A Low Cost Antialiased Space Filled Voxelization Of Polygonal Objects

Sébastien Thon, Gilles Gesquière, Romain Raffin
LSIS Laboratory
Marseille, University of Provence, France
{Sebastien.Thon, Gilles.Gesquiere, Romain.Raffin}@up.univ-mrs.fr

Abstract

In a virtual sculpting environment, we manipulate objects as a set of volume elements (voxels). In order to start the sculpture process from a polygonal object, we have to discretize this object as a set of voxels. This step is called voxelization. Several voxelization methods have already been proposed, but none matches all of our criteria.

In this paper, we propose a practical approach that handles these criteria, based on an optimized ray casting. The method provides a voxelization of the inner space of a polygonal object, and not only of its surface. It is designed to work with closed objects which may contain holes or tunnels. Even if our goal is not real time conversion, it is fast enough to provide voxelization of objects made of several thousands of triangles within few seconds. As voxelization is a 3D sampling process, it entails aliasing problems. Our method allows a reduction of aliasing by using a low cost oversampling in order to compute grey scale values for the voxels. Moreover our method has a very low cost in memory and allows unlimited voxelization size (but by disk size), even for low memory computers. Furthermore, the method can be used on a personal computer without specific graphics hardware.

Keywords: Voxelization, antialiasing, ray-casting, volume data, polygonal meshes, virtual sculpting.

1. INTRODUCTION

Numerous applications using three dimensional objects based on voxelization are found in the context of medical imaging, physical simulation or terrain visualization. In the context of virtual sculpting, we manipulate such objects [GH91][FCG00]. In our sculpture metaphor based on voxels, we can add, delete or modify the initial volume with tools created with previously made objects. As artists may want to import different objects described with triangular meshes, we need to transform these polygonal objects in a spatial enumeration [CK95], [J96], [FL00]. For our sculpture application, it is important to obtain space filled voxelization, and not only a discretization of the object surface. Furthermore, holes or tunnels must be taken into account [KC93].

However this discrete representation suffers from a blocky appearance due to the fact that only cube faces are displayed. In the context of this paper, we only discuss the voxelization process, and not the visualization of the resulting voxel set. Information on visualization can be found in [S96], [THBP90] and [CKBS90]. Among these methods, the use of gray scale levels associated to the voxels permits to reduce this blocky appearance. As our application provides such gray scale data it could be used by such visualization methods.

With our method, fast results are achieved for meshes made of several thousands of triangular polygons. We will see that the memory cost is very low. Furthermore, the method can be used on a personal computer without specific graphics hardware.

In this paper, section 2 details the current methods for producing voxelized objects. Section 3 will demonstrate our new method for producing space filled voxelization in gray scaled levels. The process is demonstrated to be effective from the viewpoints of computational time, memory use, and accuracy of representation. Results are exposed in section 4.

2. BACKGROUND

There exist several ways to perform object voxelization, by using object topology (point, curve, surface, or volume), geometry (implicit expression, Brep, CSG, parametric…) or graphics hardware. These methods have different goals like real time, volume analysis, accuracy or robustness.

Voxelization methods for parametric surface [K87] or implicitly defined objects [SC97] have been proposed. However, we focus on voxelization of polygonal objects.

[J96] discuss an efficient method based on the computation of distance from voxels to surface. It allows a smooth representation of the object with normal calculation. Jones used distance fields to voxelize objects [JS00]. However, the drawback of this distance based approach is its relative slowness due to the distance computations. As we don't make use of distance information in our application, this technique doesn't satisfy our needs.

Using a propagation of filled cells, starting from an inside one, [HW02] propose to voxelize objects closed or not, by the construction of an octree representation of the scene. As the robustness of cells classification can be inadequate, we prefer to restrict to closed objects. As the purpose of this method is to take into account very general meshes, with problems such as cracks or overlapping geometry, it is very slow.

Hardware accelerated solutions using graphics hardware have also been studied. In [FC00], clipping planes are used to display slices of equal *z* thickness of a 3D object to the frame buffer. Colored pixels read slice after slice from the frame buffer are used to reconstruct a 3D matrix of voxels. However, some problems can occur, such as holes or missed fine objects.

[KPT99] reconstruct a 3D set of voxels by rendering a polygonal object onto three z-buffer. However, this method can only be used to voxelize the surface of the object, and not its inner space. Moreover, some concavities cannot be detected.

Visualization of a voxelized volume often provides a blocky aspect to objects. Sculpting environment must provide various visualizations, for example rough but fast representation during the interactive step of sculpting, and precise but perhaps slower representation when the artist releases his tool. A first way to

ensure a smooth object surface is to use an operation that will create a new polygonal object, such as marching cubes [LC87]. This would be done each time the object is modified. As in [FPRJ00], we can use a distance field with trilinear operations on the values stored in voxels (see also in [J00]). These methods lead to a smooth object (or visually smooth) but implies global computations on the object.

In the framework of our sculpture application, we prefer gray-levels based methods such as [S94] or [SK98], as the new method presented in this paper allows the computation of gray scale per voxel. Such methods enable smooth visualization of surfaces by local computation of normals using voxels vicinity. However, visualization issues will not be tackled in this paper.

Among the presented methods, none matches all of our criteria. Indeed, we need a method that discretizes the inner space of a closed polygonal mesh as a set of voxels, and not only its surface. Cavities inside the objects must be detected. Aliasing problems must be tackled. Fast computation are also needed (although not in real time), with low memory usage. Furthermore, the method must be usable on a personal computer without specific graphics hardware.

In the following, we will propose a new method that handles all these criteria.

3. OUR VOXELIZATION METHOD

3.1 Presentation

Our method allows to compute an uniform voxelization of a 3D polygonal object. It is based on an optimized raycasting through the faces of the polygonal object, in order to determine for each voxel if it lies inside or outside the object. This is done in two steps, first a space partitioning of the object faces (section 3.2) in order to speed up the second step, and then the voxelization by raycasting trough the space partitioned faces (section 3.3).

3.2 Space partitioning

As we will need to compute in a second step (section 3.3) the intersections between rays and the faces of a polygonal object, we first partition the space in order to speed up the computations. We compute the bounding box of the polygonal object, and we partition the face of the bounding box that faces the *xy* plane by using a quadtree. Each node of the quadtree corresponds to an axis aligned box. These boxes have a depth that is equal to the depth of the object bounding box (Figure 1). Each leaf of the quadtree contains a list of the object faces that intersect the corresponding box. We subdivide the polygonal object bounding box by using this quadtree of boxes as long as a box contains a number of faces greater than a user defined threshold or until a user defined tree depth is reached.

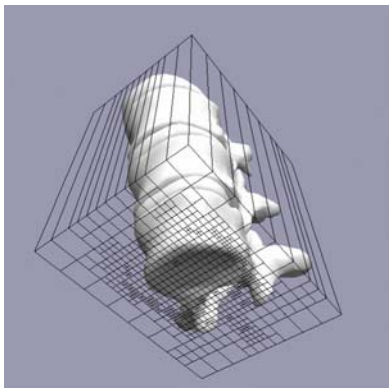


Figure 1: Partitioning of a polygonal object bounding box with a quadtree of boxes.

3.3 Voxelization

3.3.1 Presentation

Once the polygonal object space has been partitioned, and all its faces stored in a quadtree of boxes, we proceed now to the voxelization step.

We first define a 3D uniform grid embedding the polygonal object. The user defines the size of the grid as well as the number of voxels along the three axes. These values are used in a raycasting step, in order to compute the origin of each ray that is cast through the object (section 3.3.2).

Along each ray, for each traversed voxel, we determine if the voxel is inside or outside the object (section 3.3.3), and thus we give to the voxel a value indicating if it is filled or empty.

As this voxelization step is a 3D discretization of a polygonal object, it entails aliasing problems (missing details, disconnected parts, etc.). But we'll propose in section 3.3.4 an antialiasing technique by computing for each voxel a value corresponding to the ratio of the voxel volume that is inside the polygonal object.

3.3.2 Optimized raycasting

Raycasting is the core of our method. We cast a ray parallel to *z* positive axis from the center of each cell of the side of the 3D uniform grid facing *xy* plane (Figure 2). Thus, if we have a 3D grid composed of *n* x *m* x *p* cells, we only cast *n* x *m* rays (we will see in section 3.3.4 that we can cast more rays for antialiasing).

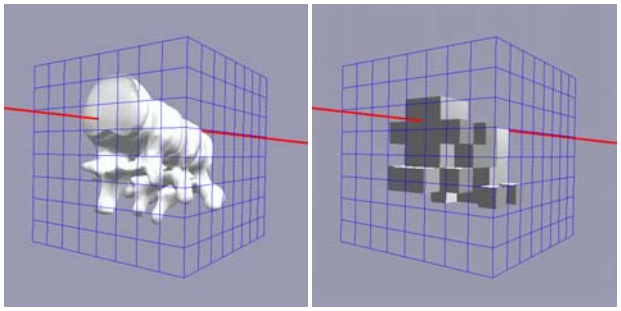


Figure 2: Rays are cast along *z* axis, from centers of cells of one side of the 3D grid (here, 8x8x8 grid as an example)

Says (x_o,y_o,z_o) the origin of a cast ray. We first determine the box corresponding to a leaf of the quadtree computed in section 3.2 that will be traversed by the ray (Figure 3), by looking in 2D

recursively in the quadtree for the box that contains the (x_o,y_o) point.

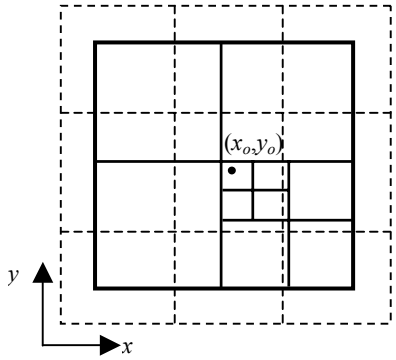


Figure 3: Choice in 2D of the box in the quadtree that contains the origin of a cast ray (=center of a uniform grid cell).

Then, we test the intersections between the ray and all the faces contained into the box. For each intersection point found, its *z* coordinate value is added in a linked list of floats. The linked list is sorted by increasing value of *z*. This linked list is emptied before casting a new ray.

Thus, we process the 3D grid in 2D cell by cell along the *xy* plane to cast a ray along *z* axis, and for each ray we compute a sorted linked list of *z* components of the intersection points with the object faces. The next step is to determine, for each voxel of the 3D grid traversed by the ray along *z* axis, if it lies inside or outside the object, in order to give it a value "inside" or "outside".

3.3.3 Inside/outside determination

In order to voxelize the polygonal object, we have to determine for each voxel traversed by the same ray if this voxel is inside or outside the object.

We use the paradigm of the determination of a point in a 2D polygon. In order to test whether a point is contained in a polygon, one has to count the number of times a ray starting at the point intersects an edge of the polygon. If this number is odd, the point is inside (Figure 4).

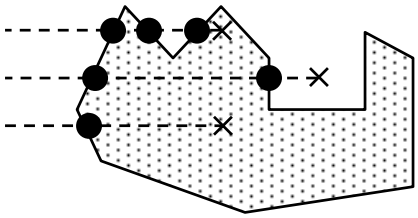


Figure 4: 2D test of a point in a polygon.

We use the same paradigm in a 3D context: in order to test whether a point is contained in a polyhedron, we count the number of times a ray starting at the point intersects a face of the polyhedron. If this number is odd, the point is inside.

But in order to speed up computation, we don't cast a ray from each voxel, because it would be far too expensive. Instead, we only cast one ray for an entire row of voxels. Thus, this is only a $O(n^2)$ complexity instead of $O(n^3)$, if we consider a cubic matrix of *n* x *n* x *n* voxels.

For each voxel traversed by the same ray along *z* axis, we compare its *z* center component value with the *z* values of the same linked list. As the values are sorted by increasing values in the list, the search is very fast. We count the number of values in the list that are greater than the *z* of the center. If this number is odd, then the center of the voxel is inside the object (Figure 5).

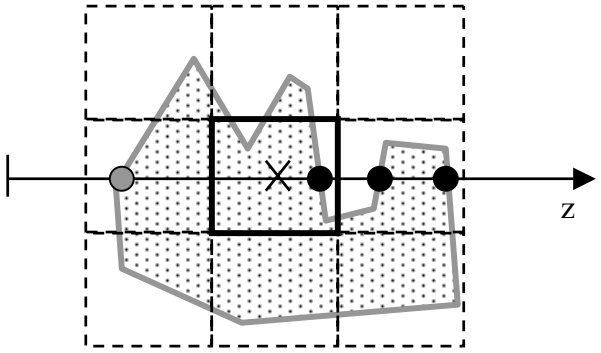
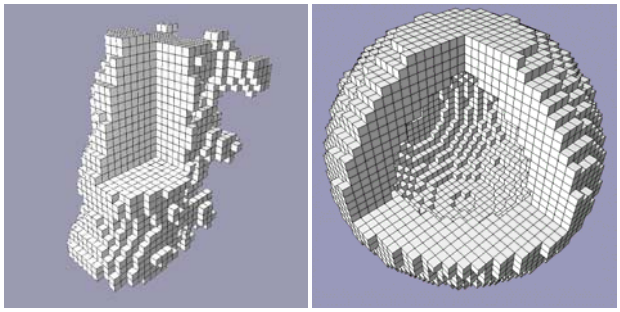


Figure 5: Test if the center of the voxel is inside the polyhedral object by counting the number of intersection points having a greater *z* in the linked list associated to the traversal ray. Here, there is an odd number of intersections (3), then the center of the voxel is inside.

Thus, this method allows to fill the space inside the resulting voxelized object, by marking all the inside voxels as filled, and not only the voxels on the surface (Figure 6a). Moreover, holes in the object volume can be taken into account, contrarily to [KPT99] (Figure 6b).



(a) (b)

Figure 6: With our voxelization method, inside voxels are filled (**a**: cut view of a vertebrae), and holes are taken into account (**b**: cut view of the cavity of a hollow sphere)

However it is important to note that we don't allocate memory for a 3D matrix corresponding to all the voxels. We don't store in memory all the voxels values that we compute. On the contrary, we write on the fly each computed voxel value in a file. Thus, our method requires very little memory space. As a consequence, this "direct to disk" technique allows the computation of very big voxelization, independently of the computer amount of memory. The only limit is disk size.

3.3.4 Antialiasing

Voxelization of a polygonal object is a 3D sampling process, and is consequently prone to aliasing problems such as missing details

or disconnected parts. Our method allows a reduction of aliasing by computing either a simple binary inside/outside value for each voxel, or a value corresponding to the amount of voxel volume that is inside the polygonal object. This value is obtained by oversampling each voxel : instead of casting only one ray through a row of voxels, we cast several rays. This oversampling can be uniform (Figure 7) or stochastic (Figure 8).

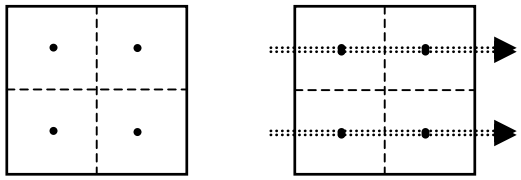


Figure 7: Uniform 2x2x2 oversampling of 4 rays trough a voxel, with two tested points along each ray. Front (a) and side view of the voxel (b).

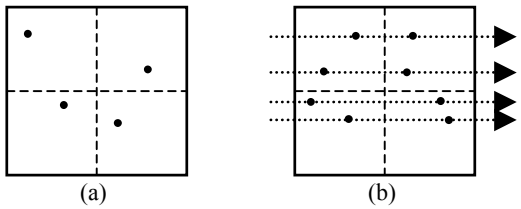


Figure 8: Stochastic 2x2x2 oversampling of 4 rays trough a voxel, with two tested points along each ray. Front (a) and side view of the voxel (b).

For each ray, we compute a linked list of intersection points with the faces in the quadtree. Then, for each voxel in the row traversed by these rays, we compute a value between "empty" and "filled". This value is obtained as follows: for each ray traversing the voxel, we consider several points on the ray and inside the voxel (uniformly along the ray in the case of uniform sampling, or at random position along the ray in the case of stochastic sampling). For each point, we test if the point is inside or outside the polygonal object like in section 3.3.3. Thus, if we oversample each voxel with n rays, and if we test m points along each of these rays, then we evaluate the inside/outside test for n x m points for each voxel. We obtain n_i points inside and n_o points outside, with $n_i + n_o = n.m$. The value given to the voxel is a byte between 0 (totally empty) and 255 (totally filled) computed by:

$$value = \frac{n_i}{n.m}.255$$

As a result, we obtain less aliased voxelizations (Figure 9). The results presented on Figure 10b exhibit gray levels corresponding to the state of the voxels. The darker the voxel, the less matter it contains. Note that our goal is not realistic rendering. In order to use efficiently these gray levels for rendering, techniques such as [S96] should be used.

Figure 9: Binary 128^3 voxelization (a) and antialiased greyscale voxelization (uniform 4x4x4 oversampling) (b).

This antialiasing technique can also be used in order to provide binary voxelization by setting a threshold on the computed oversampled values. If the value if greater than the threshold, then we consider that the voxel is inside, else it is outside. Thus, it allows to reduce the loss of details smaller than the sampling frequency (Figure 10).

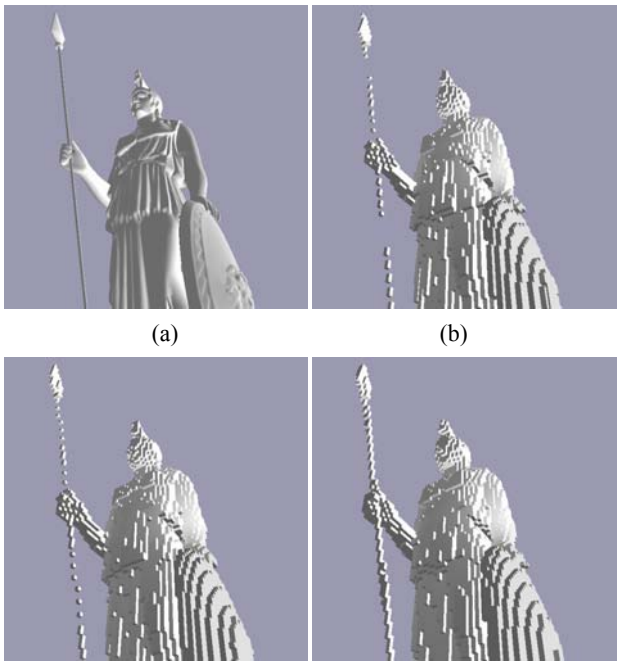


Figure 10: Polygonal model (a), binary 128^3 voxelization without antialiasing (b), uniform 2x2x2 oversampling (c), uniform 4x4x4 oversampling (d).

Computation times for antialiasing during voxelization step are presented in Table 1 for *Athena* object composed of 91016 faces.

	Without	**2x2x2**	**3x3x3**	**4x4x4**
64^3	0.016	0.11	0.266	0.515
128^3	0.094	0.656	1.625	3.109
256^3	0.703	4.235	11.094	21.203
512^3	5.266	30.078	88.141	168.547

Table 1. Antialiasing times (in sec.) for the *Athena* object.

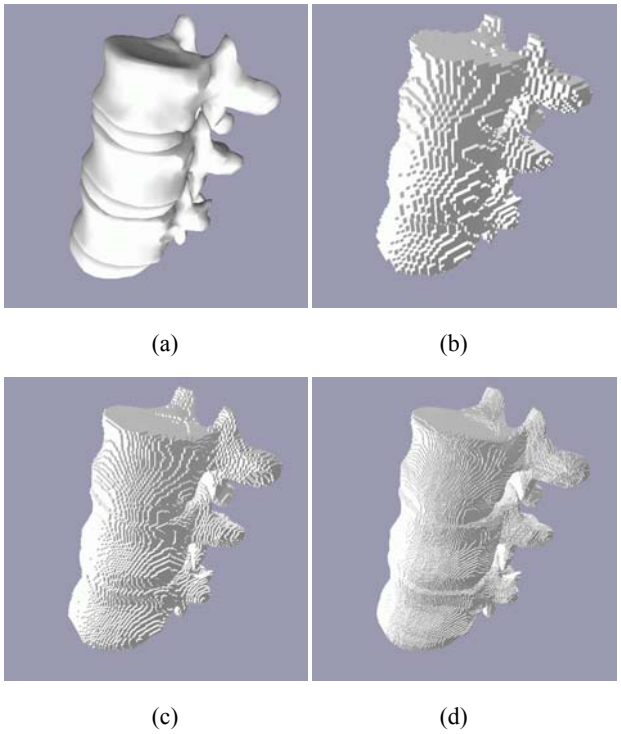
3.3.5 Algorithm

The voxelization process can be sum up as the following algorithm:

```
OPEN file FOR OUTPUT
WRITE header TO file

origin = min point of object bounding box
delta = size of a voxel
dim = voxels number along an axis (ex:256)

FOR y=0 TO dim DO
  FOR x=0 TO dim DO
    CLEAR Zlist

    // Starting point of the ray is the center
    // of the (x,y) cell
    pos.x = origin.x + (x+0.5)*delta
    pos.y = origin.y + (y+0.5)*delta
    pos.z = origin.z

    // We compute the intersections between the
    // ray and the faces in the quadtree of boxes.
    // z component of intersection points are
    // stored in the sorted linked list Zlist
    Zlist = quadtree.ray_intersect(pos)

    // zc = z component of voxel center
    zc = origin.z + delta/2;

    // For each voxel along z at the same (x,y)
    // location, we test if its center is inside
    // or outside the object
    FOR z=0 TO dim DO
      // We look in the linked list Zlist for the
      // number of intersection points along the
      // ray greater than the tested point
      nb_z_sup = Zlist.nb_z_sup(zc)

      // If this number is even, then the tested
      // point is inside the polygonal object
      IF even(nb_z_sup)
         WRITE 1 TO file    // inside voxel
      ELSE
        WRITE 0 TO file     // outside voxel
      END IF
      zc = zc + delta;
    END FOR
  END FOR
END FOR

CLOSE file
```

4. RESULTS

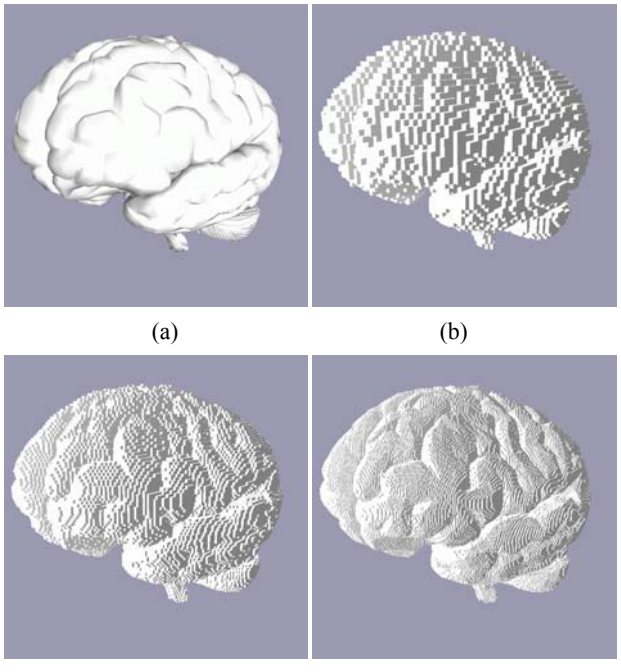
All the following results have been obtained on an Intel XEON 2.66 GHz with 512 Mo.



(a) (b)

(c) (d)

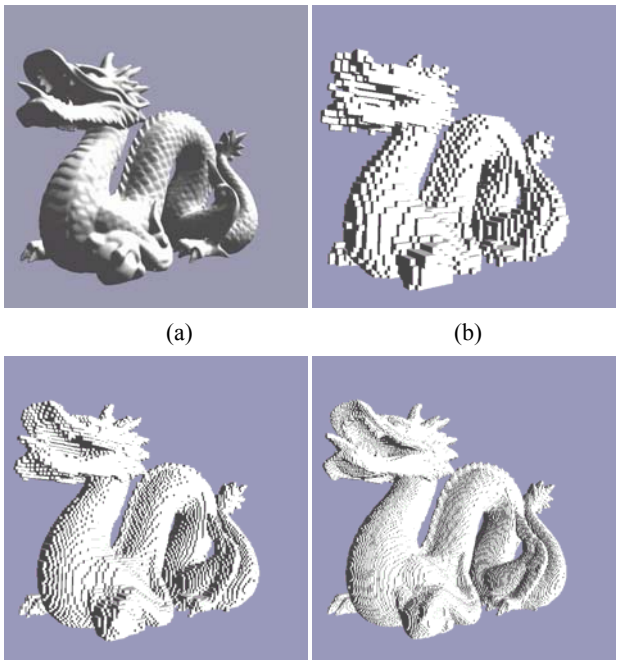
Figure 11: *Vertebrae*. Polygonal model (a) and voxelization at 64^3 (b), 128^3 (c) and 256^3 (d) levels.



(a) (b)

(c) (d)

Figure 12: *Brain*. Polygonal model (a) and voxelization at 64^3 (b), 128^3 (c) and 256^3 (d) levels.



(a) (b)

(c) (d)

Figure 13: *Dragon*. Polygonal model (a) and voxelization at 64^3 (b), 128^3 (c) and 256^3 (d) levels.

Computation times for the preceding examples are presented in Table 2. The raycasting stage is accelerated by the use of the quadtree. More precisely, the speed is very dependent on the quadtree maximal depth, as we can see in Table 3.

	Vertebrae	**Brain**	**Dragon**
Faces	10444	18847	871414
Partition (sec.)	0.125	0.36	7.047
Voxelization :	Time (sec.)		
64^3	0.047	0.094	0.75
128^3	0.172	0.407	2.922
256^3	0.87	1.844	11.578
512^3	6.766	9.829	47.89

Table 2: Partition and voxelization times (in sec.) for different polygonal objects with a maximal quadtree depth of 5.

Quadtree max depth	**4**	**6**	**8**	**10**	**12**
Partition (sec.)	5.578	9.484	12.25	12.297	13.266
Voxelization :	Time (sec.)				
64^3	2.922	0.25	0.047	0.046	0.063
128^3	11.547	0.875	0.234	0.234	0.219
256^3	46.078	3.531	1.031	1.031	1
512^3	186.5	15.844	6.125	6.032	6.453

Table 3: Influence of the quadtree maximal depth on the voxelization time (in sec.), for the *Dragon* object.

Software using our method as well as voxelized objects are available on our web page: `www.iut-arles.up.univ-mrs.fr/thon/recherche/GraphiCon2004/`

5. DISCUSSION

5.1 Advantages

Our method allows to voxelize a polygonal object by filling its inner space, and not only its surface as most of the existing voxelization methods do.

The method is based on an optimized raycasting. Although it is not designed to achieve real-time, pretty fast results are obtained as we can see in Table 2. For example, the total computation time in 256^3 mode (partition step plus voxelization step) for the vertebrae mesh counting 10444 faces is less than 1 second.

Moreover, contrarily to hardware-based methods [FC00] [FL00], this method can be used on any computer, even if this computer does not include a video card or if its video card is not powerful. Furthermore, voxelization of any size can be obtained, whereas hardware-based methods are limited by frame buffer or z-buffer maximal dimensions.

As we can see on results presented in Table 3, the speed of the voxelization step is very dependent on the maximal depth of the quadtree. As we increase the maximal depth, the partition step time is increased, but the voxelization step time is dramatically decreased. However, a compromise has to be found, because if the quadtree subdivision is too important, then its construction time during the partition step will be too high and the recursive search of a box in this tree during the voxelization step will begin to increase. For the example of the *Dragon* mesh in Table 3, the voxelization time decreases until a depth of 10, but starts to increase for bigger depth values.

If we sum the time of the partition and voxelization steps, then for the example of the *Dragon* mesh the best compromise is to use a maximal quadtree depth of 8 for important voxelizations such as 256^3, 512^3 of higher.

Our method has a low cost in memory, as we only store in memory polygonal object data as a quadtree and one linked list of floats. This is the linked list of z components of intersection points along a ray. The same list is used for a whole row of z axis aligned voxels traversed by this ray. Moreover, we don't store in memory the entire computed voxels 3D matrix, as we directly save in a file each voxel value that is computed. Thus, very large voxelizations can be done, as the only limitation is disk size.

In order to reduce aliasing problems inherent to the voxelization process, we allow the possibility to compute for each voxel a value corresponding to the proportion of its volume inside the polygonal object, instead of a simple binary "inside/outside" value.

5.2 Disavantages

As this method is based on raycasting, it is slower than hardware based techniques. However, even if it cannot be used in a real-time framework, voxelized objects are obtained within few seconds.

The polygonal objects to convert must have a closed surface, as we suppose that if a ray first encounters a face, then it is inside the object until it hits another face.

The presented method only computes uniform 3D grids of voxels. However, it could easily be extended to the creation of adaptive structures, such as an octree, in order to spare memory (disk size in our case). The subdivision criterion could be the voxel value computed in section 3.3.4 for antialiasing: if this value is greater than a given threshold, then the voxel is subdivided in 8 sub-voxels, and so on, and all these values are stored in an octree.

6. CONCLUSIONS AND FUTURE WORKS

We proposed a voxelization method for polygonal objects based on an optimized raycasting. This method allows to fill the inner space of the object with voxels. Aliasing problems inherent to the sampling process of voxelization are tackled. Even if our objective is not real-time, the method is fast enough to provide results within few seconds for polygonal objects made of several thousands of faces for large voxelizations such as 512^3.

As our method is based on a raycasting, it can be easily extended to the voxelization of other object types than polygonal objects. Thanks to the object oriented architecture of our software, it only requires to provide a class defining a new object type, such as for example implicit surfaces, nurbs or analytic objects (sphere, cone, etc.), and to provide a method for this class that compute the intersection between this object and a ray. As long as we can compute intersection points along a ray, the remainder of the process is the same as presented in this paper.

As future works, we plan to take into account polygonal objects that are not correctly closed, with missing faces. We also plan to compute adaptively the voxels in an octree.

7. ACKNOWLEDGMENTS

The authors would like to thank Eric Remy for his reading of this paper.

The *Vertebrae* and *Brain* models are free models taken from `www.3dcafe.com`. The *Athena* mesh is a free model proposed by *DeEspona* at `www.deespona.com`. The *Dragon* model has been taken from the *Large Geometric Models Archive* at `www.cc.gatech.edu/projects/large_models`.

8. REFERENCES

[CKBS90] D. Cohen, A. Kaufman, R. Bakalash, and S. Bergman. Real time discrete shading. The Visual Computer, 6:16-27, 1990.

[CK95] D. Cohen-Or and A. Kaufman. Fundamentals of Surface Voxelization. Graphical Models and Image Processing, Volume 57, Issue 6, November 1995, pp. 453-461.

[FCG00] E. Ferley, M.P. Cani, and J.D. Gascuel. Practical volumetric sculpting. The Visual Computer, 16(8):469–480, December 2000. A preliminary version of this paper appeared in Implicit Surfaces'99, Bordeaux, France, sept 1999.

[FC00] S. Fang and H. Chen. Hardware accelerated Voxelisation. Volume Graphics, Chapter 20, pp. 301– 315. Springer-Verlag, March 2000.

[FL00] S. Fang and D. Liao. Fast CSG Voxelization by Frame Buffer Pixel Mapping. ACM/IEEE Volume Visualization and Graphics Symposium 2000 (Volviz'00), pp. 43-48, Salt Lake City, UT, 9-10 October 2000.

[FPRJ00] S.F. Frisken, R.N. Perry, A. P. Rockwood, and T.R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. Proceedings of SIGGRAPH 2000, pp. 249–254, July 2000. ISBN 1-58113-208-5.

[GH91] T.A. Galyean and J.F. Hughes. Sculpting: An interactive volumetric modeling technique. Computer Graphics, 25(4):267–274, July 1991. Proceedings of SIGGRAPH'91 (Las Vegas, Nevada, July 1991).

[HW02] D. Haumont and N. Warzée. Complete Polygonal Scene Voxelization, Journal of Graphics Tools, Volume 7, Number 3, pp. 27-41, 2002.

[J96] M.W. Jones. The production of volume data from triangular meshes using voxelisation. Computer Graphics Forum, vol. 15, no 5, pp. 311-318, 1996.

[JS00] M. W. Jones and R. Satherley. Voxelisation: Modelling for Volume Graphics. In B. Girod, G. Greiner, H. Niemann, H.-P. Seidel (eds.), Vision, Modeling, and Visualization 2000, IOS Press, pp. 319-326, ISBN 1-58603-104-X.

[KPT99] E.A. Karabassi, G. Papaioannou, and T. Theoharis. A fast depth-buffer-based voxelization algorithm. Journal of Graphics Tools, 4(4):5-10, 1999.

[KC93] A. Kaufman, D. Cohen. Volume Graphics. IEEE Computer, Vol. 26, No. 7, July 1993, pp. 51-64, Ben Gurion University, Roni Yagel, The Ohio State University.

[LC87] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm, Computer Graphics, vol. 21, no. 4, July 1987, pp. 163-169.

[S94] M. Sramek. Gray level voxelisation: a tool for simultaneous rendering of scanned and analytical data. Proc. the 10th Spring School on Computer Graphics and its Applications, Bratislava, Slovak Republic, 1994, pp. 159-168.

[S96] M. Sramek. Visualization of Volumetric Data by Ray Tracing ISBN 3-85403-112-2, Austrian Computer Society, Austria 1998.

[SC97] N. Stolte and R. Caubet. Comparison between Different Rasterization Methods for Implicit Surfaces. In Rae Earnshaw, John A. Vince and How Jones, editors, Visualization and Modeling, Chapter 10, pp. 191-201, Academic Press, April 1997. ISBN 0122277384.

[SK98] M. Sramek, A. Kaufman. Object voxelisation by filtering. In: Proc. IEEE Symposium on Volume Visualization, Research Triangle Park, NC, October 1998, pp. 111-118.

[SK00] M. Sramek and A. Kaufman. vxt: A class library for object voxelisation. In Volume Graphics, pp. 119–134. Springer, 2000.

[THBP90] U. Tiede, K.H. Höhne, M. Bomans, A. Pommert, M. Riemer, G. Wiebecke. Investigation of medical 3D-rendering algorithms. IEEE Computer Graphics Applications 10:2 (1990), 41-53

[WK93] S. W. Wang and A. Kaufman. Volume sampled voxelization of geometric primitives. In Proc. Visualization 93, pp. 78– 84. IEEE CS Press, Los Alamitos, Calif., 1993.

About the authors

Sébastien Thon is an associate professor at the Provence University. He works at the LSIS laboratory. His contact email is Sebastien.Thon@up.univ-mrs.fr

Gilles Gesquière is an associate professor at the Provence University. He works at the LSIS laboratory. His contact email is Gilles.Gesquiere@up.univ-mrs.fr

Romain Raffin is an associate professor at the Provence University. He works at the LSIS laboratory. His contact email is Romain.Raffin@up.univ-mrs.fr