VoxelPipe: A Programmable Pipeline for 3D Voxelization

Jacopo Pantaleoni*
NVIDIA Research

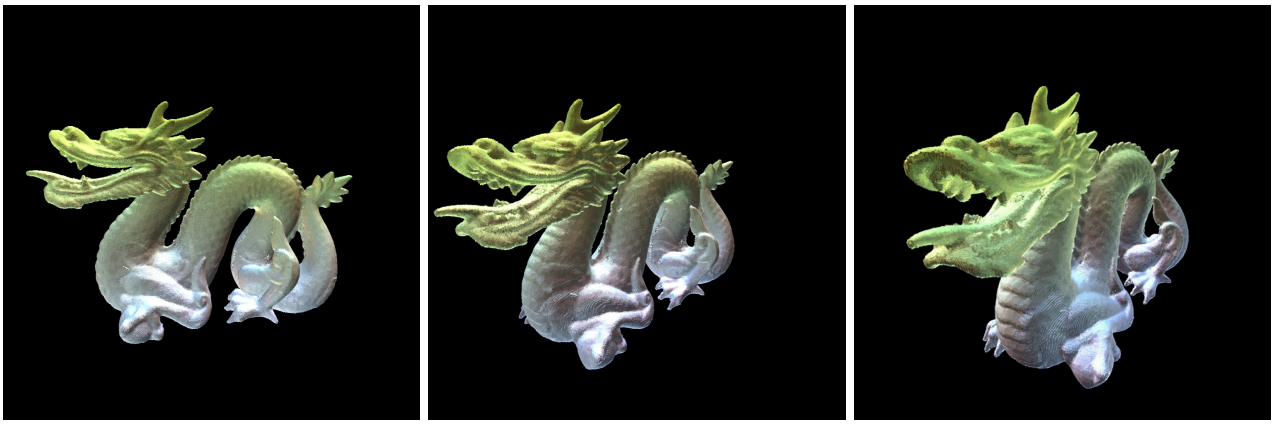

Figure 1: *A rendering of the Stanford Dragon voxelized at a resolution of 512^3, with a fragment shader encoding the surface normal packed in 16 bits. Max-blending has been used to deterministically select a single per-voxel normal, later used for lighting computations in the final rendering pass.*

Abstract

We present a highly flexible and efficient software pipeline for programmable triangle voxelization. The pipeline, entirely written in CUDA, supports both fully conservative and thin voxelizations, multiple boolean, floating point, vector-typed render targets, user-defined vertex and fragment shaders, and a bucketing mode which can be used to generate 3D A-buffers containing the entire list of fragments belonging to each voxel. For maximum efficiency, voxelization is implemented as a sort-middle tile-based rasterizer, while the A-buffer mode, essentially performing 3D binning of triangles over uniform grids, uses a sort-last pipeline. Despite its major flexibility, the performance of our tile-based rasterizer is always competitive with and sometimes more than an order of magnitude superior to that of state-of-the-art binary voxelizers, whereas our bucketing system is up to 4 times faster than previous implementations. In both cases the results have been achieved through the use of careful load-balancing and high performance sorting primitives.

CR Categories: I.3.2 [Graphics Systems C.2.1, C.2.4, C.3)]: Stand-alone systems—; I.3.7 [Three-Dimensional Graphics and Realism]: Color,shading,shadowing, and texture—Raytracing;

Keywords: voxelization, bucketing, binning, programmable pipelines, rasterization, ray tracing

*e-mail: jpantaleoni@nvidia.com

1 Introduction

Voxel representations are widely used across many fields of computational science, engineering and computer graphics, with applications ranging from finite-element stress, sound and radative transfer simulation, to collision detection, rendering [Nichols et al. 2010; Kaplanyan and Dachsbacher 2010] and 3D shape matching. The increasing ability to perform many of these tasks interactively on dynamic models is putting more emphasis on *real-time* scan conversion, or *voxelization*, the process necessary to create a voxel representation from an input surface, typically described as a triangle mesh. This process is in fact itself a compute-intensive application that can take a significant fraction of the overall computing time.

In recent years several algorithms have been proposed to exploit the computing power of massively parallel GPUs to perform triangle mesh voxelization, either relying on the exisiting fixed-function rasterization pipeline or taking advantage of the full programmability offered by CUDA [Nickolls et al. 2008]. While these studies provided custom algorithms focusing on fast and efficient binary voxelization, our work focuses on the creation of a flexible programmable pipeline, supporting both fully conservative and thin voxelizations (with 26- and 6-separating planes between the triangle and the voxel [Schwarz and Seidel 2010]), multiple boolean, floating point and vector-typed render targets, arbitrary user-defined fragment shaders and a bucketing mode generating the full list of fragments touching each voxel. We refer to the latter as *A-buffer rasterization*, as it solves the task typically needed to generate A-buffers in the context of 2D rasterization.

A simple example where the kind of flexibility provided by our system is needed can be found in the work of Crane et al [2007]: in order to take into account the interactions between a soft-body and a fluid, it's sufficient to plug in the velocity field of the soft-body into the boundary conditions of a Eulerian fluid solver. This can be easily accomplished vox-

elizing the soft-body into the same grid used by the solver, producing for each voxel both a boolean indicating the presence of a surface and a vector indicating its average velocity. Binary voxelization alone would not be sufficient. Similarly, in dynamic stress or heat transport simulations it might be desirable to voxelize the model together with an arbitrary amount of local material properties.

The design tradeoffs of such a system are very similar to those found in making a standard rasterizer with 2D output, and we tried to explore these extensively. Given current hardware capabilities, we found chunking (sort-middle) pipelines to be the most efficient solution for blending-based rasterization, whereas for A-buffer generation we found that the best approach is a feedforward (sort-last) pipeline in which the input triangles are first batched by size and orientation. To sort triangles into tiles and fragments into voxels, we relied on efficient sorting primitives [Merrill and Grimshaw 2010], avoiding all inter-thread communication that would have been necessary using queues. We have also introduced improved algorithms for triangle/voxel overlap testing, and new algorithms for careful load-balancing at all levels of the computing hierarchy. The resulting system, implemented in CUDA, is always competitive in performance and sometimes greatly superior (up to 28 times) to state-of-the-art binary voxelizers [Schwarz and Seidel 2010], despite its major flexibility, whereas our bucketing solution is up to 4 times faster than previous implementations of triangle binning algorithms used to generate uniform grids in the context of real-time ray-tracing [Kalojanov and Slusallek 2009; Kalojanov et al. 2011].

2 Related Work

Voxelization: the state-of-the-art in surface voxelization is the work by Schwarz and Seidel. Unlike previous systems relying on the fixed function rasterization hardware found in commodity GPUs [Zhang et al. 2007; Eisemann and Décoret 2006; Li et al. 2005; Dong et al. 2004; Fang and Chen 2000], their work implements accurate 6-separating and 26-separating binary voxelization using CUDA. Our system differs from theirs in three main respects: first, we extend the pipeline with support for arbitrary framebuffer types, adding full programmability through vertex and fragment shaders; second, unlike their system, which parallelizes work by assigning individual triangles to single threads, and uses global memory atomics to write to a common 3d framebuffer, we explicitly tackle potential load balancing issues using a tile-based algorithm, obtaining huge savings in scenes with uneven triangle sizes. Third, we further optimize their triangle / box overlap test.

Bucketing: most related to our bucketing / A-buffer rendering mode is the work by Kalojanov et al [2009; 2011], that describes how to bucket triangles in uniform grids for ray tracing. In their application, Kalojanov et al don't perform strictly conservative rasterization, but bucket triangles to all voxels overlapped by the triangle's axis-aligned bounding box. To the contrary, our bucketing mode supports accurate 6-separating and 26-separating voxelization. Furthermore, while their system generates a plain list of triangle ids per voxel, our fragment shaders can store any specific fragment data, including interpolated attributes generated using the fragment's barycentric coordinates.

Another interesting observation is that in their initial work Kalojanov and Slusallek generated the entire list of triangle-voxel pairs upfront and sorted it in a second pass, whereas in subsequent work Kalojanov et al [2011] showed how to construct two-level grids with an approach that is similar in spirit to our tile-based voxelization. While this was done primarily to save storage and create a better structure for ray tracing, this approach could also be seen as a load balancing technique. However, even though we found this approach to work well in the case of voxelization, where fragments going to the same voxel are merged together on-the-fly and the entire list of fragments is never stored, we found this to be suboptimal in the case of bucketing, and propose an alternative, faster algorithm.

3 Blending-Based Rasterization

Our standard voxelizer is built around a sort-middle tile-based rasterization pipeline. As such, it is divided into two stages: *coarse raster*, which divides the whole grid into relatively coarse tiles, performs triangle setup and emits the list of all triangles overlapping each tile; and *fine raster*, which assigns each tile to a single core, generates all fragments corresponding to the tile' s triangles, and performs blending in shared memory. With enough on-chip memory, the advantages of such a pipeline are two-fold: first, it performs load balancing by breaking large triangles across multiple tiles; second, it reduces expensive memory bandwidth by doing most of the heavy-lifting in the upper parts of the memory hierarchy. The next subsections discuss these stages in detail, along with our triangle/box overlap test.

3.1 Triangle / Box Overlap Test

Our triangle/box overlap test is essentially the one described by Schwarz and Seidel [Schwarz and Seidel 2010], which itself is an extension and improvement of the 2D variant proposed by Möller and Aila [2005], except we factor computations differently among triangle setup and the actual overlap test. For fully conservative 26-separating rasterization, to determine whether an axis aligned box overlaps a triangle, we first check whether the triangle plane intersects the box, and then check the intersection of the 2d projections of the triangle and the box along all three axis. In the thin 6-separating case we just check the projection along the dominant axis of the triangle normal. Using the XY plane as an example, and denoting the triangle vertices by $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$, the edges by $\mathbf{e}_i = \mathbf{v}_{i\oplus 1} - \mathbf{v}_i$, the normal by $\mathbf{n}$, and the box diagonal by $\Delta\mathbf{p}$, the setup phase starts by computing the 2d edge normals:

$$\mathbf{n}_i^{xy} = (-\mathbf{e}_{i,y}, \mathbf{e}_{i,x}) \cdot sgn(\mathbf{n}_z). \tag{1}$$

Denoting the voxel with integer grid coordinates (u, v, w) as the box with minimum corner $\mathbf{p}+(u, v, w)\cdot\Delta\mathbf{p}$ and maximum corner $\mathbf{p} + (u+1, v+1, w+1) \cdot \Delta\mathbf{p}$, we then precompute the quantities:

$$\begin{aligned} n_i^x &= \mathbf{n}_i^{xy}{}_x \cdot \Delta\mathbf{p}_x \\ n_i^y &= \mathbf{n}_i^{xy}{}_y \cdot \Delta\mathbf{p}_y \end{aligned} \tag{2}$$

and

$$d_i^{xy} = -\mathbf{n}_i^{xy} \cdot \mathbf{v}_{i,xy} + \mathbf{n}_i^{xy} \cdot \mathbf{p}_{i,xy} + \max\{0, \Delta\mathbf{p}_x \mathbf{n}_{i,x}^{xy}\} + \max\{0, \Delta\mathbf{p}_y \mathbf{n}_{i,y}^{xy}\} \tag{3}$$

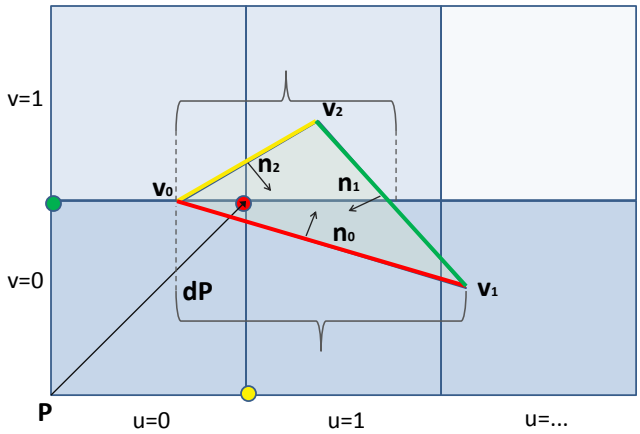

Figure 2: *An example of our 2D overlap test. For each edge, we highlight the point where we evaluate the edge function with the same color. The single-voxel test can be transformed to find the scanline intersections with an entire grid of voxels anchored at* $\mathbf{p}$.

Finally, for any given (u, v, w) voxel, the 2d overlap test is positive if:

$$\bigwedge_{i=0,1,2} \left(d_i^{xy} + v \cdot n_i^y + u \cdot n_i^x \geq 0\right). \tag{4}$$

As in [Schwarz and Seidel 2010], the quantities inside the boolean test represent the value of the edge functions at their critical point, i.e. the point which makes the function most negative. This is shown schematically in Figure 2. In practice, this means that by precomputing 9 floating point values, the amortized cost of a 2d overlap test with each voxel in a scanline becomes as little as 3 FMAs and 3 comparisons (one for each edge $i = 0, 1, 2$, for each change in the scanline variable, e.g. u). Further on, we precompute the plane equation, stored compactly as 3 coefficients by normalizing the dominant axis coordinate to 1.

Scanline intersection: another optimization that we apply in the inner loop of our 2d overlap tests is the computation of conservative bounds for the intersection of a scanline and a triangle. In fact, treating v as a constant, we can write:

$$\bigwedge_{i=0,1,2} \left(c_i + u \cdot n_i^x \geq 0\right). \tag{5}$$

Solving for u for each of $i = \{0, 1, 2\}$ leads to the following bounds on the minumum and maximum values of u for which an intersection may occur:

$$\begin{aligned} c_i + u \cdot n_i^x \geq 0 \iff \quad & n_i^x > 0: \quad u \geq -c_i/n_i^x \\ & n_i^x < 0: \quad u \leq -c_i/n_i^x \end{aligned} \tag{6}$$

Intersecting the semi-intervals obtained for each $i = \{0, 1, 2\}$ we get a range of values spanning the scanline intersection. If the range is degenerate, the scanline doesn't intersect the triangle. By precomputing $1/n_i^x$ and testing only the voxels in the given range of each scanline we can eliminate all per-voxel tests, removing 3 FMAs and 3 comparisons per voxel at the cost of three reciprocals per triangle and 3 FMULs per scanline.

3.2 Coarse Raster

The purpose of the coarse rasterizer is to perform triangle setup and emit a conservative list of all triangles touching each tile. Our implementation differs from typical chunkers in three main respects:

Triangle setup: our setup phase is fairly trivial, emitting only the integer bounding box of the triangle and its dominant axis. As discussed in the previous section, the setup for our cheapest 6-separating triangle-voxel coverage test requires computing 12 floats: given the high compute-to-bandwidth ratio of modern GPUs, it turns out it's cheaper to recompute these quantities for each tile overlapped by each triangle than to compute them once and read them several times.

Tile sorting: rather than using queues to output per-tile lists, we first output an unsorted list of $(triangle, descriptor)$ pairs, where the two lowest bits of the descriptor specify the dominant axis and the higher bits specify the tile id, and then we sort them by descriptor in a separate pass. Triangle setup and triangle-descriptor pair emission are handled by a single CUDA kernel which processes a triangle per thread. Expensive tile coverage tests are avoided at this stage, and we simply emit all tiles touched by the triangle bounding box. After the sorting phase, we generate a compact list of per-tile begin and end ranges to feed the *fine raster* stage. Compared to approaches that use per-tile queues, such as [Schwarz and Seidel 2010], we avoid the use of any inter-thread communication primitives, relying on the efficiency of the global radix-sort counting and scattering phases to dispatch each item to the corresponding list. This is generally more efficient because of the relatively low data amplification present in this stage: due to the large tile size, triangles will often touch one or a few tiles only. Moreover, as we will show in the next section, having the triangles sorted by dominant axis inside each tile segment increases SIMD efficiency in the following stage of the pipeline.

Tile splitting: In order to allow for better load balancing during the *fine raster* stage, we split physical tiles containing many triangles into *virtual tiles* with smaller triangle sets, containing at most M_{tile} triangles. The name *virtual* refers to the fact that the tiles produced by the splitting procedure overlap on the voxel grid. Hence, during *fine raster* we create a separate memory arena for each virtual tile, and perform a final merging pass to blend all virtual tiles mapping to the same physical entry in the output framebuffer. This is especially useful on low-res voxelizations, where there might be too few occupied tiles to fill the machine, or where triangle distributions might be very skewed.

3.3 Fine Raster

Fine raster gets a list of per-tile $(begin, end)$ triangle ranges as input, and needs to generate all triangle fragments at the finest scale while doing the final per-voxel blending. In order to achieve maximum performance, we allocate a pool of persistent thread blocks just as big as required to fill the machine, where each thread block keeps fetching a single tile at a time from a global queue (with one atomic per block), loading the tile in shared memory before we start the actual rasterization. During rasterization, we perform all blending in shared memory, and then copy the tile back

to global memory. Inside each thread block, we implement a further load balancing mechanism through persistent warps, where each warp keeps fetching batches of 32 triangles to process from the tile's list, until there is no batches left. The concurrent fetching can be handled with a single shared memory atomic per warp. After the batch assignment, each thread processes a single triangle, and performs the following steps:

Triangle plane / tile overlap test: in order to avoid redundant work, we first check whether the triangle plane intersects the tile. If not, the thread is done processing the triangle.

Triangle setup: after the plane test, when a thread is given a triangle to process, we perform setup as described in section 3.1, computing 9 coefficients for each involved 2d projection (1 for the 6-separating case, 3 for fully conservative rasterization).

Fragment generation & blending: after setup is done, each thread starts looping across all scanlines in the dominant projection of its triangle, computes bounds on the scanline intersection and loops across all pixels in the resulting range. For 6-separating rasterization, for each pixel we perform the 2D overlap test for this projection only and eventually emit a fragment. Assuming the dominant axis is Z, we compute the fragment's Z by evaluating the plane equation at the center of the voxel, shade it and perform blending immediately with shared memory atomics. For the 26-separating case, if the first overlap test is passed, we compute the minimum and maximum Z spanned by the triangle plane inside the voxel by evaluating the plane equation at two corners of the voxel projection, and then loop across all voxels in the column defined by this Z range performing the additional 2D overlap tests. If both tests pass, a fragment is again emitted, shaded and blended immediately.

Relying on hardware atomics for framebuffer updates sets constraints on the blending capabilities, and as a consequence the only forms of blending currently supported are taking the addition, maximum and minimum of the source and destination values. For integer render targets, binary AND and OR operators are also available. Alternatively, blending can be disabled, allowing faster but non-deterministic replacement. The implementation of more sophisticated forms of blending is reserved as future work.

Multiple render targets: multiple targets are handled by simply executing the *fine raster* phase once for each additional target, thus sharing the work done in *coarse raster*. While in theory it would be possible to optimize this further by sharing the work needed to perform voxel testing, the amount of targets would be severely limited by the amount of available shared memory. Moreover, this strategy allows to render different targets at different resolutions.

Fragment shaders: the *fine raster* entry function is templated over an arbitrary user-defined functor required to implement a simple fragment shader interface. The interface must provide a single method to process individual fragments and emit a value of the same type as that of the framebuffer. The input fragment is described by the triangle id, its vertices, the geometric normal, and the barycentric coordinates. A user-defined functor might hence implement any

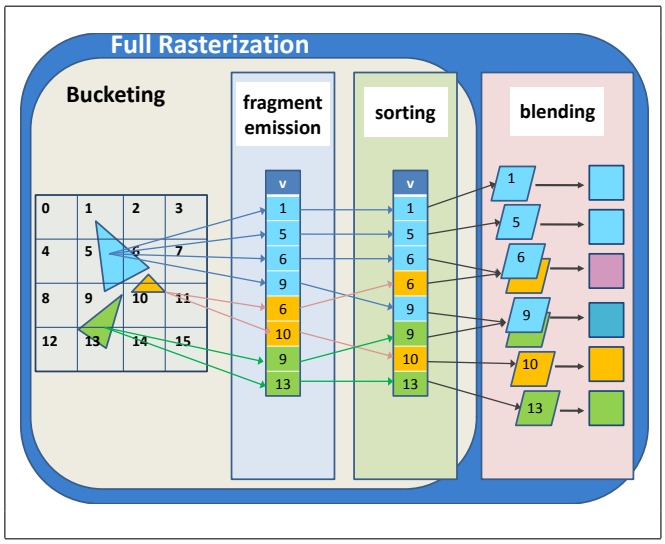

Figure 3: *A conceptual view of the voxelization pipeline. The blending stage (in pink), performing a data reduction on the fragments belonging to the same voxel, is only part of regular rasterization, and is not present in the bucketing pipeline.*

form of attribute interpolation and texture lookups, as well as dispatch the call to per-triangle shader instances through virtual methods or function calls.

Scanline redistribution: in order to reduce the number of conflicts across the per-fragment-per-voxel shared memory atomics, and to further improve load balancing and SIMT utilization, we further tried to redistribute the scanlines in the dominant axis projection of each triangle equally across each warp. In order to do this, each warp kept a circular queue of up to 64 triangles so as to make sure there were always at least 32 scanlines to process. Unfortunately, the overhead incurred by the redistribution mechanism was very significant compared to the efficiency of our voxel overlap tests, and resulted in a general slowdown.

4 A-buffer Rasterization

The generation of an A-buffer poses challenges substantially different from the ones found in blending-based voxelization. The first key difference is that the output size of the framebuffer is not limited, and in particular it's not possible to guarantee that a tile would entirely fit in shared memory.

The first strategy we employed to perform this kind of rasterization was hence based on a tiled renderer with a 2-pass fine rasterization stage: a counting phase which kept a shared memory scratchpad to count how many fragments would go into each voxel (working in exactly the same way as a blending-based fine raster pass with an integer framebuffer) was followed by a fragment emission phase to emit all fragments at a unique offset. Unfortunately, this approach suffered from two main problems: first, the amount of space required to keep a tile of 32-bit integers in shared memory limited the tile size to at most 16^3 voxels on current hardware. Second, the approach required double computation of the voxel overlap tests.

Analyzing the problem from an entirely different perspective

Figure 4: *Our test scenes. From left to right: Conference, Stanford Dragon, Turbine Blade, Hair Ball and the XYZRGB Asian Dragon.*

we hence came up with a second approach. This form of rasterization can be seen as a special variant of sorting, where the input data (i.e. the list of triangles) can be thought of as a compressed representation of the final list of elements to be sorted (the fragments), and where the decompression involves a heavy data amplification phase with potentially high variation in the per element input:output ratios. Under this perspective, bucketing is equivalent to the sum of two independent passes: fragment emission and fragment sorting. This was not the case in standard voxelization, which includes a further *reduction* stage - blending - that removes the need to ever generate, sort and store the entire list of fragments at once, and where interleaving these three stages leads to a more work- and bandwidth-efficient pipeline (because data corresponding to individual fragments can be consumed right away without going through external memory).

As very work-efficient parallel sorting algorithms have already been devised, the main problem becomes that of load-balancing the computations necessary for the potentially severe data amplification in the fragment emission phase. In order to tackle the problem we came up with a 4-stage strategy:

triangle classification: in the first stage the triangles are classified by size and dominant axis;

triangle sorting: in the second stage the triangles are sorted by class;

unordered fragment emission: in the third stage all fragments of all triangles are emitted in random order together with their voxel id;

fragment sorting: the final stage performs a single global radix-sort on the fragments to sort them by voxel id.

The crucial part of this pipeline is the triangle classification and sorting phase, which gives us the ability to collect similarly sized and similarly oriented triangles together, and process them with ad-hoc load-balancing strategies in the next stage of the pipeline. The next sections describe the specific algorithms employed in these stages.

4.1 Triangle Classification

Triangle classification starts with a simple triangle setup stage that emits the triangle bounding box and a single triangle descriptor. In order to form the descriptor, we first compute the 2D bounding box of the triangle's projection along its dominant axis. If either the horizontal or the vertical side length of the bounding box is equal to 16 or more, we classify the triangle as *large*, and set its descriptor as:

$$axis + 3 \cdot (size_u(bbox) \geq size_v(bbox)\ ?\ 0 : 1). \quad (7)$$

Otherwise, we set the descriptor as:

$$6 + \log(size_u(bbox)) \cdot 4 + \log(size_v(bbox)) + axis \cdot 16 \quad (8)$$

The constants involved in the expression have been determined to pack the resulting values tightly in the ranges $[0, 6)$ and $[6, 54)$ respectively, so as to fit the entire descriptors in as little as 6 bits (and ultimately requiring as few radix sorting passes as possible).

4.2 Fragment Emission

We perform fragment emission with two separate kernels. The first kernel processes all *small* triangles, while the second processes the ones classified as *large*.

The *small* triangle processor implements warp-level load-balancing in the same fashion used for the blending-based *fine raster*. Each thread processes a single triangle and emits fragments to a global queue. As the loop over the 2d voxel overlap tests is executed synchronously within each warp, the output location of each valid fragment in the queue is determined with a single atomic per-warp, incrementing the global queue size by the popcount of the number of positive tests. While one global memory atomic per-warp-per-fragment might seem a substantial amount of inter-thread communication, in practice we measured a minimal performance impact.

In order to maximize SIMT utilization, the *large* triangle processor adopts a different policy and assigns a single triangle per warp, where the entire warp loops over all the scanlines of the 2d projection of the triangle's bounding box, and each thread processes a single pixel of the current scanline. Again, the resulting fragments for each scanline are output using a single atomic per-warp.

5 Compile-Time Code Specialization

Our library supports hundreds of possible configurations, as the user can select among two types of voxelization, 6 blending modes, 25 framebuffer types, all possible framebuffer sizes, and provide arbitrary vertex and fragment shaders. Critical to obtaining maximum performance is to tune the internal algorithms according to the selected configuration. For example, the choice of the best tile size depends strongly on the type of framebuffer. Similarly, the inner loop of the

Scene	# of Triangles	Grid Res	Schwartz & Seidel	VP Binary	VP Float	VP A-buffer
Conference	282k	128^3	3.9 ms	3.3 ms	3.4 ms	3.8 ms
		512^3	59.3 ms	4.3 ms	8.3 ms	5.3 ms
		1024^3	237.6 ms	8.5 ms	24.0 ms	10.1 ms
Dragon	871k	128^3	3.5 ms	4.8 ms	5.0 ms	6.7 ms
		512^3	4.8 ms	5.0 ms	7.5 ms	8.7 ms
		1024^3	13.6 ms	5.9 ms	13.2 ms	11.6 ms
Turbine Blade	1.76M	128^3	3.6 ms	7.3 ms	7.9 ms	10.3 ms
		512^3	7.6 ms	6.9 ms	10.1 ms	11.6 ms
		1024^3	16.6 ms	8.4 ms	14.9 ms	12.7 ms
Hairball	2.88M	128^3	22.8 ms	12.8 ms	15.3 ms	23.8 ms
		512^3	95.0 ms	18.3 ms	38.9 ms	50.0 ms
		1024^3	266.8 ms	33.7 ms	192.8 ms	102.0 ms
XYZ RGB Asian Dragon	7.21M	128^3	11.4 ms	21.2 ms	26.0 ms	34.8 ms
		512^3	16.7 ms	22.0 ms	29.4 ms	39.9 ms
		1024^3	18.2 ms	23.6 ms	31.4 ms	43.0 ms

Table 1: *Voxelization timings for various scenes and different voxelization schemes. VP stands for VoxelPipe.*

critical code path depends strongly on the type of voxelization and the blending mode. In order to achieve this, we made extensive use of compile-time code specialization, relying heavily on the expressiveness of C++ templates: all the user visible options of our library are in fact template parameters. Through meta-programming, the particular configuration of options determines in turn the internal state of the library and the particular code path that will be executed by the rendering kernels. Without the extensive support of the C++ feature set and the flexibility offered by the CUDA runtime API developing such a library would have been highly impractical.

Scene	Stage	128^3	512^3	1024^3
Conference	`coarse raster`	0.5 ms	0.5 ms	0.6 ms
	`sorting`	0.7 ms	1.2 ms	1.4 ms
	`fine raster`	2.2 ms	2.6 ms	6.4 ms
Hairball	`coarse raster`	2.1 ms	2.3 ms	2.8 ms
	`sorting`	2.4 ms	4.8 ms	6.9 ms
	`fine raster`	7.9 ms	10.7 ms	24.0 ms
Asia Dragon	`coarse raster`	4.8 ms	4.8 ms	4.9 ms
	`sorting`	4.2 ms	7.5 ms	8.6 ms
	`fine raster`	14.0 ms	10.1 ms	10.1 ms

Table 2: *Timing breakdown for binary voxelization of three of our test scenes at different resolutions.*

6 Results

The source code for our pipeline is freely available at *http://code.google.com/p/voxelpipe/*.

We have run our algorithms on a variety of scenes with various complexity: Conference, the Stanford Dragon, the Turbine Blade, the Hair Ball and the XYZ RGB Asian Dragon (Figure 4). Our benchmark system uses a GeForce GTX480 GPU with 1GB of GPU memory, and an Intel Core i7 860 @ 2.8GHz CPU with 4GB of main memory.

In Table 1 we report scan conversion times for the blending-based voxelizer and for the A-buffer pipeline. Table 2 provides a more detailed breakdown of the timings of the individual components of our binary voxelization pipeline for a few of those scenes.

It can be noticed that scenes with very uneven geometric density (e.g. Conference and Hairball) perform substantially better than previous state-of-the-art, up to 28x faster. An exception to the rule is the Hairball scene voxelized at high resolutions in floating point buffers, performing less than 2x better than the binary voxelization from [Schwarz and Seidel 2010], and taking substantially longer than all other scenes and configurations with our own system. This is due to the combination of two problems: first, with floating point buffers, our tile size is limited to 16^3 due to shared memory constraints. This leads to a much higher amount of work in the coarse raster stage, which needs to sort more triangle/tile pairs. Second, this scene contains many long and skinny triangles generating highly variable fragment counts, leading to low SIMT utilization in the fine raster stage. While the first problem cannot be overcome without more capable hardware, we think we have potential solutions for the second, which we leave as future work.

Scene	# of Triangles	KS	VP
Conference	282k	13.5 ms	3.8 ms (3.55x)
Dragon	180k	4.0 ms	3.7 ms (1.08x)
Fairy	174k	12.0 ms	4.2 ms (2.85x)
Soda Hall	2.2M	65.0 ms	15.0 ms (4.33x)

Table 3: *Comparison between bucketing with the system from Kalojanov and Slusallek (KS) and VoxelPipe (VP).*

For bucketing, we cannot compare our results to previous state-of-the-art thorouhgly, as Kalojanov and Slusallek [2009] reported results for a limited set of grid resolutions and used an NVIDIA GTX280. As our system exploits capabilities available only on GTX 480 and higher, we divided their reported timings by 2x, which is roughly the improvement in the overall compute performance between the two GPUs. This is an optimistic estimate, as memory bandwidth went up by a smaller margin, and the bulk of the algorithm used by Kalojanov and Slusallek was bandwidth bound. The results are reported in Table 3. We decided not to compare with the more recent system from Kalojanov et al [2011], as the output of this system is a different, hierarchical data structure. However, their algorithm is similar in spirit to our initial prototype based on a tiled approach, that resulted being slower than our current solution.

Figure 5: *A preview of a real-time global illumination system built on top of our pipeline. Here voxelization is used to create a scene proxy that we use for tracing incoherent rays. As voxelization can complete in a few milliseconds even on several million triangles, the proxy can be rebuilt every frame, allowing dynamic geometry, materials and lighting. This scene runs at over 50 FPS at a resolution of* 1024×1024.

7 Summary and Discussion

We have presented a fully programmable pipeline to perform triangle voxelization. The pipeline supports both standard blending-based scan conversion and a bucketing / A-buffer mode where each fragment is recorded separately. The blending-based voxelizer further supports a variety of render targets, single-pass multiple render targets, and user-defined vertex and fragment shaders. Despite the additional flexibility, its performance is competitive with that of state-of-the-art binary voxelizers, when not superior. The A-buffer voxelizer, which can be used to perform standard triangle-in-grid binning, is 4 times faster than state-of-the-art binning algorithms used for uniform grid generation in the field of real-time ray-tracing.

Future Work: as work left for the future, we are considering to integrate memory friendly strategies to build octrees in a spirit similar to [Schwarz and Seidel 2010], as well as supporting geometry and tessellation shaders and more general or programmable blending policies. Additional research would also be needed to understand how to build a power-efficient voxelization pipeline, aided by fixed function hardware. As for standard rasterization it's in fact clear that an entirely software based solution could not achieve the same performance/watt ratios obtained by a carefully planned hardware design. Finally, in Figure 5 we show a proof of concept of a real-time global illumination system that we plan to disclose in the near future. The system relies on our voxelization pipeline to create a proxy of the scene geometry that is used to trace incoherent rays. The proxy is rebuilt every frame, allowing support of fully dynamic geometry, materials and lighting.

Aknowledgements: we thank Michael Schwarz for his kind support in providing all measurements of the system described in [Schwarz and Seidel 2010] necessary for our comparisons. Special thanks go also to Eric Enderton and Timo Aila for their precious reviews.

References

Akenine-Möller, T., and Aila, T. 2005. Conservative and tiled rasterization using a modified triangle set-up. *Journal of Graphics Tools 10*, 3, 1–8.

anonymous. 2011. AnonGI: novel algorithms for real-time global illumination. Tech. rep., Anonymous.

Crane, K., Llamas, I., and Tariq, S. 2007. GPU Gems 3. First ed. Addison-Wesley Professional, ch. 30.

Dong, Z., Chen, W., Bao, H., Zhang, H., and Peng, Q. 2004. Real-time voxelization for complex polygonal models. In *Pacific Conference on Computer Graphics and Applications*, 43–50.

Eisemann, E., and Décoret, X. 2006. Fast scene voxelization and applications. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics, SI3D 2006*, ACM SIGGRAPH.

Fang, S., and Chen, H. 2000. Hardware accelerated voxelization. *Computers & Graphics 24*, 3, 433–442.

Kalojanov, J., and Slusallek, P. 2009. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 23–28.

Kalojanov, J., Billeter, M., and Slusallek, P. 2011. Two-level grids for ray tracing on GPUs. *Computer Graphics Forum* (4).

Kaplanyan, A., and Dachsbacher, C. 2010. Cascaded light propagation volumes for real-time indirect illumination. In *I3D 2010*.

Li, W., Fan, Z., Wei, X., and Kaufman, A. 2005. Flow simulation with complex boundaries. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley Professional, ch. 47, 747–764.

Merrill, D., and Grimshaw, A. 2010. Revisiting sorting for GPGPU stream architectures. Tech. Rep. CS2010-03, Department of Computer Science, University of Virginia, February.

Nichols, G., Penmatsa, R., and Wyman, C. 2010. Interactive, multiresolution image-space rendering for dynamic area lighting. *Comput. Graph. Forum 29*, 4, 1279–1288.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable parallel programming with cuda. *ACM Queue 6*, 2, 40–53.

Schwarz, M., and Seidel, H.-P. 2010. Fast parallel surface and solid voxelization on gpus. In *ACM SIGGRAPH Asia 2010 papers*, ACM, New York, NY, USA, SIGGRAPH ASIA '10, ACM Computer Society, 179:1–179:10.

Zhang, L., Chen, W., Ebert, D. S., and Peng, Q. 2007. Conservative voxelization. *The Visual Computer 23*, 9-11, 783–792.