DOI: 10.1111/cgf.15262

COMPUTER GRAPHICS *forum*
Volume 44 (2025), number 1, e15262

Dynamic Voxel-Based Global Illumination

Alejandro Cosin Ayerbe,[1] Pierre Poulin[2] and Gustavo Patow[1]

[1]ViRVIG-UdG, Girona, Spain
aca_1983_@hotmail.com, gustavo.patow@udg.edu
[2]Université de Montréal, Montréal, Canada
poulin@iro.umontreal.ca

Abstract

Global illumination computation in real time has been an objective for Computer Graphics since its inception. Unfortunately, its implementation has challenged up to now the most advanced hardware and software solutions. We propose a real-time voxel-based global illumination solution for a single light bounce that handles static and dynamic objects with diffuse materials under a dynamic light source. The combination of ray tracing and voxelization on the GPU offers scalability and performance. Our divide-and-win approach, which ray traces separately static and dynamic objects, reduces the re-computation load with updates of any number of dynamic objects. Our results demonstrate the effectiveness of our approach, allowing the real-time display of global illumination effects, including colour bleeding and indirect shadows, for complex scenes containing millions of polygons.

Keywords: rendering, global illumination, real-time rendering

CCS Concepts: • Computing methodologies → Rendering; Ray tracing; Visibility

1. Introduction

Offline rendering can nowadays deliver among the most realistic images for cinema and architecture. Its achievements are often directly linked to its ability to generate visual effects involving ubiquitous global illumination. However, its real-time implementation poses several challenges that led to much progress over the years. Among the main significant approaches, let us mention the use of simplified light transport [DS05, NSW09, DS06], hybrid offline/online approaches such as light probes [SL17] and pre-computed irradiance [GSHG98] and targeted rendering approaches [KLM*19].

Despite all these great efforts, accurate global illumination in real time remains elusive. There are several reasons for this, but the main ones are the stark contrast with the computational power of modern hardware, no matter how impressive it could be, and the computations required for global illumination, initially revealed by the *rendering equation* [Kaj86]. In particular, visibility computations and light exchanges account for most of this complexity, with the former being often ignored in favour of the latter [DS05, NSW09, DS06]. Solutions trying to tackle both challenges simultaneously are scarce [AP22], requiring cumbersome and often bloated pre-computed data structures without any real-time update possibilities. Allowing for dynamic geometry and moving light sources exacerbates this situation.

In this paper, we set out to solve the global illumination exchange between surfaces in real time, also considering dynamic geometry and moving light sources. In particular, our contributions are as follows:

- A real-time single-bounce global illumination algorithm with higher quality and better temporal stability than state-of-the-art techniques from academic and commercial game engines. Our algorithm achieves those results by adopting a voxelized approach and performing a lazy and hybrid irradiance evaluation through ray tracing (only for voxels directly visible from the camera). It reduces the cost of ray tracing by following a divide-and-win approach. It also implements data structures that scale linearly and seamlessly with the number of voxels, allowing a better voxel structure utilization. It is capable of updating the visibility of objects in complex scenes, including dynamic objects, leading to renderings that closely resemble ground truth with better irradiance gathering and temporal stability.
- An efficient algorithm capable of higher quality and performance with a lower memory footprint and voxel resolution than

Figure 1: *Two frames (clipped, to the right) from a Full HD animation (one frame on the left). They are computed with our real-time global illumination algorithm. The images show a large dragon moving through a static city. We can observe changes in the colour bleeding from the nearby geometry on the dragon, and on the darker (respectively brighter) buildings from the street corner when the dragon is closer to (respectively further from) the back gate in shadow.*

academic techniques like *Voxel Cone Tracing* using a *Sparse Voxel Octree* implementation.

- An improvement for voxel-based global illumination techniques preventing light leaks from dynamic objects onto static ones by storing irradiance in separate textures. Our work also allows for an efficient main rendering pass, where each fragment is evaluated only once and requires a small, fixed number of three texture samples for irradiance computation.

Traditional and state-of-the-art techniques in the video game industry cannot achieve our higher levels of quality, including higher frequencies unless they use a brute force approach requiring massive resources. Our solutions form the first steps in a flexible system that we hope to generalize to full global illumination, including multiple light sources, multiple light bounces and arbitrary materials.

The paper is organized as follows. After reviewing significant related work in Section 2, we give an overview of our technique in Section 3, before going into its many details in Section 4. Then, we follow with an analysis of its results in Section 5 and comparisons of key features with other solutions, with support for potential extensions in Section 6. We also cover our current limitations and point to solutions in Section 7. Finally, we revisit our contributions and conclude in Section 8.

2. Previous Work

Over the years, many efforts have been devoted to the development of real-time global illumination techniques. Although many successful ideas have been investigated, the final goal of real-time, dynamic global illumination remains elusive because of its high computational burden and existing graphics computing hardware limitations.

Among the first proposals, Virtual Point Lights (VPLs) have had a significant impact. The original work of Keller [Kel97] proposes to use shadow maps to compute the visibility from VPLs at the light source position. Graphics hardware renders an image with shadows for each particle used as a VPL source, and the resulting image is composited in an accumulation buffer. This line of research has continued with the work by Greger *et al.* [GSHG98], introducing Irradiance Volumes, a technique that pre-computes irradiance in a two-level grid, later used in real-time calculations. Sloan *et al.* [SKS02] develop a technique called Pre-computed Radiance Transfer (PRT), that, after a preprocessing stage, produces shadows, inter-reflections and diffuse-to-glossy materials for dynamic environments. Dachsbacher and Stamminger [DS05] create reflective shadow maps (RSMs), that allow indirect illumination effects without taking into account occlusions to achieve a cheaper and more scalable technique. Dachsbacher *et al.* [DSDD07] circumvent these occlusions with *anti-radiance*, a technique that shoots *negative* irradiance into the scene to remove the effect of dynamic occluders since a last update. Kaplanyan and Dachsbacher [KD10] extend the original Irradiance Volumes with Light Propagation Volumes, where spherical harmonics store the spatial and angular distribution of light in the scene. Their main advantages are that they do not require any pre-computation and that accounting for occlusion and participating media are included.

Screen-space techniques have more recently offered a popular avenue for real-time global illumination. Nalbach *et al.* [NRS14] build a surfel-based representation of the scene, and then splat light samples onto a multi-resolution framebuffer, allowing one-bounce indirect illumination for dynamic scenes. Mara *et al.* [MMNL16] use G-buffers from *deferred-shading* techniques to perform depth peeling combined with multi-bounce global illumination, distributed over multiple frames to generate global illumination effects. Kol *et al.* [KBLE18] propose to render the scene multiple times with a combined scene-and-view hierarchical representation. Concurrently traversing this data structure, combined with sharing renderings among views, allows for sublinear performance for scene complexity and the number of scene views. Majercik *et al.* [MGNM19] extend Irradiance Volumes by computing global illumination for dynamic scenes with different materials. Other techniques, such as PRT, add new insights into efficient storage systems for this kind of illumination. For instance, Currius *et al.* [CDAS20] use convolutional neural networks to estimate light-field values through spherical Gaussians, but only for static scenes. Also, techniques such as light probes [MMNL17, MGNM19], as used in Unreal Engine's RTXGI [MMSM21], are efficient irradiance-field-with-visibility representations, which are quite popular in current video game production environments. More recently, Wright *et al.* [WNK22] have implemented Lumen, which has become state-of-the-art for global illumination in the video game industry.

Voxelization of scene geometry is central to our approach, and it has been supported by several contributions. Wang and Kaufman [WK93] are the first to propose an efficient 3D voxelization algorithm. Through an analytical 3D anti-aliasing technique, it generates gap-free voxel models. Beckhaus *et al.* [BWS02] adapt it to the GPU, and Zhang *et al.* [ZCEP07] add GPU-conservative voxelization to improve performance and memory footprint. The efficient and robust algorithm for GPU voxelization by Schwarz and Seidel [SS10] allows for thinner gap-free surfaces. The seminal work of Crassin and Green [CG12] builds a Sparse Voxel Octree from a voxelized volume, providing acceleration structures built and entirely exploited on the GPU. Heitz and Neyret [HN12] use an enriched voxel representation of detailed surfaces to improve smooth transitions between levels of detail, local illumination, occlusion and anti-aliasing. Vicini *et al.* [VJK21] use voxels to represent both opaque and unstructured geometric aggregates for volumetric scene representations; they can generate approximate higher-quality levels of detail.

In general, for global illumination, long pre-computation times have always been a major drawback [JKG16, SL17, KTHS06]. Voxel-based techniques, however efficient in their calculations for global illumination, are no exception. For instance, the work by Crassin *et al.* [CNS*11] requires complex GPU data structures, in combination with mipmapping, to reduce costs of the irradiance computations. In another example, the fast voxel path tracer of Thiedemann *et al.* [THGM11] within a voxelized volume suffers from back projecting the hit points into a RSM [DS05] for irradiance computations. These techniques need to build additional data structures from the voxelized scene, including costly scene post-processing techniques. As examples of extensions to the previously mentioned papers, Sugihara *et al.* [SRS14] extend Voxel Cone Tracing (VCT) using voxel information just for visibility estimation, back-projecting into layered RSMs, but with the limitation of one RSM per light source. Chen and Chien [CC16] improve on both techniques by considering only the set of lit voxels in a scene for the irradiance computations. This precludes the need for one RSM per light source, which may scale poorly in terms of performance and memory but can simultaneously solve the many-light problem for the voxel-based global illumination techniques. Papaioannou [Pap11] also combines Irradiance Volumes and voxel approaches, where RSMs approximate irradiance values through spherical harmonics in a uniform grid, considerably increasing performance by avoiding shadow construction. Recently, Ayerbe and Patow [AP22] also use a voxel-based technique to compute global illumination, but their use of complex-to-update data structures precludes any use of dynamic geometry and lighting. We note that the technique that we present in this paper can achieve higher levels of quality, including higher frequencies, than can traditional techniques such as VCT and light probe-based techniques, unless they resort to a brute-force approach that would require massive resources.

ReSTIR and its family of related techniques form one of the most promising approaches for the objective of real-time global illumination. Briefly, ReSTIR [BWP*20] is a technique devised to help render scenes with many light sources but making the rendering much less noisy than basic path tracing algorithms. This results from continuously finding and updating the most contributing light for a pixel based on its neighbour pixels and its illumination computed in previous (temporal) frames. Ouyang *et al.* [OLK*21] extend this concept with an effective path-sampling algorithm for indirect illumination, re-sampling multi-bounce indirect light paths obtained by path tracing. Their technique has been generalized [LKB*22] to introduce resampled importance sampling (GRIS), allowing RIS on correlated samples, with unknown PDFs taken from various domains, providing practical guidelines for algorithms, and enabling advanced path re-use between pixels via complex shift mappings. Recent work extends ReSTIR to world-space sample re-use [Boi21, BJW21].

Our technique falls within the family of Irradiance Volume techniques, as we voxelize the scene and consider each generated voxel as a small 'cubemap', computing irradiance by casting 128 rays from each voxel face. This avoids any manual scene setup of light probes as required in general by techniques from this family, and decouples the camera viewpoint from the scene geometry and screen resolution. Our technique supports a dynamic light source and any number of dynamic scene objects, achieving a lower ray-tracing cost through a divide-and-win approach, where we update each voxel face's visibility by casting rays through only dynamic scene objects.

3. Overview

Our technique consists of several steps, executed in sequence. First, two initial steps are performed *only once* per scene:

- Voxelization of static scene geometry: This first voxelization builds basic structures that will not change and that are used to cache and compute irradiance for static scene objects. See Figure 2(a,b).
- Static voxel visibility: We cast 128 rays per face of static voxels through only the static geometry, caching information

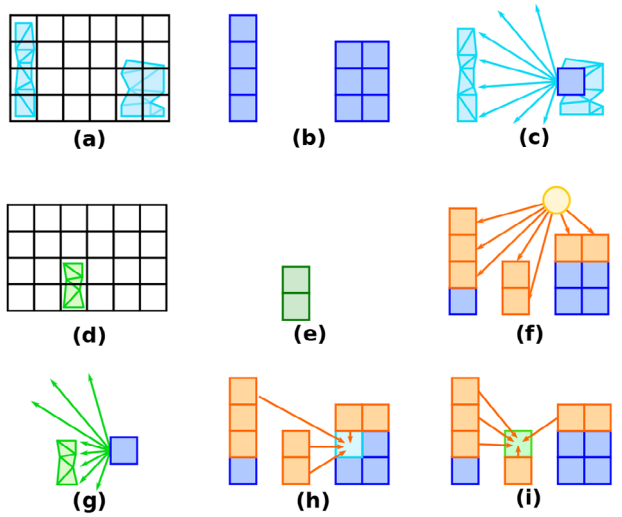


Figure 2: *Pipeline followed in our technique: (a, b) voxelization of the static geometry, (c) per-static-voxel ray-tracing visibility caching for the static geometry, (d, e) voxelization of the dynamic geometry, (f) direct illumination to all voxels, (g) per-static-voxel ray-tracing visibility caching for the dynamic geometry, (h) static voxel irradiance gathering from lit static and dynamic voxels (bright blue), (i) dynamic voxel irradiance gathering from lit static and dynamic voxels (bright green).*

for each ray from the intersections to further accelerate light bounce computations, contributing to improving irradiance gathering and performance. The reflectance of each static voxel is stored in a 3D texture. See Figure 2(c).

Then, the next steps are performed sequentially at *each scene update*:

Voxelization of dynamic scene geometry: This step is equivalent to the first voxelization step on static scene geometry but applied only to the dynamic geometry. It includes the creation of a 3D texture to store reflectance. See Figure 2(d,e). We perform a visibility test for each voxel (static or dynamic), flagging the voxels visible from a light source. See Figure 2(f).

Compute camera visible voxels: We generate a G-Buffer of the scene geometry and compute pixel-perfect ray-traced shadows. We determine from the G-Buffer all voxels visible in the camera. Processing only a subset of voxels allows a much higher performance, as focusing on this subset enables our technique to improve each voxel's visibility information, leading to higher-quality results. We also consider a 5^3 neighbourhood around each dynamic voxel for irradiance computations, allowing our work to achieve high temporal stability.

Update static voxel visibility: For each static voxel visible in the camera, we cast 128 rays per voxel face through only the dynamic geometry to update its visibility. We use the same 128 directions as in the initial stage. For each ray direction, we save whether the closest geometry is the initial static one or a new one due to dynamic scene objects. See Figure 2(g). This divide-and-conquer approach contributes to an increased performance, while our compact data structures allow a lower memory footprint.

Light bounce for static voxels: We first process static voxels, gathering irradiance at the level of a voxel face. We only need to know, for each voxel face and ray direction, whether to pick the static or dynamic visible geometry information computed in the previous step. See Figure 2(h).

Light bounce for dynamic voxels: We follow the same approach as for the light bounce of static voxels, but with dynamic voxels. See Figure 2(i). By storing irradiance in separate 3D textures in the last two steps, we avoid light leaks from dynamic onto static objects, which is an improvement in voxel-based techniques.

Smoothing irradiance and padding: After all light bounce computations, we perform filtering for both the static and the dynamic voxel computed irradiances. Then, we compute the average irradiance value stored as 'padding' for all empty voxels having at least one neighbouring static voxel to avoid shadow leaks.

Main rendering step: Finally, we use the G-buffer information to perform the final pass, sampling from up to three of the six 3D textures where we have stored irradiance, making our approach view-independent. This highly efficient main rendering pass represents a contribution compared to other voxel-based techniques.

4. Method

4.1. Voxelization and static visibility

We perform boundary voxelization with the method by Takeshige [Tak15]. Each primitive (*e.g.* a triangle) from each static object in the scene is rendered with depth test and face culling deactivated, with an orthographic camera aligned to one of the main axis directions (x, y or z), selecting the one that maximizes the primitive's face area. This choice prevents the primitive from being drawn with a steep slope when viewed from the viewport; this is the main reason for gaps in the voxelized results.

A voxel V of indices (V_x, V_y, V_z) in a grid of resolution $N \times N \times N$ is uniquely encoded in 1D as

$$Encode(V) = N^2 V_x + N V_y + V_z. \quad (1)$$

Figure 3 illustrates the relations between our different buffers.

We store in the *static voxel buffer* only the voxels generated from the voxelization of the static geometry, *i.e.* the *static voxels*. For each static voxel, we write its index encoded from Equation (1). We write the index in an N^3 3D texture where each static voxel has been stored in the static voxel buffer, allowing fast access to the voxel information. Afterward, we compute and cache visibility for each generated static voxel in the static voxel buffer, storing the results in a separate buffer (the *static visibility buffer*).

For each static voxel face, we cast 128 rays from the face centre towards the static scene geometry. This set of rays is always the same, where each ray direction has been slightly jittered. The set of rays is transformed in the local frame of reference for each voxel face. Then, for each intersection of a ray with the static scene

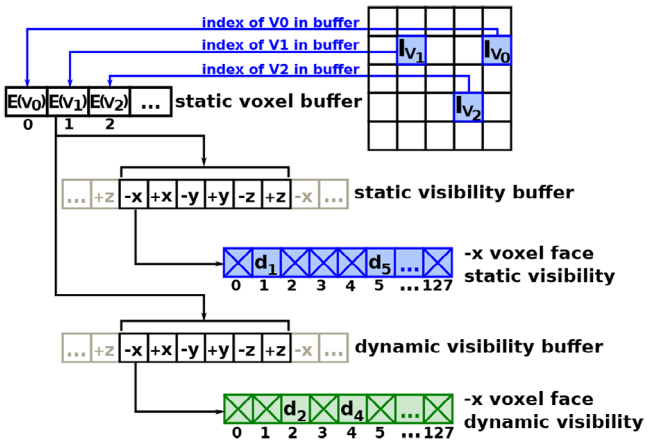


Figure 3: *Voxel visibility caching data structures. After voxelization of the static geometry, we store only the static voxels generated (here, three voxels V_i with i in $\{0, 1, 2\}$) in the* static voxel buffer*, encoding the indices of each voxel using Equation (1) as $E(V_i)$. We store in an N^3 3D texture the 1D index where each static voxel has been stored (here, three voxels I_{V_i}) in the static voxel buffer for quick access to the information of a specific voxel. Each static voxel has visibility information for static and dynamic geometry in the per-voxel face visibility buffers, storing distances to each intersection in the static (here, distances d_1 and d_5 for rays of face $-x$ at indices 1 and 5) and dynamic geometry (here, d_2, d_4 for rays of face $-x$ at indices 2 and 4), caching for each voxel face ($\pm x$, $\pm y$, $\pm z$) the results of initially casting 128 rays through the static geometry and the same 128 rays for each scene update, but only for the dynamic geometry.*

geometry, we store the distance from the ray origin in scene coordinates with 16-bit precision (2 bytes) and the x and y coordinates of the normal at the intersection with 7-bit precision, plus one bit for the sign of the local normal z (2 bytes in total). Thus, each voxel requires a total of 6 faces $\times$ 128 rays $\times$ 4 bytes $=$ 3 KB. As mentioned, this compact data structure enables our work to have a low memory footprint, enabling us to perform fast visibility updates of each static voxel by ray tracing only dynamic scene objects (as explained later), achieving a high-quality irradiance gathering and increased performance.

We also store the reflectance of the static voxels in an N^3 3D texture, following the implementation of Crassin *et al.* [CNS*11]. The average reflectance value of the contributions of all fragments generated at a specific voxel is computed with a spin-lock at the shader level. Our technique proceeds, at the byte level, to test whether a specific static voxel is empty or occupied. For this purpose, a byte is set in a small buffer, at the index given by encoding the static voxel 3D indices using Equation (1). A 128^3 voxelized scene only requires a buffer of size $128^3 = 2$ MB to map all possible values.

4.2. Dynamic objects

In the presence of dynamic parts in a 3D scene, we follow several steps to update the global illumination. As we did for static voxels, we follow the method by Takeshige [Tak15] to perform a boundary voxelization of each dynamic object, avoiding voxel gaps in the result. For each generated dynamic voxel, we store its reflectance in a 3D texture, and its encoded index (using Equation 1) in a voxel buffer (the *dynamic voxel buffer*), separate from the one used for the static objects. We also set a byte in a small buffer at the same index as the encoded value of the dynamic voxel coordinates, to test whether specific voxel coordinates are empty or occupied by a dynamic voxel.

4.3. Lit voxels

As part of the light bounce information required, we need to determine whether each voxel is visible (*i.e.* lit) from a light source. For each voxel, because it forms a volume, we consider a grid of 2^3 points inside the voxel volume. For each point, a ray query toward a light source is performed against the scene geometry. If the number of visible points is above a threshold value of four (four was experimentally determined to give the best visual results), we consider the voxel as being lit. We work with a buffer at the byte level to store information on whether each voxel is lit. We flag with value 1 a byte at the index given by encoding the coordinates of the voxel, using Equation (1). With this information, our technique avoids casting shadow rays when performing light bounce computations, which contributes to achieving higher performance.

4.4. Camera visible voxels

We follow a deferred shading approach to rendering the scene, storing reflectance, world position and normal direction during a G-buffer pass. We also implement pixel-perfect shadows through ray-tracing queries against the scene geometry, storing the result of the shadow ray in a free channel in the G-buffer. We compute light bounce information only for static and dynamic voxels visible from the camera. As already mentioned, processing only a reduced number of voxels greatly increases performance and allows our technique to spend more computations on each processed voxel. This enriches the per-voxel visibility information, allowing high-quality results with fewer voxels. For this purpose, we work at the byte level with two buffers of size equal to the total number of voxels in the voxelization. One buffer flags static voxels visible from the camera, and the other flags dynamic voxels. During the shadow computation process, for each shadow ray query against the scene geometry, we compute the voxel coordinates of the ray origin obtained from the G-buffer and flag the byte at the index given by encoding the voxel coordinates (using Equation 1) in the corresponding buffer, depending on whether it belongs to a static or a dynamic object. For this reason, a voxel can be flagged as both static and dynamic.

To guarantee that our technique offers correct irradiance values for any dynamic object regardless of its shape and trajectory, including instant translations, we flag as visible all 5^3 voxels in the neighbourhood of each dynamic voxel, regardless of whether other dynamic objects are detected. This approach represents one of the contributions from our technique, achieving temporal stability for dynamic objects, as it ensures that irradiance values will be ready for linear interpolation for those fragments generated from the geometry of dynamic scene objects, without the risk of introducing darker values during interpolation caused by neighbouring voxel

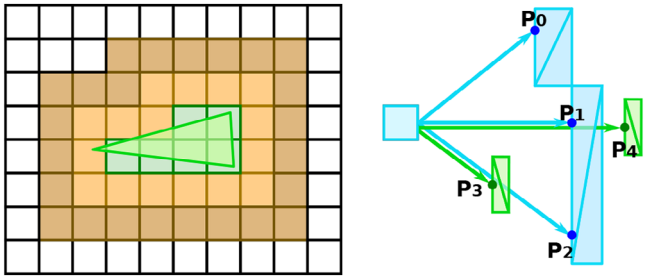


Figure 4: *Left: During a scene update, dynamic scene geometry (bright green triangles) is voxelized, generating a set of dynamic voxels (in green). We add a layer of voxels around each dynamic voxel thus generated to maintain more uniform irradiance interpolation values (light brown) for those fragments closer to the border of the generated initial dynamic voxels. Given that the geometry does not always generate voxels due to primitives not covering the centre of each voxel (green geometry in light brown voxels), an extra layer of padding is added (dark brown). Right: After the initial static geometry, visibility caching per voxel face, the voxel in a bright blue comes across intersection points P_0, P_1 and P_2 with static scene geometry. When tracing rays in the same directions only through dynamic scene geometry during a scene update, updating visibility, P_2 is discarded in favour of P_3, which is closer to the ray origin, while point P_1 is kept for being closer.*

irradiances not being computed and having a value of 0.0. See Figure 4 (left) for an example. This incurs an extra cost when computing irradiance for dynamic scene objects, later on, but it has proven to be the best option to reduce temporal aliasing for dynamic scene objects. In our experience, a 3^3 neighbourhood works correctly for most scenes. Also, note that this process only adds an extra N^2 complexity for an N^3 voxelization resolution (for simplicity in this text, we use the same resolution in all three dimensions) since the criteria to flag new dynamic voxels for irradiance computation happen at the level of surfaces, not volumes. See Figure 4 (left). We store the light bounce irradiance results in 3D textures, interpolating between voxels to compute irradiance arriving at a fragment. Due to the use of linear interpolation, fragments closer to the border of a static voxel will sample darker irradiance values. To reduce this artefact, for each empty voxel, we compute the average irradiance of all occupied neighbouring static voxels. We further improve memory access with a tiled approach of 2^3 voxels into a tile, instead of processing voxels one by one. In this step, we flag each occupied static voxel, and every tile around it (more details in Section 4.7). We also follow a tiled approach for irradiance filtering, flagging each tile having at least one static occupied voxel. Finally, in a *compute* step after the camera ray dispatch (described at the beginning of this section), the flagged static and dynamic voxels are stored in separate buffers, as well as the indices of the flagged tiles for padding and irradiance filtering for later processing.

4.5. Voxel visibility update

Before we perform light bounce computations for the camera's visible static and dynamic voxels, we need to update their visibility. In the case of static voxels, for each voxel face, we cast 128 rays in the same directions as in the initial static visibility step (Section 4.1). However, we cast the rays only through the dynamic scene geometry, allowing faster computations than if done with the whole scene geometry. This divide-and-conquer approach allows our technique to achieve a better performance, while using ray tracing enables a better irradiance gathering, leading to higher quality results. If there is a ray intersection with dynamic geometry and the dynamic voxel at the coordinates of the intersection is lit, we check whether there was a cached static voxel intersection for that same ray direction. In case there was, we compare the distances traversed by the two rays in scene coordinates, and if the intersection with the dynamic geometry is closer, we consider that the ray *sees* a dynamic scene object. Otherwise, the ray sees the same initial static object. See Figure 4 (right). We store a bitmask for each ray of a voxel face to specify whether to select the dynamic or the static voxel for the subsequent stage of light bounce computations. We also keep the initial cached static voxel visibility information in case a dynamic object moves *away* from a ray direction, at the expense of a reasonable memory footprint. For dynamic voxels, we cast 128 rays for each voxel face in an acceleration structure containing the whole scene geometry with both static and dynamic objects. The information of the intersection is stored in a visibility buffer (the *dynamic visibility buffer*).

4.6. Light bounce

From Section 4.4, we have a buffer with the camera's visible static voxels and another buffer with the camera's visible dynamic voxels. The next step in our pipeline is to compute irradiance arriving at each voxel face of each camera visible voxel. By default, our technique computes irradiances for all six voxel faces to guarantee they can be correctly interpolated for voxels containing complex geometries. It works well also for simpler geometries when normal directions differ greatly from the geometrical normals. We start performing light bounce computations for static voxels, where for each voxel face, we dispatch a workgroup of 128 threads, with each thread processing a direction from the initial 128 directions. We use a bitmask from the Section 4.5 to retrieve the correct voxel information (the dynamic visible voxel closer than the initially visible static voxel or the visible static voxel itself). If the voxel analysed is lit, we compute the irradiance arriving from the light source to that voxel, and the irradiance coming from that voxel to the one being processed using the differential area form-factor formula [Sri16]. This allows static voxels to compute irradiance from other voxels generated from both static and dynamic geometries. The irradiance computed by each thread is added following a parallel reduction approach. We follow the same approach to compute irradiance for the camera's visible dynamic voxels.

We described in Section 4.5 how updates of the dynamic voxel visibility trace rays in an acceleration structure containing all the scene objects. This allows dynamic voxels to compute irradiance from other voxels generated from static and dynamic geometries. In both cases, the thread number 0 of each workgroup stores the accumulated irradiance in the corresponding texture. In our experiments, we have identified several incompatibilities when merging the static and dynamic irradiance information linked to the same

voxel, mostly due to the simplified operations used to compute dynamic irradiance in the main rendering step. For that reason, we use two separate sets of six 3D textures with 16-bit precision to store each type of voxel irradiance (static or dynamic), storing in each texture the irradiance gathered per voxel face (six faces per voxel, and so six 3D textures).

Light leaks are a common issue in voxel-based global illumination techniques. Our technique effectively adds a new contribution to this family of techniques by avoiding leaks from dynamic onto static objects because we split irradiance for both kind of scene objects. Due to the small number of samples used per voxel face, we apply a simple temporal filter at each frame update to increase the stability of irradiance computations. We follow Refs. [NSL*07, PSK*16] with $E_t = \alpha E_t + (1-\alpha)E_{t-1}$, where E_t is the current update irradiance and E_{t-1} is the accumulated irradiance from previous updates. We use $\alpha = 0.03$. To guarantee that irradiance values in dynamic voxels are updated in case no dynamic scene objects intersect with them, we keep a cool-down factor for each voxel, fading completely its irradiance value after 0.3 s from the last update.

4.7. Smoothing irradiance and padding

The last step of the light bounce computations consists of filtering both the static and dynamic voxel irradiances and then computing the average irradiance value stored as 'padding' for all empty voxels having at least one neighbouring static voxel. As mentioned in Section 4.4, to minimize texture access, we divide the voxelization volume in tiles of two voxels per dimension, leading for each tile to a total of 2^3 voxels. We build one buffer for filtering with the index of each tile that has static voxels. We build another buffer for padding, with the index of each tile that contains empty voxels with at least one static voxel as a neighbour. A tile index is computed in the same way as a voxel index.

For filtering computations, for each tile, we dispatch six workgroups (one per voxel face), each made of eight threads. First, each thread tests which voxels in the tile are occupied by a static voxel. Then, the threads store in a shared variable the coordinates of the voxels from which they need to load irradiance, to perform filtering. We apply a 3^3 Gaussian filter to each voxel, with each tile loading up to $4^3 = 64$ values. In the worst performance case of a per-voxel approach, all eight voxels in a tile and all 26 neighbouring voxels would be occupied, meaning there would be $3^3 \times 8 = 216$ texture loads. Our tiled approach reduces that number to $4^3 = 64$, thanks to sharing the loaded values between the voxels in the tile, saving $216 - 64 = 152$ texture load operations, accounting for 70% of the total operations. This approach is another factor that allows our work to achieve a higher performance.

The padding process is similar, where the texture load operations are done for each empty voxel in the tile with any neighbour occupied by a static voxel, storing the average irradiance value in the originally empty voxel. See Figure 5 for a 2D illustration of the tiled filtering and padding computations. We do not perform padding for the dynamic voxels as we compute the irradiance of all neighbouring voxels of any occupied dynamic voxel, as mentioned in Section 4.4 and shown in Figure 4 (left). The Gaussian filtering of dynamic voxels also has a kernel of size 3^3.

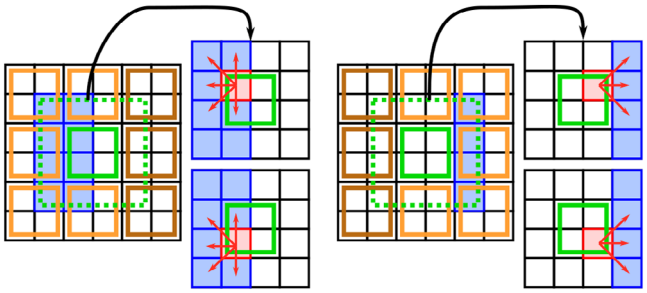


Figure 5: *Left: For irradiance Gaussian filtering, we mark the tiles that have at least one static voxel (bright orange and green squares on the left and central columns). We use a 3^3 kernel. As can be seen in the tile being processed (green) and its surroundings (dotted green), for each occupied static voxel, we load the irradiance of all neighbouring static voxels (red arrows). In this 2D example, the top left tile needs to load irradiance from six texture coordinates, four of which are re-used by the bottom left static voxel. Right: The padding process is similar. We mark the tiles with empty voxels having at least one neighbouring static voxel (bright orange and green squares in the central and right columns). For each empty voxel in the tile being processed (green), we load the irradiance of all neighbouring static voxels (red arrows) found in the tile and its surroundings (dotted green), to compute the average value. The top right tile voxel in 2D needs to load three irradiance values, two of them re-used later by the bottom right tile voxel.*

4.8. Main rendering

In the main rendering step of the scene, we sample from the corresponding irradiance texture set depending on whether we are drawing a static or a dynamic scene object, interpolating between the per-voxel face irradiance textures to match the contribution for the fragment's normal. Our technique supports normal mapping and only needs three texture samples with no mipmap interpolation per pixel, plus one sample for the fragment's material reflectance in a post-processing pass, which re-uses the information from the G-buffer. As noted, this simple and efficient main rendering pass represents a substantial contribution to voxel-based global illumination techniques. The padding added for the static voxels guarantees that the geometry closer to voxel boundaries will not appear darker when using linear interpolation for the sampling from irradiance textures. As mentioned in Section 4.4 and shown in Figure 4 (left), computing irradiance for all neighbours of each dynamic voxel allows for a smoother transition for dynamic geometries, decoupling from the geometry position within the voxel. This offers a smoother transition of computed irradiance as dynamic geometries move between voxels, greatly reducing temporal aliasing. See Figure 6 and the accompanying video material. The irradiance for any voxel is always computed at its geometric centre. The last step before displaying on screen is FXAA anti-aliasing.

5. Results

We developed our technique in our own C++ framework using the Vulkan API for graphics, compute and ray-tracing shaders. All measurements were done with an NVIDIA GeForce RTX 2060 GPU

Figure 6: *Details from a different viewpoint for the* Rocks *scene with a voxelization resolution of* 128^3. *Left: Using a* 5^3 *neighbourhood for dynamic voxels, running at 124 fps with 721 dynamic visible voxels. Centre: Scene without any neighbourhood for dynamic voxels, running at 137 fps with 61 dynamic visible voxels. Right: Difference image between the two images using NVIDIA's* FLIP *[ANAM*20] tool.*

with 6GB of VRAM on a computer with a Ryzen 7 3800XT AMD CPU and 16GB of RAM. When measuring performance, we took five different captures for each scene and voxelization resolution with NVIDIA NSight, recording the time required by each GPU task in the frame and computing the average and standard deviation values. We did all our tests at full HD resolution (1920 × 1080).

Upon acceptance, the complete source code of our technique will be available for download on *GitHub* at https://github.com/AlejandroC1983/dvbgi.

5.1. Dynamic update

We show in Table 1 the numbers of visible static and dynamic voxels (top) and the breakdown of timings in milliseconds (bottom) for the different steps that our technique requires to perform a scene update for the scenes *Rocks* and *Factory*. Currently, an update involves changes in either the position or direction of a light source or a transformation of a dynamic scene object, performing the same type of computations in both cases. We can see how the timings for dynamic scenes range from 6.8 to 19.6 ms. If we account only for the steps related to the update (*i.e.* the steps for the first four rows of timings in Table 1), our technique's computations range from 3.0 ms for *Rocks* and 2.3 ms for *Factory*, both at a voxelization resolution of 32^3; up to 9.7 ms for *Rocks* and 14.9 ms for *Factory* at a voxelization resolution of 128^3. The timings for the step *Main scene + FXAA* include the main rendering step, which has a maximum value of 0.38 ms for the *Rocks* scene and a maximum value of 0.4 ms for the *Factory* scene. These results illustrate our contribution to a fast main rendering step, which only requires three texture samples for irradiance computation.

In the *Factory* scene, when the voxelization resolution is doubled in all three dimensions, the total number of static and dynamic voxels processed increases by an approximate average factor of 3.6× and 2.3×, respectively, with computation times increasing with an average factor of about 1.5×. As the voxelization resolution increases, the *Static voxel update* (*i.e.* static visibility update, light bounce, Gaussian filtering and padding) and *Dynamic voxel update* steps (*i.e.* dynamic visibility update, light bounce and Gaussian filtering) take up most of the time in most of the cases. For the scene *Rocks*, the average cost to process static voxels is 0.000927 ms, while the average cost to process dynamic voxels is 0.001626 ms. For the scene *Factory*, the average cost to process static voxels is 0.000481 ms, while the average cost to process dynamic voxels is 0.000821 ms, the ray-tracing visibility update step being the main reason for the extra cost, computed against the whole scene geometry instead of only the dynamic scene geometry, as shown in Table 2.

Our technique is dependent on the number of voxels to process. We see how lower voxelization resolutions allow for a lower quality

Table 1: *Average and standard deviation values of static and dynamic visible voxels processed, and breakdown of timings (in milliseconds) of the different steps followed during each scene update. Evaluations were done for two scenes with dynamic objects and a light source, for three voxelization resolutions. The scene* Rocks *(1.1M triangles) used for measurement is taken from a similar viewpoint as in Figure 7, with a dynamic vehicle and a skinned mesh character. The scene* Factory *(2.3M triangles) is from a viewpoint as in Figure 8, with two large dynamic objects and eight skinned mesh characters. Samples of each scene at each voxelization resolution can be found in the video sequences in the accompanying material.*

Scene	*Rocks*			*Factory*		
Voxelization resolution	32^3	64^3	128^3	32^3	64^3	128^3
Visible static voxels	404 ± 2	1562 ± 4	5797 ± 14	976 ± 19	3241 ± 51	11183 ± 186
Visible dynamic voxels	567 ± 27	1017 ± 22	2229 ± 116	1475 ± 56	3644 ± 60	9981 ± 60
Lit voxels	0.1 ± 0.0	0.2 ± 0.0	0.5 ± 0.0	0.1 ± 0.0	0.2 ± 0.0	0.4 ± 0.01
Camera visible voxels	0.8 ± 0.0	0.9 ± 0.0	1.3 ± 0.03	0.6 ± 0.0	0.6 ± 0.0	0.9 ± 0.01
Static voxel update	0.9 ± 0.04	1.8 ± 0.11	4.5 ± 0.23	0.6 ± 0.02	1.6 ± 0.07	5.2 ± 0.21
Dynamic voxel update	1.2 ± 0.10	1.6 ± 0.04	3.4 ± 0.19	1.0 ± 0.02	3.0 ± 0.08	8.4 ± 0.43
Ray-traced shadows	2.3 ± 0.01	2.0 ± 0.20	1.5 ± 0.01	0.8 ± 0.07	0.8 ± 0.06	1.0 ± 0.15
Main scene + FXAA	2.3 ± 0.13	2.4 ± 0.01	2.5 ± 0.16	3.7 ± 0.01	3.8 ± 0.16	3.7 ± 0.01
Total	7.6 ± 0.19	8.9 ± 0.24	13.7 ± 0.34	6.8 ± 0.07	10.0 ± 0.19	19.6 ± 0.50

Table 2: *Average and standard deviation timings (in milliseconds) for the static voxel visibility update for three voxelization resolutions and two scenes, performing ray tracing only through the dynamic geometry, or through all the static and dynamic geometries. The scenes are as described in the caption of Table 1.*

Scene	*Rocks*			*Factory*		
Voxelization resolution	32^3	64^3	128^3	32^3	64^3	128^3
Dynamic geometry	0.5 ± 0.04	0.7 ± 0.04	1.4 ± 0.05	0.3 ± 0.02	0.7 ± 0.01	2.2 ± 0.18
Full scene geometry	1.9 ± 0.16	2.4 ± 0.01	8.2 ± 0.11	0.8 ± 0.12	2.9 ± 0.11	11.5 ± 0.14

Table 3: *Breakdown of average and standard deviation timings (in milliseconds) of the update times for static and dynamic voxels for three voxelization resolutions and two scenes. Half of the static objects are randomly set as dynamic during loading, in addition to the already existing dynamic objects. The scenes are as described in the caption of Table 1.*

Scene	*Rocks*			*Factory*		
Voxelization resolution	32^3	64^3	128^3	32^3	64^3	128^3
Visible static voxels	257 ± 1	866 ± 1	2972 ± 1	521 ± 4	1546 ± 38	5198 ± 88
Visible dynamic voxels	1355 ± 6	4329 ± 25	14182 ± 70	3280 ± 12	10947 ± 178	36016 ± 254
Static voxel update	0.9 ± 0.1	2.0 ± 0.3	3.9 ± 0.3	1.1 ± 0.09	2.5 ± 0.17	6.2 ± 0.38
Dynamic voxel update	3.9 ± 0.21	10.2 ± 0.29	31.1 ± 1.46	7.7 ± 0.16	26.5 ± 0.98	53.2 ± 2.59
Other	5.5 ± 0.12	5.9 ± 0.54	6.7 ± 0.27	5.9 ± 0.29	6.7 ± 0.38	7.3 ± 0.48
Total	10.3 ± 0.26	18.1 ± 0.68	41.7 ± 1.51	14.7 ± 0.34	35.7 ± 1.06	66.7 ± 2.66

global illumination at a lower cost. In comparison, higher voxelization resolutions allow for higher quality global illumination with better-defined indirect shadows, at a cost that grows, however, at a lower rate than the number of voxels processed.

Table 2 shows how the timings required to ray trace the full scene geometry when performing the static voxel visibility update (within the *Static voxel process* mentioned in Table 1), range between 2.7× and 5.9× longer than the time needed to ray trace only the dynamic scene geometry (*e.g.* 8.2 vs. 1.4 ms for 128^3 for scene *Rocks*), compensating for the extra memory required to initially store the static voxel visibility for each static voxel face.

We show in Table 3 measurements when randomly setting half of the static objects as dynamic at loading, in addition to the original dynamic objects. By doing so, we force the dynamic voxelization and dynamic voxel processing steps to carry a larger weight of the global illumination updates. To simplify the comparison with Table 1, we group all non-relevant steps (*Lit voxels*, *Camera visible voxels*, *Ray-traced shadows* and *Main scene + FXAA*) from Table 1 in the *Other* row of Table 3, since their values are similar to those in Table 1. We can see that as the number of dynamic objects increases, the total number of dynamic voxels to process grows. Except for scene *Rocks* at voxelization resolution 32^3, the cost to process each static voxel is always lower than the cost to process each dynamic voxel, with an average value of 0.0016 ms per static voxel versus 0.0022 ms per dynamic voxel for scene *Rocks*, and 0.0013 ms per static voxel versus 0.0017 ms per dynamic voxel for scene *Factory*. We observe a similar cost as in Table 1, where only a few scene objects are dynamic, showing how our method scales proportionally as the number of dynamic objects increases.

Table 4: *Ray-tracing budget breakdown per frame in million rays for static voxel visibility update, dynamic voxel visibility update and lit voxels computations for three voxelization resolutions and two scenes as described in Table 1, and the cost per pixel for different rendering resolutions: 1K (1920 × 1080), 2K (3840 × 2160) and 4K (7680 × 4320).*

Scene	*Rocks*			*Factory*		
Voxelization resolution	32^3	64^3	128^3	32^3	64^3	128^3
Visible static voxels	404	1562	5797	976	3241	11,183
Visible dynamic voxels	567	1017	2229	1475	3644	9981
Static voxel update	0.4	1.2	4.7	0.8	2.5	8.6
Dynamic voxel update	0.5	1.0	2.1	1.3	3.3	9.5
Lit voxels	0.03	0.1	0.6	0.02	0.1	0.5
Total	0.93	2.3	7.4	2.12	5.9	18.6
Rays per pixel at 1K	0.4	1.1	3.6	1.0	2.8	9.0
Rays per pixel at 2K	0.1	0.3	0.9	0.3	0.7	2.2
Rays per pixel at 4K	0.03	0.07	0.2	0.06	0.2	0.6

5.2. Ray-tracing budget

Table 4 shows the total cost to update the static voxel visibility, dynamic voxel visibility and lit voxel computations for the scenes in Table 1. We observe how the ray-tracing budget varies between 0.93M rays for the *Rocks* scene at a voxelization resolution of 32^3, to 18.6M rays for the *Factory* scene at a voxelization resolution of 128^3. It is worth noting that, due to our divide-and-win approach in ray tracing, only dynamic scene objects update static voxel visibility, so the ray-tracing cost in milliseconds for the *Static voxel update*

Table 5: *Memory usage for three voxelization resolutions and two scenes (see Table 1). We include the total number of static voxels in each case and adapt the size of the buffers used to store dynamic voxel information to an upper-bound estimate for dynamic voxels.*

Scene	*Rocks*			*Factory*		
Voxelization res.	32^3	64^3	128^3	32^3	64^3	128^3
Static voxels	3779	16,122	70,512	2648	14,866	61,972
Buffer (MB)	32.3	108.6	456.9	28.3	112.4	441.6
Texture (MB)	2.0	16.0	128.0	2.0	16.0	128.0
Total (MB)	34.3	124.6	584.9	30.3	128.4	569.6

case is much lower than ray tracing the whole scene, as can be seen in Table 1. In our technique, the ray-tracing budget ranges between 0.4 and 9.0 rays per pixel for a 1K resolution, between 0.1 and 2.2 rays per pixel for a 2K resolution, and between 0.03 and 0.6 rays per pixel for a 4K resolution. This shows how our technique is not affected by rendering image resolutions due to our view-independent approach to storing irradiance at scene voxels.

5.3. Memory

Table 5 shows the memory usage for the *Rocks* (1.1M triangles) and *Factory* (2.3M triangles) scenes. We can see in this case how the number of static voxels grows with an approximate factor of 4.6× each time the voxelization resolution is doubled in all three dimensions (8×), while the memory usage grows with a proportionality factor of 4.2×, approximately. As the scene voxels are predominantly static, most of the allocated memory stores static voxel-related information. We adapt the size of the buffers to store dynamic voxel information to an upper-bound estimate of the dynamic voxels needed in each case.

5.4. Large dynamic objects

We present in Table 6 measurements (in milliseconds) of each step followed in a scene update, for three different voxelization resolutions, and for each one of the three different scenes shown in Figure 1. With timings for scene updates ranging from 11.4 to 22.2 ms, the times required by our technique (steps from the first four rows on timings presented in Table 6) range from 2.9 ms at a voxelization resolution of 32^3, to 12.6 ms at a voxelization resolution of 128^3. We observe a similar behaviour as in Section 5.1, with a total number of static and dynamic voxels processed increasing by approximate average factors of 3.3× and 2.5×, respectively, as voxelization resolution is doubled in all three dimensions. In contrast, computation times increase at an average factor of 1.35×. Static and dynamic voxel updates take most of the time, with the cost to process dynamic voxels increasing faster due to the extra cost of performing ray tracing on the whole scene geometry. Our technique shows how large dynamic objects have an update cost that grows proportionally to the number of dynamic voxels processed, with a cost to process dynamic voxels ranging between 0.0021 and 0.0026 ms.

5.5. Dynamic voxel neighbourhood

As discussed in Section 4.4, when computing irradiance, we surround each dynamic voxel with a 5^3 neighbourhood, which guarantees irradiance values will be present when interpolating irradiance for fragments generated by the geometry of dynamic scene objects, avoiding any possible darkened values as a result of the interpolation. See Figure 4 (left). As a consequence, dynamic scene objects have temporal stability, avoiding flickering. There is a trade-off between quality and performance in this regard. In Figure 6, we show an animated character in the scene *Rocks*, with a voxelization resolution of 128^3 when using a 5^3 neighbourhood in the leftmost part, running at 124 fps with 721 dynamic voxels. We show the same animated character and pose *without* using a neighbourhood in the central part of the figure, running at 137 fps with 61 dynamic voxels. The differences in lighting between both cases are shown in the rightmost part using NVIDIA's FLIP [ANAM*20] tool. These differences, although small when comparing static images, are the source of noticeable flickering, which can be better appreciated in the video sequences from the accompanying material.

Table 7 shows the number of visible dynamic voxels processed and FPS for two scenes: *Rocks*, with the same viewpoint as in

Table 6: *Average number of visible static and dynamic voxels processed (top) and average timings breakdown in milliseconds (bottom) with a standard deviation of the different steps followed during each scene update, for three different voxelizations for each scene shown in Figure 1. We used a 1.7M triangle version of Amazon's* Bistro *where distant invisible geometries were removed, adding a large dynamic object made of 41K triangles. The measurements from the same viewpoint as in Figure 1 were recorded at five different times for each case, taking GPU task timing values from an NVIDIA NSight capture.*

Scene	*Bistro white dragon*			*Bistro red dragon*			*Bistro blue dragon*		
Voxelization resolution	32^3	64^3	128^3	32^3	64^3	128^3	32^3	64^3	128^3
Visible static voxels	215 ± 4	672 ± 11	2335 ± 37	216 ± 5	696 ± 17	2397 ± 43	223 ± 2	687 ± 19	2330 ± 85
Visible dynamic voxels	523 ± 17	1231 ± 16	3414 ± 36	517 ± 15	1234 ± 23	3443 ± 62	441 ± 27	936 ± 17	2584 ± 61
Lit voxels	0.2 ± 0.0	0.3 ± 0.0	1.0 ± 0.15	0.2 ± 0.0	0.3 ± 0.0	1.0 ± 0.14	0.2 ± 0.0	0.3 ± 0.0	0.9 ± 0.09
Camera visible voxels	0.6 ± 0.0	0.7 ± 0.01	1.1 ± 0.11	0.6 ± 0.0	0.7 ± 0.0	1.2 ± 0.12	0.6 ± 0.0	0.7 ± 0.0	1.0 ± 0.08
Static voxel update	0.9 ± 0.09	1.4 ± 0.15	2.2 ± 0.12	1.0 ± 0.11	1.5 ± 0.08	2.2 ± 0.05	0.7 ± 0.07	1.2 ± 0.14	2.0 ± 0.08
Dynamic voxel update	1.2 ± 0.09	2.9 ± 0.03	8.3 ± 0.76	1.1 ± 0.03	3.3 ± 0.39	8.2 ± 1.08	0.9 ± 0.05	2.2 ± 0.34	6.8 ± 0.82
Ray-traced shadows	4.8 ± 0.32	5.3 ± 0.52	5.1 ± 0.65	4.3 ± 0.12	5.6 ± 0.17	4.8 ± 0.22	4.5 ± 0.34	4.9 ± 0.44	4.7 ± 0.13
Main scene + FXAA	4.2 ± 0.17	4.1 ± 0.02	4.5 ± 0.42	4.6 ± 0.73	4.2 ± 0.28	4.3 ± 0.36	4.5 ± 0.60	4.5 ± 0.67	4.5 ± 0.64
Total	11.9 ± 0.38	14.7 ± 0.54	22.2 ± 1.11	11.8 ± 0.75	15.6 ± 0.52	21.7 ± 1.18	11.4 ± 0.70	13.8 ± 0.88	19.9 ± 1.06

Table 7: *Average and standard deviation values of visible dynamic voxels processed and FPS for three different voxel neighbourhood sizes for the scenes* Rocks *(1.1M triangles), with a similar viewpoint as in Figure 7 with a vehicle and a skinned mesh character as dynamic objects, and* Factory *(2.3M triangles), with a similar viewpoint as in Figure 8, where eight skinned mesh characters and two ships are dynamic objects.*

Scene	*Rocks*			*Factory*		
Voxelization resolution	32^3	64^3	128^3	32^3	64^3	128^3
Visible dynamic voxels (5^3 voxel neighbourhood)	567 ± 27	1017 ± 22	2229 ± 116	1475 ± 56	3644 ± 60	9981 ± 60
Visible dynamic voxels (3^3 voxel neighbourhood)	176 ± 22	378 ± 45	1015 ± 50	606 ± 17	1615 ± 8	4602 ± 68
Visible dynamic voxels (no voxel neighbourhood)	27 ± 8	62 ± 18	202 ± 46	106 ± 2	314 ± 5	1001 ± 23
FPS (5^3 voxel neighbourhood)	131 ± 3	112 ± 3	73 ± 2	147 ± 1	100 ± 2	51 ± 1
FPS (3^3 voxel neighbourhood)	143 ± 2	127 ± 4	84 ± 2	164 ± 2	121 ± 3	69 ± 1
FPS (no voxel neighbourhood)	151 ± 2	134 ± 2	93 ± 1	173 ± 3	144 ± 2	89 ± 2

Figure 7, with a vehicle and a skinned mesh character as dynamic objects, and *Factory*, with the same viewpoint as in Figure 8 and where two large ships and eight skinned mesh characters are dynamic objects. As we can see, as the size of the voxel neighbourhood decreases from our initial configuration with 5^3 voxels to 0, the number of visible dynamic voxels processed decreases to a range between 0.5% and 1% of the original numbers for the scenes *Rocks* at voxelization resolution 32^3 and *Factory* at voxelization resolution 128^3, respectively, while FPS increase between 15% and 74% for the same scenes.

6. Discussion

Even as an academic prototype, our technique achieves higher quality and performance than global illumination techniques currently used in the state-of-the-art video game industry (*e.g.* Light Propagation Volumes, RTXGI, and Lumen). Although performance remains a crucial point for us, we prioritize quality over performance when compared with other techniques. We tested it on medium and large dynamic scenes built with assets common in current video games (from the Quixel [Epi] asset library), with light interactions from one light source between complex static and dynamic scene objects. To facilitate comparison between different rendering techniques that do not produce the same results due to different implementations, we propose to use the Enhanced Image Colour Transfer method [JR21] with the same parameter values in each case. We also provide the original unaffected sets of images in Figures 9 and 10. As an extra caution, we removed as many stages as possible from the rendering pipelines when measuring other rendering techniques (Unreal Engine and Godot) to maximize their performance for a more fair comparison. Even though, it is important to mention that our timings of those rendering engines may include other features and improvements not present in our technique.

Figure 7 on page 14 shows a large scene (*Rocks*, 1.1M triangles) comparison between our technique at voxelization resolution 128^3 at 73 fps (b), voxelization resolution 64^3 at 112 fps (c), and voxelization resolution 32^3 at 132 fps (d), and the best global illumination techniques currently used in video games: Lumen at 74 fps (e), Voxel Cone Tracing at 137 fps (f), Light Propagation Volumes at 250 fps (g) and RTXGI at 68 fps (h). As can be seen when comparing the leftmost zoomed-in image at the bottom row for each detail image with the ground truth (a), our technique can reproduce the green, blue and red reflections on the concrete slab observed in the ground truth better than any other technique. LPV and VCT offer results with a much lower quality than any other technique, with the coloured reflections missing. As can be observed, the image fromLumen suffers from small artefacts (like the white light behind the animated character). RTXGI suffers from noise and shows a flatter and dimmer reflection when compared to our technique. Only our technique at voxelization resolutions of 128^3 and 64^3 and RTXGI show a green tone on the animated character, with the other techniques showing uniform (flat) or white tones. Regarding the animated car details in the rightmost zoomed-in image at the bottom row of each image, our technique shows brighter results, while no technique is capable of achieving close similarities with the ground truth. RTXGI and LPV present the most similar outcomes, while Lumen suffers from some artefacts. For the zoomed-in image at the top row for each image, we can see how our technique conveys the blue/red/green tones from lit areas about 10–15 m away, while all remaining techniques fall into a flat, *i.e.* more uniform, result. VCT has a better performance than ours, at a cost of much lower quality at any voxelization resolution, displaying completely flat results. LPV has also better performance than our technique, but shows much lower quality results than ours at voxelization resolutions 128^3 and 64^3, while also showing temporal instability, with some light flickering as the car moves. This can be seen in the video sequences from the accompanying material.

We use a different, larger and open-world scene *Factory* (2.3M triangles) in Figure 8 for comparison between our technique at voxelization resolution 128^3 at 53 fps (b), voxelization resolution 64^3 at 100 fps (c) and voxelization resolution 32^3 at 147 fps (d) and the top-tier Global illumination techniques currently used in video games: Lumen at 54 fps (e), Voxel Cone Tracing at 11 fps (f), Light Propagation Volumes at 190 fps (g) and RTXGI at 55 fps (h), with a ground truth rendered with Blender Cycles with 4096 samples per pixel (a).

When comparing details from the leftmost zoomed-in image at the top row for each rendering technique (blue ship moving through the scene), we can see that LPV and VCT show flatter results, while our technique at voxelization resolutions 64^3 and 32^3, Lumen, and RTXGI achieve results closer to the ground truth. However, we can observe lighting artefacts for Lumen (the darker right part of the

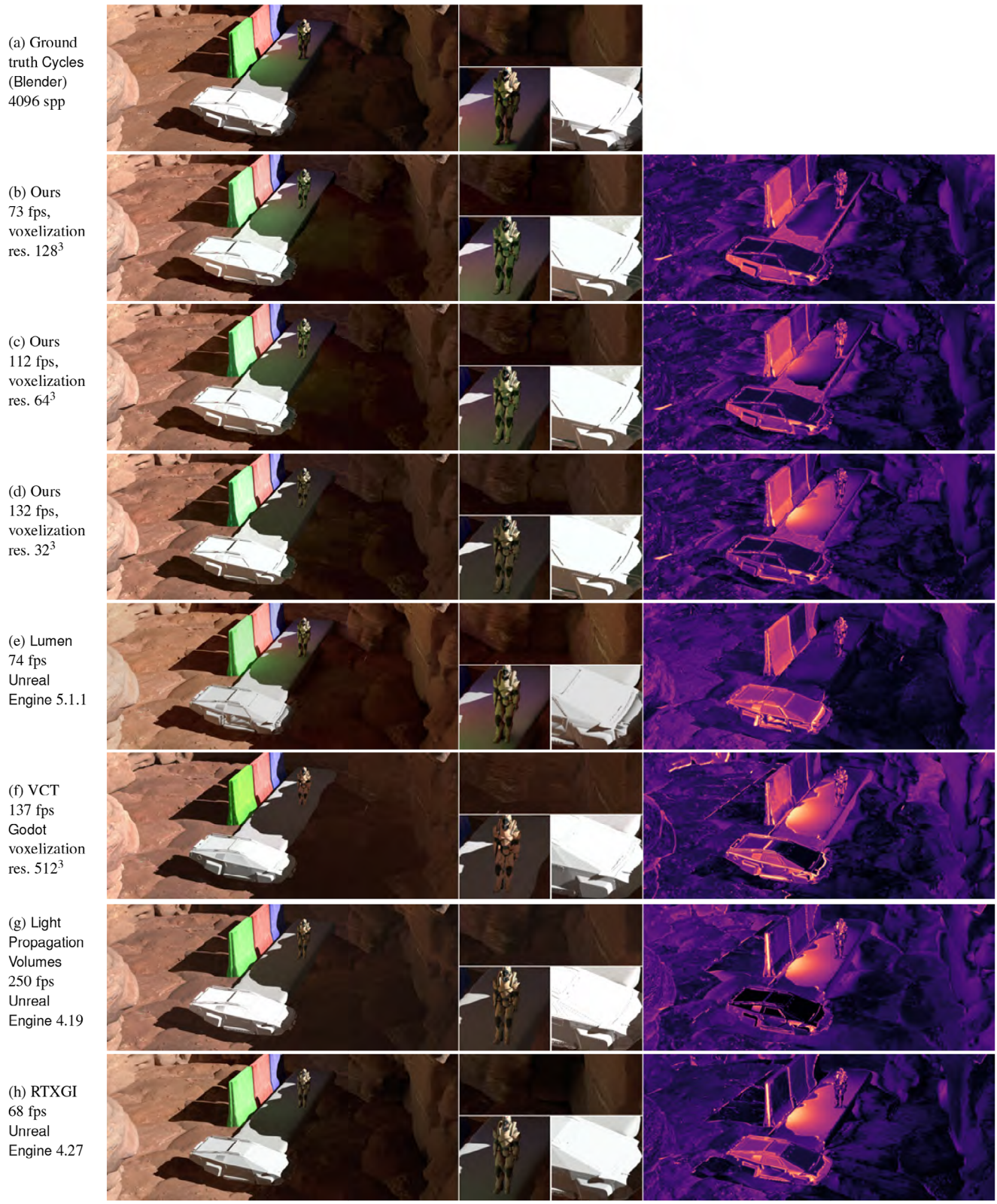


Figure 7: *Comparison of rendering techniques on the scene* Rocks *(1.1M triangles), built with assets from the* Quixel *[Epi] asset library. In the rightmost column, we show the difference with the ground-truth rendering using NVIDIA's* FLIP *[ANAM*20] tool.*

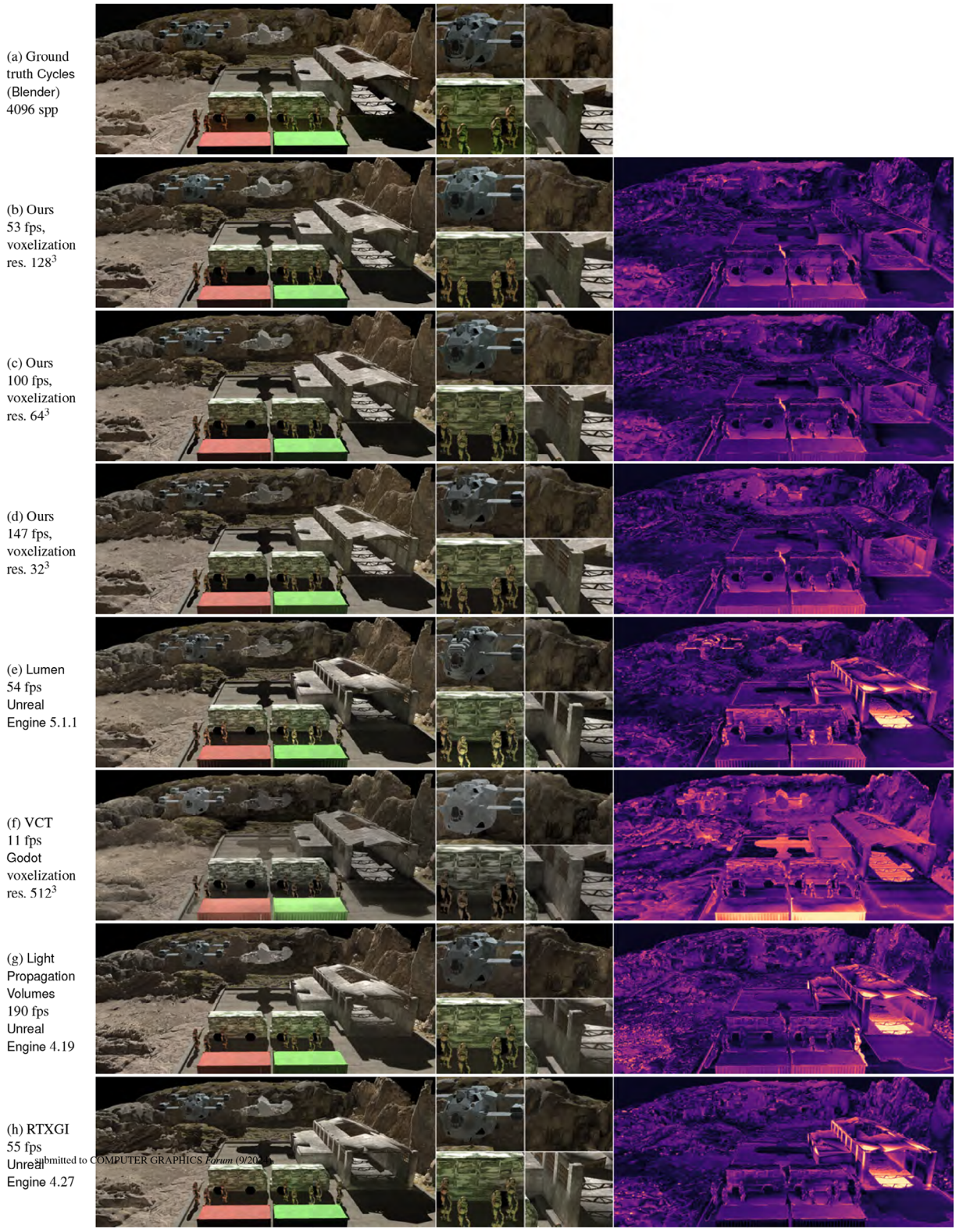


Figure 8: *Comparison of rendering techniques on the scene* Factory *(2.3M triangles), built with assets from the* Quixel *[Epi] asset library. The rightmost column shows the difference with the ground-truth rendering using NVIDIA's* FLIP *[ANAM*20] tool.*

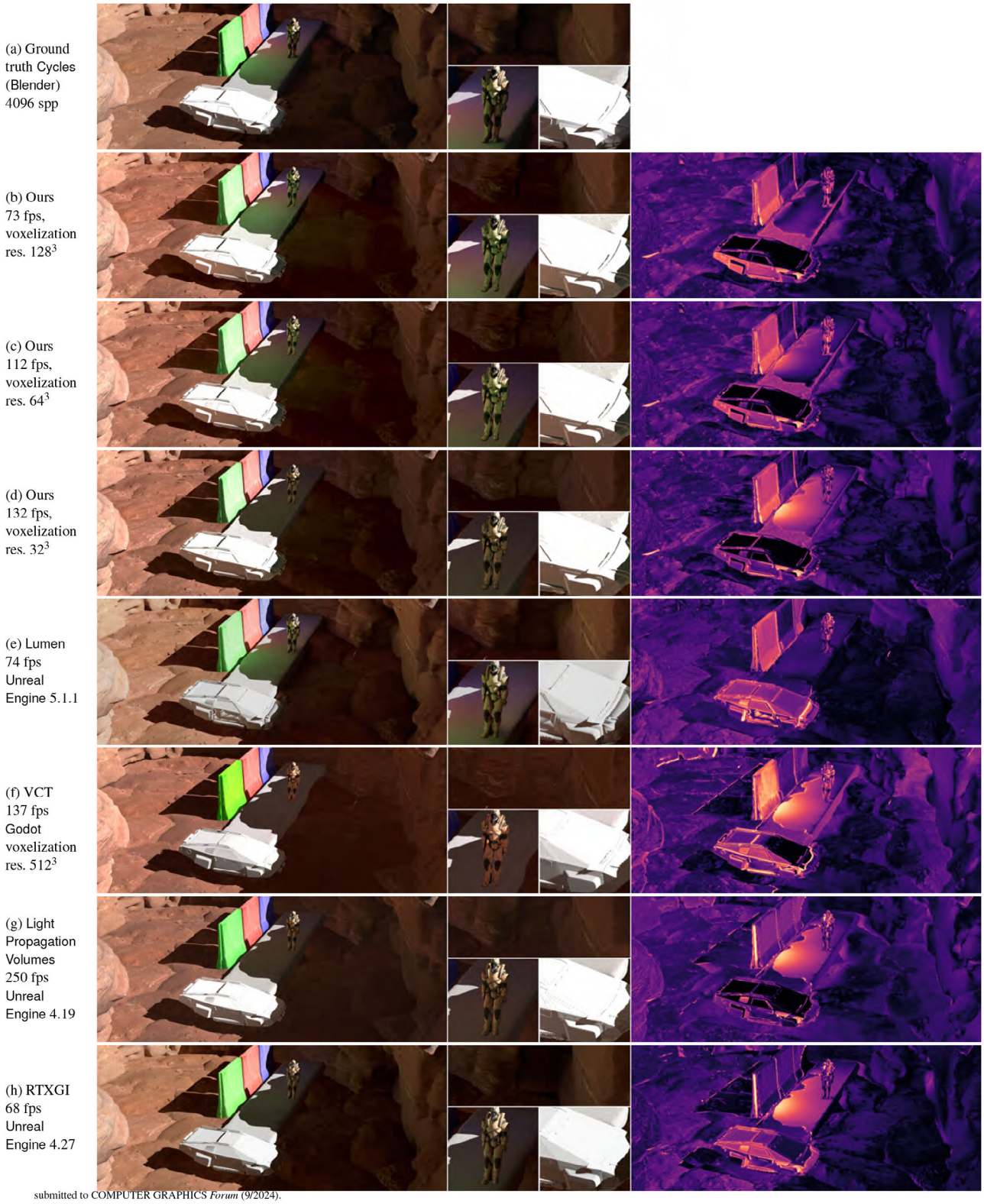


Figure 9: *Original images from Figure 7 not using the* Enhanced Image Colour Transfer *method [JR21]. In the rightmost column, we show the difference with the ground-truth rendering using NVIDIA's* FLIP *[ANAM*20] tool.*

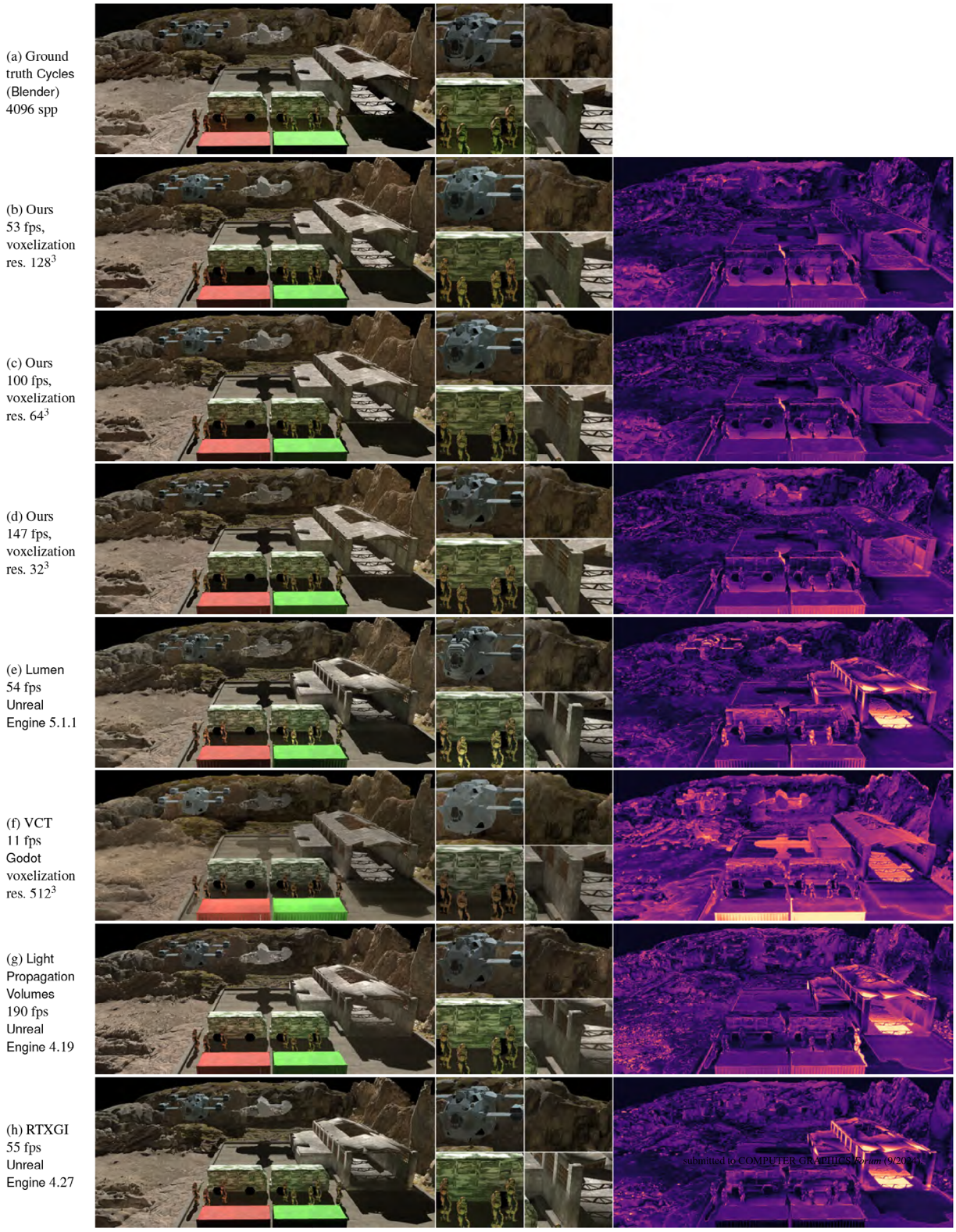


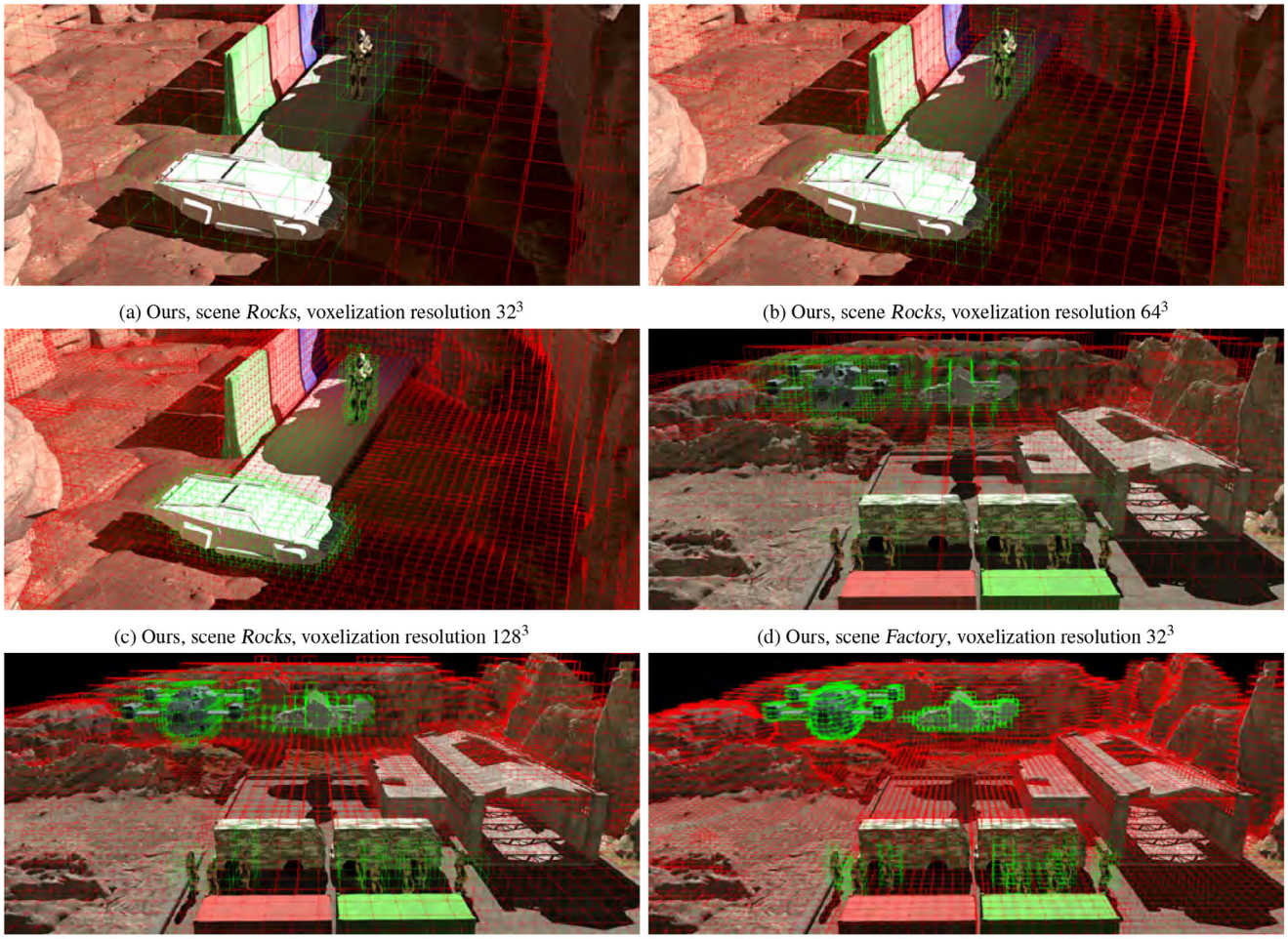
Figure 10: *Original images from Figure 8 not using the* Enhanced Image Colour Transfer *method [JR21]. The rightmost column shows the difference with the ground-truth rendering using NVIDIA's* FLIP *[ANAM*20] tool.*

ship). When it comes to temporal stability (this can be seen in the accompanying material video sequences), our technique offers stable results, while LPV and RTXGI show artefacts and sudden changes in lighting as the ship moves, especially with RTXGI. Lumen offers better temporal stability than LPV and RTXGI, but it is still not correct, as we can observe flickering on the right side of the ship as it moves. We can see how Godot's VCT shows a low performance (11 fps) with this scene, especially noticeable when compared with its performance in our previous test scene *Rocks*. This is due to the poorly performing implementation of VCT by Godot. We believe future implementations of VCT in this engine will achieve better performance and stability across different scenes. For a comparison between our work and a fully optimized version of VCT, please refer to the academic comparison with VCT in the last part of this section. If we compare the details from the leftmost zoomed-in image at the bottom row for each rendering technique (four animated characters) with the ground truth, we can see how only our technique at voxelization resolution 128^3 and RTXGI are capable of reproducing a green tone on the animated characters, although RTXGI shows much brighter results on the vehicle in the back. Lumen achieves more intense but flatter results, similar to our technique at voxelization resolution 64^3, while the remaining techniques and our technique at voxelization resolution 32^3 show mostly flat results. Regarding temporal stability, all techniques produce correct results. We can see how, as the characters move away from the green container, the received green irradiance fades away. In the case of Lumen, the fading barely changes when compared with other techniques. When we compare the details from the rightmost zoomed-in image at the top row for each rendering technique (rock formation) with the ground truth, we can see how our technique at all voxelization resolutions, especially at 64^3 and 128^3, resembles the ground-truth results. Lumen can achieve a better occlusion but fails to match the quality of our technique, offering results more similar to RTXGI. VCT and LPV differ from the ground truth. Lastly, if we compare the details from the rightmost zoomed-in image at the bottom row for each rendering technique (building corner and roof) with the ground truth, our technique at voxelization resolutions 64^3 and 128^3 achieves occlusion in the corner and a green tone similar to the ground truth. VCT, LPV and our technique at voxelization resolution 32^3 offer mostly flat results. RTXGI reproduces a green tone but fails regarding occlusion with flat results. Lumen shows good occlusion and green tone but suffers from some bright areas on the columns and next to the entrance, as well as some artefacts on the roof. In this scene, our technique at voxelization resolutions 64^3 and 128^3 offers better quality and temporal stability than any other technique analysed, with the case of 64^3 performing better than Lumen (100 vs. 54 fps). For a better evaluation of our technique we show in Figure 11 the voxel grid for the scenes *Rocks* and *Factory* at voxelization resolutions of 32^3, 64^3 and 128^3. These results provide another example of how our work reaches better quality and temporal stability than state-of-the-art techniques from commercial game engines.

Voxel Cone Tracing (VCT) is considered the reference in voxel-based global illumination. In Figure 12, we show a comparison with the scene *Sponza Atrium* between VCT with a Sparse Voxel Octree implementation [CNS*11], using material courtesy of Crassin [Cra21], with our technique using a configuration as similar as possible to the one provided. We also applied the Enhanced Image Colour Transfer method [JR21] to the images in Figure 12, to facilitate comparison, applying the same parameter values for all images except for the ground-truth rendering, which was used as reference. Our ground-truth rendering has over 1200 samples per pixel and is done withLuxCore in Blender. By using different path tracing reference algorithms for ground-truth renderings, we broaden our comparison spectrum, while LuxCore also offers better internal adjustments for this scene. We provide the original unchanged images in Figure 13. As can be seen in the top, leftmost zoomed-in image in Figure 12 (hanging blue fabric) when comparing with the ground truth, our work at all voxelization resolutions shows a correct shade behind the fabric, while both VCT samples show clear light leaks. When comparing the bottom, leftmost zoomed-in image from Figure 12 (coloured curtains) with the ground truth, we can see how VCT displays colour bleeding on the arches above each curtain, with our work at voxelization resolutions 64^3 and 128^3 displaying colour bleeding as well, with a lower intensity. If we compare the rightmost zoomed-in image (floor tiles in shade) from Figure 12 with the ground truth, we can see how our work at voxelization resolutions 64^3 and 128^3 displays clear bluish and reddish colour bleeding from light bouncing off the curtains, thanks to a better irradiance gathering. VCT only achieves colour bleeding for voxelization resolution 512^3 which, when compared with the ground truth, quickly fades away and has a dim bluish colour bleeding, while VCT for voxelization resolution 128^3 shows flat results. These results show how our work at voxelization resolution 64^3 achieves an overall better quality than VCT at voxelization resolution 128^3. Table 8 shows the memory usage of our technique and the Sparse Voxel Octree implementation used in VCT, for different voxelization resolutions for the scene *Sponza Atrium* in Figure 12. Crassin [Cra21] also provided a value of 613 fps at a voxelization resolution of 128^3 on an RTX 3090 GPU for this scene, which is around 275 fps on the GPU that we used for testing (an RTX 2060). As mentioned at the beginning of this section, we prioritize quality over performance when comparing our work with other techniques. As discussed in the zoomed-in images from Sponza in Figure 12, our work generates at voxelization resolution 64^3 higher quality results and also higher FPS than

Table 8: *Memory usage for* Voxel Cone Tracing *(VCT) following a* Sparse Voxel Octree *implementation, and for our technique (ours) at five different voxelization resolutions for a static scene* Sponza Atrium*, with the camera and light setup as in Figure 12. We include the total number of static voxels in each case, the main factor being related to memory usage. Information for our technique at voxelization resolutions* 256^3 *and* 512^3 *are approximated based on the average values of the ratio between values at voxelization resolutions* 64^3 *and* 32^3*, and the ratio between values at voxelization resolutions* 128^3 *and* 64^3.

Scene	*Sponza Atrium*				
Voxelization	32^3	64^3	128^3	256^3	512^3
VCT: Memory (MB)	1.2	6.5	34.1	164.9	813.2
Ours: Memory (MB)	35.5	158.1	751.9	3462.1	15941.2
VCT: FPS	N/A	N/A	275	N/A	N/A
Ours: FPS	307 $\pm$0.03	286 $\pm$0.02	214 $\pm$0.09	179	150
Ours: Static voxels	5201	23,975	109,241	0.5M	2.3M



(a) Ours, scene *Rocks*, voxelization resolution 32^3

(b) Ours, scene *Rocks*, voxelization resolution 64^3

(c) Ours, scene *Rocks*, voxelization resolution 128^3

(d) Ours, scene *Factory*, voxelization resolution 32^3

(e) Ours, scene *Factory* scene, voxelization resolution 64^3

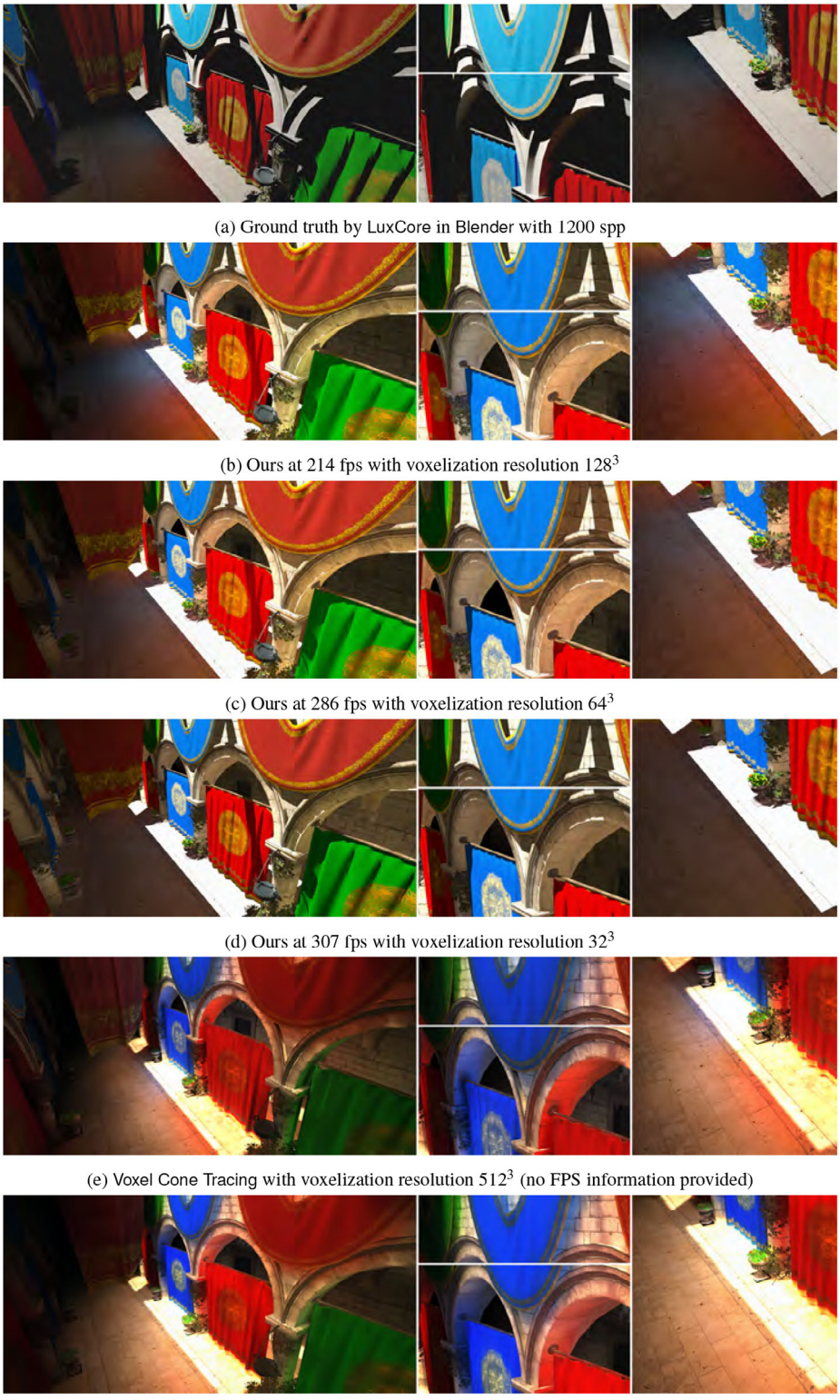
(f) Ours, scene *Factory*, voxelization resolution 128^3

Figure 11: *Voxel grids for the scenes* Rocks *and* Factory *at voxelization resolutions* 32^3*,* 64^3 *and* 128^3*, showing static (red) and dynamic voxels (green).*

VCT at a higher voxelization resolution of 128^3 (286 vs. 275 fps). Regarding memory consumption, VCT needs at least voxelization resolution 256^3 to achieve colour bleeding on the floor tiles and to match or improve our results at voxelization resolution 64^3. This requires from VCT significantly more memory than our work at a much smaller voxelization resolution of 64^3 (158.1 vs. 164.9 MB). These results show how our work offers better quality and performance, and a lower memory consumption than VCT, representing another contribution to the voxel-based global illumination field.

Among other academic techniques, Light Propagation Volumes by Kaplanyan and Dachsbacher [KD10] have been analysed on our test scenes *Rocks* and *Factory* in Figures 7 and 8. Following our criteria to prioritize quality over performance when comparing with other techniques, we can see how performance is higher in LPV than in our work, but at the expense of a much lower quality, showing in several cases mostly flat results, with also temporal instability in both scenes. Other techniques not included in our comparison are RSMs from Dachsbacher and Stamminger [DS05], which we estimate would have a better performance but lower quality than our work since it does not take into account occlusion; and PRT by Sloan *et al.* [SKS02], which we estimate would have a better quality than our work but would not fully support dynamic scenes since the only possible interaction with dynamic objects consists in projecting the pre-computed results from static objects onto dynamic objects and using pre-stored animations that require many pre-computations.

Our technique follows an irradiance volume approach, generating a grid of voxels where irradiance is stored per face of each voxel, not requiring any probe edition. We achieve great quality results for large, dynamic, complex scenes similar to offline global illumination renderers. At the ray tracing level, we follow a divide-and-conquer approach considering only dynamic scene objects, typically a smaller subset of all scene objects. As discussed in this section and shown in Figure 12 and Table 8, our technique offers better performance and quality with a lower memory consumption than academic techniques such as Sparse Voxel Octree-based Voxel Cone Tracing. By storing irradiance on two separate texture sets,



(a) Ground truth by LuxCore in Blender with 1200 spp

(b) Ours at 214 fps with voxelization resolution 128^3

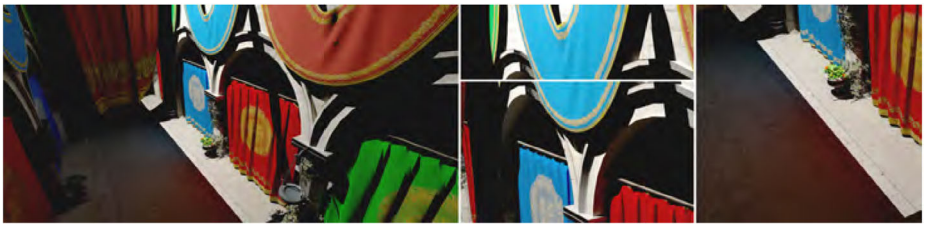
(c) Ours at 286 fps with voxelization resolution 64^3

(d) Ours at 307 fps with voxelization resolution 32^3

(e) Voxel Cone Tracing with voxelization resolution 512^3 (no FPS information provided)

(f) Voxel Cone Tracing at 275 fps with voxelization resolution 128^3

Figure 12: *Comparison of rendering techniques on the scene* Sponza Atrium.

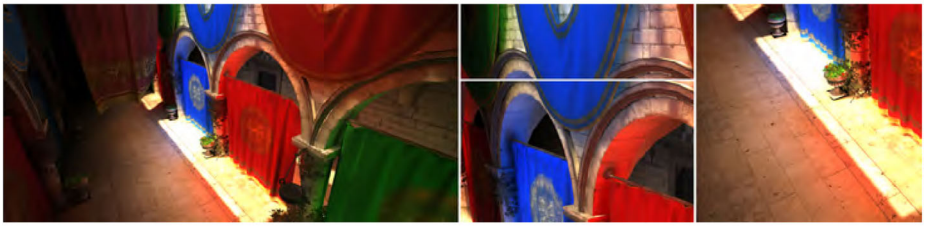


(a) Ground truth by LuxCore in Blender with 1200 spp

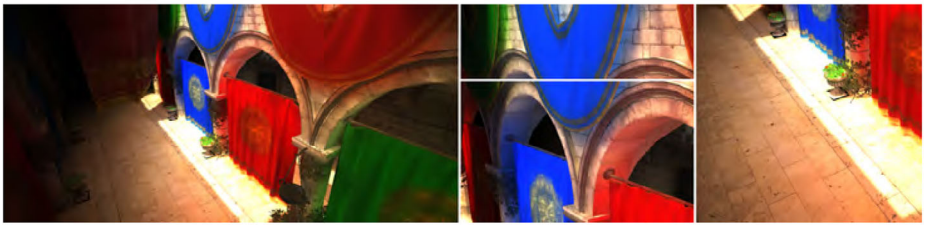
(b) Ours at 214 fps with voxelization resolution 128^3

(c) Ours at 286 fps with voxelization resolution 64^3

(d) Ours at 307 fps with voxelization resolution of 32^3



(e) Voxel Cone Tracing with voxelization resolution 512^3 (no FPS information provided)



(f) Voxel Cone Tracing at 275 fps with voxelization resolution 128^3

Figure 13: *Original images from Figure 12 not using the* Enhanced Image Colour Transfer *method [JR21].*

we avoid light leaks from dynamic onto static objects as can be seen in all our test scenes including dynamic scene objects (Figures 1, 7 and 8). Traditional voxel-based global illumination techniques require casting several diffuse cones per fragment. We achieve a low-cost final rendering using a simple post-processing pass that requires a fixed number of three texture samples at the same mipmap level for irradiance computations.

7. Limitations

As with other global illumination techniques, ours has also its shortcomings. We summarize them in this section with possible improvements.

Even though our technique does not require large voxelization resolutions to achieve good results, the amount of memory required per voxel face remains important concerning memory growth. To store visibility information, we use four bytes for each one of the 128 ray directions per-voxel face. We need two visibility sets for static voxels (one to cache static voxel visibility and one to store updated dynamic visibility) and one set for dynamic voxels. We could ignore the direction of the normal at the intersected geometry, storing only the distance from the ray origin, allowing us to re-construct the intersection information. In this case, only half of the memory would be required for both static and dynamic voxels.

In our technique, voxels generated due to dynamic scene objects require computing a large set of neighbouring voxels to avoid flickering, as mentioned in Figure 4 (left). This approach guarantees that irradiance values will be ready for interpolation, without resulting in darkened values for any fragment generated by the dynamic objects. In most cases, a 3^3 neighbourhood is enough, while in others computing a 5^3 neighbourhood provides the correct results. As the density of dynamic scene objects grows, irradiance computations for some of those voxels can be shared among nearby dynamic scene objects, but in general, the number of dynamic voxels to process can grow considerably as the number of dynamic scene objects increases, as can be seen in Table 3. Although the computations grow proportionally with the number of voxels to process, as mentioned in Section 5.1, the direct consequence is a higher cost for the dynamic voxel update, which is generally more expensive due to casting rays in an acceleration structure containing all the scene objects. Caching static visibility for each dynamic voxel that needs to be computed, requiring only to update visibility in a less expensive acceleration structure with only the dynamic scene objects, and keeping its information in a memory pool until it is no longer needed could contribute to improving performance.

In Table 4, we expose the ray-tracing budget required by our implementation. To achieve good results, which generally requires using at least voxelization resolution 64^3, involves a cost of 1.1–2.8 rays per pixel, above the limit of those devices with poor ray-tracing performance such as the XBox X|S and PS5 (0.25 to 0.5 rays per pixel). By splitting the voxel visibility updates for each voxel face through four frames, casting only a subset of 32 rays from the total 128 ray directions considered per voxel face, we could reduce those numbers below 0.7 rays per pixel at full HD resolution.

Our system handles a single light source. In our implementation, adding support for more light sources is straightforward. We just consider a small extra buffer per light source to store per-voxel-lit information at the level of a voxel, doing the computation only once for any number of times the voxel-lit information is needed later in the light bounce step.

Figure 14: *Detail from the* Bistro *scene from a different viewpoint. A dim reddish tone is visible on the left part of the metal rails holding the red awnings, due to light leakage through the thin geometry of the awnings.*

We currently implemented a single light bounce. Performing extra bounces involves updating irradiance for an extended set of voxels. To do so, we start with the set of voxels visible from the camera. We need to add, for each camera visible voxel, all static and dynamic visible voxels from its visibility information. Some can fall outside the view of the camera. Since this can represent a large number of voxels to process, to reduce the impact on performance, some strategies could be followed, such as discarding those voxels that are beyond a distance threshold from the voxels visible to the camera. These light bounces would take information from previous bounces from the irradiance textures. We leave this as an avenue for further research.

Our technique renders diffuse materials. Adding support for glossy materials could be done with two small changes. First, a post-processing pass in the G-buffer would detect all intersected scene objects covered with a glossy material. Each intersection would cast a new ray in the reflected direction along the surface normal (simulating a specular material). If the new ray intersects any scene geometry, we would add its corresponding static or dynamic voxel to the list of voxels for which we need to compute a light bounce. The second change would involve informing in the main rendering step which voxel is needed to perform the glossy material computations. We would need to store the coordinates of the voxel intersected by the new ray for each pixel of the main rendering step, with a memory increase similar to adding a 24-bit render target.

Regarding thin geometries, voxel-related techniques can suffer from light leakage and extra-darkening. Thanks to the irradiance filtering computations, extra-darkening is generally avoided, while computing visibility per voxel face through ray tracing contributes to minimizing light leakage. We could also detect some cases where thin geometries are present, as in Figure 14. The process of irradiance filtering for dynamic voxels involves acquiring irradiance

Figure 15: *Irradiance filtering can lead to light leaks in certain situations on dynamic objects, as shown in these details from the* Bistro *scene. Left: A blue tint is visible on the dragon's body and slightly on the back of its head due to irradiance filtering from neighbouring voxels that have visibility to geometry from the dragon's blue body part, shown in Figure 1. Right: Disabling irradiance filtering avoids this issue but may lead in some cases to some extra-darkening.*

from neighbouring voxels. If those neighbouring voxels have visibility of scene objects with irradiance values that largely differ from the voxel's irradiance, light leaks may eventually be generated, as shown in Figure 15.

Our technique relies on minimal ray tracing, compute and bandwidth capabilities, benefiting from our implementation in the Vulkan API, which maximizes GPU usability. The viability of our technique on low-end GPUs, such as integrated GPUs and bandwidth-limited GPUs used on mobile devices, remains to be analysed, requiring a re-design of the most computationally demanding workloads.

8. Conclusions

We presented a new technique belonging to the Irradiance Volumes family for the real-time display of diffuse global illumination. We store irradiance per voxel face, computing visibility through ray tracing. We follow a divide-and-win approach, casting each ray in a representation of the scene containing only dynamic scene objects, to lower the cost of ray tracing, even if dynamic scene objects represent the majority of the scene objects. By using voxels as locations where to compute irradiance for both static and dynamic scene objects, we do not require any manual scene edition. Our technique achieves a higher quality, temporal stability and performance than the techniques used in the industrial state of the art for global illumination in video games (Voxel Cone Tracing, Light Propagation Volumes, RTXGI and Lumen) in large and complex scenes with modern assets, animated characters and dynamic scene objects and light sources. As future options, we consider implementing glossy and specular materials and participating media while optimizing all areas of our implementation (memory usage, dynamic and static voxel updates and ray-tracing budget).

Acknowledgements

This research was partially funded by Grant PID2021-122136OB-C22 funded by MICIU/AEI/10.13039/501100011033, by ERDF A way of making Europe and by grant RGPIN-2020-05117 from Natural Sciences and Engineering Research Council of Canada.

Conflicts of Interest

The authors declare no conflicts of interest.

References

[ANAM*20] Andersson P., Nilsson J., Akenine-Möller T., Oskarsson M., Åström K., Fairchild M. D.: FLIP: A difference evaluator for alternating images. *Proceedings of the ACM in Computer Graphics and Interactive Techniques 3*, 2 (2020), 1–23.

[AP22] Ayerbe A. C., Patow G.: Clustered voxel real-time global illumination. *Computers & Graphics 103* (Apr. 2022), 75–89. doi: https://doi.org/10.1016/j.cag.2022.01.005.

[BJW21] Boksansky J., Jukarainen P., Wyman C.: Rendering many lights with grid-based reservoirs. In *Ray Tracing Gems II*. Apress, Berkeley (2021), pp. 351–365. doi: https://doi.org/10.1007/978-1-4842-7185-8_23.

[Boi21] Boissé G.: World-space spatiotemporal reservoir reuse for ray-traced global illumination. In *SIGGRAPH Asia 2021 Technical Communications* (Dec. 2021), ACM. doi: https://doi.org/10.1145/3478512.3488613.

[BWP*20] Bitterli B., Wyman C., Pharr M., Shirley P., Lefohn A., Jarosz W.: Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics 39*, 4 (Aug. 2020). doi: https://doi.org/10.1145/3386569.3392481.

[BWS02] Beckhaus S., Wind J., Strothotte T.: Hardware-based voxelization for 3D spatial analysis. In *Proceedings of the 5th International Conference on Computer Graphics and Imaging* (2002), vol. *20*, pp. 47–54.

[CC16] Chen Y.-Y., Chien S.-Y.: Lighting-driven voxels for memory-efficient computation of indirect illumination. *The Visual Computer 32*, 6-8 (June 2016), 781–789. doi: https://doi.org/10.1007/s00371-016-1235-y.

[CDAS20] Currius R. R., Dolonius D., Assarsson U., Sintorn E.: Spherical Gaussian light-field textures for fast precomputed global illumination. *Computer Graphics Forum 39* (2020), 133–146.

[CG12] Crassin C., Green S.: *Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer*. CRC Press, Boca Raton, 2012, pp. 303–319.

[CNS*11] Crassin C., Neyret F., Sainz M., Green S., Eisemann E.: Interactive indirect illumination using voxel cone tracing: A

preview. In *I3D'11: Symposium on Interactive 3D Graphics and Games* (2011), ACM, pp. 207–207. doi: http://doi.acm.org/10.1145/1944745.1944787.

[Cra21] Crassin C.: Personal communication, 2021.

[DS05] Dachsbacher C., Stamminger M.: Reflective shadow maps. In *I3D'05: Proceedings of the Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), ACM, pp. 203–231. doi: http://doi.acm.org/10.1145/1053427.1053460.

[DS06] Dachsbacher C., Stamminger M.: Splatting indirect illumination. In *Proceedings of the Symposium on Interactive 3D Graphics and Games* (2006), ACM, pp. 93–100.

[DSDD07] Dachsbacher C., Stamminger M., Drettakis G., Durand F.: Implicit visibility and antiradiance for interactive global illumination. *ACM Transactions on Graphics (TOG) 26*, 3 (2007), 61.

[Epi] Epic Games: Quixel megascans. https://quixel.com/megascans/. Accessed: 2023-08-05.

[GSHG98] Greger G., Shirley P., Hubbard P. M., Greenberg D. P.: The irradiance volume. *IEEE Computer Graphics and Applications 18*, 2 (Mar. 1998), 32–43. doi: https://doi.org/10.1109/38.656788.

[HN12] Heitz E., Neyret F.: Representing appearance and pre-filtering subpixel data in sparse voxel octrees. In *EGGH-HPG'12: Proceedings of the Fourth ACM SIGGRAPH/ Eurographics Conference on High-Performance Graphics* (2012), Eurographics Association, pp. 125–134.

[JKG16] Jendersie J., Kuri D., Grosch T.: Real-time global illumination using precomputed illuminance composition with chrominance compression. *Journal of Computer Graphics Techniques 5*, 4 (2016), 8–35.

[JR21] Johnson T., Renzullo M.: Further enhanced image colour transfer. https://github.com/TJCoding/Enhanced-Image-Colour-Transfer-2 (2021). Accessed 26 September 2024.

[Kaj86] Kajiya J. T.: The rendering equation. *ACM SIGGRAPH Computer Graphics 20*, 4 (Aug. 1986), 143–150.

[KBLE18] Kol T. R., Bauszat P., Lee S., Eisemann E.: MegaViews: Scalable many-view rendering with concurrent scene-view hierarchy traversal. *Computer Graphics Forum 38*, 1 (July 2018), 235–247. doi: https://doi.org/10.1111/cgf.13527.

[KD10] Kaplanyan A., Dachsbacher C.: Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010), ACM, pp. 99–107.

[Kel97] Keller A.: Instant radiosity. In *SIGGRAPH'97: Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques* (1997), ACM, pp. 49–56. doi: https://doi.org/10.1145/258734.258769.

[KLM*19] Koskela M., Lotvonen A., Mäkitalo M., Kivi P., Viitanen T., Jääskeläinen P.: Foveated real-time path tracing in visual-polar space. In *Eurographics Symposium on Rendering —DL-only and Industry Track* (2019), The Eurographics Association, pp. 367–374. https://doi.org/10.2312/sr.20191219.

[KTHS06] Kontkanen J., Turquin E., Holzschuch N., Sillion F. X.: Wavelet radiance transport for interactive indirect lighting. In *Eurographics Symposium on Rendering* (2006), The Eurographics Association, pp. 161–171.

[LKB*22] Lin D., Kettunen M., Bitterli B., Pantaleoni J., Yuksel C., Wyman C.: Generalized resampled importance sampling. *ACM Transactions on Graphics 41*, 4 (July 2022), 1–23. doi: https://doi.org/10.1145/3528223.3530158.

[MGNM19] Majercik Z., Guertin J.-P., Nowrouzezahrai D., McGuire M.: Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques (JCGT) 8*, 2 (June 2019), 1–30. http://jcgt.org/published/0008/02/01/.

[MMNL16] Mara M., McGuire M., Nowrouzezahrai D., Luebke D.: Deep G-buffers for stable global illumination approximation. In *HPG'16: Proceedings of High Performance Graphics* (2016), Eurographics Association, pp. 87–98.

[MMNL17] McGuire M., Mara M., Nowrouzezahrai D., Luebke D.: Real-time global illumination using precomputed light field probes. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D 2017* (Feb. 2017), pp. 11. https://casual-effects.com/research/McGuire2017LightField/index.html.

[MMSM21] Majercik Z., Marrs A., Spjut J., McGuire M.: Scaling probe-based real-time dynamic global illumination for production. *Journal of Computer Graphics Techniques (JCGT) 10*, 2 (May 2021), 1–29. http://jcgt.org/published/0010/02/01/.

[NRS14] Nalbach O., Ritschel T., Seidel H.-P.: Deep screen space. In *I3D'14: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2014), ACM, pp. 79–86. https://doi.org/10.1145/2556700.2556708.

[NSL*07] Nehab D., Sander P. V., Lawrence J., Tatarchuk N., Isidoro J. R.: Accelerating real-time shading with reverse reprojection caching. In *Graphics Hardware* (2007), vol. *41*, pp. 61–62.

[NSW09] Nichols G., Shopf J., Wyman C.: Hierarchical image-space radiosity for interactive global illumination. *Computer Graphics Forum 28* (2009), 1141–1149.

[OLK*21] Ouyang Y., Liu S., Kettunen M., Pharr M., Pantaleoni J.: ReSTIR GI: Path resampling for real-time path tracing. *Computer Graphics Forum 40*, 8 (Nov. 2021), 17–29. doi: https://doi.org/10.1111/cgf.14378.

[Pap11] Papaioannou G.: Real-time diffuse global illumination using radiance hints. In *HPG'11: Proceedings of the ACM*

SIGGRAPH Symposium on High Performance Graphics (2011), ACM, pp. 15–24. http://doi.acm.org/10.1145/2018323.2018326.

[PSK*16] Patney A., Salvi M., Kim J., Kaplanyan A., Wyman C., Benty N., Luebke D., Lefohn A.: Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics (TOG) 35*, 6 (2016), 1–12.

[SKS02] Sloan P.-P., Kautz J., Snyder J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques* (2002), ACM, pp. 527–536.

[SL17] Silvennoinen A., Lehtinen J.: Real-time global illumination by precomputed local reconstruction from sparse radiance probes. *ACM Transactions on Graphics (TOG) 36*, 6 (Nov. 2017), 230:1–230:13. doi: http://doi.acm.org/10.1145/3130800.3130852.

[Sri16] Sridhar S.: *Digital Image Processing* (2nd edition). Oxford University Press, New Delhi, India, 2016.

[SRS14] Sugihara M., Rauwendaal R., Salvi M.: Layered reflective shadow maps for voxel-based indirect illumination. In *HPG'14: Proceedings of High Performance Graphics* (2014), Eurographics Association, pp. 117–125. http://dl.acm.org/citation.cfm?id=2980009.2980022.

[SS10] Schwarz M., Seidel H.-P.: Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics (TOG) 29*, 6 (2010), 1–10.

[Tak15] Takeshige M.: The basics of GPU voxelization. https://developer.nvidia.com/content/basics-gpu-voxelization (2015). Accessed 26 September 2024.

[THGM11] Thiedemann S., Henrich N., Grosch T., Müller S.: Voxel-based global illumination. In *I3D'11: Symposium on Interactive 3D Graphics and Games* (2011), ACM, pp. 103–110. http://doi.acm.org/10.1145/1944745.1944763.

[VJK21] Vicini D., Jakob W., Kaplanyan A.: A non-exponential transmittance model for volumetric scene representations. *ACM Transactions on Graphics (TOG) 40*, 4 (2021), 1–16.

[WK93] Wang S. W., Kaufman A. E.: Volume sampled voxelization of geometric primitives. In *Proceedings Visualization '93* (Berlin, Heidelberg, 1993), IEEE, Springer-Verlag, pp. 78–84.

[WNK22] Wright D., Narkowicz K., Kelly P.: Lumen: Real-time global illumination in unreal engine 5. Advances in real-time rendering in games. Course at SIGGRAPH 2022. 2022. https://advances.realtimerendering.com/s2022/index.html#Lumen.

[ZCEP07] Zhang L., Chen W., Ebert D. S., Peng Q.: Conservative voxelization. *The Visual Computer 23*, 9-11 (2007), 783–792.

Supporting Information

Additional supporting information may be found online in the Supporting Information section at the end of the article.

Supporting Information