Two Simple Single-pass GPU methods for Multi-channel Surface Voxelization of Dynamic Scenes

Athanasios Gaitatzes[1], Pavlos Mavridis[2] and Georgios Papaioannou[2]

[1]Department of Computer Science, University of Cyprus
[2]Department of Informatics, Athens University of Economics & Business, Greece
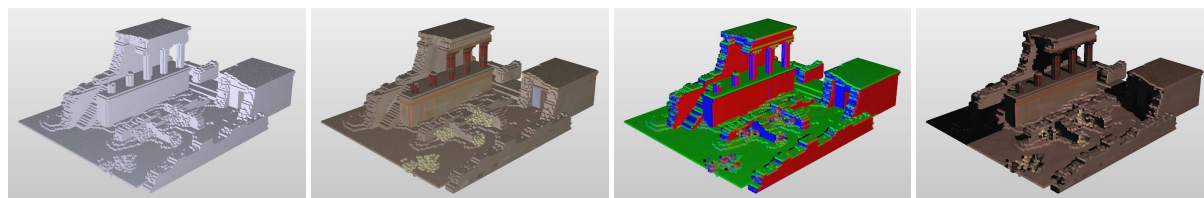

Figure 1: *Voxelization of the Knossos model (109170 triangles) into a 128^3 grid. The volumes in the order that they appear are the occupancy volume, the albedo volume, the normals volume and the lighting volume.*

Abstract

An increasing number of rendering and geometry processing algorithms relies on volume data to calculate anything from effects like smoke/fluid simulations, visibility information or global illumination effects. We present two real-time and simple-to-implement novel surface voxelization algorithms and a volume data caching structure, the Volume Buffer, which encapsulates functionality, storage and access similar to a frame buffer object, but for three-dimensional scalar data. The Volume Buffer can rasterize primitives in 3d space and accumulate up to 1024 bits of arbitrary data per voxel, as required by the specific application. The strength of our methods is the simplicity of the implementation resulting in fast computation times and very easy integration with existing frameworks and rendering engines.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and frame-buffer operations

1 Introduction

Volume representation of polygonal models is an important basic operation for many applications in computer graphics and related areas. Polygonal models, for example, have often been substituted by volume representations to remove unnecessary complexity for certain calculations, to provide a uniform sampling of the underlying data, to structure multi-resolution information in an easily and rapidly accessible manner or to enhance the models with additional data. Voxelization methods have been used in domains as diverse as global illumination computation [THGM11], fluids simulation [CLT07] and ambient occlusion computation [PMP10].

Surface voxelization describes the process of turning a scene representation consisting of discrete geometric entities (e.g. triangles) into a three-dimensional regular spaced grid that captures the surface of the scene. Each cell of the grid encodes specific information about the scene. In the case of binary voxelization, a cell represented by single bits in a bitmask stores whether geometry is present in it or not. In a multi-valued voxelization, occupancy is extended to represent the (scalar) coverage of a voxel by the geometry and can also store additional spatial information.

We present a novel voxelization algorithm and volume data-caching structure, the Volume Buffer, which encapsu-

lates functionality, storage and access similar to a *frame buffer object*, but for three-dimensional data. The Volume Buffer can rasterize primitives in 3d space and accumulate up to 1024 bits of arbitrary data per voxel, as required by the specific application, by using up to 8 floating point render targets, as necessary (where 8 is currently the maximum available number of MRTs). The strength of our method is the simplicity of the implementation (about 15 lines of geometry shader code and 1 line of pixel shader code - see Section 3.2) resulting in fast computation times and a very easy integration with existing engines and rendering frameworks.

Our multi-channel voxelization algorithm runs entirely on the GPU and can generate volume data from arbitrary complex and dynamic models in real time. The proposed volume sampling technique is not limited to providing an occupancy volume representation of the scene, but also a complete attribute set for complex calculations (i.e. in global illumination calculations an albedo buffer, a normal buffer, a direct lighting buffer can be generated). This way heavy computations are disassociated from the surface representation data, thus making the method suitable for both primitive-order and screen-order rendering, such as deferred rendering. We do not require watertight models nor is our method dependent on the depth complexity of the scene.

2 Previous Work

Many voxelization algorithms with various properties have been devised. Among the most relevant real-time approaches are variations of the XOR slicing method that was first presented by Chen et al. [CF98] and Fang et al. [FC00]. The algorithm rendered the geometry once for each slice of the volume grid, each time restricting the view volume to this slice. It required watertight models and suffered from multiple passes over the geometric data, once for each texture slice per sweep axis in order to correctly assign the geometry into voxels.

At the same period, a depth-buffer-based voxelization method appeared by Karabassi et al. [KPT99], which performed a fast volume rasterization of arbitrary geometry but could not voxelize correctly the cavities of objects. Passalis et al. [PTTK07] proposed a depth-peeling multi-directional generalization of the above technique, lifting its concavity restrictions. Unfortunately, their algorithm requires a number of depth layers equal to the scene depth complexity in each sweeping direction, rendering it practical mostly for single object voxelization.

Dong et al. [DCB*04] encode binary voxels in separate bits of multiple multi-channel render targets, allowing to treat many slices in a single rendering pass. A fragment's depth is used to derive the voxel and its bit is set via additive alpha blending. A triangle is rendered only if its normal's dominant direction is parallel to the current axial sweep. Unfortunately, the performance is influenced by the dynamic update of the sorted triangles required by deformable objects or dynamic scenes. Eisemann et al. [ED06] presented an extension to this approach achieving higher performance. Their method, taking advantage of the same efficient encoding, uses the RGBA-channels of a texture as a binary mask to encode the boundary of the scene geometry. The depth of a fragment is used as an indicator as to which bit in the mask has to be set by using the more robust bitwise or-blending. The resulting representation though, frequently exhibits holes as only one viewing direction is considered in the original implementation.

Forest et al. [FBP09] suggest a hierarchical volumetric representation by offering an extension to Dong et al. [DCB*04] method. Furthermore, Zhang et al. [ZCEP07] proposed to use a conservative rasterization approach to capture more details of the scene geometry. Another method based on slicing was presented by Crane et al. [CLT07]. They used the geometry shader to intersect all triangles of the scene with each plane of the three dimensional grid to successively fill each layer.

Schwarz et al. [SS10] directly build a hierarchical volume representation using a GPGPU triangle processing algorithm. It can achieve sparse, high resolution voxelization but it is complex, requires GPGPU architectures and GPU context switching. Thiedemann et al. [THGM11] introduced an interactive volume-based global illumination method, where the spatial occupancy and color data are generated by injecting a geometry texture atlas containing point samples of the polygonal geometry. Their method requires model preprocessing and extra storage for the texture atlas and is sensitive to the point sampling rate and surface deformations.

Contrary to our methods, these approaches do not allow for the storage of multi-channel scalar data at the location of each voxel.

3 Overview of Voxelization procedure

The goal of our voxelization method is to reduce the rasterization and unnecessary clipping operations over the entire volume grid, while sending the geometry from host to device only once per slicing direction. To this end, we regard a slice-order voxel fragment generation along a principal axis.

A volume covering the extents of the scene is created and updated at every frame or whenever the environment changes. In order to sample the triangles coherently, we take three perpendicular volume sweep planes and each triangle is selectively rasterized only to the plane of maximum projection (i.e. to the direction where its normal is mostly aligned with). Therefore the primitives are rasterized *only once*. This ensures that the triangles' surface is densely sampled.

The main operation in both proposed voxelization methods is the clipping or "slicing" of the incoming triangles against the boundaries of each volume slice. The difference between the two methods is where the triangle clipping op-
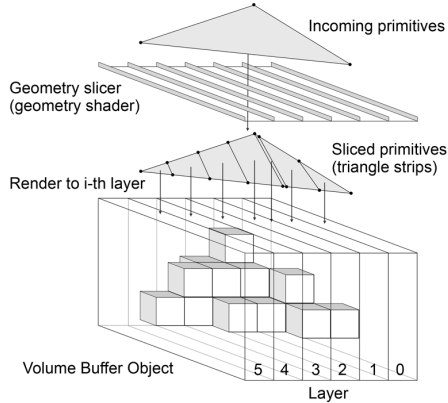
Figure 2: *Geometry shader triangle slicing. The incoming triangles are sliced into stripes and each stripe is rasterized into the associated layer. (See Algorithm 1)*

eration takes place. In the first method (see Section 3.1) each triangle is clipped against the current volume slice, in a geometry shader, allowing only the valid parts of the triangle to go through for rasterization. In the second method (see Section 3.2) each triangle is rasterized in each volume slice it intersects and the fragments are further clipped in the pixel shader.

The final volume is generated from the fusion of the three intermediate passes into a single multi-buffer by using the MAX blending operation. We substituted the OR operation commonly used in binary voxelization, as in our case, we deal with scalar data. Since all fragments have to be treated, face culling and z-test are disabled and hence no z-buffer is attached to the *frame buffer object*. All volume multi-channel attributes are computed and rendered simultaneously (e.g. occupancy, albedo, normals etc.) using *multiple*
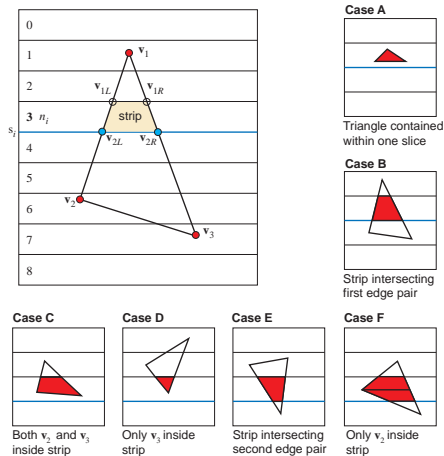

Figure 3: *The six possible triangle strip configurations with respect to the volume grid. (See Algorithm 1)*

Algorithm 1: Geometry Shader used for triangle slicing (Z-Pass). (ECS: Eye Coordinate Space)

Input: $v1$, $v2$, $v3$ - the $\triangle$ vertices
Data: z slice thickness (in volume sweep ECS)
Result: $\triangle$ sliced into stripes and rasterized into the appropriate volume layer. New $\square$ is emitted with generated vertices $v1L$, $v1R$, $v2L$ and $v2R$ per slice.

if $\triangle$ *not aligned with Z-axis* **then** return
sort vertices according to Z-axis.
layer ← minimum slice index for the first vertex
slice ← current slice depth in ECS
if $v3$ *depth is* $\geq$ *slice* **then**   CASE A
     Emit $\triangle$ $v1$, $v2$, $v3 \rightarrow$ layer
     return
if $v2$ *depth is* $\geq$ *slice* **then** $v1L \leftarrow v1R \leftarrow v1$
else $v1L \leftarrow v2$; $v1R \leftarrow v1$
$v2L$, $v2R \leftarrow$ Intersect Edges (slice)
Emit $\square$ $v1R$, $v1L$, $v2L$, $v2R \rightarrow$ layer
repeat
     slice += z thickness ; layer ++
     $v1L \leftarrow v2L$; $v1R \leftarrow v2R$
     if $v2$ *depth is* $\geq$ *slice* **then**   CASE B
         $v2L$, $v2R \leftarrow$ Intersect Edges (slice)
     else
         if $v3$ *depth is* $<$ *slice* **then**
             if $v2$ *depth was* $\geq$ *slice* **then**   CASE C
                $v2L \leftarrow v2$; $v2R \leftarrow v3$
             else   CASE D
                $v2L \leftarrow v2R \leftarrow v3$
         else
             if $v2$ *depth was* $<$ *slice* **then**   CASE E
                $v2L$, $v2R \leftarrow$ Intersect Edges (slice)
             else   CASE F
                $v2L$, $v2R \leftarrow$ Intersect Edges ($v2$ depth)
                Emit $\square$ $v1R$, $v1L$, $v2L$, $v2R \rightarrow$ layer
                $v1L \leftarrow v2L$; $v1R \leftarrow v2R$
                $v2L$, $v2R \leftarrow$ Intersect Edges (slice)
     Emit $\square$ $v1R$, $v1L$, $v2L$, $v2R \rightarrow$ layer
until $v3$ *depth* $<$ *slice*

---

render targets into corresponding volume buffers (see Figure 1).

We take advantage of the OpenGL extension for layered rendering. It allows a geometry shader to write to the build-in special variable gl_Layer thus enabling the rendering of primitives to arbitrary volume texture layers computed at run time and eliminating the multiple passes over the incoming data or the restriction to record only a binary volume representation in one pass.

3.1 Geometry Shader Triangle Slicing

To assign the geometry (or parts thereof) to the appropriate buffer layer (see Figure 2) we intersect each triangle with the
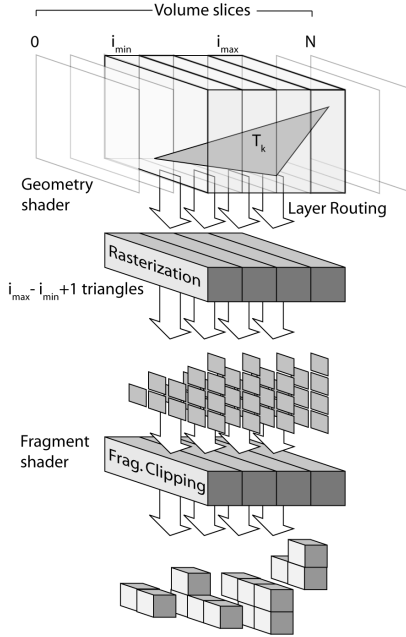
Figure 4: *Pixel shader clipping method. The Geometry shader rasterizes each triangle into all the volume slices it intersects and the Pixel shader discards the fragments based on their depth (See Algorithm 2).*

Eye Space Coordinates (ECS) of the volume slices of each axial sweep. If the triangle is aligned with the major axis of the specific pass, its vertices are sorted in ascending order. If the triangle is contained within one slice (Figure 3, Case A) the triangle is exported as is and the exit condition is met. Otherwise, we clip the triangle's edges against the planes perpendicular to the major axis (slice boundaries), producing a surface strip for each slice that the polygon intersects. At each step, we decide on the configuration of the triangle (see Figure 3) and whether a triangle split needs to occur or not. At each split, a quad-shaped triangle strip is being generated and rasterized to the appropriate volume layer (see Algorithm 1). The pixel shader is virtually empty, computing just the multi-channel data that an application requires.

3.2　Pixel Shader Fragment Clipping

The second algorithm is very simple as the geometry shader does not do any triangle clipping. Rather the method relies on fragment rejection in the pixel shader.

For each directional voxelization, each triangle in the scene passes from a geometry shader (see Algorithm 2) where, if it is aligned to the current sweeping direction, it is rasterized into *all* the volume slices that it intersects. The slice boundaries are computed in Normalized Device Coordinates (NDC) and passed to the pixel shader where fragments are *discarded* if their depth is outside these boundaries. The process is demonstrated in Figure 4.

Algorithm 2: Geometry and Pixel Shaders used for triangle rasterization (Z-Pass). (ECS: Eye Coordinate Space, NDC: Normalized device Coordinates).

Input: $v1, v2, v3$ - the $\triangle$ vertices
Data: z slice thickness and z volume min (in volume sweep ECS)
Result: $\triangle$ rasterized into the appropriate volume layer.

/* Geometry Shader */

flat out zMin, zMax // directed to pixel shader

if $\triangle$ *not aligned with Z-axis* **then** return

$sMin \leftarrow min\ triangle\ z - z\ volume\ min/z\ slice\ thickness$
$sMax \leftarrow max\ triangle\ z - z\ volume\ min/z\ slice\ thickness$

for *slice between sMin and sMax* **do**
　　zMin $\leftarrow$ min depth of *slice* in NDC
　　zMax $\leftarrow$ max depth of *slice* in NDC
　　layer $\leftarrow$ *slice*
　　Emit $\triangle$ $v1, v2, v3 \rightarrow$ layer

/* Pixel Shader */

flat in zMin, zMax

if *frag depth not between (zMin, zMax)* **then** discard
else write data to volume

4　Implementation

In order to create the data storage structure, we generate on the GPU a uniform spatial partitioning structure in real-time. For the voxelization, the user has the option to request several attributes to be computed and stored into floating point buffers for later use. Among them are surface attributes like albedo and normals but also dynamic lighting information and radiance values in the form of a compact spherical harmonics (SH) coefficients representation.

5　Performance and Comparisons

We have integrated the multi-channel voxelization algorithm in a real-time deferred renderer using OpenGL and GLSL. We tested our methods for multiple models and various voxel grid resolutions. The results, reported in the following tables, were obtained on an Intel Core i7 860 @ 2.80GHz with 8 GB of RAM and equipped with an nVIDIA GeForce GTX 285 GPU with 1 GB of memory.

We compare our two methods based on the number of vertices that a geometry shader can output as the running times can vary greatly even for small changes to the number of requested output vertices. The number of vertices emitted from the geometry shader triangle slicing method is $3 + 4n$ (we detect the emittance of a triangle and do not produce a degenerate quad) and from the pixel shader clipping method is $3n$, where n is the number of slices that a triangle spans.

We observe (see Table 1) that both methods have approximately the same running speed and produce the same num-

Model	Grid size	Grid actual	Memory (MB)	Geometry slicing vertices out				Pixel clipping vertices out				#voxels
				7	11	15	19	6	9	12	15	
Bunny 69451 triangles	64^3	$53 \times 64 \times 41$	1.06	1.74	1.79	2.03	2.51	1.13	1.15	1.28	1.58	5.3K
	128^3	$106\times128\times 82$	8.49	2.37	2.38	2.66	3.19	1.71	1.72	1.86	2.16	22K
	256^3	$213\times256\times165$	68.64	5.92	5.97	6.43	6.98	4.92	4.98	5.12	5.48	89.6K
	512^3	$425\times512\times330$	547.85	28.2	28.6	29.3	30.1	26.9	27.3	27.5	27.8	–
Sponza II 219305 triangles	64^3	$39 \times 27 \times 64$	0.51	3.99	4.03	4.52	5.38	2.88	2.91	3.24	4.01	20K
	128^3	$79 \times 54 \times128$	4.17	5.10	5.13	5.78	6.74	3.53	3.57	3.94	4.79	100K
	256^3	$157\times107\times256$	32.81	8.38	8.40	9.26	10.66	5.93	6.02	6.48	7.51	445K
	512^3	$315\times214\times512$	263.32	21.92	22.01	23.03	24.86	18.44	18.64	19.23	20.51	1980K
Dragon 871414 triangles	64^3	$64 \times 62 \times 29$	0.88	69.1	70.6	71.3	72.0	70.0	71.2	74.1	74.9	5.4K
	128^3	$128\times123\times 57$	6.85	75.4	75.8	76.0	76.3	75.1	75.4	75.8	76.2	22.5K
	256^3	$256\times247\times114$	55.00	77.1	77.5	77.8	78.3	76.9	77.3	77.6	78.0	93K
	512^3	$512\times493\times229$	441.00	88.1	88.7	89.4	90.3	88.3	89.1	89.6	90.4	–

Table 1: *Running time (in ms) for the construction of a half-float (16bit) single channel Occupancy Volume buffer for the two surface voxelization methods, based on the number of vertices that the geometry shader outputs. The third column gives the actual grid sizes as tight volume grids are generated dynamically. The last column reports the number of the resulting voxels.*

ber of voxels. The pixel shader clipping method achieves slightly better results but when the number of triangles that need to be processed increases (i.e. Dragon model) then the two methods are equivalent.

The quality of the voxelization depends on the number of volume slices each triangle spans. The smaller the limit of output vertices of the geometry shader, the higher the probability that the triangle will be partially sliced, resulting in empty voxels. However, due to the fact that triangles are selectively processed in the volume sweep plane of maximum projection, this is seldom the case.

Figure 1 depicts the multi-channel voxelization of the Knossos model, an open environment with no watertight surfaces (e.g. the ground).

In Figure 5 we compare the voxelization of the pixel shader clipping method for various geometry shader output vertices. If we request too few output vertices from the geometry shader (eg. 3 vertices) then holes start to appear in the voxelization. A higher output vertex count gradually remedies this issue. In many effects, such as in the case of diffuse global illumination (e.g. virtual point light injection), a high vertex output limit would not be necessary, since even a sparse or incomplete volume representation can still work satisfactorily due to the low-frequency nature of secondary diffuse light transport. On the contrary, fluid effects (e.g. water) require a higher vertex output limit in order to produce dense volumes as the granularity of the volume affects the simulation as a whole.

Apart from the occupancy buffer, where virtually no computations are involved, the construction speed of the rest of the buffers depends on the computations that are involved in their creation. Table 2 lists the minimum required time to

write to 1, 2 or 3 *multiple render targets* without performing any computations.

For the sake of comparative examination (see Table 3), we have implemented a modified version of the method by Fang et al. [FC00], which supports multi-channel data. The big difference in the running times is attributed to the number of passes that Fang et al. do over the geometry data which increases their timings especially for large models.

In addition we show the timing results for our implementation of Eisemann et al. [ED06] binary occupancy-only voxelization method. The running time is the fastest of all but their method cannot take into account partial occupancy or transparency.

6 Discussion

The decision for the choice of method depends mostly on the GPU architecture. Implementations for non-unified architectures may favor the geometry shader approach (see Section 3.1), if the pixel shader cores are intensively used and vice versa. For unified architectures, the expected load in terms of triangle count is indicative of the best approach.

Grid size	Geometry slicing 15 vertices output MRTs used			Pixel clipping 9 vertices output MRTs used		
	1	2	3	1	2	3
64^3	2.36	2.54	2.70	1.33	1.33	1.52
128^3	3.22	4.12	5.18	2.90	2.90	3.96
256^3	10.2	17.4	25.1	15.8	15.9	24.3

Table 2: *Comparison of the running time (in ms) for the bunny model for a floating (32bit) four channel buffer and different sizes of multiple render targets (MRT).*

Figure 5: *Comparison of the voxelization using the pixel shader clipping method at 256^3 volume resolution with 3 and 9 geometry shader vertices output. The number of voxels produced are 52382 and 89696 (complete voxelization) respectively.*

Furthermore, certain GPU implementations do not favor the execution of complex geometry shaders with large output primitive streams. Our pixel shader approach (see Section 3.2) is very simple to implement but produces a lot of fragments in the geometry shader. These are rejected in the pixel shader but on architectures with small bandwidth, this could be an issue. It is a reason to favor the first method which produces exactly the fragments that are going to be rasterized in the final volume slices.

For scenes with slow or gradual animations, the three directional voxelization steps could be interleaved, recalculating only one axis pass in each frame, further reducing the volume buffer creation time by a factor of 3.

7   Conclusion

We have presented two methods for surface voxelization of dynamic scenes. The two strong points of the methods are the ability to generate multi-channel data at high per-

Model	Grid size	Fang et al.		Eisemann et al.	
		Time (ms)	**#voxels**	**Time (ms)**	**#voxels**
Bunny (69451 tris)	64^3	20.3	5.5K	0.171	2.1K
	128^3	40.8	22.2K	0.174	18.6K
	256^3	83.5	90.3K	0.21	145.4K
	512^3	181	–	0.61	1124K
Sponza II (219305 tris)	64^3	77.4	20.2K	0.73	6.6K
	128^3	155	102K	1.09	60K
	256^3	310	452.3K	2.53	408.3K
	512^3	629	2090K	7.11	3283K
Dragon (871414 tris)	64^3	3206	5.7K	35.1	1.5K
	128^3	7710	23.2K	35.3	11.2K
	256^3	–	94.5K	36.5	91.9K
	512^3	–	–	38.8	739.6K

Table 3: *Comparison of the running time and voxels produced by different approaches.*

formance and their simplicity in implementation and integration into existing frameworks in order to create anything from effects like smoke/fluid simulations, visibility computations or global illumination effects.

8   Acknowledgments

References

[CF98]   CHEN H., FANG S.:   Fast voxelization of three-dimensional synthetic objects. *Journal of Graphics Tools 3*, 4 (1998), 33–45. 2

[CLT07]   CRANE K., LLAMAS I., TARIQ S.: Real-time simulation and rendering of 3d fluids. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley Professional, 2007, ch. 30, pp. 723–739. 1, 2

[DCB*04]   DONG Z., CHEN W., BAO H., ZHANG H., PENG Q.: Real-time voxelization for complex polygonal models. In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference* (Washington, DC, USA, 2004), PG '04, IEEE Computer Society, pp. 43–50. 2

[ED06]   EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), I3D '06, ACM, pp. 71–78. 2, 5

[FBP09]   FOREST V., BARTHE L., PAULIN M.: Real-time hierarchical binary-scene voxelization. *Journal of Graphics, GPU, and Game Tools 14*, 3 (2009), 21–34. 2

[FC00]   FANG S., CHEN H.: Hardware accelerated voxelization. *Computers & Graphics 24*, 3 (2000), 433–442. 2, 5

[KPT99]   KARABASSI E.-A., PAPAIOANNOU G., THEOHARIS T.: A fast depth-buffer-based voxelization algorithm. *Journal of Graphics Tools 4*, 4 (1999), 5–10. 2

[PMP10]   PAPAIOANNOU G., MENEXI M. L., PAPADOPOULOS C.: Real-time volume-based ambient occlusion. *IEEE Transactions on Visualization and Computer Graphics 16* (September 2010), 752–762. 1

[PTTK07]   PASSALIS G., THEOHARIS T., TODERICI G., KAKADIARIS I. A.: General voxelization algorithm with scalable gpu implementation. *Journal of Graphics, GPU, and Game Tools 12*, 1 (2007), 61–71. 2

[SS10]   SCHWARZ M., SEIDEL H.-P.: Fast parallel surface and solid voxelization on gpus. In *ACM SIGGRAPH Asia 2010 papers* (New York, NY, USA, 2010), SIGGRAPH ASIA '10, ACM, pp. 179:1–179:10. 2

[THGM11]   THIEDEMANN S., HENRICH N., GROSCH T., MÜLLER S.: Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 103–110. 1, 2

[ZCEP07]   ZHANG L., CHEN W., EBERT D. S., PENG Q.: Conservative voxelization. *Vis. Comput. 23* (August 2007), 783–792. 2