Visually Mining the Datacube using a Pixel-Oriented Technique

David Auber, Noël Novelli, Guy Melançon

Visually Mining the Datacube using a Pixel-Oriented Technique

David Auber
CNRS UMR 5800 LaBRI
Bordeaux, France
David.Auber@labri.fr

Noël Novelli
CNRS UMR 6166 LIF
Marseille, France
Noel.Novelli@lif.univ-mrs.fr

Guy Melançon
INRIA Futurs / CNRS UMR 5506 LIRMM
Montpellier, France
Guy.Melancon@lirmm.fr

Abstract

This paper introduces a new technique easing the navigation and interactive exploration of huge *multi-dimensional datasets. Following the pixel-oriented paradigm [8], the key ingredients enabling the interactive navigation of extreme volumes of data rely on a set of functions bijectively mapping data elements to screen pixels. The use of the mapping from data elements to pixels constrain the computational complexity for the rendering process to be linear with respect to the number of rendered pixels on the screen as opposed to the dataset size. Our method furthermore allows the implementation of usual information visualization techniques such as zoom and pan, anamorphosis and texturing. As a proof-of-concept, we show how our technique can be adapted to interactively explore the Datacube, turning our approach into an efficient system for visual datamining. We report experiments conducted on a Datacube containing 50 millions of items. To our knowledge, our technique outperforms all existing ones and push the scalability limit close to the billion of elements. Supporting all basic navigation techniques, and being moreover flexible makes it easily reusable for a large number of applications.*

1. Introduction

The design of new visualization methods and tools becomes even more necessary with the continuously increasing volume of available data, which poses a problem that obviously cannot be solved by relying solely on the increase of CPU power. According to the "How much information" project developed at Berkeley, one exabyte of data (1 million terabytes) was produced in 2001, with 99,997% being exclusively available digitally (see (Keim 2001)). In 2003, that quantity seen as individual data production corresponded to 800 megabytes per person in one year on the whole planet (Peter and Varian 2003). A number of research fields now contribute in their own way to the design of methods and tools to exploit this richness of information, among which visual approaches experience growing success.

This abundance of information of course brings its load of questions and problems to solve. Companies most often store their data in structured database making it possible to couple visualization applications with their own information systems, helping users to interactively query the database.

The volume of data is but only one of the issues that must be addressed here. Indeed, in addition to the dataset size, the number of possible queries the user can run on the dataset grows exponentially with respect to the number of attributes. The Datacube is a device intrinsically capturing that complexity. Put simply, a Datacube encodes all the possible OLAP (On Line Analytical Processing) queries that can be processed on a database. The cost of (interactive) OLAP queries is relieved by pre-computing all possible logical combinations of aggregate functions on a database. OLAP queries are used to forage datawarehouses where data is usually stored chronologically. Typical application domains of OLAP are marketing, client relationship management, analysis and prediction of user or consumer behaviors. Actual research projects address more complex problems dealing with spatial (geographical) data or multimedia data, often supporting decision making systems. Exploring corporate data, thus running dynamic queries online, analysts get a better understanding of situations ultimately leading to decision making or to the definition of corporate strategies.

The technique we describe in the present paper supports visual and interactive exploration of large volumes of data stored into a datacube. The exploration provide the user with the possibility of looking at the whole data cube, instead of just a subset of tuples resulting from a specific query. Locating higher level structural patterns, the analyst can then zoom in the cube and locate regions of the cube that be further studied and visualized. We build our solution from pixel-oriented principles astutely applied to datacubes resulting in an efficient visual mining technique.

2. Datacube basics

Cube operators, or Datacubes, were introduced [6] to efficiently support multiple aggregations which is widely used in OLAP databases [1, 3, 4, 13, 7]. They provide multiple points of view on metrics of interest, thus supporting common decision making or analysis tasks. More precisely, measured values are aggregated at various levels of granularity across dimensions which can be viewed as criterion guiding the analysis.

	Publications		
	Author	**Type**	**Title**
1	Auber	conf	infovis
2	Auber	conf	iccvg
3	Auber	conf	iv
4	Auber	book	lncs
5	Auber	conf	GD
6	Delest	conf	infovis
7	Delest	conf	iv
8	Delest	journal	lncs
9	Domenger	conf	iccvg
10	Domenger	conf	tcvg
11	Domenger	conf	infovis

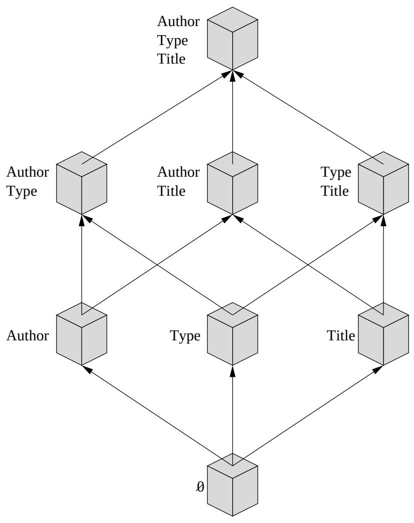
Table 1. Publications (papers) are stored according to three dimensions: $Author$, $Type$ (of publication) and (journal or conference) $Title$.

A Datacube is computed based on the content of an attribute table extracted from a database or datawarehouse. The set attributes or table columns is divided in two parts. The dimensions or categories, called Dim, gather criterion under which the analysis is usually conducted. Table 1, for example, gathers attributes concerning papers published by several authors. The table itself, called $Publications$ contains a set of dimensions (columns) $Dim = \{Author, Type, Title\}$ that can be be searched according to possible values for $Authors$, $Type$ and/or $Title$.

As mentioned above, a Datacube captures all possible aggregation of tuples sharing common attributes values. In other words, the original table can be somehow enriched using an aggregate function $COUNT$ counting entries on a subset $S \subset Dim$ of dimensions. Equivalently, this amounts to allow tuples to have any value (ALL) for dimensions not in S. An additional column then stores the result of this aggregation and contains the number of corresponding tuples. As an example, Table 2 shows the datacube inferred from the attribute table in Table 1 in tabular format.

	Corresponding Datacube of Publications			
	Author	**Type**	**Title**	**Count**
1	ALL	ALL	ALL	11
2	Auber	ALL	ALL	5
3	Delest	ALL	ALL	3
4	Domenger	ALL	ALL	3
5	ALL	conf	ALL	9
6	ALL	book	ALL	1
7	ALL	journal	ALL	1
8	ALL	ALL	infovis	3
9	ALL	ALL	iccvg	2
10	ALL	ALL	iv	2
11	ALL	ALL	lncs	2
12	ALL	ALL	GD	1
13	ALL	ALL	tcvg	1
14	Auber	conf	ALL	4
15	Auber	book	ALL	1
16	Delest	conf	ALL	2
…	…	…	…	…
34	ALL	conf	GD	1
35	ALL	conf	tcvg	1
36	Auber	conf	infovis	1
37	Auber	conf	iccvg	1
38	Auber	conf	iv	1
39	Auber	book	lncs	1
40	Auber	conf	GD	1
41	Delest	conf	infovis	1
42	Delest	conf	iv	1
43	Delest	journal	lncs	1
44	Domenger	conf	iccvg	1
45	Domenger	conf	tcvg	1
46	Domenger	conf	infovis	1

Table 2. The datacube inferred from the publication list in Table 1.



Lattice of cuboïds or powerset of dimensions $\{Author, Type, Title\}$ from the $Publications$ relation in Table 2.

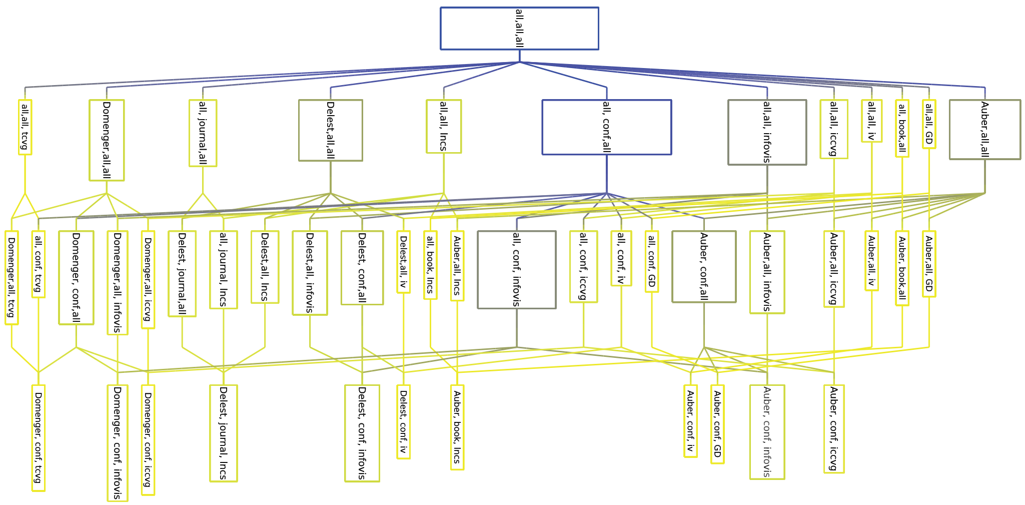


Figure 1. Semi-Lattice of all possible values for dimensions $\{Author, Type, Title\}$ (Table 2).

As one can observe, the effect of the aggregate function on combination of dimensions can be easily described as a powerset of dimensions. In our example, combinations have been grouped according to the number of aggregated dimensions. In the first line, all dimensions have been aggregated; as a result the $COUNT$ attribute for this line simply evaluates to the total number of publications. Lines 2–13 correspond to aggregation over each dimensions, resulting in a count of publications according to every possible values of all attributes; that is, the first few lines gives the number of publications of each author, followed by the number of publications by type (number of journal papers, conference papers, etc.). For sake of simplicity, the table does not explicitly list lines 17 to 33. Observe also that the last lines of the datacube simply duplicate the original table.

The reading of the datacube is easy since each line can be interpreted in a straightforward manner. The datacube presented in Table 2 is however extremely small. In reality, the tabular form of a datacube would expand over tens or hundreds of pages and is not at all appropriate for interactive exploration. Indeed, the size of the datacube can be computed as follows. Write $Dim = \{a_1, \ldots, a_k\}$ (where a_i stands for "attribute i") and denote by $A_1, \ldots, A_q$ the set of attribute values for dimensions $a_1, \ldots, a_k$.

Let $\mathcal{A} \subset Dim$ be a subset of all possible dimensions. Denote by $Q(\mathcal{A})$ the number of tuples that coincide on dimensions in $\mathcal{A}$ (that is, the set of tuples that coincide when their column attributes in $\mathcal{A}$ are replaced by ALL as in Table 2). The size of the datacube is then equal to $\sum_{\mathcal{A} \subset Dim} Q(\mathcal{A})$. Observe that this value is bounded above by $\Pi_{i=1}^{k}(1 + |A_i|)$ since $Q(\mathcal{A})$ is bounded above by $\Pi_{i \notin \mathcal{A}} |A_i|$. Thus, in any case, we must suspect the datacube size to grow exponentially as the number of dimensions increases.

2.1. Cuboïds lattice

Given a relation (table) R with dimensions $Dim = \{a_1, \ldots, a_k\}$ together with an aggregate function f, the cube operator is usually expressed as a query:

```
SELECT           $a_1, \ldots, a_k, f(M)$ FROM $R$
GROUP BY CUBE    $a_1, \ldots, a_k$
```

Such a query achieves all the "group-by" operations according to all combinations of dimensions belonging to the powerset over Dim. Each sub-query performing a single "group-by" operations associated with a subset $\mathcal{A} \subset Dim$ yields a *cuboïd* [7]. The image on the previous page illustrates the lattice of cuboïds associated with the powerset over $Dim = \{Author, Type, Title\}$. The powerset of dimensions can be represented by a lattice of cuboïds which naturally maps to a 3D cube since $|Dim| = 3$.

Various approaches addressing the computation of the datacube make use of sorting or hash-based techniques [5], and study optimization strategies allowing to reuse results previously obtained during execution [1, 16, 13, 12]. More precisely, they convert the dimension lattice in a processing tree providing the order according to which cuboïds are to be computed.

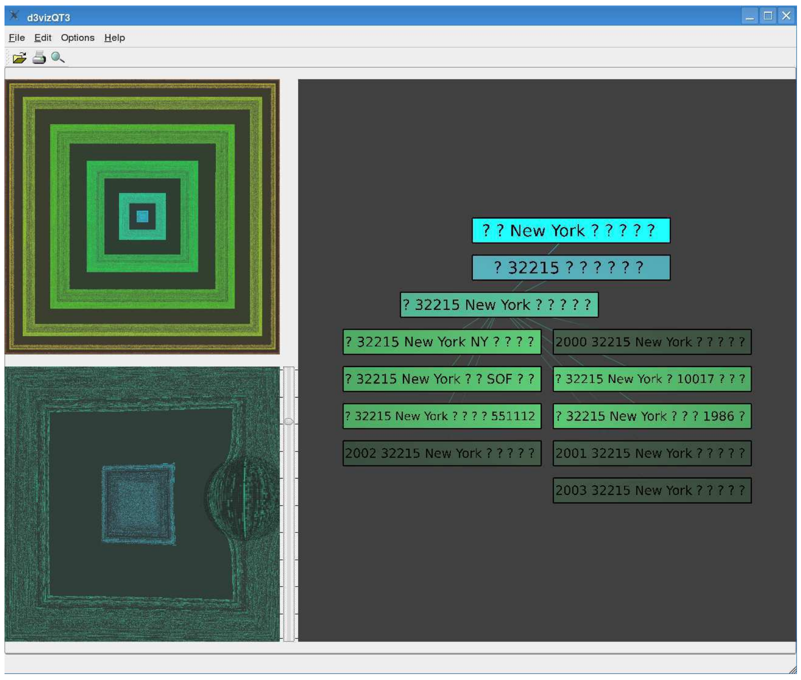

d3vizQT3
File
Edit
Options
Help
? ? New York ? ? ? ? ?
? 32215 ? ? ? ? ? ?
? 32215 New York ? ? ? ? ?
? 32215 New York NY ? ? ? ?
2000 32215 New York ? ? ? ? ?
? 32215 New York ? ? SOF ? ?
? 32215 New York ? 10017 ? ? ?
? 32215 New York ? ? ? ? 551112
? 32215 New York ? ? ? 1986 ?
2002 32215 New York ? ? ? ? ?
2001 32215 New York ? ? ? ? ?
2003 32215 New York ? ? ? ? ?

Figure 2. Infovis Contest 2005 semi-lattice: (top left) the whole semi-lattice drawn using a spiral layout, (bottom left) a zoom + a fisheye on semi-lattice,(right) a node link diagram of a part of the semi-lattice.

3. Visualization of multidimensional data

The lattice of cuboïds solely encapsulates the order relation between subsets of dimensions implicitly stored in the datacube. It does not however unfold relations between data elements or attribute values. Indeed, the cuboïd lattice does not reflect the size of tuple sets underlying each “group-by” operation, for instance. It is however possible to unfold these details using a different visual representation, as shown in Figure 1 (previous page).

This node-link representation is more compact than the tabular form (Table 2), since it makes use of intrinsic links between tuples. The cuboïd lattice structure provides a natural ranking: tuples sitting at rank i (from left to right) are those containing $|Dim| - i$ occurrences of ALL. Predecessors of a tuple are easily obtained by replacing attribute values by the ALL generic value in all possible ways. This semi-lattice can thus be drawn using drawing algorithms for directed acyclic graphs, for instance.

This semi-lattice representation provides a lot of visual information about the data and enables to find interesting phenomena in the data. For instance, node size in Figure 1 is mapped to the number of tuples underlying each query. This visual artefact makes it straightforward to see that all authors have published a majority of their papers in conferences as shows the bigger blue border box in the middle of the second row. Going down one level, we moreover see that they publish most of their papers in the same conference. Node link diagrams such as the one in Figure 1 are efficient for datacubes (or more generally for datasets) of about 1000 nodes. Hierarchical graph drawing algorithm can be used to optimize edge crossing and ensure better readability. Going for larger datasets however requires alternative visual representations.

Lots of work has already been done on database visualization. For instance, in Polaris [19], Stolte et al. propose an interactive tool that enables to interactively build a set of relational queries and visualize the results. Multimensionality is dealt with by letting the user select several 2D views built from all pairs of dimensions. Specific visualizations are furthermore computed depending on the type of attributes. For details on Stolte et al.’s projection technique for multidimensional data see [18]. Stolte et al. also make use of datacubes, building *lattices of datacubes* to offer multiscale navigation of the data at different levels of detail.

VisDB [9, 10] by Keim et al. have developed pixel-oriented techniques mapping data elements to single pixels, thus making optimal use of the available screen space. Our work builds on top of pixel-oriented principles, as we shall see in the forthcoming sections.

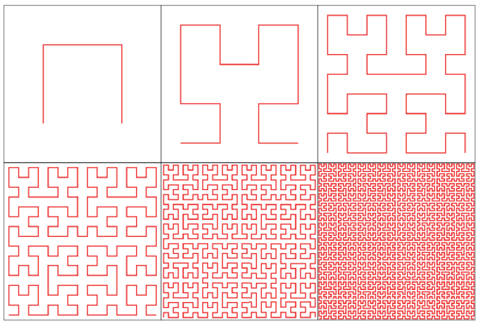
3.1. Pixel-oriented visualization for datacubes

From now on, we shall use the 2005 Information Visualization Contest dataset as a running example. This dataset is made of 458916 tuples over eight dimensions and two measures. The size of the semi-lattice built from this dataset contains 56 808 846 items. Figure 2 shows three different views of the InfoVis 2005 Contest semi-lattice. These snapshots illustrate how pixel-oriented views and node link diagrams can be combined in order to support interactive exploration of the semi-lattice. We use an overview window which can be zoomed in. Starting from visual patterns that can be located and navigated in the left windows, the user can query the underlying data and rely on a hierarchical layout of a small subset of the data.

The first ingredient (or sub-process, see the visualization pipeline, section 4 below) when building pixel-oriented visualization is to select a mapping from data elements to pixels on the screen. Basically, this is done using a bijective function so that each pixel maps back to a single data element. Other sub-processes enter what is often called the "visualization pipeline" as discussed in section 4. The pixel-oriented paradigm introduced by Keim (see [8]) can be seen as an optimization problem: pixels should be mapped so that neighbor pixels in the data are placed close to each other on the screen. The technique uses a linear order on data elements, often inferred from a (totally ordered) selected attribute, which can be seen as a map from the data space onto a line segment. The mapping onto a 2D portion of the plane then uses a "space-filling curve".

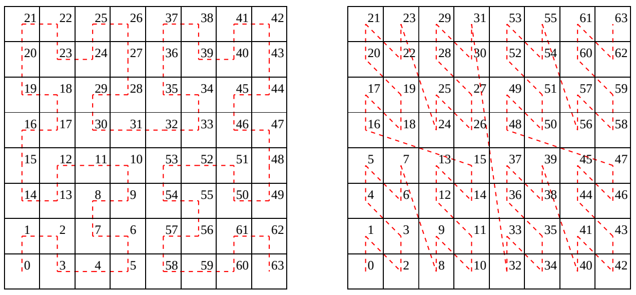
3.1.1 Fractal curves: Peano-Hilbert and Morton

Space-filling curves go back to Peano and Hilbert (see [14]). These curves are built using a basic pattern made of adjacent finite line segments embedded in the plane. Each of the segments is then replaced with the rescaled original pattern. Iterating the process, we get a sequence of 1D curves embedded in 2D space with growing complexity (see 3.1.1). The mathematical object we obtain when iterating this construction infinitely many times is a curve that fills a bounded rectangle: the curve passes through each 2D point only once.



The first six iterations of the Hilbert curve.

The construction iteratively assigns 2D coordinates to a finite sequence of integers. Indeed, the n^{th} curve of the Hilbert construction contains 4^n 2D-points (joints between finite line segments) and can thus be used to map an ordered list of 4^n data elements onto the plane. Depending on the number of available pixels and/or of the number of data elements, the appropriate iteration can be used to map the data elements on the screen. In [11] Lawder and King provide an efficient algorithm to compute this bijective 1D to 2D mapping both ways based on a state diagram representing each state of the construction of the Peano-Hilbert curve. Their algorithm allows to compute screen coordinates from integer index (or data index from screen coordinates) in time $\lceil log(N)/log(4) \rceil$ where N is the number of data items.



1D to 2D mapping for the (left) Peano-Hilbert curve, (right) Morton curve.

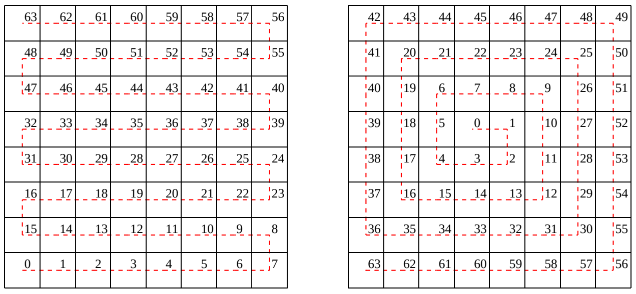
The Morton curve is another example of space-filling curve that proved to be useful in pixel-oriented visualization (see [8]). The Morton curve possesses a nice property linking geometry (coordinates) and arithmetic (index): screen coordinates $(x_1x_2 \ldots x_n, y_1y_2 \ldots y_n)$ map to index $(x_1y_1x_2y_2 \ldots x_ny_n)$ where each $x_i, y_j)$ are 0-1 bits. Hence, again the Morton 1D to 2D map (and vice-versa) can be computed in time $\lceil log(n)/log(4) \rceil$.

3.1.2 Scanline and spiral curves

Fractal curves have been shown to provide good heuristic to the optimization problem underlying pixel-oriented visualization [8]. Designing pixel-oriented visualization for the datacube however requires to take additional elements into account. Not only do we have to place nearby data elements close to each other on the screen, but we furthermore wish to reveal the ranking of elements in the semi-lattice. For this reason, we have explored other non-fractal 2D embeddings.

The next illustration shows a first example of a non-fractal space filling curve. Scanline (left) orders pixels top to bottom and left to right just as letters are implicitly ordered on the page of a book. Incidentally, Keim uses this ordering when building recursive pixel-oriented patterns [8].

One advantage of the scanline mapping function, in addition to the ease of computation, is its ability to deal with various aspect ratio, which reveals to be practical in numerous applications. Writing W for the window width towards which data elements are mapped, the coordinates for element i can be straightforwardly computed through the map $i \rightarrow (x_i = i \ mod \ W, y_i = i/W)$. Conversely, the index of pixel (x_i, y_i) is $i = y_i \cdot W + x_i$.



Scanline curve (left) and Spiral curve (right).

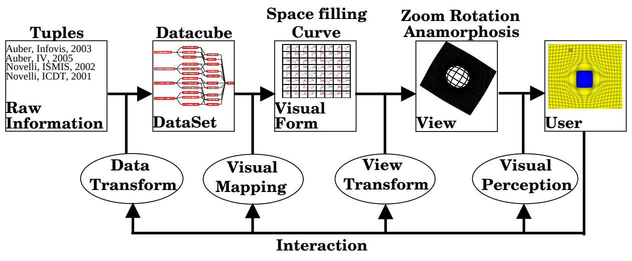
The spiral curve depicted on the right, also studied by Keim [8], however proved to be the best candidate for revealing the ranking of elements. Elements are first ordered according to their rank in the semi-lattice, and then ordered according to a selected attribute. In doing so, we are able to map each layer of the semi-lattice to a ring in the spiral pattern, coupling this mapping with visual cues such as color hue associated with rank (see Figure 2).

The spiral curve has one more property which reveals to be important when dealing with extremely large datasets. Writing d for the distance from a ring R to the center (half width of a centered square), the number of pixels on R is $8 \cdot d$. Thus the amount of pixels drawn with a spiral of width W is: $\sum_{i=0}^{w} 8 \cdot i = \frac{8 \cdot (w \cdot (w-1)}{2} = 4 \cdot (w^2 - w)$. As a consequence, the screen position of the i^{th} element can be computed in constant time (and conversely).

4. Efficient traversal of the visualization pipeline

The visualization process is often depicted as pipelined sub-processes taking care of the various operations that must be conducted from raw data ultimately to a graphical view on a screen. The diagram below is inspired from Card et al. [2], and provides a depiction of the pipeline adapted to our visualization.

As can be seen, the pipeline not only captures the order in which operations must be carried, but also indicates how user interaction can trigger lower order processes. Acting on the screen will for instance trigger computations on part of the data (second box from the right), itself affecting the layout (third box) and visual cues (fourth box). The visualization process, when described from the user standpoint (see Spence [17] for instance), often consists in a discovery loop going back and forth in the visualization pipeline. Let us comment on three transformations guiding the computational process underlying the pipeline, mapping each step to the datacube visualization:



Visualization pipeline (adapted from [2])

- The first step consists in transforming raw data into an efficient and easily exploitable format. In our case, this step amounts to the computation of the datacube.

- The second step computes visual attributes (such as color, shape, texture, position, . . . ; see [20]) for each data elements. In our case, following a pixel-oriented methodology, each data element is assigned 2D coordinates and color hue depending on a selected numerical attribute. In this case, computing the shape of a data element is irrelevant. Note however that we must perform other types of transformations when we build the node-link view on the right panel, showing part of the semi-lattice (see Figure 2).

- The third step consists in building the graphical view shown on the screen. Various types of operations can be applied to the graphical view itself, not requiring to go back to earlier sub-processes. Put simply, the view can be seen as a camera that can move or rotate around an object. Different types of lens, zoom, wide angle (or fisheye) allow the application of focus + context techniques.

Observe that each of these sub-processes has its own requirement and impact on the final cost for building a particular visualization, both in terms of memory requirements and time complexity.

Storing visual attributes such as coordinates for all of the N data elements, requires a minimum of $8 \cdot N$ bytes. Additional data must be stored in order to insure efficient data management when processing usual operations such as clipping, or when dealing with occlusion issues. Following [15], quad-trees (or kd-tree data structures) is a good choice, but nevertheless require to use at least 12 bytes per data elements to store ids and tree edges (pointers) for links between parent and child nodes.

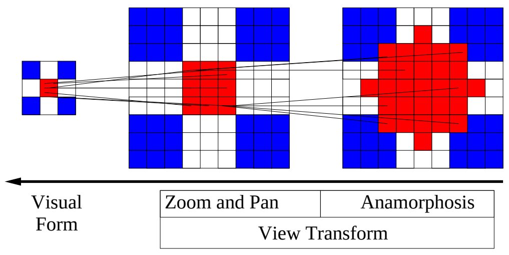
A naïve implementation of the visualization pipeline requires at least 20 bytes memory per item, which amounts to at least 1 billion bytes ($\sim$ 1 Gigabytes) for a 50 million elements dataset. Memory management and rendering time are thus central issues when displaying dataset of that size. Now observe that the number of pixels displayed on the screen is relatively small compare to the size of the underlying dataset we need to manage. As we shall see, the time and memory complexity actually depends on the number of *displayed* elements (ultimately equal to the number of available pixels) instead of the number of data elements. That is, the various processes involved in the pipeline will obey a time complexity proportional to screen size, although all dataset elements must remain available in main memory. The main interest is that we can store many more data elements than there are pixels and yet provide a real-time visual navigation of the whole dataset.

5. Visual mapping

Let us first consider the simplest situation, where the dataset size *equals* the number of available pixels. We can then index elements using one of the space-filling curve described above to map elements to integer coordinates, which can then be identified with screen coordinates. Assuming the user clicks on a single pixel, the integer coordinates of the pixel can then be used to directly access the corresponding data element. No intermediary computation or memory storage is needed to traverse the pipeline. This operation can be done in constant time if we moreover use the spiral or scanline curve.

Now, let us look at the case where the number of data elements exceeds the number of available pixels. Data elements must then be sampled in order to build an overview. A straightforward strategy is to divide up the dataset into subsets of identical size containing successive data elements and to pick one representative in each. This scenario is advantageous because it does not require additional computing time, but merely to compose the overall bijection with basic modular arithmetic. It is also realistic, since successive elements are similar. Indeed, computing average values over each of the subsets, for instance, would require additional computing time without having a decisive impact on the final image.

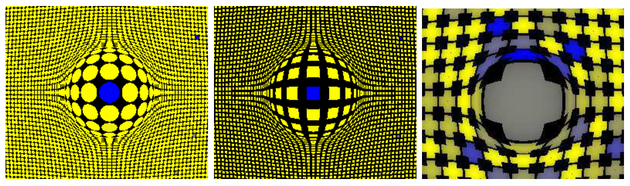
The case where the dataset contains less elements than there are available pixels is of interest since the additional screen space can be used to enrich the visualization, as we shall explain. Formally speaking, going from the screen back to the dataset is then a surjective mapping (since each pixel maps back to more than one data element).



Sending pixels surjectively back to data elements allows the insertion of glyphs in the final image.

5.1. Shapes and glyphs

Selecting individual elements becomes difficult if not impossible when data elements map to single pixels. Basic mechanisms can be used to ease such interaction. Starting from a one-to-one data to pixel mapping and zooming into the image brings us to the situation where pixels map surjectively onto data elements. Screen coordinates then map to fractional coordinates. Rounding up coordinates then selects a representative element. Observe that the rounding procedure actually identifies neighbor pixels (a rectangular region on the screen) that all map to the same data element. We can furthermore compose this operation with filters to show various shapes instead of plainly mapping the same value to all neighbor pixels.



Glyphs as side effects of surjective pixel mapping circle (left), square (middle), cross (right) together with fisheye distortion.

6. Discussion and future work

We have shown how pixel-oriented pixel visualization can be designed in order to visualize very large scale datasets built from datacubes. Fluid interaction is performed based on constant time traversal of the visualization pipeline, keeping computational complexity proportional to the number of pixels in the final image on the screen. Selection of single data elements is guaranteed through zoom and fisheye distorsion enabling the insertion of glyphs involving local surjective mapping.

Our method has been validated against a real-life dataset, namely the 2005 IEEE Information Visualization Contest. We were able to actually identify patterns that were found in data by participants. Our technique was also benchmarked using “randomly” generated datasets containing 96 936 757 elements. Building the internal data structure took approximately 20 second on an Intel centrino Duo Core T2600 with 2 Gigabytes of memory (a multi-threaded implementation of our technique used C++, to take full advantage of the Duo Core technology).

This paper concentrated more on computational issues showing how constant time pipeline traversal, and consequently fluid interaction, can be achieved. A longer version of the paper will include a detailed discussion of our case study, also looking at usability issues. More details concerning the computation of the datacube, adequately adapting its computation to the visualization, also need to be revealed. Finally, several properties of space filling curves can be exploited to optimize our data structure and astutely use hard disk swap.

Acknowledgements. This research was funded through the French grant “ACI Jeunes chercheurs”: Cube de Données – Construction et Navigation Interactive.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *VLDB'96*, pages 506–521, 1996.

[2] S. Card, J. Mackinlay, and B. Schneiderman. *Readings in Information Visualization: Using Vision to Think*. 1999.

[3] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[4] P. Deshpande, J. Naughton, K. Ramasamy, A. Shukla, K. Tufte, and Y. Zhao. Cubing Algorithms, Storage Estimation, and Storage and Processing Alternatives for OLAP. *Bulletin of IEEE*, pages 3–11, 1997.

[5] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 1999.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *ICDE'96, New Orleans, Louisiana*, pages 152–159, 1996.

[7] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[8] D. A. Keim. Designing pixel-oriented visualization techniques: Theory and applications. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):59–78, 2000.

[9] D. A. Keim and H. Kriegel. VisDB: Database exploration using multidimensional visualization. *Computer Graphics and Applications*, (5):40–49, 1994.

[10] D. A. Keim and H.-P. Kriegel. Visdb: a system for visualizing large databases. *SIGMOD Rec.*, 24(2):482, 1995.

[11] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In B. Lings and K. G. Jeffery, editors, *BNCOD*, volume 1832 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2000.

[12] K. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. In *VLDB'97, Athens, Greece*, pages 116–125, 1997.

[13] K. Ross, D. Srivastava, and D. Chatziantoniou. Complex Aggregation at Mutiple Granularities. In *EDBT'98*, LNCS vol. 1377, pages 263–277. Springer Verlag, 1998.

[14] H. Sagan. *Space-Filling Curves*. Springer, 1994.

[15] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[16] S. Sarawagi, R. Agrawal, and A. Gupta. On Computing the Data Cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[17] R. Spence. *Information Visualization*. ACM Press, 2000.

[18] C. Stolte, D. Tang, and P. Hanrahan. Multiscale visualization using data cubes "infovis 2002 best paper". In *INFOVIS*, pages 7–14. IEEE Computer Society, 2002.

[19] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Vis. Comput. Graph.*, 8(1):52–65, 2002.

[20] C. Ware. *Information Visualization: Perception for design*. Interactive Technologies. Moragn Kaufmann, 2000.