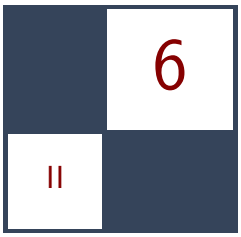

Progressive Screen-Space Multichannel Surface Voxelization

Athanasios Gaitatzes and Georgios Papaioannou

6.1 Introduction

An increasing number of techniques for real-time global illumination effects rely on volume data. Such representations allow the fast, out-of-order access to spatial data from any deferred shading graphics pipeline stage as in [Thiedemann et al. 11, Mavridis and Papaioannou 11, Kaplanyan and Dachsbacher 10]. For dynamic environments where both the geometry of the scene and the illumination can arbitrarily change between frames, these calculations must be performed in real time. However, when the per frame time budget is limited due to other, more important operations that must take place while maintaining a high frame rate, the fidelity of full-scene voxelization has to be traded for less accurate but faster techniques. This is especially true for video games, where many hundreds of thousands of triangles must be processed in less than 2–3 ms. In this chapter we present the novel concept of *progressive voxelization*, an incremental image-based volume generation scheme for fully dynamic scenes that addresses the view-dependency issues of image-based voxelization within the above time constraints.

Screen-space volume generation methods provide very fast and guaranteed response times compared to geometry-based techniques but suffer from view-dependency. More specifically, any technique that is performed entirely in screen space (as in deferred shading) considers only geometry that has been rendered into the depth buffer and thus has the following strong limitations: First, it ignores geometry located outside the field of view. Second, it ignores geometry that is inside the view frustum but occluded by other objects. Yet these geometry parts may have a significant influence on the desired final result (see our indirect illumination case study in this article).

In single-frame screen-space voxelization, volume attributes already available as fragment data in view-dependent image buffers are transformed and rasterized

(*injected*) into the volume buffer to form a partial volume of the observed space. These commonly include the view camera G-buffers like depth, albedo, normals, and the light sources' *reflective shadow maps* (RSMs) [Dachsbacher and Stamminger 05]. The injection procedure is explained in more detail in [Kaplanyan 09] and Section 6.2.2. Since the only volume samples that can be produced in each frame are the ones that are visible in at least one of the images available in the rendering pipeline, each time the (camera or light) view changes, a new set of sample points becomes available and the corresponding voxels are generated from scratch to reflect the newly available image samples. Thus the generated volume will never contain a complete voxelization of the scene. This leads to significant frame-to-frame inconsistencies and potentially inadequate volume representations for the desired volume-based effect, especially when the coverage of the scene in the available image buffers is limited.

To alleviate the problems of screen-space voxelization techniques, but maintain their benefit of predictable, controllable, and bound execution time relative to full-scene volume generation methods, we introduce the concept of *progressive voxelization* (PV). The volume representation is incrementally updated to include the newly discovered voxels and discard the set of invalid voxels, which are not present in any of the current image buffers. Using the already available camera and light source buffers, a combination of volume injection and voxel-to-depth-buffer reprojection scheme continuously updates the volume buffer and discards invalid voxels, progressively constructing the final voxelization.

The algorithm is lightweight and operates on complex dynamic environments where geometry, materials, and lighting can change arbitrarily. Compared to single-frame screen-space voxelization, our method provides improved volume coverage (completeness) over nonprogressive methods while maintaining its high performance merits.

We demonstrate our technique by applying it as an alternative voxelization scheme for the *light propagation volumes* (LPV) diffuse global illumination method of [Kaplanyan and Dachsbacher 10]. However, being a generic multiattribute scalar voxelization method, it can be used in any other real-time volume generation problem.

6.2 Overview of Voxelization Method

Our progressive voxelization scheme is able to produce stable and valid volume data in a geometry-independent manner. As the user interacts with the environment and dynamic objects move or light information changes, new voxel data are accumulated into the initial volume and old voxels are invalidated or updated if their projection in any of the image buffers (camera or light) proves inconsistent with the respective available recorded depth. For a schematic overview see Figure 6.1, and for a resulting voxelization see Figures 6.4 and 6.5.

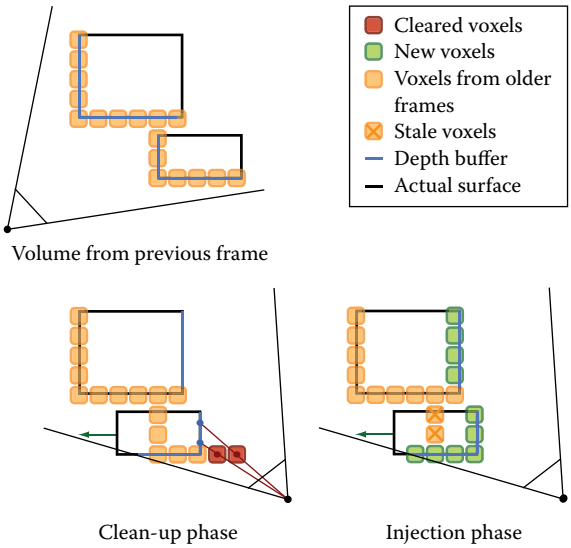

Figure 6.1. Schematic overview of the algorithm. During the cleanup phase each voxel is tested against the available depth images. If the projected voxel center lies in front of the recorded depth, it is cleared; otherwise it is retained. During the injection phase, voxels are "turned-on" based on the RSM-buffers and the camera-based depth buffer.

In each frame, two steps are performed: First, in a *cleanup* stage, the volume is swept voxel-by-voxel and the center of each voxel is transformed to the eye-space coordinate system of the buffer and tested against the available depth image value, which is also projected to eye-space coordinates. If the voxel lies closer to the image buffer viewpoint than the recorded depth, the voxel is invalidated and removed. Otherwise, the current voxel attributes are maintained. The update of the volume is performed by writing the cleared or retained values into a separate volume in order to avoid any atomic write operations and thus make the method fast and a very broadly applicable one. At the end of each cleanup cycle, the two volume buffers are swapped. After the cleanup phase, samples from all the available image buffers are injected into the volume (similar to the LPV method [Kaplanyan 09]).

When multiple image buffers are available, the cleanup stage is repeated for each image buffer, using the corresponding depth buffer as input for voxel invalidation. Each time, the currently updated (read) and output (written) buffers are swapped. The current image buffer attributes are then successively injected in the currently updated volume. The whole process is summarized in Figure 6.1.

```
in vec3 voxel_position, voxel_tex_coord;
uniform float voxel_r; // voxel radius
uniform sampler3D vol_shR, vol_shG, vol_shB, vol_normals;

void main (void)
{
  vec4 voxel_pos_wcs = vec4 (voxel_position, 1.0);
  vec3 voxel_pos_css = PointWCS2CSS (voxel_pos_wcs.xyz);
  vec3 voxel_pos_ecs = PointWCS2ECS (voxel_pos_wcs.xyz);
  vec3 zbuffer_ss = MAP_-1To1_0To1 (voxel_pos_css);
  float depth = SampleBuf (zbuffer, zbuffer_ss.xy).x;
  vec3 zbuffer_css = vec3 (voxel_pos_css.xy, 2.0*depth-1.0);
  vec3 zbuffer_ecs = PointCSS2ECS (zbuffer_css);

  vec3 voxel_mf_wcs = voxel_pos_wcs.xyz + voxel_r * vec3(1.0);
  voxel_mf_wcs = max (voxel_mf_wcs,
                      voxel_pos_wcs.xyz + voxel_half_size);
  vec3 voxel_mb_wcs = voxel_pos_wcs.xyz + voxel_r * vec3(-1.0);
  voxel_mb_wcs = min (voxel_mb_wcs,
                      voxel_pos_wcs.xyz - voxel_half_size);
  vec3 voxel_mf_ecs = PointWCS2ECS (voxel_mf_wcs);
  vec3 voxel_mb_ecs = PointWCS2ECS (voxel_mb_wcs);
  float bias = distance (voxel_mf_ecs, voxel_mb_ecs);

  vec4 shR_value = SampleBuf (vol_shR, voxel_tex_coord);
  vec4 shG_value = SampleBuf (vol_shG, voxel_tex_coord);
  vec4 shB_value = SampleBuf (vol_shB, voxel_tex_coord);
  vec4 normal_value = SampleBuf (vol_normals, voxel_tex_coord);

  if (voxel_pos_ecs.z > zbuffer_ecs.z + bias) { // discard
    normal_value = vec4 (0,0,0,0);
    shR_value = shG_value = shB_value = vec4 (0,0,0,0);
  }

  // keep
  gl_FragData[0] = normal_value;
  gl_FragData[1] = shR_value;
  gl_FragData[2] = shG_value;
  gl_FragData[3] = shB_value;
}
```

Listing 6.1. Cleanup phase fragment shader.

6.2.1 Cleanup Phase

Throughout the entire voxelization process, each voxel goes through three state transitions: "turn-on," "turn-off," and "keep" (see Listing 6.1). The "turn-on" state change is determined during the injection phase. During the cleanup stage we need to be able to determine if the state of the voxel will be retained or turned off (cleared). For each one of the available depth buffers, each voxel center $\mathbf{p}_v$ is transformed to eye-space coordinates $\mathbf{p}'_v$; accordingly the corresponding image buffer depth $Z(\mathbf{p}')$ is transformed to eye-space coordinates z_e.

Expressing the coordinates in the eye reference frame (Figure 6.2), if $\mathbf{p}'_{v,z} > z_e$ the voxel must be cleared, as it lies in front of the recorded depth boundary in the

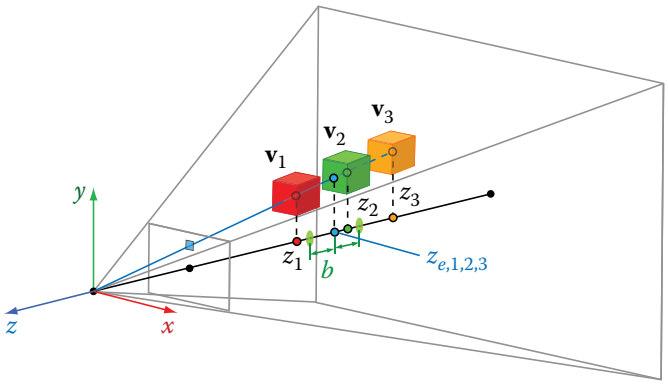

Figure 6.2. Cleanup stage: Voxels beyond the boundary depth zone are retained (orange), while voxels closer to the buffer center of projection are rejected (red). Voxels that correspond to the depth value registered in the buffer must be updated (green).

image buffer. However, the spatial data are quantized according to the volume resolution and therefore a bias b has to be introduced in order to avoid rejecting boundary samples. Since the depth comparison is performed in eye-space, b is equal to the voxel's $\mathbf{p}_v$ radius (half diagonal) clamped by the voxel boundaries in each direction. Therefore the rejection condition becomes

$$\mathbf{p}'_{v,z} > z_e + b.$$

The example in Figure 6.2 explains the cleanup and update state changes of a voxel with respect to the available depth information in an image buffer. All voxels in the figure correspond to the same image buffer sample with eye-space value $z_{e,1,2,3}$. Voxel $\mathbf{v}_1$ is rejected (cleared) because z_1 is greater than $z_{e,1,2,3} + b$. Voxel $\mathbf{v}_2$ must be updated since it lies within the boundary depth zone $[z_{e,1,2,3} - b, z_{e,1,2,3} + b]$. Finally, voxel $\mathbf{v}_3$ is retained, since it lies beyond the registered depth value.

6.2.2 Injection Phase

In the injection phase, a rectangular grid of point primitives corresponding to each depth image buffer is sent to a vertex shader that offsets the points according to the stored depth. The points are subsequently transformed to world space and finally to volume-clip space. If world-space or volume clip-space coordinates are already available in the buffers, they are directly assigned to the corresponding injected points. The volume clip-space depth is finally used to determine the slice in the volume where the point sample attributes are accumulated (see Listing 6.2). At the end of this stage, the previous version of the scene's voxel representation has been updated to include a partial voxelization of the scene based on the newly

```
// Vertex-Shader Stage

flat out vec2 tex_coord;
uniform sampler2D zbuffer;

void main (void)
{
  tex_coord = gl_Vertex.xy;
  float depth = SampleBuf (zbuffer, tex_coord).x;

  // screen space --> canonical screen space
  vec3 pos_css = MAP_0To1_-1To1 (vec3 (gl_Vertex.xy, depth));

  // canonical screen space --> object space
  vec3 pos_wcs = PointCSS2WCS (zbuffer_css);

  // world space --> clip space
  gl_Position = gl_ModelViewProjectionMatrix *
                vec4 (pos_wcs, 1.0);
}

// Geometry-Shader Stage

layout(points) in;
layout(points, max_vertices = 1) out;

uniform int vol_depth;
flat in vec2 tex_coord[];
flat out vec2 gtex_coord;

void main (void)
{
  gtex_coord = tex_coord[0];
  gl_Position = gl_PositionIn[0];
  gl_Layer = int (vol_depth * MAP_-1To1_0To1 (gl_Position.z));

  EmitVertex();
}
```

Listing 6.2. Injection phase using a geometry shader to select the destination slice of the volume for the point samples.

injected point samples. The resolution of the grid of 2D points determines how much detail of the surfaces represented by the depth buffer is transferred into the volume and whether or not the geometry is sparsely sampled. If too few points are injected, the resulting volume will have gaps. This may be undesirable for certain application cases, such as the LPV method [Kaplanyan 09] or algorithms based on ray marching.

6.2.3 Single-Pass Progressive Algorithm

In order to transfer the geometric detail present in the G-buffers to the volume representation and ensure a dense population of the resulting volume, a large resolution for the grid of injected points must be used. However, the injection stage

involves rendering the point grid using an equal number of texture lookups and, in some implementations, a geometry shader. This has a potentially serious impact on performance (see Figure 6.8), especially for multiple injection viewpoints.

We can totally forgo the injection phase of the algorithm and do both operations in one stage. Using the same notation as before, the logic of the algorithm remains practically the same. If the projected voxel center lies in front of the recorded depth (i.e., $\mathbf{p}'_{v,z} > z_e + b$), it is still cleared. If the projected voxel center lies behind the recorded depth (i.e., $\mathbf{p}'_{v,z} < z_e - b$), the voxel is retained; otherwise it is turned-on (or updated) using the attribute buffers information. The last operation practically replaces the injection stage.

As we are effectively sampling the geometry at the volume resolution instead of doing so at higher, image-size-dependent rate and then down-sampling to volume resolution, the resulting voxelization is expected to degrade. However, since usually depth buffers are recorded from multiple views, missing details are gradually added. A comparison of the method variations and analysis of their respective running times is given in Section 6.5.

6.3 Progressive Voxelization for Lighting

As a case study, we applied progressive voxelization to the problem of computing indirect illumination for real-time rendering. When using the technique for lighting effects, as in the case of the LPV algorithm of [Kaplanyan 09] or the ray marching techniques of [Thiedemann et al. 11, Mavridis and Papaioannou 11], the volume attributes must include occlusion information (referred to as *geometry volume* in [Kaplanyan 09]), sampled normal vectors, direct lighting (VPLs), and optionally surface albedo in the case of secondary indirect light bounces. Direct illumination and other accumulated directional data are usually encoded and stored as low-frequency spherical harmonic coefficients (see [Sloan et al. 02]).

Virtual point lights (VPLs) are points in space that act as light sources and encapsulate light reflected off a surface at a given location. In order to correctly accumulate VPLs in the volume, during the injection phase, a separate volume buffer is used that is cleared in every frame in order to avoid erroneous accumulation of lighting. For each RSM, all VPLs are injected and additively blended. Finally, the camera attribute buffers are injected to provide view-dependent dense samples of the volume. If lighting from the camera is also exploited (as in our implementation), the injected VPLs must replace the corresponding values in the volume, since the camera direct lighting buffer provides cumulative illumination. After the cleanup has been performed on the previous version of the attribute volume V_{prev}, nonempty voxels from the separate injection buffer replace corresponding values in V_{curr}. This ensures that potentially stale illumination on valid volume cells from previous frames is not retained in the final volume buffer. In Figure 6.3 we can see the results of progressive voxelization and its application to diffuse indirect lighting.

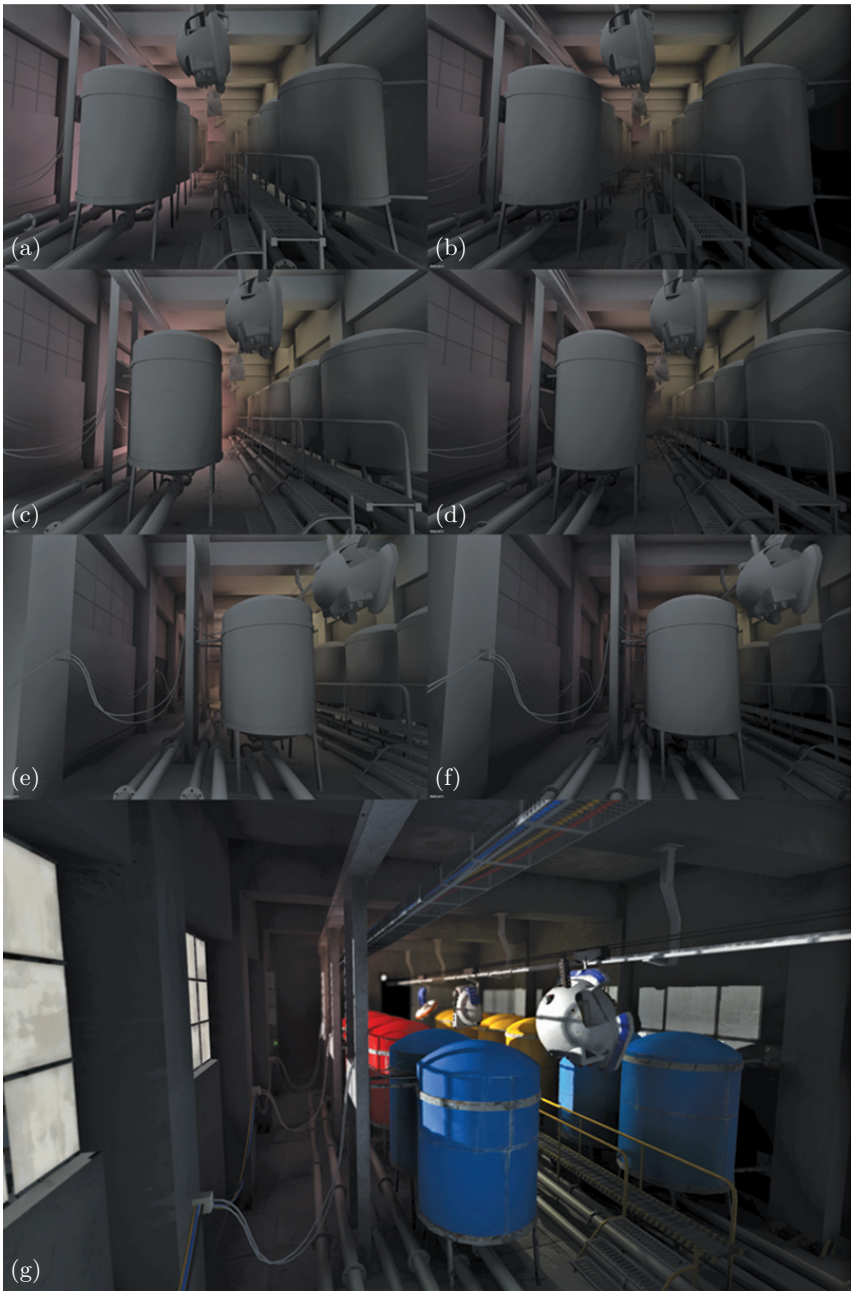

Figure 6.3. (a–f) As the camera moves left to right, we observe correct indirect illumination. (g) Final rendering of the room.

6.4 Implementation

The progressive voxelization method runs entirely on the GPU and has been implemented in a deferred shading renderer using basic OpenGL 3.0 operations on a NVIDIA GTX 285 card with 1 GB of memory. We have implemented two versions of the buffer storage mechanism in order to test their respective speed. The first uses 3D volume textures along with a geometry shader that sorts injected fragments to the correct volume slice. The second unwraps the volume buffers into 2D textures and dispenses with the expensive geometry processing (respective performance can be seen in Figure 6.8).

The texture requirements are two volume buffers for ping-pong rendering (V_{prev}, V_{curr}). Each volume buffer stores N-dimensional attribute vectors $\mathbf{a}$ and corresponds to a number of textures (2D or 3D) equal to $\lceil N/4 \rceil$, for 4-channel textures. For the reasons explained in Section 6.3 an additional N-dimensional volume buffer is required for lighting applications. In our implementation we need to store surface normals and full color spherical harmonics coefficients for incident flux in each volume buffer, which translates to 3×4 textures in total.

In terms of volume generation engine design, the user has the option to request several attributes to be computed and stored into floating-point buffers for later use. Among them are surface attributes like albedo and normals, but also dynamic lighting information and radiance values in the form of low-order spherical harmonics (SH) coefficients representation (either monochrome radiance or full color encoding, i.e., separate radiance values per color band). In our implementation the radiance of the corresponding scene location is calculated and stored as a second-order spherical harmonic representation for each voxel. For each color band, four SH coefficients are computed and encoded as RGBA float values.

6.5 Performance and Evaluation

In terms of voxelization robustness, our algorithm complements single-frame screen-space voxelization and supports both moving image viewpoints and fully dynamic geometry and lighting. This is demonstrated in Figures 6.4 and 6.5. In addition, in Figure 6.6, a partial volume representation of the Crytek Sponza II Atrium model is generated at a 64^3 resolution and a 128^2-point injection grid using single-frame and progressive voxelization. Figures 6.6(a) and (b) are the single-frame volumes from two distinct viewpoints. Figure 6.6(c) is the progressive voxelization after the viewpoint moves across several frames. Using the partial single-frame volumes for global illumination calculation, we observe abrupt changes in lighting as the camera reveals more occluding geometry (e.g., left arcade wall and floor in Figures 6.6(d) and (e)). However, the situation is gradually remedied in the case of progressive voxelization, since newly discovered volume data are retained for use in following frames (Figures 6.6(f) and (g)).

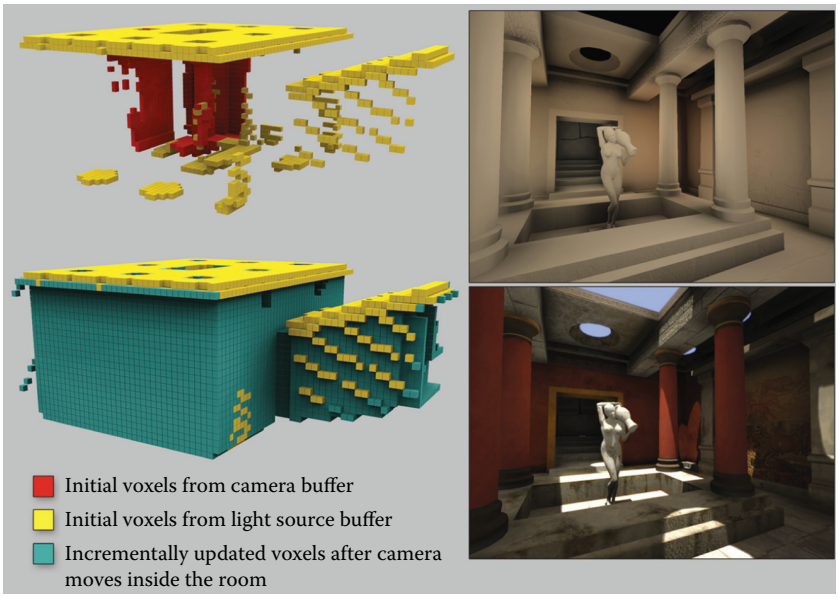

Figure 6.4. Left: Screen-space voxelization after one step of the process having injected the camera and light buffers (top), and voxelization of the scene after the camera has moved for several frames (bottom). Right: Example of resulting indirect illumination (top) and final illumination (bottom).

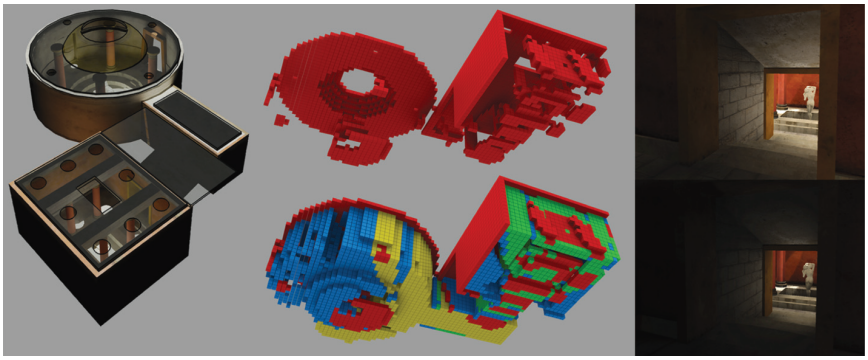

Figure 6.5. Progressive voxelization of a scene. Red voxels correspond to screen-space voxelization using image buffers from the current frame only, while other colors refer to voxels generated during previous frames using PV. On the right, volume-based global illumination results using the corresponding volumes. PV (top) achieves more correct occlusion and stable lighting.

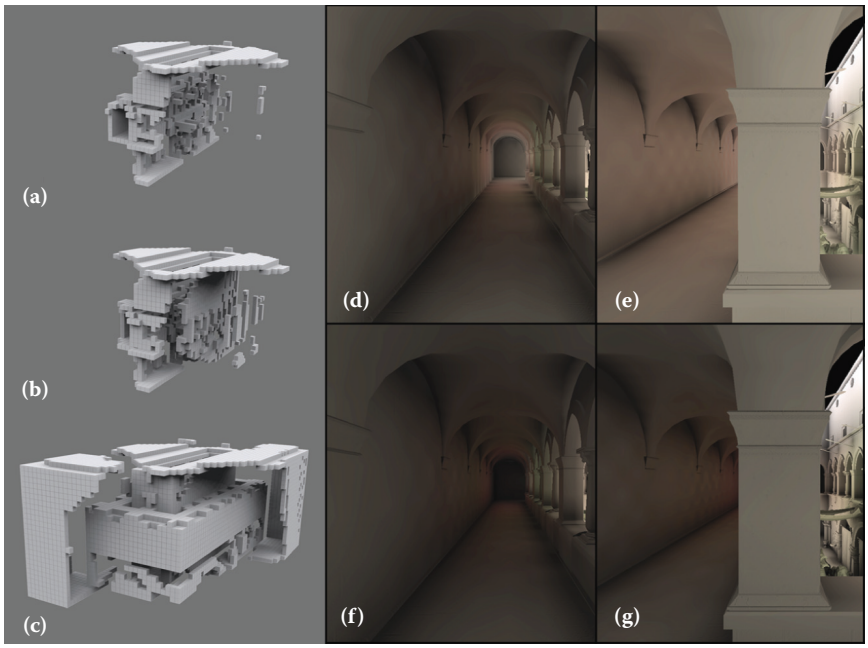

Figure 6.6. Comparison of the voxelization of the Crytek Sponza II Atrium. (a, b) Single-frame screen-space voxelization from two distinct viewpoints where it is not possible to capture all environment details as no information exists in the buffers. (c) Progressive voxelization produced over several frames. (d, e) Indirect lighting buffers corresponding to the single frame voxelization of (a) and (b). (f, g) PV indirect lighting buffers (of the voxelization in (c)).

Figure 6.7 demonstrates progressive voxelization in a dynamic environment in real time. In particular, it shows an animated sequence of a scene with moving and deformable objects, as well as the corresponding voxelization from the camera viewpoint. Observe how the wall behind the closed door is not initially present in the volume, but after the door opens, it is gradually added to the volume and remains there even after the door swings back. The same holds true for the geometry behind the character. Notice also how the voxels representing the articulated figure correctly change state as the figure moves.

Figure 6.8 shows a decomposition of the total algorithm running time into the cleanup and injection stage times respectively versus different volume buffer resolutions for three different injection grid sizes using the 3D volume textures implementation (left) and the 2D textures implementation (center). For fixed injection grid resolutions, we have observed that injection times are not monotonically increasing with respect to volume size as one would expect. The performance also decreases when the buffer viewpoint moves close to geometry.

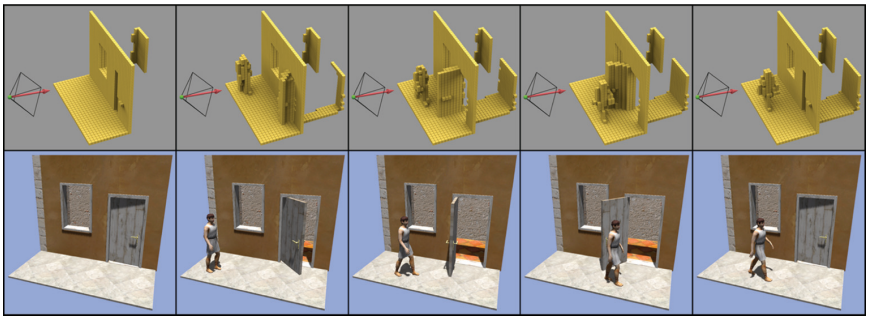

Figure 6.7. Screen-space voxelization of a dynamic scene containing an articulated object using only camera-based injection.

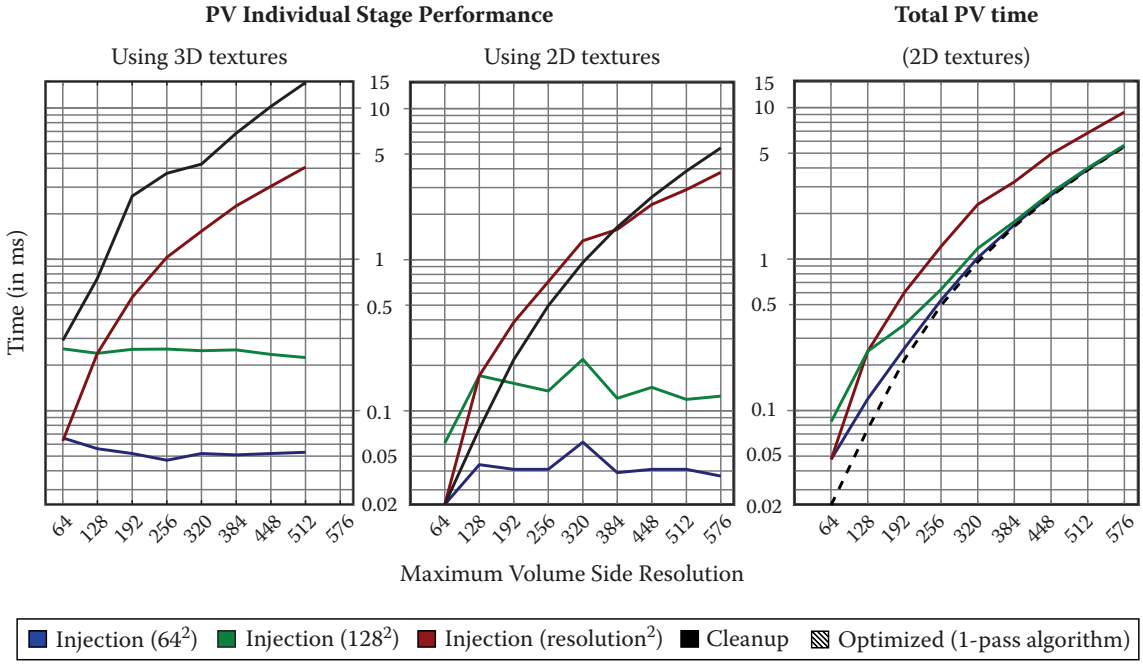

Figure 6.8. Running time (in ms) for the cleanup and injection stages against different volume resolutions for the Crytek Sponza II Atrium model using the 3D volume textures implementation (left) and the 2D textures implementation (center). We used a single G-buffer (camera) as input and one multiple render target (four floats) as output. Injection is measured for three different grid sizes, one being proportional to the volume side. We also show the total progressive voxelization times (right). Note that the performance of the optimized progressive voxelization is identical to that of the cleanup stage.

We attribute this to the common denominator of both cases, namely the fact that pixel overdraw is induced, as points are rasterized in the same voxel locations. This is particularly evident in the blue curve of the 64^2 injection stage graph of Figure 6.8 (left). Note that this behavior is an inherent attribute of injection techniques in general; screen-space voxelization methods depend heavily on the sampling rate used. When this rate is incompatible with the voxel space resolution, holes might appear (undersampling). To ensure adequate coverage of the voxel grid, dense screen-space point samples are drawn, which in turn leads to overdraw problems in many cases. One can use an injection grid proportional to the volume resolution, which partially alleviates the overdraw issue but in turn decreases performance as can be seen in the red curve of the injection graph of Figure 6.8 (left and center).

The time required for a single-frame screen-space voxelization (one G-buffer) equals the time of our injection stage plus a very small overhead to clear the volume buffer, since the two operations are equivalent. Thus, the only difference in the execution time of progressive voxelization is the cleanup stage time. With regard to the quality of the two methods, PV offers more stable and accurate results as new viewpoints gradually improve the volume.

The total voxelization time (Figure 6.8, right) is the sum of the cleanup and injection stages. As the cleanup stage performance depends only on the volume resolution and not on the injection grid size, it vastly improves the voxelization quality compared to using only screen-space injection from isolated frames, at a constant overhead per frame. Especially when applied to global illumination calculations, where small volumes are typically used, the version of the algorithm that uses 2D textures (Figure 6.8, center) has a significantly lower execution footprint. This is because it is not influenced by the geometry shader execution of the 3D textures version (Figure 6.8, left), though both methods are affected by pixel overdraw during injection.

The performance of the optimized progressive voxelization is identical to that of the cleanup stage as expected, since it is essentially a modified cleanup stage. It follows that the dual stage version performance will always be slower than the optimized one.

The maximum volume resolution reported is due to hardware resource limitations on the number and size of the allocated buffers and not algorithm bounds.

In Table 6.1 we report the voxelization performance results for several scenes using our method and the geometry-based multichannel full scene voxelization method of [Gaitatzes et al. 11], which, as ours, is based on the rendering pipeline (GPU). We show a big speed improvement even when adding to the whole process the G-buffers creation time.

In Table 6.2 we report on the quality of our voxelization method. The camera was moved around the mesh for several frames, in order for the algorithm to *progressively* compute the best possible voxelization. For several models and resolutions we show the Hausdorff distance between the original mesh and the

Scene	Grid	GS	G-buffers	PV
	Size	4-floats	Creation	4-floats
Conference	128^3	31.73	3.2	0.28
(282K tris)	512^3	64.67		4.93
Dragon	128^3	198.33	59	0.18
(871K tris)	512^3	–		6.98
Turbine Blade	128^3	265.7	121	0.14
(1.76M tris)	512^3	–		5.37
Hairball	128^3	436.2	–	0.33
(2.88M tris)	320^3	–		4.04

Table 6.1. Voxelization timings (in ms) of various scenes using progressive voxelization (PV) and the geometry slicing (GS) method of [Gaitatzes et al. 11] with 11 output vertices. We present the total (injection + cleanup) performance values of our 2D textures implementation using an injection grid proportional to the volume size, which is our algorithm's worst case as can be seen from the red plot of Figure 6.8.

Scene	Grid	Hausdorff	
	Size	% d_H	(X, Y)
Bunny	64^3	0.3289	0.2168
	128^3	0.1694	0.1091
(69.5K tris)	256^3	0.1064	–
Dragon	64^3	0.3621	0.2565
	128^3	0.1878	0.1289
(871K tris)	256^3	0.1256	0.0645
Turbine Blade	64^3	0.3457	0.2763
	128^3	0.1821	0.1424
(1.76M tris)	256^3	0.1232	0.0697

Table 6.2. Comparison of a full voxelization. We record the normalized (with respect to the mesh bounding box diagonal) average Hausdorff distance (percent). Mesh X is the original mesh to be voxelized and Y is the point cloud consisting of the voxel centers of the voxelization using PV (column 3) and a geometry-based full scene voxelization (column 4).

resulting voxelization using the PV method (see column 3). We notice that our voxelized object (voxel centers) is on average 0.1% different from the original mesh. In addition, we report the Hausdorff distance between the original mesh and the geometry-based full scene voxelization of [Gaitatzes et al. 11] (see col-

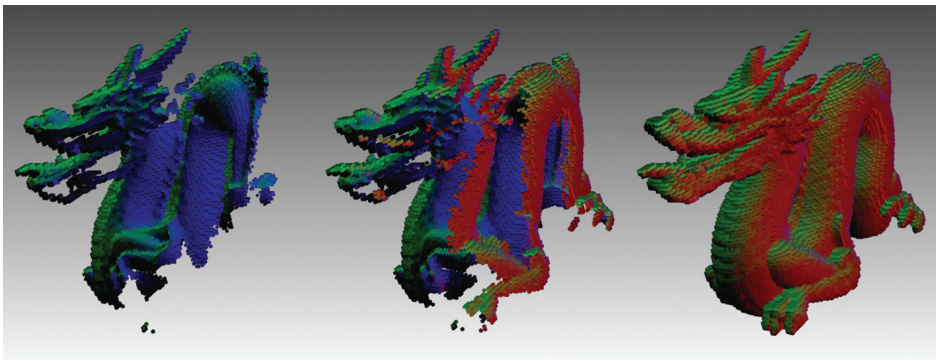

Figure 6.9. A series of voxelizations of the dragon model at 128^3 resolution showing the normal vectors. The voxelization is incrementally updated over several frames as the camera moves around the model.

umn 4). We observe that the difference between the corresponding volumes is in the 0.01% range.

In Figure 6.9 we show a series of voxelizations of the dragon model using only the camera G-buffers. In addition, we show the respective Hausdorff distance between the original dragon model and the computed voxel centers (see plot in Figure 6.10). The voxelization is incrementally updated and improved over

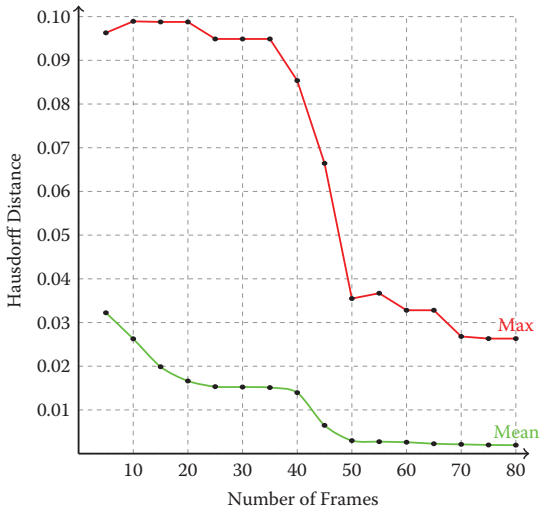

Figure 6.10. The decreasing Hausdorff distance between the original dragon model and the computed progressive voxelizations of Figure 6.9.

several frames as the camera does a complete rotation around each of the principal axis for an equal amount of frames. As the animation progresses, we observe that the Hausdorff distance decreases as the process converges to a full voxelization.

6.6 Limitations

One limitation of our method is that the cleanup phase will only remove invalid voxels that are visible in any of the current image buffers (camera multiple render targets and light RSMs). The visible invalid voxels will be removed from the voxelization the next time they appear in the image buffers. However, the correctness of the voxelization cannot be guaranteed for existing voxels that are not visible in any buffer. For moving geometry, some progressively generated voxels may become stale, as shown in the case of the bottom right of Figure 6.1. Nevertheless, in typical dynamic scenes, the stale voxels are often eliminated either in subsequent frames due to their invalidation in the moving camera buffer or due to their invalidation in other views in the same frame (see Figure 6.11).

Another limitation is that the extents of the voxelization region must remain constant throughout volume updates; otherwise computations are performed with stale buffer boundaries. When the bounding box of the scene is modified or the scene changes abruptly or it is reloaded, the attribute volumes must be deleted and progressively populated again. This is also the reason why the cascaded light propagation volumes method of [Kaplanyan and Dachsbacher 10] could not take advantage of progressive voxelization for the cascades near the user, as the method assumes that they follow the user around, constantly modifying the current volume extents.

Figure 6.11. Correct indirect shadowing effects and color bleeding: Stale voxels from one view (behind the tank) are effectively invalidated in other views (reflective shadow map).

6.7 Conclusion

We have presented a novel screen-space method to progressively build a voxelization data structure on the GPU. Our method achieves improved quality over nonprogressive methods, while it maintains the high performance merits of screen-space techniques.

6.8 Acknowledgments

The Atrium Sponza II Palace in Dubrovnik model was remodeled by Frank Meinl at Crytek. The original Sponza model was created by Marko Dabrovic in early 2002. The Bunny and Dragon models are provided courtesy of the Stanford University Computer Graphics Laboratory.

Bibliography

[Dachsbacher and Stamminger 05] Carsten Dachsbacher and Marc Stamminger. "Reflective Shadow Maps." In *Symposium on Interactive 3D Graphics and Games (I3D)*, pp. 203–231. New York: ACM, 2005.

[Gaitatzes et al. 11] Athanasios Gaitatzes, Pavlos Mavridis, and Georgios Papaioannou. "Two Simple Single-Pass GPU Methods for Multi-channel Surface Voxelization of Dynamic Scenes." In *Pacific Conference on Computer Graphics and Applications—Short Papers (PG)*, pp. 31–36. Aire-la-Ville, Switzerland: Eurographics Association, 2011.

[Kaplanyan 09] Anton Kaplanyan. "Light Propagation Volumes in CryEngine 3." SIGGRAPH Course: Advances in Real-Time Rendering in 3D Graphics and Games, SIGGRAPH 2009, New Orleans, LA, August 3, 2009.

[Kaplanyan and Dachsbacher 10] Anton Kaplanyan and Carsten Dachsbacher. "Cascaded Light Propagation Volumes for Real-Time Indirect Illumination." In *Symposium on Interactive 3D Graphics and Games (I3D)*, pp. 99–107. New York: ACM, 2010.

[Mavridis and Papaioannou 11] Pavlos Mavridis and Georgios Papaioannou. "Global Illumination Using Imperfect Volumes." Presentation, International Conference on Computer Graphics Theory and Applications (GRAPP), Algarve, Portugal, 2011.

[Sloan et al. 02] Peter-Pike Sloan, Jan Kautz, and John Snyder. "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments." In *29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pp. 527–536. New York: ACM, 2002.

[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 (I3D)*, pp. 103–110. New York: ACM, 2011.