Accelerating Volume Rendering With Quantized Voxels

Benjamin Mora, Jean-Pierre Jessel, René Caubet

Institut de Recherche en Informatique de Toulouse

Université Paul Sabatier, 118 Rte de Narbonne, 31062 Toulouse Cedex 4, France

Abstract

We present here a new algorithm for accelerating volume rendering with an orthographic projection. Because volume rendering handles huge data sets, a reduction in the computational cost of voxel projection is required to obtain interactive volume rendering. We satisfy this issue by using the possibilities of orthographic projection that allows the quantization of voxel positions by subdividing the pixels. The same projection properties are given for all the voxels with the center falling within the same pixel subdivision. In contrast with classical algorithms that require several instructions to compute either the next traversed voxel or the next rasterized pixel, our method needs only one addition instruction and one addressing instruction that is sufficient to determine one projected pixel. Splatting can also have a decisive advantage of it. Our algorithm is well suited for low-end platforms when no hardware acceleration is available. Experimental results show that our rendering rate is better than other existing methods. This algorithm might allow obtaining real-time volume rendering on conventional computers soon.

Keywords: Volume rendering, splatting, interactivity, orthographic viewing, low-end platform, quantization, rasterization.

1 INTRODUCTION

Literature on volume visualization has prospered during the last decade. Recently, most publications have focused on hardware accelerated algorithms [18, 20], because they fully take advantage of the brute computational power. The latter is available through either 3D texture mapping hardware or custom-made architectures. It results in an interactive rendering with 256^3 volumes, which is not possible on low-end PC platform yet. However, there are several drawbacks with hardware acceleration, and price is the first one (while graphics workstations using 3D textures are expensive, custom-made architectures are really out of price for the end-user). On the other hand, personal computers become increasingly impressive. Soon, the solution might come from plugged cards [19], but nowadays we mustn't avoid pure software algorithms that make volume rendering available on every platform.

In this way, if one has to implement a fast volume rendering algorithm, two kinds of methods are available: image-order [3,8,16] and object-order [9, 10, 11, 22]. While the object-order algorithms determine the contributions of each voxel on the image plane, the image-order algorithms compute each pixel one by one. Thus, advantages and disadvantages result from both of them. Image-order techniques take advantage of an easy and fast way of traversing data in order to obtain a good image quality. The main drawback is that one voxel can be encountered either several times or never, which is impossible with object-order techniques. Furthermore, a coherent run of the memory is possible with the latter. However, the object-order methods have the great disadvantage of an expensive computational cost for projection processing. Efficient schemes have been studied, and the shear-warp rendering [9, 19, 22] is currently considered as the fastest software algorithm. Nevertheless, it also has several gaps, and we will demonstrate further the superiority of our method in most cases.

Thus with the aim of choosing the best way to implement an algorithm, technical hardware specifications of the platform must be parsed. Personal computers suffer from two kinds of restrictions that affect the rendering frame rate. First, the computational power, which has been increasing, is still limited. Furthermore, the memory bandwidth and the memory latency are also a problem in the rendering of large data sets. While reduction of the computations seems possible, reduction of the memory bandwidth is more difficult, if we except frequency domain renderings [5]. Thus, methods exist in order to either treat the only significant voxels or bypass hidden voxels, but there is always a minimal number of voxels to compute, especially when the volume is semi-transparent. Furthermore, the memory performances do not increase as much as processor performances. It therefore seems obvious that in most cases, object-order algorithms are more appropriate with personal computers because less memory bandwidth is required. Our method is based on this principle.

Therefore, an object-order algorithm must perform a fast voxel projection in order to be efficient. The fastest way seems to be the shear-warp method because one voxel fits with one pixel of the intermediate image. However, several quality drawbacks appear in this method. First, the two needed resampling steps blur the image and then some information is lost. Secondly, voxels are used as faces during the projection, and then some artifacts appear according to the viewpoint. Third, one ray per voxel limits the image quality. Other object-order methods, like splatting [10, 15, 17, 22], are more expensive, but produce better images. It is obvious that speed and quality are always conflicting.

This paper shows that it is possible to reduce greatly this conflict by using quantized voxels, which allow fast projections of voxels, even if a good image quality is required. Section 2 of this paper explains the main idea of quantized voxel and how to apply it to the voxel projection and to splatting. Section 3 focuses on some implementation choices and section 4 gives the first results of this new method. Finally, analysis and comparisons with other algorithms are done in section 5, which show the competitiveness of our algorithm in most cases.

2 ALGORITHM

2.1 Principle

Preprocessed tables are well known from the programmers as function accelerators. An understandable example is the cosine function where tens of clock cycles are needed. The use of a lookup table allows a result to be obtained immediately (i.e. 1 cycle if the data is into the cache). It is of course an approximate result, and the accuracy depends on the table size. It is acceptable or not, according to the application requirements.

We use this idea for accelerating the processing of the voxel projection. Each voxel is reduced to a quantized voxel before being processed. The “quantized” adjective means that there is a limited set of distinct voxels that is assumed to be representative of all the voxels. Every quantized voxel has its own properties like color, position, and orientation. Fortunately, these properties are often independent, which allows the separated quantization of properties. The use of quantized values for accelerating volume rendering is not new, especially for opacity and shading [4, 9], where lookup tables are often used. However, it has never been extended to the whole voxel processing, and to voxel projection in particular, which is the focus of our article. This can be performed by constraining volume rendering to orthographic projections, because the same view (see fig. 1) is available for all the voxels [11]. This restriction is often used [9, 11, 19], since perspective projection is unneeded in most cases. Therefore, it is possible to reduce any voxel to a quantized voxel that has close properties, with a minimal error. Because both quantized voxel properties are well known and quantized voxels are limited, all the information that is required for voxel treatment can be quickly preprocessed. Then each time a voxel is projected, appropriate tables give values like color, pixel address, etc…

As mentioned above, quantifying properties is possible when properties are independent. Our paper focuses on the projection quantization that allows a fast rasterization scheme. Section 2.2 describes how to perform an efficient projection of the voxels, and section 2.3 extends it to splatting.

2.2 Projection of Voxels

Projection of voxels is much time consuming. This is why quantization of the position property can accelerate volume rendering greatly. We use an orthographic projection in order to perform this quantization. By considering that voxels are parallelepipeds, then the voxel projection is delimited by a convex hexagon that varies only according to the viewpoint. Thus the projection of each voxel is a translated copy of a template hexagon (see Fig. 1).

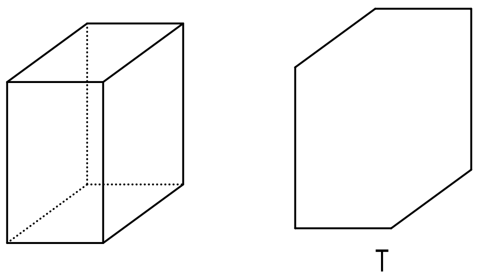


Figure 1:Orthographic view of a voxel and its projected hexagon.

We are now going to introduce several symbols for a better algorithm understanding. All these symbols are defined into the 2D image plane.

In the aim of making a quantization of the position property, the center of the projected hexagon is chosen as criterion. Let T be the template hexagon of the voxel projection, H(C) denotes the translated hexagon of T, with the C point as center. The IN operator is given by:

$$\begin{aligned} &\text{IN (hex, p) = True} && \text{The p point is within the hex hexagon} \\ &\text{IN (hex, p) = False} && \text{Otherwise} \end{aligned} \tag{1}$$

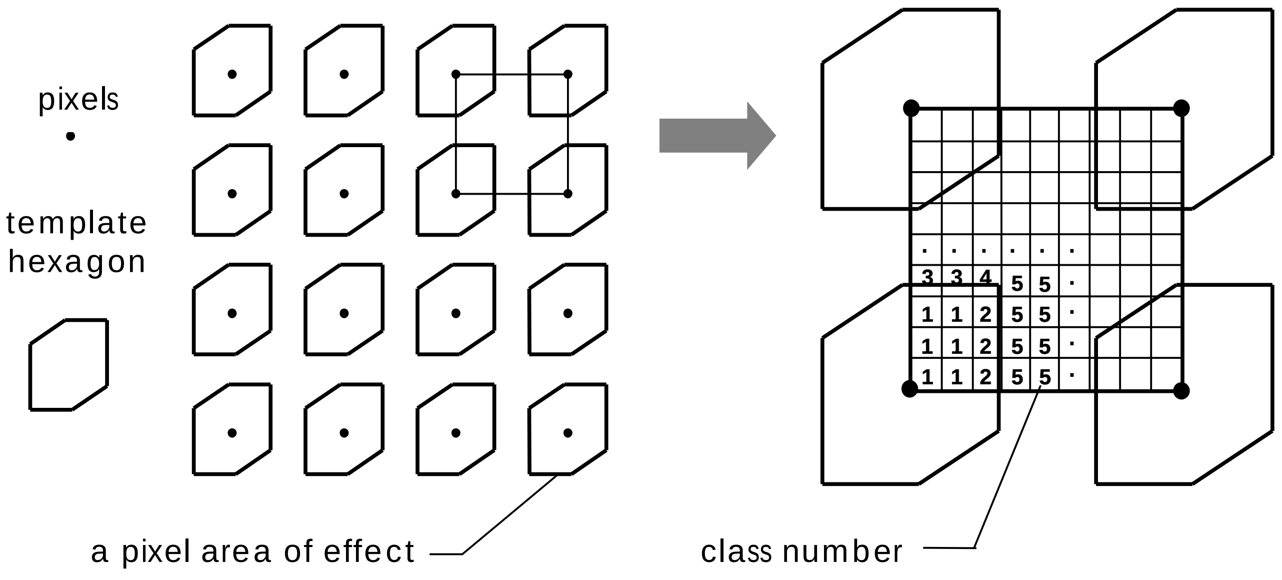
The area of effect concept is introduced now. Let P be any pixel of the image plane, its area of effect (on the image plane) is given by:

$$AOE_H (P)= \{C \mid IN (H (C), P)=True\} \tag{2}$$

It can be summed up as follows: the P pixel belongs to the H(C) hexagon (i.e. P is within the voxel projection) if and only if the C projection of the voxel center belongs to the P area of effect. It is obvious that the area of effect shape only depends on the template hexagon shape. An interesting property of the template hexagon is that its center is a symmetry center. Hereby, AOE_H (P) is equal to H(P) in fact (we do not demonstrate it). Figure 2 shows an example of the areas of effect of the image pixels.

Thus, it is possible to relate the image pixels to a voxel projection represented with a C center. It is defined by:

$$Projection (C)=\{P_i \mid IN (AOE_H (P_i), C) = True\} \tag{3}$$



(a) (b)

Figure 2 : Quantization of the image plane areas of effect.

The equation (3) states that a point list can be associated with a C projected center. This list is created by looking for all the areas of effect (see fig. 2) that contain the C point. If we suppose the position of the C projected center has a limited number of states (i.e. the center position is quantized), then each state can be associated with a point list describing the projected pixels. These lists will be created during a preprocessing step and will be used for solving the voxel projection.

Because the image plane areas of effect are repetitive, as shown in figure 2 (a) above, quantization can be reduced to a square formed by four neighboring pixels. So now, the point lists must be expressed relative to the square position. The edges areas of effect crossing this square divide it into several regions. Then one unique point list per region can be used, wherever the projection of the voxel center points at within a region. However, it takes much computation to solve exactly the region to which the center projection falls. This problem is solved by subdividing the square and by associating point lists with each subdivision. Figure 2 (b) shows that there are two kinds of subdivisions. First, an empty subdivision (free of edges) ensures that the same point list can be used for any voxel center that falls within this subdivision. Next, there are subdivisions crossed by an area of effect edge. This case requires additional computations because the side, which the projection of the voxel center belongs to, must be fixed. It can be done only during the voxel projection by solving the equation (5), where a, b and c are the edge equation parameters stored in an array (during the preprocessing step). The x and y values are the projection coordinates (related to the square) of the voxel center.

$$a.x + b.y \geq -c \tag{5}$$

This test, which is done for each edge of the subdivision, takes two multiplications, one addition, and one comparison. Fortunately, the more the subdivisions are, the greater the probabilities of falling into an empty subdivision are. It results that this additional computation does not increase perceptibly the rendering time.

Thus, two lists per subdivision are required in fact for solving a voxel projection: a list of pixels within the projection and a list of undecided pixels. Because the undecided pixels need to perform tests (5), an equation number referring to an equation list is associated with each undecided pixel.

Fast Implementation

Because the efficiency of our algorithm clearly depends on its implementation, we give some details about it here. Therefore, several parameters must be taken into account. For example, each time a voxel is projected, our algorithm accesses preprocessed tables several times. Consequently, it is important that all the tables remain into the data cache during the execution. This constrains us to choose tables as small as possible and the subdivision size must be considered carefully.

In order to reduce the table size, a segmentation of the subdivided square is done during the preprocessing step. It results from the fact that many subdivisions have the same properties (point list), and thus they can be regrouped. Subdivisions are classified according to the area of effect edges with a region-growing algorithm. Therefore, a class number is given to each subdivision (see fig. 2) instead of storing two point lists per subdivision. In practice, the number of classes is very low, and rarely exceeds one hundred. For example, the figure 2 square can be segmented in nineteen regions: five empty regions, eight regions that are crossed by one line, and six regions that are crossed by two lines. Therefore, one octet is enough to store the class number and the subdivision size can be set to 64 or 128, which take respectively 4 Kb or 16 Kb in memory. Point lists are reduced too, because there is only one list per class. However, a small list of edge equations is needed by classes that are crossed by one edge at least.

The point list implementation must also be optimized. Storing two relative coordinates per pixel would be a naive implementation. Instead of this, a pixel can be addressed by a single integer coordinate that is related to the square. This method demands the image dimensions to be known, but allows a better efficiency of memory addressing. For example (see fig. 2 right part), if the size of the image is 512×512, then the top left corner pixel will be referenced with 0 (0×512+0), the top right corner pixel with 1 (0×512+1), and the bottom right corner pixel with 513 (1×512+1). In that way, the ith pixel affected by the voxel projection can be indexed as following:

$$\text{Pixel\_Coordinate} = \text{Square\_Coordinate} + A[i] \qquad (4)$$

A is an array of integer values associated with the subdivision affected by the projection of the voxel center. Because the pixel coordinate and the square coordinate are also integer values, it takes only one addition and one addressing operation a pixel. That's the reason why our method is so fast, as the section 4 proves.

Until now, we have not explained the preprocessing steps that compute the projection lists. Because several methods are possible to make these lists, we quickly describe here the algorithm we use. It can be divided into three steps. First, a rasterization step computes the intersections between the square subdivisions and all the edges of all the areas of effect around this square. It results in one temporary edge list per subdivision that allows a segmentation step where the subdivisions are classified (the edge list is used as criterion). Once the classification is made, it only remains to set the point lists for each class. The undecided pixel list of a class is quickly determined from the edge list. The sure pixel list is set by taking an arbitrary point of the square belonging to the class and by looking for all the areas of effect that contain it. Although the preprocessing time is subdivision size dependent, results show that our implementation takes a very short part of the rendering time.

2.3 Splatting Application

Splatting is a powerful object-order algorithm that provides high quality images, but it is not currently fast enough to achieve interactive volume rendering. We show that using quantization allows splatting to reach an interactive frame rate. The main idea is the same as previously, but instead of projecting hexagonal shapes, we use fuzzy splats with a gaussian kernel.

An important factor slowing the splatting algorithm is the calculating of the voxel contribution into the screen plane. A footprint table is nowadays largely used for accelerating splatting. Pixels are mapped into the footprint table in order to calculate the voxel energy. However, it requires several instructions per pixel. Thus 7 additions and multiplication per pixel are used by [15], without allowing for addressing instructions. Our algorithm performs only 2 addressing instructions (with cache hits) and 1 integer addition per pixel, which is greatly faster than other existing methods.

Like in section 2.1, a template square, formed with four neighboring pixels, is subdivided. Then it is assumed that every splat falling in the same subdivision behaves as the splat adjusted on the subdivision center (fig. 3). This quantized behavior is represented with two lists a subdivision. A first list stores the pixel indices (cf. section 2.2) that indicate which pixel is affected by the splat. The second list gives the kernel function value associated with the pixel index. These lists are calculated each time the view point changes. However, this preprocessing step is very fast.

Rendering is made as the previous algorithm, except that the weight value is now integrated to the shading processing, and no correction (5) is needed.

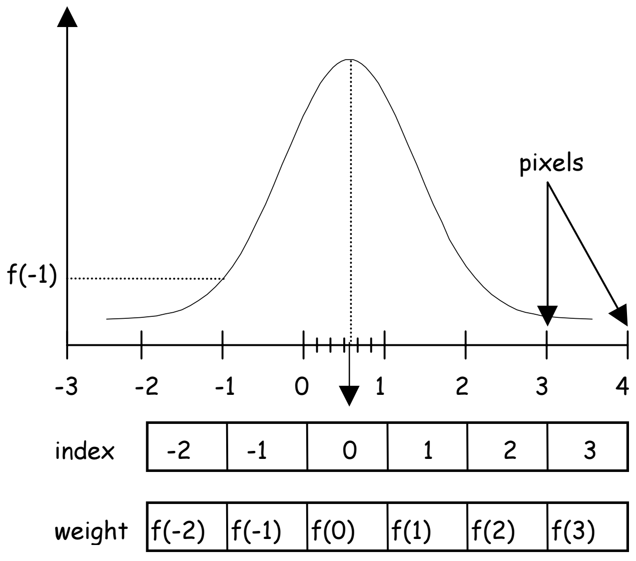


Figure 3: Splatting quantization.

Error Overestimation

Actually, the splat center never falls on a subdivision center, so the position quantization makes an error. The following proves that this error is both measurable and low.

Our algorithm uses a gaussian function (6) as kernel. This function is continuous and derivable at each point.

$$f(x) = b.e^{-r.x^2} \qquad (6)$$

These properties permit us to use the following theorem:

$$\forall x, |f'(x)| \leq a \Rightarrow |f(x_0 + h) - f(x_0)| \leq a.|h| \qquad (7)$$

This theorem states that if we solve the maximum value of the derivative function, we can overestimate the error made by a translation of an h value. So the derivative of f (x) is needed:

$$f'(x) = -2.r.b.x.e^{-r.x^2} \tag{8}$$

Solving the extreme values of a function is generally done by finding values such as the derivative is equal to zero. The following result is given without demonstration:

$$f''(x) = 0 \Rightarrow x = \pm \frac{1}{\sqrt{2r}} \tag{9}$$

With equation (8) and (9), we can then overestimate the derivative. Let n the subdivision factor of the square, the maximum distance between any point of a subdivision and its center is:

$$\frac{1}{n\sqrt{2}} \tag{10}$$

Finally, mixing (7), (8), (9) and (10) gives the error overestimation:

$$\varepsilon \le \frac{b\sqrt{r}}{n\sqrt{e}} \tag{11}$$

This error is valid for a pixel, but it is also an overestimation of the final pixel color. It can be proved easily by considering the final color as a weighted sum of erroneous colors. Therefore, we do not demonstrate it here.

A numerical example is now performed to illustrate the algorithm accuracy. The b value is set to 0.446 and r is set to 2.0 [22]. The subdivision factor is set to 32 (i.e. 1024 subdivisions in all), which seems to be a good compromise. The error overestimation is then 0.012. This error is practically invisible because it is assumed that the standard human eye can't distinguish more than 64 gray levels. Furthermore, the mean value is amply under this score because it is an overestimation.

3 IMPLEMENTATION

This section describes choices that we made in order to get a fast implementation of the two algorithms presented.

3.1 Shading

Shading provides additional clues for the image interpretation, especially when one wants to visualize surfaces. However, its complexity prohibits interactive volume rendering. So, a simplified model must be chosen. Again, quantization is used to solve this problem. Normal vectors are mapped into integer values during a preprocessing step [4] that is only performed one time every volume. Then, a look-up table converts voxel normal vectors to color values [9]. This table is quickly preprocessed each time the viewpoint and the lightning are modified.

A look-up table is also be used to both classify voxels and return opacity values. Therefore, an interactive classification of the volume is possible. Finally, voxels are treated as uniformly shaded.

3.2 Skipping Transparent Voxels

Several techniques have been used [9, 16] in order to skip transparent voxels efficiently. A run-length encoding of the volume is widely used, but suffers from the fact that it needs to be calculated each time the voxel classification changes. Our approach uses a hierarchical two-level grid. Each first-level element is made of a voxel grid and a discretized vector grid. Furthermore, the element stores the minimum value of the voxel grid, which permit to avoid empty spaces. This grid is computed only once for all the volume.

3.3 Volume Traversal

A back-to-front traversal of the volume is chosen. Its advantage is that an accumulation buffer is not needed. Alpha-blending operations are then simplified. The main drawback is that skipping hidden voxels, as in front-to-back methods, is impossible. So front-to-back algorithms are optimized for surface rendering, while back-to-front algorithms are optimized for semi-transparent rendering. A front-to-back implementation of our algorithm for faster rendering of surface is possible, but it will not be discussed here.

In object-order algorithms, voxels are treated one by one. The advantage is that the projected center of the voxel is quickly computed from the previous one by performing an addition for each coordinate. Then, determining the subdivision is easily done with an arithmetic computation of the coordinates.

4 EXPERIMENTAL RESULTS

The first results of our new volume rendering method are given here. Two techniques are presented. First, voxels are projected using the algorithm described in section 2.2. The second one is dedicated to the splatting algorithm (section 2.3).

Three platforms have been tested. A Pentium II Xeon at 450 MHz, which currently represents the standard low-end PC platform, states as the reference. In order to compare from other existing methods, we use a SGI indigo 2 workstation with a R4400 processor clocked at 250 MHz. Finally, our program has also been run on a SGI octane workstation with a R10000 processor at 175 MHz.

Two medical CT data sets have undergone the tests. A well-known 256×256×225 head data set will allow us to compare our works with others classical algorithms in section 5. We also use a bigger 512×512×85 torso data set for more results.

Five renderings have been performed from these two data sets. First, the skull was rendered with a binary voxel classification that does not require pixel accumulation. For a better image quality, the skull has also been rendered with a fuzzy classification. Then accumulation is now needed. After that, a face threshold has been applied on the head data set, and the rendering was performed with both voxel projection (binary classification) and splatting. Finally, a semi-transparent torso result gives us more information.

Now some compilation parameters are given for more understanding. The number of subdivision is 64 for the voxel projection renderings. This number is reduced to 32 for splatting renderings, in order to keep data in cache. The normals have also been discretized into 8192 different values. The resultant images are made of 512×512 pixels. Finally, the program has been written in C language without using SIMD specific instructions.

The speed results are showed in Table 1. Times include all the rendering pipeline (i.e. classification, preprocessing an rendering), and times are averaged according several viewpoints.

Table 1: *Average rendering times (in seconds) implemented on three platforms.*

Data Set	PII Xeon	R10000	R4400
Skull	0.23	0.53	0.75
Skull (smoothed)	0.28	0.65	1.3
Face	0.95	2.2	3.7
Face (Splatting)	3.6	5.2	9.7
Torso	1.3	2.6	4.8

The second table indicates the different preprocessing times. The skull values are equivalent to the face values and so, results are not shown here.

Table 2: *Average times of the preprocessing step according the platform and the method.*

Data Set	PII Xeon	R10000	R4400
Face	< 0.01	< 0.01	0.04
Face (Splatting)	0.06	< 0.01	0.04
Torso	0.01	< 0.01	0.01

The third table gives the number of voxels that are computed for each rendering. The treated column gives the number of voxels contributing to the final image, while the inspected column informs us about the total number of voxels that are analyzed. The difference between these two columns rates the two-level grid efficiency.

Table 3: *Number of voxels (in thousands) that are inspected and computed.*

Data Set	Treated Voxels		Inspected Voxels	
	Number	%	Number	%
Skull	912	6.2	2300	15.6
Skull (smoothed)	1132	7.7	2660	17.6
Face	4330	29.3	6224	42.2
Face (Splatting)	4330	29.3	6224	42.2
Torso	6030	27.0	8532	38.3

5 DISCUSSION AND COMPARISON

This section discusses the previous results and compares them with two other algorithms [16, 23] for a better interpretation.

The table 1 shows that the first advantage of quantized voxels is speed, as it was expected in section 2. With the projection method, we can see that the maximum reached frame rate is 4 Hertz with a 256×256×225 volume (skull rendering), and full data rendering is around one second, which allows a highly interactive tool. Splatting takes also advantage of it, but it is slower because the splat size is greatly bigger than a voxel projection. However, splatting is rendered interactively, which was not possible before with standard computers, and quality is added to the image. We will discuss about quality only in the comparison section because this factor is more difficult to evaluate.

One can be surprised of the performance gap between the Pentium processor and the fast R10000 processor. It seems that our algorithm take advantage of the full speed L2 cache architecture of the Pentium. Furthermore, it is well known that the integer pipeline of R10000 processors is less efficient than Pentium processors. Therefore, it seems obvious that our algorithm is well suited to PC platforms.

Preprocessing times are given in table 2. This time is independent of the number of treated voxel, and this is the minimum rendering time that one can expect. Results indicate that this stage does not increase significantly the rendering process. Better still, it will not interfere with real-time volume rendering if the data set is reduced.

In order to perform an interactive classification, a hierarchical two-level grid structure has been used. However, table 3 shows that this method is not really perfect. Thus, regarding the skull rendering, only 6 percents of voxels are projected, but more than 15 percents are analyzed. It appears that the lower the number of treated voxel is, the less our structure is efficient. A min-max octree could be a better approach [9], but it goes beyond the subject. This paper is focussed on the acceleration induced by the quantized voxels. In that way, the torso rendering example proves it. Up to four millions and half voxels can be shaded, accumulated and projected in one second.

Comparisons

Now, a comparison of the quantized voxel method with other algorithms [16, 23] is made. Shear-warp [9] is chosen because it is currently considered as the reference method in software volume rendering. The second approach referenced here is a recently published ray-casting algorithm [16]. It is easier to make a comparison by using both the same head data set and similar processors.

A shear-warp implementation is presented in [23]. The used computer is a SGI indigo 2 workstation at 150 MHz. The given average time is 2.19 sec. for a fast classification algorithm. Our skull rendering is achieved in 0.75 sec. on the same workstation, but clocked at 250 MHz. By interpolating this result (but performances never increase linearly to the frequency), we might expect a rendering time around 1.25 sec., which is superior to shear-warp. Nevertheless, while the shear-warp image is rendered on a 256×256 image, ours is obtained on a 512×512 image. Furthermore, our threshold setting includes 6 percents of voxels of the entire volume, whereas the shear-warp tests only considers 5 percents. These facts allow us to say that quantized voxels are greatly faster than the shear-warp.

A quality comparison is more difficult to achieve, and results are often subjective. So, the reader is invited to refer to the compared articles in order to make his own opinion. Because our method uses a flat shading when voxels are projected, a poor image quality may be expected. If it is true that the image is aliased, it seems that more details are visible (teethes) in our rendering than the given image in [23]. It can be explained both by the fact that we use a bigger resolution, and because shear-warp made two resampling steps, that partially blurs the image. The first resampling stage happens when slices are processed on the intermediate image. Every voxel of a slice has the same resampling that leads to a reduced image quality. For example, when all weights are equal, it is a well-known blurring filter that is applied to the projected slice. The intermediate image must also undergo a resampling step before becoming the final image.

Nevertheless, a comparison wouldn't be complete without a study of a ray-casting algorithm. A very efficient ray-casting implementation seems to be [16]. A boundary encoding structure encloses the interesting voxel. Therefore it is possible to project the boundary voxels in a preprocessing step in order to determinate both the viable rays, and the beginning voxel of each ray. This algorithm quickly skips empty voxels.

An implementation has been made on SGI power challenge with R10000 processors at 195 MHz. The given time of the skull rendering is 0.47 sec, while our method obtain 0.53 on a 175 MHz processor. So times can be considered equivalent, but, the ray-casting image size has only 256x256 pixels, and interactive classification is not activate for this rendering. Furthermore, these renderings greatly favor ray-casting algorithms because early ray termination is often possible, while our method does not take advantage of it (see section 3.3). In spite of everything, our method is still competitive.

Quality comparison is now more difficult to achieve. Because trilinear interpolation is used in [16], one can expect a very good image quality, which is really the case on the proposed images. However, the low resolution of this image can limit this factor. Our skull rendering is more aliased, but combining both a fuzzy classification of the skull and a higher resolution greatly reduce this problem for a minimal performance penalty. Then, quality is as good as a low-resolution ray-casting.

Splatting takes also advantage of quantization. The splat quantization avoids the footprint rasterization that is time consuming. Thus splatting is accelerated as never before, up to ten times faster, and interactive splatting is now conceivable. Our implementation shows that it is possible to render more than one million voxels per second on a low-end processor. We do not make any splatting comparison, because time differences are really too large to be compared [15, 17].

6 CONCLUSION & FUTURE WORKS

We have proposed a new method for fast estimation of the voxel projection by quantifying the voxel positions. Results clearly show the speed superiority of our method compared to existing methods, due to a high fill-rate. We demonstrate that interactive volume rendering is nowadays possible on low-end platforms. Furthermore, we present a variation of this method oriented to splatting that clearly outperforms the classical use of footprint tables. Splatting can be considered as a fully interactive method by now.

This paper could be a way to a new kind of algorithm that is based on quantized voxels. Soon, it should be possible to obtain real-time volume rendering on 256^3 volumes. We are investigating methods to map this algorithm into perspective rendering, which seems to be a hard task, and for increased surface rendering and image quality.

7 ACKNOWLEDGEMENTS

We thank the CIRAD (Montpellier, France) for their contribution through the C2000 software.

References

[1] W. Lorensen and H. Cline, “Marching cubes: A high resolution 3D surface construction algorithm”, SIGGRAPH’87, 1987, pp. 163-169.

[2] J. Kajiya and B. Von Herzen, “Ray tracing volume densities”, SIGGRAPH’84, July 1984, pp. 165-174.

[3] J. Amanatides, A. Woo, “A fast voxel traversal algorithm for raytracing”, Eurographics’87, 1987, pp. 3-9.

[4] A. S. Glassner, “Normal coding”, Graphics gems, Academic press 1990, pp. 257-264.

[5] Takashi Totsuka and Marc Levoy, “Frequency domain volume rendering”, SIGGRAPH 93, 1993, pp. 271-278.

[6] N. Stolte and René Caubet, “Discrete ray-tracing of Huge Voxel Spaces”, Eurographics’95, vol. 14, no. 3, 1995, pp. 383-394.

[7] M. Levoy, “Display of surfaces from volume data”, IEEE Comp. Graph. & App., Vol. 8, no. 5, 1988, pp. 29-37, 1988.

[8] M. Levoy, “Efficient raytracing of volume data”, ACM Transactions on graphics, vol. 9, no. 3, 1990, pp. 245-261.

[9] P. Lacroute and M. Levoy, “Fast volume rendering using a shear-warp factorization of the viewing transformation”, SIGGRAPH’94, 1994, pp. 451-458.

[10] L. Westover, “Footprint evaluation for volume rendering”, SIGGRAPH’90, 1990, pp. 367-376.

[11] J. Wilhelms and A. Van Gelder, “A coherent projection approach for direct volume rendering”, SIGGRAPH’91, 1991, pp. 275-284.

[12] P. Sabella, “A rendering algorithm for visualizing 3D scalar fields”, SIGGRAPH’88, 1988, pp. 51-58.

[13] R. A. Drebin, L. Carpenter and P. Hanrahan, “Volume rendering”, SIGGRAPH’88, 1988, pp. 65-74.

[14] D. Laur and P. Hanrahan, “A progressive refinement algorithm for volume rendering”, SIGGRAPH 91, 1991, pp. 285-288.

[15] K. Mueller and R. Yagel, “Fast perspective volume rendering with splatting by utilizing a ray-driven approach”, Proc. Visualization’96, 1996, pp. 65-72.

[16] M. Wan, A. Kaufman and S. Bryson, “High performance presence-accelerated ray casting”, Proc. Visualization '99, 1999, pp. 379-386.

[17] K. Mueller, T. Möller and R. Crawfis, “Splatting without the blur”, Proc. Visualization'99, 1999, pp. 363-371.

[18] R. Westermann and T. Ertl, “Efficiently using graphics hardware in volume rendering applications”, SIGGRAPH’98, 1998, pp. 169-177.

[19] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer and L. Seiler, “The volumepro real-time ray-casting system”, SIGGRAPH’99, 1999, pp. 251-260.

[20] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibsonv, W. Hiatt and H. Ohkami, “EM-Cube: an architecture for low-cost real-time volume rendering”, Proc. 1997 Siggraph/Eurographics workshop on graphics hardware”, august 1997, pp. 131-138.

[21] R. Yagel and A. Kaufman, “Template-based volume viewing”, Proc. Eurographics’92, vol.11, no.3, pp. 153-167.

[22] K. Mueller and R. Crawfis, “Eliminating popping artifacts in sheet buffer-based splatting”, Proc. Visualization’98, 1998, pp. 239-245.

[23] P. Lacroute, “Fast volume rendering using a shear-warp factorization of the viewing transformation”, Technical report, September 95.

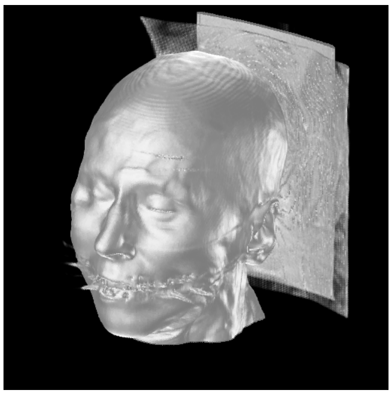


Figure 5a : Volume rendering of a face with voxel projection (0.95 sec.).

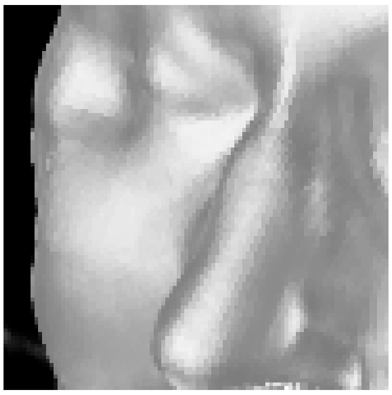


Figure 6a: Zoomed volume rendering of a face with voxel projection.

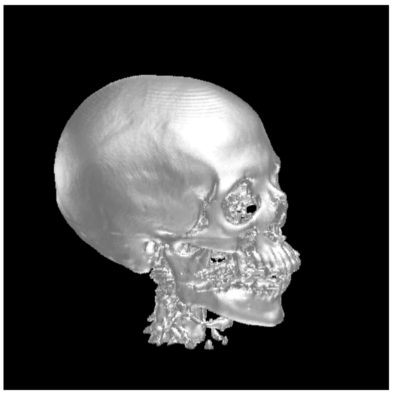


Figure 7a: Volume rendering of a skull with voxel projection (0.23 sec.).

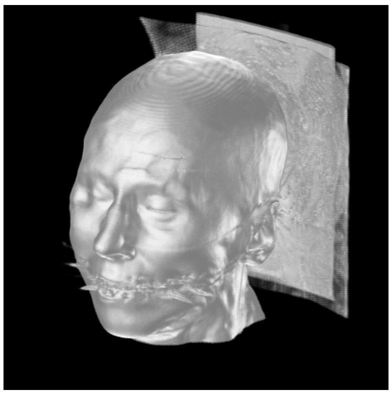


Figure 5b : Volume rendering of a face with splatting (3.6 sec.).

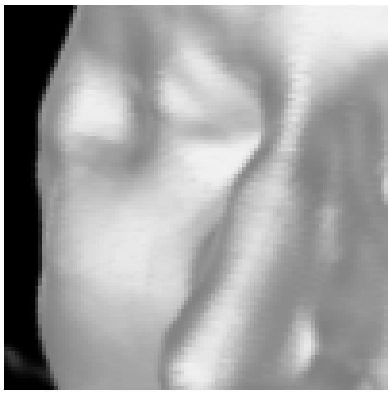


Figure 6b: Zoomed volume rendering of a face with splatting.

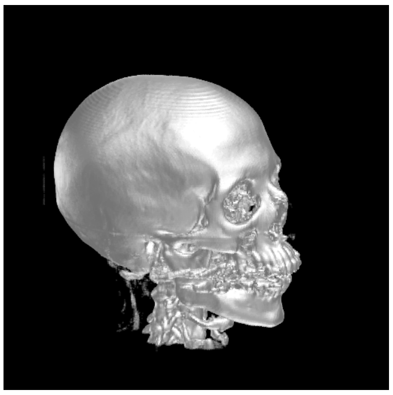


Figure 7b: Smoothed volume rendering of a skull with voxel projection (0.28 sec.).

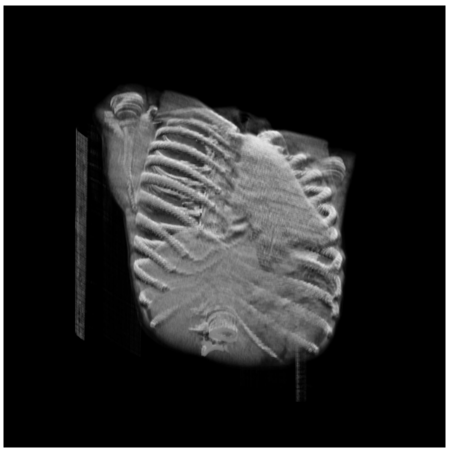


Figure 8: Volume rendering of a torso (1.3 sec.).

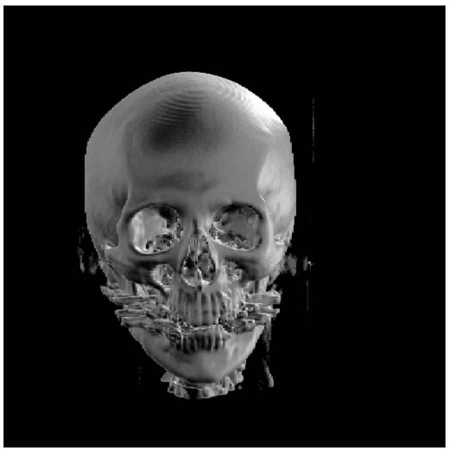


Figure 9: Smoothed volume rendering of a skull with voxel projection (0.28 sec.).