Vol. 14, No. 3: 21–34

Real-Time Hierarchical Binary-Scene Voxelization

Vincent Forest, Loic Barthe and Mathias Paulin
University of Toulouse - IRIT

Abstract. Volumetric representations provide the localization of shapes in space. When such representation is created on the fly from the geometry, it becomes very useful for a wide range of applications (constructive solid geometry (CSG), shape repair, collision detection, etc. Using the advanced functionalities provided by recent GPUs (geometry shaders, 32-bit integer texture format and bitwise operators), we show how to compute a robust scene voxelization and octree construction in a few milliseconds from any hardware-supported rasterizable geometry. Our hierarchical volumetric representation is thus especially well-suited for hierarchical computation of fully dynamic scenes.

1. Introduction

Voxels are an intuitive discrete representation of three-dimensional space. They provide the information where the geometry is located in space; the process of computing the voxel representation is called *voxelization*. [Kaufman and Shimony 86]. When representing a surface, voxels can be seen as a regular grid storing binary data: the voxel either intersects the surface of the object or not. The natural extension of voxels to multi-resolution is the octree. An octree node stores the presence or absence of the surface in its subdivision up to the grid resolution,i.e., if its corresponding volume in space intersects the surface. Thus, our voxel representation only stores surface position without considering inside/outside information.

Although several software voxelizations lead to high rendering quality [Wang and Kaufman 93, Haumont and Warze 02, Widjaya et al. 03], their per-

1086-7651/09 $0.50 per page

formance remains insufficient to achieve real-time voxelization of objects or scenes composed of millions of triangles. Based on the idea that the voxelization process is nothing more than a 3D rasterization, the hardware rasterizer can be used to accelerate the voxelization of complex models. Following this observation, Zhao [Dong et al. 04] rasterizes triangles in three intermediate sheet buffers oriented in the direction of each axis. During a pre-processing step, the triangles are sorted according to their orientation in order to define in which sheet buffers they are to be rendered. The rasterization consists in defining which voxels are intersected by the rendered primitive (Figure 1). Unfortunately, the voxelization is limited to triangle primitives and the performance is influenced by a dynamic update of the sorted triangles required by deformable objects. Eisemann and Décoret presented an extension to this approach [Eisemann and Décoret 06]. They avoid its drawbacks by voxelizing the scene in a slice map with a common rasterization pass. However, the resulting voxelization may exhibit holes in the boundary representation.

In this paper we present a robust regular hierarchical voxel representation, similar to a regular binary octree, built from a robust scene-voxelization process (Section 2). The structure generation is performed on the GPU-side in a few milliseconds, does not require any specific treatment by the reasterizable primtive, and is independent of the rasterizable primitive. In contrast to previous hardware-accelerated voxelizations, our algorithm computes a *multi-resolution* voxel representation. In addition, it does not rely on any pre-processing step [Dong et al. 04], and the resulting representation does not exhibit holes [Eisemann and Décoret 06]. Finally, this hierarchical structure merges the geometric primitives, i.e., the voxel, with the space-partitioning data structure. As a result, it is particularly well-suited for many domains from visibility computations (Section 3) to hierarchical collision detection.

2. Hierarchical Voxel Data Structure

A voxel representation is a uniform grid that can be generated very quickly, even on complex models. Following the same starting idea as Eisemann and Décoret [Eisemann and Décoret 06] and Zhao [Dong et al. 04], we first build a simple voxel representation on the GPU, i.e., a regular grid whose voxels only store a single bit (1) if it intersects the surface or (0) if not. The outlines of the grid are given by the projection parameters and its x- and y-resolution is the one of the viewport. During the rasterization, all generated fragments are treated and the z-value is used to identify the depth position of the intersecting voxel rather than perform a z-test. In common graphics hardware, each pixel of the frame buffer is encoded with four bytes (RGBA). In the grid, these four bytes are used to discretize the z-direction into 32 cells, i.e., one bit defines a cell in a column of 32 voxels. The grid resolution is then $w \times h \times 32$.

Our voxelization algorithm exploits the capabilities offered by the G80 GPU to build a larger data structure very efficiently in a single rendering pass using a very simple fragment program. Then, we show how to provide a multi-resolution volumetric representation of the scene via an octree-like data structure computed with an un-perceivable impact on performance.

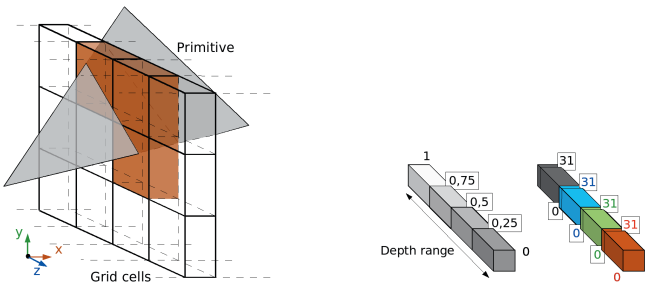
2.1. Grid Representation

The G80 introduces the 32-bit precision integer format for both computation and storage. We use this format for encoding the voxel representation in a 32-bit integer RGBA texture (Figure 1(b)). We then multiply the z-resolution by 4 without adding new textures. Additionally, the use of a multiple render target (MRT) allows us to combine several 32-bit integer RGBA textures. Thus, each additional render target increases the depth precision by 128 bits

The G80 offers 8 MRTs resulting in a depth resolution of 1024. Moreover, the frame buffer can be partitioned to render a depth range of the grid in a tile. In practice, common applications rarely exceed a resolution of 512^3 corresponding to 512 (width)×512 (height)×4 (MRT)×4 (channels)×4 (bytes per channel) = 16MB. Note that, combining MRTs and tiling theoretically allows the creation of a 4096^3 grid (8GB of memory).

2.2. Real-Time Voxelization

In order to simplify explanations, we present our data-structure construction with a grid size of 128^3. We set the projection parameters and the viewport according to the grid border and its (x, y)-resolution. For a non-deformed voxelization, the camera projection has to be orthographic. To perform direct rendering in the RGBA integer texture, it is attached to a frame buffer object



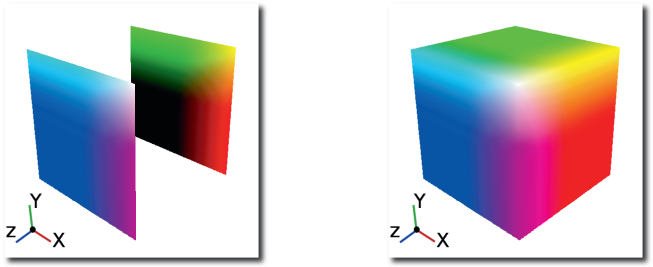
(a) Voxelization (b) Depth range

Figure 1. (a) The voxelization process sets the voxels that intersect the primitive to 1. (b) Discretized representation of the grid depth range using a 128-bit RGBA integer texture.

(FBO). Since all fragments have to be treated, face culling and z-test are disabled and hence no z-buffer is attached to the FBO.

During rendering, we define for each rasterized fragment the intersected voxel, and we set its corresponding bit to 1. To bypass the non-linearity of the depth buffer, Eisemann and Décoret use the real distance to the camera computed in a vertex program. Then, they perform a texture look-up to find the slice in which the fragment falls. In our approach, neither vertex shader computation nor texture look-up is necessary. In a fragment program, we compute the world space coordinates of the incoming fragment (fPosW.z). According to the grid boundaries (maxZ and minZ), we compute its corresponding depth in the grid and identify the intersected voxel. To compute the corresponding 128-bit mask, we first define the subgroup of 32-bits, i.e., a color channel, intersected by the rasterized fragment. Then we set to 1 the intersected bit in the selected color channel, and we write the resulting mask in the frame buffer. Finally, we take advantage of the G80 logical operations on 32-bit integers to OR the 128-bit mask with the frame buffer. This very fast process gives us a first voxelization using only some rendering-state settings and the following very short fragment program:

```
!!NVfp4.0
#=============================================================================
#  Voxelization fragment program writen with the NV_gpu_program4 opengl extension
#=============================================================================
PARAM vtw[4]     = {state.matrix.program[0]};# viewport to world space matrix
PARAM gridParams = program.local[0];         # {maxZ, minZ, 127/(maxZ-minZ), 0}
ATTRIB fPos      = fragment.position;
OUTPUT oCol      = result.color;

TEMP r0, r1;
# 1) Transform the fragment position in world space
SWZ      r1,     fPos,   x, y, z, 1;         # r0.z = fPos.z in World Space <=> fPosW.z
DP4      r0.z,   vtw[2], r1;
MUL      r0.z,   r0.z,   fPos.w;

# 2) Compute the index of the intersection between
# the surface and the grid in the Z direction
SUB      r0.x,   r0.z,   gridParams.y;       # r0.x = fPosW.z mapped in [0, 127]
MUL      r0.x,   r0.x,   gridParams.z;

# 3) Retreive the channel in which the intersected
# voxel is stored:
#   - 0  <= index < 32  --> red channel
#   - 32 <= index < 64  --> green channel
#   - 64 <= index < 96  --> blue channel
#   - 96 <= index < 128 --> alpha channel
SGE.CC0  r1,     r0.x,   { 0, 32, 64,  96};  # CC0=r1=intersected channel is set to 1
SLT.CC0  r1(NE0),r0.x,   {32, 64, 96, 128};

# 4) Set to one the intersected voxel in the channel
DP4      r0.y,   r1,     { 0, 32, 64,  96};  # r0.y = id of the bit in 128-bits mask
SUB      r0.x,   r0.x,   r0.y;               # r0.x = id of the bit in selected channel
TRUNC.U  r0.x,   r0.x;
SHL.U    oCol(NE0), 0x01, r0.x;              # oCol = resulting 128-bits mask

END
```



(a) one projection direction (b) three projection directions

Figure 2. Voxelization with an orthographic projection of an axis-aligned cube. (a) Previous voxelization process [Eisemann and Décoret 06]. (b) Our robust single-pass voxelization algorithm.

One of the main drawbacks of the Eisemann and Décoret method is that the voxel representation exhibits holes where the slope of a surface is close to the grid's z-axis as illustrated in Figure 2. We overcome this limitation and provide a final robust voxelization in a very simple and efficient manner. Our solution performs three renderings placing the camera toward the $-z$-, $-y$- and $-x$-direction of the grid. Then, a GPGPU pass performs the union of the three grids in the final grid texture (Figure 3). However, to avoid the geometry bottleneck generated by multiple renderings, we transform the geometry following the three grid directions ($-x$, $-y$, $-z$) in the geometry processors. The transformed geometries are rendered, respectively, in three layers of a layered texture, and the final GPGPU pass performs their union. Note that this approach is quite similar to Zhao's. However, we are neither limited to triangles nor committed to a pre-processing step. In addition, thanks to the geometry program and its instantiations capabilities, the geometry is sent only once to the graphics hardware, thus reducing the bandwidth bottleneck.

2.3. Multiresolution Spacial Partitioning Data Structure

Since voxels are stored in a 2D texture we use a mipmap process to generate our hierarchical structure. At each mip-level, a texel is a column of voxels

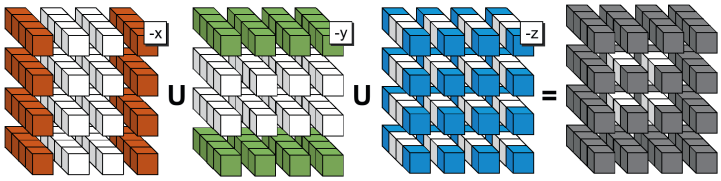


Figure 3. Combination of the three grids built from the $-x$-, $-y$- and $-z$-directions.

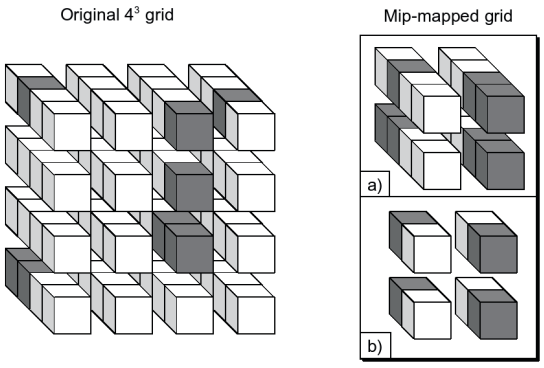


Figure 4. Illustration of our octree-like construction by mip-reduction of a 4^3 grid. (a) Mip-reduction only in the x- and y-direction. (b) Mip-reduction in x-, y- and z-directions.

in the z-direction. It indicates if voxels of its four previous mip-level texels intersect a primitive or not. Above the finest mip-level, a texel is generated as the OR operation of its four previous mip-level texels (Figure 4(a)). In this representation, all mip-levels have the same z-resolution. Note that this "over depth precision"-representation may be used to save memory accesses. Indeed, only one texture fetch is needed to access a column of voxels in the z-direction. Obviously, if it is required by the application, it is possible to perform a mip reduction in z using the OR operation on groups of bits of the same texel (Figure 4(b)). This generates a standard octree representation.

The same texture format is used for all mip-levels and so the octree-like representation is saved in a single mipmapped texture. Hence, a mip-level directly corresponds to a depth level in the octree. The use of texture tiling to store depth ranges when the depth of the grid exceeds 128 makes this representation especially well-suited for GPUs up to 2048^3 grids.

2.4. Advantages and Drawbacks

Our hierarchical structure offers several advantages. First, the voxelization is a hardware process leading to high performance. The data structure is stored on the GPU, and it never has to be transferred on the CPU even for the hierarchy construction that is performed by a GPGPU redux pass [Harris 05]. Moreover, the texture representation of the data structure is very efficiently accessed by common shaders. Our algorithm also holds for any hardware-supported rasterizable primitive (points, triangles, etc.). Note that binary alpha-textured objects are also supported. In this case, the alpha-kill must

Fragment processing	Knot	Tree	Spheres
enabled	2.5	17.5	1.5
disabled	2.5	17.3	1.4

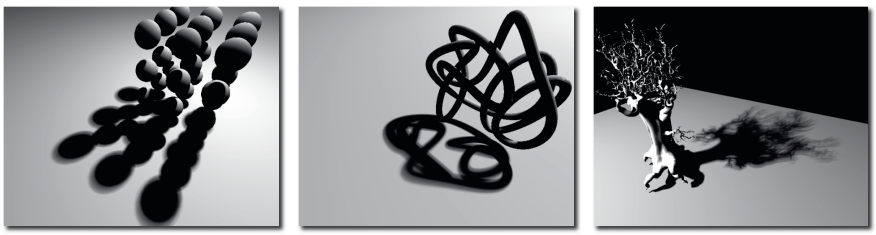
Table 1. Performance of our robust voxelization algorithm; time in milliseconds; grid resolution, 128^3. Note that the bottleneck of the voxelization process is the fix function rather than our fragment program since performances remain unchanged when fragment processing is disabled.

be performed in the fragment program, since no test has to be applied on generated fragments. In addition, our structure is both very fast to compute and independent of the object animation, deformation, and storage (CPU- or GPU-side). Hence, it naturally deals with dynamic scenes composed of complex objects eventually animated or deformed via the GPU.

We point out that the overall time required by the voxelization computations includes first the GPU's transformation and rasterization steps of the primitives (performed by common fixed functions) and then the voxelization via our fragment shader. When the voxelization fragment shader is disabled, the performance remains unchanged (Table 1). Hence, the bottleneck of the voxelization process is in fact the fix functions effectiveness, as for any rendering of a geometry via a GPU rasterization.

Our data structure generation is also a very fast process. With a naïve CPU implementation, generating the mip-levels of a 128^3 grid on an Intel Core 2 Duo E6600 takes 0.7 ms. Using a GPGPU pass to avoid data transfer from GPU to CPU, we generate the hierarchy from the same grid in less than 0.1 ms.

As any voxel-based representation, the main drawback of the proposed voxelization is its memory consumption. Indeed, even though we store only a single bit per voxel, the memory footprint becomes prohibitive when we deal with a thin octree resolution, e.g., 1GB for a 2048^3 resolution).



(a) Spheres (b) Knot (c) Tree

Figure 5. Illustration of our test scenes. (a) A cubic form composed of 9×9 spheres (6,848 polygons); (b) a knot (52,992 polygons); (c) a tree (607,608 polygons).

(a) (b)

Figure 6. Illustration of our real-time hierarchical voxelization on a video-game scene. (Objects and textures ©Valve Corporation, used with permission.)

We define the voxelization as the process setting to 1 the voxels that are intersected by a triangle (Figure 1). However, the proposed real-time voxelization is based on hardware-accelerated rasterization. In general situations, the rasterization rendering algorithm generates fragments only when the *pixel center* is covered by the primitive. In order to generate fragments for every pixel partially covered by a geometric primitive, one has to perform a conservative rasterization of the voxelized geometry [Hasselgren et al. 05].

3. Utilization of Our Dynamic Multi-Resolution Data Structure

Due to its GPU-friendly implementation and its very fast generation, our hierarchical representation is useful for a wide variety of applications. Indeed,

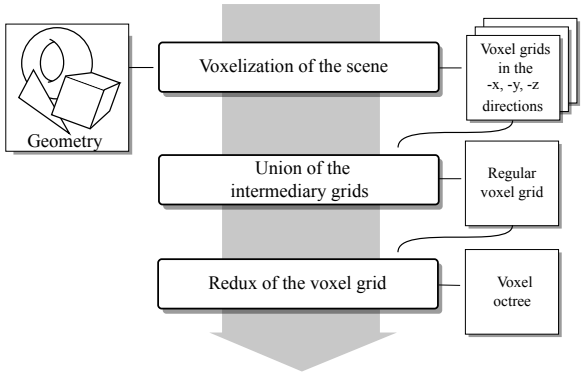


Figure 7. Overview of our voxelization algorithm. All computations are performed on the GPU without transferring data to main memory.

we enrich the original rasterizable shape representation on the fly with a dynamic multi-resolution volumetric structure. This structure is computed in a very simple way (Figure 7). This simple process provides the object with a double shape/multi-resolution volume representation.

In this section, we show how we efficiently take advantage of this representation to accelerate ray-traced shadows on the GPU. Then, we discuss several other applications in which our multi-resolution structure can lead to valuable investigations and results.

3.1. Ray Tracing on the GPU for Visibility Computation

A visibility test states if a point is visible or not from another one. This test only requires the knowledge of the presence or absence of a primitive in between the two points. In our application, the visibility tests are performed using a GPU ray-tracer, and we show how it takes advantage of our octree-like space-partitioning data structure. As a result, we provide physically plausible soft shadows.

To compute shadows we shoot rays from fragments to samples on light sources through the octree, and we check if rays intersect a voxel set at 1 (i.e., intersect a primitive). Lefebvre et al. proposed a GPU-based algorithm that efficiently accesses a specific cell in an octree [Lefebvre et al. 05]. Despite some similarities, this technique does not solve the octree traversal by a ray.

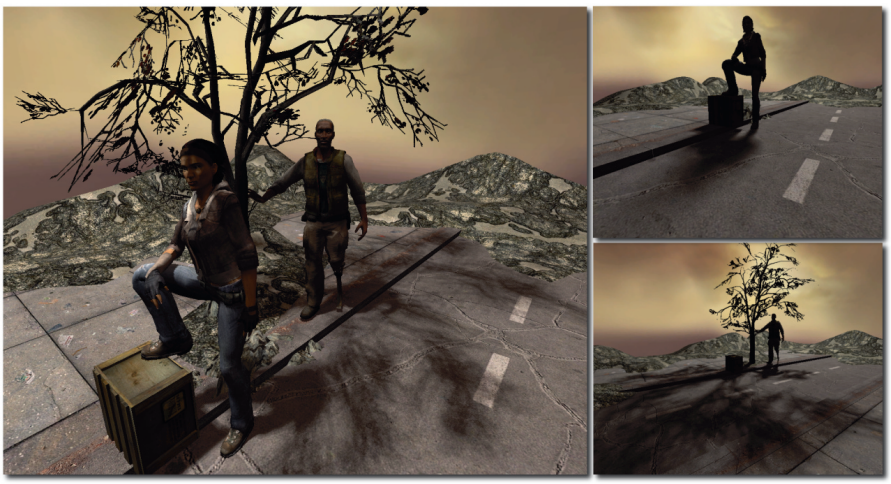


Figure 8. Illustration of our ray-traced soft shadows on a scene voxelized in a 512^3 octree. Number of shadow ray per fragment: 128.

3.1.1. Octree Traversal

Revelles et al. proposed a very simple and efficient parametric algorithm for octree traversal on the CPU [Revelles et al. 00]. We reformulate and simplify this algorithm according to our octree-like representation and its implementation on the GPU. In the following, we describe the specific steps of the reformulation. Thus, we refer to [Revelles et al. 00] for a complete description of the algorithm.

We identify a node at a depth d as $I_d = (x, y, z)$ where x and y are the texture coordinates and $1 - z$ is the mip-level, all mapped in the $[0, 1]$ range. The octree level $d = 0$ corresponds to the maximal mip-level in the mipmap hierarchy. If the octree is intersected by the ray, the octree traversal is initialized with $I_0 = (0, 0, 0)$ (the octree root) and ends when the ray reaches the light, leaves the octree, or hits a primitive. At a given depth, the octree traversal is performed by incrementing the normalized texture coordinates in

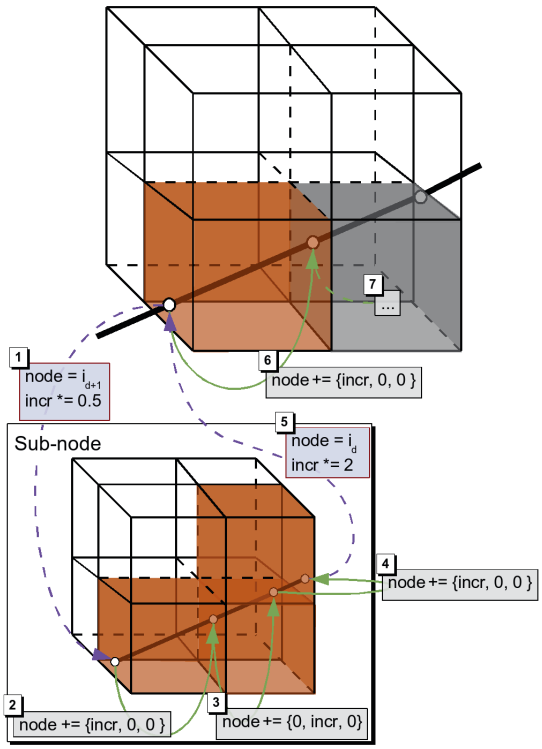


Figure 9. To define the next node of the same level, we add the octree-depth–dependent increment to the previous node according to the traversal direction. This increment has to be divided by two for each octree-refinement.

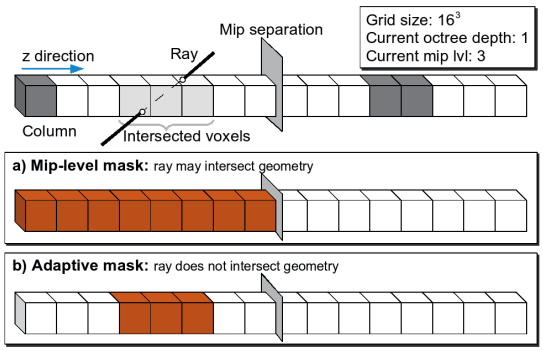


Figure 10. Comparison between a mip-level mask and an adaptive mask to define which voxels may be the leafs of the current octree node. Note that the adaptive mask avoids unnecessary node refinements.

the direction of the next visited node. This increment incr is the voxel size ($\in [0, 1]$) at the current depth. It is first set to 1 at I_0, and it is divided by 2 at each octree-depth refinement (Figure 9).

In order to test if there is an intersection, we have to read the node value. In our mipmap hierarchy, the node is accessed by mapping the normalized texture coordinates I_d in the [0, mip-level.size]^3. In the z-direction, we store the full resolution. Hence we have to retrieve which voxels are effectively intersected by the ray. We take advantage of the full resolution in z to define a small mask whose width is the exact number of voxels hit by the ray in the column. These voxels are identified by computing the inner and outer intersections between the ray and the column (Figure 10(b)). This optimization allows us to directly eliminate nodes that would have been considered as potentially intersecting a primitive and refined (Figure 10(a)). Note that Revelles et al.'s method only deals with rays of positive directions. Thus, for negative values, node traversal must be reversed. In our GPU algorithm, this is done with reversed texture coordinates ($I_d = 1 - I_d$) and a negative increment.

3.1.2. Performance Analysis

Our implementation is based on OpenGL while the ray-tracer is written with the pseudo-assembly language proposed by the `NV_gpu_program4` extension. We measure the performances on *three* test scenes (Figure 6), on a Linux workstation composed of a Core 2 Duo E6600 with 4GB of memory and a Geforce 8800GTX.

As pointed out in Section 2.4, the update of our space-partitioning data structure has a negligible impact on the performances. As a consequence, the effectiveness of our ray-tracer only relies on the visibility computation and the

Rays per fragment	Knot	Tree	Spheres
1	0.029	0.046	0.017
4	0.104	0.127	0.054
8	0.201	0.230	0.118
16	0.390	0.437	0.236
32	0.771	0.844	0.513
128	3.053	3.313	1.963
512	12.104	13.096	7.662

Table 2. Influence of the number of light samples per fragment on the performance. Image resolution, 1024×768; time in seconds; octree resolution, 128^3.

number of rays. In practice, the computation times are linear with respect to the number of rays (Table 2), and the performance decreases when, while increasing its resolution, the octree captures new details or topology of the represented geometry (Table 3). Indeed, the intersection query then requires more recursion in the octree hierarchy.

The ALU horsepower of the GPUs increases more quickly than their memory bandwidth. As a consequence, we have to privilege arithmetic operations rather than memory accesses. Our hierarchical algorithm fulfills this constraint. Indeed, only a single memory access is required to check if a region is empty while in a regular grid all voxels in the region must be accessed.

Octree resolution	Knot	Tree	Spheres
$\mathbf{8^3}$	0.008	0.024	0.004
$\mathbf{16^3}$	0.012	0.029	0.006
$\mathbf{32^3}$	0.016	0.034	0.009
$\mathbf{64^3}$	0.022	0.039	0.013
$\mathbf{128^3}$	0.029	0.046	0.017
$\mathbf{256^3}$	0.056	0.085	0.030
$\mathbf{512^3}$	0.067	0.103	0.041

Table 3. Influence of the octree resolution on the performance. Image resolution, 1024×768; time in seconds; 1 ray per fragment.

3.2. Possible Utilization of Our Dynamic Multi-Resolution Data Structure

Visibility computations in our structure can be used to accelerate pre-computed [Kontkanen and Aila 06] or real-time [Bunnell 05] ambient occlusion. Beyond purely visibility computations, our structured multi-resolution surface repre-

sentation is also well-suited for shadow-volume culling and clamping [Lloyd et al. 04] as well as hierarchical collision detection.

A voxel can store more than a simple bit. Obviously, the larger the number of bits, the smaller the maximal octree size. If we store a byte or more, an obvious extension is the application of our voxel representation to coarse distance fields or, more generally, discrete potential-field representations [Sud et al. 04].

Finally, our real-time hierarchical voxelization is particularly well-suited to represent dynamic obstacles in fluid-solid interaction [Crane et al. 07].

References

[Bunnell 05] Michael Bunnell. "Dynamic Ambient Occlusion and Indirect Lighting." In *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pp. 223–233. Reading, MA: Addison Wesley, 2005.

[Crane et al. 07] Keenan Crane, Ignacio Llamas, and Sarah Tariq. "Real-Time Simulation and Rendering of 3D Fluids." In *GPU Gems 3*, edited by Hubert Nguyen, pp. 633–675. Reading, MA: Addison Wesley, 2007 2007.

[Dong et al. 04] Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng. "Real-time Voxelization for Complex Polygonal Models." In *PG '04 Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, pp. 43–50. Washington, DC: IEEE Computer Society, 2004.

[Eisemann and Décoret 06] Elmar Eisemann and Xavier Décoret. "Fast Scene Voxelization and Applications." In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 71–78. New York: ACM Press, 2006. Available online (http://artis.imag.fr/Publications/2006/ED06).

[Harris 05] Mark Harris. "Mapping Computational Concepts to GPUs." In *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pp. 493–508. Reading, MA: Addison Wesley, 2005.

[Hasselgren et al. 05] Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. "Conservative Rasterization." In *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pp. 677–694. Reading, MA: Addison Wesley, 2005.

[Haumont and Warze 02] Denis Haumont and Nadine Warze. "Complete Polygonal Scene Voxelization." *journal of graphics tools* 7:3 (2002), 27–41.

[Kaufman and Shimony 86] Arie Kaufman and Eyal Shimony. "3D Scan-Conversion Algorithms for Voxel-Based Graphics." In *Workshop on Interactive 3D Graphics*, pp. 45–75. New York: ACM Press, 1986.

[Kontkanen and Aila 06] Janne Kontkanen and Timo Aila. "Ambient Occlusion for Animated Characters." In *Proc. EUROGRAPHICS Symposium on Rendering*,

edited by Thomas Akenine-Möller Wolfgang Heidrich, pp. 343–348. Aire-la-Ville: Switzerland: Eurographics Association, 2006.

[Lefebvre et al. 05] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. "Octree Textures on the GPU." In *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pp. 595–613. Reading, MA: Addison Wesley, 2005.

[Lloyd et al. 04] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. "CC Shadow Volumes." In *SIGGRAPH Sketches 2004*, p. 146. New York: ACM Press, 2004.

[Revelles et al. 00] J. Revelles, Carlos Ureña, and M. Lastra. "An Efficient Parametric Algorithm for Octree Traversal." *Journal of WSCG* (2000), 212–219.

[Sud et al. 04] Avneesh Sud, Miguel Otaduy, and Dinesh Manocha. "DiFi: Fast 3D Distance Field Computation Using Graphics Hardware." In *Proc. EUROGRAPHICS '04*, pp. 557–566. Aire-la-Ville, Switzerland: Eurographics Association, 2004.

[Wang and Kaufman 93] Sidney W. Wang and Arie E. Kaufman. "Volume Sampled Voxelization of Geometric Primitives." In *VIS '93: Proceedings of the 4th Conference on Visualization '93*, pp. 78–84. Washington, DC: IEEE Computer Society, 1993.

[Widjaya et al. 03] Haris Widjaya, Torsten Müller, and Alireza Entezari. "Voxelization in Common Sampling Lattices." In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, p. 497. Washington, DC: IEEE Computer Society, 2003.

Web Information:

http://jgt.akpeters.com/papers/ForestEtAl09

Vincent Forest, University of Toulouse - IRIT, 118 Route de Narbonne, F-31062 Toulouse CEDEX 9 (France) (Vincent.Forest@irit.fr)

Loic Barthe, University of Toulouse - IRIT, 118 Route de Narbonne, F-31062 Toulouse CEDEX 9 (France) (Loic.Barthe@irit.fr)

Mathias Paulin, University of Toulouse - IRIT, 118 Route de Narbonne, F-31062 Toulouse CEDEX 9 (France) (Mathias.Paulin@irit.fr)

Received January 16, 2008; accepted in revised form November 20, 2009.