VI 2

Practical Binary Surface and Solid Voxelization with Direct3D 11

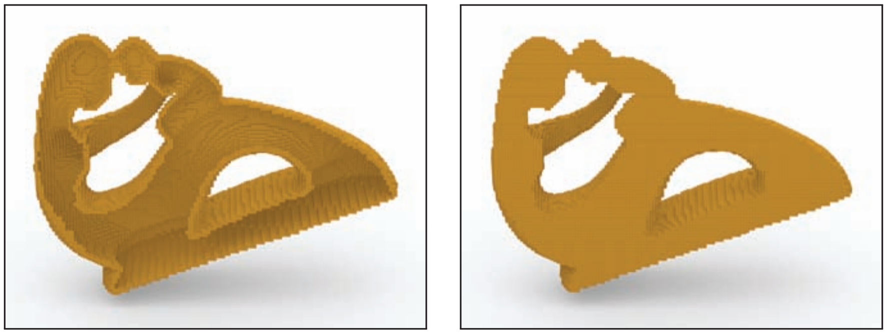
Michael Schwarz

2.1 Introduction

Regular, discrete representations of potentially complex signals are routinely used in many fields. They provide a comfortable domain in which to work, often facilitate processing, and are largely independent from the represented signal's complexity. In computer graphics, we encounter such representations mainly in the form of two-dimensional images, like the final rendering result or a shadow map. Their three-dimensional analog, and the focus of this chapter, is *voxelizations* stored in voxel grids. They offer a volumetric representation of a scene, where each grid cell, referred to as a *voxel*, encodes that part of the scene that is located within the cell. In case of a *binary voxelization*, this encoding is particularly simple: merely two states are distinguished, where a set voxel indicates the presence of some, and an unset voxel the absence of any, scene object.

Largely orthogonal to this encoding, two main flavors of voxelizations can be distinguished (see Figure 2.1). In a *surface voxelization* (also called boundary voxelization), the scene is interpreted as consisting solely of surfaces, that is, all closed objects are assumed hollow. Therefore, only voxels overlapped by a surface (like a scene triangle) will be nonempty. By contrast, a *solid voxelization* treats all objects as solids, and hence, any voxel interior to an object will be set. Note that this basically requires the objects to be closed.

Binary voxelizations are useful for many applications, ranging from collision detection [Zhang et al. 07, Eisemann and Décoret 08] to ambient occlusion [Reinbothe et al. 09], soft shadows [Forest et al. 10], area light visibility [Nichols et al. 10], and volumetric shadows [Wyman 11]. Unless confined to static settings, they all mandate that the voxelization be created on the fly at real-time speed. This chapter describes how this goal can be achieved using Direct3D 11, covering the process of turning an input scene, given as a collection



(a) Surface voxelization. (b) Solid voxelization.

Figure 2.1. In a surface voxelization (a), the voxels overlapping the scene's surfaces are set, whereas in a solid voxelization (b), the voxels that are inside a scene object are set. (For illustration purposes, the fronts of the voxelizations have been cut away, revealing the interiors.)

of triangles, into a binary surface or solid voxelization. We first discuss implementations that build on the standard graphics pipeline and its rasterization stage; these are rather simple and easy to implement but also suffer from several shortcomings. Addressing some of them, we subsequently investigate alternative realizations that employ DirectCompute.

2.2 Rasterization-Based Surface Voxelization

With surface voxelization basically being a three-dimensional extension of rasterization, it is natural to try to pose this process in terms of rasterization and harness the according existing hardware units. Assuming a target voxel grid of size $w \times h \times d$, a simple approach is to render the scene into a deep framebuffer of resolution $w \times h$ with (at least) d bits per pixel (see Figure 2.2). Hence, for each triangle, all pixels covered are determined, and a fragment is generated for each of them. In the invoked pixel shader, those voxels within the voxel column represented by the according deep pixel are identified that are actually covered by the triangle. The resulting voxel-column pixel value is then output by the shader and incorporated into the existing framebuffer.

2.2.1 Challenges

Unfortunately, this appealing approach faces several problems in practice. A major issue is that in ordinary rasterization, a pixel is only considered to be covered by a triangle if the triangle overlaps the pixel's center. This implies that no fragments are generated for all those pixels that a triangle partially overlaps without simultaneously overlapping their centers. Consequently, no voxels are

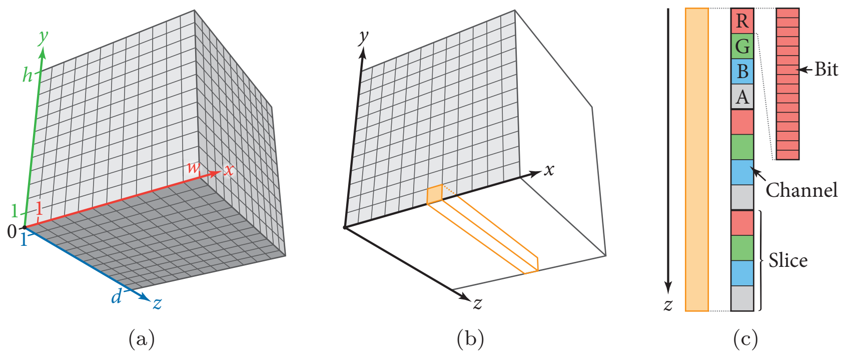
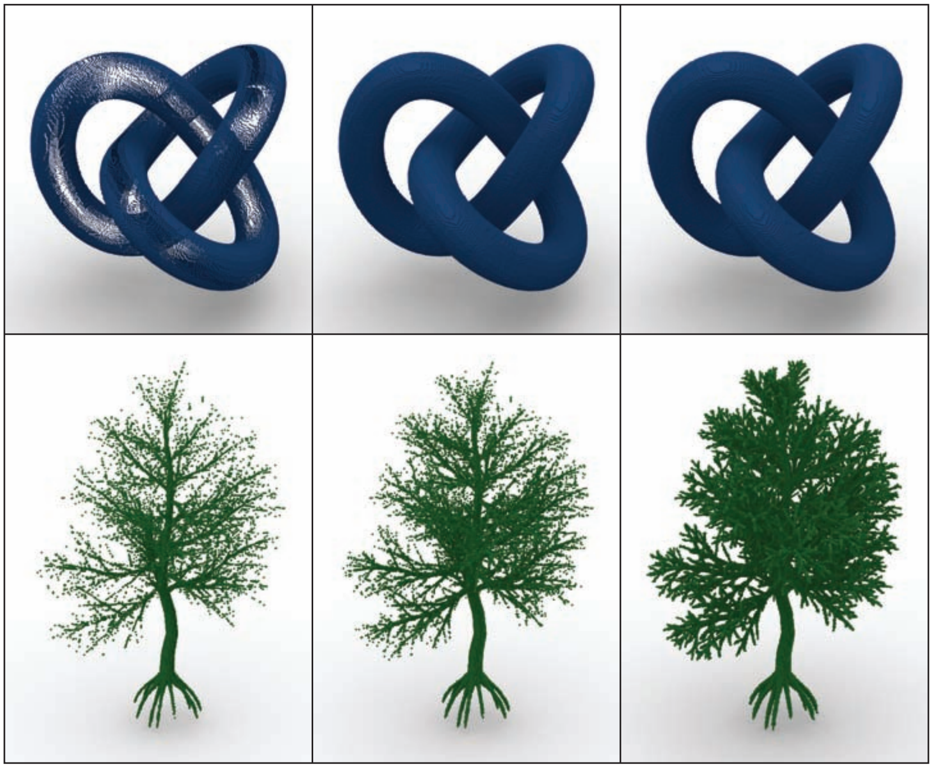


Figure 2.2. (a) We assume a voxel grid of size $w \times h \times d$ and define the *voxel space* such that each voxel has a footprint of size 1^3 and the grid covers the range $[0, w] \times [0, h] \times [0, d]$. (b) The grid can be thought of as a 2D image with deep pixels (each representing a voxel column). (c) These deep pixels may be encoded in multiple multichannel texture slices.

set in these cases, making resulting voxelizations routinely suffer from gaps and miss thin structures (see Figure 2.3). The situation can be somewhat alleviated by rendering the scene not just from one side of the grid (xy) but from three orthogonal sides (xy, xz, and yz) [Forest et al. 10]. This necessitates additional temporary buffers and a merging step, though, and still easily misses many voxels. The only real solution is to always generate a fragment if any part of a pixel's area is overlapped by a triangle. Current hardware, however, does not provide for this so-called *conservative rasterization*. While it can be emulated in software by enlarging each triangle appropriately [Hasselgren et al. 05] (e.g., in the geometry shader), this obviously incurs a significant overhead. Therefore, it is not surprising that an according voxelization algorithm [Zhang et al. 07] turns out to often be rather slow (and it also suffers from some robustness issues). For now, we will simply ignore this problem, but we will tackle it later in Section 2.4 when adopting a compute-based approach.

Another issue arises from updating the voxel grid to incorporate the voxel column output for a fragment. For correct results, obviously a bitwise OR operation is needed. One solution is to realize the deep framebuffer by multiple multichannel render targets, using the single bits of all the channels for encoding the voxel states (see Figure 2.2(c)), and to perform bitwise-OR blending. This was first demonstrated by Dong et al. [Dong et al. 04] (using additive blending, though); the specific framebuffer encoding is sometimes referred to as *slicemap* [Eisemann and Décoret 06]. While this approach works well when using OpenGL, it is not possible with Direct3D, as recent versions no longer expose the hardware's still-existing bitwise blending functionality (also known as logical pixel operations), which once was used for color-index rendering.



(a) Rasterized (1 dir). (b) Rasterized (3 dirs). (c) Conservative.

Figure 2.3. (a) If surface voxelization is performed with ordinary rasterization, voxels are frequently missed, leading to perforated surfaces and unconnected voxel clouds. (b) Executing this process from three orthogonal directions and combining the results helps in the case of closed surfaces (like the torus knot), but generally still fails badly if thin structures are involved (like in the leaf-deprived Italian maple). (c) By contrast, conservative voxelization yields correct results.

2.2.2 Approach

Fortunately, Direct3D 11 introduces the ability to write to arbitrary positions of a resource from within the pixel shader and further supports atomic updates to this end. These new features finally allow for pursuing a rasterization-based surface voxelization approach with Direct3D. The basic idea is to replace the bitwise blending of the pixel shader output into the deep framebuffer by performing atomic bitwise update operations on the voxel grid within the pixel shader.

This leads to the following overall approach: First, a resource for storing the voxel grid of size $w \times h \times d$, along with an according unordered access view (UAV), needs to be created. This can be a texture (array) or a buffer, with the latter

```
RWBuffer<uint> g_rwbufVoxels;

struct PSInput_Voxelize {
  float4 pos      : SV_Position;
  float4 gridPos : POSITION_GRID;
};

PSInput_Voxelize VS_Voxelize(VSInput_Model input) {
  PSInput_Voxelize output;
  output.pos = mul(g_matModelToClip, input.pos);
  output.gridPos = mul(g_matModelToVoxel, input.pos);
  return output;
}

float4 PS_VoxelizeSurface(PSInput_Voxelize input) : SV_Target {
  int3 p = int3(input.gridPos.xyz / input.gridPos.w);
  InterlockedOr(g_rwbufVoxels[p.x * g_stride.x + p.y * g_stride.y +
                              (p.z >> 5)], 1 << (p.z & 31));

  discard;
  return 0.0;
}
```

Listing 2.1. Vertex and pixel shaders for rasterization-based surface voxelization.

being the best choice for most applications. As for data format, a 32-bit integer type has to be chosen to allow for atomic updates. Furthermore, either a render target or a depth/stencil target of minimum size $w \times h$ needs to be available.

For creating the voxelization, this target is bound as the only output-merger-stage target, and a viewport of size $w \times h$ is set. Moreover, the voxel grid is reset by clearing its UAV with zeroes, and this UAV is bound. We then render the scene, using the vertex and pixel shaders from Listing 2.1, which adopt the buffer layout shown in Figure 2.4 as a concrete example. The vertex shader transforms

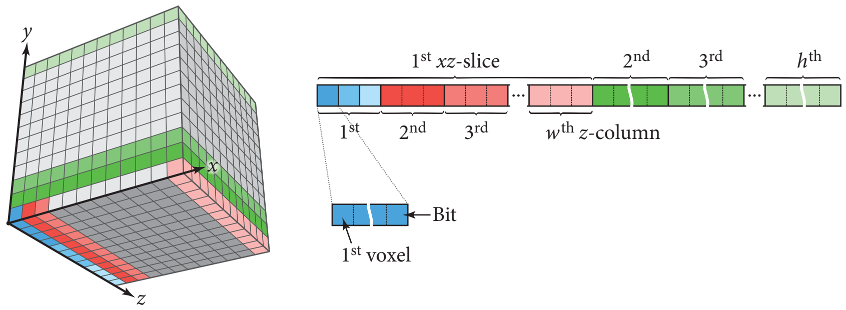


Figure 2.4. To store the voxel grid in a buffer, we linearize it first along the z-direction, packing 32 consecutive voxels into the bits of a 32-bit (unsigned) integer buffer value, then along the x-direction, and finally along the y-direction.

the vertex position into clip space, mapping the grid's xyz-extent to the clip volume (i.e., to $[-1,1]^2 \times [0,1]$), and into voxel space, mapping the grid's xyz-extent to $[0,w] \times [0,h] \times [0,d]$. Based on the clip-space coordinates, each triangle is rasterized, and a fragment is generated for each pixel (voxel column) whose center is overlapped. In the pixel shader, the voxel-space coordinate is used to directly set the according voxel in the voxel grid with an atomic OR operation. Subsequently, the fragment is discarded. Note that even though the render or depth/stencil target is thus never updated, it is still required, as otherwise no fragments would be generated.

2.2.3 Discussion

Compared to approaches using bitwise blending, confining them to OpenGL, the presented technique is similarly easy to implement and has both advantages and disadvantages. On the downside, it is often somewhat slower, especially in case of larger voxel grids. This is because random atomic updates to the voxelization are less optimized and efficient than blending executed by the graphics hardware's dedicated raster operation (ROP) units.

A big advantage of the described approach, however, is that the data organization is significantly nicer. Using a buffer resource and storing the voxel grid as a linearized 3D array, each voxel can be easily accessed via simple indexing into this buffer. In particular, this makes working with the generated voxelization rather straightforward. Moreover, it decouples the voxel grid from the side of the grid from which the scene is rendered. Consequently, if the scene is rendered from all three axis directions in order to reduce the artifacts resulting from non-conservative rasterization, a single voxelization buffer can be shared, both saving space and avoiding a final merging step.

By contrast, a deep framebuffer consisting of multiple multichannel render targets, typically stored in a texture array, makes addressing cumbersome because within a render target (texture slice) individual channels cannot be accessed via indexing; instead, conditional expressions are required. Furthermore, as the concrete number of render targets and channels depends on d, supporting variable grid depths d necessitates multiple pixel shaders—one for each render target count. Finally, for large values of d, the needed number of render targets can exceed the maximum number supported by the hardware, requiring multiple passes to fill the voxel grid.

2.3 Rasterization-Based Solid Voxelization

In solid voxelization, we seek to determine all voxels that are interior to some object. Typically, a voxel is considered interior if its center is interior, and we adopt this criterion, too. This leaves us with the task of computing, for each voxel center, whether or not it is located inside some scene object. Assuming that the

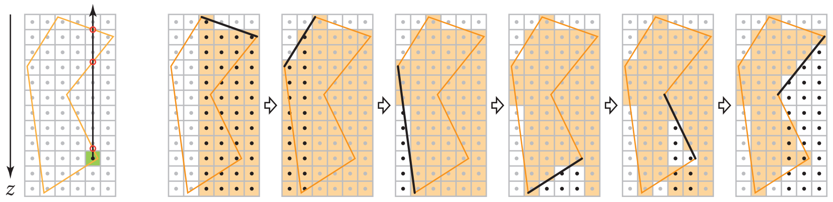


Figure 2.5. A voxel is inside an object if a ray shot upward from its center intersects the scene an odd number of times (far left). These intersection parities can be determined by rendering all triangles, each time flipping all voxels whose centers are below the respective triangle.

scene consists solely of closed, watertight objects and that no object is contained in another one, a point can be classified accordingly by shooting a ray from this point in an arbitrary direction and counting the number of intersections with the scene. If this number is even, the ray entered as many objects as it exited, and hence, the point must be outside. Conversely, an odd count indicates that the point is interior.

Instead of shooting a ray from each voxel center, the parity of the intersection count can be determined equally well using the following approach [Fang and Chen 00, Eisemann and Décoret 08]: First, all voxels are initialized to unset. Subsequently, we loop over all triangles. For each, the state of all voxels whose center is "below" this triangle is flipped. This accounts for the fact that if a ray would be shot from such a center "upward," it would intersect the triangle, increasing the intersection count and hence flipping its parity. Consequently, the final voxel state correctly reflects the parity of the total intersection count and thus the voxel's inside/outside classification (see Figure 2.5).

Using ordinary rasterization from the voxel grid's xy-side, identifying the voxels "below" a triangle is straightforward: for each fragment, corresponding to a whole voxel column, we merely have to select all voxels in this column whose z-index is larger than the fragment's voxel-space z-coordinate minus 0.5.[1]

Consequently, for a practical implementation, the same code as for the surface voxelization can be used. The only modification required affects the pixel shader: instead of setting the voxel corresponding to the fragment's voxel-space coordinates via a bitwise OR, we now have to flip the state of all voxels whose center is below the fragment via atomic XOR operations. Moreover, depth clipping should be turned off (or the clip-space z-coordinate set to a constant $\tilde{z} \in [0, 1]$ in the vertex shader); otherwise, some intersections may be missed if part of the scene is in front of the grid, potentially leading to wrong results.

Note that because a voxel's state is determined solely by the inside/outside classification of its center, accurate results are obtained with ordinary rasteri-

[1]This half-voxel adjustment is because we consider the voxel's center.

zation. This is unlike the situation with surface voxelization, where, instead, conservative rasterization is generally required for correctness.

2.4 Conservative Surface Voxelization with DirectCompute

Having seen that using the existing rasterization hardware for voxelization is rather simple, we now turn to an alternative approach based on DirectCompute.[2] It basically boils down to writing our own 3D software rasterizer, which is directly executed on the GPU. This allows us to adapt the rasterizer to the specific problem of voxelization and to exploit arising optimization potential. Being no longer bound to the design decisions and behavior of the graphics hardware's rasterizer, we are, in particular, free to adopt any criterion we want to determine whether a triangle covers a pixel or voxel. Consequently, the fundamental problem of surface voxelization encountered in Section 2.2 can be addressed, namely that the hardware rasterizer does not generate fragments for pixels that are partially covered without their respective centers being covered. Recall that this causes many voxels to incorrectly not be set, severely restricting the usefulness of according techniques.

2.4.1 Approach

Our software surface voxelizer processes all scene triangles in parallel, spending one thread per triangle. In the executed compute shader, at first, the triangle's vertices are fetched and transformed into voxel space. The bounding box of the transformed triangle is then determined and subsequently clipped against the voxel grid. In the case that the clipped bounding box is empty, the triangle is entirely outside the voxel grid, and we are done. Otherwise, the shader loops over all voxels within the bounding box that are potentially overlapped by the triangle. For each of these candidate voxels, an overlap test is performed, and, if the test passes, the voxel is set with an atomic OR operation.

To facilitate a practical implementation of this basic approach, some details should be pointed out. Firstly, the input scene geometry may be made available to the voxelizer in various forms. This could be a list of transformed vertex triplets, each representing a triangle, possibly collected in a stream-output buffer after vertex processing with an ordinary vertex shader. Another possibility is to provide a vertex buffer and an index buffer, and to perform any vertex processing in the compute shader together with the transformation into voxel space.

The chosen set of potentially overlapped voxels may basically be any superset of the actually overlapped voxels. Thus, a simple, conservative approach is just to consider all voxels that are (partially) within the triangle's bounding box. As

[2]This approach heavily builds on our previous CUDA-based work [Schwarz and Seidel 10].

each candidate voxel needs to be further investigated, subjecting it to an overlap test, and this test is far from free, performance benefits from better strategies that reduce the number of tested voxels that fail the test and that, hence, would ideally not have been considered in the first place. We will come back to this in Section 2.4.2.

Concerning the overlap test, we are fairly free to choose any criterion we like to define when an overlap occurs. For instance, by testing whether the triangle overlaps the voxel's 3D extent at least partially, a *conservative surface voxelization* is obtained. An according fast triangle/voxel overlap test is detailed in Section 2.4.2.

Finally, when looping over multiple voxels that are represented by different bits of the same 32-bit buffer value, it is advantageous to not immediately update the voxelization buffer for each set voxel among them. Instead, these updates should be buffered in a local 32-bit register and, once the last voxel has been processed, collectively written to the voxelization buffer with a single atomic OR operation.

2.4.2 Triangle/Voxel Overlap Testing

For conservative surface voxelization, we need to determine which voxels are at least partially overlapped by a certain triangle. To this end, we adopt a triangle/box overlap test [Schwarz and Seidel 10] that lends itself to a GPU implementation. It comprises several simpler tests that all have to succeed for an overlap to occur; if any of them fails, the voxel is not covered by the triangle.

Bounding box overlap. Given a triangle T and a voxel V, it is first checked whether the bounding box of T and V overlap. Since, by construction, we only test voxels that are (at least partially) within T's bounding box, this check is redundant and can be omitted.

Plane overlap. Subsequently, we have to test whether the plane of T overlaps V. Suppose that T has the three vertices $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$, all specified in voxel-space coordinates, and that V is located at index (x, y, z) within the voxel grid, thus corresponding to the box defined by the two voxel-space corners $\mathbf{x} = [x, y, z]$ and $\mathbf{x}' = [x+1, y+1, z+1]$. We then compute the normal $\mathbf{n}$ of T and determine that pair of opposing voxel corners $(\mathbf{c}_1, \mathbf{c}_2)$ that best aligns with $\mathbf{n}$ (see Figure 2.6):

$$c_{1,x} = \begin{cases} 1, & n_x > 0, \\ 0, & n_x \le 0, \end{cases} \qquad c_{1,y} = \begin{cases} 1, & n_y > 0, \\ 0, & n_y \le 0, \end{cases} \qquad c_{1,z} = \begin{cases} 1, & n_z > 0, \\ 0, & n_z \le 0; \end{cases}$$

$$c_{2,x} = 1 - c_{1,x}, \qquad c_{2,y} = 1 - c_{1,y}, \qquad c_{2,z} = 1 - c_{1,z}.$$

Note that these corners are expressed relative to $\mathbf{x}$. If and only if they lie on different sides of the triangle's plane (or one lies exactly on the plane), the plane

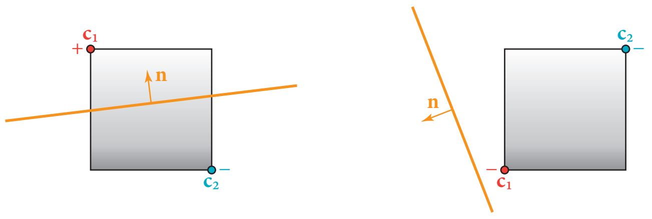


Figure 2.6. For testing whether a plane overlaps a voxel, the two opposing voxel corners $\mathbf{c}_1$ and $\mathbf{c}_2$ for which the vector $\mathbf{c}_1 - \mathbf{c}_2$ best aligns with the plane normal $\mathbf{n}$ are determined. An overlap occurs if they are located in different half spaces of the plane (as on the left).

overlaps V. This can be easily checked by inserting the two voxel extrema $\mathbf{x}+\mathbf{c}_1$ and $\mathbf{x}+\mathbf{c}_2$ into the plane equation and comparing the results' signs:

$$\big(\mathbf{n}\cdot(\mathbf{x}+\mathbf{c}_1-\mathbf{v}_0)\big)\big(\mathbf{n}\cdot(\mathbf{x}+\mathbf{c}_2-\mathbf{v}_0)\big) = (\mathbf{n}\cdot\mathbf{x}+d_1)(\mathbf{n}\cdot\mathbf{x}+d_2) \leq 0, \tag{2.1}$$

where $d_k = \mathbf{n}\cdot(\mathbf{c}_k - \mathbf{v}_0)$. If the signs differ, the product of the results is negative; it is zero if one of the corners is located on the plane.

2D triangle/box overlap. Finally, we have to test whether the triangle and the voxel overlap in all three, mutually orthogonal, 2D main projections (xy, xz, and yz). Such a 2D test can be efficiently realized with *edge functions* [Pineda 88]. An edge function is simply (the left-hand side of) a 2D line equation for one triangle edge:

$$e_i(\mathbf{p}) = \mathbf{m}_i \cdot (\mathbf{p} - \mathbf{w}_i), \tag{2.2}$$

where $i \in \{0,1,2\}$ is the index of the edge going from $\mathbf{w}_i$ to $\mathbf{w}_{(i+1) \bmod 3}$, $\{\mathbf{w}_i\}$ are the 2D triangle's vertices, and $\mathbf{m}_i$ denotes the edge's normal. This normal points to the inside of the triangle and is given by

$$\mathbf{m}_i = [w_{(i+1) \bmod 3,y} - w_{i,y}, w_{i,x} - w_{(i+1) \bmod 3,x}]$$

if the triangle is oriented clockwise and by

$$\mathbf{m}_i = [w_{i,y} - w_{(i+1) \bmod 3,y}, w_{(i+1) \bmod 3,x} - w_{i,x}]$$

in the case of counterclockwise orientation. Consequently, a point $\mathbf{p}$ is inside a triangle if the edge functions for all three edges yield a nonnegative result.

Taking the xy-projection as a concrete example, the triangle's 2D projection T^{xy} is given by the vertices $\mathbf{w}_i = [v_{i,x}, v_{i,y}]$. Its orientation can easily be determined by checking the triangle's normal $\mathbf{n}$; if $n_z > 0$, it is oriented counterclockwise. The voxel's 2D footprint V^{xy} corresponds to the box with corners $\mathbf{b} = [x,y]$ and $\mathbf{b}' = [x+1, y+1]$.

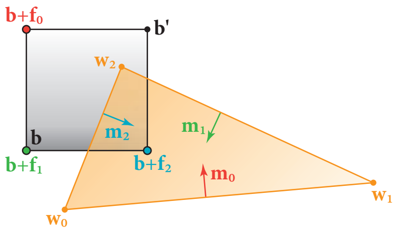


Figure 2.7. To determine whether a box (defined by corners $\mathbf{b}$ and $\mathbf{b}'$) and a triangle (with vertices $\mathbf{w}_0$, $\mathbf{w}_1$, and $\mathbf{w}_2$) overlap, for each triangle edge, we check whether the corresponding critical box corner $\mathbf{b} + \mathbf{f}_i$, implied by the edge's normal $\mathbf{m}_i$, is on the interior side of the edge line.

To test whether T^{xy} and V^{xy} overlap, we determine for each edge i that corner $\mathbf{f}_i$ of V^{xy} to which the edge's normal $\mathbf{m}_i$ points, that is,

$$f_{i,x} = \begin{cases} 1, & m_{i,x} > 0, \\ 0, & m_{i,x} \le 0, \end{cases} \qquad f_{i,y} = \begin{cases} 1, & m_{i,y} > 0, \\ 0, & m_{i,y} \le 0 \end{cases}$$

(see Figure 2.7), and evaluate the edge function for it. It is easy to see that only if $\mathbf{f}_i$ is on the interior side of the edge, that is, $e_i(\mathbf{b} + \mathbf{f}_i) \ge 0$, can there be any overlap of T^{xy} and V^{xy}. Hence, we have to check whether this is fulfilled for all three edges. Actually, it turns out that this is also sufficient for an overlap to occur if the bounding box of T^{xy} and V^{xy} overlap, which is always the case in our setup.

Single-triangle, many-voxel testing. Since, in general, a triangle is tested against multiple voxels for overlap, it is reasonable to compute all quantities that depend only on the triangle in the beginning and then to reuse them for the individual voxel overlap tests. Employing the reformulation

$$e_i(\mathbf{b} + \mathbf{f}_i) = \mathbf{m}_i \cdot (\mathbf{b} + \mathbf{f}_i - \mathbf{w}_i) = \mathbf{m}_i \cdot \mathbf{b} + g_i,$$

with $g_i = \mathbf{m}_i \cdot (\mathbf{f}_i - \mathbf{w}_i)$, this means that in a setup phase, the triangle normal $\mathbf{n}$, the distances d_1 and d_2, as well as the quantities $\mathbf{m}_i$ and g_i (for $i = 0, 1, 2$ and each of the three projections xy, xz, and yz) are determined. The actual per-voxel overlap test then merely requires checking whether the expression in Equation (2.1) holds and if $\mathbf{m}_i \cdot \mathbf{b} + g_i \ge 0$ for all edges and 2D projections.

Simplifications. In several situations, it is possible to simplify the described triangle/voxel overlap test and thus reduce its cost. For instance, if the bounding box of the triangle (before clipping to the voxel grid) covers just a one-voxel-thick line of voxels, all voxels are overlapped by the triangle and no further per-voxel test is necessary. Similarly, if the bounding box corresponds to a one-voxel-thick voxel grid slice, only the corresponding 2D triangle/box overlap subtest needs to be performed for each voxel.

Candidate voxel selection. But even in the general case, a noticeable simplification is possible by integrating the overlap test with the selection of potentially overlapped voxels. Key is the observation that a triangle's voxelization is basically two-dimensional and that, hence, visiting all voxels in its bounding box, a 3D volume, is wasteful. This leads to the following strategy: First, determine the grid side that best aligns with the triangle's plane. Then look at the corresponding 2D projection, and loop over the voxel columns covered by the triangle's bounding box, applying the according 2D triangle/box overlap subtest to each. If a column passes the test, determine the range of voxels overlapped by the triangle's plane; their number is at most three per column. If the range comprises just a single voxel, it is guaranteed to be overlapped. Otherwise, the voxels are subjected to the two remaining 2D triangle/box overlap subtests to derive their final overlap status. More details are given in our original publication [Schwarz and Seidel 10].

2.4.3 Discussion

In conservative surface voxelization, generally, significantly more voxels are set than in rasterization-based surface voxelization. In Figure 2.3, for instance, increases of 132% and 305% for the torus knot and the Italian maple, respectively, occur (with respect to rasterization from one direction). Moreover, unlike in ordinary rasterization, all voxels overlapped by an edge shared by multiple triangles are set by each of these triangles, further increasing the number of voxel updates. Together with the fact that the overlap test is much more expensive, this causes conservative voxelization to be slower than surface voxelization using the rasterization-based overlap criterion.

Concerning performance, it is important to understand the implications of pursuing a triangle-parallel approach. First, the scene has to feature a large enough number of triangles to provide enough data parallelism. Moreover, the triangles should produce roughly the same amount of work. For instance, if the scene has a ground plane consisting of only two triangles, the two threads dedicated to their voxelization have to process an excessive number of voxels, easily causing all other threads to finish early, leaving many shader cores idle. In the case of such unfavorable configurations, a simple remedy is to tessellate the scene accordingly. Another option is to distribute the voxelization of a single triangle over multiple threads. This can be achieved by a blocking approach, where each triangle is assigned to those macro grid cells it overlaps, or a tiling approach, where each of the three grid front sides is split into coarse tiles, and each triangle is assigned to those tiles it overlaps from that side to which it aligns best. Each macrocell and tile, respectively, is then processed by a thread group, looping over all assigned triangles. An according implementation is left as an exercise for the interested reader.

As DirectCompute offers less opportunities for fine-tuning than CUDA, reaching performance comparable to optimized CUDA-based implementations is often hard to achieve and is partially at the mercy of the runtime and the driver. Hence, the accompanying source code does not focus on utmost performance but on legibility and structural cleanness, hopefully facilitating both adaptations and retargeting to other compute languages or platforms.

2.5 Solid Voxelization with DirectCompute

Obviously, realizing a software voxelizer using DirectCompute is not restricted to surface voxelization. In this section, we demonstrate an according approach for solid voxelization, which combines ideas from our previous solid and hierarchical, sparse solid voxelization algorithms [Schwarz and Seidel 10].

Similar to the rasterization-based algorithm from Section 2.3, we perform a 2D rasterization of the scene from one side of the voxel cube (xz this time, assuming the data layout from Figure 2.4). For each pixel whose center is covered, we determine the first voxel in the corresponding voxel column whose center is below the processed triangle and flip the state of this voxel with an atomic XOR operation. The other voxels in the column that are below this voxel are not flipped; instead, their update is deferred to a separate propagation pass that is executed once all triangles have been rendered (see Figure 2.8). This means that, after the rasterization, a set voxel indicates that all voxels below should be flipped. To obtain the final solid voxelization, these flips are carried out by the subsequent propagation pass. For each voxel column, it visits all voxels from top to bottom, flipping a voxel's state if the preceding voxel is set. This

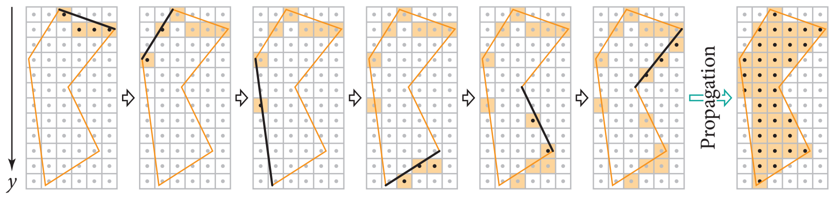


Figure 2.8. During the solid voxelization's initial rasterization pass, in each voxel column covered by a triangle, the state of the first voxel whose center is below the triangle is flipped. The state of the other voxels below is only updated in a subsequent propagation pass, which progresses from top to bottom, flipping a voxel's state if its neighbor directly above is set.

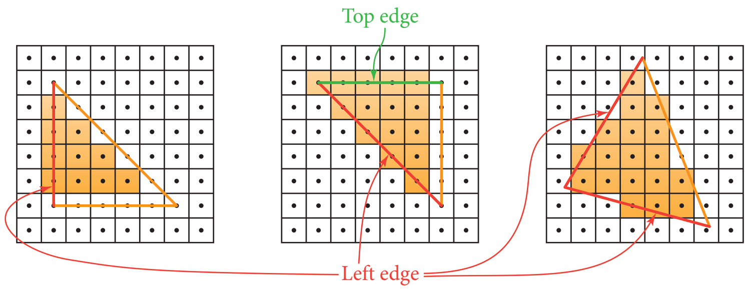


Figure 2.9. In standard rasterization, a pixel is covered by a triangle if its center is located within the triangle. In case the pixel center lies exactly on a triangle edge, it is not considered inside unless the edge is a left or a top edge.

distribution of voxel flips over two passes saves memory bandwidth and reduces the number of atomic update operations and random memory accesses compared to a single-pass approach that directly flips all voxels below, which helps performance.

For the rasterization, we again pursue a triangle-parallel approach, dedicating one thread per triangle. In the compute shader, the triangle's vertices are fetched and transformed, and their bounding box is computed and clipped against the voxel grid. If the bounding box is nonempty, we subsequently loop over all voxel columns (extending in the y-direction) within the xz-extent of the bounding box. For each column, we test in the 2D xz-projection whether the center is overlapped by the triangle. If that is the case, we compute the first voxel in this column whose center is below the triangle and flip the voxel's state.

The 2D overlap test can be efficiently realized by employing the edge functions e_i from Equation (2.2) and evaluating them at the column's 2D center position $\mathbf{q} = [x + \frac{1}{2}, z + \frac{1}{2}]$; if all three results $e_i(\mathbf{q})$ are nonnegative, $\mathbf{q}$ is inside. There is a caveat, though: if $\mathbf{q}$ lies exactly on a triangle's edge, $\mathbf{q}$ is not only inside this triangle but also inside the adjacent triangle with which this edge is shared.[3] Hence, two surface intersections are reported, with the according voxel flips canceling out each other. Except for the case of a silhouette edge, this obviously leads to wrong results because actually only one intersection with the multitriangle surface occurs. A robust remedy is to adopt a consistent fill-rule as employed in hardware rasterization, which assigns an on-edge point to exactly one side of the edge. Using Direct3D's top-left convention, an on-edge point only belongs to a triangle if the edge is a left edge or a horizontal top edge (see Figure 2.9). This can be checked by looking at the edge's normal $\mathbf{m}_i$, leading to the following

[3]Since we are assuming closed, watertight geometry, this is always the case.

overlap criterion:

$$\bigwedge_{i=0}^{2}\Big(\underbrace{e_i(\mathbf{q}) > 0}_{\text{interior}} \vee \Big(\underbrace{e_i(\mathbf{q}) = 0}_{\text{on edge}} \wedge \big(\underbrace{m_{i,x} > 0}_{\text{left edge}} \vee \underbrace{(m_{i,x} = 0 \wedge m_{i,y} < 0)}_{\text{top edge}}\big)\Big)\Big).$$

Since the edge-classification terms solely depend on the edge's normal, they need only be evaluated once. However, for all this to work, it is necessary that when processing the triangles sharing an edge, consistent numerical results are obtained for the common edge. One way to achieve this is to enforce a consistent ordering of an edge's two vertices.

The subsequent propagation pass proceeds slice-wise in the y-direction, operating on all voxel columns in parallel. A single thread simultaneously processes 32 columns that are consecutive in the z-direction. By design, for each y-slice, the voxels of these columns are represented by different bits of the same 32-bit buffer value; this also provides the motivation for selecting to perform rasterization from the xz-side. The employed compute shader loops over all slices from top to bottom. At each slice, it fetches the buffer value encoding the according voxel states for the 32 columns. This value is then XORed with the value for the previous slice, thus propagating state flips, and written back to the buffer.

2.6 Conclusion

This chapter has shown that (and how) GPU-accelerated binary surface and solid voxelization can be realized with Direct3D 11. On the one hand, we explored how the existing rasterization hardware can be harnessed for this task and how to cope with the absence of bitwise blending. The presented solution is characterized by random atomic buffer writes from within a pixel shader to update the voxelization. While accurate results are obtained for solid voxelization, the quality of surface voxelizations suffers from the inappropriate overlap test performed by ordinary rasterization, which is inherent to all such approaches.

Partially motivated by this shortcoming, we also, on the other hand, looked into how the whole voxelization process can be implemented in software using DirectCompute. Pursuing a simple triangle-parallel approach, for each triangle, all potentially affected voxels or voxel columns, respectively, are considered, applying an overlap test for each to determine their state. For surface voxelization, we detailed a triangle/voxel overlap test that yields a conservative surface voxelization, which obviates the deficiencies of its rasterization-based cousin. By contrast, solid voxelization relies on consistent rasterization, and we described an according approach that defers voxel state flips to a separate pass to improve performance. Overall, the covered techniques demonstrate that resorting to a compute-based software implementation for executing the voxelization offers a large degree of flexibility, and it is up to the reader to explore new overlap criteria, load balancing schemes, and voxel grid representations.

Bibliography

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

[Eisemann and Décoret 06] Elmar Eisemann and Xavier Décoret. "Fast Scene Voxelization and Applications." In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2006*, pp. 71–78. New York: ACM Press, 2006.

[Eisemann and Décoret 08] Elmar Eisemann and Xavier Décoret. "Single-Pass GPU Solid Voxelization for Real-Time Applications." In *Proceedings of Graphics Interface 2008*, pp. 73–80. Toronto: Canadian Information Processing Society, 2008.

[Fang and Chen 00] Shiaofen Fang and Hongsheng Chen. "Hardware Accelerated Voxelization." *Computers & Graphics* 24:3 (2000), 433–442.

[Forest et al. 10] Vincent Forest, Loic Barthe, and Mathias Paulin. "Real-Time Hierarchical Binary-Scene Voxelization." *Journal of Graphics, GPU, and Game Tools* 14:3 (2010), 21–34.

[Hasselgren et al. 05] Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. "Conservative Rasterization." In *GPU Gems 2*, edited by Matt Pharr, Chapter 42, pp. 677–690. Reading, MA: Addison-Wesley, 2005.

[Nichols et al. 10] Greg Nichols, Rajeev Penmatsa, and Chris Wyman. "Interactive, Multiresolution Image-Space Rendering for Dynamic Area Lighting." *Computer Graphics Forum* 29:4 (2010), 1279–1288.

[Pineda 88] Juan Pineda. "A Parallel Algorithm for Polygon Rasterization." *Proc. SIGGRAPH '88 (Computer Graphics)* 22:4 (1988), 17–20.

[Reinbothe et al. 09] Christoph K. Reinbothe, Tamy Boubekeur, and Marc Alexa. "Hybrid Ambient Occlusion." In *Eurographics 2009 Annex (Areas Papers)*, pp. 51–57, 2009.

[Schwarz and Seidel 10] Michael Schwarz and Hans-Peter Seidel. "Fast Parallel Surface and Solid Voxelization on GPUs." *ACM Transactions on Graphics* 29:6 (2010), 179:1–179:9.

[Wyman 11] Chris Wyman. "Voxelized Shadow Volumes." In *Proceedings of High Performance Graphics 2011*, 2011.

[Zhang et al. 07] Long Zhang, Wei Chen, David S. Ebert, and Qunsheng Peng. "Conservative Voxelization." *The Visual Computer* 23:9–11 (2007), 783–792.