Visual Comput (2007) 23: 535–543
DOI 10.1007/s00371-007-0097-8

ORIGINAL ARTICLE

C.J. Ogáyar
A.J. Rueda
R.J. Segura
F.R. Feito

Fast and simple hardware accelerated voxelizations using simplicial coverings*

Published online: 19 April 2007

C.J. Ogáyar (✉) · A.J. Rueda · R.J. Segura · F.R. Feito
Departamento de Informática
Escuela Politécnica Superior
Universidad de Jaén
Campus Las Lagunillas, Edif. A3
23071 Jaén, Spain
{cogayar, ajrueda, rsegura, ffeito}@ujaen.es

Abstract Voxelization of solids, that is the representation of a solid by a set of voxels that approximates it, is an operation with important applications in fields like solid modeling, physical simulation or volume graphics. Moreover, the new generation of affordable 3D raster displays has renewed the interest on fast voxelization algorithms, as the scan-conversion of a solid is a basic operation on these devices.
In this paper a hardware accelerated method for computing a voxelization of a polyhedron is presented. The algorithm is simple, efficient, robust and handles any kind of polyhedron (self-intersecting, with or without holes, manifold or non-manifold). Three different implementations are described in detail. The first is a conventional implementation in the CPU, the second is a hardware accelerated implementation that uses standard OpenGL primitives, and the third exploits the capabilities of modern GPUs by using vertex programs.

Keywords Voxelization algorithms · Graphics hardware · Volume graphics

1 Introduction

Voxelization of solids is concerned with converting geometric objects from their continuous geometric representation into a set of voxels that approximates it [21]. Voxel representation of solids has applications in solid modeling, volume graphics and physical simulation. It has been extensively used for rendering objects which are difficult to represent by traditional surface representations, like clouds, fire, smoke or terrain models [21]. A new application of voxelization techniques is the 3D scan-conversion of models for the emerging 3D raster displays [2, 6, 24]. This new technology represents the evolution from traditional 2D displays to a new generation of devices that can generate real 3D images, not depending on the position of the observer. Although they are still expensive and quite uncommon, the use of 3D displays will become widespread in the future. These devices work in a similar way to 2D displays: an object has to be previously 3D scan-converted to a voxel framebuffer in order to be visualized.

The voxelization method described in this paper was already presented in [27] as an extension of a 2D rasterization algorithm proposed by the same authors in previous works [28]. This is a fast CPU-based algorithm that handles a wide variety of polyhedral solids: with or without holes, self-intersecting, manifold or non-manifold, and in contrast to most previous approaches, computes the voxelization of the interior of the solid. In this paper, we

*This work has been partially granted by the Ministerio de Ciencia y Tecnología of Spain and the European Union by means of the ERDF funds, under the research project TIN2004-06326-C03-03 and by the Conserjería de Innovación, Ciencia y Empresa of the Junta de Andalucía, under the research project P06-TIC-01403.

describe how this method can be adapted to exploit graphics hardware in two different implementations: the first only uses standard OpenGL primitives whereas the second takes advantage of the programmable capabilities of current GPUs.

The remainder of this paper is structured as follows. Section 2 reviews the existing literature on voxelization techniques, focusing on hardware accelerated methods. In Sect. 3, we describe the basic voxelization algorithm and its theoretical background. Section 4 is focused on the explanation of the tetrahedra scan-conversion. In Sect. 5, we describe the CPU-based and hardware accelerated implementations of the algorithm. Section 6 shows some experimental results. Finally, in Sect. 7 we summarize the main contributions of our work.

2 Previous works

The first generation of voxelization algorithms were extensions of different 2D scan-conversion techniques to 3D [19, 20]. In general these solutions are designated as binary voxelization algorithms, because the result is a simple classification of the voxels from voxel space as selected/unselected. Although several of these algorithms guarantee the properties of separability and minimality of the final voxelization [1, 15], they suffer from aliasing problems. This motivated a new generation of algorithms focusing on the quality and accuracy of the final voxelization. In order to do it, several filtering [29, 31] or distance field techniques [13, 16] were used. In contrast to binary approaches, these algorithms assign a density value to each voxel that reflects its degree of occupancy by the object.

So far, the cited methods only perform a boundary voxelization of the solid. The voxelization of the interior of the solid is usually not addressed because it is considered computationally expensive or unnecessary for certain applications. However, at least two solutions are available: the early work of Lee and Requicha [22] and more recently, the algorithm proposed by Rueda et al. [27].

In the last years graphics hardware have been exploited to compute fast voxelizations of solids. In contrast to the conventional approaches, most of these hardware accelerated algorithms compute the voxelization of the entire solid. Fang and Chen [7] proposed a method that works moving a cutting plane, called Z-plane, parallel to the projection plane, with a constant step size in a front-to-back order. Initially, the viewport and the step size are set to the dimensions of the voxel space. The algorithm defines the current orthogonal viewing volume as the thin space between two consecutive Z-planes, and renders all the surface primitives using standard OpenGL rendering procedures. The clipping mechanism of the graphics engine ensures that only the parts of the surfaces within the viewing volume are displayed, defining a slice of the voxelization space. Finally the information of each slice is copied from the framebuffer to a 3D texture where the voxelization is stored. However, this method may have problems when the model to voxelize contains one or more faces perpendicular to the *near* and *far* planes of the viewing volume. The OpenGL driver does not display these faces, leading to incorrect results with cracks. Fang and Cheng report other special cases in which their voxelization algorithm fails. These special cases have a difficult solution that may depend on the specific polygon rasterization and clipping procedures of each vendor's OpenGL implementation.

Another simple and fast hardware accelerated voxelization method is that of Karabassi et al. [17]. This method uses six Z-buffers that are computed by rendering the solid twice per axis in opposite directions using orthographic projection. The information in the six Z-buffers is combined in main memory to construct the voxelization: a voxel is inside the solid if it is inside the Z-buffer values for all three pairs. Unfortunately this method can only be applied to a restricted subset of closed solids. Although a later improvement of this algorithm [25] can handle a higher range of solids, it cannot be considered a general voxelization method.

Li et al. [23] have recently included a brief description of a new hardware accelerated voxelization approach in the presentation of their flow simulation method. It is based on the technique known as "depth peeling," using the stencil buffer to compute a list of layers of voxels in each axis direction. The algorithm requires rendering the geometry once per layer in each axis direction, and therefore is quite efficient, as a typical solid can usually be decomposed into a few layers. However, in contrast to previous hardware accelerated approaches, this is a boundary-only voxelization method.

3 Voxelization algorithm

The theoretical basis of our voxelization algorithm is the point-in-tetrahedron inclusion test of Feito et al. [9]. Given an arbitrary origin point O and a polyhedron G defined by the triangular faces $f_0, f_1, \ldots, f_n$, then let $S = \{T_1, T_2, \ldots, T_n\}$ be a covering of G with 3D-simplexes (tetrahedra) T_i defined by O and the triangular face f_i. Then an arbitrary point P is inside polyhedron G if:

$$\sum_i \text{sign}(T_i) \cdot \text{incl}(T_i, P) > 0, \tag{1}$$

where $\text{incl}(T_i, P) = 1$ when $P \in T_i$, and 0 otherwise. On the other hand $\text{sign}(T_i) = +1$ when the vertices of the triangular faces of the tetrahedron T_i follow a counterclockwise ordering, -1 when they follow a clockwise ordering, and 0 when the tetrahedron is degenerated. The following lemma expresses the same idea in a different and more useful way:

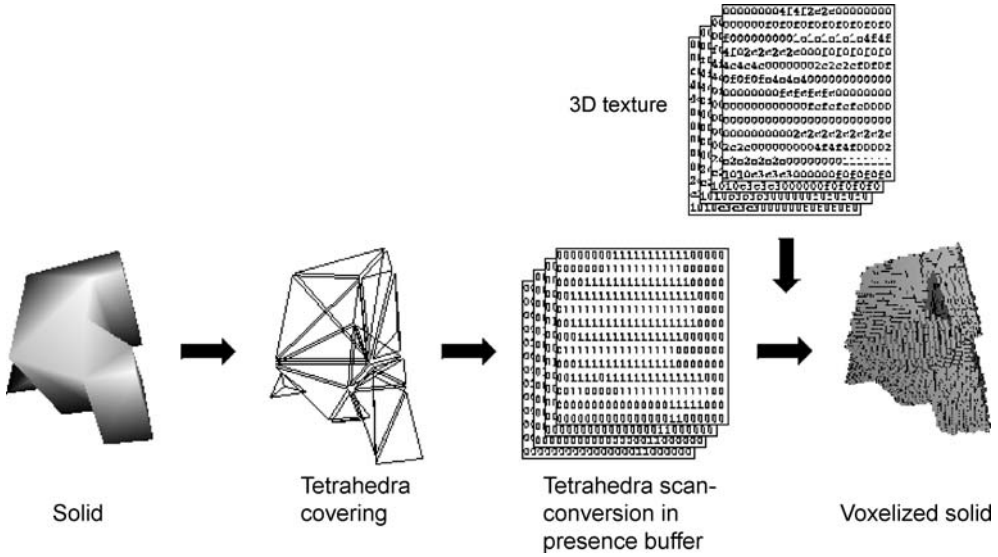

Fig. 1. Solid voxelization based on tetrahedra coverings

Lemma 1. *Let G be a polyhedron, and O an arbitrary origin point. Let $S = \{T_1, T_2, \ldots, T_n\}$ be the covering of G with tetrahedra defined by O and each triangular face of G. A point P inside G is covered by an odd number of tetrahedra from S.*

Proof. The Jordan curve theorem ensures that a ray starting at O and touching point P intersects an odd number of triangular faces of G after P. These faces generate an odd number of tetrahedra in S covering the point P. □

The voxelization algorithm for polyhedra consisting of triangular faces is outlined below. This approach is also valid for polyhedra with polygonal faces if these are previously tessellated or covered with a fan of triangles with a common origin point [27].

1. Compute the centroid of the polyhedron. Set this point as origin O.
2. Take a triangular face $\triangle ABC$ of the polyhedron and construct the tetrahedron $\triangle ABCO$.
3. Scan-convert the tetrahedron $\triangle ABCO$ in the *3D presence buffer*.
4. Return to Step 2 until all the faces of the polyhedron have been processed.
5. The final state of the 3D presence buffer represents the voxelization of the polyhedron.

The presence buffer is a 3D array of *presence values*, with the same dimension as the voxel space. Each voxel has an associated presence value, which can be represented with a single bit. A value 1 in its presence value indicates that the voxel belongs to the solid whereas a 0 value indicates the opposite. The scan-conversion of a tetrahedron in the presence buffer is done by flipping all the presence values covered by it. Once every tetrahedron has been scan-converted, Lemma 1 ensures a presence value 1 only in those voxels that approximate the polyhedron. Then the voxelization stored in the presence buffer can be directly used for any purpose, encoded by an efficient spatial data structure like an octree, or transferred to a 3D display framebuffer in order to visualize it. Additionally, some color information can be applied [5], which can be generated by a volumetric function or interpolated from a 3D map or sampled data (see Fig. 1).

The center of mass of the polyhedron is the best choice for the origin of the tetrahedra because the average size of the tetrahedra is smaller, which implies a lower total amount of voxels to be touched. This has been confirmed by our experiments with different origin positions. Translating the center of mass to the origin of coordinates also simplifies many computations during the tetrahedra scan-conversion.

The described algorithm is simple, robust and flexible: it can handle any kind of polyhedron, including non-convex, self-intersecting or holed, as its underlying principle is the Jordan curve theorem.

4 3D scan-conversion of tetrahedra

As it has just been shown, the most important step in the voxelization algorithm is the scan-conversion of a tetrahedron in the presence buffer. For this purpose, we propose an approach based on the scan-conversion of successive slices of the tetrahedron, similar to the scanline algorithm for polygons [12]. Let $\triangle ABCD$ be an arbitrary tetrahedron, as depicted in Fig. 2. The method is given by four steps.

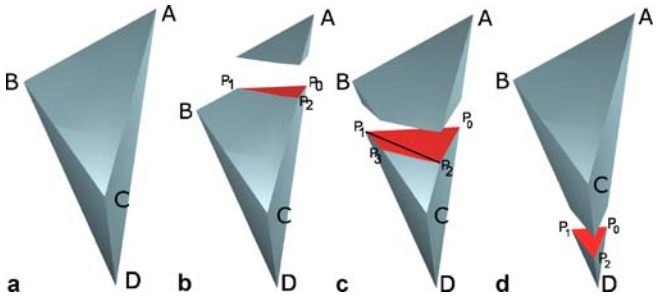

Fig. 2. Slicing a tetrahedron

Table 1. Edges required for point interpolation depending on the sweep plane relative position

Point	$A_y > y_s \geq B_y$	$B_y > y_s \geq C_y$	$C_y > y_s \geq D_y$
P_0	$\overline{AD}$	$\overline{AD}$	$\overline{AD}$
P_1	$\overline{AB}$	$\overline{BD}$	$\overline{BD}$
P_2	$\overline{AC}$	$\overline{CD}$	$\overline{CD}$
P_3	–	$\overline{BC}$	–

1. Choose a slicing direction. We will assume that slicing is done moving a plane along the y axis. Sort the vertices of the tetrahedron by their y coordinate. Let A be the vertex with higher y coordinate, B the next, and so on with C and D (see Fig. 2a). Sweeping starts at $y_s = A_y$ and finishes at $y_s = D_y$.
2. Compute the intersections of the edges of the tetrahedron with the current sweep plane. We denote these points P_0, P_1, P_2, P_3, as shown in Fig. 2b. These intersections can be computed by a simple linear interpolation or applying a faster incremental approach. Table 1 shows the edges that must be used to compute these points, depending on the value of y_s. Notice that point P_3 only appears in the interval $B_y > y_s \geq C_y$, as can be seen in Fig. 2c.
3. Voxelize the slice y_s of the tetrahedron. This can be done by simply scan-converting the triangle $\triangle P_0 P_1 P_2$, shown in Fig. 2b. In the interval $B_y > y_s \geq C_y$, a second triangle $\triangle P_3 P_2 P_1$ must also be scan-converted (see Fig. 2c). During this operation, the presence values of all the voxels x, y_s, z covered by the triangles must be flipped.
4. Decrement y_s and repeat steps 2 and 3 until $y_s = D_y$.

5 Algorithm implementations

The algorithm described in Sect. 3 can be easily implemented using simple data structures [27]. The presence buffer can be represented in main memory by a 3D array of bits, and each tetrahedron can be scan-converted to this array by using the method described in the previous section.

During the execution of a CPU-based implementation of the algorithm, most time is spent in the scan-conversion of 2D triangles. We can take advantage of graphics hardware to perform this task more efficiently. In the rest of this section two hardware accelerated implementations of this algorithm are described in detail.

5.1 Hardware accelerated implementation

A hardware accelerated implementation using OpenGL primitives is outlined below. Instead of computing a voxelization by iterating over the tetrahedra set, the scan-conversion of the tetrahedra is done in parallel for each 2D slice of the presence buffer:

1. Create a presence buffer with the dimensions of the voxel space.
2. Initialize y_s to the dimension of the voxel space.
3. Clear the presence buffer and set the logical pixel operation to GL_XOR. Initialize the drawing color to (1, 1, 1).
4. Compute the list of tetrahedra that intersect slice y_s, that is, those that verify $A_y > y_s \geq D_y$.
5. Compute the intersection points P_0, P_1, P_2, P_3 of each tetrahedron in the list with the current slice. Draw the triangles $\triangle P_0 P_1 P_2$ and $\triangle P_3 P_2 P_1$.
6. Transfer the current slice to a 3D texture or a data structure in CPU main memory.
7. Decrement y_s and return to Step 3 until $y_s = 0$.

Transferring each slice to a 3D texture is interesting for two reasons: it is more efficient than copying the voxels back to the CPU and also allows a direct application of volume rendering methods [3]. As soon as required, it can be partially or fully retrieved to main memory in order to perform any additional processing. The current generation of GPUs with PCI-express technology has dramatically improved the efficiency of this operation. The first alternative can be implemented by rendering to a P-buffer and copying to 3D texture slice or using the framebuffer object extension to render to a texture object in an efficient and straightforward way.

The main drawback of the previous approach is that a large set of triangles must be computed in main memory and sent to the graphics hardware in each slice, causing a significant traffic overhead. These triangles change from one slice to the next, preventing the use of vertex arrays or display lists.

5.2 GPU-based implementation

The programming capabilities of modern GPUs have been extensively exploited in the last years to develop new efficient solutions to well-known problems [11, 26]. GPU

programming can reduce CPU-GPU traffic because the basic geometry can be sent only once and then arbitrarily transformed in each frame by a vertex program. Therefore, the problems of the hardware accelerated implementation described in the previous section can be overcome if the set of triangles is updated from one slice to the next in the GPU.

The vertex program computes the position of the points P_0, P_1, P_2, P_3 for a given tetrahedron and the current slice, applying linear interpolation on the corresponding edges, as described in Table 1. The coordinates of the tetrahedron vertices A, B, C, D are passed to the program as varying parameters, as well as the point index (0-3), and an identifier of the triangle the point belongs to (0 for $\triangle P_0P_1P_2$ and 1 for $\triangle P_3P_2P_1$). On the other hand the current slice and the model-view projection matrix are passed as uniform parameters. The full Cg code [10] of the vertex program is shown in Listing 1.

Listing 1. Cg source of the vertex program for tetrahedra voxelization

```
#define IS_MAIN_TRIANGLE (vertexData.x == 0)
#define VERTEX_INDEX (vertexData.y)
#define IN_INTERVAL(v, a, b) (a > v && v >= b)
#define INTERP(A, B, slice) (lerp (A, B, (slice - A[1])
    / (B[1] - A[1])))

void tetraVoxelization(
    // x->triangle type (0-1) y->vertex index (0-4)
    in int2 vertexData: POSITION,
    in float3 tVertexA, // Vertex A
    in float3 tVertexB, // Vertex B
    in float3 tVertexC, // Vertex C
    in float3 tVertexD, // Vertex D
    out float4 resultVertex: POSITION, // Computed vertex
    out float4 resultColor: COLOR, // Vertex color
    uniform float slice, // Current slice
    uniform float4x4 modelViewProjectionMatrix)
    {
        // Initialize vertex position outside view volume
        float4 pos = float4(10000, 10000, 0, 1);

    // Process only if the slice intersects the tetrahedron
    if (IN_INTERVAL(slice, tVertexA[1], tVertexD[1]))
        {
            // Process only if:
            // -The triangle is P0P1P2 or
            // -The triangle is P2P1P3 and the slice intersects
            // its active interval (B.y, C.y)
            if (IS_MAIN_TRIANGLE ||
                IN_INTERVAL(slice, tVertexB[1], tVertexC[1]))
            {
                if (VERTEX_INDEX == 0) // p0
                pos.xy = INTERP(tVertexA,tVertexD,slice).xz;
                else
                if (VERTEX_INDEX == 1) // p1
                    pos.xy = (slice >= tVertexB[1] ?
                        INTERP(tVertexA, tVertexB, slice).xz:
                        INTERP(tVertexB, tVertexD, slice).xz);
                else
                if (VERTEX_INDEX == 2) // p2
                    pos.xy = (slice >= tVertexC[1] ?
                        INTERP(tVertexA, tVertexC, slice).xz:
                        INTERP(tVertexC, tVertexD, slice).xz);
                else
                if (VERTEX_INDEX == 3) // p3
                    pos.xy = INTERP(tVertexB,tVertexC,slice).xz;
            }
        }

    resultVertex = mul(modelViewProjectionMatrix, pos);
    resultColor = float4(1, 1, 1, 1);
}
```

If the full tetrahedra set is sent to the graphics pipeline in each slice, the vertex program must check that the current slice intersects the tetrahedron. If the result is negative, all its vertices must be culled. The three vertices of the triangle $\triangle P_3P_2P_1$ are also culled when the slice is outside the interval $B_y > y_s \geq C_y$. The third version of the voxelization algorithm, using GPUs, works as follows:

1. Create a presence buffer with the dimensions of the voxel space.
2. Initialize y_s to the dimension of the voxel space. Setup the *modelViewProjectMatrix*.
3. Compile a display list with two triangles, $\triangle P_0P_1P_2$ and $\triangle P_3P_2P_1$ per tetrahedron, setting the tetrahedron vertices A, B, C, D as the varying parameters of the vertex program (see Listing 2). Alternatively, an array of vertices and four parameter arrays can be used instead.
4. Clear the presence buffer and set the logical pixel operation to GL_XOR.
5. Set the uniform parameter *slice* to y_s and call the display list. In the vertex array implementation, the set of tetrahedra that verify $A_y > y_s \geq D_y$ is sent to the GPU.
6. Copy the current slice from the framebuffer to a 3D texture or CPU main memory.
7. Decrement y_s and return to Step 4 until $y_s = 0$.

Listing 2. Vertex program setup for tetrahedron t

```
// The tetrahedron vertices have been previously sorted
// by y-coordinate

// Send tetrahedron vertices A, B, C, D to the
// vertex program
cgGLSetParameter3fv(tVertexA, t[0]);
cgGLSetParameter3fv(tVertexB, t[1]);
cgGLSetParameter3fv(tVertexC, t[2]);
cgGLSetParameter3fv(tVertexD, t[3]);

// Send triangle vertices

// Main triangle
glVertex2i(0, 0); // P0
glVertex2i(0, 1); // P1
glVertex2i(0, 2); // P2

// Secondary triangle
glVertex2i(1, 3); // P3
glVertex2i(1, 2); // P2
glVertex2i(1, 1); // P1
```

In this implementation, the solid voxelization is almost entirely solved by the GPU, saving a very considerable amount of time and space in the CPU. Another advan-

tage of this approach is that once the display list has been compiled or the vertex arrays have been set up, several voxelizations of the entire solid or different parts of it, at different resolutions can be efficiently computed. An interesting advantage of an implementation based on vertex arrays is that the minimal set of tetrahedra that intersects each slice can be computed and sent to the GPU. In contrast, the implementation based on display lists requires the full set of tetrahedra to be compiled in a previous stage, and sent to the GPU in each slice scan-conversion. In this case the vertex program must be responsible for culling the tetrahedra that do not intersect the current slice as we explained before. Figure 3 shows two voxelizations computed by this approach, using different voxel resolutions.

Both hardware accelerated methods described in this section allow the use of a fragment program to compute arbitrary properties per voxel in a very simple and efficient way. Example shown in Fig. 4 illustrates the results of extending our voxelization algorithm with a 3D Perlin noise generator implemented as a fragment program.

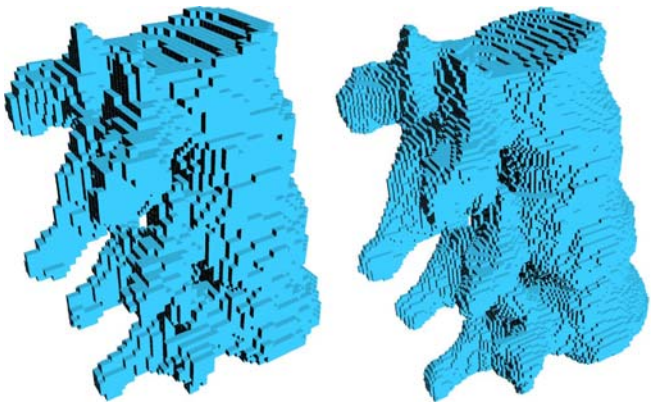

Fig. 3. Examples of voxelizations at different resolutions

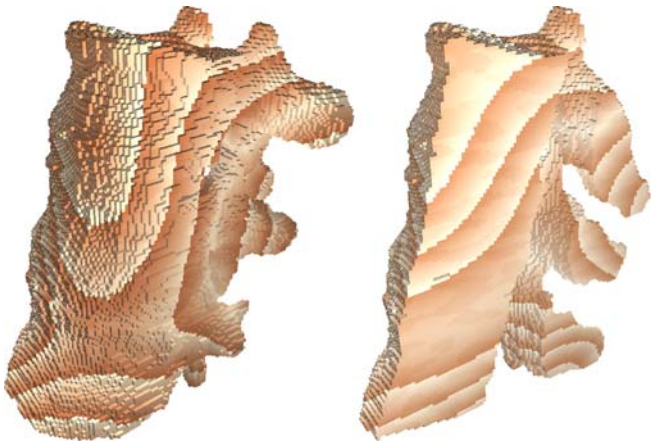

Fig. 4. Use of a 3D Perlin noise generator to set a color per voxel

6 Experimental results

We have compared the three implementations of our algorithm (CPU-based, hardware accelerated, and GPU-based) against the hardware accelerated approaches of Passalis et al. [25] and Fang and Chen [7]. The timings of the hardware accelerated, GPU-based and Fang and Chen approaches include the transfer to texture 3D in GPU memory. The extra time required to transfer the voxelization to the CPU is shown in the last row. All the algorithms have been implemented in C++, using the same compiler and optimizations. The tests were run on a Pentium IV 3.3Ghz. with a GeForce 7800GTX on Windows XP, using the models shown in Fig. 5.

Table 2 shows how the GPU-based implementation of our algorithm outperforms the approaches of Passalis et al. and Fang and Chen, running between 2 and 10 times faster, depending on the resolutions and complexity of the model. Surprisingly, the hardware accelerated implementation of the algorithm does not show better results than the CPU-based implementation. As we pointed out in the previous section, the weakest aspect of this implementation is the need to compute and send a different set of triangles from the CPU to the GPU for each slice. This causes an important penalty that is not compensated for the fast hardware accelerated triangle rasterization.

Although the method of Passalis et al. is hardware accelerated, the voxelization is really computed in the CPU, which makes it less competitive than full hardware accelerated methods like our GPU-based implementation or the approach of Fang and Chen. However, it has a nice feature: its processing time is almost independent on the complexity of the model. The performance of the method of Fang and Chen scales reasonably well with the complexity of the model and the resolution of the voxel space but in general runs slower than our GPU-based approach.

7 Conclusions

In this work we have presented a simple, robust and efficient method for the voxelization of polyhedra which can be easily implemented using common graphics hardware or GPUs with vertex program support. It is based on a decomposition of the polyhedron into a set of tetrahedra with a common origin (usually its centroid) and its scan-conversion into a special 3D presence buffer. The hardware accelerated approach uses graphics hardware to efficiently scan-convert the intersections of the set of tetrahedra with a given slice of the presence buffer. The GPU-based approach goes beyond, computing these intersections with a simple vertex program, and therefore minimizing the intervention of the CPU in the voxelization process. This GPU-based implementation shows a better performance than other commonly used hardware accelerated approaches.

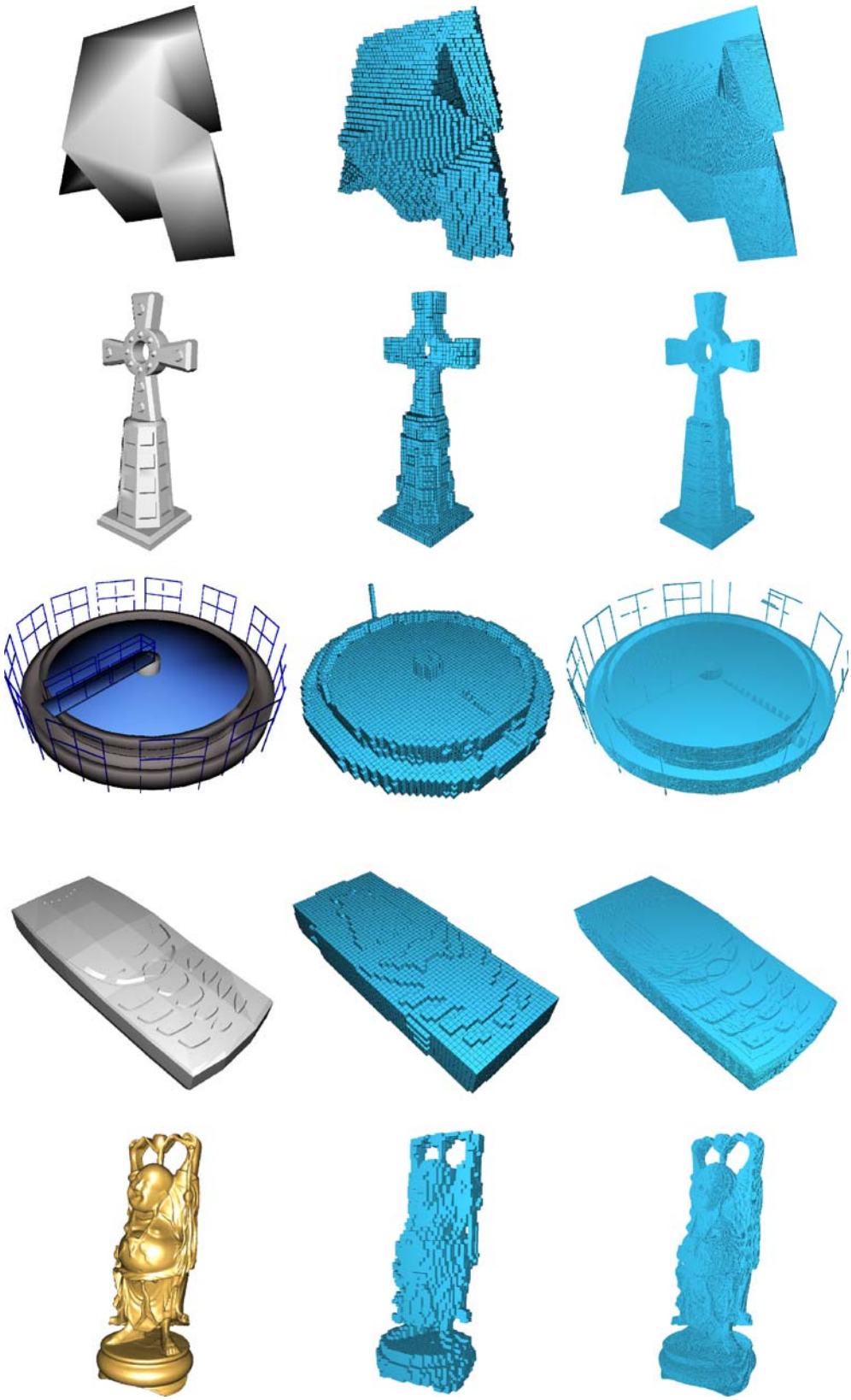

Fig. 5. Models used during the experiments. Results after voxelization at 64 and 512 resolutions are also shown

Table 2. Voxelization times (in secs.) of the three implementations of our algorithm: CPU-based (New), hardware accelerated (NewH) and GPU-based (NewG) against the algorithms of Passalis et al., and Fang & Chen, at 64^3, 128^3, 256^3, 512^3 voxel resolutions. The last rows show the extra time required to transfer the voxelization back to CPU

Model	Triang.	64^3 New	NewH	NewG	Passalis	Fang	128^3 New	NewH	NewG	Passalis	Fang
Simple	42	0.0021	0.0202	0.0038	0.0283	0.0040	0.0012	0.0233	0.0074	0.2097	0.0080
Cross	2366	0.0136	0.0184	0.0133	0.0282	0.0038	0.0344	0.0542	0.0224	0.2174	0.0083
Depot	10591	0.0659	0.0498	0.0143	0.0514	0.0889	0.0694	0.0942	0.0228	0.2830	0.1659
Mobile	25946	0.0401	0.0605	0.0141	0.0398	0.0170	0.0741	0.1274	0.0256	0.2256	0.0344
Buddha	100000	0.4150	0.2636	0.0621	0.0659	0.0452	0.8224	0.5622	0.1097	0.2617	0.1167
T. CPU		–	0.0055	0.0055	–	0.0055	–	0.0239	0.0293	–	0.0293
Model	Triang.	256^3 New	NewH	NewG	Passalis	Fang	512^3 New	NewH	NewG	Passalis	Fang
Simple	42	0.0778	0.1468	0.0153	1.6980	0.0271	0.0521	1.0103	0.0357	14.0875	0.1461
Cross	2366	0.1394	0.1915	0.0408	1.6711	0.0286	0.4800	1.1633	0.0876	14.0419	0.1498
Depot	10591	0.3500	0.2809	0.0414	1.8743	0.3428	1.6407	1.3164	0.0946	14.1317	0.6959
Mobile	25946	0.1805	0.3450	0.0522	1.7322	0.1074	0.6434	1.4803	0.0917	14.3993	0.3071
Buddha	100000	1.6979	1.3069	0.1296	1.8952	0.2322	4.1155	1.9352	0.2361	14.2695	0.6104
T. CPU		–	0.1406	0.1406	–	0.1406	–	0.9363	0.9363	–	0.9363

We believe that geometry instancing [4] would improve our GPU-based implementation. Using this feature, only one triangle and a texture with the instancing information have to be sent to the GPU in each frame. The geometry processor uses the instancing information to generate as many copies of the triangle as necessary. The advantages of this approach are evident: it avoids constructing and sending large triangle display lists or vertex arrays to the GPU. However, an OpenGL extension to handle this feature (GL_EXT_draw_ instanced) has been unavailable until very recently, and it is currently only supported by the NVIDIA GeForce 8 series GPUs.

References

1. Andres, E., Nehlig, P., Francon, J.: Tunnel-free supercover 3D polygons and polyhedra. Comput. Graph. Forum **16**, C3–C13 (1997)
2. Blundell, B., Schwarz, A.: Volumetric Three-Dimensional Display Systems. Wiley, New York (2000)
3. Cabral, B., Cam, N., Foran, J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In: Proceedings of the 1994 IEEE Symposium on Volume Visualization, pp. 91–98 (1994)
4. Carucci, F.: Inside Geometry Instancing. In: GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation, pp. 47–68. Addison-Wesley, Boston (2004)
5. Chen, M., Tucker, J.V.: Constructive volume geometry. Comput. Graph. Forum **19**, 281–293 (2000)
6. Ebert, D., Bedwell, E., Maher, S., Smoliar, L., Downing, E.: Realizing 3D visualization using crossed-beam volumetric displays. Commun. ACM **42**, 101–107 (1999)
7. Fang, S., Chen, H.: Hardware accelerated voxelization. Comput. Graph. **24**, 433–442 (2000)
8. Feito, F., Torres, J.C.: Orientation, simplicity and inclusion test for planar polygons. Comput. Graph. **19**, 596–600 (1995)
9. Feito, F., Torres, J.C.: Inclusion test in general polyhedra. Comput. Graph. **21**, 23–30 (1997)
10. Fernando, R., Kilgard, M.J.: The Cg Tutorial: The Definitive Guide to Programmable Real-time Graphics. Addison-Wesley, Boston (2003)
11. Fernando, R.: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley, Boston (2004)
12. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.H.: Introduction to Computer Graphics. Addison-Wesley, Boston (1994)
13. Frisken, F.S.: Using distance maps for accurate surface representation in sampled volumes. In: IEEE Symposium on Volume Visualization, pp. 23–30 (1998)
14. Haumont, D., Warzie, N.: Complete polygonal scene voxelization. J. Graph. Tools **7**, 27–41 (2002)
15. Huang, J., Yagel, R., Filippov, V., Kurzion, Y.: An accurate method to voxelize polygonal meshes. In: IEEE Symposium on Volume Visualization, pp. 119–126 (1998)
16. Jones, M.W.: The production of volume data from triangular meshes using voxelisation. Comput. Graph. Forum **15**, 311–318 (1996)
17. Karabassi, E., Papaioannou, G., Theoharis, T.: A fast depth-buffer-based voxelization algorithm. ACM J. Graph. Tools **4**, 114–124 (2002)
18. Kaufman, A., Shimony, E.: 3D scan-conversion algorithms for voxel-based graphics. In: Proceedings ACM Workshop on Interactive Graphics, pp. 45–76 (1986)
19. Kaufman, A.: Efficient algorithms for 3D scan-conversion of parametric curves, surfaces and volumes. Comput. Graph. **21**, 171–179 (1987)
20. Kaufman, A.: Efficient algorithms for scan-conversing 3D polygons. Comput. Graph. **12**, 213–219 (1988)
21. Kaufman, A., Cohen, D., Yagel, R.: Volume graphics. IEEE Comput. **26**, 51–64 (1993)
22. Lee, Y. T, Requicha, A.: Algorithms for computing the volume and other integral properties of solids. Commun. ACM **25**, 635–650 (1982)
23. Li, W., Fan, Z., Wei, X., Kaufman, A.: Flow simulation with complex boundaries. In: GPU Gems 2: Programming Techniques for

High-Performance Graphics and General Purpose Computation, pp. 747–763. Addison-Wesley, Boston (2005)
24. Pastoor, S., Kiesewetter, R.: 3-D displays: A review of current technologies. DISPLAYS **17**, 100–110 (1997)
25. Passalis, G., Kakadiaris, I.A., Theoharis, T.: Efficient Hardware Voxelization. In: Proceedings of the Computer Graphics International, pp. 374–377 (2004)
26. Pharr, M.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, Boston (2005)
27. Rueda, A.J., Segura, R., Feito, F.R., Ogayar, C.: Voxelization of solids using simplicial coverings. In: Proceedings of WSCG'2004, pp. 227–234 (2005)
28. Rueda, A.J., Segura, R., Ruiz de Miras, J., Feito, F.R.: Rasterizing complex polygons without tesselations. Graph. Models **26**, 805–814 (2004)
29. Šrámek, M., Kaufman, A.: A. Alias-free voxelization of geometric objects. IEEE Trans. Visual. Comput. Graph. **5**, 251–267 (1999)
30. Šrámek, M., Kaufman, A.: VXT: a C++ class library for object voxelization. In: Proceedings of the International Workshop on Volume Graphics, pp. 119–134 (1999)
31. Wang, S.W., Kaufman, A.: Volume-sampled 3D modelling. IEEE Comput. Graph. Appl. **14**, 26–32 (1994)
32. Woo, M., Nedider, J., Davis, T., Shreiner, D.: The OpenGL Programming Guide, 3rd. edn. Addison-Wesley, Boston (1999)




C.J. OGAYAR is a researcher in the Department of Computer Science at the University of Jaén (Spain). He received his B.Sc. in Computer Science and his Ph.D. in Computer Science from the University of Granada. He has wide software programming experience in computer science, including solid modeling, computational geometry, real-time graphics and GPU programming. He has also been working as a freelancer on web development, database management and user interfaces. His research interests includes virtual reality, GIS and game engine design.

A.J. RUEDA is lecturer in the Department of Computer Science, University of Jaén (Spain). He received his B.Sc. in Computer Science from the University of Granada and his Ph.D. in Computer Science from the University of Malaga. During his Ph.D. studies, he developed the layer-based decomposition of objects in 2D and 3D as well as several practical algorithms based on this representation. His research interests include topics like geometric algorithms, spatial decompositions, spatial data structures, and lately, acceleration of geometric algorithms on programmable GPUs.

R.J. SEGURA is a lecturer in the Department of Computer Science at the University of Jaén (Spain). He received a B.Sc. in Computer Science at the University of Granada in 1994, where he received his Ph.D. in Computer Science in 2001. He has been working on several topics in computer graphics, including solid modeling, computational geometry, virtual reality and GPGPU. He is a scientific advisor of the Computer Graphics Laboratory in the Research Technical Service of the University of Jaén, specialized in 3D scanning and rapid prototyping.

F.R. FEITO is full professor in the Department of Computer Science at the University of Jaén (Spain), received his B.Sc. in Mathematics at the University Complutense of Madrid and his Ph.D. in Computer Science from the University of Granada (Spain). He was the head of the department of Computer Science of the University of Jaén from 1993 to 1997, Vice Chancellor in charge of Studies and Quality from 1997 to 1999, and Vice Chancellor in charge of Research and International Affairs from 1999 to 2002 at the same university. His research interests include formal methods for computer graphics, geometric modeling, computational geometry and geographical information sciences. Currently, he is teaching at the Escuela Politécnica Superior of Jaén, and is the head of the department and the Graphics and Geomatics Research Group.