library is trivial, however extracting the texture to use on faces resulting from clipping would require an algorithm very similar to an actual texture map-
per; therefore it was decided to implement the entire algorithm in software. While this visualization technique is fast enough for smooth animation even without hardware acceleration it has the drawback that it can’t visualize data within the MDD object unless depth clipping is applied. It is therefore not meant for visualizing volumetric data, but it can help a lot by allowing fast positioning, rotation and clipping of the MDD object before switching to a more complex visualization technique like the voxel renderer.

The texture mapping algorithm requires calculating the world coordinates of each pixel on the cube’s surface and transforming them to the cube’s internal coordinates (texture coordinates) to determine the texel to map to the surface. This algorithm can be optimized enormously by using a scanplane approach: the scanplane for a given scanline $y_p$ is spanned by the first unit vector and a vector from the viewer to the scanline $y_p$ in the viewplane at $z = z_p$, i.e. the vectors $(1,0,0)^T$ and $(0,y_p,z_p)^T$. Let $V = \{v_i = (x_i,y_i,z_i)^T, 1 \leq i \leq n\}$ be the list of $n$ ordered vertices spanning a face to render, already clipped to a clipping plane at $z = c_2 > 0$, which we extend by appending the first vertex to the end of the list, i.e. $v_{n+1} := v_1$. We determine the potential intersection (world) coordinates of the face’s edges $e_i$ connecting two consecutive vertices $v_i$ and $v_{i+1}$ as

$$v_i^{is} = (x_i^{is}, y_i^{is}, z_i^{is})^T = v_i + \tau_i(v_{i+1} - v_i), \quad 1 \leq i \leq n,$$

where $\tau_i = \frac{y_i z_p - y_p z_i}{y_p(z_{i+1} - z_i) - z_p(y_{i+1} - y_i)}$, provided the denominator is not 0, in which case the edge is parallel to the scan plane and may be ignored. Those edges where $\tau_i \in [0,1]$ actually intersect the scanplane; assuming convex faces and provided the face intersects the viewplane at all, there will always be exactly two edges $e_l$, $e_r$ that do, where the perspective projection of $v_l^{is}$ is left of the one of $v_r^{is}$, i.e. $\frac{x_l^{is}}{z_l^{is}} \leq \frac{x_r^{is}}{z_r^{is}}$ or without the division $x_l^{is} z_r^{is} \leq x_r^{is} z_l^{is}$. These intersection points are then translated into texture coordinates $v_l^{tx}$, $v_r^{tx}$ using the transformation $t(v) = B(v - o)$ where $o$ is the origin of the texture coordinate system in world coordinates and $B$ is a matrix in $\mathbb{R}^{3 \times 3}$ performing the base transformation from the world coordinate system into the texture coordinate system\(^5\). Analogously, the world coordinates of a point $v$ on the face’s surface that is projected to position $x_p$ in scanline $y_p$ can be calculated as

$$v = v_l^{tx} + \mu(v_r^{tx} - v_l^{tx}) = v_l^{tx} + \mu \Delta v^{tx},$$

where $\mu = \frac{x_p z_l^{is} - x_l^{is} z_p}{x_l^{is}(z_r^{is} - z_l^{is}) - z_p(x_r^{is} - x_l^{is})}$, provided the denominator is not 0, in which case the edge is projected to a point and may be ignored. To get the texture coordinates for this position we don’t have to do the expensive base transformation explicitly, although $t(v)$ is not a linear operation, but can simply apply $\mu$ to the corresponding texture coordinates:

$$v^{tx} = t(v_l^{tx} + \mu(v_r^{tx} - v_l^{tx})) = B(v_l^{tx} + \mu((v_r^{tx} - o) - (v_l^{is} - o))) - o) = B(v_l^{is} - o) + \mu(B(v_r^{is} - o) - B(v_l^{is} - o)) = v_l^{tx} + \mu(v_r^{tx} - v_l^{tx}) = v_l^{tx} + \mu \Delta v^{tx}.$$  

$v^{is}$, $\Delta v^{is}$, $v^{tx}$ and $\Delta v^{tx}$ need only be calculated once for each scanline. That reduces the complexity of computing $v^{tx}$ to 1 division and 7 multiplications per pixel, in contrast to the naive approach which would require solving a 3D equation system and a base transformation for each pixel. A further optimization calculates $v^{tx}$ only every $m$ pixels and does a linear interpolation in between, resulting in another substantial speedup without noticeable distortions for small $m$. As a consequence the texture mapper is easily fast enough for smooth animation.

### 3.3. Voxel Renderer

The voxel renderer was designed to visualize volumetric data like tomograms, simulation data etc. It uses a raycasting algorithm i.e. for each pixel it shoots a ray from the observer through the MDD object, the ray is traversed from the position of the observer to infinity, accumulating colour and opacity information from all cells it intersects [21]. Once opacity has reached a user-defined threshold value the ray is terminated and an average of the colours accumulated so far is used as the colour for the pixel currently processed. Simply using the colour of the frontmost, non-empty cell fails to produce any meaningful results for MDD data with even low amounts of noise.

Because the voxel renderer has to be generic, there are a lot of parameters that can be changed by the users to adapt the renderer to the data; rendering a CFD vectorfield requires totally different parameter values than rendering a tomogram, for instance\(^5\). As an example, the voxel renderer can operate on floating point data directly, as demonstrated in figure 4 where it was used to visualize 4D climate data. Most importantly, the parameters concerning ray termination are

- $c_l$, $c_r$: lower and upper threshold values for the cells. Cells whose values lie outside this threshold range are interpreted as totally transparent.
- $q_w$: the weight quantisation that determines the correlation between cell values and cell weight. The cell weight is the cell value multiplied by $2^{-q_w}$; the cell’s

\(^5\)Synthetically created data usually doesn’t contain substantial amounts of noise like sampled data does, for a start.
opacity and colour is scaled by the cell weight. This system approximates exponential absorption laws.

- \( w_{\text{max}} \): the weight threshold. When the sum of cell weights has reached this value, the ray is terminated.

If lighting effects are desired, an approximation for the surface normal in the termination cell is required \cite{12}. The usual way to approximate the surface normal is to use the normalized gradient \( \nabla M \). This is very problematic in noisy data: if the cell- and weight threshold are high enough to avoid termination in noise, they often lead to the ray penetrating into the volume so deep that the gradient in the termination cell differs considerably from the actual surface, resulting in random distributions of the normal vectors. The effect can be ameliorated to a certain degree by using a larger number of cells to approximate the normal vector. We chose to use the cube \( \omega(x, s) \) spanned by the space diagonal from \((x_1 - s, x_2 - s, x_3 - s)\) to \((x_1 + s, x_2 + s, x_3 + s)\) as the kernel with size \( s \) of the cell at position \( x = (x_1, x_2, x_3) \). In addition there are a number of rotation-symmetric kernel functions \( f_k(r) \) which are convoluted discretely with the volume data. There are currently three kernel functions available:

- \( f_k^{avg}(r) = 1 \), a function that averages over all cells within the kernel;
- \( f_k^{gauss}(r) = e^{-r^2} \), a rotation-symmetric gaussian;
- \( f_k^{lin}(r) = 1 - \frac{r}{s\sqrt{3}} \), a rotation-symmetric hat function.

where \( r = \|x\|_2 \). The discrete kernel function can be easily implemented as a lookup table, thus improving efficiency in most cases and making the folding algorithm independent of the kernel function used. Figure 2 shows a tomogram rendered using \( f_k^{gauss} \) with a kernel size of 2.

With the weighted sum of pixel colours \( c \) and the sum of cell weights \( m \), the algorithm for one pixel is as follows:

1. Initialize \( c \) and \( m \) to 0. Determine \( r_f \) and \( r_b \), the texture coordinates of the cells on the surface of the volume bounding box intersected by the ray in front and in back.

2. Traverse the volume from \( r_f \) to \( r_b \) in \( \|r_b - r_f\|_\infty + 1 \) steps; this is the smallest number of steps where the coordinates from one position to the next change by at most 1 in all dimensions.

3. If the cell value \( v \) at the current position is within the threshold range \([c_l, c_h]\): calculate the cell weight \( w \) as described above. If the MDD base type is an integer value, apply ceiling rounding. Update the weighted sums using \( c = c + wv \) and \( m = m + w \).

4. If \( m \geq w_{\text{max}} \): terminate the ray. The pixel (base) colour is \( \frac{c}{m} \). If lighting is to be used, approximate the normal vector in the termination cell using the approach presented above and transform the base colour as described in Section 3.4. In this case there’s also the option to ignore the base colour at this point and use the same, user defined base colour for all pixels; this variant should be used if only the location of the surface is of interest rather than its texture, i.e. the cell values are only used for ray termination.

The raycasting algorithm can be optimized in a way similar to the texture mapper. The intersection points of the cube’s faces with the scanplane can be calculated as in (1), resulting in an intersection plane consisting of \( j \) vertices.
\[ \{ v_i^{xz} = (x_i^{xz}, y_i^{xz}, z_i^{xz})^T, 1 \leq i \leq j \}. \] For each horizontal position \( x_p \) in scanline \( y_p \), the world coordinates of the intersection points of the ray with the cube’s faces can then be calculated as in (2) and the transformation to texture coordinates is given by (3) again. What makes the voxel renderer considerably more complex than the texture mapper is the iteration between the two intersection points which means a large part of the volume must be swept by rays. Depending on the fill factor of the data cube, rendering times can vary substantially: if the cube contains a lot of cells outside the cell thresholds, the rays will on average take much longer to terminate, thereby elongating the rendering process. Typical rendering times for cubes around the 10MB mark and images of about 300*300 pixels are in the general area of 1–2 seconds on modern hardware.

![Figure 3. The voxel renderer visualizing an ear](image)

The area containing the ear was first determined as \([120:190, 120:190, 0:20]\) using the texture mapper from Section 3.2. This area was then upsampled by a factor of 4 in all dimensions, resulting in an MDD object with the spatial domain \([120:403, 120:403, 0:83]\) which was then visualized using the voxel renderer. The image to the top was created without lighting, the one to the bottom with light coming from the left. The entire head can be seen in figure 2.

### 3.4. Light model

The light model used optionally by the voxel- and the height field renderer is designed to produce realistic images rather than aiming for total physical accuracy; also considering the generality of MDD, “physical accuracy” is rather meaningless in some cases\(^6\). The colour \( c \) of a pixel is calculated from its base colour \( c_0 \) and the orientation of the surface it belongs to relative to the light source. Let \( \alpha \in [-\pi, \pi] \) be the angle between the surface normal and the vector from the pixel to the light source\(^7\), \( \beta \) the maximum value of \( |\alpha| \) where the light source should have any effect on \( c \) and \( \gamma \) the maximum value of \( |\alpha| \) where white light should be added to \( c \). Then we can define the weight function template

\[
 w(\alpha, \xi) = \begin{cases} 
 \frac{\cos \alpha - \cos \xi}{1 - \cos \xi} & \text{if } \cos \alpha > \cos \xi \\
 0 & \text{otherwise}
\end{cases}
\]

and the weighting factors \( w_f(\alpha) = w(\alpha, \beta) \) and \( w_h(\alpha) = w(\alpha, \gamma) \). Both weighting factors are in \([0, 1]\) and are maximal where \( \alpha = 0 \), i.e. when the light hits the surface at a right angle. With the ambient light level \( a \in [0, 1] \), a gain factor \( w_g \) regulating the addition of white light and \( c_g \) the projection of \( c_b \) to greyscales, a pixel is shaded using the following equation:

\[
 c = c_b \cdot \left( a + w_f(\alpha)(1 - a) \right) + c_g \cdot \gamma \cdot w_h(\alpha). \tag{4}
\]

The first part of (4) describes the darkening of the colour depending on the orientation of the surface relative to the light source; the second part models the addition of white light and thus simulates the way some surfaces appear white when hit directly by the light source, irrespective of their actual colour.

### 3.5. Colourspace Mapping

In many cases the cell values \( x \in D \) can’t be used as colours themselves but have to be mapped to pseudo-colours representing the cell values \([9]\), for instance for floating point base types. The approach chosen here is mapping a value \( x \) to the RGB colour space with a function \( f_{cm}(x) : D \rightarrow \mathbb{R}^3 \), using the same transfer function \( f_{tf}(t(x), \mu, \sigma) : [0, 1] \rightarrow [0, 1] \) for all three colour components, where \( t(x) : D \rightarrow [0, 1] \) is a normalization of \( x \), \( \mu \) describes the localization and \( \sigma \) is the distribution. Then

\[
 f_{cm}(x) = \begin{pmatrix} r(x) \\ g(x) \\ b(x) \end{pmatrix} = \begin{pmatrix} f_{tf}(t(x), \mu_r, \sigma_r) \\ f_{tf}(t(x), \mu_g, \sigma_g) \\ f_{tf}(t(x), \mu_b, \sigma_b) \end{pmatrix}.
\]

\(^6\) However, there is nothing particularly physical about viewing volumetric scalar fields, which do not correspond to the light emission of physical objects. Consequently, in visualization we are usually interested not so much in “physical realism” as in well-defined mappings of the equation parameters onto meaningful visual quantities. . . .” [13]

\(^7\) We use a point light source rather than one with spatial extent.
Figure 4. Two examples for visualizing 4D climate data

These images show visualizations of 4D climate data over a floating point base type. The image to the top is a 3D hypercube visualized with the voxel renderer (Section 3.3), followed by colourspace mapping (Section 3.5). The image to the bottom was created from a 2D projection with the heightfield renderer (Section 3.1), using the full light model from Section 3.4.

rView offers 4 transfer functions with different characteristics:

- \( f_{\text{gauss}}^{t}(t, \mu, \sigma) = e^{-\frac{(t-\mu)^2}{2\sigma^2}} \) is a standard Gaussian function with mean value \( \mu \) and variance \( \sigma \)
- \( f_{\text{hat}}^{t}(t, \mu, \sigma) = \begin{cases} 1 - \frac{|t-\mu|}{\sigma} & |t-\mu| < \sigma \\ 0 & \text{otherwise} \end{cases} \)
  is a hat function centered at \( \mu \) with width \( 2\sigma \)
- \( f_{\text{rect}}^{t}(t, \mu, \sigma) = \begin{cases} 1 & |t-\mu| < \sigma \\ 0 & \text{otherwise} \end{cases} \)
  is a rectangular pulse centered at \( \mu \) with width \( 2\sigma \)
- \( f_{\text{asympt}}^{t}(t, \mu, \sigma) = \begin{cases} 0 & \mu < t \\ 1 - e^{-\frac{t-\mu}{\sigma}} & \text{otherwise} \end{cases} \)
  is a smooth, asymmetric function starting at \( \mu \) and converging towards 1 for \( t \to \infty \).

The values of the \( \mu_c \) and \( \sigma_c \), \( c \in \{r, g, b\} \) can be set by the user as explicit numerical values. A graphical user interface also allows visualizing the transfer functions and modifying their parameters by dragging the curves with the mouse, with optional immediate refresh of the visualized data.

The normalization function \( t(x) = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \) requires \( x_{\text{min}} \) and \( x_{\text{max}} \), the minimum and maximum value of the data to visualize. Apart from manual specification there are 3 ways to set \( x_{\text{min}} \) and \( x_{\text{max}} \) currently supported by rView:

- Full range: use the total range allowed by the base type. Usually not a good approximation, but doesn’t require analysis of the MDD values;
- MDD range: use the range of values actually present in the MDD object. Good global mapping, but the MDD values must be analyzed.
- Projected range: use the range of values present in the currently visible projection of the MDD object. Good local mapping which requires the MDD values in the currently visible projection to be analyzed, but makes it impossible to relate the colours different projections to each other.

In case \( x \) is outside \([x_{\text{min}}, x_{\text{max}}]\), which is possible if \( x_{\text{min}} \) or \( x_{\text{max}} \) were set manually, it’s clipped to the nearest legal value.

4. Related Work

Related work in the database field can be found as specialized solutions and in object-relational database systems research. Many special-purpose systems have been built, such as QBISM [23] and Visible Human interfaces [19]. They differ from RasDaMan and rView in several respects: they are streamlined to particular data structures (such as 3D integer images) and applications [23] and frequently even particular data sets [19]. Furthermore, they sometimes operate on single MDD items only as opposed to RasDaMan where retrieval allows to state ad hoc selection predicates on MDD sets. Object-relational database technology attempts to extend the relational model with user-defined attribute types and corresponding operations. Arrays, however, are not an attribute type, but a type template which has to be instantiated with the spatial extent (dimension and upper/lower boundaries per dimension) as well as the cell type. As object-relational systems cannot handle such templates, a new type has to be implemented by the user for every particular situation. Further, correctness of the operations’ implementation is completely up to the developer; RasDaMan, on the other hand, deduces the complete query functionality from the type definition, hence there is (i) no programming effort and (ii) no danger of compromising the server. Furthermore, RasDaMan allows arbitrary tiling, which reduces not only the amount of data to transfer but also the amount...
of data the server has to touch, as well as various compression methods that may be applied to these tiles, including wavelet techniques, which allow not only the compression of binary masks ("regions" in QBISM) but any kind of MDD in a generic fashion.

Visualization of database contents or retrieval results has received much attention [22, 11]. Beside visual support for query formulation [1], a focus has been on mapping relational tuples and their resp. attribute values to some visual appearance. Arrays, however, differ from relational data in that array cells have a clearly defined "spatial" neighbourhood whereas relational tuples obtain such a neighbourhood only through a "spatial" interpretation. Consequently, techniques such as Circle Segments [2] where this proximity is not preserved are not accepted by the user community.

On the visualization of spatial 2D/3D data, pioneer work has been done by Tufte [17, 18], and Bertin [5]. Higher-dimensional spaces are addressed by Worlds-within-Worlds [7] where n-D spaces are partitioned by nesting lower-dimensional (max. 3D) coordinate systems for display. Contrary to this, rView employs the general RasQL query mechanisms to obtain a (user-defined) mapping to displayable dimensions. Besides, rView offers more flexibility in the rendering and display modes, based on techniques which have been developed in imaging and computer graphics and which have been adapted to the specific database needs.

For instance QBISM was limited to 3D medical images, whereas the combination of RasDaMan and rView can visualize data of any dimensionality and complex base type, not only the kind encountered in medical imaging. Furthermore their visualization techniques were surface-based only, whereas rView’s voxel renderer can also visualize internal structures of MDD objects. We therefore consider the combination of rView and RasDaMan a far more generic approach to the problem than QBISM.

5. Conclusion

We presented the raster data visualization tool rView, a frontend to the RasDaMan DBMS. It allows visualizing a wide range of MDD types as they appear in typical RasDaMan databases in up to 3D modes, independent of their number of dimensions and the base type. Data can be preprocessed by scaling with various algorithms, among other operations. Visualization techniques range from rather abstract approaches like tables and charts to voxel rendering of floating point volumetric data followed by colourspace mapping with the aid of user-configurable transfer functions. This set of visualization techniques has proven a very powerful tool in the use and demonstration of the RasDaMan DBMS to audiences from many different application areas and will be expanded and improved in the future according to our requirements. Of course, rView is not intended to be used as a domain-specific end-user tool; it rather comprises a database demonstration tool and a library of building blocks for the development of streamlined interfaces.

References


