Hybrid Computational Voxelization Using the Graphics Pipeline

Randall Rauwendaal and Mike Bailey
Oregon State University

Figure 1. The XYZ RGB Asian Dragon voxelized at 128^3, 256^3, and 512^3 resolutions.

Abstract

This paper presents an efficient computational voxelization approach that utilizes the graphics pipeline. Our approach is hybrid in that it performs a precise gap-free computational voxelization, employs fixed-function components of the GPU, and utilizes the stages of the graphics pipeline to improve parallelism. This approach makes use of the latest features of OpenGL and fully supports both conservative and thin-surface voxelization. In contrast to other computational voxelization approaches, our approach is implemented entirely in OpenGL and achieves both triangle and fragment parallelism through its use of geometry and fragment shaders. By exploiting features of the existing graphics pipeline, we are able to rapidly compute accurate scene voxelizations in a manner that integrates well with existing OpenGL applications, is robust across many different models, and eschews the need for complex work/load-balancing schemes.

1. Introduction

The ability to produce a fast and accurate voxelization is highly desirable for many applications, such as intersection computation, hierarchy construction, ambient occlusion, and global illumination. There are many approaches to achieving such voxelizations, which balance tradeoffs between accuracy, speed, and memory consumption. We make a distinction between requirements that are orthogonal to each other, for instance, binary voxelization vs. voxelization that requires blending at active voxels. Naturally, binary voxelization has lower memory requirements, since it is sufficient to use a single bit to indicate whether a voxel is active.

We also consider surface voxelization vs. solid voxelization. Solid voxelization marks any voxel on or within a model as active (and thus requires watertight geometry), whereas surface voxelization considers only those voxels in contact with the surface of the model; this criteria can be further split by defining the separability requirement. A conservative voxelization marks *any* voxel that comes in contact with the surface as active, and is thus 26-separable, while a thin voxelization is 6-separable.

Additionally, since voxelization discretizes a scene into regular volumetric elements, as voxel density increases, the memory requirements of maintaining such a dense data structure become prohibitive, since generally most of the scene consists of empty space. Many approaches attempt to mitigate these high memory requirements by constructing a sparse hierarchical voxel representation that retains the voxel's regular size, but clusters similar regions (empty or solid) into a tree structure, typically an octree.

Initially, we make a distinction between two primary voxelization approaches on the GPU—computational approaches that completely avoid the graphics pipeline ([Schwarz and Seidel 2010; Schwarz 2012], [Pantaleoni 2011]), compared to rasterization-based approaches to voxelization.

Then, we present our hybrid approach to voxelization. While we still utilize the GPU as a massively parallel compute device, we do not abandon the standard graphics pipeline to do so. Instead, we build on its strengths, allowing it to perform the triangle-fragment workload balancing that it does so well with rasterization, and applying this to voxelization. This frees us from having to delve into optimal tiling- and triangle-sorting strategies in order to balance an inherently unbalanced workload of non-uniform triangles.

In this paper, we touch upon many voxelization techniques. In Section 2, we cover the relevant work in the field. In Section 3, we discuss first triangle-parallel and fragment-parallel approaches and then how we combine them for our hybrid implementation. Additionally, we discuss several voxel-list construction methods as well as a method to correctly interpolate attributes using barycentric coordinates. This is followed by our results, Section 4, a discussion of our findings and potential future work, Section 5, and our conclusions, Section 6.

2. Related Work

2.1. Graphics Pipeline

The various approaches to voxelization take many forms and each approach must balance several properties. One of the earlier approaches to utilize the graphics pipeline, Fang and Chen [2000] constructed a surface voxelization by rasterizing the geometry for each voxel slice while clamping the viewport to each slice. Li et al. [2005] introduced "depth peeling," which reduced the number of rendering passes by capturing one level of surface-depth complexity per render pass. These approaches tend to miss voxels and often must be applied once along each orthogonal plane to capture missed geometry. Dong et al. [2004] utilized binary encoding to store voxel occupancy in separate bits of multi-channel render targets, allowing them to process multiple voxel slices in a single rendering pass. This latter approach is sometimes referred to as a *slicemap* [Eisemann and Décoret 2006].

Another approach, such as conservative voxelization [Zhang et al. 2007], uses the conservative rasterization technique of Hasselgren et al. [2005]. This approach amplified single triangles to as many as nine triangles by expanding triangle vertices to pixel-sized squares and outputting the convex hull of the resultant geometry. Sintorn et al. [2008] improved on this method by ensuring that fewer triangles be generated during triangle expansion, while Hertel et al. [2009] found it was most effective to simply expand triangles by half the diagonal of a pixel and discard extra fragments in the fragment shader.

Some voxelization techniques also target solid voxelization; generally, these must restrict their input geometry to closed, watertight models and classify voxels as either interior or exterior. As surface geometry is voxelized, entire columns of voxels are set. The final classification is based on the count, or parity, of the voxel, an odd value indicating a voxel as interior, and an even value indicating exterior. In GPU-hardware, this corresponds to applying a logical XOR, which is supported by the frame buffer. Fang and Chen [2000] presented such an approach using slice-wise rendering, while Eisemann and Décoret [2008] developed a high-performance single-pass approach.

Most recently, Crassin and Green [2012] have developed an approach that operates similarly to the fragment-parallel component of our scheme discussed in Section 3.3, exploiting the recently exposed ability to perform random texture writes in OpenGL using the image API. By constructing an orthographic projection matrix per-triangle in the geometry shader, they were able to rely on the OpenGL rasterizer to voxelize their geometry.

2.2. Computational Voxelization

Recently, approaches have been developed that take an explicitly computational approach to voxelization without utilizing fixed-function hardware. Schwarz and Sei-

del [2010] implemented a triangle-parallel voxelization approach in CUDA, which achieved accurate 6- and 26-separating binary voxelization into a sparse hierarchical octree. Pantaleoni's VoxelPipe implementation [2011] took a similar approach while fully supporting a variety of render targets and robust blending support. Both approaches also employed a tile-based voxelization.

Like the work of Schwarz and Seidel [2010], our approach supports both conservative (26-separating) and thin (6-separating) voxelization. Separability (26 or 6), a topological property defined by Cohen-Or and Kaufman [1995], means that no path of N-adjacent (26 or 6) voxels exists that connects a voxel on one side of the surface with a voxel on the other side. Two voxels are 26-adjacent if they share a common vertex, edge, or face, and 6-adjacent if they share a face. Our ability to support multiple render targets and texture formats like Pantaleoni [2011] is limited only by the restrictions present in the OpenGL image API.

3. Voxelization

Whereas previous techniques relied exclusively on the graphics pipeline, or rejected it completely for a computational approach, we demonstrate how to find a middle ground to use the techniques of computational voxelization within the framework of the graphics pipeline. First, however, we must introduce both the triangle-parallel technique (Section 3.2) and the fragment-parallel technique (Section 3.3); these two techniques make up the primary components of our hybrid approach (Section 3.4). Both techniques use the same 3D extension of the triangle/box overlap tests [Akenine-Möller 2001] as in the work of Schwarz and Seidel [2010] and Pantaleoni [2011]. These approaches differ from each other primarily in their factorization of the computational-overlap testing and the methods in which they try to achieve optimal parallelism.

3.1. Fundamentals

3.1.1. Triangle/Voxel Overlap

The problem of finding an intersection between a triangle $\mathcal{T}$ (with vertices $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$ and edges $\mathbf{e}_i = \mathbf{v}_{(i+1) \mod 3} - \mathbf{v}_i$) and a voxel $\mathbf{p}$ can be approached by first reducing the number of triangle-voxel pairs to consider, and then reducing the computation required to confirm an intersection between a triangle and a voxel. We initially consider the potential intersection between a triangle and the set of all voxels; conceptually, the process is executed in the following order.

1. Reduce the set of potential voxel intersections to only those that overlap the axis-aligned bounding volume $\mathbf{b}$ of the triangle.
2. Iterate over this reduced set of voxels (from $\mathbf{b}_{\min}$ to $\mathbf{b}_{\max}$) and discard any that do not intersect the triangle's plane.

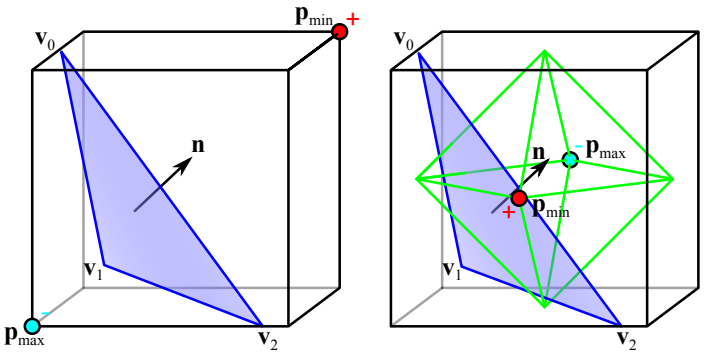

Figure 2. Points $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$ for 26-separable voxelization (left) and for 6-separable voxelization (right). Note that for 6-separable voxelization, we are actually testing for intersection of the diamond shape inscribed inside the voxel as compared to the entire voxel in the 26-separable case.

3. If the triangle's plane divides the voxels, test all three of its 2D planar projections $\left(\mathcal{T}^{XY}, \mathcal{T}^{YZ}, \mathcal{T}^{ZX}\right)$ to confirm overlap.

The above steps depend upon point-to-plane and point-to-line distance calculations. For example, the plane-overlap test computes the signed distance to the plane from two points on opposite ends of the voxel; let us call these points $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$. If these distances have opposite signs, i.e., $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$ are on opposite sides of the plane, this indicates overlap. The selection of $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$ determines the separability of the resultant voxelization (see Figure 2).

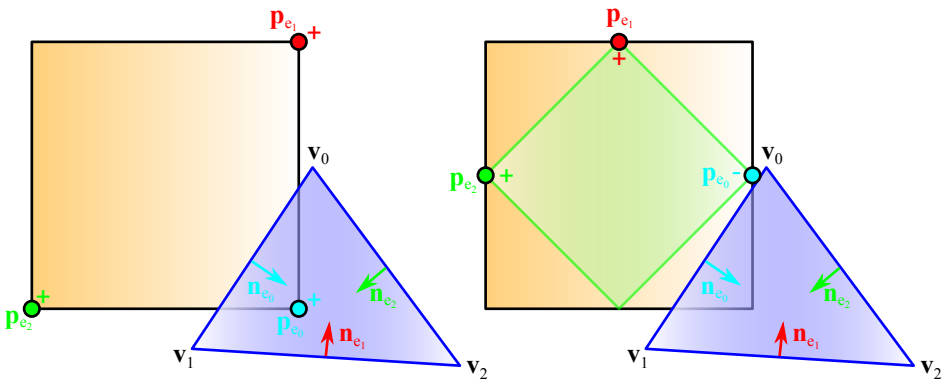

Figure 3. The points $\mathbf{p}_{e_i}$ for 26-separable voxelization (left) and for 6-separable voxelization (right). Similar to the plane-overlap test, the 6-separable voxelization is actually testing against the diamond inscribed inside the voxel's planar projection.

Similarly, when testing the triangle projections $\left(\mathcal{T}^{\mathrm{XY}},\mathcal{T}^{\mathrm{YZ}},\mathcal{T}^{\mathrm{ZX}}\right)$ against their respective voxel projections $\left(\mathbf{p}^{\mathrm{XY}},\ \mathbf{p}^{\mathrm{YZ}},\ \mathbf{p}^{\mathrm{ZX}}\right)$, we use the projected inward-facing edge normals ($\mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}$ for $i=0,1,2$) to select the "most interior" point on the box for each edge ($\mathbf{e}_i^{\mathrm{XY}}$, $\mathbf{e}_i^{\mathrm{YZ}}$, $\mathbf{e}_i^{\mathrm{ZX}}$ for $i=0,1,2$); if all projected edge-to-interior point distances are positive, this indicates overlap within that projection (see Figure 3).

3.1.2. Factorization

As described in previous work [Schwarz and Seidel 2010; Schwarz 2012], the points $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$ and $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i=0,1,2$) are determined with the aid of an offset vector, known as a *critical point*, which is determined by the relevant normal. However, if we take the distance calculations and refactor them such that minimal computation occurs while iterating over the voxels, i.e., factor out all computations not directly dependent on the voxel coordinates of $\mathbf{p}$, we can actually simplify the expressions so that that the critical point and the points $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$ and $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i=0,1,2$) need never be determined. Instead, we substitute per-triangle variables $d_{\min}$, $d_{\max}$ and $d_{\mathbf{e}_i}^{\mathrm{XY}}$, $d_{\mathbf{e}_i}^{\mathrm{YZ}}$, $d_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i=0,1,2$), which represent the factored out components of the distance calculation not dependent on the voxel coordinates.

3.1.3. Optimization

There are several ways in which we can optimize this process with an eye towards reducing the amount of computation that occurs in the innermost loops of our bounding-box traversal.

1. Pre-compute all per-triangle variables, which includes the triangle normal $\mathbf{n}$, the nine planar-projected edge normals $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}$,$\mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i=0,1,2$), and the eleven factored variables $d_{\mathbf{e}_i}^{\mathrm{XY}}$, $d_{\mathbf{e}_i}^{\mathrm{YZ}}$, $d_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i=0,1,2$), $d_{\min}$, and $d_{\max}$.

2. Determine the dominant normal direction and use this to select the orthogonal plane of maximal projection (XY, YZ, or ZX), then iterate over the component axes of this plane first; the remaining axis we shall refer to as the depth axis.

3. Test the 2D projected overlap with the orthogonal plane of maximal projection first.

4. Replace the plane-overlap test with an intersection test along the depth axis to determine the minimal necessary range to iterate over (rather than the entire range of the bounding box along the depth-axis).

5. Test the remaining two planar projections for overlap.

If all of these tests succeed, we can confirm that triangle $\mathcal{T}$ intersects voxel $\mathbf{p}$. Pseudocode for both the conservative and thin voxelization routine is provided in the Appendix (Section 7).

3.2. Triangle-Parallel Voxelization

The most natural approach to voxelization of an input mesh is to parallelize on the input geometry (i.e., the triangles). Schwarz [2012] implemented such an approach in a Direct3D Compute shader as a single pass. Others [Schwarz and Seidel 2010; Pantaleoni 2011] implemented a multi-pass approach to improve parallelism. Schwarz and Seidel [2010] improved coherence by dividing the triangle-box intersection code into nine different voxel-dependent cases; 1D bounding boxes along each axis; 2D bounding boxes in each coordinate plane; and 3D bounding boxes for three dominant normal directions. Unfortunately this requires a two-pass approach, and while it results in high thread coherence (since kernels operate exclusively on similar triangles), it is quite complex and exceeds the number of image units commonly available. However, we can reduce this by a factor of three, allowing all 1D, 2D, and 3D cases to be treated the same by performing a simple transformation, as discussed in Section 3.3.1.

Input geometry is first transformed into "voxel-space," that is the space ranging from $(0,0,0)^{\mathrm{T}}$ to $(\mathbf{V}_{\mathrm{x}},\ \mathbf{V}_{\mathrm{y}},\ \mathbf{V}_{\mathrm{z}})^{\mathrm{T}}$ in the vertex shader. Second, an intersection routine implemented in the geometry shader, as described in Section 3.1.1, performs the voxelization, the performance of which can be seen in Figure 4. It is readily apparent that a naïve triangle-parallel approach only performs well in scenes that exhibit certain characteristics, for instance, the evenly tessellated XYZ RGB Dragon and Stanford Bunny models, both scenes that exhibit even and regular triangulation. For any scene that contains large triangles (such as might be found on a wall) like the

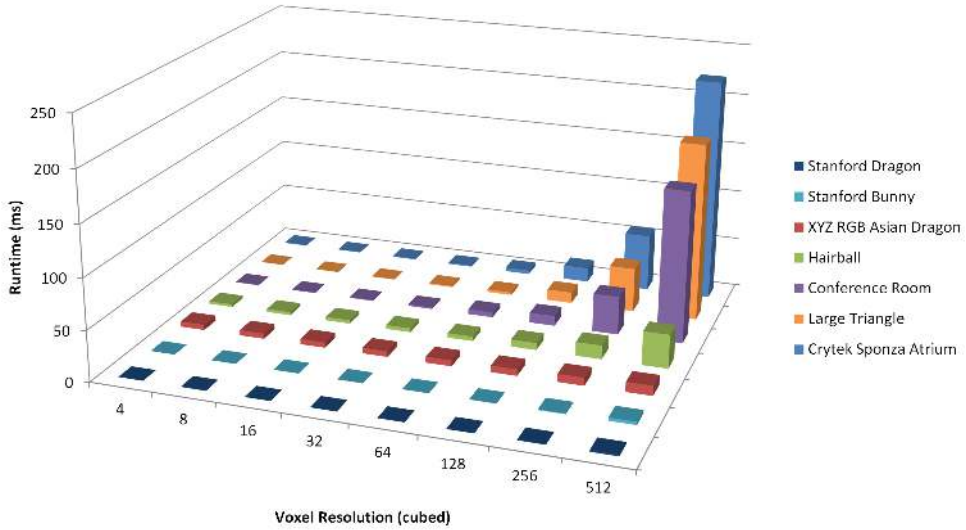

Figure 4. Performance of a naïve triangle-parallel voxelization for various scenes. Poor performance is found for scenes containing large polygons and performance decreases predictably with an increase in voxel resolution.

Crytek Sponza Atrium, the Conference Room, or even, the pathological worst case—a single large scene-spanning triangle—the naïve triangle-parallel approach has no mechanism by which to balance the workload, and the voxelization must wait while individual threads work alone to voxelize large triangles.

3.3. Fragment-Parallel Voxelization

This observation of poor work balance in unevenly tessellated scenes is what led Schwarz and Pantaleoni to introduce complex tile-assignment and sorting stages to their voxelization pipelines. Our fragment-parallel voxelization is based on the observation that much of our triangle-intersection routine can simply be moved to the fragment shader, providing the opportunity for vastly more parallelism. Thus, we exploit the fragment stage of the OpenGL pipeline as a sort of ad-hoc single-level of dynamic parallelism. There are several implementation particulars required to ensure a gap-free voxelization, which will be discussed in a later section. The performance results of our single-pass fragment-parallel implementation can be observed in Figure 5; most noteworthy is the fact that it performs very well on the exact scenes with which the triangle-parallel voxelization struggled and most poorly on scenes with large amounts of fine detailed geometry (XYZ RGB Dragon & Hairball).

The fragment-parallel implementation is far more unique and must be adapted to the pipeline in order to produce a correct voxelization. (Crassin and Green [2012] de-

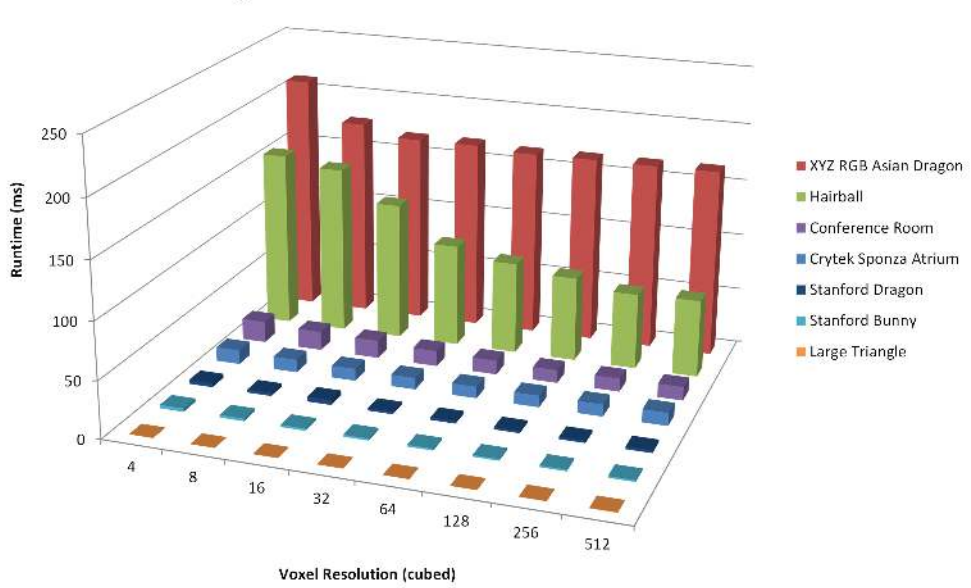

Figure 5. Performance of fragment-parallel voxelization. This technique exhibits poor performance in scenes with large numbers of small triangles. Performance degradation is exacerbated as the ratio of voxel-size to triangle-size increases.

scribe a similar approach.) Our utilization of the fragment stage allows us to benefit from the rasterization and interpolation acceleration provided by the graphics hardware. However, there are two issues we must concern ourselves with when endeavoring to produce a "gap-free" voxelization: (1) gaps within triangles caused by an overly oblique "camera" angle, and (2) gaps between triangles caused by OpenGL's rasterization rules.

As in the triangle-parallel approach values $\mathbf{n}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}$, $d_{\mathbf{e}_i}^{\mathrm{XY}}$, $d_{\mathbf{e}_i}^{\mathrm{YZ}}$, $d_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i = 0, 1, 2$), $d_{\min}$, and $d_{\max}$ are precomputed. However, in this implementation they are calculated in the geometry shader and passed as `flat` non-varying attributes to the fragment shader. Essentially, we allow the rasterizer to take over iterating over the axes of the dominant planar projection, leaving the fragment shader to confirm overlap with the dominant plane, calculate the depth-intersection range according to the desired separability rules, and confirm the remaining two planar projections. In the pseudocode in the Appendix, lines 15–20 in Figure 17 and lines 14–20 in Figure 18 would be moved into the fragment shader.

3.3.1. Gap-Free Triangles

We can solve the first problem listed above (gaps within triangles caused by overly oblique camera angles), illustrated in Figure 6, in one of two ways, both of which rely on determining the dominant normal direction of the triangle. The first approach relies on constructing an orthographic projection matrix per-triangle, which views the triangle against the axis of its maximum projection as determined by the dominant normal direction. Alternately, we can change the input geometry, again based on the dominant normal direction, such that the XY plane is always the axis of maximum projection. This can be accomplished by a simple hardware-supported

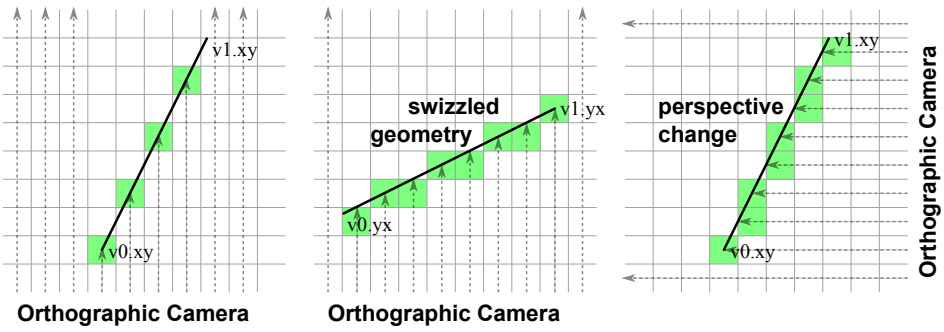

Figure 6. Naïve rasterization on input geometry can lead to gaps in the voxelization. This can be solved in two ways: the center image demonstrates swizzling the vertices of the input geometry, while the image on the right demonstrates changing the projection matrix.

vector swizzle:

$$\forall_{i=0}^{2}\mathbf{v}_{i,\mathrm{xyz}} = \begin{cases} \mathbf{v}_{i,\mathrm{yzx}}, & \mathbf{n}_{\mathrm{x}} \text{ dominant;} \\ \mathbf{v}_{i,\mathrm{zxy}}, & \mathbf{n}_{\mathrm{y}} \text{ dominant;} \\ \mathbf{v}_{i,\mathrm{xyz}}, & \mathbf{n}_{\mathrm{z}} \text{ dominant.} \end{cases}$$

However, we must be sure to "unswizzle" when storing in the destination texture. Additionally, a similar triangle-swizzling approach can be used to reduce the number of cases taken in the approach of Schwarz and Seidel [2010]. With triangle swizzling, the number of cases drops from nine to three, one for each of the 1D, 2D, and 3D cases. Figure 7 depicts the selection of the largest triangle projection based on the dominant normal direction.

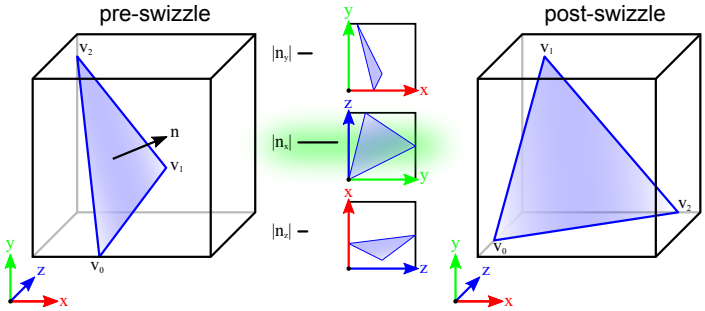

Figure 7. The largest component of the normal **n** of the original triangle determines the plane of maximal projection (XY, YZ, or ZX) and the corresponding swizzle operation to perform.

3.3.2. Conservative Rasterization

The second problem (gaps between triangles caused by OpenGL's rasterization rules) illustrated in Figure 6 can be solved with conservative rasterization. This technique ensures that every pixel that touches a triangle is rasterized, which is counter to how the hardware rasterizer works. There are several approaches to overcome this problem, which generally involve "dilating" the input triangle. Hasselgren et al. [2005] dilated input triangles by expanding triangle vertices into pixel-sized squares and computing the convex hull of the resultant geometry. Tessellation of this shape can be computed in the geometry shader. Hasselgren also proposed computing the bounding triangle of the dilated geometry from the previous approach and simply discarding in a fragment shader all fragments outside of the AABB. Hertel et al. [2009] proposed a similar approach, computing the dilated triangle $\mathcal{T}'$ by constructing a triangle of intersecting lines parallel to the sides of the original triangle $\mathcal{T}$ at a distance of l, where l is half the length of the pixel diagonal (see Figure 8 for examples of these techniques).

With the Hertel approach, the dilated vertices $\mathbf{v}_i'$ of $\mathcal{T}'$ can be easily computed as

$$\mathbf{v}_i' = \mathbf{v}_i + l\left(\frac{\mathbf{e}_{i-1}}{\mathbf{e}_{i-1}\cdot\mathbf{n}_{\mathbf{e}_i}} + \frac{\mathbf{e}_i}{\mathbf{e}_i\cdot\mathbf{n}_{\mathbf{e}_{i-1}}}\right).$$

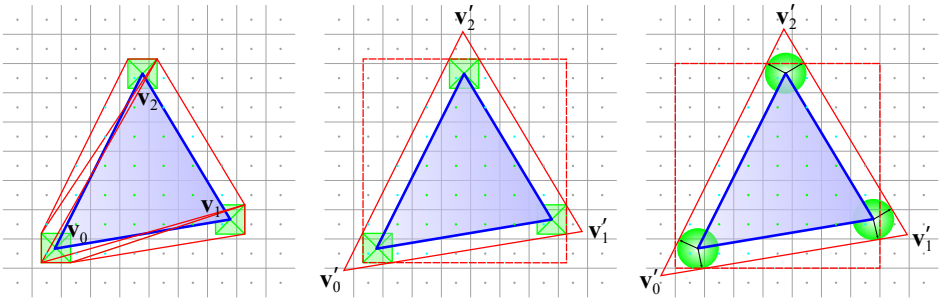

Figure 8. Various conservative rasterization techniques required in order to produce a "gap-free" voxelization. The first two images are from [Hasselgren et al. 2005]; the leftmost image shows the approach of expanding triangle vertices to the size of a pixel and tessellating the resultant convex-hull; the middle image simply creates the minimal triangle to encompass the expanded vertices, and relies on clipping to occur later in the pipeline. The rightmost approach is from [Hertel et al. 2009] and simply expands the triangle by half the length of the pixel diagonal and also relies on clipping to remove unwanted pixels.

In our case, working on a 2D triangle projection in a premultiplied voxel space l will always be $\sqrt{2}/2$.

It should be noted that conservative rasterization has the potential to produce unnecessary overhead in the form of fragment threads that are ultimately rejected in the final voxelization intersection test. As triangles get smaller and l remains constant, the size of the dilated triangle $\mathcal{T}'$ compared to the size of the original triangle $\mathcal{T}$ causes the ratio $\frac{\text{area}(\mathcal{T})}{\text{area}(\mathcal{T}')}$ to become smaller. This ratio can be used to approximate an upper bound on the expected efficiency of per-triangle fragment thread utilization. This goes part of the way in explaining the fragment-parallel technique's poor performance in highly tessellated scenes with many small triangles, which is actually exacerbated further by poor quad utilization for small triangles. Since texture derivatives require

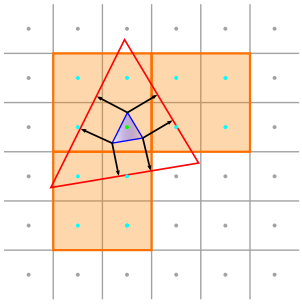

Figure 9. Sub-voxel-sized triangle exhibiting thread utilization of only $8.\bar{3}\%$ after triangle dilation. Note, that this can actually get much worse depending on the triangle configuration.

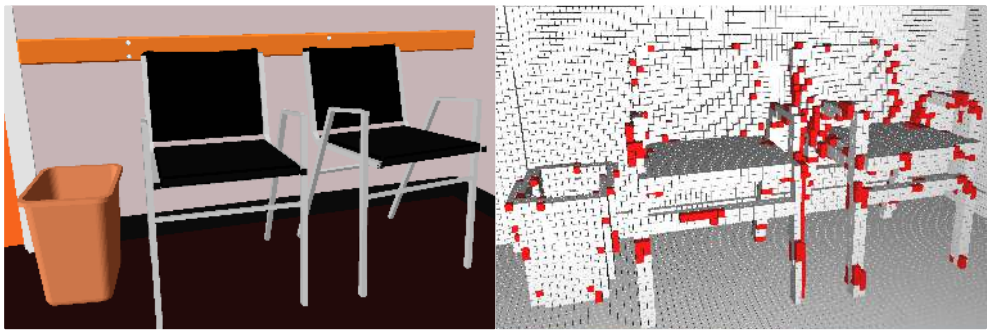

Figure 10. Thin (6-separable) voxelization of the Conference Room scene illustrating false positives (in red) resulting from a naïve conservative-rasterization based voxelization.

neighbor information, even if only one pixel of a quad is covered, the entire quad is launched. This means that triangles smaller than a voxel will utilize only 25% of the threads allocated to them *before* triangle dilation is taken into account. After triangle dilation, thread utilization can be significantly worse (see Figure 9) and in scenes with millions of sub-voxel-sized triangles, can lead to massive oversubscription and poor performance

Additionally, it was our observation that voxelization methods that relied purely on raster-based conservative voxelization methods tended to be overly conservative along their edges where clipping against the AABB couldn't help them, resulting in false positives (see Figure 10). Since our approach maintains a computational intersection test inside the fragment shader, these voxels are still culled.

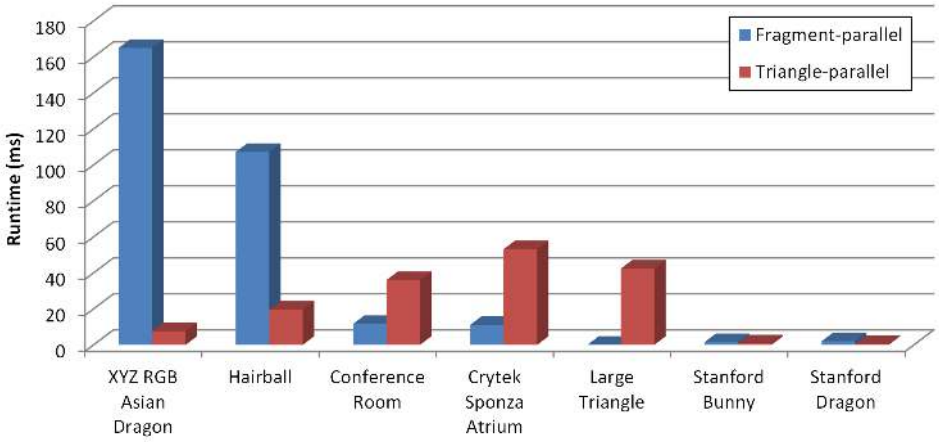

Figure 11. Comparison of the relative performance of triangle-parallel and fragment-parallel techniques. Note, where one technique performs poorly, the other performs well.

3.4. Hybrid Voxelization

The performance of both single-pass techniques is illustrated in Figure 11. When using the fragment shader to increase the available parallelism, the worst-case scenario for the triangle-parallel approach becomes the best case for the fragment-parallel case; conversely, the best-case for the fragment-parallel approach is the worst case for the triangle-parallel approach. We, therefore, developed a hybrid approach, one in which large triangles are divided into fragment-threads using the fragment-parallel technique and small triangles are voxelized using the triangle-parallel technique, thus avoiding poor thread utilization and oversubscription.

We take care to preserve coherent execution among our shader threads with the introduction of a classification stage to our pipeline prior to voxelization (see Figure 12), which outputs corresponding index buffers according to each triangle's classification. These classified index buffers are then used to voxelize the corresponding geometry using the appropriate technique.

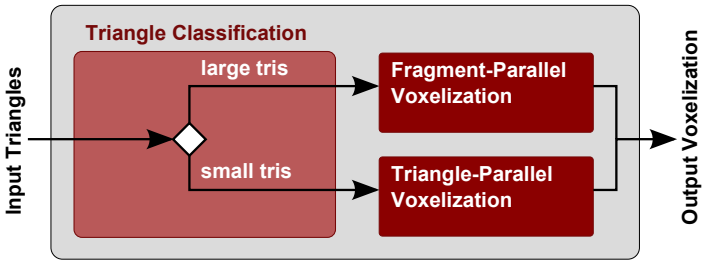

Figure 12. A simple classification routine run before the voxelization stage allows the creation of a hybrid voxelization pipeline and the optimal voxelization approach according to per-triangle characteristics.

3.4.1. Triangle-Selection Heuristic

The main feature of the hybrid-voxelization approach lies in the heuristic used for determining whether a triangle is most suitable for voxelization using a triangle-parallel approach or a fragment-parallel approach. While other approaches [Schwarz and Seidel 2010] are dependent on voxel extents of triangle bounding boxes, we have already determined that the fragment-parallel approach will handle all large triangles, and the triangle-parallel approach will handle all small triangles.

The heuristic for the selection of a cutoff value can be approached in many different ways; for instance, the size of the dilated triangle area $(\mathcal{T}')$ most accurately represents the number of potential voxel intersections to be evaluated in the fragment stage, but it is not a fair representation of the amount of work required in the triangle-parallel stage should the triangle be classified as small. Furthermore, the dilated triangle has a minimum size, which must be considered when undilated triangles approach

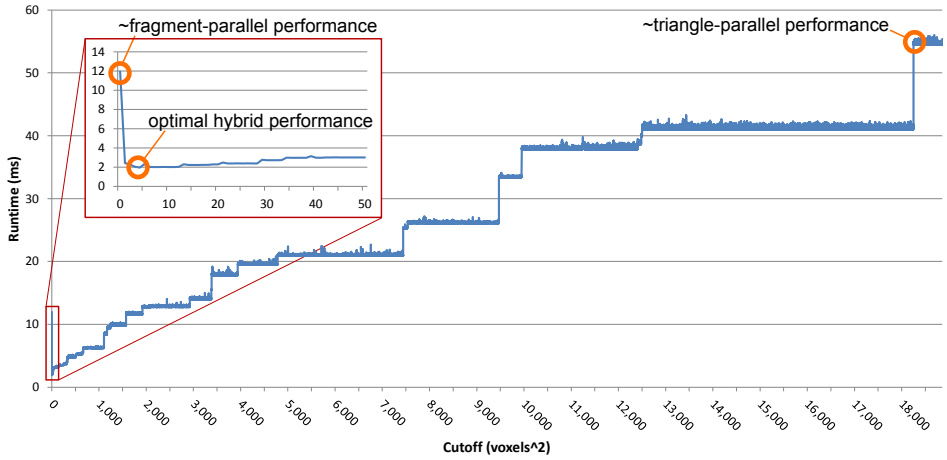

Figure 13. Hybrid voxelization performance of Crytex Sponza Atrium @ 256^3 voxel resolution. Initially at zero, all triangles are classified as "large" and therefore voxelized by the fragment-parallel shader. As the cutoff value (measured in voxel area) increases, triangles are classified and assigned to either the triangle-parallel or fragment-parallel approaches. As the cutoff continues to increase, performance exhibits a stair-step pattern as triangles are reclassified. Eventually, all triangles are classified as "small" and performance reverts to that of the triangle-parallel approach.

zero area. The 3D voxel-extents provide a good indication of the amount of iteration required to voxelize a triangle in the geometry stage; however, since the depth-range is calculated, the 2D-projected voxel-extents provide a closer representation of the actual work performed. Additionally, we could consider the ratio of $\frac{\text{area}(\mathcal{T})}{\text{area}(\mathcal{T}')}$, which, as it varies from 0 to 1, indicates very small to very large triangles, respectively.

In our experiments, we found that simply considering the 2D projected area of the triangle $\mathcal{T}$ worked best, and, for most scenes, a cutoff value of just a few voxel-units squared provided a good starting cutoff value for triangle classification. In Figure 13, we can see the full range of voxelization performance—varying from that of the fragment-parallel approach at a cutoff of zero to the performance of the triangle-parallel approach once the cutoff is large enough to encompass all triangles. Note that Figure 13 represents an unreasonable range of cutoff values; this is meant to illustrate the performance characteristics as the cutoff value changes. Generally, there is a fairly large range of cutoff values corresponding to near-optimal performance.

We are, however, most interested in the cutoff value that will provide the minimal voxelization time, and these values tend to occur at much lower values. Figure 14 shows only the earlier range of cutoff values. Examination of the data confirms that for most inputs a cutoff value of just a few voxels squared provides for optimal voxelization timing. It is conceivable that a bracketing search could determine and adjust this value automatically [Press et al. 2007].

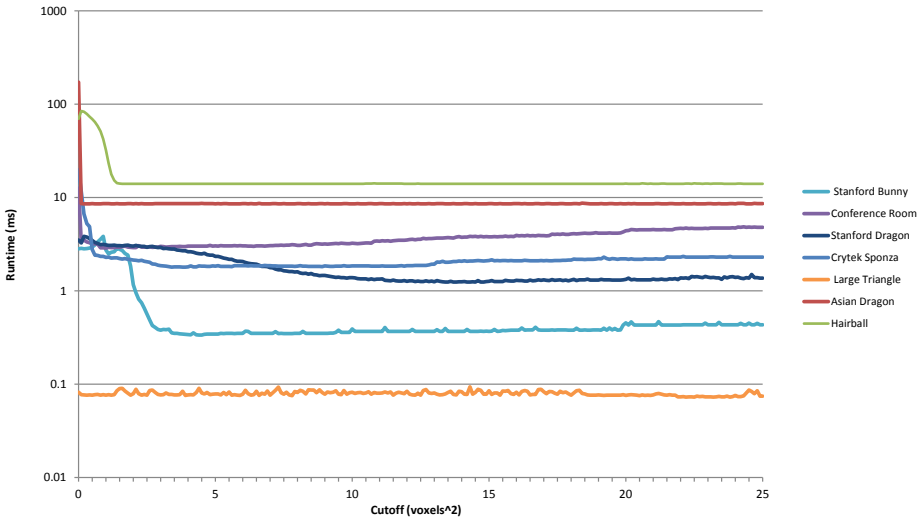

Figure 14. Hybrid voxelization performance @ 256^3 voxel resolution. The logarithmic performance graph of the hybrid voxelization technique displays a lower range of cutoff values such that the optimal cutoff can be clearly discerned.

3.4.2. Optimization

In order to avoid the necessity of separate output buffers for all input attributes, we output only index buffers, which are then used to render only the appropriate subset of the geometry with the voxelization method as determined by the classifier. On many scenes, this allowed us to achieve improved performance over either the fragment-parallel or the triangle-parallel approach alone. However, when we examine the performance of a scene ideally suited to the triangle-parallel approach, like the XYZ RGB Dragon, we observe that the best performance that can be achieved with our triangle-classifier is approximately twice that of the triangle-parallel approach alone. This can be explained by the amount of work it takes to process the seven million triangles in the scene. Each triangle is extremely small (generally less than the size of a voxel) and takes relatively little work to voxelize, and similarly little work to classify. In this case, run-time is dominated by the overhead of creating threads, rather than the work done in each thread, and with our current approach we have doubled the number of threads to be created. Fortunately, we can exploit the fact that in our classification we employ the triangle-parallel approach only for small triangles. Combined with the fact that the number of small triangles in a scene almost always dominates the number of large triangles, we can dramatically decrease the overhead of our hybrid-voxelization pipeline.

As illustrated in Figure 15, by moving the triangle-parallel voxelization into the classification shader and deferring only the larger triangles to be voxelized by the

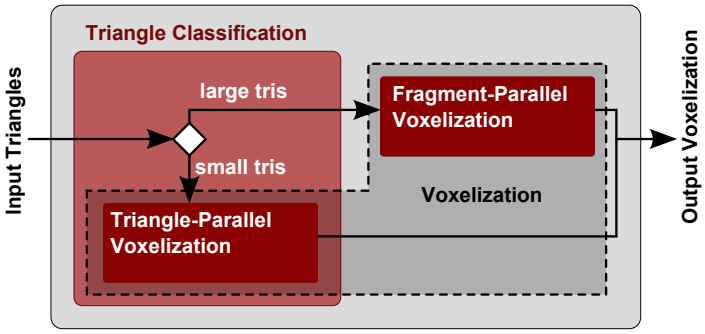

Figure 15. Our final hybrid voxelization implementation mitigates the cost of processing the input geometry twice by immediately voxelizing input triangles classified as "small" and deferring only those triangles considered to be "large."

fragment shader, we effectively reduce a two-pass approach to a just slightly over one-pass approach. This means, that while all triangles are processed at least once, only a few are processed twice. Furthermore, since the overhead of classification and voxelization of small triangles is so low, this makes our hybrid approach competitive on all scenes, even those tailored for a triangle-parallel approach. The full pipeline is shown in Figure 16, illustrating the voxelization of the XYZ RGB Dragon scene.

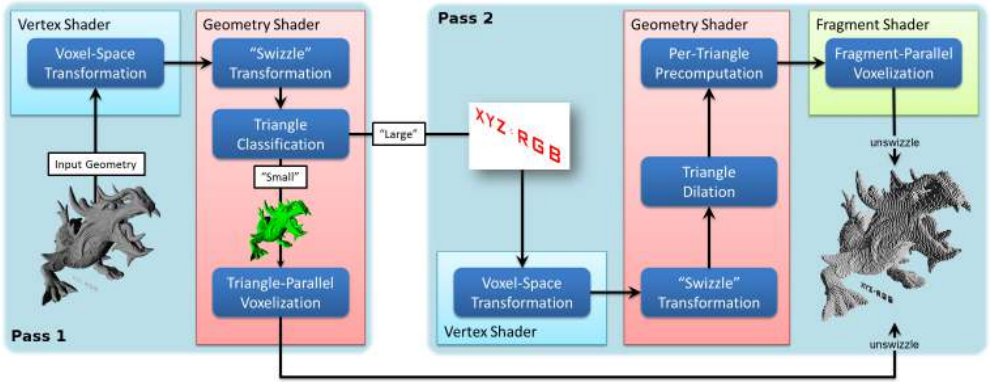

Figure 16. Full pipeline including shader stages. Note, that while there are two "passes," only a very small subset of the geometry that is classified as "large" is processed twice.

3.5. Voxel-List Construction

Though we are primarily concerned with producing a voxelization stored in a dense 3D texture, our technique can also be useful to produce a sparse "voxel-list." Previously, one would have to perform a dense voxelization and then perform a reduction, such as HistoPyramid compaction [Ziegler et al. 2006], in order to produce such a

list. However, with hardware support for atomic operations, this step can now be skipped. We can instead use an atomic counter to increment the index of an output buffer used to store the voxel's coordinates. Crassin and Green [2012] used such a technique to generate their "voxel-fragment-list," which they then used to construct a sparse hierarchical octree. With such an approach, multiple elements may refer to the same voxel location, which are later merged in hierarchy creation. To avoid duplicate voxel assignments, a dense 3D `r32ui` texture can be employed to provide mutexes at each voxel location. By using an `imageAtomicCompSwap` operation at the voxel location, we can restrict incrementing the atomic counter to a single thread accessing the voxel location. This can be beneficial when the voxelization includes additional attribute outputs and there is not enough memory for a dense 3D texture for each attribute.

The reduced memory requirements of voxel-lists must be weighed against increased voxelization time. The use of atomic operations directly impacts voxelization performance, particularly in situations where many threads are attempting to access the same voxel. We observed that the additional voxel culling provided by a rigorous computational intersection test helped significantly in reducing the number of write conflicts for the atomics to resolve. It should be noted that when outputting attribute buffers, on some architectures correct averaging of attribute information (colors, normals, etc.) may require emulation of (as of yet) unsupported atomic operations [Crassin and Green 2012].

3.6. Attribute Interpolation

Attribute interpolation must be handled manually in the triangle-parallel approach. But since it uses the graphics pipeline, the fragment-parallel approach can exploit the fixed-function interpolation hardware provided by the rasterizer. Since the fragment-parallel voxelization method relies on triangle dilation to ensure a conservative voxelization, care must be taken to correctly interpolate triangle attributes across the dilated triangle. To accomplish this, we calculate the barycentric coordinates of the dilated triangle vertices $\mathbf{v}'_i$ with respect to the undilated triangle vertices $\mathbf{v}_i$ using signed area functions:

$$\lambda_i\left(\mathbf{v}'_i\right) = \frac{\text{area}\left(\mathbf{v}'_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}\right)}{\text{area}\left(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\right)}.$$

Applying the barycentric coordinates computed at the dilated triangle vertices $\mathbf{v}'_i$ to the vertex attributes, i.e., vertex colors, normals, or texture coordinates $\mathbf{t}_i$, we can calculate corresponding dilated attributes $\mathbf{t}'_i$:

$$\mathbf{t}'_i = \lambda_0\left(\mathbf{v}'_i\right)\mathbf{t}_0 + \lambda_1\left(\mathbf{v}'_i\right)\mathbf{t}_1 + \lambda_2\left(\mathbf{v}'_i\right)\mathbf{t}_2.$$

By passing dilated attributes in from the geometry shader to the vertex shader in this manner, we ensure that attributes interpolate across the undilated region of the

dilated triangle in the same manner as they would on the undilated triangle; this holds regardless of the dilation factor l applied.

4. Results

We tested our hybrid voxelization approach against several different models at various voxel resolutions and compared the results to purely triangle-parallel and purely fragment-parallel implementations, as well as to the data available from previous approaches [Schwarz and Seidel 2010; Pantaleoni 2011; Crassin and Green 2012]. We included the XYZ RGB Asian Dragon as an example of a pathological worst-case scenario for the fragment-parallel approach; we also included a single scene-spanning triangle as a pathological worst case for the triangle-parallel approach. All results were generated on an Intel Core i7 950 @ 3.07 GHz with an NVIDIA GeForce GTX 480. Table 1 shows the performance comparison of the different techniques, and, additionally, the percentage of time spent in the first and second pass of the hybrid voxelization approach (see Figure 16).

Although both Dragons, the Bunny, and the Hairball represent less than ideal conditions for our approach as they do not have a large distribution of triangle sizes, we are able to obtain better performance than the competing techniques in all but one instance. In several cases, the purely triangle-parallel approach beats the hybrid approach, which is understandable considering that those scenes are ideally suited

Model	**Grid size**	6-separating (thin) binary voxelization						
		Triangle-parallel	Fragment-parallel	Hybrid @voxels^2	Pass 1/Pass 2	Schwarz & Seidel	VoxelPipe	Crassin & Greene (680)
large triangle (1 tri)	128^3	10.62	0.03	0.04 @na	36.1%/63.9%			
	256^3	42.4	0.06	0.07 @na	22.1%/77.9%			
	512^3	169.7	0.22	0.19 @na	12.0%/88.0%			
XYZ RGB Asian Dragon (7,219,045 tris)	128^3	6.37	165.2	8.51 @2.0	99.9%/0.1%	11.36	21.2	
	256^3	7.70	165.0	8.57 @1.7	99.7%/0.3%	14.73		
	512^3	9.80	164.6	10.3 @1.4	99.8%/0.2%	16.67	22.0	
Crytek Sponza Atrium (262,267 tris)	128^3	13.4	10.65	1.11 @2.8	87.7%/12.3%			
	256^3	53.2	11.13	1.80 @3.9	71.6%/28.3%			
	512^3	208.7	11.87	3.68 @3.1	52.8%/47.2%			
Conference (331,179 tris)	128^3	9.23	11.47	1.41 @0.5	68.5%/31.5%	3.9	3.3	
	256^3	36.04	11.62	1.82 @1.7	69.2%/30.8%			
	512^3	141.2	11.94	3.01 @0.9	52.2%/47.8%	59.3	4.3	
Stanford Bunny (69,666 tris)	128^3	0.28	1.58	0.19 @1.8	88.1%/11.9%	0.60		
	256^3	0.82	1.55	0.34 @4.5	91.6%/8.4%	0.89		
	512^3	3.12	1.82	1.08 @12.7	93.0%/7.0%	2.35		
Stanford Dragon (100,000 tris)	128^3	0.25	2.13	0.26 @13.3	97.8%/2.2%	3.44	4.8	1.19
	256^3	0.51	2.09	0.52 @5.9	93.4%/6.6%	3.96		
	512^3	1.61	2.25	1.25 @13.7	88.6%/11.4%	4.44	5.0	1.38
Hairball (2,880,000 tris)	128^3	7.09	74.8	7.37 @2.3	99.89%/0.11%	22.8	12.8	
	256^3	13.73	67.1	14.0 @2.4	99.94%/0.06%			
	512^3	33.47	68.4	33.9 @8.0	99.97%/0.03%	95.0	18.3	

Table 1. Running time (in ms) for different voxelization approaches. Voxelizations are binary and performed into a single-component dense 3D texture. The large triangle cutoff is listed as "na" since there are no suitable triangles to be reassigned.

to the triangle-parallel approach. It should be noted that in all such cases, besides the pathological worst case (the Asian Dragon), the hybrid approach was within 3% of the triangle-parallel approach, indicating the low overhead of our multi-pass approach. Despite its simple classification scheme, our approach provides a performance improvement for binary voxelization over its competitors, including that of Crassin and Green [2012] which used superior hardware (GTX 680). It should be noted that the cutoff values are likely to be highly architecture-dependent; we would expect them to change when executed on Nvidia's Kepler or AMD's Southern Islands architecture.

5. Discussion

We implemented a wide variety of voxelization and conservative rasterization techniques in our experiments. Our implementations targeted the capabilities described in the OpenGL 4.2 specification. Our approach relied on the ability to perform texture-writes to arbitrary locations enabled by the image API. Our classification approach relied on indirect buffers to enable the asynchronous execution of the voxelization stage. A benefit of our OpenGL implementation is that it avoids the performance penalty of context switching and implicit synchronization points present in a CUDA or OpenCL implementation. With the introduction of OpenGL 4.3, the triangle-parallel approach could easily be implemented in a Compute shader, but it remains to be seen if there is an advantage to this.

Another application of our initial classification scheme (see Figure 12), could be to "pre-classify" scenes. Then, by maintaining two index buffers, hybrid-voxelization could be used without the cost of classification. This would be most sensible when applying a non-voxel-dependent triangle classifier in scenarios where the orientation of the voxels may change relative to the scene geometry.

We found that several of our results agreed with other approaches [Sintorn et al. 2008; Hertel et al. 2009] that geometry amplification of the first Hasselgren technique led to performance degradations. We also found that atomic operations more greatly impacted the triangle-parallel approach, likely due to the fact that each triangle-parallel thread is responsible for more writes than each fragment-parallel thread.

Future work could exploit *true* dynamic parallelism facilities currently only available in CUDA 5 to spawn exactly one thread for each triangle/voxel pair. While this would still obviate the need for complex tiling and sorting strategies, it would unfortunately remove the ability to exploit the remaining fixed-function hardware present on the GPU exposed to the graphics pipeline. Additionally, we would like to explore more robust cutoff-prediction strategies, techniques for automatic minimum detection, and more sophisticated classification approaches.

6. Conclusion

This paper has shown how a GPU-accelerated computational surface voxelization can be achieved without using CUDA or OpenCL. Our hybrid approach to voxelization leverages the strengths of the graphics pipeline to improve parallelism where it is most needed without sacrificing the quality of the voxelization. It exhibits superior performance to existing techniques, especially on scenes with non-uniform triangle distributions.

7. Appendix

In this appendix, we provide pseudocode for both conservative (Figure 17) and thin (Figure 18) voxelization routines to clarify their implementation.

function conservativeVoxelize($\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{b}_{\min}, \mathbf{b}_{\max}, \mathit{unswizzle}$)
  $\mathbf{e}_i \leftarrow \mathbf{v}_{(i+1) \mod 3} - \mathbf{v}_i$
  $\mathbf{n} \leftarrow \text{cross}(\mathbf{e}_0, \mathbf{e}_1)$
  $\mathbf{n}_{\mathbf{e}_i}^{\text{XY}} \leftarrow \text{sign}(\mathbf{n}_\text{z}) \cdot (-\mathbf{e}_{i,\text{y}}, \mathbf{e}_{i,\text{x}})^\text{T}$
  $\mathbf{n}_{\mathbf{e}_i}^{\text{YZ}} \leftarrow \text{sign}(\mathbf{n}_\text{x}) \cdot (-\mathbf{e}_{i,\text{z}}, \mathbf{e}_{i,\text{y}})^\text{T}$
  $\mathbf{n}_{\mathbf{e}_i}^{\text{ZX}} \leftarrow \text{sign}(\mathbf{n}_\text{y}) \cdot (-\mathbf{e}_{i,\text{x}}, \mathbf{e}_{i,\text{z}})^\text{T}$
  $d_{\mathbf{e}_i}^{\text{XY}} \leftarrow -\left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{XY}}, \mathbf{v}_{i,\text{xy}} \right\rangle + \max\left(0, \mathbf{n}_{\mathbf{e}_i,\text{x}}^{\text{XY}}\right) + \max\left(0, \mathbf{n}_{\mathbf{e}_i,\text{y}}^{\text{XY}}\right)$
  $d_{\mathbf{e}_i}^{\text{YZ}} \leftarrow -\left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{YZ}}, \mathbf{v}_{i,\text{yz}} \right\rangle + \max\left(0, \mathbf{n}_{\mathbf{e}_i,\text{x}}^{\text{YZ}}\right) + \max\left(0, \mathbf{n}_{\mathbf{e}_i,\text{y}}^{\text{YZ}}\right)$
  $d_{\mathbf{e}_i}^{\text{ZX}} \leftarrow -\left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{ZX}}, \mathbf{v}_{i,\text{zx}} \right\rangle + \max\left(0, \mathbf{n}_{\mathbf{e}_i,\text{x}}^{\text{ZX}}\right) + \max\left(0, \mathbf{n}_{\mathbf{e}_i,\text{y}}^{\text{ZX}}\right)$
  $\mathbf{n} \leftarrow \text{sign}(\mathbf{n}_\text{z}) \cdot \mathbf{n}$ // ensures $z_{\min} < z_{\max}$
  $d_{\min} \leftarrow \langle \mathbf{n}, \mathbf{v}_0 \rangle - \max(0, \mathbf{n}_\text{x}) - \max(0, \mathbf{n}_\text{y})$
  $d_{\max} \leftarrow \langle \mathbf{n}, \mathbf{v}_0 \rangle - \min(0, \mathbf{n}_\text{x}) - \min(0, \mathbf{n}_\text{y})$
  for $\mathbf{p}_\text{x} \leftarrow \mathbf{b}_{\min,\text{x}}, \ldots, \mathbf{b}_{\max,\text{x}}$ **do**
    for $\mathbf{p}_\text{y} \leftarrow \mathbf{b}_{\min,\text{y}}, \ldots, \mathbf{b}_{\max,\text{y}}$ **do**
      if $\forall_{i=0}^{2} \left(\left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{XY}}, \mathbf{p}_\text{xy} \right\rangle + d_{\mathbf{e}_i}^{\text{XY}} \geq 0 \right)$ **then**
        $z_{\min} \leftarrow \max\left(\mathbf{b}_{\min,\text{z}}, \left\lfloor (-\langle \mathbf{n}_\text{xy}, \mathbf{p}_\text{xy} \rangle + d_{\min}) \frac{1}{\mathbf{n}_\text{z}} \right\rfloor\right)$
        $z_{\max} \leftarrow \min\left(\mathbf{b}_{\max,\text{z}}, \left\lceil (-\langle \mathbf{n}_\text{xy}, \mathbf{p}_\text{xy} \rangle + d_{\max}) \frac{1}{\mathbf{n}_\text{z}} \right\rceil\right)$
        for $\mathbf{p}_\text{z} \leftarrow z_{\min}, \ldots, z_{\max}$ **do**
          if $\forall_{i=0}^{2} \left(\left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{YZ}}, \mathbf{p}_\text{xy} \right\rangle + d_{\mathbf{e}_i}^{\text{YZ}} \geq 0 \wedge \left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{ZX}}, \mathbf{p}_\text{xy} \right\rangle + d_{\mathbf{e}_i}^{\text{ZX}} \geq 0 \right)$ **then**
            $V\left[\mathit{unswizzle} \cdot \mathbf{p}\right] \leftarrow$ **true**
end function

Figure 17. Pseudocode for a conservative (26-separable) computational voxelization. This implementation assumes that the inputs, $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{b}_{\min}$, and $\mathbf{b}_{\max}$, are pre-swizzled; *unswizzle* represents a permutation matrix used to get the unswizzled voxel location.

1: **function** thinVoxelize($\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{b}_{\min}, \mathbf{b}_{\max}, \mathit{unswizzle}$)
2: $\quad \mathbf{e}_i \leftarrow \mathbf{v}_{(i+1) \mod 3} - \mathbf{v}_i$
3: $\quad \mathbf{n} \leftarrow \text{cross}(\mathbf{e}_0, \mathbf{e}_1)$
4: $\quad \mathbf{n}_{\mathbf{e}_i}^{\text{XY}} \leftarrow \text{sign}(\mathbf{n}_\text{z}) \cdot \left(-\mathbf{e}_{i,\text{y}}, \mathbf{e}_{i,\text{x}}\right)^\text{T}$
5: $\quad \mathbf{n}_{\mathbf{e}_i}^{\text{YZ}} \leftarrow \text{sign}(\mathbf{n}_\text{x}) \cdot \left(-\mathbf{e}_{i,\text{z}}, \mathbf{e}_{i,\text{y}}\right)^\text{T}$
6: $\quad \mathbf{n}_{\mathbf{e}_i}^{\text{ZX}} \leftarrow \text{sign}(\mathbf{n}_\text{y}) \cdot \left(-\mathbf{e}_{i,\text{x}}, \mathbf{e}_{i,\text{z}}\right)^\text{T}$
7: $\quad d_{\mathbf{e}_i}^{\text{XY}} \leftarrow \left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{XY}}, 0.5 - \mathbf{v}_{i,\text{xy}} \right\rangle + 0.5 \cdot \max\left(\left|\mathbf{n}_{\mathbf{e}_i,\text{x}}^{\text{XY}}\right|, \left|\mathbf{n}_{\mathbf{e}_i,\text{y}}^{\text{XY}}\right|\right)$
8: $\quad d_{\mathbf{e}_i}^{\text{YZ}} \leftarrow \left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{YZ}}, 0.5 - \mathbf{v}_{i,\text{yz}} \right\rangle + 0.5 \cdot \max\left(\left|\mathbf{n}_{\mathbf{e}_i,\text{x}}^{\text{YZ}}\right|, \left|\mathbf{n}_{\mathbf{e}_i,\text{y}}^{\text{YZ}}\right|\right)$
9: $\quad d_{\mathbf{e}_i}^{\text{ZX}} \leftarrow \left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{ZX}}, 0.5 - \mathbf{v}_{i,\text{zx}} \right\rangle + 0.5 \cdot \max\left(\left|\mathbf{n}_{\mathbf{e}_i,\text{x}}^{\text{ZX}}\right|, \left|\mathbf{n}_{\mathbf{e}_i,\text{y}}^{\text{ZX}}\right|\right)$
10: $\quad \mathbf{n} \leftarrow \text{sign}(\mathbf{n}_\text{z}) \cdot \mathbf{n}$ // ensures $z_{\min} < z_{\max}$
11: $\quad d_{\text{cen}} \leftarrow \langle \mathbf{n}, \mathbf{v}_0 \rangle - 0.5 \cdot \mathbf{n}_\text{x} - 0.5 \cdot \mathbf{n}_\text{y}$
12: $\quad$ **for** $\mathbf{p}_\text{x} \leftarrow \mathbf{b}_{\min,\text{x}}, \ldots, \mathbf{b}_{\max,\text{x}}$ **do**
13: $\quad\quad$ **for** $\mathbf{p}_\text{y} \leftarrow \mathbf{b}_{\min,\text{y}}, \ldots, \mathbf{b}_{\max,\text{y}}$ **do**
14: $\quad\quad\quad$ **if** $\forall_{i=0}^{2} \left(\left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{XY}}, \mathbf{p}_{\text{xy}} \right\rangle + d_{\mathbf{e}_i}^{\text{XY}} \geq 0 \right)$ **then**
15: $\quad\quad\quad\quad z_{\text{int}} \leftarrow \left(-\langle \mathbf{n}_{\text{xy}}, \mathbf{p}_{\text{xy}} \rangle + d_{\text{cen}}\right) \frac{1}{\mathbf{n}_\text{z}}$
16: $\quad\quad\quad\quad z_{\min} \leftarrow \max\left(\mathbf{b}_{\min,\text{z}}, \lfloor z_{\text{int}} \rfloor\right)$
17: $\quad\quad\quad\quad z_{\max} \leftarrow \min\left(\mathbf{b}_{\max,\text{z}}, \lceil z_{\text{int}} \rceil\right)$
18: $\quad\quad\quad\quad$ **for** $\mathbf{p}_\text{z} \leftarrow z_{\min}, \ldots, z_{\max}$ **do**
19: $\quad\quad\quad\quad\quad$ **if** $\forall_{i=0}^{2} \left(\left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{YZ}}, \mathbf{p}_{\text{xy}} \right\rangle + d_{\mathbf{e}_i}^{\text{YZ}} \geq 0 \wedge \left\langle \mathbf{n}_{\mathbf{e}_i}^{\text{ZX}}, \mathbf{p}_{\text{xy}} \right\rangle + d_{\mathbf{e}_i}^{\text{ZX}} \geq 0 \right)$ **then**
20: $\quad\quad\quad\quad\quad\quad V\left[\mathit{unswizzle} \cdot \mathbf{p}\right] \leftarrow$ **true**
21: **end function**

Figure 18. Pseudocode for a thin (6-separable) computational voxelization. This implementation assumes that the inputs, $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{b}_{\min}$, and $\mathbf{b}_{\max}$, are pre-swizzled; *unswizzle* represents a permutation matrix used to get the unswizzled voxel location.

Acknowledgments

This work was funded by a gift from Intel Corporation's Visual Computing Academic Program. Additionally, we would like to thank the Stanford University Computer Graphics Laboratory for the Dragon and Bunny models, and Crytek for its improved version of the Sponza Atrium model originally created by Marko Dabrovic. We also thank Anat Grynberg and Greg Ward for the Conference Room model. Special thanks go also to Patrick Neill for his valuable review.

References

AKENINE-MÖLLER, T. 2001. Fast 3D Triangle-Box Overlap Testing. *journal of graphics tools 6*, 1, 29–33. 18

COHEN-OR, D., AND KAUFMAN, A. 1995. Fundamentals of Surface Voxelization. *Graphical Models and Image Processing 57*, 6, 453–461. 18

CRASSIN, C., AND GREEN, S. 2012. Octree-based sparse voxelization using the gpu hardware rasterizer. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. A K Peters/CRC Press, Boston, MA. 17, 22, 31, 32, 33

DONG, Z., CHEN, W., BAO, H., ZHANG, H., AND PENG, Q. 2004. Real-Time Voxelization for Complex Polygonal Models. In *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, Washington, DC, USA, 43–50. 17

EISEMANN, E., AND DÉCORET, X. 2006. Fast Scene Voxelization and Applications. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games. I3D*, ACM, New York, NY, USA, 71–78. 17

EISEMANN, E., AND DÉCORET, X. 2008. Single-Pass GPU Solid Voxelization for Real-Time Applications. In *Proceedings of Graphics Interface 2008*, Canadian Information Processing Society, Toronto, Ont., Canada, 73–80. 17

FANG, S., AND CHEN, H. 2000. Hardware Accelerated Voxelization. *Computers and Graphics 24*, 3, 433–442. 17

HASSELGREN, J., AKENINE-MÖLLER, T., AND OHLSSON, L. 2005. Conservative Rasterization. In *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley, Reading, MA, 677–690. 17, 24, 25

HERTEL, S., HORMANN, K., AND WESTERMANN, R. 2009. A Hybrid GPU Rendering Pipeline for Alias-Free Hard Shadows. In *Eurographics 2009 Areas Papers*, Eurographics Association, Aire-la-Ville, Switzerland, 59–66. 17, 24, 25, 33

LI, W., FAN, Z., WEI, X., AND KAUFMAN, A. 2005. GPU-Based Flow Simulation with Complex Boundaries. In *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley, Reading, MA, 747–764. 17

PANTALEONI, J. 2011. VoxelPipe: A Programmable Pipeline for 3D Voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, 99–106. 16, 18, 21, 32

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 2007. *Numerical Recipes: The Art of Scientific Computing*, 3 ed. Cambridge University Press, New York, NY, USA. 28

SCHWARZ, M., AND SEIDEL, H.-P. 2010. Fast Parallel Surface and Solid Voxelization on GPUs. *ACM Transactions on Graphics 29*, 6 (Proceedings of SIGGRAPH Asia 2010) (Dec.), 179:1–179:9. 16, 18, 20, 21, 24, 27, 32

SCHWARZ, M. 2012. Practical Binary Surface and Solid Voxelization with Direct3D 11. In *GPU Pro 3: Advanced Rendering Techniques*, W. Engel, Ed. A K Peters/CRC Press, Boca Raton, FL, USA, 337–352. 16, 20, 21

SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample Based Visibility for Soft Shadows using Alias-Free Shadow Maps. In *Proceedings of the Nineteenth Eurographics Symposium on Rendering (EGSR '08)*, Eurographics Association, Aire-la-Ville, Switzerland, 1285–1292. 17, 33

ZHANG, L., CHEN, W., EBERT, D. S., AND PENG, Q. 2007. Conservative Voxelization. *Vis. Comput. 23*, 9, 783–792. 17

ZIEGLER, G., TEVS, A., THEOBALT, C., AND SEIDEL, H.-P. 2006. On-the-Fly Point Clouds through Histogram Pyramids. In *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)*, European Association for Computer Graphics (Eurographics), Aire-la-Ville, Switzerland, 137–144. 30

Author Contact Information

Randall Rauwendaal
Oregon State University
2121 Kelley Engineering Center
Corvallis, OR 97331
randall@rauwendaal.net
http://www.rauwendaal.net

Mike Bailey
Oregon State University
2117 Kelley Engineering Center
Corvallis, OR 97331
mjb@eecs.oregonstate.edu
http://eecs.oregonstate.edu/people/bailey

Randall Rauwendaal and Mike Bailey, Hybrid Computational Voxelization Using the Graphics Pipeline, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 1, 15–37, 2013
http://jcgt.org/published/0002/01/02/

Received:	2012-11-01		
Recommended:	2013-01-15	Corresponding Editor:	Chris Wyman
Published:	2013-03-18	Editor-in-Chief:	Morgan McGuire