Point-Tessellated Voxelization

Yun Fei*
Tsinghua National Laboratory for Information Science and Technology, P. R. China

Bin Wang†
School of Software, Tsinghua University, P. R. China

Jiating Chen‡
Department of Computer Science and Technology, Tsinghua University, P. R. China

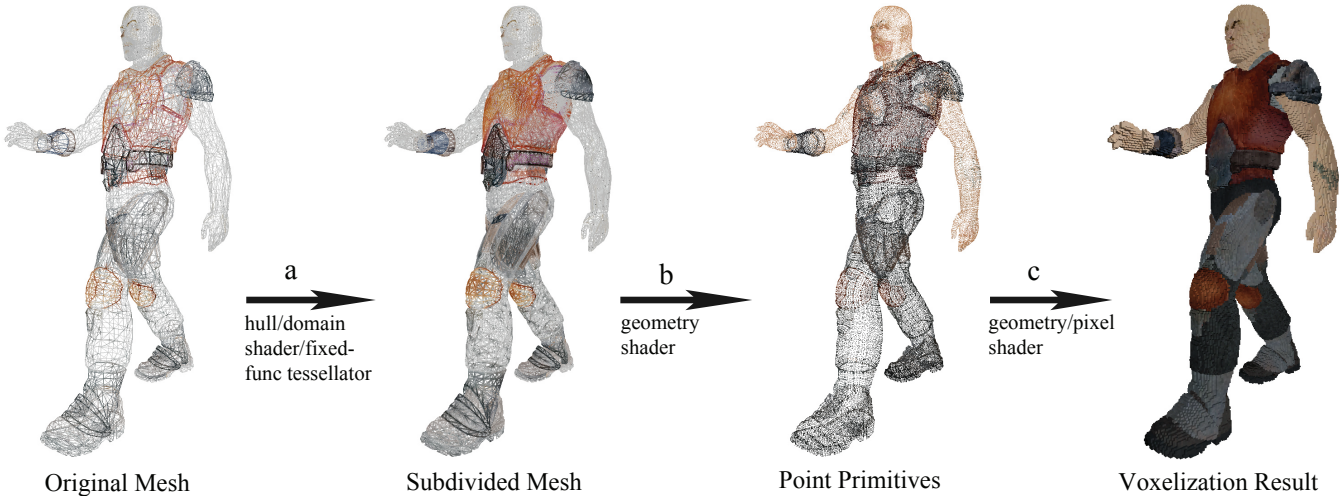


Figure 1: Different stages in our GPU-based framework for real-time multi-valued surface voxelization (which costs $0.46ms$ for resolution at 128^3, and $9.10ms$ for resolution at 512^3, with $22.1k$ triangles divided in 5 subsets). Only one rendering pass is required.

Abstract

Applications such as shape matching, visibility processing, rapid manufacturing, and 360 degree display usually require the generation of a voxel representation from a triangle mesh interactively or in real-time. In this paper, we describe a novel framework that uses the hardware tessellation support on the graphics processing unit (GPU) for surface voxelization. To generate gap-free voxelization results with superior performance, our framework uses three stages: *triangle subdivision, point generation,* and *point injection.* For even higher temporal efficiency we introduce PN-triangles and displacement mapping to voxelize meshes with rugged surfaces in high resolution.

Our framework can be implemented with simple shader programming, making it readily applicable to a number of real-time applications where both development and runtime efficiencies are of concern.

Keywords: voxelization, tessellation, surface subdivision, PN-triangles, point-based rendering, real-time rendering.

Index Terms: Computer Graphics [I.3.5]: Computational Geometry and Object Modeling—Curve, surface, solid and object representations

*e-mail: fyun@acm.org
†e-mail: wangbins@tsinghua.edu.cn
‡e-mail: chenjt04@gmail.com

1 Introduction

Applications such as shape matching (e.g., collision detection [15]), visibility processing (e.g., ambient occlusion [20] and global illumination [23]), rapid prototyping design (for 3D printing or laser sintering) [25], and 360 degree display [12] often require real-time computation of voxel representations from meshes via a technique called voxelization. Conservative voxelization, the most popular variant, uses all the voxels that overlap with one or more surfaces of the mesh.

State-of-the-art methods for conservative voxelization [19, 21] use the triangle-voxel overlapping test to decide whether a voxel is covered by a triangle. These methods provide fast and accurate results but may introduce computational overheads when the mesh is of high resolution and the voxel grid is of relatively low resolution. This situation would happen, for instance, when voxelizing the Asian Dragon mesh into a 128^3 grid (see the statistical data about Figure 10b). In a mesh whose triangles are all much smaller than the size of a voxel, usually only a tiny number of triangles straddle a voxel border, to cover more than one voxel. In this context, we should notice two facts. Firstly, for most triangles that do not straddle voxel borders, the voxels containing them can be determined from the centroids of those triangles. Secondly, in the case of binary voxelization, a voxel can be marked as "covered" as long as there is one triangle covering the voxel, but in the case mentioned above, hundreds of triangles contained in a single voxel are tested to determine whether the voxel is "covered".

Naive pipeline-based voxelization would only consider whether the center of a triangle is in a voxel. This method is suitable for the special case mentioned above, but cannot be generalized since, typically, there are always some triangles that occupy several voxels. Therefore, sampling more points on a triangle is necessary.

In recent years, graphics processing units (GPUs) have come to

support real-time tessellation; the latest model takes four clock cycles to tessellate one triangle [24]. Tessellation is a simple and efficient way to sample a large number of points on a triangle (and with a carefully selected tessellation factor, the sampled points can be uniformly distributed [14]) currently used for subdividing a patch into smaller shapes. Specifically, our tessellation algorithm implemented in Direct3D 11 or OpenGL 4.0 uses three hardware stages: the *hull shader*, which generates control points, the *fixed-function tessellator*, which generates barycentric coordinates for the tiled triangles, and the *domain shader*, which generates new vertices according to the barycentric coordinates and the control points.

In this paper, we propose a simple algorithm for real-time surface voxelization. The size of the resulting voxel grid is only limited by the size of the video memory. We target systems where accuracy is not a stringent requirement, such as video games or virtual reality applications. Hence, we relax the requirement from conservative to watertight (or gap-free) voxelization. With this relaxation, we can dramatically improve the efficiency by avoiding the triangle-voxel overlapping test. Gap-free voxelization is ensured when each triangle is smaller than its destination voxel. As shown in Figure 1, we achieve this feature by subdividing the mesh with a GPU hardware tessellator (a), placing a point primitive on the centroid of each subdivided triangle (b), and rasterizing these points into a volume texture (c). In our framework, the voxelization can be recomputed over the entire grid in each frame; thus, arbitrary deformation is supported.

While higher performance can be achieved by voxelizing a low-resolution version of the triangle mesh, this would introduce visible artifacts. To maintain the quality of voxelization while achieving higher performance, we develop an extension called *displacement voxelization*. Displacement mapping [3] is introduced after the triangle subdivision process, and the positions and normals of PN-triangles are used to reconstruct the details of the mesh. The result of displacement voxelization on a low-resolution mesh is visually comparable to the result of conventional voxelization on a high-resolution mesh, and has superior performance.

We summarize the key features of our framework as follows:

Efficiency: As shown in Figure 10, our approximated approach is demonstrated to have low computational overheads in most cases. We compared it with the state-of-the-art approaches including the pipeline-based technique [27] and methods using customized rasterizers based on Compute Unified Device Architecture (CUDA) [19, 21].

Parsimony: This is the most prominent feature of our framework, making it easily integrable into a general system using common graphics APIs, such as Direct3D or OpenGL. Moreover, our technique can be implemented with only a few modifications from basic tutorials/documentations on hardware tessellation [6, 22], which makes our technique easier for developers who are not knowledgeable in CUDA, or who are working on a rapid prototype that needs real-time voxelization.

Generality: Existing applications of voxelization need to store either one bit or multiple values in a voxel. Our method supports both requirements, providing flexibility for different applications.

2 Related Work

Numerous solutions have been developed for real-time voxelization. A multi-pass method proposed by Fang et al. [11] renders the geometries once for each slice of the volume while limiting the depth of view to the boundary of the slice. In modern hardware, this can be done in one pass by slicing the instanced objects in a geometry shader [4]. However, these slice-based techniques are limited to simple meshes (since for 1024 layers/slices, the mesh can be instanced 1024 times, which means culling and rasterizing the mesh 1024 times). Moreover, the number of instances is limited by current hardware. Similar to our work, Li et al. [16] propose scattering point primitives into a voxel grid but employ depth peeling to sample points from geometries, a process that is costly for scenes with complex surfaces, and suffers from missing thin features.

Another branch of real-time solutions [8, 9, 10] encodes binary voxels in separate bits of multiple render targets, and voxelizes the scene with bitwise blending. The common issue with these methods is that one axis of the grid is restricted to a bit-length of 1024, which means that only 1 bit can be generated per voxel for a 1024^3 grid. Furthermore, bitwise blending is not supported in Direct3D 11, which is extensively used in current game development. Conversely, in our framework the bit-length and the number of layers in the grid are only limited by the number of render targets (1024 bits per voxel for the current eight render targets) and the size of the video memory. Additionally, our method can be implemented in either Direct3D or OpenGL. A recent work [23] uses a texture atlas for voxelization, achieving high performance but requiring that a model be appropriately mapped onto a 2D texture. This mapping needs pre-computation, and is not feasible for dynamic deformable objects.

Some recent methods use customized 3D rasterizers implemented on CUDA [19, 21, 27]. A per-pixel depth range for a triangle is derived, and an optimized overlapping computation is taken between each triangle-voxel pair. These methods generate accurate results, and we use these results as ground truth for reference and demonstrate that we can achieve a well-approximated result.

Another branch of recent works [5, 13] introduces the use of octree data structures to store massive voxelization results. Since our work focuses on how to convert the mesh into the voxel representation, storage and rendering the voxel are beyond our scope. However, our technique can be easily integrated into an octree-based storage framework. The voxelization results (either the point or the voxel data) can be copied into a piece of mapped-pinned host memory and used for building the octree.

3 Point-Tessellated Voxelization

3.1 Choosing a Tessellation Factor

In the context of hardware tessellation, the three vertices of a triangle are used directly as three control points in the hull shader [6, 22]. New vertices are generated in the domain shader and their positions are interpolated from the barycentric coordinates generated by the tessellator. To control the level of subdivision, we need to set the hardware tessellation factor. For each triangular patch, this factor is a vector containing four scalars: three edge scalars (denoted by $T_i, i = 1,2,3$ below) and one central scalar (denoted by T_C below). The hardware tessellator is designed similar to the work proposed by Gong et al. [14]. Each input triangle is recursively subdivided into one or more concentric rings, which are later used to determine the positions of newly generated vertices. According to Gong's work, the newly generated vertices are linearly distributed on each edge of a ring according to the scalar of that edge, and the rings are uniformly distributed from the centroid to the edges in each input triangle. Since we want to ensure that each cell occupied by the triangle edge contains at least one tessellated vertex from that edge, we use an adaptive tessellation factor per tessellated edge and relate it to the number of cells that an edge may occupy. Given the positions $P_i \in R^3, i = 1,2,3$ of the three vertices of a triangle, and the side length of a cell l_{cell} in the volumetric grid, the tessellation factor is calculated as below:

$$\begin{cases} E_{ij} = \gamma |P_i - P_j| / l_{cell}, i, j \in \{1,2,3\}, i \neq j \\ T_1 = max_{x,y,z} E_{12}, T_2 = max_{x,y,z} E_{23}, T_3 = max_{x,y,z} E_{13}, \\ T_C = (T_1 + T_2 + T_3)/3. \end{cases} \quad (1)$$

The γ in the equation above is a constant global safety factor. Section 4.1 discusses this equation and the coefficient γ in detail.

3.2 Point Cloud: from Triangles to Voxels

The triangle soup from the tessellator is poured into its next stage in the pipeline, the geometry shader. For voxelization, we could rasterize these triangles directly into a volume texture; nevertheless, there is a better strategy. Since point-based rendering (PBR) is cheap for the hardware rasterizer, we replace each subdivided triangle with a point primitive placed at its center. These points are called *surfels* in terms of PBR. Each surfel represents a micro-triangle in the original model. Its position is simply computed in world space as:

$$P_{centroid} = (P_{vert_1} + P_{vert_2} + P_{vert_3})/3. \quad (2)$$

This $P_{centroid}$ was empirically chosen. Prior to this, we also try other possible locations, such as the incenter (center of the incircle), the vertices, or both vertices and center of mass, and find that the center of mass provides balance between performance and accuracy. The results of experiment are shown in Figure 2.

The points are generated in the geometry shader. We also transform the point cloud into the space of a 3D texture in this stage. For one surfel, we directly calculate the position of the texel corresponding to it in the 3D texture, and output a value into the texel in the pixel shader. Since each triangle has been modeled as a point, it is unnecessary to compute the exact size and orientation of each surfel.

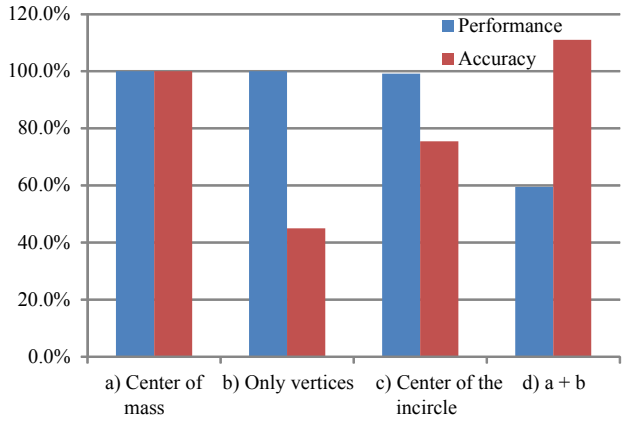
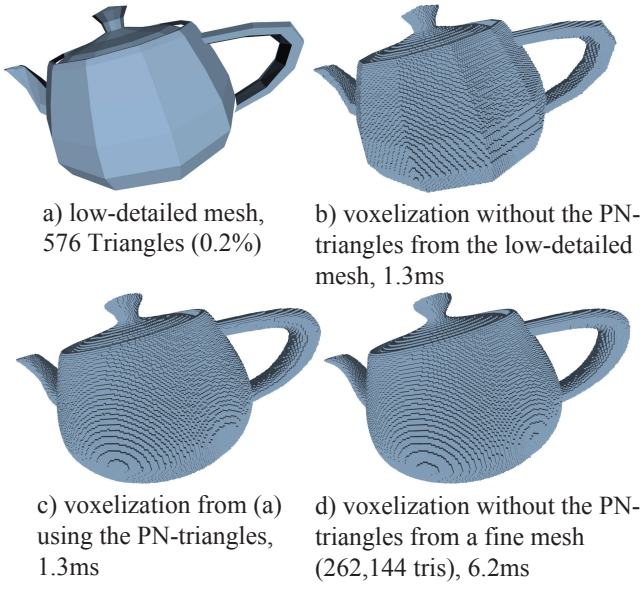


Figure 2: Comparisons between different choices of locations for point-shooting, regarding (a) as the reference. Although using both vertices and face centroids improves accuracy (about 10%), large performance losses (about 40%) are introduced.

In the case of binary voxelization, we simply output a constant value for each point in the pixel shader to indicate occupancy. Nonetheless, many applications may store the scalars or vectors contained in the triangles (colors, normals, etc.) into the voxels (called *Multi-valued Voxelization*). If only the average values across multiple triangles are required, additive blending can work well. If some applications require that the values from the point clouds all remain accessible in a voxel, we can compress them using spherical harmonics, similar to the method proposed by Makadia et al. [18].

3.3 Displacement Voxelization

A pervasive dilemma in voxelization is that higher accuracy demands a finer mesh, which leads to high storage requirements and low performance (Figure 3a); a low-detail mesh leads to unacceptable voxelization (Figure 3c). In this section, we extend our framework to displacement voxelization which is suitable for meshes with rugged surfaces. The voxelization can be even faster if using displacement mapping to add details of the geometry on the fly. Our



a) low-detailed mesh, 576 Triangles (0.2%)

b) voxelization without the PN-triangles from the low-detailed mesh, 1.3ms

c) voxelization from (a) using the PN-triangles, 1.3ms

d) voxelization without the PN-triangles from a fine mesh (262,144 tris), 6.2ms

Figure 4: Voxelization on a Utah Teapot. The size of the grid is 256^3.

method is based on *curved point-normal triangles* (or PN-triangles for short) [26]. PN-triangles provide a simple and efficient technique for subdivision with normal and position interpolation, offering a general solution to give any mesh a smoother aspect [2]. The fact that using the PN-triangles without a displacement map can produce a smoother voxelization result is shown in Figure 4. Nevertheless, a very smooth object can often be defined effectively with non-uniform rational basis splines (NURBs) or Bezier functions, and voxelizing these parametric surfaces is more efficient and accurate [17]. Therefore, in this paper we are mainly concerned with offering a more detailed voxelization from a low-detailed mesh and its displacement map.

When using PN-triangles, only the position and normal (and texture coordinates for displacement mapping) of each vertex are used. This property maintains the parsimony and efficiency of our framework. For PN-triangles-based displacement mapping, we follow the work proposed by Doggett et al. [7]. The GPU implementations of this method can be found in several commonly accessible SDKs [6, 22]. Since only the final points are affected by the displacement mapping, we sample the displacement texture in the geometry shader, just before the points are output into the volumetric grid.

The displacement voxelization result of a simplified mesh is visually similar to the result of its high-resolution version. Figure 3 shows that few differences can be noticed between our result (Figure 3d) and the reference (Figure 3e). The displacement map is generated using *ZBrush* at the resolution of 1024^2. The performance is improved four-fold over the straightforward method. A point that needs to be noted is that after the subdivision, the original displacement map will not be perpendicular to the new triangles, which may introduce artifacts. However, in our experiments, this fact hardly affected the results.

4 Results and Discussion

All the results in this paper were generated with a Core-i7 920 processor and a GTX480 display card. We implemented our approach with Direct3D 11.

The models are intended to demonstrate a wide range of applications. The *soldier* model in Figure 1 is a skinned mesh used to show the process of our method, and to highlight the low overhead

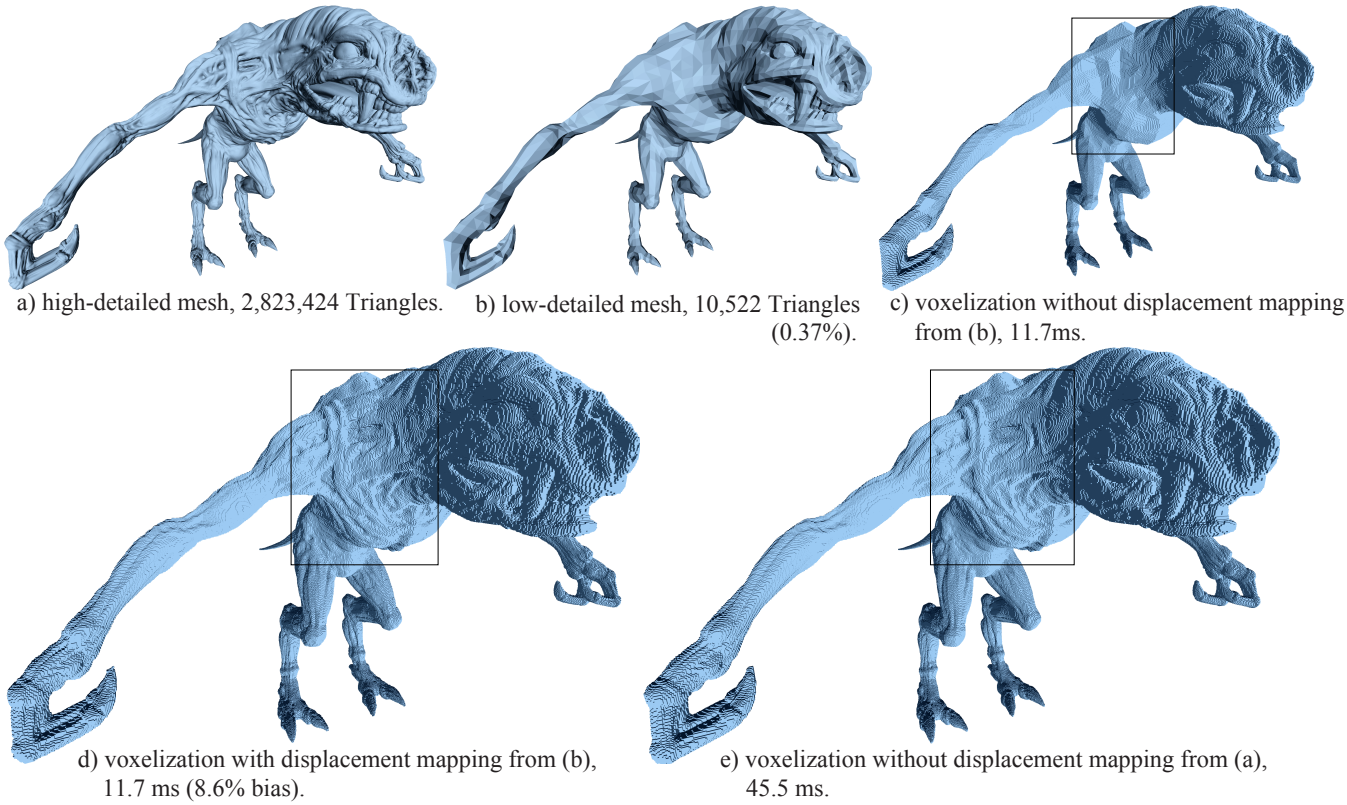


Figure 3: Comparisons between straightforward and displacement voxelizations on a monster from Metro 2033 (with grid size: 1024^3). Compared with the voxelization from the straightforward method, displacement voxelization accurately reconstructs thin features, such as the flesh, the spurs and the ribs (notice the parts in the black box), with almost no performance loss, and is four times faster than directly voxelizing a high-detailed mesh. Please zoom-in for a detailed look.

when voxelizing animated triangles.

The models in Figure 10 are used to compare our method with the state-of-the-art approaches [19, 21, 27] for various levels of geometric complexity. As shown in the table of Figure 10, in most cases, our method demonstrates higher performance than those methods and is significantly superior to the rasterization method [27].

We also notice that in some rare cases, our method can be slower than the latest method [19]. Therefore, we evaluate our technique for different stages of the pipeline, and find that the cost on the tessellation stage sharply increased as we change the grid from 128^3 to 1024^3. This fact illustrates that the hardware tessellator in the GTX480 is not very efficient at a high tessellation factor. To ensure that our technique can perform better on future hardware, we also try the up-to-date GTX560Ti, and find that the tessellation is never the most costly stage (Figure 6).

The *leafy olive* model in the top row of Figure 5 highlights the necessity of using tessellation. Compared to the result without tessellation (Figure 5a), the geometric continuity and stability of our method is shown in Figure 5b. As our method is not conservative but merely watertight, there may be some voxels missing. Nonetheless, these artifacts are usually acceptable in applications where complete accuracy is required, such as collision or shading in video games. We evaluate these artifacts in the bottom row of Figure 5 using a color-coded voxelization result of *Stanford Buddha*. We note that 1) all results are visually similar, despite the presence of artifacts, and 2) the artifacts can be eliminated when the mesh is over-tessellated.

To show that our solution can handle very detailed parts of a complex mesh, we also show a voxelization result of the Sibenik Cathedral at a resolution of 1792^3 (Figure 11e). Note that the strips on the wall and vault are well captured.

4.1 Balance between Performance and Accuracy

Safety Factor: We use $\gamma = 1$ in all the instances in this paper, producing a watertight result. However, if higher accuracy is required, a higher γ can be used. The safety factor γ affects the performance as well as the accuracy. The relationship between γ, accuracy and performance can be seen from the chart in Figure 5. Empirically, given a bias proportion value p, the required γ can be computed from a fitted logarithmic function:

$$\gamma = -B \times ln(Ap). \tag{3}$$

The parameter $A = 1/p_1$ is determined from the bias p_1 computed with $\gamma = 1$. This bias is the ratio of the number of missing voxels and the number of all occupied voxels in the ground truth. Another parameter B is computed in a similar way using various γ. With the corresponding p obtained from computing the bias proportion, we estimate: $B = -E(\gamma/ln(p/p_1))$, where E is the expectation value. Theoretically, the two parameters A and B should be mesh-specific; practically, however, the bias is stable under deformations that happened in all the meshes that we have tested, so we use constant parameters.

It is difficult to prove that Equation 1 can theoretically guarantee the watertight property. However, by using the ray-marching test we can show that this property holds for all the meshes used in this paper. We shoot rays from six directions (orthogonal to the x-y, y-z and x-z planes, respectively) towards the grid, checking whether

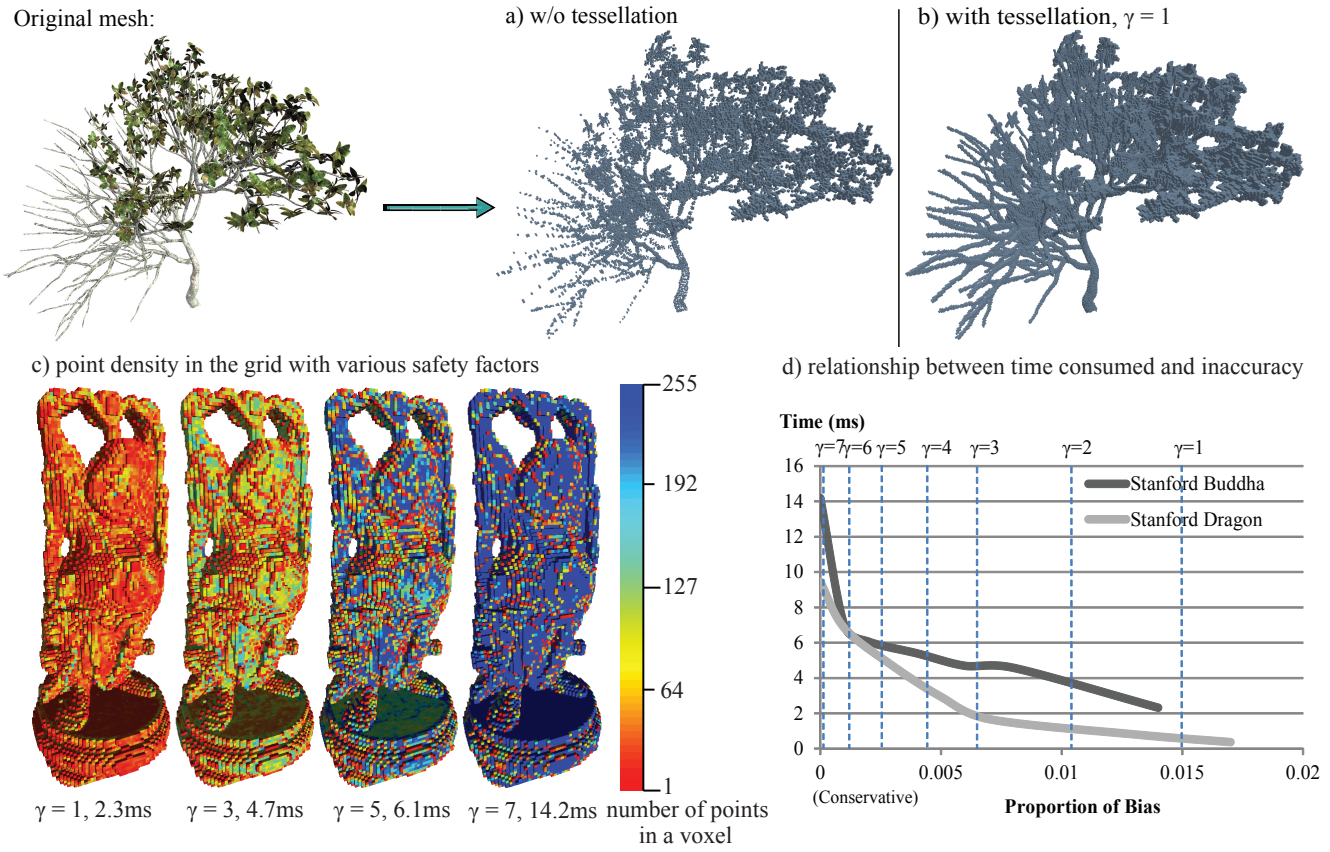


Figure 5: Top row: Comparisons between voxelization with and without tessellation. Bottom row: A Stanford Buddha is voxelized with different safety factors, from 1 to 7 (over-tessellated). The number of points falling into one voxel is visualized in a color-coded style in (c). In (d), the relationship between performance and bias is demonstrated. With higher safety factors, our gap-free results (on both Stanford Buddha and Stanford Dragon) rapidly converge to the conservative ground truth.

the intersected voxels (or the grid boundary if no voxel is hit by the rays) are close to the mesh surface. If the distance from the centroid of a voxel to the original triangle plane is less than the side length of a voxel, we deem the voxel to be "close" to the surface. We perform this test for each mesh used in this paper, and the results show that all of the intersected voxels are close to their original triangle planes, which demonstrates that the watertight property holds.

User Study: To further investigate the efficiency of our method, we conduct a user study. For each mesh in this paper, the ground-truth voxelization result and our approximated voxelization result (with $\gamma = 1$) were simultaneously shown to 20 users (at a size of 256^3, randomly positioned on the left or right of the screen. The visualized figures could be freely zoomed or rotated in three dimensions). The users (who were all graduated students of graphics and well-versed in conservative voxelization) were asked to choose which one was more likely to be the ground truth. We recommended that they take less than two minutes to make their decision. The users showed no obvious preference for the ground truth (on average the users chose our result as the ground truth half of the time, as shown in Figure 7). This study demonstrated that the bias in our approximation was visually unnoticeable.

4.2 Handling Multi-Pass Subdivision

Some special applications may need to voxelize a mesh containing just a few triangles into a large grid (e.g., voxelizing a tetrahedron that has only 4 triangles into a grid of 1024^3). In that case, we have to tessellate the mesh into a very fine level with additional subdivision passes since the tessellation factor is limited (up to 63 in current hardware, generating $5000 \sim 6000$ triangles for one triangle). This makes our method less convenient. Fortunately, the necessary number of passes can be empirically estimated as:

$$n = log_{N_{63}} N_e, \tag{4}$$

where N_{63} is the number of newly generated triangles when setting the tessellation factor for each triangle as 63, which could be achieved using the *pipeline statistics query* after a rendering pass for testing. N_e is the expected number of surfels generated, empirically, estimated as:

$$N_e = \frac{3N_{tri}l_{avg}^2}{2l_{cell}^2}. \tag{5}$$

This equation is abstracted from our observation of the tessellation pattern [14, 1]. An accurate equation determined from the recursive mechanism of the hardware tessellator [14] is unnecessary. N_{tri} is the number of triangles in the mesh, l_{avg} is the average length of edges of all triangles, and l_{cell} is the side length of a grid cell. During our experiment, we find the average of N_{63} for any triangle achieves more than 5000, especially when the size of a triangle is large. This makes the number of passes n always very small: it never goes beyond 3 in practice (if it did, the number of triangles generated from one triangle would be over 2^{32}). Some samples of voxelization of simple meshes are proposed in Figure 8.

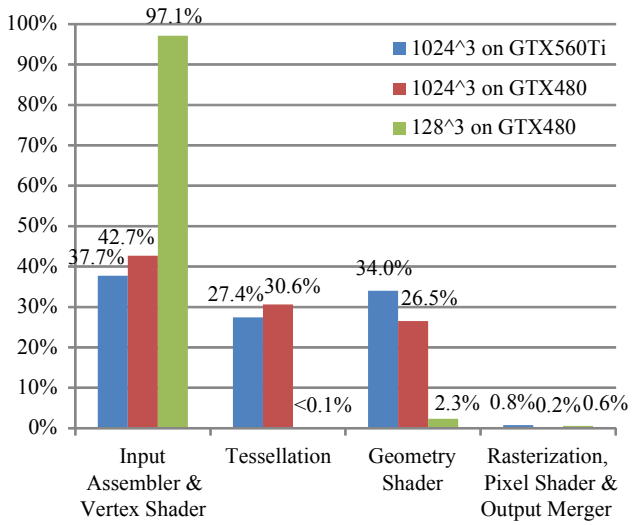


Figure 6: The proportion of time spent on various stages in different situations. The data was obtained using the Stanford Dragon mesh.

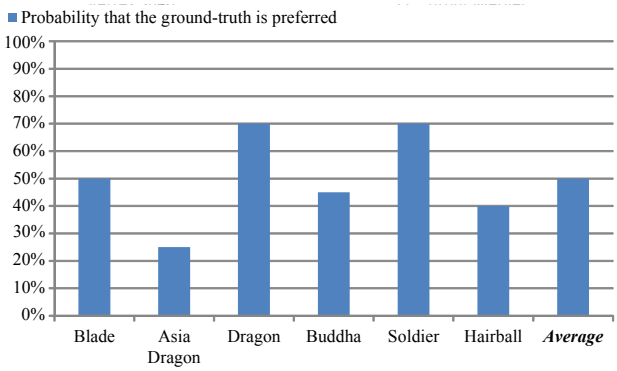


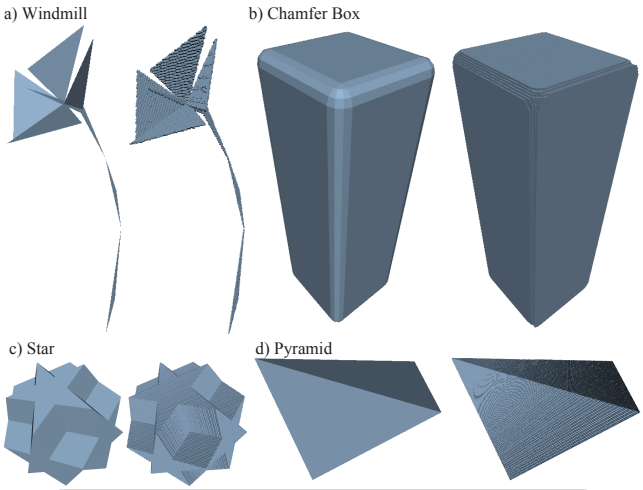
Figure 7: The user study result.

4.3 Application: Real-time Voxel-based Ambient Occlusion

Ambient occlusion (AO) is an approximation to the indirect lighting produced by skylight, and can be straightforwardly implemented by sampling the geometry near a pixel that is assumed to be blocked from the skylight. Many AO algorithms have been developed in recent decades. We harness the power of our method to render *Hybrid Ambient Occlusion* (HAO), initially proposed by Reinbothe et al. [20]. Instead of sampling the geometry directly, the HAO method samples the geometry by marching rays through a voxelization result of the scene and treating each non-void voxel as a 3D proxy of the nearby geometries. We integrate our method with HAO by directly substituting our voxelization process for their method. The result is shown in Figure 11.

5 Comparisons

Compared with slice-based methods [4, 8], our one-pass method is much more convenient and efficient. Compared with methods that use bitwise operations [8, 9, 10], our method can handle voxel grids that are only limited by the video memory, with multiple values stored in each voxel, making our method applicable to a broader range of applications. Compared with methods that use customized rasterization [19, 21, 27], our technique is more usable for programmers who require a working voxelization method with high efficiency. Compared with *atlas voxelization* [23], our method can handle more general and complicated scenes.



Scene	Grid	Time (ms)	Bias
a) Windmill (8 tris, 1 level)	128^3	0.2	1.0%
	512^3	5.2	0.1%
	1024^3	40.0	0.3%
b) Pyramid (6 tris, 2 levels)	128^3	0.2	0.1%
	512^3	3.2	0.0%
	1024^3	10.9	0.0%
c) Chamfer Box (204 tris, 2 levels)	128^3	0.4	0.7%
	512^3	2.7	0.1%
	1024^3	7.0	0.1%
d) Star (96 tris, 2 levels)	128^3	0.4	3.1%
	512^3	6.5	1.1%
	1024^3	31.3	0.0%

Figure 8: The voxelization results on some simple meshes, "level(s)" means the number of pass(es) used for the subdivision. The pictures are rendered using a grid with a resolution at 256^3.

6 Limitations and Future Works

Currently, our method does not focus on volumetric voxelization (or solid voxelization), which is reserved for future work. Likewise, our method does not aim for accurate voxelization with no missing voxels (Figure 9), which may be required in some specific applications. Moreover, the bias can be increased when the size of grid is increased, or when applying displacement voxelization to coarse meshes. Additionally, our technique has no overwhelming advantage in performance when the number of triangles is very low while the grid resolution is high. Combining the existing accurate approaches into our framework without violating the parsimony is also a promising direction for the future.

7 Conclusions

In this paper, we described the integration of hardware-based triangle subdivision and point-based rendering into an easily approachable framework for surface voxelization. The results are well approximated to the ground truth, making our technique especially suitable for applications that do not have a stringent requirement for accuracy but do require development and runtime efficiency.

Acknowledgements

The research was supported by National Basic Research Program of China (2010CB328001), National Science Foundation

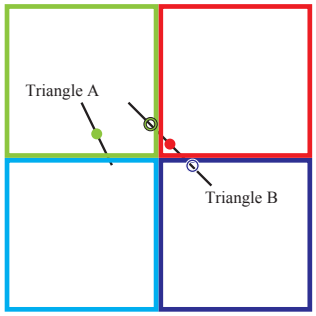


Figure 9: The 2D illustration of the possible bias in our method. Equation 1 is insufficient for a completely accurate result. For example, triangle A is smaller than a voxel, but from its center only the green voxel can be detected. Moreover, the voxel detected in a lower subdivision may become lost in a higher subdivision. For example, we can detect the red voxel through triangle B; by subdividing the triangle B into two triangles, however, we can detect the green and the blue voxels, but will miss the red voxel.

of China (61003096), and National High-tech R&D Program (2012AA040902).

REFERENCES

[1] Tessellation pattern viewer. In *NVIDIA Direct3D 11 SDK*. NVIDIA Corporation, 2010.

[2] T. Boubekeur and C. Schlick. Generic mesh refinement on gpu. In *Proceedings of HWWS '05*, pages 99–104, 2005.

[3] R. L. Cook. Shade trees. In *Proceedings of SIGGRAPH '84*, pages 223–231, 1984.

[4] K. Crane, I. Llamas, and S. Tariq. Real-time simulation and rendering of 3d fluids. In *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 633–675. Addison-Wesley Professional, 2007.

[5] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of I3D '09*, pages 15–22, 2009.

[6] K. Dmitriev. Pn-patches. In *NVIDIA Direct3D 11 SDK*. NVIDIA Corporation, 2010.

[7] M. Doggett. Displacement mapping. *ATI Research*, 2003.

[8] Z. Dong, W. Chen, H. Bao, H. Zhang, and Q. Peng. Real-time voxelization for complex polygonal models. In *Proceedings of PG '04*, pages 43–50, 2004.

[9] E. Eisemann and X. Décoret. Fast scene voxelization and applications. In *Proceedings of I3D '06*, pages 71–78, 2006.

[10] E. Eisemann and X. Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of GI '08*, pages 73–80, 2008.

[11] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433–442, 2000.

[12] G. Favalora, J. Napoli, D. Hall, R. Dorval, M. Giovinco, M. Richmond, and W. Chun. 100 million-voxel volumetric display. In *Proceedings of SPIE '02*, volume 4712, pages 300–312, 2002.

[13] E. Gobbetti, F. Marton, and J. A. Iglesias Guitián. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24:797–806, 2008.

[14] M. Gong. Parallel triangle tessellation. In *US Patent App. 12/629,623*, pages 1–22, 2009.

[15] O. S. Lawlor and L. V. Kalée. A voxel-based parallel collision detection algorithm. In *Proceedings of ICS '02*, pages 285–293, 2002.

[16] W. Li, Z. Fan, X. Wei, and A. Kaufman. Gpu-based flow simulation with complex boundaries. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 747–764. Addison-Wesley Professional, 2005.

[17] F. Lin, H. Seah, Z. Wu, and D. Ma. Voxelization and fabrication of freeform models. *Virtual and Physical Prototyping*, 2(2):65–73, 2007.

[18] A. Makadia, A. Patterson, and K. Daniilidis. Fully automatic registration of 3d point clouds. In *Proceedings of CVPR '06*, pages 1297–1304, 2006.

[19] J. Pantaleoni. Voxelpipe: A programmable pipeline for 3d voxelization. In *Proceedings of HPG '11*, pages 1–8, 2011.

[20] C. Reinbothe, T. Boubekeur, and M. Alexa. Hybrid ambient occlusion. *Computer Graphics Forum*, pages 51–57, 2009.

[21] M. Schwarz and H.-P. Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Transactions on Graphics*, 29:179:1–179:10, 2010.

[22] J. Story. Pn-triangles 11. In *ATI Radeon™ SDK*. AMD Graphics Products Group, 2010.

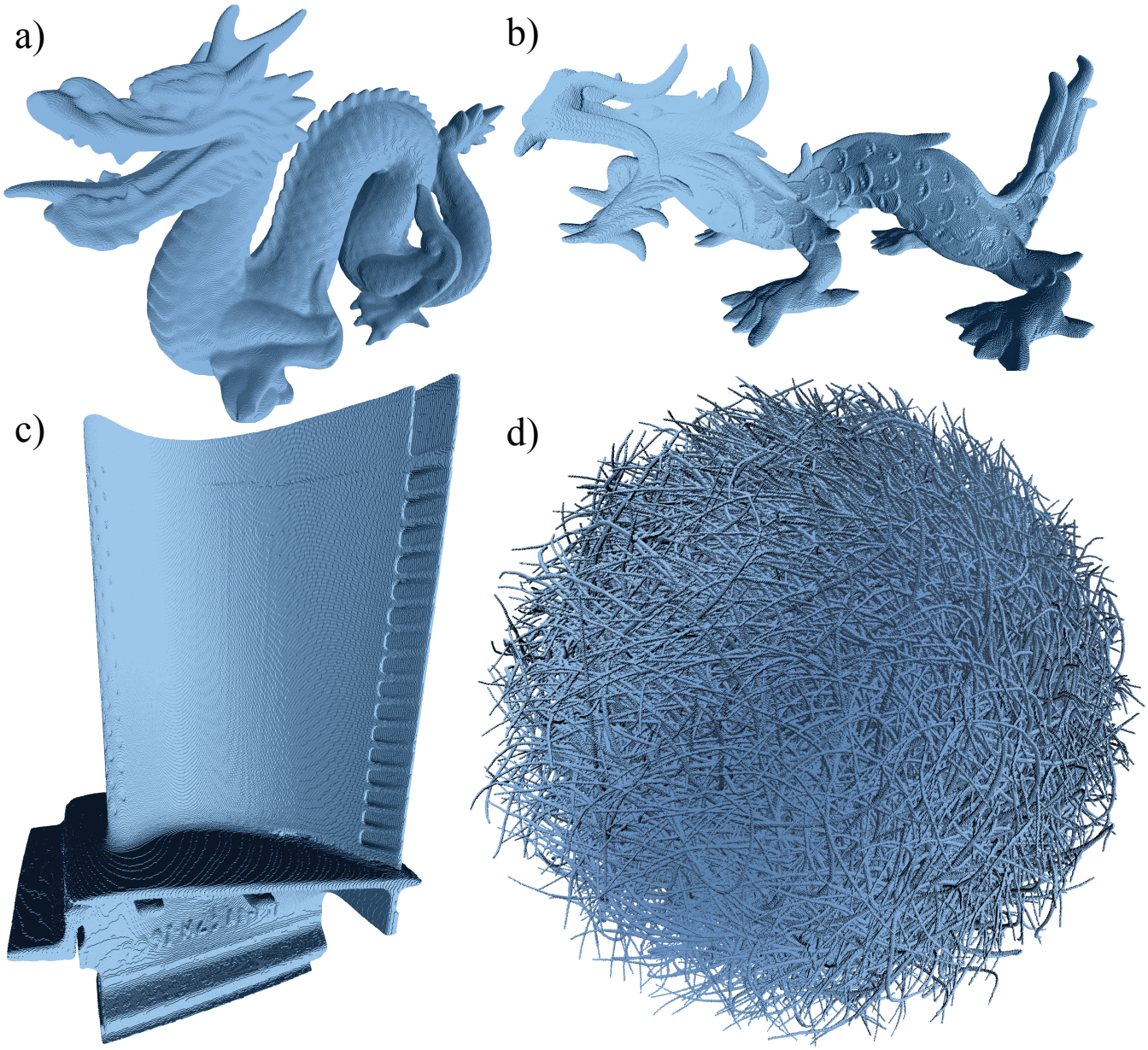
[23] S. Thiedemann, N. Henrich, T. Grosch, and S. Müller. Voxel-based global illumination. In *Proceedings of I3D '11*, pages 103–110, 2011.

[24] D. Triolet. Report: Nvidia geforce gtx 580 & sli, 2010.

[25] P. K. Venuvinod and W. Ma. *Rapid prototyping: laser-based and other technologies*. Springer, 2001.

[26] A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell. Curved pn triangles. In *Proceedings of I3D '01*, pages 159–166, 2001.

[27] L. Zhang, W. Chen, D. S. Ebert, and Q. Peng. Conservative voxelization. *The Visual Computer*, 23:783–792, 2007.



Scene	Grid	Zhang et al.[27]	Schwarz et al.[21]	Pantaleoni[19]	Ours				
		Time (ms)	Time (ms)	Time (ms)	Time (ms)	Ratio			Bias
a) Stanford dragon (871,414 Tris)	128^3	44.9	3.5	4.8	0.4	112.3x	8.8x	12.0x	2.4%
	512^3	54.9	4.8	5.0	2.3	23.9x	2.1x	2.2x	4.1%
	1024^3	73.7	13.7	5.9	12.8	5.8x	1.1x	0.46x	3.7%
b) Asian dragon (7,218,906 Tris)	128^3	257.6	16.7	21.2	0.4	644.0x	41.8x	53.0x	1.7%
	512^3	365.2	24.5	22.0	2.7	135.3x	9.1x	8.1x	3.8%
	1024^3	508.3	24.6	23.6	9.7	52.4x	2.5x	2.4x	3.3%
c) Hairball (2,880,000 Tris)	128^3	-	22.8	12.8	5.9	-	3.9x	2.2x	2.8%
	512^3	-	95.0	18.3	7.5	-	12.7x	2.4x	4.4%
	1024^3	-	266.8	33.7	56.8	-	4.7x	0.6x	3.7%
d) Turbine Blade (1,765,388 Tris)	128^3	-	3.6	7.3	0.5	-	7.2x	14.6x	3.3%
	512^3	-	7.6	6.9	5.3	-	1.4x	1.3x	3.4%
	1024^3	-	16.6	8.4	7.3	-	2.3x	1.2x	2.3%

Figure 10: The voxelization results (1024^3) of: a) the Stanford dragon, b) the XYZ RGB Asian dragon, c) the Hairball, and d) the Turbine blade. The results are visualized with one box representing a voxel. Please zoom in for more details. The table compares the performance and bias with other methods. "Bias" means the proportion of voxels missing from the ground truth. The timings for other methods are directly taken from their publications, which also used a GTX480. A "-" means that the scene is not evaluated in the original publication.

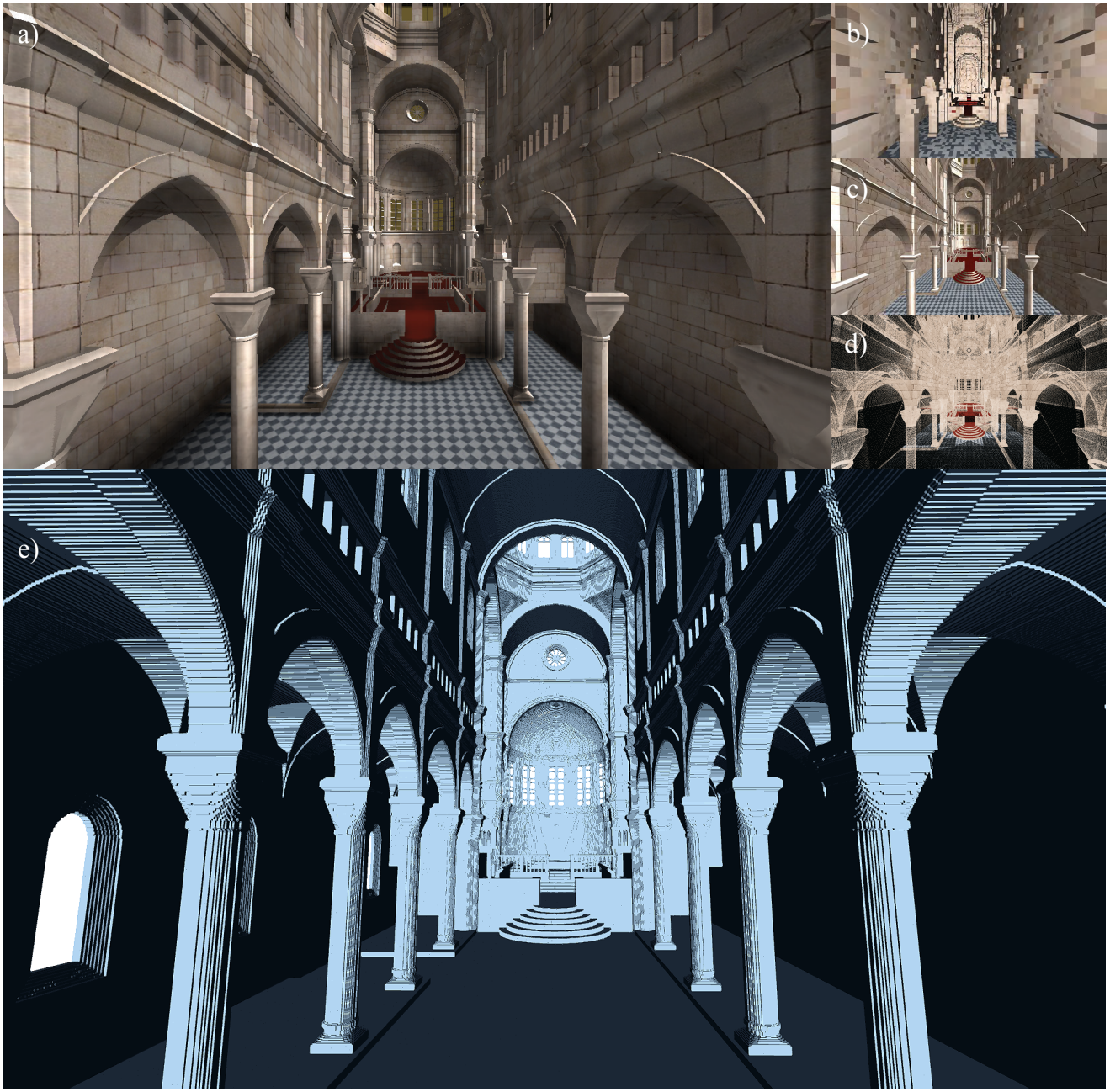


Figure 11: Applying our method for real-time hybrid ambient occlusion (HAO): a) our HAO result, 16.2ms, b) the voxelized Sibenik Cathedral, c) the result without HAO, d) Sibenik Cathedral with tessellated points, and e) voxelization with resolution at 1792^3.