

OpenGL Insights

Edited by
Patrick Cozzi and Christophe Riccio



CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer

22

Cyril Crassin and Simon Green

22.1 Introduction

Discrete voxel representations are generating growing interest in a wide range of applications in computational sciences and particularly in computer graphics. Applications range from fluid simulation [Crane et al. 05], collision detection [Allard et al. 10], and radiative transfer simulation to detail rendering [Crassin et al. 09, Crassin et al. 10, Laine and Karras 10] and real-time global illumination [Kaplanyan and Dachsbacher 10, Thiedemann et al. 11, Crassin et al. 11]. When used in real-time contexts, it becomes critical to achieve fast *3D scan conversion* (also called *voxelization*) of traditional triangle-based surface representations [Eisemann and Décorêt 08, Schwarz and Seidel 10, Pantaleoni 11].

In this chapter, we will first describe an efficient OpenGL implementation of a simple surface voxelization algorithm that produces a regular 3D texture (see Figure 22.1). This technique uses the GPU hardware rasterizer and the new image load/store interface exposed by OpenGL 4.2. This section will allow us to familiarize the reader with the general algorithm and the new OpenGL features we leverage.

In the second part, we will describe an extension of this approach, which enables building and updating a sparse voxel representation in the form of an octree structure. In order to scale to very large scenes, our approach avoids relying on an intermediate-regular grid to build the structure and constructs the octree directly. This second approach exploits the draw indirect features standardized in OpenGL 4.0 in order to allow synchronization-free launching of shader threads during the octree construction, as well as the new *atomic counter* functions exposed in OpenGL 4.2.



Figure 22.1. Real-time voxelization of dynamic objects into a sparse voxel octree (Wald’s hand 16K triangles mesh voxelized sparsely in approximately 5.5 ms) and use of the technique for a voxel-based global illumination application.

One of our main motivations in this work has been to investigate the usability of the hardware graphics pipeline for fast and real-time voxelization. We will compare the performance of our approach to the recent work of Pantaleoni [Pantaleoni 11], which uses CUDA for regular-grid thin voxelization, and detail the performance of our sparse-octree building approach. A typical real-time usage of our dynamic voxelization inside a sparse voxel octree has been demonstrated recently as part of the voxel-based global illumination approach described in [Crassin et al. 11].

22.2 Previous Work

Previous work on 3D voxelization makes a distinction between two kinds of surface voxelization: *thin voxelization*, which is a *6-separating* representation of a surface (cf. [Huang et al. 98]) and fully *conservative voxelization*, where all voxels overlapped by a surface are activated, or *26-separating* (Figure 22.2). Although our method could easily be extended to fully conservative voxelization, in this chapter we will only describe the case of thin voxelization. Thin voxelization is cheaper to compute and is often more desirable in computer graphics applications.

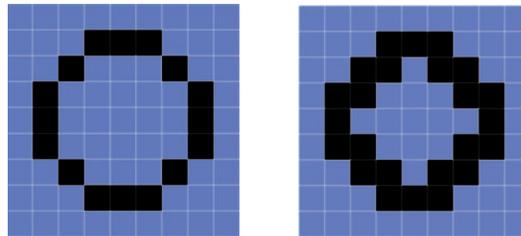


Figure 22.2. Examples of a 4-separating (left) and an 8-separating (right) 2D line rasterization equivalent to 6-separating and 26-separating surface voxelizations in 3D.

In recent years, many algorithms have been proposed that exploit GPUs by performing triangle mesh voxelization. Early approaches used the fixed-function pipeline found in commodity graphics hardware of the time. Previous hardware-based approaches [Fang et al. 00, Crane et al. 05, Li et al. 05] were relatively inefficient and suffered from quality problems. Due to the lack of random write access, these approaches had to use a multipass rendering technique, processing the volume slice by slice and retransforming the entire geometry with each pass. In contrast, [Dong et al. 04, Zhang et al. 07, Eisemann and Décoret 08] process multiple slices at a time by encoding voxel grids with a compact binary representation, achieving higher performance but limited to binary voxelization (only storing a single bit to represent an occupied voxel).

Newer voxelization approaches take advantage of the freedom offered by the *compute mode* (CUDA or OpenCL) available on modern GPUs [Schwarz and Seidel 10, Pantaleoni 11]. Instead of building on the fixed-function hardware, these approaches propose pure data-parallel algorithms, providing more flexibility and allowing new original voxelization schemes like direct voxelization into a *sparse octree*. However, using only the compute mode of the GPU means that these approaches don't take advantage of the powerful fixed-function graphics units, particularly the hardware rasterizer, that effectively provide a very fast point-in-triangle test function and sampling operation. With increasing industry focus on power efficiency for mobile devices, utilizing efficient fixed-function hardware is increasingly important. Our method combines the advantages of both approaches, taking advantage of the fast fixed-function graphics units while requiring only a single geometry pass and allowing sparse voxelization thanks to the most recent evolutions of the GPU hardware.

22.3 Unrestricted Memory Access in GLSL

Previous graphics-based approaches (not using compute) were limited by the fact that all memory write operations had to be done through the ROP (fragment operation) hardware, which does not allow random access and 3D-addressing because only the current pixel could be written. Recently, the programming model offered by OpenGL shaders has changed dramatically, with GLSL shaders acquiring the ability to generate side effects and to dynamically address arbitrary buffers and textures, for example, the OpenGL 4.2 specification standardized *image units* access in GLSL (previously exposed through the `EXT_shader_image_load_store` extension). This feature, only available on Shader Model 5 (SM5) hardware, gives us the ability to perform read/write access as well as atomic read-modify-write operations into a single mipmap level of a texture from any GLSL shader stage. Beyond textures, linear memory regions (*buffer objects* stored in GPU global memory) can also be easily accessed with this feature using “buffer textures” bound to a GLSL `imageBuffer`.

In addition, the NVIDIA-specific extensions `NV_shader_buffer_load` and `NV_shader_buffer_store` (supported on Fermi-class SM5 hardware), provide similar functionality on linear memory regions, but they do this through C-like pointers in GLSL, and the ability to query the global memory address of any buffer object. This approach simplifies the access to buffer objects and allows arbitrary numbers of discontinuous memory regions (different buffer objects) to be accessed from the same shader invocation, while only a limited number of image units can be accessed by a given shader (this number is implementation dependent and can be queried using `GL_MAX_IMAGE_UNITS`).

These new features dramatically change the computation model of GPU shaders and give us the ability to write algorithms with much of the same flexibility as CUDA or OpenCL, while still taking advantage of the fast fixed-function hardware.

22.4 Simple Voxelization Pipeline

In this section we will present an initial simple approach to directly voxelize into a regular grid of voxels stored in a 3D texture. Our voxelization pipeline is based on the observation that, as shown in [Schwarz and Seidel 10], a *thin surface voxelization* of a triangle \mathcal{B} can be computed for each voxel \mathcal{V} by testing if (1) \mathcal{B} 's plane intersects \mathcal{V} , (2) the 2D projection of the triangle \mathcal{B} along the dominant axis of its normal (one of the three main axes of the scene that provides the largest surface for the projected triangle) intersects the 2D projection of \mathcal{V} .

Based on this observation, we propose a very simple voxelization algorithm that operates in four main steps inside a single draw call (illustrated in Figure 22.3). First, each triangle of the mesh is projected orthographically along the dominant axis of its normal, which is the one of the three main axes of the scene that maximizes the projected area and thus maximizes the number of fragments that will be generated during the conservative rasterization. This projection axis is chosen dynamically on a per-triangle basis inside a geometry shader (see Figure 22.4), where information about the three vertices of the triangle is available. For each triangle, the selected axis

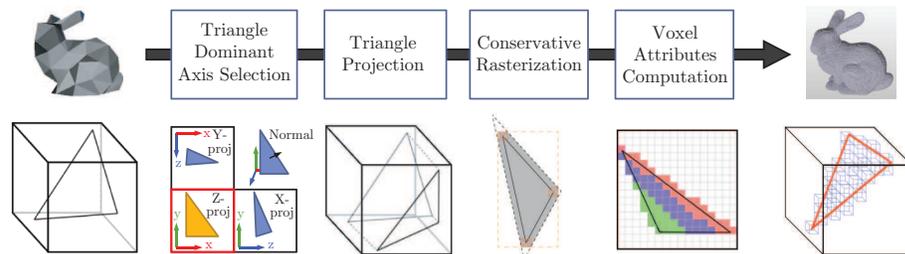


Figure 22.3. Illustration of our simple voxelization pipeline.

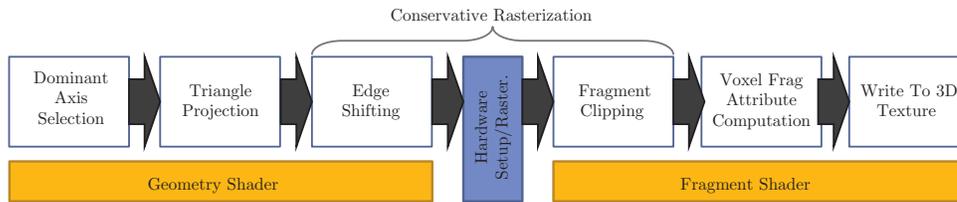


Figure 22.4. Implementation of our voxelization pipeline on top of the GPU rasterization pipeline.

is the one that provides the maximum value for $l_{\{x,y,z\}} = |\mathbf{n} \cdot \mathbf{v}_{\{x,y,z\}}|$, with \mathbf{n} the triangle normal and $\mathbf{v}_{\{x,y,z\}}$ the three main axes of the scene. Once the axis selected, the projection along this axis is simply a classical orthographic projection, and this is calculated inside the geometry shader.

Each projected triangle is fed into the standard setup and rasterization pipeline to perform 2D scan conversion (rasterization, see Figure 22.4). In order to get fragments corresponding to the 3D resolution of the destination (cubical) voxel grid, we set the 2D viewport resolution (`glViewport(0, 0, x, y)`) to correspond to lateral resolution of our voxel grid (for instance 512×512 pixels for a 512^3 voxel grid). Since we rely on image access instead of the standard ROP path to the framebuffer to write data into our voxel grid, all framebuffer operations are disabled, including depth writes, depth testing (`glDisable(GL_DEPTH_TEST)`) and color writes (`glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)`).

During rasterization, each triangle generates a set of 2D fragments. Each of these fragments can correspond to the intersection of the triangle with one, two or three voxels along its direction of projection. Indeed, due to our choice of the dominant triangle axis for projection (and the use of cubic voxels), the depth range of a triangle across a 2D pixel can only span a maximum of three voxels in depth. For each 2D fragment, the voxels actually intersected by the triangle are computed within the fragment shader, based on position and depth information interpolated from vertices' values at the pixel center, as well as screen-space derivatives provided by GLSL (`dFdx()`/`dFdy()`).

This information is used to generate what we call *voxel fragments*. A voxel fragment is the 3D generalization of the classic 2D fragment and corresponds to a voxel intersected by a given triangle. Each voxel fragment has a 3D integer coordinate inside the destination voxel grid, as well as multiple attribute values.

Voxel-fragment attributes are usually a color, a normal, and any other useful attribute one would want to store per voxel, depending on the application. As usual, these values can be either interpolated on pixel centers from vertex attributes by the rasterization process or sampled from the traditional 2D surface textures of the model using interpolated texture coordinates. In our demo implementation, we only store one color value as well as one normal vector (used for shading during the rendering of the voxel grid) per voxel.

Finally, voxel fragments are written directly from the fragment shader into their corresponding voxel inside the destination 3D texture, where they must be combined. This is done using image load/store operations as detailed in Section 22.4.2.

22.4.1 Conservative Rasterization

Although it is very simple, this approach does not ensure a correct thin (6-separating planes [Schwarz and Seidel 10]) voxelization. This is due to the fact that only the coverage of the center of each pixel is tested against the triangles to generate fragments during the rasterization step. Thus, a more precise *conservative rasterization* must be employed to ensure that a fragment will be generated for each pixel touched by a triangle. The precision of the coverage test could be enhanced by relying on multisample antialiasing (MSAA), but this solution only delays the problem a little further and still misses fragments in the case of small triangles. Instead, and similarly to [Zhang et al. 07], we build upon the second conservative rasterization approach proposed in [Hasselgren et al. 05]. We will not detail the technique here, and we invite the reader to refer to [Hasselgren et al. 05] for more details.

The general idea is to generate, for each projected triangle, a slightly larger bounding polygon that ensures that any projected triangle touching a pixel will necessarily touch the center of this pixel and thus will get a fragment emitted by the fixed-function rasterizer. This is done by shifting each triangle edge outward in order to enlarge the triangle using the geometry shader (Figure 22.4). Since the exact bounding polygon that does not overestimate the coverage of a given triangle is not triangle-shaped (Figure 22.5), the excess fragments outside the bounding box are killed in the fragment shader after rasterization. This approach entails more work in the fragment shader but, in practice, is faster than computing and generating exactly the correct bounding polygon inside the geometry shader.

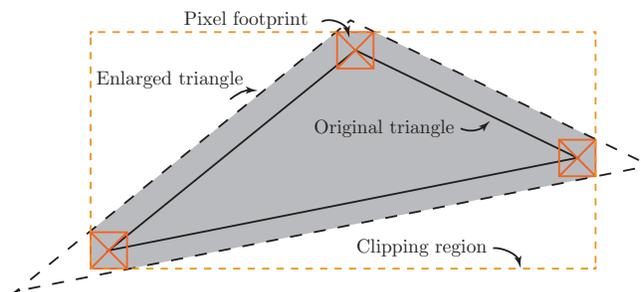


Figure 22.5. Bounding polygon of a triangle used for conservative rasterization.

22.4.2 Compositing Voxel Fragments

Once voxel-fragments have been generated in the fragment shader, their values can be written directly into the destination 3D texture using image load/store operations. However, multiple voxel fragments from different triangles can fall into the same destination voxel in arbitrary order. Since voxel fragments are created and processed in parallel, the order in which they will be written is not predictable, which leads to write-ordering issues and can create flickering and non-time-coherent results when dynamically revoxelizing a scene. In standard rasterization, this problem is handled by the ROP units, which ensure that fragments are composed in the framebuffer in the same order as their source primitives have been issued.

In our case, we have to rely on atomic operations. Atomic operations guarantee that the read-modify-write cycle is not interrupted by any other thread. When multiple voxel fragments end up on the same voxel, the most simple desirable behavior is averaging all incoming values. For specific applications, one may want to use more sophisticated combination schemes like coverage-based combination, but this goes beyond the scope of this chapter.

Averaging values using atomic operations. To average all values falling into the same voxel, the simplest way is to first sum all values using an atomic add operation and then divide this sum by the total number of values in a subsequent pass. To do so, a counter must be maintained per voxel, and we rely on the *alpha channel* of the RGBA color values we store per voxel for this purpose.

However, image-atomic operations are restricted to 32-bit signed/unsigned integer types in OpenGL 4.2 specification, which will rarely correspond to the texel format used in the voxel grid. We generally want to store RGBA8 or RGBA16F/32F color components per voxel. Thus, the `imageAtomicAdd` function cannot be used directly as is to do the summation.

We emulate an *atomic add* on such types by relying on a compare-and-swap `atomicCompSwap()` operation using the function detailed in Listing 22.1. The idea is to loop on each write until there are no more conflicts and the value, with which we have computed the sum has not been changed by another thread. This approach is a lot slower than a native `atomicAdd` would be but still allows a functionally correct behavior while waiting for the specification to evolve. On NVIDIA hardware, an `atomicCompSwap64` operating on 64-bit values can be used on global memory addresses (`NV_shader_buffer_store`), which allows us to cut by half the number of operations and thus provides a two times speedup over the cross-vendor path. Unfortunately, this process is not exposed for image access, which requires the voxel grid to be stored inside the global memory instead of the texture memory.

A second problem appears when using an RGBA8 color format per voxel. With such a format, only 8 bits are available per color component, which quickly causes overflow problems when summing the values. Thus, the average must be computed incrementally each time a new voxel fragment is merged into a given voxel. To do

```

void imageAtomicFloatAdd(layout(r32ui) coherent volatile uimage3D imgUI, ivec3 ↵
    coords, float val)
{
    uint newVal = floatBitsToUint(val);
    uint prevVal = 0; uint curVal;

    //Loop as long as destination value gets changed by other threads
    while( (curVal = imageAtomicCompSwap(imgUI, coords, prevVal, newVal)) != prevVal)
    {
        prevVal = curVal;
        newVal = floatBitsToUint((val + uintBitsToFloat(curVal)));
    }
}

```

Listing 22.1. AtomicAdd emulation on 32-bit floating point data type using a compare-and-swap operation.

this, we simply compute a *moving average* using the following formula:

$$C_{i+1} = \frac{iC_i + x_{i+1}}{i + 1}.$$

This can be done easily by slightly modifying the previous swap-based atomic add operation as shown in Listing 22.2. Note that this approach will only work if all data to be stored, including the counter, can be swapped together using one single atomic operation.

```

vec4 convRGBA8ToVec4(uint val){
    return vec4( float((val&0x000000FF)), float((val&0x0000FF00)>>8U), float((val&0x0000FF0000)>>16U), float((val&0xFF000000)>>24U) );
}
uint convVec4ToRGBA8(vec4 val){
    return (uint(val.w)&0x000000FF)<<24U | (uint(val.z)&0x000000FF)<<16U | (uint(val.y)&0x000000FF)<<8U | (uint(val.x)&0x000000FF);
}

void imageAtomicRGBA8Avg(layout(r32ui) coherent volatile uimage3D imgUI, ivec3 ↵
    coords, vec4 val) {
    val.rgb*=255.0f; //Optimise following calculations
    uint newVal = convVec4ToRGBA8(val);
    uint prevStoredVal = 0; uint curStoredVal;
    //Loop as long as destination value gets changed by other threads
    while( (curStoredVal = imageAtomicCompSwap(imgUI, coords, prevStoredVal, newVal)) ↵
        != prevStoredVal) {
        prevStoredVal = curStoredVal;
        vec4 rval=convRGBA8ToVec4(curStoredVal);
        rval.xyz=(rval.xyz*rval.w); //Denormalize
        vec4 curValF=rval+val; //Add new value
        curValF.xyz/=curValF.w; //Renormalize
        newVal = convVec4ToRGBA8(curValF);
    }
}

```

Listing 22.2. AtomicAvg on RGBA8 pixel type implemented with a moving average and using a compare-and-swap atomic operation.

22.4.3 Results

Table 22.1 shows execution times (in milliseconds) of our voxelization algorithm on the Stanford dragon mesh (871K triangles, Figure 22.6), for 128^3 and 512^3 voxel resolutions, with and without conservative rasterization, and with direct write or merging of values (Section 22.4.2). All timings have been done on an NVIDIA GTX480.

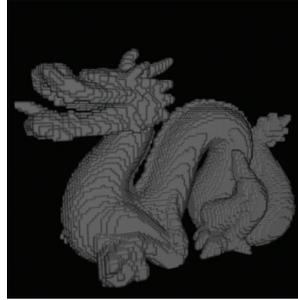


Figure 22.6. Stanford dragon voxelized into a 128^3 voxels grid.

Fermi and Kepler hardware support 32-bit floating point (FP32) atomic add operation on both images and global memory pointers, which is exposed through the `NV_shader_atomic_float` extension. Times marked with a star correspond to the results obtained with this native `atomicAdd` operation instead of our emulation. The right table compares our results using an FP32 voxel grid with VoxelPipe [Pantaleoni 11].

As can be seen, our approach provides as good or even better results than [Pantaleoni 11] when no merging is done (which does not give the same voxelization result) or when native atomic operations can be used (as is the case for R32F and RG16 voxel formats). For RG16 voxel formats (two normalized short integers), we perform the merging inside each voxel using the native `atomicAdd` operating on an unsigned `int` value, which works as long as the 16 bits per component do not overflow.

However, performance drops dramatically when we use our atomic emulation in floating-point format (R32F, nonstarred results) or our atomic moving average on RGBA8 formats (Section 22.4.2). Our FP32 `atomicAdd` emulation appears up to 25 times slower than the native operation when a lot of collisions occur. Paradoxically

		Std. raster.		Cons. raster.		VoxelPipe	
Format	Res	Write	Merge	Write	Merge	Write	Merge
R32F	128	1.19	1.24* /1.40	1.63	2.41* /62.1	4.80	5.00
	512	1.38	2.73* /5.15	1.99	5.30* /30.74	5.00	7.50
RG16	128	1.18	1.24	1.63	2.16		
	512	1.44	2.38	2.03	4.46		
RGBA8	128	1.18	1.40	1.63	69.80		
	512	1.47	5.30	2.07	31.40		

Table 22.1. Execution time (in milliseconds) of our voxelization algorithm and comparison with VoxelPipe on the Stanford dragon mesh. Times marked with a star correspond to the results obtained with the hardware `atomicAdd` operation instead of our emulation.

in these cases, lower resolution voxelization ends up slower than higher resolution, due to the increase in the number of collisions encountered per voxel.

22.5 Sparse Voxelization into an Octree

The goal of our sparse voxelization is to store only the voxels that are intersected by mesh triangles instead of a full grid in order to handle large and complex scenes and objects. For efficiency, this representation is stored in the form of a sparse-voxel octree in the spirit of [Laine and Karras 10] and [Crassin et al. 09]. To simplify explanations in the following sections, we will use the compute terminology and describe our algorithm in terms of *kernels* and launching of *threads*. The way we actually perform such compute-like thread execution in OpenGL will be described in Section 22.5.6.

22.5.1 Octree Structure

Our sparse-voxel octree is a very compact pointer-based structure, implemented similarly to [Crassin et al. 09]. Its memory organization is illustrated in Figure 22.7. The root node of the tree represents the entire scene; each of its children represents an eighth of its volume and so forth for every node.

Octree nodes are stored in linear video memory in a buffer object called the *octree pool*. In this buffer, nodes are grouped into $2 \times 2 \times 2$ *node tiles*, which allows us to store a single *pointer* in each node (actually an *index* into the buffer) pointing to eight child nodes. Voxel values can be stored directly into the nodes in linear memory or can be kept in *bricks* associated with the node tiles and stored in a big 3D texture. This node-plus-brick scheme is the one used in [Crassin et al. 11] to allow fast trilinear sampling of voxel values.

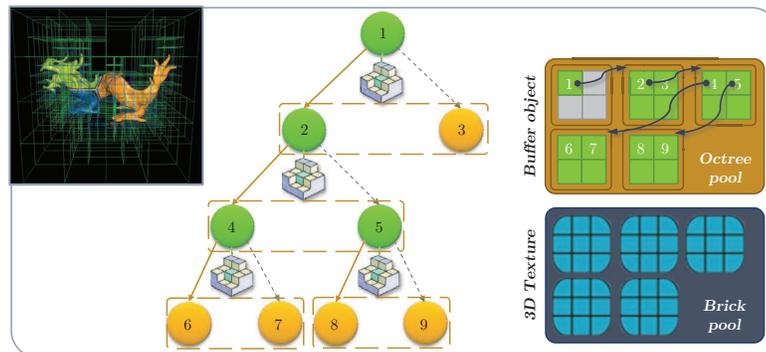


Figure 22.7. Illustration of our octree structure with bricks and its implementation in video memory.

This structure contains values for all levels of the tree, which allows querying filtered voxel data at any resolution and with increasing detail by descending the tree hierarchy. This property is highly desirable and was strongly exploited in our global illumination application [Crassin et al. 11].

22.5.2 Sparse-Voxelization Overview

In order to build the octree structure, our sparse-voxelization algorithm builds upon the regular grid voxelization we presented earlier. Our entire algorithm is illustrated in Figure 22.8. The basic idea of our approach is very simple.

We build the structure from top to bottom, one level at a time, starting from the 1-voxel root node and progressively subdividing nonempty nodes (intersected by at least one triangle) in each successive octree level of increasing resolution (step 2 in Figure 22.8). For each level, nonempty nodes are detected by voxelizing the scene at the resolutions corresponding to the resolution of the level, and a new tile of 2^3 subnodes is created for each of them. Finally, voxel-fragment values are written into the leaves of the tree and mipmapped into the interior nodes (steps 3 and 4 in Figure 22.8).

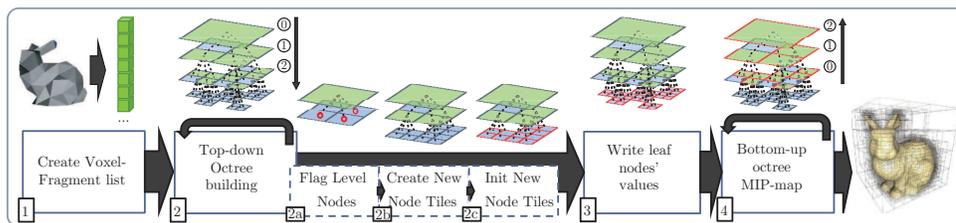


Figure 22.8. Illustration of our octree-building steps.

22.5.3 Voxel-Fragment List Construction Using an Atomic Counter

Actually revoxelizing the entire mesh multiple times, once for each level of the octree, would be very costly. Instead, we chose to voxelize it only once at the maximum resolution of the deepest octree level, and to write generated voxel fragments into a *voxel fragment list* (step 1 in Figure 22.8). This list is then used instead of the triangle mesh to subdivide the octree during the building process.

Our voxel-fragment list is a linear vector of entries stored inside a preallocated buffer object. It is made up of multiple arrays of values, one containing the 3D coordinate of each voxel fragment (encoded in one 32-bit word with 10 bits per component and 2 unused bits) and the others containing all the attributes we want to store. In our demo implementation we only keep one color per voxel fragment.

In order to fill this voxel-fragment list, we voxelize our triangle scene similarly to how we did in the first part of this chapter. The difference here is that instead of

directly writing voxel fragments into a destination 3D texture, we append them to our voxel fragment list. To manage the list, we store the index of the next available entry (that is also a counter of the number of voxel fragments in the list) as a single 32-bit value inside another buffer object.

This index needs to be accessed concurrently by thousands of threads appending voxel values, so we implement it with a new atomic counter (introduced with OpenGL 4.2). Atomic counters provide a highly optimized atomic increment/decrement operation on 32-bit integer variables. In contrast to the generic `atomicInc` or `atomicAdd` operations that allow dynamic indexing, atomic counters are designed to provide high performance when all threads operate on the same static memory region.

22.5.4 Node Subdivision

The actual subdivision of all nodes of a given octree level is done in three steps as illustrated in Figure 22.9. First, the nodes that need to be subdivided are flagged, using one thread per entry of the voxel-fragment list. Each thread simply traverses the octree from top to bottom, down to the current level (where there is no node linking subnodes), and *flags* the node in which the thread ended. Since multiple threads will end up flagging the same octree nodes, this allows us to gather all subdivision requests for a given node. This flag is implemented simply by setting the most significant bit of the children pointer of the node.

Whenever a node is flagged to be subdivided, a set of $2 \times 2 \times 2$ subnodes (a tile) needs to be allocated inside the octree pool and linked to the node. Thus, in a second step, the actual allocation of these subnode tiles is performed by launching one thread per node of the current octree level. Each thread first checks the flag of its assigned node, and if it is marked as touched, a new node tile is allocated and its index is assigned to the `childNode` pointer of the current node. This allocation of

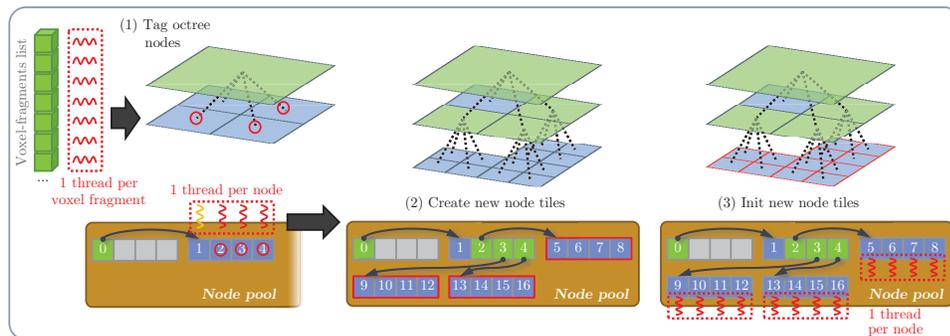


Figure 22.9. Illustration of the three steps performed for each level of the octree during the top-down construction with thread scheduling.

new node tiles inside the octree pool is done using a shared atomic counter, similarly to what we do for the voxel-fragment list (Section 22.5.3).

Finally, these new nodes need to be initialized, essentially to *null* child node pointers. This is performed in a separate pass so that one thread can be associated with each node of the new octree level (Figure 22.9, step 3).

22.5.5 Writing and Mipmapping Values

Once the octree structure has been built, the only remaining task is to fill it with the values from the voxel fragments. To do so, we first write the high-resolution voxel fragment values into the leaf nodes of the octree. This is achieved using one thread per entry of the voxel-fragment list. Each thread uses a similar scheme to the regular grid to splat and merge voxel-fragment values into the leaves (Section 22.4.2).

In a second step, we mipmap these values into the interior nodes of the tree. This is done level-per-level from bottom to top, in $n - 1$ steps for an octree of n levels. At each step, we use one thread to average the values contained in the eight sub-nodes of each non-empty node of the current level. Since we built the octree level-by-level (Section 22.5.2), node tiles get automatically sorted per level inside the octree pool. Thus, it is easy to launch threads for all nodes allocated in a given level to perform the averaging. These two steps are illustrated in Figure 22.8 (steps 3 and 4).

22.5.6 Synchronization-Free Compute-Like Kernel Launch Using draw indirect

In contrast to using CUDA or OpenCL, launching kernels with a specific number of threads (as we described) is not trivial in OpenGL. We propose to implement such kernel launches by simply using a vertex shader triggered with zero input vertex attributes. With this approach, threads are identified within the shader using the `gl_VertexID` built-in variable that provides a linear thread index.

Since our algorithm is entirely implemented on the GPU, all data necessary for each step of our approach are present in video memory. In order to provide optimal performance, we want to avoid reading back these values to the CPU to be able to launch new kernels since any readback will stall the pipeline. Instead, we rely on indirect draw calls (`glDrawArraysIndirect`) that read the call parameters from a structure stored within a buffer object directly in video memory.

This allows us to batch multiple kernel launches for successive steps of our algorithm with the actual thread configuration (the number of threads to launch and the starting offset) depending on the result of previous launches with absolutely zero CPU synchronization. Such GPU-driven kernel launch is currently not possible either in CUDA or in OpenCL.

We modify launch parameters using lightweight kernel launches with only one thread in charge of writing correct values into the draw indirect structure through global memory pointers.

With this approach, different kernels launched successively can potentially get scheduled at the same time on the GPU, and read/write ordering between two kernels is not ensured. When one kernel depends on the result of a previous kernel for its execution, we ensure that the data will be available to the threads of the second kernel by using *memory barriers* (`glMemoryBarrier()` command).

22.5.7 Results and Discussion

Table 22.2 shows computation time (in milliseconds) for the different steps of our algorithm on three representative scenes. Times marked with a star correspond to the results when atomic-based fragment merging is activated. The maximum voxelization resolution is 512^3 (9 octree levels). We use RGBA32F voxel values stored into a buffer object in global memory, and all timings have been done on a Kepler-based NVIDIA GTX680. We can observe that most of the time is spent in the octree construction, especially flagging the nodes (Section 22.5.4). Overall performance is 30% to 58% faster compared to a Fermi-based GTX480, and the atomic fragment merging is up to 80% faster.



Figure 22.10. The Sponza scene voxelized into our octree structure at a maximum resolution of respectively 512^3 , 256^3 , and 64^3 voxels and rendered without filtering.

Scene	Frag list	Octree Construction				Write	Mipmap	Total
		Flag	Create	Init	Total			
Hand	0.17	0.89	0.18	0.35	1.42	0.35/ 0.9*	0.55	2.49/ 3.04*
Dragon	3.51	4.93	0.22	0.49	5.64	2.01/ 3.05*	0.78	11.94/ 12.98*
Sponza	2.07	5.65	0.37	1.32	7.34	2.25/ 3.94*	2.09	13.75/ 15.44*

Table 22.2. Step-by-step execution time (in milliseconds) of our sparse octree voxelization for three different scenes. Times marked with a star correspond to the results when atomic-based fragment merging is activated.

Figure 22.10 shows the results of voxelizing the Sponza atrium scene into octree structures of different resolutions. We used this octree construction algorithm inside the voxel-based global illumination technique described in [Crassin et al. 11]. In this approach, a static environment must be quickly prevoxelized, and then at runtime, dynamic objects must be updated in real time inside the structure. Thanks to our fast voxelization approach, we were able to keep this structure update under 15% of the whole frame time.

Currently, one of the weakness of our approach is the requirement of preallocating the octree buffer with a fixed size. Although this may seem like a problem, it is in fact often desirable to manage this buffer as a cache, similar to what is proposed in [Crassin et al. 09].

22.6 Conclusion

In this chapter, we presented two approaches to voxelize triangle meshes, one producing a regular voxel grid and one producing a more compact sparse voxel octree. These approaches take advantage of the fast rasterization hardware of the GPU to implement efficient 3D sampling and scan conversion. Our approach dramatically reduces the geometrical cost of previous graphics-based approaches, while in most cases providing similar or slightly higher performance than state-of-the-art compute-based approaches. Although it was not detailed here, our approach supports a fast dynamic update of the octree structure, allowing us to merge dynamic objects inside a static prevoxelized environment, as demonstrated in [Crassin et al. 11]. Details can be found in the accompanying source code. Possible future work includes optimizing the voxel merging as well as the conservative rasterization implementation. In fact, the new NVIDIA Kepler architecture already improves atomic operation performance considerably.

Acknowledgments. We would like to thank Crytek for its improved version of the Atrium Sponza Palace model originally created by Marko Dabrovic. We would also like to thank the Stanford University Computer Graphics Laboratory for the Dragon model, as well as Ingo Wald for his animated hand model.

Bibliography

- [Allard et al. 10] Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, and Paul Kry. “Volume Contact Constraints at Arbitrary Resolution.” In *ACM Transactions on Graphics, Proceedings of SIGGRAPH 2010*, pp. 1–10. New York: ACM, 2010. Available online (<http://hal.inria.fr/inria-00502446/en/>).
- [Crane et al. 05] Keenan Crane, Ignacio Llamas, and Sarah Tariq. “Real-Time Simulation and Rendering of 3D Fluids.” In *GPU Gems 2*, pp. 615–634. Reading, MA: Addison Wesley, 2005.

- [Crassin et al. 09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. “GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering.” In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, 2009. Available online (<http://artis.imag.fr/Publications/2009/CNLE09>).
- [Crassin et al. 10] Cyril Crassin, Fabrice Neyret, Miguel Sainz, and Elmar Eisemann. “Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels.” In *GPU Pro*, pp. 643–676. Natick, MA: A K Peters, 2010. Available online (<http://artis.imag.fr/Publications/2010/CNSE10>).
- [Crassin et al. 11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. “Interactive Indirect Illumination Using Voxel Cone Tracing.” In *Computer Graphics Forum (Pacific Graphics 2011)*, 2011.
- [Dong et al. 04] Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng. “Real-Time Voxelization for Complex Polygonal Models.” In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference, PG '04*, pp. 43–50. Washington, DC: IEEE Computer Society, 2004. Available online (<http://dl.acm.org/citation.cfm?id=1025128.1026026>).
- [Eisemann and Décoret 08] Elmar Eisemann and Xavier Décoret. “Single-Pass GPU Solid Voxelization for Real-Time Applications.” In *Proceedings of graphics interface 2008, GI '08*, pp. 73–80. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2008.
- [Fang et al. 00] Shiaofen Fang, Shiaofen Fang, Hongsheng Chen, and Hongsheng Chen. “Hardware Accelerated Voxelization.” *Computers and Graphics* 24:3 (2000), 433–442.
- [Hasselgren et al. 05] Jon Hasselgren, Tomas Akenine-Mller, and Lennart Ohlsson. “Conservative Rasterization.” In *GPU Gems 2*. Reading, MA: Addison Wesley, 2005.
- [Huang et al. 98] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. “An Accurate Method for Voxelizing Polygon Meshes.” In *Proceedings of the 1998 IEEE Symposium on Volume Visualization, VVS '98*, pp. 119–126. New York: ACM, 1998. Available online (<http://doi.acm.org/10.1145/288126.288181>).
- [Kaplanyan and Dachsbacher 10] Anton Kaplanyan and Carsten Dachsbacher. “Cascaded Light Propagation Volumes for Real-time Indirect Illumination.” In *Proceedings of I3D*, 2010.
- [Laine and Karras 10] Samuli Laine and Tero Karras. “Efficient Sparse Voxel Octrees.” In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, pp. 55–63. New York: ACM Press, 2010.
- [Li et al. 05] Wei Li, Zhe Fan, Xiaoming Wei, and Arie Kaufman. “Flow Simulation with Complex Boundaries.” In *GPU Gems 2*, pp. 615–634. Reading, MA: Addison Wesley, 2005.
- [Pantaleoni 11] Jacopo Pantaleoni. “VoxelPipe: A Programmable Pipeline for 3D Voxelization.” In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11*, pp. 99–106. New York: ACM, 2011. Available online (<http://doi.acm.org/10.1145/2018323.2018339>).
- [Schwarz and Seidel 10] Michael Schwarz and Hans-Peter Seidel. “Fast Parallel Surface and Solid Voxelization on GPUs.” In *ACM SIGGRAPH Asia 2010 papers, SIGGRAPH ASIA*

- '10, pp. 179:1–179:10. New York: ACM, 2010. Available online (<http://doi.acm.org/10.1145/1866158.1866201>).
- [Thiedemann et al. 11] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Müller. “Voxel-Based Global Illumination.” In *Symposium on Interactive 3D Graphics and Games, Proceedings of I3D*, pp. 103–110. New York: ACM, 2011.
- [Zhang et al. 07] Long Zhang, Wei Chen, David S. Ebert, and Qunsheng Peng. “Conservative Voxelization.” *The Visual Computer* 23 (2007), 783–792. Available online (<http://dl.acm.org/citation.cfm?id=1283953.1283975>).