Complete Polygonal Scene Voxelization

D. Haumont and N. Warzée
IDS - Information and Decision Systems
Université Libre de Bruxelles

Abstract

We present a fast and robust tool for automatically converting complete polygonal scenes into volumetric representations. A wide range of scenes are handled by storing the status (inside/outside) of the volumetric space areas in the cells of an octree. The algorithm first looks for a point in the scene for which the status can be univocally determined. When such a point is found, it propagates its status to the surrounding visible cells. This iterative two steps procedure is repeated for different parts of the scene until the status of all the cells has been determined. The algorithm's advantage is the coherence with the rendered geometry. Due to this fact, the approach is able to deal with complex geometry and exhibits robust solutions for a broad range of scenes containing numerous artifacts, such as cracks, holes, overlapping geometries, interpenetrating meshes, double walls and fuzzy borders.

1 Introduction and Previous Work

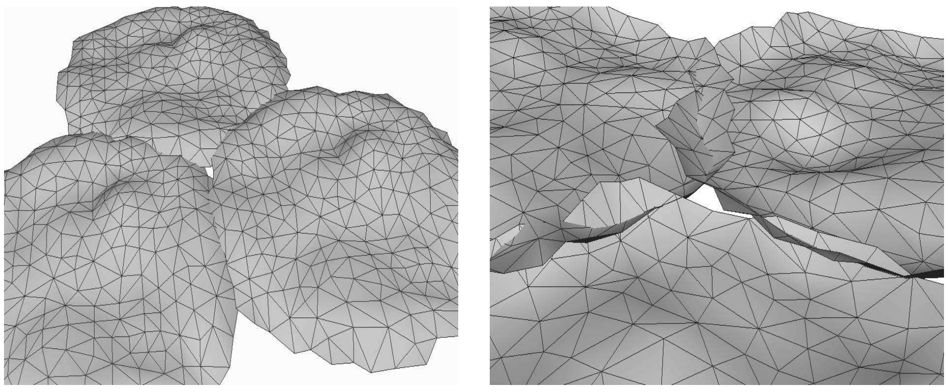
Voxelization consists in converting polygonal models to discrete 3D voxel grids. This conversion provides with complementary data structures very useful for many algorithms: model simplification [1][2], model repair [3][4], visibility determination [5], 3D morphing [6], volume visualization [7] and collision detection [8]. In [9], Taosong He goes further and suggests to use only volumetric techniques to design a complete volumetric environment.

Algorithms converting single polygonal model have been successfully developped [10], and some can handle models containing many defects. Among them, Nooruddin proposes in [3] to repair single models including cracks, holes, T-joints, double walls and self intersections via two different voxelization processes called ray counting and ray stabbing. In [4], Kolb and John give an implementation supporting standard OpenGL hardware acceleration. Their method implicitly supposes that the sampled object is surrounded by outside space, and renders the object geometry several times from different exterior points of view. The z-buffer information is then used to compute the voxelization.

Unfortunately, none of these previously published algorithms are suitable to voxelize complete scenes with fuzzy borders, for which the voxelized geometry is not surrounded by outside space. We propose here a general algorithm to convert complex large scenes handling a wide variety of artifacts and fuzzy borders. Our method can be seen as an extension of the previous ones [3][5]. It is designed to support the scenes that can be found in current computer games.

2 Input Scenes and Design Choices

This work was done in collaboration with the computer games company Appeal, which was working on the game "Outcast 2". For this game the company did not want to restrict the artists' liberty in level design by dictating restrictive modeling rules. The scenes designed consist of several hundred thousand polygons. Because these scenes do not have any topological restriction, the major difficulty comes from their simple polygon soup geometry. It is essential for this case to have a robust voxelization process that is able to deliver a complementary volumetric data structure useful for many algorithms. Unfortunately, their scenes have a lot of degeneracies principally coming from the modeling process based on geometry instanciation: some parts of the scene are made from different basic objects, replicated at different locations with scale factors and distortion parameters. Even if instanciation is an elegant way to reduce the memory cost, it introduces artifacts such as interpenetrating geometry, double walls and holes (cf. figure 1).



Front view | Back View

Figure 1: Degeneracies caused by instanciation. Note the patches interpenetration and the hole due to bad patches positioning.

Our voxelization algorithm was designed with an aim of generality and robustness:

- Generality because input scenes are very heterogeneous: indoor, outdoor and a mixture of the two.

- Robustness because of the artifacts: cracks, holes, overlapping geometries, interpenetrating meshes, double walls and fuzzy borders.

Fuzzy borders indicate that the borders of the scene have not been completely modelled. Outdoor scenes terminated by far away mountains are good examples, the invisible part of the scene vanishing in a confusing way (cf. figure 2).

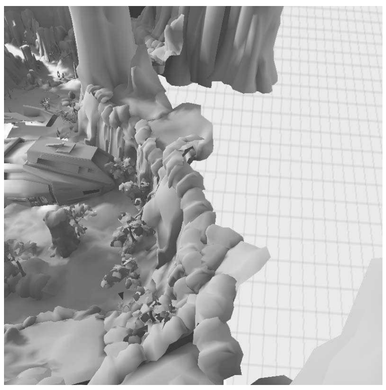


Figure 2: Landscape from Outcast 2 showing fuzzy borders.

In this case, ray counting and ray stabbing cannot be used: a ray traced through the geometry cannot be supposed to finish in an "outside" location, and a test ray that starts outside the model does not necessarily have an even number of intersections with the geometry.

The only sure postulate is the orientation of all polygonal faces of the initial model. Our algorithm relies on this constraint.

3 Voxelization Process

We consider here a solid volumetric representation, where each voxel has a density value of zero or one. A value of one represents a voxel entirely inside the matter, called an 'inside cell'. A zero value represents a voxel entirely outside the matter, called an 'outside cell'. In order to deal with large scenes, the volumetric representation is stored in an octree instead of the traditional full resolution voxel grid. After the creation of this octree, the second step is answering the question: "are the leaf cells of the octree totally inside or totally outside?". The general idea is to propagate the status of a cell, called the seed cell, to all the cells visible from this one. This approach is possible because we use homogeneous cells (called 'pure cells') as seeds, and two mutually visible cells will always be on the same side of the geometry (cf. figure 3) [1].

[1]This is sufficient but not necessary: two cells on the same side of the geometry are not always mutually visible.

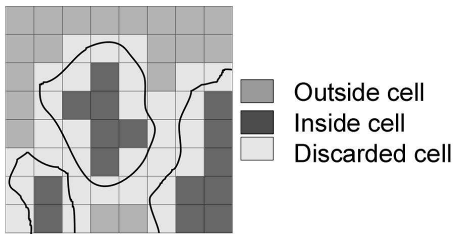


Figure 3: Pure cells utilisation. Inhomogeneous cells are discarded: other mutually visible cells are on the same side of the geometry.

The determination of the seed cell status is based on the triangles orientation of the initial geometry. The section 3.1 gives an overview of the voxelization algorithm and each part of it is detailed in the sections that follow.

3.1 Algorithm Overview

First of all, the octree data structure, which will be used to store the volumetric information, is created (*OctreeCreation* algorithm §3.2). The status of the cells are left undetermined. These statuses are determined in the second phase, which begins with the choice of a seed cell (*SeedChoice* algorithm §3.4.4). When a candidate seed has been found, its status is fixed by the *SolveStatus* algorithm (§3.3) which also computes a confidence value that represents the certainty of the determination. When this value shows that our determination is safe enough, the cell can be used as a seed for the *PropagateStatus* algorithm (§3.4.2). Otherwise, the algorithm looks for another seed cell. These solving processes are iterated until all the octree cells are known (cf. figure 4).

3.2 *OctreeCreation* algorithm

The creation of the volumetric octree requires finding the portions of space that do not contain any geometry triangles. In practice, these regions are the cubic cells of the future octree. The maximum depth of the octree fixes the sample resolution of the voxelization process. If the smallest cell has a size s, the smallest geometry detail that will be certainly sampled is of size 2*s.

To speed up the creation process, the input scene is stored during a preprocess in a binary axis-aligned bounding box (AABB) tree, whose leaves contain the scene geometry. This tree is called “InputTree”.

The root of the volumetric octree is the smallest cube containing the whole scene. Each octree cell is tested against the InputTree’s hierarchy to see if it contains any geometry. The bounding boxes of the InputTree’s nodes are used before testing the triangles themselves. In most cases, it is not even necessary to test any triangle in order to determine the status of the cell. When necessary, the fast AABB-triangle intersection test from [11] is used. The cubes containing

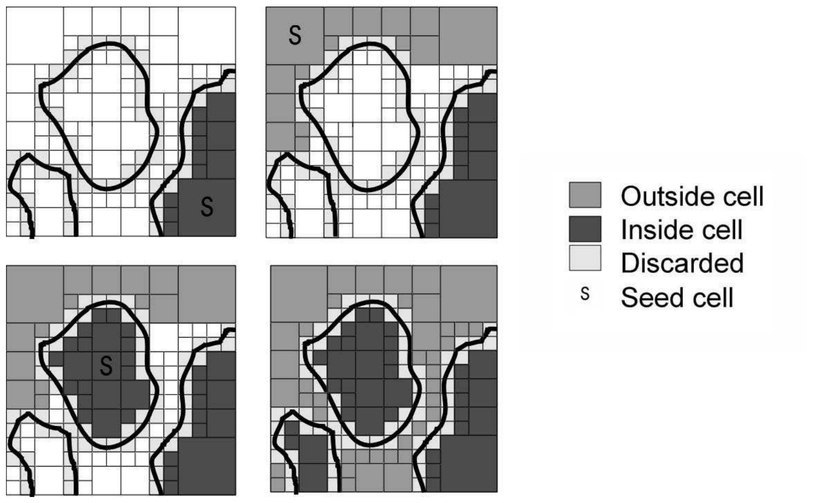


Figure 4: Different Steps of Propagation. After a seed has been chosen, its status is determinated and is propagated to the visible cells. The first three images show the beginning of the process while the last image represents the final result.

geometry are refined until the maximum depth is reached. Leaf cells of maximum depth still containing geometry triangles are discarded, since the rest of the algorithm cannot handle them (cf. §3.3.2).

3.3 *SolveStatus* algorithm

Let us first suppose that there are no holes and no hanging edges in the geometry. The determination of the status of the cell is based on triangle orientation: a point outside the geometry can only see front-facing triangles. However, an inside point can also see front-facing faces due to the intersecting geometries. The following criteria are therefore used:

- A point is inside if one (or more) back-facing triangle is visible
- A point is outside if every visible triangle is front-facing

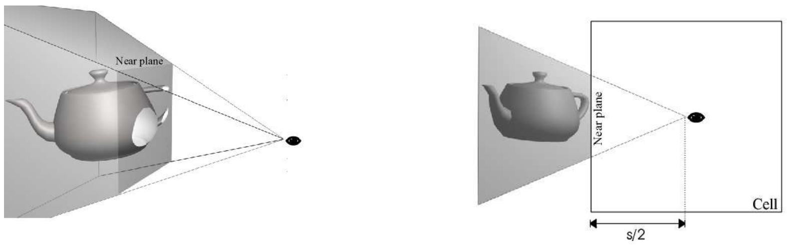
The determination of the visible triangles uses OpenGL renderings to take advantage of hardware acceleration. For this purpose, two sided rendering is introduced: back-facing triangles will be drawn in red, and front-facing triangles in blue (see Appendix for details). A camera with a 90-degree field-of-view is positioned in the center of the cell. To create a cube map, the polygonal scene is rendered in the six orthogonal directions by two sided renderings. The renderings are sped up by a hierarchical view-frustum culling of the AABB InputTree [12]. If there is one (or more) red pixel in the cube map, the cell is classified as inside. Otherwise, the cell is classified as outside. Of course, each face of the cube map is drawn, read back and tested successively, and the discovery of a red pixel allows us to skip the processing of the remaining cube map faces.

3.3.1 Hole Handling

The finding of a small number of red pixels in the cube map is not sufficient to classify the cell as being inside the matter when the geometry of the scene has some holes. These pixels might come from a hole through which the camera sees back-facing triangles. That is the reason why the number of red pixels found in the cube map must be greater than a given threshold to classify a cell as inside. To cope with bigger holes, this condition was further strengthened by imposing that inside cells have a minimum number of red pixels in several opposed faces of the cube map.

3.3.2 Positioning the near plane of the camera

If there is some geometry between the viewpoint of the camera and the near plane, the algorithm produces false results: the geometry is clipped and the camera sees faces that would have been hidden by the removed geometry (cf. figure 5). Thanks to the creation process, the octree cells still containing geometry have been refined (or discarded). Because the seed cells are always pure, the clipping problems can be avoided by always placing the near plane inside the cell. To help avoid precision problems, the near plane is set as far as possible, at a distance s/2 where s is the size of the cubic cell.



Near plane too far away Correct positioning

Figure 5: Positioning of the near plane.

3.3.3 Confidence

The confidence term is introduced to avoid propagating erroneous information due to local degeneracies, such as large holes in the geometry. Every cell is not allowed to become a seed cell, and the *SolveStatus* algorithm must return a sufficient confidence term to allow the *PropagateStatus* algorithm to begin.

This confidence test is based on the number of faces of the cube map on which red pixels are visible. An inside cell must see red pixels on at least 4 of the faces to be allowed to propagate its status. An outside cell cannot see any red pixels in order to be able to propagate its status.

3.4 Propagation process

3.4.1 Marking procedure

To immediately know which part of the octree remains unknown and has to be processed by the propagation algorithm, cells are marked when their status is known:

- A leaf cell is marked as known as soon as its status has been fixed.
- An intermediate cell is marked as known when all its children cells are determined.

This marking process enables to skip entire subparts of the octree when they have already been successfully treated.

3.4.2 ***PropagateStatus*** algorithm

The *PropagateStatus* algorithm is quite similar to the *SolveStatus* algorithm and makes use of OpenGL renderings to propagate the status of a seed cell to its surrounding visible cells (cf. algorithm 1). A camera is placed at the center of the cell and used to draw the scene into a cube map with rendering options disabled (back-face culling, blending, texturing and lighting). The six corresponding depth maps are then extracted and will be used by the *Octree-Propagation* algorithm to propagate the status of the seed cell to other octree cells.

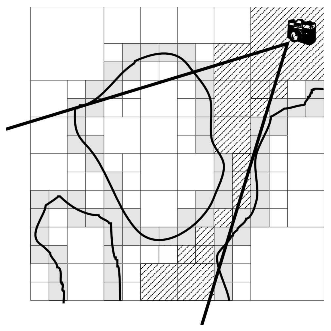


Figure 6: Propagation Mechanism. The cells acquiring the seed status with this camera orientation are shaded in gray.

In the *OctreePropagation* algorithm, the volumetric octree is traversed recursively for each of the six view directions, stepping only into the nodes not yet marked as known (cf. algorithm 1). The projection of each leaf cell's vertices is calculated and its depth is compared with the corresponding value in the z-buffer: if one of the projected vertices has a smaller depth than the stored depth, the cell is said to be visible from the seed cell and acquires its status. Otherwise, the status of the projected cell remains unknown (cf. figure 6). Each leaf cell that has acquired the status of the seed is marked as known and will not be treated again (cf. algorithm 2).

Algorithm 1 Propagation algorithms (1)

```
function PropateStatus (Octree, InputTree, Status, Seed)
var D: Direction
     Buffer: array of float
{
repeat
   D := Iterate_On_Orthogonal_Directions;
   PositionCamera (Seed_center, D);
   Render (InputTree);
   ReadZBuffer(Buffer);
   OctreePropagation (Octree, Buffer, Status);
until All_The_Cube_Map_Is_Treated
}
function OctreePropagation (Octree, Buffer, Status)
var Current: Volumetric_Octree_Node
{ Current:= Octree;
if (Is_Leaf (Current)) then
   LeafHandling (Current, Buffer, Status);
else
   repeat
      Current := Iterate_On_Children_Node (current);
      if (!Is_Already_Known(Current)) then
         OctreePropagation (Current, Buffer, Status);
      end if
   until ( Current = nil);
   if All_Children_Are_Known(Octree) then
      Mark_As_Known(Octree);
   end if
end if
}
```

If there is a hole in the geometry, the propagation process could propagate its status through the hole, and would therefore result in a false prediction. A counter-mechanism avoids this problem: each cell must be classified several times as inside (or outside) before acquiring its final status. The projection process mimicks a rendering process, so the recursive traversal of the octree can also benefit from view-frustum culling.

Algorithm 2 Propagation algorithms (2)

```
function LeafHandling (Cell, Buffer, Status)
var
    Depth: Float
    x, y : Integer;
    V: Vertices
{ V := First_Vertex (Cell);
repeat
  (x,y, Depth) := Project (V);
  if (Depth < Buffer [x,y]) then
    Increment_Inside_Or_Outside_Counter (Cell, Status);
    if ((Inside_Counter > Threshold) or (Outside_Counter > Threshold))
    then
      Give_Status_To_Cell (Status, Cell);
      Mark_As_Known(Cell);
      return;
    end if
  end if
  V:=Iterate_On_Vertices (Cell)
until (V = nil);
}
```

3.4.3 Mixing *SolveStatus* and *PropagateStatus* algorithms

It is possible to interlace the *SolveStatus* and the *PropagateStatus* algorithms, in order to reduce the total number of renderings: depth maps used in the *PropagateStatus* could be extracted during the *SolveStatus* algorithm. Unfortunately, there is an important overhead when some cells do not have a sufficient confidence level: the *PropagateStatus* algorithm will not be launched and the depth maps will be discarded, losing the benefit of "pre-reading" them. This might be problematic when the reading back of the z-buffer is time consuming, as it is the case with the current graphic cards. That is the reason why it has been chosen to render the geometry twice and to access the z-buffer only when necessary.

3.4.4 *SeedChoice* algorithm

The seed choice is of great importance when using a propagation algorithm because the speed of convergence is closely linked to this choice. In our case, a good heuristic would be to choose a seed cell seeing a lot of other cells, in a fully undetermined region, rather than an isolated cell that has a poor chance of propagating its status to other ones. That is why the largest cells are selected first because their center, where the camera is positioned, is far away from any geometry, increasing so the probability to see many other cells. The second criterion used to choose among several cells of equal size is based upon the density of volumetric information already computed in the area close by the cells. This can be approximated by assigning a weight to each candidate cell. Before launching the propagation algorithm, the volumetric octree is traversed recursively to assign a weight to each node. The weight of a leaf is set to one if the cell is unknown and to zero if known. For nodes with children the weight is the sum of the weights of the children. To compute the weight of the seed cell, the path from the root to the seed cell is followed, and the weights of all encountered nodes are summed. This sum is a good indicator of the number of undetermined cells geometrically located near the seed cell. The selected seed is the cell with the maximum weight.

4 Results

The algorithms described above were used to voxelize several scenes from the Outcast 2 game. All the experiments were done with a computer with a Pentium III processor (800 Mhz) and a GeForce 2 GTS graphic card. These results must be taken as a feasibility demonstration of our algorithms, and do not intend to be an in-depth study. The references times are only given as indicative values.

The first scene represents an architectural model of a house consisting of 28291 triangles. It was sampled with a 34 m wide octree of depth 8. The second scene represents a landscape on which a space ship has crashed. The scene consists of 112426 triangles. It was sampled with a 256 m wide octree of depth 9. The third scene represents a stone landscape containing architectural models and underground rooms (cf. figure 7, figure 8 and figure 9). The scene consists of 464143 triangles. It was sampled with a 537 m wide octree of depth 10.

The total voxelization time, its distribution between modules and additional data are summarized in the table below.

Time Elapsed	Scene 1 (depth 8)	Scene 2 (depth 9)	Scene 3 (depth 10)
OctreeCreation	39%	51%	63%
SolveStatus	7%	7%	4%
SeedChoice	25%	25%	20%
PropagateStatus	29%	17%	13%
Total Time	8 min.	37 min.	45 min.

Number of renderings	Scene 1	Scene 2	Scene 3
SolveStatus	4822	13736	9978
PropagateStatus	11455	27551	29796

Even if the algorithm was designed for complete scenes, our method can voxelize different portions of the scene independently. Instead of converting the whole scene in one session, it is possible to quickly convert smaller regions of interest: an octree of depth 6 is precise enough to sample the geometry details and can be calculated in a few seconds.

5 Discussion

The total time of voxelization is very dependent on the configuration and the quality of the input scene. The configuration is important because the propagation algorithm will sample much faster open scenes with large visibility areas than intricate architectural models. The quality must be taken into account because scenes without degeneracies can be treated with smaller thresholds and relaxed conditions of propagation. In contrast highly degenerated scenes must be voxelized with high thresholds and strengthened propagation conditions. An in-depth study of the parameters (seed choice, threshold, classify conditions) would be useful as future work.

Thanks to the use of graphic hardware and renderings, the total number of scene triangles does not affect the algorithm performance if visibility culling techniques are used (e.g view-frustum culling and/or PVS).

6 Conclusion

Voxelization is a powerful graphic tool because it provides volumetric information which can be valuable for many algorithms, including those for visibility and simplification. We introduced a very general voxelization process, designed to sample complete polygonal scenes containing many artifacts, as can be found in current computer games. The advantages of our method are robustness and generality, since no restriction is made on the input scene which can just be a polygon soup. We have shown that this algorithm is able to sample complex heterogeneous scenes containing many artifacts and hundreds thousands of triangles.

7 Acknowledgments

This research is supported by a grant of the Walloon Region (Belgium) and Appeal s.a.. We would like to thank C. Chaudy, X. Baele, V. Henriet, O. Debeir and P. Van Ham for all their help and advice.

8 Appendix: two sided rendering

Two sided rendering can easily be done with standard OpenGL features. After having enabled the back-face culling capabilities of OpenGL
(glEnable (GL_CULL_FACE)), we draw the scene twice. The first time, the culling parameter is set to cull back-facing triangles (glFrontFace(GL_CCW)) and the scene is drawn in blue. Then it is drawn a second time in red, with the culling parameter set to cull front-facing triangles (glFrontFace(GL_CW)).

9 Web Information

The figures from this paper are available online at:
http://www.acm.org/jgt/papers/HaumontWarzee02

References

[1] C. Andújar, *Octree-based Simplification of Polyhedral Solids.* PhD thesis, Universitat Politecnica de Catalunya (Barcelona, Spain), 1999.

[2] T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang, "Voxel-based object simplification," *Proceedings of IEEE Visualization*, pp. 296–303, October 1995.

[3] F. Nooruddin and G. Turk, "Simplification and repair of polygonal models using volumetric techniques," *Technical Report GITGVU-99-37*, 1999.

[4] A. Kolb and L. John, "Volumetric model repair for virtual reality applications," *Proceedings of Eurographics 2001, Short presentations.*

[5] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion, "Conservative volumetric visibility with occluder fusion," *Proceedings of SIGGRAPH 2000*, pp. 229–238.

[6] J. Gomes, L. Darsa, B. Costa, and L. Velho, *Warping and Morphing of Graphical Objects.* Morgan Kaufmann Publishers, 1998.

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

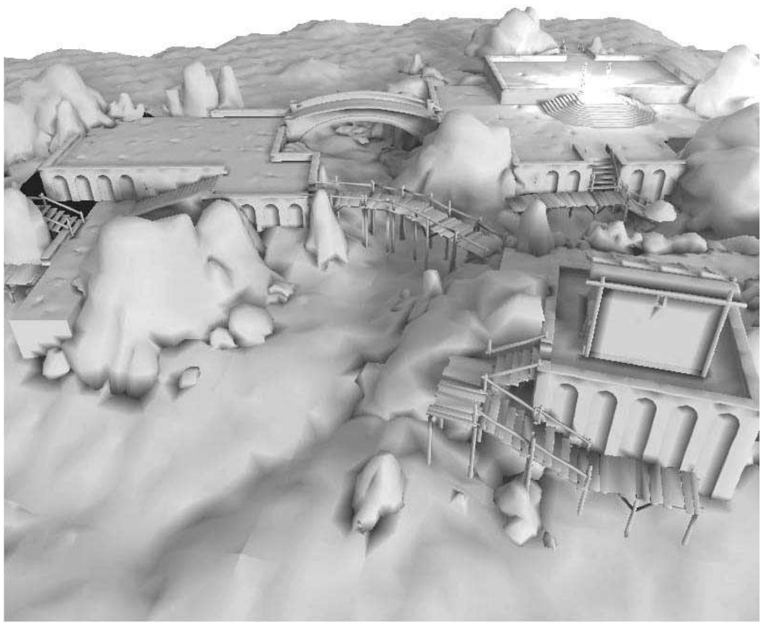
[8] S. F. Gibson, "Beyond volume rendering: Visualization, haptic exploration, and physical modeling of voxel-based objects," *Technical Report TR95-04*, January 1995.

[9] T. He, *Volumetric Virtual Environments.* PhD thesis, State University of New York at Stony Brook, 1996.

[10] E.-A. Karabassi, G. Papaioannou, and T. Theoaris, "A fast depth-buffer-based voxelization algorithm," *Journal of Graphics Tools*, vol. 4, no. 4, pp. 5–10, 1999.

[11] T. Akenine-Möller, "Fast 3d triangle-box overlap testing," *Journal of Graphics Tools*, vol. 6, no. 1, pp. 29–33, 2001.

[12] U. Assarsson and T. Möller, "Optimized view frustum culling algorithm for bounding boxes," *Journal of Graphics Tools*, vol. 5, no. 1, pp. 9–22, 2000.



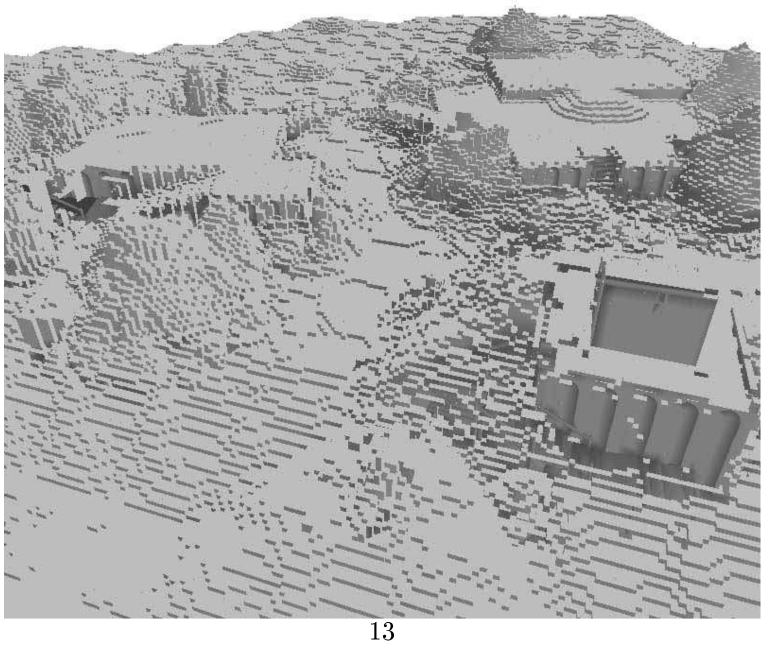


Figure 7: Voxelization of the test scene 3: polygonal scene (top), volumetric representation (bottom).

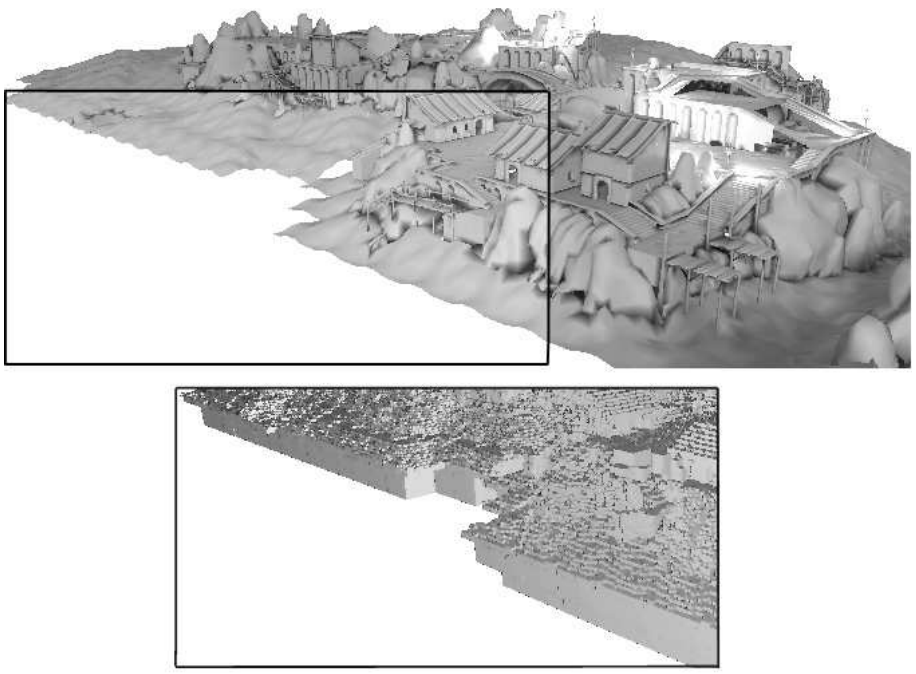


Figure 8: Fuzzy border handled by the algorithm.

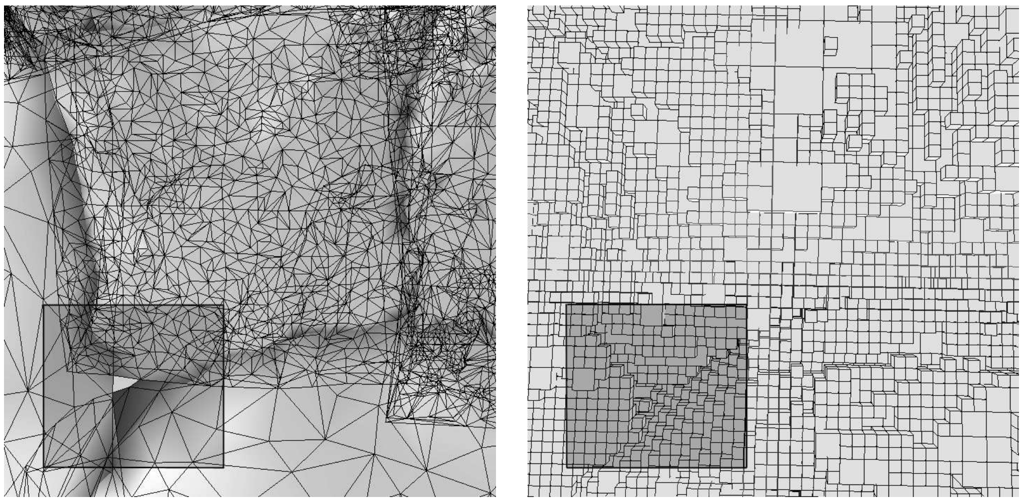


Figure 9: Overlapping geometry and a hole handled by the algorithm.