

# Splatting multiresolution volume data using the Feature Graph

J. Campos      A. Puig      D. Tost

January 11, 2008

## Abstract

We propose to represent classified datasets as a feature graph storing different graphical models and attributes for each feature. This graph allows us to render each feature according to its own characteristics. In addition, we show that various features of the graph storing volume information at different resolution levels can be rendered together using a view-aligned splatting method. Moreover, we propose a 2D kernel function for splats that is easy to tune and generates smaller footprints that reduce the render time. Our algorithm provides images with less blur. It enhances the boundary of the features while avoiding the subdivision of homogeneous regions of the volume.

## 1 Introduction

In scientific visualization, there is a growing interest for the visual and semantic quality of the rendered images and for the interfaces through which users specify the set of visualization parameters [vW05]. Classification has been recognized [Pa01] as one of the major issues in visualization. It is indeed an important step of data exploration, because it is used to determine which materials are at every sampled position. However, the visualization problems do not finish once a dataset has been classified. On the contrary, a true talent of graphical design and skills in illustration are necessary to obtain meaningful images of a pre-classified dataset, images that emphasize relevant focused structures while keeping some details of the context surrounding elements. This is precisely one of the current problems of visualization: the design of the images is delegated to users of visualization

applications, scientists and physicians who, generally, do not have the required graphical skills for it, and, moreover are overwhelmed by the complexity of the interfaces. This is why many efforts are now being done to provide visualization software with mechanisms that ease or automatize the computation of rendering parameters such as cameras [BE06], transparency [GDF03] and sparsness [VKG05] of different features. On a large extent, these techniques rely on a previous step of classification and segmentation of the different features of the dataset.

Depending on the methods chosen to render classified datasets, it may be necessary to design data structures that provide a direct access to the graphical model of each feature. Examples of such structures are the lists of voxels used in two-level rendering (*RenderLists* [HMBG01] and *objects sets* [HBH03]) and the run-length encoding the feature identifiers of the voxels for multimodal rendering [FPT06]. More complex topological representations of datasets based on their skeleton have also been proposed for animation [CSW<sup>+</sup>03]. However, all these structures represent the same volumetric information for all the features. Other structures such as octrees [WG94] support different levels of resolution, but, since they are based on a spatial subdivision of space, they are not convenient to separate segmented structures.

In this paper, we propose to represent datasets as a graph of features providing direct access to each feature. The features can be represented with a different model and at a different resolution level. Thus, they can be rendered separately according to their own characteristics. When different features are rendered together, two main problems must be solved: occlusion and transition between features. In this paper, we address the rendering of features represented by voxels at different resolutions. We use a view-aligned splatting algorithm capable of handling voxels of different sizes. In addition, we propose to compute the voxels splat using as kernel function a beta function that is easy to tune and generates smaller footprints that reduce the render time. Our algorithm provides images with less blur. It enhances the boundary of the features while avoiding the subdivision of homogeneous regions of the volume.

## 2 Related work

### Data structures for the representation of classified datasets

Most spatial decomposition data structures used in volume rendering, such as pyramids [LH91], octrees [DKC00] and BSP trees [LMK03], have been designed

to provide space leaping mechanisms and to compress volume data. However, less bibliography address the problem of representing segmented and pre-classified or *tagged* data. Tiede et al. [TSH98] propose to use together with the original voxel model an additional volume containing an identifier (ID) of the region to which each voxel belongs. The drawback of this approach is that it does not provide a direct access to the features. Ferré et al. [FPT06] partially solve this problem by using a run-length codification of the voxels identifiers which can be used to skip non selected features during rendering. A direct representation of the features are the *RenderLists* [HBH03]) used for two-level rendering. This structure stores lists of voxels of all the features for each slice of the volume. It supports only disjoint features of the same resolution. An alternative idea is to partition the voxel model into sub-models, hence preserving the spatial ordering of the voxels inside the sub-models. Specifically, Li et al [LMK03] use disjoint boxes to ease depth sorting. Puig et al. [PTN00] support overlapping voxel sub-models, because these sub-models are organized in a graph that drives the data traversal. This graph represents the limbs and joints of the cerebral vascular structure that are supposed to be disjoint so that each voxel sub-model masks the overlapping ones. Finally, for volume animation and deformation, Singh et al. [SSC03] and later Walton et al. [WJ06] represent topological skeletons of the human body that associate voxel sub-models to each segment of the skeleton. All these approaches represent disjoint features, with the same resolution. Our goal is provide a structure capable of handling non-disjoint features, each of them with its own convenient resolution.

## Splatting tagged volumes

One of major advantages of splatting is that reduces significantly aliasing because of the smooth kernels decay [Wes89]. Unfortunately, this smooth decay and the kernels' overlapping at the edges produce blurred boundaries. Several methods have been proposed to overcome this drawback. Mueller et al. [MMC99] use of a post-shade scheme instead of the traditional pre-shaded approach. However, generally, post-shading has a higher cost than pre-shading because it projects a larger number of voxels and performs shader more times per voxel. Huang et al. [HCS98] employ different kernels depending on the proximity of voxels to edges and the strength of those edges. This method is effective for iso-surface edges, but not for micro-edges within the iso-range. Meredith and Ma [MM01] use a multiresolution octree. For each node, they choose the coarser resolution and check if its corresponding splat size on screen is below a given threshold. If it is greater than the threshold, they descend to a finer resolution. This way,

they avoid both aliasing and blurring by choosing the largest splats whenever it is possible without blur. Birkfellner et al. [Ba05] use smaller kernels to minimize the blurring artifacts, and displace stochastically the voxels positions to avoid that the splats follow aligned patterns.

In addition, in the visualization of tagged volumes the determination of ID boundaries at subvoxels resolution causes aliasing effects. Both problems have been addressed in the bibliography in the context of ray-casting [TSH98] [KCOY03] and texture mapping [VHN<sup>+</sup>05]. Mueller et al. [MLK03] do not work explicitly with tagged volumes, but they identify voxels of the different regions of interest (ROI) of a dataset and migrate their density range to private intervals. They apply intensity-flipping at the transition between features achieving a smooth decay at the boundaries. However, this approach is hard to extend to tagged data, segmented using more criteria than the property range only.

Finally, for perspective projections, Mueller et al. [MMI<sup>+</sup>98] propose to use different kernel sizes according to the distance to reduce perspective aliasing. Zwicker et al. [ZPvBG01] apply *elliptical weighted average* non uniform kernels to provide different footprints and low-pass levels depending on the distance from the observer.

In our approach, features are represented at a convenient resolution. Specifically, we represent *surface features*, i.e features representing regions boundaries at a higher resolution than internal regions. Thus the boundary splats are naturally smaller than the others, which reduces blurring. Moreover, our model handles various levels of resolution for each feature. Therefore, if a feature is far from the viewer or it is out of the user focus, a coarser resolution is used and it is rendered blurry. On the contrary, higher resolution are used for focused features. This reduces the aliasing of perspective projection and provides focus+context visualizations.

### 3 The Multi-resolution Feature Graph

#### Definition

Our model represents classified voxel models. The classification can be done according to  $nc$  different non-exclusive criteria that can be expressed as boolean conditions. Therefore, after classification, for each voxel  $v$ , we have a set of  $nc$  boolean values (*labels*)  $r_i(v), i = [1..nc]$  that indicate if the voxel fulfils or not each boolean expression. As an example, if the classification criteria are: bone,

fat, arm, leg and toe, a voxel of the toe bone will have the following labels (1, 0, 0, 1, 1), as it also belongs to the leg.

The voxel model can be clustered into different sets according to any combination of these labels. Although the theoretical number of clusters is  $2^{nc}$ , in fact, many of them are empty because all the combinations are not possible. In our example, possible clusters are: “bone”, “bones and toe”, “fat and arm”, but the cluster “bone and fat” is empty. We define the concept of *feature* as a non-empty cluster. More precisely, a *feature* ( $f$ ) is a set of values for a subset of the boolean expressions used in the classification such that at least one voxel of the model fulfills these expressions. We can associate to any feature the set of voxels of the volume that have the same label values. Then, sets of voxels of a feature can be totally or partially included into sets of voxels of other features. In our example, the set of voxels of the feature “bone and toe” is totally included in the set of voxels of features “bone” “toe”.

We represent the features and the inclusion relationship as a graph. Specifically, the *Feature Graph* is a directed acyclic graph such that each node represents a feature and each arc represents the relationship of inclusion of the corresponding sets of voxels. The direction of the arcs goes from the larger to the smaller included feature. The entry point to the graph is a feature ( $f_{all}$ ) that includes all the features that do not have an ancestor. The leaf nodes of the graph correspond to features that do not include any other feature. Each feature of the graph has its own optical properties used for rendering. Finally, in order to provide a direct index to the feature nodes of the graph model, we use a *Feature Hash Table*.

## Sets of voxels

We define the set of voxels of a feature as  $SOV(f_i)$ . Since the  $SOV$  of the leave nodes are disjoint and are totally included in the  $SOV$  of their ancestors, in order to avoid redundancies, we only store  $SOV$ s in the leaf nodes of the graph. Then, the  $SOV$  of an intermediate feature can be obtained by recursively traversing its descendant nodes down to the leaves. Voxels of the  $SOV$ s can be represented either as submodels of voxels or voxels lists. In the former case, although the  $SOV$ s are disjoint, their bounding box can overlap. We assign a zero value to the voxels of a submodel that do not belong to the feature, so that a voxel has a non-zero value in only one unique leave  $SOV$ . This way, spatial ordering inside  $SOV$ s is preserved. This representation is convenient if the voxels distribution is compact. In the second case, the access to the voxels of a feature is direct, it is not necessary to traverse its submodel and check for the property value. It is convenient when

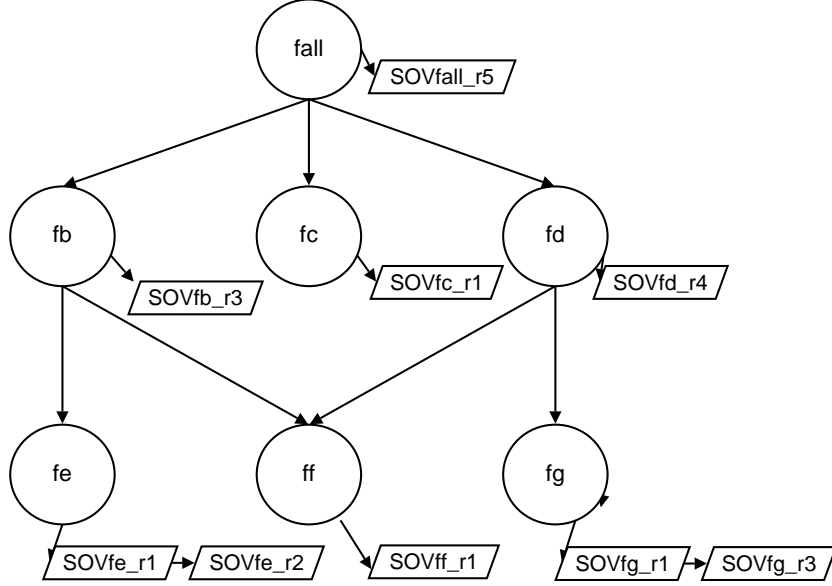


Figure 1: The *Feature Graph* that provides direct access to the data models associated to each feature, which can be stored at different resolutions.

the features are sparse and spread in the volume. As a drawback, it is necessary to store the voxels coordinates and there is no spatial ordering inside the *SOV*. Accordingly, we choose the type of *SOV* representation according to the spatial distribution of the feature and to the rendering algorithm.

The homogeneous regions of a *SOV* can also be compacted using different voxel sizes. When *SOVs* are implemented as subvoxel models, this is done using an octree of each submodel. Alternatively, the list of voxels stores only the black nodes of the octree with their associated size. Furthermore, since the *SOVs* can be stored in different files, efficient out-of-core traversal strategies can be applied.

## Multiresolution

Because of its hierarchical nature, and because it clusterizes the voxels, the *Feature Graph* can handle multiresolution representations of the voxel model. To do so, we only need to compute the sets of labels at coarser levels of representations of the voxel model. It should be noted that some clusters that were non-empty at the higher resolution can become empty at a lower resolution. Therefore, not all the features of the higher resolution level exist at lower levels. Consequently, some features can be leaves at a resolution and intermediate nodes at another res-

olution. Accordingly to our policy of avoiding redundancies, we store the *SOVs* only at the leaves of each resolution level. At rendering time, when a resolution level of a feature is required, the feature node that have the closest level of resolution in the selected hierarchy feature is visualized.

Finally, each graph feature could be represented by different models, such as a surface model extracted from a *SOV* and stored as a polygon list. A generic multiresolution *Feature Graph* is shown in Figure 1.

## 4 Graph Construction

The construction of the Features Graph takes as input the set of voxels, labeled according to the  $n_c$  criteria. It is composed of 3 steps: (a) the clustering of the voxels into sets of voxels with identical labels, (b) the derivation of the features hierarchy -creation of the nodes and arcs of the *Feature Graph*-, and (c) a recursive compression through the graph hierarchy to obtain multiresolution sets. Figure 2 shows the schema of this construction process.

The clusters computed in the first step are actually the *SOVs* of the leave features of the graph at the highest resolution. To compute them, we traverse the labeled voxel model using the set of labels of each voxel as a hash code of the *Feature Hash Table*. Each voxel is added to the *SOV* indexed by its hash code. At this point of the process, the *SOVs* are represented as lists of voxels. Once they are filled, we are able to compute their bounding box and to create a voxel submodel for each of them. These voxel submodels are compressed as octrees. Then, either the octrees are used directly to represent the *SOVs* or they are traversed to create a new list-based representation of the *SOVs* with voxels of various sizes.

In the second step of the process we construct the graph. First, we create the leave nodes and we associate to them the computed *SOVs*. Next, we derive the graph hierarchy through an iterative process based on boolean operations between the feature labels. This process recursively creates common ancestor nodes for all the nodes that share a bit of their hash code. When the hierarchy is built, the *Feature Hash Table* is updated to reference to new created nodes. At the end of the process, if the graph is composed of various disconnected hierarchies, we create the feature ( $f_{all}$ ) as the common root.

Finally, we compute the sets of voxels at coarser levels of representations of the original voxel model and repeat the clustering process for larger voxels. Then, the *SOVs* computed with these coarser clusters of voxels are distributed through the graph hierarchy and attached to the corresponding nodes. Thus, each node can

contain different resolution level *SOVs*.

## 5 View-aligned splatting of the Feature Graph

Rendering the Feature Graph depends on how the *SOVs* are represented, as voxel sub-models or lists. In this paper, we focus on the second representation. Since lists do not preserve spatial ordering, we have chosen the view aligned splatting approach that transforms and inserts orderly the voxels in view-aligned buckets.

### Graph traversal

The input of the rendering is a set of feature keys that have to be rendered called *user query*. In addition, user can specify the resolution at which the selected features need to be rendered, or this resolution can be computed automatically depending on the ratio pixel/voxel of the projection and the feature’s distance to the camera view reference point.

The feature keys of the user query are used to access the *Feature Hash Table* in order to fetch the selected nodes of the graph. The *SOVs* to be rendered are obtained by recursively traversing the selected features and their descendants. At each feature node of the graph, the selected resolution level is checked against the node resolution level. If the resolution level of the node is the same or higher than the desired one, the voxels of the *SOV* associated to the node are view transformed and inserted into the sheet buffers. In the opposite case, the node descendants are recursively visited and the same procedure is applied again. The optical properties applied for a *SOV* are inherited from the selected ancestor feature node. Besides, as all of voxels of a *SOV* share optical properties, they are set once at the beginning of each *SOV* traversal. Then, we iterate on the voxels of the *SOV* and we fill the sheet-buckets corresponding to the different voxel locations. Each voxel contribution to the volume rendering integral is defined by a set of kernel sections that fall within a set of cutting planes (sheet-buckets). A different set of kernel sections is computed for each footprint size defined by each resolution value. Pre-integrated kernel sections are used for fast rasterization and each sheet-bucket position stores the corresponding kernel index. At the end of the selected features traversal, the composition of all sheets or buckets in FTB order is performed.



## Splatting kernels

The kernel function used for splatting is usually based on the Normal distribution. However, we have found that this function is highly sensitive to changes in the visual parameters. The ratio pixel-voxel in zooming views, the shading function used, the optical properties of the feature and the opacity required define the final size of the footprint to keep continuity between neighbor voxels. For instance, holes may appear when voxel’s opacity decreases, requiring changes in the footprint size. These effects are specially noticeable at the boundary between features in tagged volume visualizations. We propose to use a Beta function as the *generic footprint function*, which improves the rendering performance using smaller footprints and that is easier to tune under visual changes.

The Beta-function lets controlling separately the openness, height and width of the curve. The general Beta-function is  $Beta(x, a, b) = x^a(1 - x)^b$ , where the parameters  $a$  and  $b$  control the shape of the curve in range  $[0, 1]$ . Hence, scaling the range and giving identical values to  $a$  and  $b$ , we can obtain the set of symmetric curves showed in Figure 3. This figure shows that it is possible to force the curve’s width (i.e.  $[-5..5]$ ) while controlling the curve’s shape: lower  $a, b$  values produce curves more opened.

$$Beta(x, a, b) = k_{scale}x^a(1 - x)^b \quad (1)$$

Figure 4 shows that we can use a smaller footprint based on beta distribution to obtain a splat similar to a normal distribution based footprint.

Image-aligned splatting slices the interpolation kernels by a series of cutting planes aligned parallel to image plane. The kernel sections share the same weight distribution but they have different radius depending on the cutting plane. We use a kernel base distribution and a scaling factor as [MC98] to obtain them. Figure 5 shows a single kernel composed of seven stacked section footprints.

In addition, an alpha-blend operator is used to compose the kernel sections that fall within the same sheet-bucket, instead the the *add* operator proposed in [MC98]. This alpha composition provides a smoother color transitions between overlapped kernel sections. Figure 5 illustrates the smooth interfaces between different optical properties and resolutions.

## 6 Simulations

The proposed *Feature Graph* model has been tested with two real datasets. All the tests were run on an AMD 64 3200+ with 1MB of RAM and an NVidia GeForce 6600 256MB. The datasets are tagged volumes with different material properties for each label. They are tested on our software platform *Hipo* [CPT06].

The resulting images of these simulations are placed in the Table 1. The first column shows the pictures of rendering the Feature Graph with all the sets of voxels at a resolution of  $256^3$  using an emission+absorption shading model (*emabs*). In contrast, the second column shows renderings of the Feature Graph at different resolutions and different shading models according to a Focus+Context schema. The focus is rendered at the original  $256^3$  resolution while the context is sub-sampled at coarser resolutions (up to  $128^3$ ). Even more, the surface of what we focus on, is rendered using a Lambert shading model while all the rest is rendered following the *emabs* model. Besides, the second row of each dataset contains a zoomed view of it.

The first model contains the classical engine block dataset, with a geometry of  $256 \times 256 \times 152$ . The first column shows that *beta footprints* and  *$\alpha$ Composition* provide smooth images with good visual quality even at zoomed views. The ratio pixels/voxel is 1.42 in the upper row, and 4.93 in the second row. In this case, the Feature Graph has been used to access directly to the voxels of the two selected features and render each one with different transfer functions. The last column shows how the Feature Graph can be used to obtain Focus+Context images. The focus -in blue color- is rendered using a Lambert shading model in its surface to enhance its edges, while the context -in red color- is rendered at a low resolution. The second dataset is the computer tomography of a frog, with a geometry of  $256 \times 236 \times 72$ . In this case the ratio pixels/voxels of the first column is 1.74 in its upper row and 3.86 in the last row. The last column also has Focus+Context images where the focus is the nervous and venous systems and the rest of the frog body is the context environment.

Finally, Figure 6 shows a comparison of the engine block dataset rendered using the Beta footprints or the usual Normal distribution based footprints. The visual quality is equivalent, although beta footprints are smaller. As a consequence, in the example, the execution is twice faster when using beta footprints.

## 7 Conclusions and future work

We have proposed to represent classified datasets as a feature graph. Our data structure represents each feature at its own convenient level of resolution. Moreover, it can handle multiresolution representation. We have shown how to render the graph using a view-aligned splatting method. To do so, we have proposed a 2D kernel function for splats that is easy to tune and generates smaller footprints that reduce the render time.

Our graph can be extended in many ways. First, we want to explore how to render it with other algorithms than view-aligned splatting. Storing the *SOVs* of the features as 3D textures seems a promising way to render the model using hardware-driven texture mapping or ray-casting. In addition, we want to add other geometric models to the features, as for instance a polygonal model of the surfaces extracted in a pre-process. This way, we would be able to render some features as surfaces and other as volumes. Again, depth sorting problems and transition artifacts between the different geometrical model will need to be solved. Finally, we would like to assign levels of importance to the features, that combined with their degree of focus would provide better automatic means of selecting for each feature its convenient graphical model, resolution and optical properties.

## References

- [Ba05] W. Birkfellner and al. Wobbled splatting: a fast perspective volume rendering method for simulation of x-ray images from ct. *Physics in Medicine and Biology*, 50:73–84, 2005.
- [BE06] S. Bruckner and M. E.Gröller. Exploded views for volume data. *IEEE Trans. on Visualization and Computer Graphics*, 12(5):1077–1084, 2006.
- [CPT06] J. Campos, A. Puig, and D. Tost. Efficient focus+context visual exploration of volume datasets. In *EG/ACM SIGGRAPH SIAGC’06*, pages 79–88, 2006.
- [CSW<sup>+</sup>03] M. Chen, D. Silver, A. S. Winter, V. Singh, and N. Cornea. Spatial transfer functions: a unified approach to specifying deformation in volume modeling and animation. In *Volume Graphics’03*, pages 35–44, 2003.

- [DKC00] F. Dong, M.A. Krokos, and G. Clapworthy. Fast volume rendering and data classification using multiresolution min-max octrees. *Comput. Graph. Forum*, 19(3), 2000.
- [FPT06] M. Ferré, A. Puig, and D. Tost. Decision trees for accelerating uni-modal, hybrid and multimodal rendering models. *The Visual Computer*, 3:158–167, 2006.
- [GDF03] C. Gutwin, J. Dyck, and C. Fedak. The effects of dynamic transparency on targeting performance isosurfaces. In *Graphics Interface'03*, pages 105–112, 2003.
- [HBH03] M. Hadwiger, C. Berger, and H. Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics Hardware. In *IEEE Visualization '03*, pages 40–45, 2003.
- [HCS98] J. Huang, R. Crawfis, and D. Stredney. Edge preservation in volume rendering using splatting. In *IEEE Visualization '98*, pages 63–69, 1998.
- [HMBG01] H. Hauser, L. Mroz, G. Bisch, and M.E. Gröller. Two-level volume rendering. *IEEE Trans. on Visualization and Computer Graphics*, 7(3):242–252, 2001.
- [KCOY03] A. Kadosh, D. Cohen-Or, and R. Yagel. Volume graphics. *IEEE Trans. on Visualization and Computer Graphics*, 9(4):580–586, 2003.
- [LH91] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *ACM Computer Graphics*, 25(4):285–318, July 1991.
- [LMK03] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture based volume rendering. In *IEEE Visualization 2003*, pages 317–324, 2003.
- [MC98] H. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. *IEEE Visualization '98*, pages 239–246, 1998.
- [MLK03] K. Mueller, S. Lakare, and A. Kaufman. Volume exploration made easy using feature maps. In *Workshop on Scientific Visualization*, 2003.

- [MM01] Jeremy Meredith and Kwan-Liu Ma. Multiresolution view-dependent splat based volume rendering of large irregular data. *2001 Symp. on Parallel and Large-Data Visualization and Graphics*, pages 93–99, 155, October 2001.
- [MMC99] K. Mueller, T. Möller, and R. Crawfis. Splatting without the blur. In *IEEE Visualization’99*, pages 363–371, 1999.
- [MMI<sup>+</sup>98] K. Mueller, T. Möller, J. E. Swan II, R. Crawfis, N. Shareef, and R. Yagel. Splatting errors and antialiasing. *IEEE Trans. on Visualization and Computer Graphics*, 4(2):178–191, 1998.
- [Pa01] H. Pfister and al. The transfer function bake-off. *IEEE Computer Graphics & Applications*, 21(3):16–22, 2001.
- [PTN00] A. Puig, D. Tost, and M. I. Navazo. A hybrid model for vascular tree structures. In *Data Visualization*, pages 125–135, 2000.
- [SSC03] V. Singh, D. Silver, and N. Cornea. Real-time volume manipulation. In *Volume Graphics*, pages 45–52, 2003.
- [TSH98] U. Tiede, T. Schiemann, and K. H. Hohne. High quality rendering of attributed volume data. *Proc. IEEE Visualization*, pages 255–262, 1998.
- [VHN<sup>+</sup>05] F. Vega, P. Hastreiter, R. Naraghi, R. Fahlbusch, and G. Greiner. Smooth volume rendering of labeled medical data on consumer graphics hardware. In *Proc. of SPIE Medical Imaging 2005*, pages 13–21, 2005.
- [VKG05] I. Viola, A. Kanitsar, and M. E. Gröller. Importance-driven feature enhancement in volume visualization. *IEEE Trans. on Visualization and Computer Graphics*, 11(4):408–418, 2005.
- [vW05] J. J. van Wijk. The value of visualization. In *IEEE Visualization’05*, pages 76–86. IEEE Press, 2005.
- [Wes89] L. Westover. Interactive volume rendering. In *Volume Visualization Workshop*, pages 9–16, 1989.

- [WG94] J. Wilhems and A. Van Gelder. Multidimensional trees for controlled volume rendering and compression. *ACM Symp. on Volume Visualization*, 11:27–34, October 1994.
- [WJ06] S. J. Walton and M. W. Jones. Volume wires: a framework for empirical non-linear deformation of volumetric datasets. *The Journal of WSCG*, 14:81–88, 2006.
- [ZPvBG01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA volume splatting. In *IEEE Visualization'01*, pages 29–36, 2001.

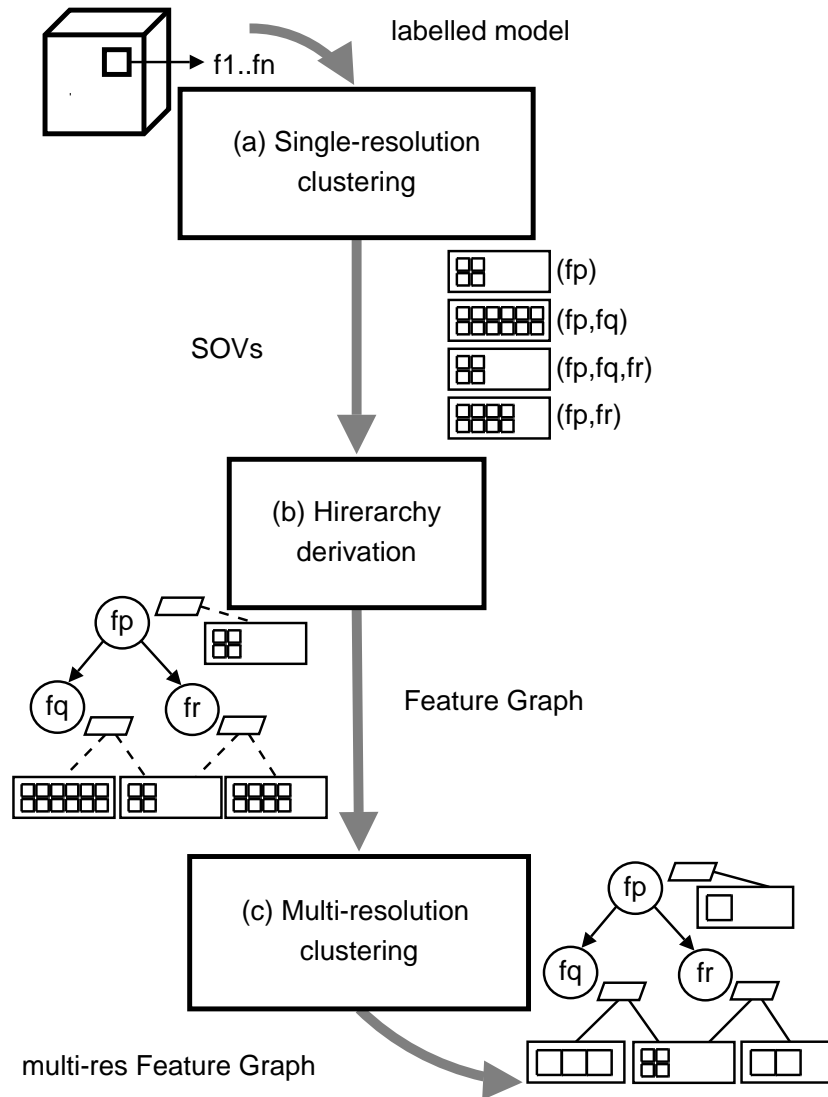


Figure 2: ATENCI!!!!!! cal refer grfic per a que no hi hagi SOVs comparits  
 !!! !!! !!! !!! !!! !!! !!! !!! !!! Feature Graph Construction: (a) single  
 resolution clustering (b) features hierarchy derivation, and (c) multiresolution clustering.

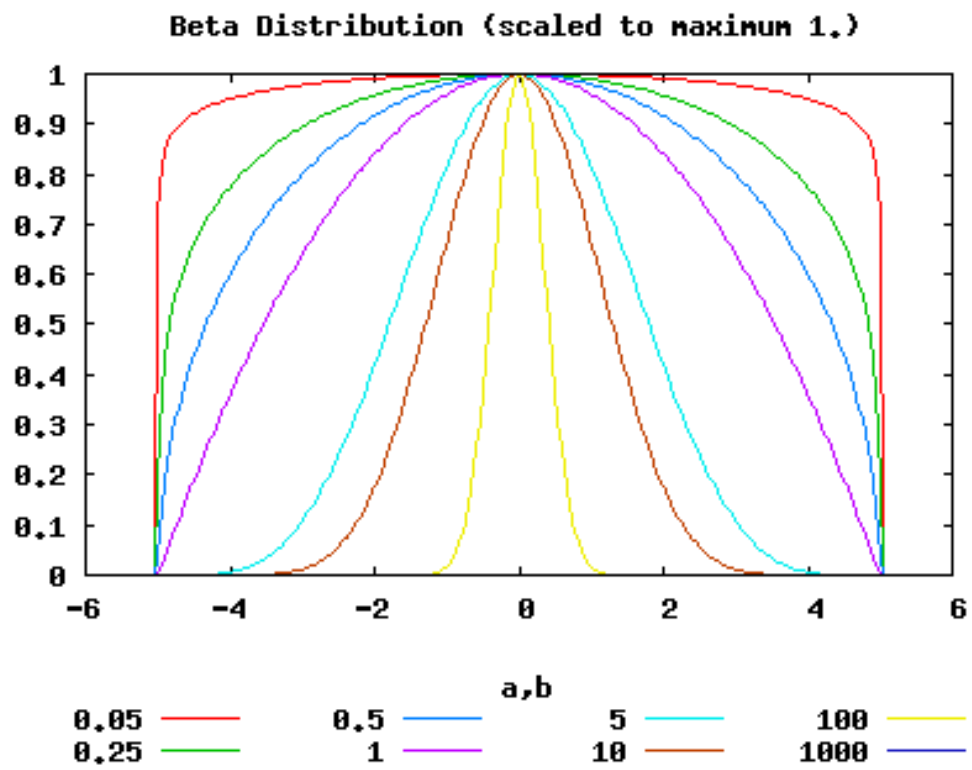


Figure 3: Beta D. with different exponent values.



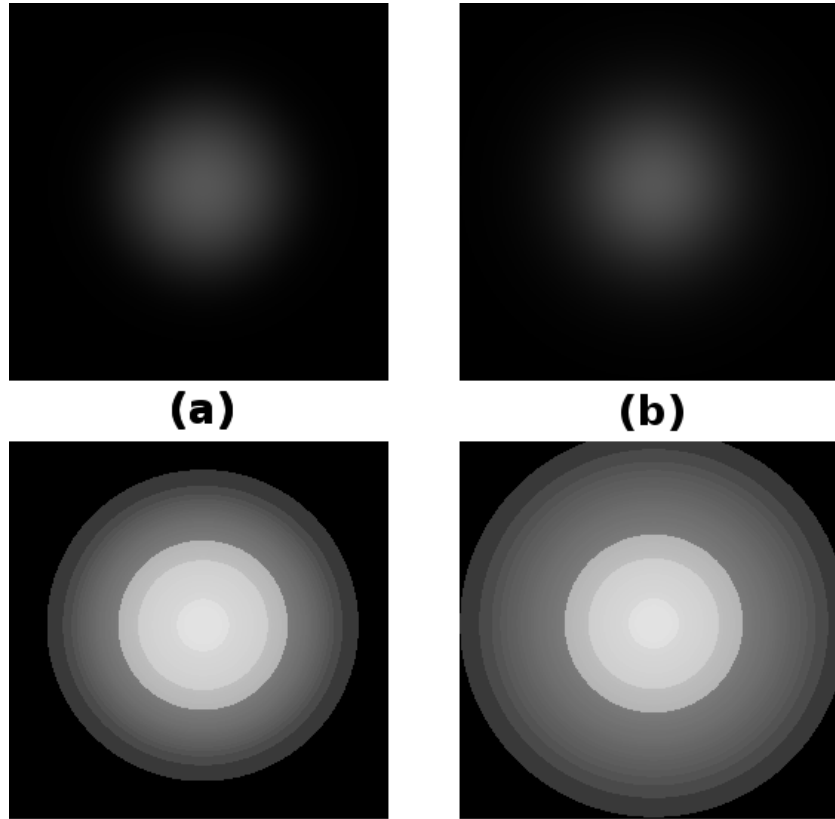


Figure 4: Splots: [top row] The 256 pixels *beta-footprint* (a) requires a 26.8% smaller footprint than the 350 pixels *gauss-footprint* (b); [bottom row] The same images but filtered to highlight the values even when they become close to zero. They reveal that the *gauss-footprint* has more extension to fall to almost zero than the *beta-footprint*.

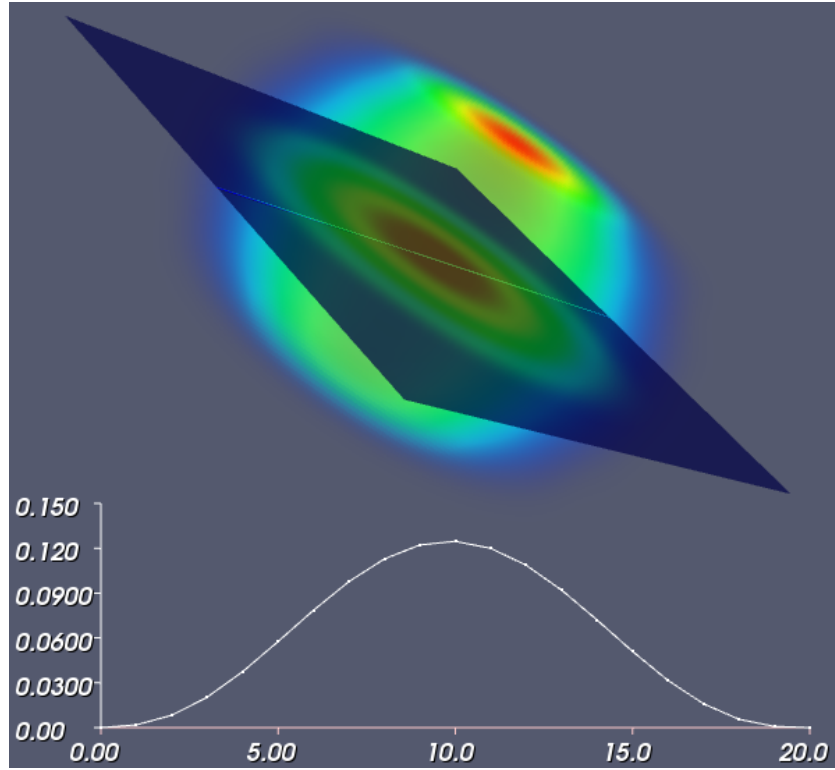


Figure 5: Stacked set of different *beta-footprints* corresponding to a single kernel and a graph displaying the footprint's weights along the central footprint.

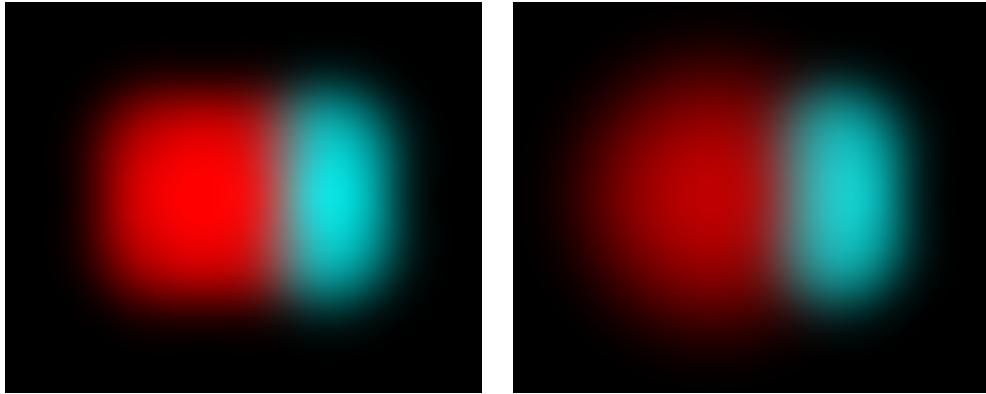


Figure 6: Interface of two resolutions: [left] six voxels at resolution 1 (the first four in red color and the last two in cyan color) [right] one voxel at resolution 2 (red) and two voxels at resolution 1 (cyan).

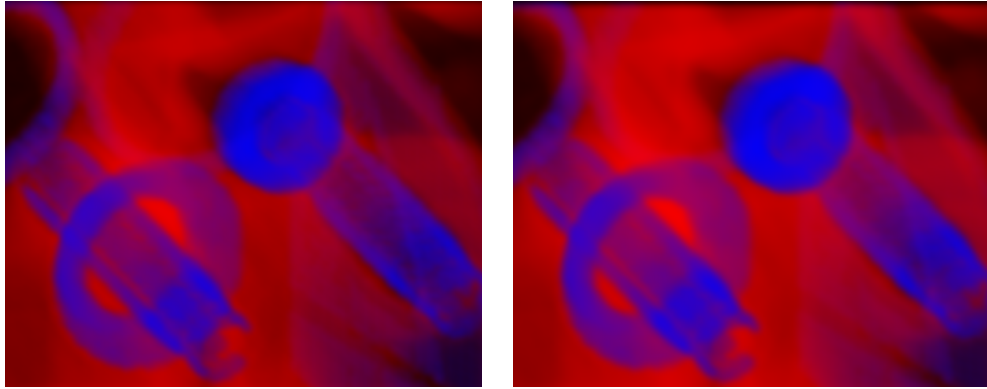
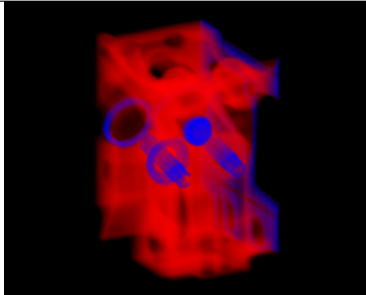
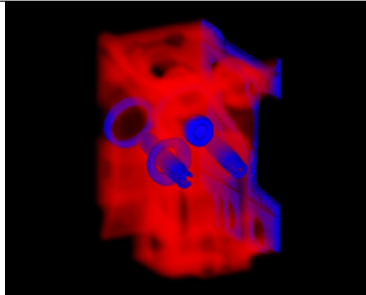
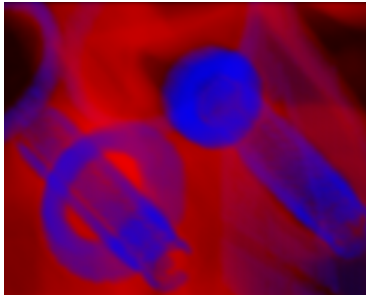

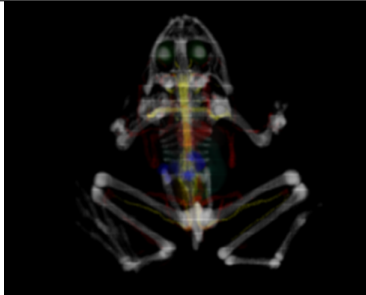
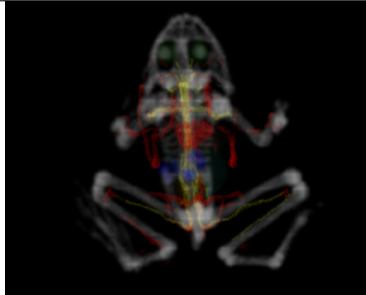
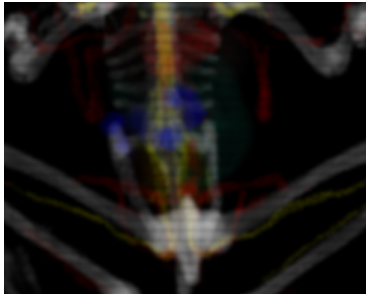
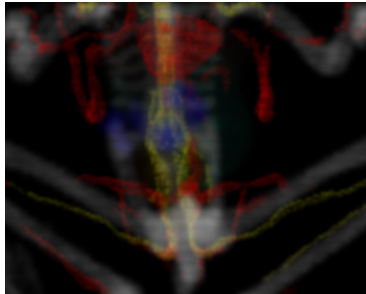


Figure 7: Beta and Gauss footprints used with real datasets: the image rendered using 25x25 pixel beta footprints [left] has mainly the same appearance than the one rendered using 35x35 pixel gauss footprints [right], but requires half the time to be rendered.

Single res. - emabs	Multi.res. - emabs & lambert
Engine dataset	
	
	
Frog dataset	
	
	

**Table 1** - Two real datasets rendered using the *Feature Graph*, the *Beta footprint* and the  $\alpha$ *Composition*