Fast and Simple Hardware Accelerated Voxelizations Using Simplicial Coverings ⋆

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

Departamento de Informática

Escuela Politécnica Superior

Universidad de Jaén

Campus Las Lagunillas, Edif. A3

23071 Jaén (Spain)

1 Introduction

Voxelization of solids is concerned with converting geometric objects from their continuous geometric representation into a set of voxels that best approximates the continuous object. Voxel representation of solids has applications in Solid Modeling, Volume Graphics and Physical Simulation. It has extensively used for rendering objects which are difficult to represent by traditional surface representations, like clouds, fire, smoke or terrain models [Kau93]. A new

⋆ This work has been partially granted by the Ministry of Science and Technology of Spain and the European Union by means of the ERDF funds, under the research projects TIC2001-2099-C03-03 and TIN2004-06326-C03-03.

Email addresses: ajrueda@ujaen.es (A. J. Rueda), cogayar@ujaen.es (C. Ogáyar), rsegura@ujaen.es (R. J. Segura), ffeito@ujaen.es (F. R. Feito).

application of voxelization techniques is the 3D scan-conversion of models for the emerging 3D raster displays [BS00,PK00,EBM*99]. This new technology represent 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 rather 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 method described in this paper is an extension of a 2D rasterization algorithm for polygons proposed by the same authors in previous works [RSR*04]. Both methods are conceptually very simple, and avoid the use of tessellations, complex data structures or previous sortings of edges and faces. This voxelization algorithm was already presented in [RSF04], where a conventional non-hardware accelerated implementation was described.

2 Previous works

The simplest approach to compute a voxelization consists of testing the inclusion of the center of every voxel in the solid. This inclusion test can be solved by applying a simple crossing count or a more sophisticated method. The main drawback of this method is its poor performance: the number of inclusion tests can be extremely high even when working with moderated voxel resolutions. Another straightforward way to voxelize a solid is based on the extension of the 2D scanline algorithm to 3D. In this case a plane is used to sweep the solid following an axis aligned direction (e. g. y axis). For each slice of voxels, a set of rays is cast (e. g. one per row, following x direction) to compute a list

of intersections. Then, the list of intersections is used in a scanline algorithm way to scan-convert the slice of voxels.

Huang [HYF*98] describes a method for voxelizing planar objects which provides topological conformity through geometric measurements. This method eliminates common voxelization artifacts at edges and vertices. It is based on 3D discrete spaces and separability, that is: in order to voxelize a plane (or a polygon) two parallel planes are built, so the plane to be voxelized lies between then (all planes are parallel). This method works fine, but it does not allow the voxelization of the inner part of the solid.

Sramek [SK99a] introduces the Voxelization Model (V-model), which is an alias-free voxelization method for geometric objects. The V-model of an object represents it in a three-dimensional continuous space by a trivariate density function. This function is sampled during the voxelization and the resulting values are stored in a volume buffer. Several filtering and interpolation methods can be applied to the surface density profile. This method allows an alias-free discrete representation of an object, but it does not take care of the interior of the solid.

Fang [FC00] proposes a method based on the use of the graphics hardware. This algorithm proceeds in a similar way to the 3D scanline algorithm, moving a cutting plane, called Z-plane, parallel to the projection plane, with a constant step size in a front-to-back order. The thin space between two adjacent Z-planes within the volume space defines a slice. For each new Z-plane, the algorithm defines the new slice as the current orthogonal viewing volume, and renders all the surface primitives using standard OpenGL rendering procedures. Since the boundary planes of the viewing volume are used as clipping

planes in OpenGL, the clipping mechanism of the graphics engine will ensure that only the parts of the surfaces within the slice are displayed. The resulting frame buffer image from the display of this slice will become one slice of the resulting volume.

The method presented by Haumont [HW02] converts complete polygonal scenes into voxelized representations. It stores the status (in/out) of the volumetric space areas in the cells of an octree. First, the algorithm looks for a point in the scene for which the status can be determined; second, the status is propagated to the surrounding visible cells. This two steps are repeated until the status of all the cells in the octree is determined. The advantage of this method is its robustness, it can successfully handle issues like cracks, holes, interpenetrating meshes and overlapping geometries. The drawbacks of this technique are its noticeable slow performance and high memory requirements.

Jones [Jon96] presents a method which voxelizes a model using a point to triangle distance function. With this approach, each voxel on the grid is treated as a point, and its distance to each triangle of the model is calculated. There are several optimizations to enhance the performance, but in general it is a slow method. Like other approaches, it does not take care of the interior of the solid.

3 Voxelization algorithm

The theoretical basis of our voxelization algorithm is the point-in-tetrahedron inclusion test of Feito et al. [FT97]. Given an arbitrary origin point O and a polyhedron G defined by the triangular faces $f_0, f_1, \ldots, f_n$, then we define

$S = \{T_1, T_2, \ldots, T_n\}$ as 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

$$\sum_i sign(T_i) \cdot incl(T_i, P) > 0$$

where $incl(T_i, P) = 1$ when $P \in T_i$, and 0 otherwise. On the other hand $sign(T_i) = +1$ when the vertices of the triangular faces of the tetrahedron T_i follow a counter-clockwise ordering, 1 when they follow a clockwise ordering, and 0 when the tetrahedron is degenerated. This test can be reformulated as shows the following lemma:

Lemma 1 *Let be G a polyhedron, and O an arbitrary origin point. Let be $S = \{T_1, T_2, \ldots, T_n\}$ 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 edges of G after P. These edges generate an odd number of triangles in S covering point P. □

The voxelization algorithm for polyhedra consisted of triangular faces is outlined below. Of course this approach is valid for polyhedra consisted of general polygonal faces if these are previously tessellated.

(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 dimensions that 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 implies flipping all the presence values covered by it. Once all the tetrahedron have been scan-converted, Lemma 1 ensures a presence value 1 only in those voxels that belong to the polygon. 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. In the last case, some additional color information must be applied, which can be generated by a volumetric function or interpolated from a 3D map of sampled data (see Figure 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. 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.

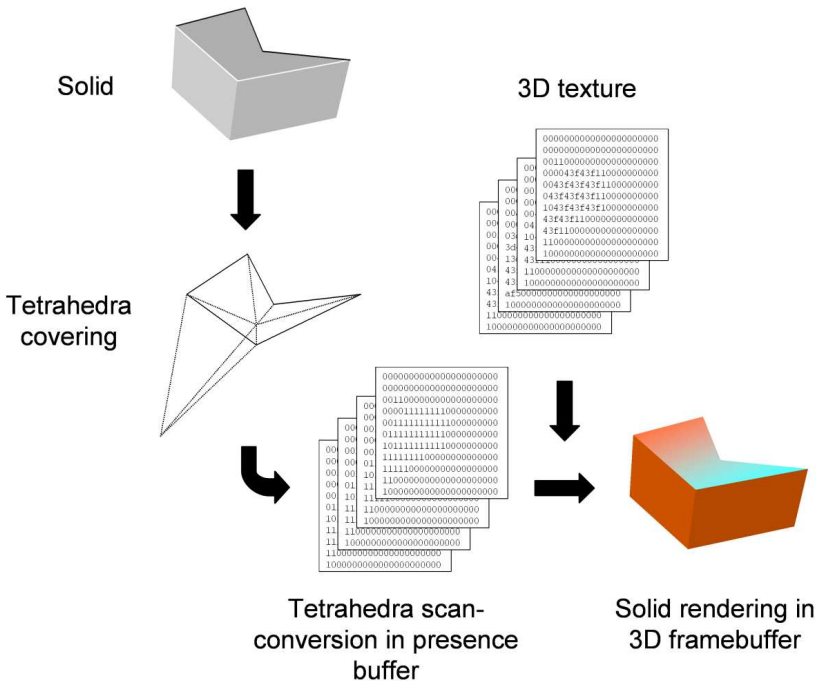


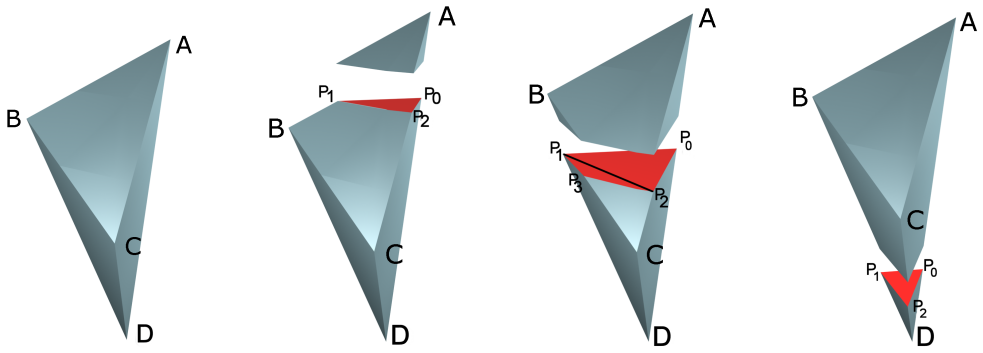
Fig. 1. Scan-conversion of solids on a 3D display.

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 tetrahedra in the presence-buffer. For this purpose, we propose a approach based on the scan-conversion of successive slices of the tetrahedron, similar to the scanline algorithm for polygons [FDF*94]. Let $\triangle ABCD$ be an arbitrary tetrahedron, as depicted in Figure 2. The method is given by four steps.

(1) Choose a slicing direction. We will assume slicing is done moving a plane along y axis. Sort the vertices of the tetrahedron by their y coordinate. We assume A is the vertex with higher y coordinate, B the next, and so on with C and D (see Figure 2). Sweeping starts at $y_s = A.y$ and finishes at $O.y$.

(2) Compute the intersections of the edges of the tetrahedron with the current



(a) (b) (c) (d)

Fig. 2. Slicing a tetrahedron

	$A_y > y_s \geq B_y$	$B_y > y_s \geq C_y$	$C_y > y_s \geq D_y$
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	-	$\overline{BC}$	-

Table 1

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

sweep plane. We denote these points P_0, P_1, P_2, P_3, as shown in Figure 2.b. This 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 depicted in Figure 2.

(3) Voxelize the slice y_s of the tetrahedron. This can be done by simply scan-converting the triangle $\triangle P_0P_1P_2$ using a simple scanline approach. In the interval $B_y > y_s \geq C_y$, an second triangle $\triangle P_3P_2P_1$ must also be scan-

converted. 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 $y_s = O_y$.

5 Conventional and hardware-accelerated implementations

The described algorithm can be easily implemented using simple data structures. The presence buffer can be represented in main memory by a 3D array of bits and each tetrahedron is scan-converted to this array by using the method described in the previous section. See [RSF04] for a in-depth description of this implementation and time comparisons with other approaches.

During the execution of a conventional 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. 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 p-buffer with the dimensions of the voxel space (i. e. 128x128).

(2) Initialize y_s to the dimension of the voxel space.

(3) Clear the p-buffer and set the logical pixel operation to GL_ XOR. Set 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 points intersection points P_0, P_1, P_2, P_3 for the current slice and each tetrahedron of the list. Draw the triangles $\triangle P_0P_1P_2$ and $\triangle P_3P_2P_1$.

(6) Transfer the current slice to a 3D texture or a data structure in 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 efficient and allows a direct application of a volume rendering methods. As soon as required, this information can be retrieved to main memory in order to perform any additional processing.

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 adapter in each slice, introducing a significant traffic overhead. These triangles change from one slice to the next, preventing the use of vertex arrays or display lists. This problem can be avoided if the set of triangles is updated from one slice to the next in the own adapter by using a vertex program.

The vertex program computes the position of the points P_0, P_1, P_2, P_3 for a given tetrahedron and a current slice, applying linear interpolation on the corresponding edges, as described in table 1. The coordinates of the tetrahedra vertices A, B, C, D are passed to the program as varying parameters, as well as the index of the point (0-3), necessary to compute the interpolation, and an identifier of the triangle that owns the vertex (1 for $\triangle P_0P_1P_2$ and 0 for $\triangle P_3P_2P_1$). On the other hand the size of the voxel, the coordinates of the corner of the minimal bounding cube of the object with minimal xyz coordinates, the current slice and the model-view projection matrix are passed as uniform parameters. The full Cg code of the vertex program is shown below:

```
struct VertexResult // Output data
{
```

```
  float4 position: POSITION;
  float4 color: COLOR0;
};

#define IN_MAIN_TRIANGLE(A) A.vertexInfo[0]
#define VERTEX_INDEX(A) A.vertexInfo[1]
#define INSIDE_INTERVAL (v,a,b) (a < v && v <= b)
#define INTERP(A,B,s) (lerp (A, B, (s - A[2]) / (B[2] - A[2])))
#define CULL_VERTEX(A) (A = float4 (-1, -1, -1, 1))

VertexResult main (
           int2 vertexData: POSITION, // x->triangle type, y->vertex index
           float3 tetraVertexA,       // Vertex A coordinates
           float3 tetraVertexB,       // Vertex B coordinates
           float3 tetraVertexC,       // Vertex C coordinates
           float3 tetraVertexD,       // Vertex D coordinates
           uniform float3 minBound,   // Min xyz corner of the bounding box
           uniform float voxelSize,
           uniform float slice,       // Current slice
           uniform float4x4 modelViewProjectionMatrix)
{
  VertexResult result;
  float4 vertexPos;
  // Convert the tetrahedron vertices from world to voxel space
  tetraVertexA = (tetraVertexA - minBound) / voxelSize;
  tetraVertexB = (tetraVertexB - minBound) / voxelSize;
  tetraVertexC = (tetraVertexC - minBound) / voxelSize;
  tetraVertexD = (tetraVertexD - minBound) / voxelSize;

  // If the full set of tetrahedra is going to be sent in each slice,
  // test here if the slice intersects the tetrahedron. If the test fails, cull vertex

  // If secondary triangle, only process when By > slice >= Cy
  if ( IN_MAIN_TRIANGLE (IN) ||
    INSIDE_INTERVAL (slice, IN.tetraVertexB[1],   IN.tetraVertexC[1]) )
  {
     if ( VERTEX_INDEX (IN) == 0 )    // Process p0
       vertexPos =
         float4 (INTERP (IN.tetraVertexA, IN.tetraVertexD, slice).xy, 0, 1);
     else
```

```
    if ( VERTEX_INDEX (IN) == 1 )    // Process p1
      vertexPos = float4 ((slice >= IN.tetraVertex2[1]) ?
        INTERP (IN.tetraVertexA, IN.tetraVertexB, slice).xy :
        INTERP (IN.tetraVertexB, IN.tetraVertexD, slice).xy, 0, 1);
    else
    if ( VERTEX_INDEX (IN) == 2 )   // Process p2
      vertexPos = float4 ((slice >= IN.tetraVertex3[1]) ?
        INTERP (IN.tetraVertexA, IN.tetraVertexC, slice).xy :
        INTERP (IN.tetraVertexC, IN.tetraVertexD, slice).xy, 0, 1);
    else
    if ( VERTEX_INDEX (IN) == 3 )   // Process p3
      vertexPos =
        float4 (INTERP (IN.tetraVertexB, IN.tetraVertexC, slice).xy, 0, 1);
  }
  else
    CULL_VERTEX (vertexPos);

  result.position = mul (modelViewProjectionMatrix, vertexPos);
  result.color = float4 (1, 1, 1, 1);
  return result;
}
```

If the full tetrahedra set is sent to the graphics pipeline in each slice, the vertex program must check that the current slice does intersect the tetrahedron. If the result is negative, the vertex must be culled. This can be avoided if some preprocessing is done, computing the relevant tetrahedra for each slice. 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 programmable GPUs, works as follows:

(1) Create a p-buffer with the dimensions of the voxel space.

(2) Initialize y_s to the dimension of the voxel space. Setup the *modelView-ProjectMatrix*, *voxelSize* and *minBound* uniform parameters.

(3) Compile a display list with two triangles, $\triangle P_0P_1P_2$ and $\triangle P_3P_2P_1$ per

tetrahedron, setting the varying parameters to the tetrahedron vertices A, B, C, D. This is illustrated in the code fragment below. Alternatively, an array of vertices and four parameter arrays can be used instead.

```
// Setup for tetrahedron nt. The tetrahedra vertices have been previously sorted.

// Send A, B, C, D to the six vertices
cgGLSetParameter3fv (tetraVertexA, tetraList[nt][0]);
cgGLSetParameter3fv (tetraVertexB, tetraList[nt][1]);
cgGLSetParameter3fv (tetraVertexC, tetraList[nt][2]);
cgGLSetParameter3fv (tetraVertexD, tetraList[nt][3]);

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

// Triangle 0
glVertex2i (0, 3); // P3
glVertex2i (0, 2); // P2
glVertex2i (0, 1); // P1
```

(4) Clear the p-buffer and set the logical pixel operation to GL_ XOR.

(5) Set the uniform parameter *slice* to y_s. Draw the triangles.

(6) Transfer the current slice from the framebuffer to a 3D texture or main memory.

(7) Decrement y_s and return to step 4 until $y_s = 0$.

In this implementation, the solid voxelization is almost entirely solved by the GPU. This is very interesting because computing a voxelization of a solid for high resolutions requires a high amount of time and space. Another advantage of this approach is that once the display list has been compiled, or the vertex arrays have been setup and sent to the graphics adapter, several voxelizations of the entire solid or different parts of it, at different resolutions can be effi-

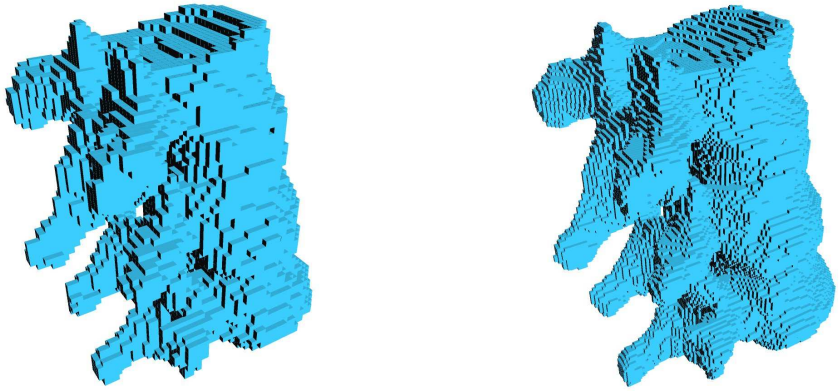


Fig. 3. Examples of voxelizations at different resolutions.

ciently computed. The main different between the vertex arrays and display lists implementations is that the first one sends the minimal set of tetrahedra to each layer, in contrast to the second one, which requires the full set of tetrahedra to be compiled in a previous stage. 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 a color per voxel in a very simple and efficient way. Example shown in Figure 4 illustrates the results of a applying a 3D Perlin noise generator implemented as a fragment program to a voxelization computed by one of the previous methods.

6 Experimental results

We have compared our two approaches, hardware-accelerated, and GPU accelerated using vertex programs against the classical boundary-only voxelization algorithm of Sramek's [SK99a]. Our implementation of this method was done by adapting the code from his VXT library. All the algorithms have been im-

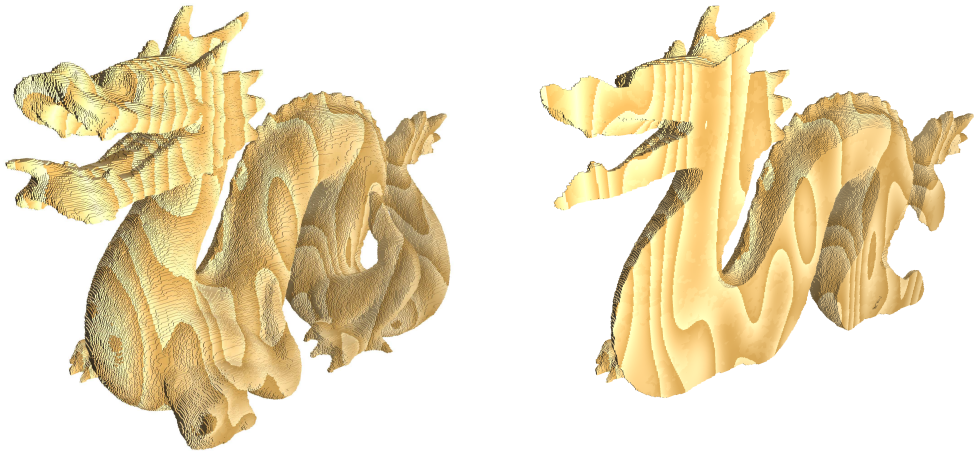


Fig. 4. Use of a 3D Perlin noise generator to set a color per voxel.

plemented in C++, using the same compiler, data structures, coding style and optimizations. The tests ran on a AMD64 2.2Ghz with 1 GB of RAM and a GeForce 6800 on Windows XP, using the models shown in Figure 5.

Tables 3 and 2 show the excellent performance of the algorithm against a conventional method. Sramek's method only beats our approach at low resolutions with large models. This is due to the time required to build the display list during the setup phase, which is dependent on the number of vertices. In the voxelization phase this delay can not be recovered because of the low resolution. Our approach performs better as the number of vertices and triangles of the model keeps low and the voxelization resolution keeps high. The strongest point of Sramek's algorithm is that time grow slowly with an increasing number of triangles, and is mainly dependent on the voxelization resolution.

Table 2 illustrates the noticeable time differences between the GPU-based approach based on vertex arrays and display lists. The execution times of the second implementation are remarkably better, although its performance falls abruptly with large models. The reason for this fall may be in the size of the

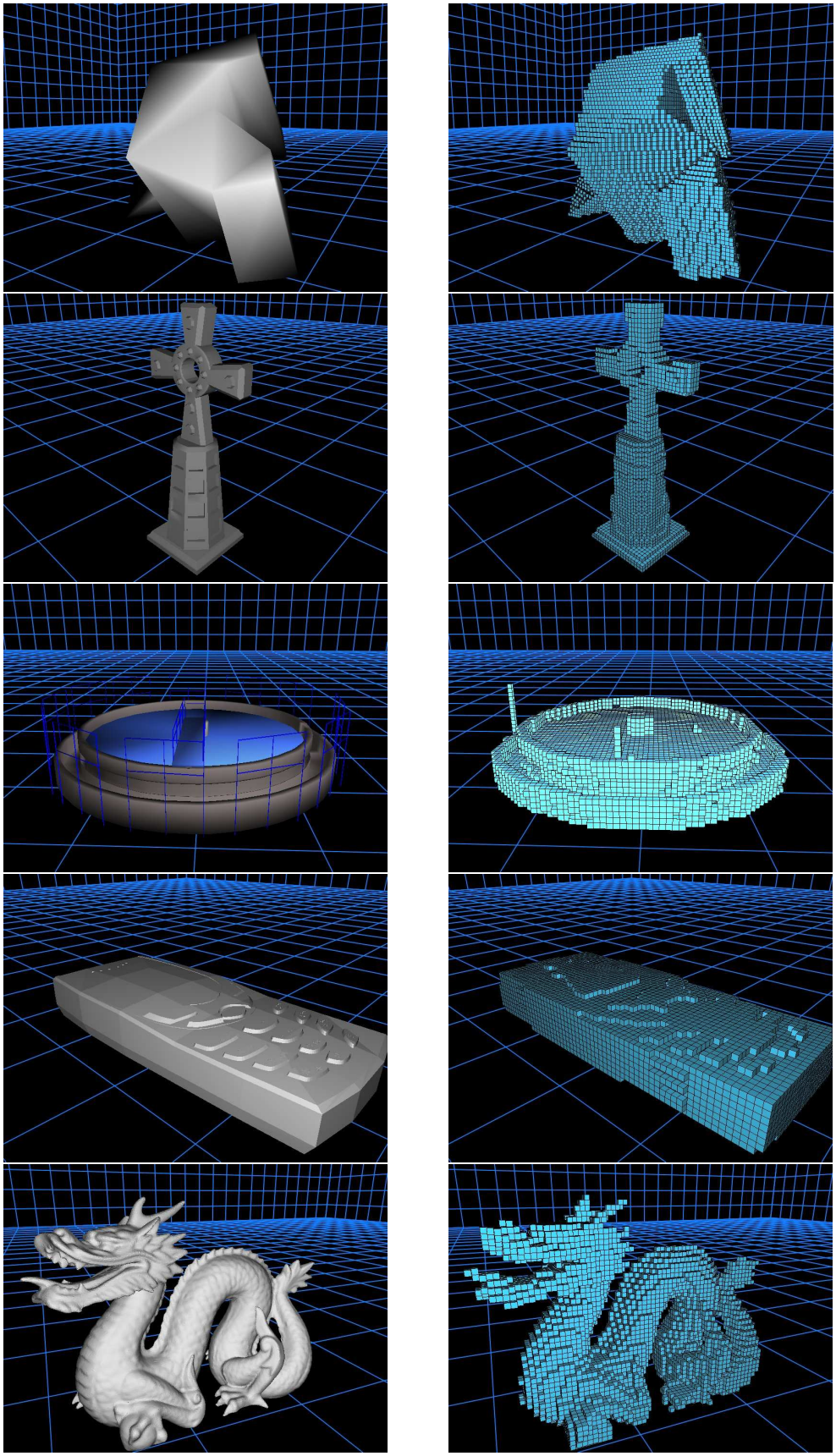


Fig. 5. Models used for the voxelization tests.

			64^3			128^3		
Model	Vertices	Triangles	New HA	New VP	Sramek	New HA	New VP	Sramek
Simple	42	25	0.0088	0.0031	0.0563	0.0032	0.0032	0.6034
Celtic Cross	2366	1849	0.0147	0.0014	0.0363	0.0292	0.0037	0.4554
Depot	10591	5466	0.0511	0.0009	0.0693	0.1092	0.0035	0.5791
Mobile Phone	25946	13025	0.0994	0.0012	0.0656	0.1988	0.0023	0.4883
Dragon	202520	100207	1,2508	1.7029	0.2887	2.3448	4.0169	0.7611

Table 2

Voxelization times (in secs.) of the hardware-accelerated implementation (New H), GPU vertex program implementation using display lists (New VP) and Shramek's boundary-only conventional method for 64^3 and 128^3 voxel resolutions.

			256^3			512^3		
Model	Vertices	Triangles	New HA	New VP	Sramek	New HA	New VP	Sramek
Simple	42	25	0.0134	0.0058	4.4940	0.0696	0.0108	43.9656
Celtic Cross	2366	1849	0.0634	0.0047	3.7144	0.1886	0.0110	46.0224
Depot	10591	5466	0.2319	0.0042	3.9336	0.5476	0.7433	38.6934
Mobile Phone	25946	13025	0.6091	0.8746	3.7207	0.9039	1.9770	49.0143
Dragon	202520	100207	4.7673	8.6638	4.0288	9.2517	18.0581	40.4131

Table 3

Voxelization times (in secs.) of the hardware-accelerated implementation (New H), GPU vertex program implementation using display lists (New VP) and Shramek's boundary-only conventional method for 256^3 a 512^3 voxel resolutions.

display list, and it is likely to be dependent on the graphics hardware used. Although slower, vertex arrays show a more linear and predictable behaviour.

	64^3		128^3		256^3		512^3	
Model	VA	DL	VA	DL	VA	DL	VA	DL
Simple	0.0030	0.0031	0.0017	0.0032	0.0030	0.0058	0.0077	0.0108
Celtic Cross	0.0069	0.0014	0.0100	0.0037	0.0201	0.0047	0.0369	0.0110
Depot	0.0935	0.0009	0.1869	0.0035	0.3853	0.0042	0.7272	0.7433
Mobile Phone	0.2337	0.0012	0.4522	0.0023	0.8746	0.6091	1.8530	1.9770
Dragon	1.6914	1.7029	3.3484	4.0169	6.6215	8.6638	13.5793	18.0581

Table 4

Voxelization times (in secs.) of the GPU implementations, based on vertex arrays (VA) and display lists (DL).

7 Conclusions

In this work we have presented a simple and efficient method for the voxelization of polyhedra which can be easily implemented using common graphics hardware or GPUs with vertex program support. Both implementations have two advantages: they are very fast, and release the CPU from the expensive computations implied in the voxelization process.

As we have seen in the previous section, the performance of the GPUs approach is noticeable worse when working with large models, because of the high number of vertices and triangles that have to be processed in the setup phase. We believe that this setup phase can be faster if the tetrahedra information is stored in a vertex texture and accessed from the vertex program during the voxelization. This avoids the construction of the parameter arrays in the vertex program approach, and allows the compilation of smaller display lists.

References

[BS00] Blundell, B., Schwarz, A. Volumetric Three-Dimensional Display Systems. *John Wiley*, 2000.

[EBM*99] Ebert, D., Bedwell, E., Maher, S., Smoliar, L. Downing, E. Realizing 3D visualization using crossed-beam volumetric displays. *Communications of the ACM*, 42, pp. 101-107, 1999.

[FC00] Fang, S., Chen, H. Hardware accelerated voxelization. *Computer & Graphics*, 24, pp. 433-442, 2000.

[FT95] Feito, F., Torres, J. C. Orientation, simplicity and inclusion test for planar polygons. *Computer & Graphics*, 19, pp. 596-600, 1995.

[FT97] Feito, F., Torres, J.C. Inclusion test in general polyhedra. *Computer & Graphics*, 21, pp. 23-30, 1997.

[FDF*94] Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. H. *Introduction to Computer Graphics*, Addison Wesley, 1994.

[HW02] Haumont, D., Warzie, N. Complete polygonal scene voxelization. *Journal of Graphics Tools*, 7, pp. 27-41, 2002.

[HYF*98] Huang, J., Yagel, R., Filippov, V., Kurzion, Y. An accurate method for voxelizing polygon meshes. *Proceedings of the IEEE symposium on Volume Visualization*, pp. 119-126, 1998.

[Jon96] Jones, M. W. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, **15**, 5, pp. 311-318. 1996.

[Kau93] Kaufman, A., Cohen, D., Yagel, R. Volume Graphics. IEEE Computer, **26**, 7, pp. 51-64, 1993.

[PK00] Pastoor, S., Kiesewetter, R. 3-D displays: A review of current technologies. *DISPLAYS*, 17, pp. 100-110, 1997.

[RSF04] Rueda, A. J., Segura, R., Feito, F. R., Ogayar, C. Voxelization of solids using simplicial coverings *Proceedings of WSCG'2004*, pp. 227-234, 2005.

[RSR*04] Rueda, A. J., Segura, R., Ruiz de Miras, J., Feito, F. R. Rasterizing complex polygons without tesselations. *Graphical Models*, 26, pp. 805-814, 2004.

[SK99a] Sramek, M., Kaufman, A. A. Alias-free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics*, 5, pp. 251-267, 1999.

[SK99b] Sramek, M., Kaufman, A. VXT: a C++ class library for object voxelization. Proceedings of the International Workshop on Volume Graphics, 1999.

[WND99] Woo, M., Nedider, J., Davis, T., Shreiner, D. *The OpenGL Programming Guide, 3rd. edition*, Addison Wesley, 1999