



Real-time Discrete Visibility Fields for Ray-Traced Dynamic Scenes

Beril Günay^a, Ahmet Oğuz Akyüz^{a,*}

^aMiddle East Technical University, Department of Computer Engineering, Ankara, Turkey

ARTICLE INFO

Article history:

Received March 10, 2025

Keywords: Ray tracing, Real-time rendering, Visibility, Environment lighting

ABSTRACT

Environment lighting plays a crucial role in enhancing the realism of rendered images. As environment light sources are distributed over a sphere or hemisphere, the rendering process involves tracing multiple rays per pixel. This process is computationally expensive and therefore achieving real-time results is challenging. Many studies focus on visibility functions to reduce the number of traced rays, often achieved through precomputation methods. This study improves upon a recent CPU-based visibility precomputation method designed for rendering solely static scenes, enabling its use for dynamic scenes. Our key contribution is the parallelization of this algorithm across multiple compute shader invocations which well integrates into the modern real-time rendering pipeline. By analyzing the performance-quality impact of various parameters, we aim to discover the best configuration for a given scene. With our technique the entire preprocessing stage can be executed for a scene with more than 200K triangles in 4 ms, allowing for more than a 50 FPS performance including precomputation and rendering.

© 2025 Elsevier B.V. All rights reserved.

1. Introduction

It has long been noted that representing the light distribution of a rendered scene using an environment map greatly enhances realism [1]. By simply changing the environment map, the entire “look-and-feel” of the scene can be altered, creating the feeling that objects belong to their new environment [2]. However, rendering with environmental lighting is computationally expensive due to the requirement of tracing multiple rays per pixel. This is particularly necessary for diffuse objects, as they collect and reflect light from and to a large set of directions. Consequently, achieving real-time results becomes challenging.

To overcome this challenge, various precomputation methods were explored to represent and use environment lighting during rendering. These methods involve calculating and storing specific information offline, allowing for quick integration and

computation during runtime. Scattered points [3], irradiance volumes [4], light field probes [5], local light grids [6], radiance transfer [7], shadow fields [8], clustered principle components [9], light transport paths [10], and light maps [11] are among the notable precomputation techniques. Other methods involve precomputing visibility [12, 13, 14], as the visibility function holds significant importance as it helps to decrease the number of traced rays due to occlusion [15, 16, 17].

Among this latter group of techniques, a notable work was recently presented by Yang et al. [18] that involves storing the visibility information around each point in a uniform grid called discrete visibility fields (DVF). This information, which is created offline, is later used during rendering. When combined with hardware-accelerated ray tracing [19, 20], this approach can achieve real-time rendering performance with realistic environment lighting.

The primary drawback of DVF is that precomputation is done prior to runtime, limiting its application to static scenes [18]. Any animation or change in the scene geometry requires recreating the DVF, which is a costly operation. In this work, we

*Corresponding author: Tel.: +90-312-210-5565;
e-mail: beril.gunay@metu.edu.tr (Beril Günay),
aoakyuz@metu.edu.tr (Ahmet Oğuz Akyüz)

propose a GPU-based precomputation algorithm to enable the usage of DVF for dynamic scenes. We achieve this by distributing the precomputation between different compute shaders. As shown in the following sections, parallelizing the DVF algorithm is not trivial and requires careful application of parallelization techniques to achieve real-time performance.

2. Previous Work

Using environment maps for realistic lighting is a long-studied problem in computer graphics [1, 21, 22]. While older work relied on using traditional images to represent the environment, the use of high dynamic range (HDR) imagery paved the way to represent environments as actual light fields [23, 24]. Depending on whether forward or backward rendering pipelines are used, environment maps can be represented in various formats such as cubemaps [21], equilateral projections, and light probes [25].

In devices with lower computational budgets, it is customary to represent environments using light maps [26]. Light maps not only contain the infinitely far away environment light, but also the effect of objects on each other such as shadows and color bleeding. The lighting information at each point in the scene is precomputed into a texture using a realistic light simulation algorithm [27, 28] and is efficiently accessed during runtime. Static light maps prohibit dynamic changes in the scene geometry, although some recent algorithms can alleviate this restriction [29, 30].

Several other precomputation techniques also exist. Irradiance maps store light distribution of the environment as a small number of spherical harmonic coefficients discarding its high-frequency components [31, 32]. This is particularly useful for diffuse objects as the effect of high-frequency lighting on them is negligible.

Capitalizing on this idea, Sloan et al. also represent the light transfer over object surfaces as spherical harmonic coefficients [7]. This precomputed information is accessed during runtime for self-shadowing and self-interreflections allowing for real-time rendering of such advanced effects. Several improvements of this technique address its limitations such as large memory footprint [9] and low-frequency and distant lighting requirements [33, 34].

Visibility of a surface point from a light source or from another surface point is an important factor for computing direct and indirect lighting. Shadow mapping related techniques precompute this information from the light's point of view into a texture and use it in subsequent rendering passes [35]. While shadow mapping is not practical for environment lighting due to the wide distribution of incoming light, several techniques can be used to discretize it to a smaller number of samples [24, 36, 37].

Many other visibility-based methods exist to improve performance and reduce variance during ray tracing [12, 15, 38]. A recent algorithm designed for GPU-based ray tracing was proposed by Yang et al. [18]. In their work, the authors address the challenges of achieving real-time results with environmental lighting due to the necessity of tracing numerous rays. They

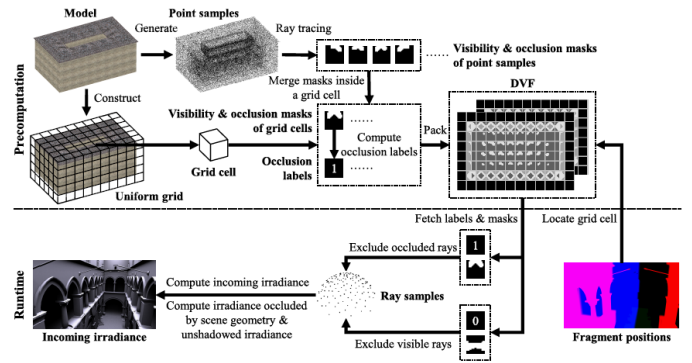


Fig. 2: Overview of algorithms in [18]. The figure is reproduced as-is from its original source.

argue that the evaluation of the visibility function is the key to rendering with environmental lighting. As a solution, they present precomputed Discrete Visibility Fields (DVF) that store the visibility information of static scenes in a uniform grid.

Because our work is designed to extend this algorithm to dynamic scenes, we review the fundamental principles behind this algorithm in the following sections.

2.1. Discrete Visibility Fields

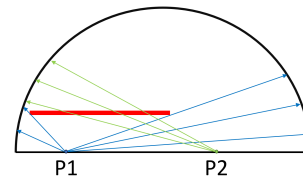


Fig. 1: The fundamental concept of DVF. The red bar is an occluder.

The fundamental concept of DVF is illustrated in Figure 1. If a surface point is mostly occluded such as the point P_1 , non-occluded (i.e. visible) directions are traced to calculate the irradiance. If a surface point is mostly visible by the environment such as P_2 , occluded rays are traced to calculate the occluded irradiance, which is

then subtracted from the total irradiance to obtain the actual irradiance. Visible and occluded directions in the vicinity of the point, as well as whether the points in that vicinity are mostly occluded or not, are calculated in the precomputation stage.

In the runtime phase, the information stored in DVF is utilized to reduce the number of traced rays. With this method, real-time rendering is achieved. However, because the precomputation phase is done on the CPU, no changes to the scene geometry are possible during runtime.

2.1.1. Precomputation algorithm

In the precomputation phase, a DVF is created that stores an approximation of the visibility of geometry contained within each grid cell that represents a voxelization of the scene. The precomputation algorithm used in [18] is provided in Algorithm 2 (see Appendix). The input of the algorithm is the mesh to be rendered and the output is the DVF.

During precomputation, first, the surface of the mesh is uniformly sampled and N_p points are generated. For each of these points, N_r rays are generated around the normal vector using the

Table 1: Compute shaders used for the precomputation of DVF.

#	Task	Dispatch method
1	Calculate triangle areas and normals	per triangle
2	Calculate partial sum of areas (Up Sweep)	based on SSBO size
3	Calculate cumulative sum of areas (Down Sweep)	based on SSBO size
4	Create DVF	per triangle
5	Compute occlusion labels l_{occ} (Parallel sum reduction)	per grid cell

- Generating uniform random points on the mesh surfaces, 56
- Creating visibility rays for each point and tracing them, 57
- Creating octahedral maps using ray intersection results, 58
- Converting the octahedral maps into occlusion labels. 59

Generating random points uniformly on a mesh surface requires a sampling strategy based on triangle area weights. This requires prior knowledge of the area of each triangle, the sum of the total mesh surface area, and the total number of points to be sampled. After the number of point samples for each triangle is calculated, coordinates for point samples must be generated randomly. For each of these points, rays must be generated with Hammersley sequence and they must be traced. Occlusion and visibility results must be written to an image array in the GPU. Lastly, occlusion labels must be determined.

We introduce a compute shader based solution to manage all of the mentioned tasks and perform precomputation on the GPU. Since Vulkan ray tracing extensions are used for implementing the runtime algorithm in [18], we utilize GPU semaphores and pipeline barriers to synchronize data between compute and graphics pipelines.

We present an overview of our methodology in Figure 3. First, initialization tasks are carried out by the CPU. Then, the rendering loop takes effect. Compute commands for DVF precomputation are executed by the GPU. Once DVF is ready, graphics commands are executed by the GPU, and the frame is rendered. Animation can be achieved by updating the transformation matrices on the CPU (e.g. based on user input) and recomputing the bounding box information again using Vulkan ray tracing extensions [42].

3.1. Compute Pipelines

The compute pipeline is carried out by 5 shader stages as shown in Table 1. The actual precomputation algorithm is implemented in Compute Shaders 4 & 5. Preceding shaders are used to extract the necessary data from the scene such as the triangle normals, areas, and sum of the areas.

In Figure 4, the compute shaders and resource accesses are provided. The compute shaders are represented with blue boxes. These shaders are dispatched sequentially and they alter the state of the resources when they are invoked. The column to the left side of the purple arrow shows the state of the resources before that shader's invocation. In the same column, the read/write access type of the shader is also presented using

Hammersley sequence with cosine-weighted hemisphere sampling [39, 28]. Each of these rays is rotated randomly around the normal, and they are traced. The octahedral map index of the ray is found by octahedral mapping [40]. If a ray intersects with the scene, 1 is written to $O_s^V(s)$ in the corresponding index. Otherwise, 1 is written to $O_s^O(s)$ in the corresponding index. These maps represent the occluded and visible directions for each point sample.

After $O_s^V(s)$ and $O_s^O(s)$ for all points are created, the scene is voxelized and the visibility of each cell (i.e., voxel) is approximated by OR operation of the point samples that fall into the same cell. The cell visibility maps are termed as $O_c^V(c)$ and $O_c^O(c)$. In the last step, the occlusion label $l_{occ}(c)$ of each grid cell is found by comparing the total number of occluded and visible directions stored in $O_c^V(c)$ and $O_c^O(c)$.

In our work, we replace this precomputation algorithm with a GPU-based implementation using compute shaders, which makes it possible to alter or animate the scene geometry during runtime.

2.1.2. Runtime Algorithm

In the runtime phase, the precomputed DVF is used as input to optimize the rendering process. The details of the algorithm are explained below and its pseudocode is provided in Algorithm 3 for completeness.

To calculate the incoming irradiance $E(\mathbf{x})$ of fragment \mathbf{x} , first the grid cell c containing \mathbf{x} is found and $l_{occ}(c)$ is obtained. N_r rays are generated around the normal vector using the Hammersley sequence with cosine weights [39, 28]. Each of these rays is rotated randomly around the normal using a blue noise texture [41], and the octahedral map index of the ray is found. Visibility $V_c(c, \omega)$ and occlusion $O_c(c, \omega)$ masks are extracted from the precomputed DVF.

If the number of occluded directions in the cell c is larger than the number of visible directions ($l_{occ}(c) = 1$), the ray is traced only if its direction, ω , is visible in cell c . Conversely, if the number of visible directions in cell c is larger than the number of occluded directions ($l_{occ}(c) = 0$), the ray is traced only if its direction is occluded in cell c . As a result, either $E(\mathbf{x})$ or $E_{occ}(\mathbf{x})$ is obtained respectively. If $l_{occ}(c) = 1$, $E(\mathbf{x})$ is found by subtracting occluded irradiance $E_{occ}(\mathbf{x})$ from the unshadowed irradiance $E_{unshad}(\mathbf{n})$.

The radiance $L_i(\omega)$ and unshadowed irradiance $E_{unshad}(\mathbf{n})$ terms are calculated with precomputed constant values. These precomputed values are obtained with irradiance map approximation method using spherical harmonics [31, 32]. This runtime algorithm is implemented by using Vulkan ray tracing extension VK_KHR_ray_query to perform ray tracing within the fragment shader for real-time performance [42].

3. Proposed Method

The method summarized in the previous section uses an offline approach for computing the DVF limiting its usage to static scenes. In this section, we explain how the DVF idea can be realized on the GPU so that dynamic scenes can be supported in real time. The precomputation method can be divided into 4 main steps:

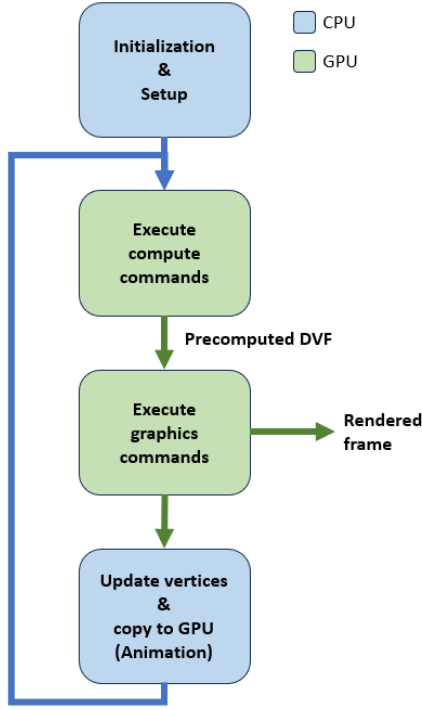


Fig. 3: The high-level overview of our method.

		CS 1	CS 2	CS 3	CS 4	CS 5	
		read	write	readwrite			
RESOURCES	CPU animation update	Resource state after CPU	Resource state after CS1	Resource state after CS2	Resource state after CS3	Resource state after CS4	Resource state after CS5
Vertices SSBO	Update with new vertices	V	V	V	V	V	V
Indices SSBO		I	I	I	I	I	I
Normals SSBO		--	N	N	N	N	N
Areas SSBO	Clear Areas SSBO	0	A	Partial sum A	Cumulative sum A	Cumulative sum A	Cumulative sum A
Areas sum SSBO	Clear Areas SSBO	0	0	0	Total sum A	Total sum A	Total sum A
DVF Masks Image Array		--	--	--	0	DVF	DVF
Acceleration structure	Update with new vertices	BLAS TLAS	BLAS TLAS	BLAS TLAS	BLAS TLAS	BLAS TLAS	BLAS TLAS
Occlusion label SSBO		--	--	--	--	--	Occlusion labels
Blue noise texture		Texture	Texture	Texture	Texture	Texture	Texture

Fig. 4: Resource usage of compute shaders.

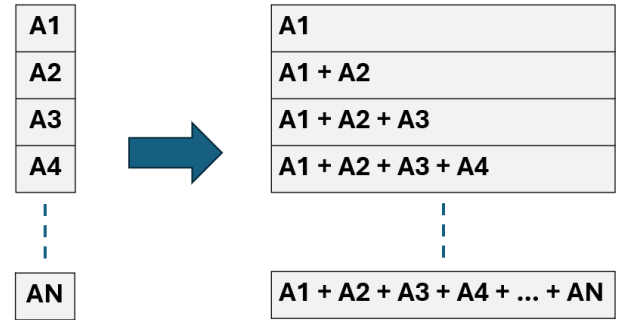


Fig. 5: Cumulative sum of triangle areas.

1 colors. For example, Compute Shader 1 reads data from *Vertices SSBO*¹ and *Indices SSBO* and writes data to *Areas SSBO* and *Normals SSBO*. The resources that are modified by a certain shader are marked with red arrows. The initial values of *Vertices SSBO* and *Indices SSBO* are determined by the CPU.

2 There are pipeline barriers placed between compute shaders that share resources. This way synchronization among compute shaders is established. For example, Compute Shader 2 is not executed until *Areas SSBO* and *Normals SSBO* are written by Compute Shader 1.

3.1.1. Compute Shader 1: Area and Normal Calculation

12 This shader is dispatched per triangle. Triangle area and normal are calculated and written to their respective locations in *Areas SSBO* and *Normals SSBO*.

3.1.2. Compute Shaders 2 & 3: Adaptive Prefix Sum

16 These shaders calculate the cumulative sum of triangle areas as illustrated in Figure 5. The prefix sum algorithm is employed for parallel sum operation.

19 *Parallel prefix sum.* Parallel prefix sum is an algorithm that leverages parallel processing capabilities to calculate the cumulative sum of an array [44, 45]. It is comprised of 2 parts: Up-sweep and down-sweep. In the up-sweep (reduce) phase partial sums are calculated and in the down-sweep (distribute) phase these partial sums are used to update values in the array. Figure

6 is provided for clarity. $A_{M..N}$ symbolizes the sum of elements from A_M to A_N .

Both parts of the prefix sum can be executed within a single compute shader. However, the local size of a compute shader is limited by *maxComputeWorkGroupInvocations* and this number depends on the hardware [46]. In our implementation *maxComputeWorkGroupInvocations* is 1024, meaning that a single compute shader can calculate the cumulative sum of a maximum 2048 triangles. This number is too small for the scenes used in our implementation. To overcome this constraint, prefix sum is divided into 2 compute shaders. Depending on the triangle number of the scene, the up-sweep and down-sweep shaders are dispatched multiple times.

Compute Shaders 2 & 3: Up- and Down-Sweep shaders. These shaders collectively compute the cumulative sum of triangle areas. Shader 2 first updates *Areas SSBO* with multiple partial sum values. Shader 3 writes the total sum into *Areas Sum SSBO*, then processes *Areas SSBO* to compute the cumulative sums. The operations of these shaders are exemplified in Figures 7 and 8 where the workgroup size is defined as 2 and *Areas SSBO* size is 16. Each color represents the elements on which a spe-

¹SSBO: shader storage buffer object [43].

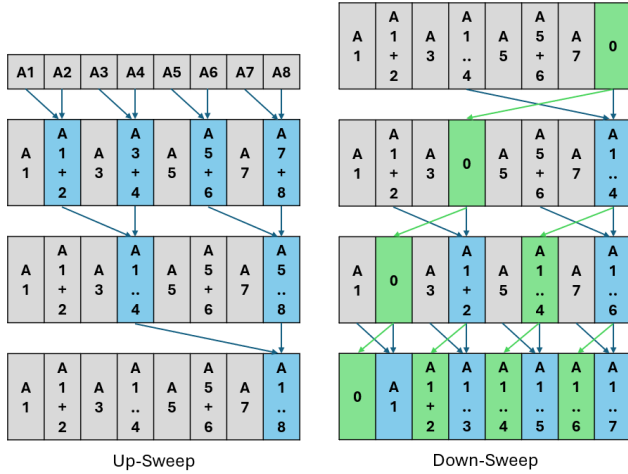


Fig. 6: Parallel prefix sum demonstration.

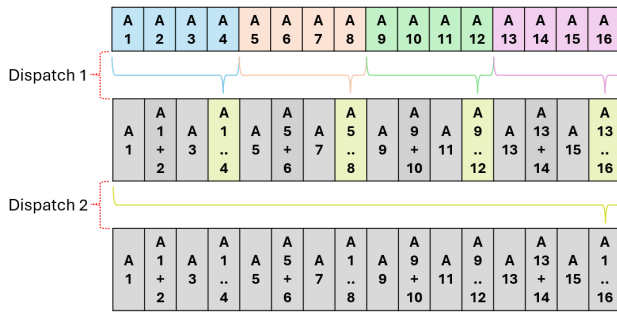


Fig. 7: Up Sweep shader dispatch example (shader 2).

Algorithm 1: Prefix sum parameter calculation

Input : N : # of triangles in the scene,
 $wgSize$: workgroup (wg) size (# of threads per wg)
Output: $passCnt$: # of dispatches,
 $wgCntPerPass$: buffer for the # of wgs per pass

- 1 $wgBufSize = 2 \times wgSize \triangleright$ two elements processed per thread
- 2 $passCnt = \lceil \log_{wgBufSize}(N) \rceil$
- 3 $bufferSize = \lceil \frac{N}{wgBufSize} \rceil \times wgBufSize \triangleright$ pad to $wgBufSize$
- 4 **for** $passCnt$ times **do**
- 5 append $(bufferSize/wgBufSize)$ into $wgCntPerPass$ buffer
- 6 $bufferSize = bufferSize/wgBufSize$
- 7 **end for**

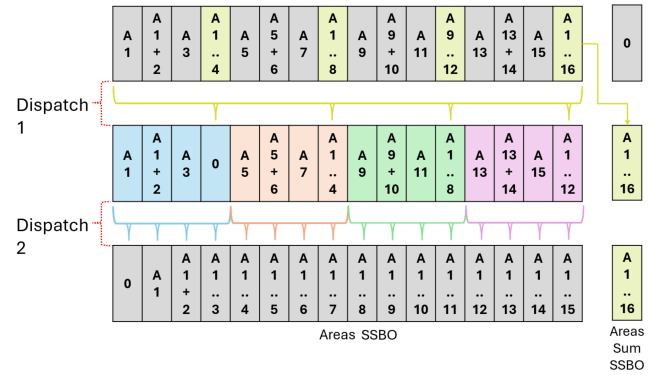


Fig. 8: Down Sweep shader dispatch example (shader 3).

cific invocation operates.

The number of dispatches and workgroups for each dispatch is determined as shown in Algorithm 1 before compute pipelines are constructed. In the example above, this algorithm computes 2 passes with 4 workgroups in the first pass and 1 in the second (these values are used in inverse order for shader 3). Each workgroup operates on the region of *Areas SSBO* based on their workgroup ID. After the first pass, shaders only operate on the indices of *Areas SSBO* that contain the results of the previous pass.

3.1.3. Compute Shader 4: DVF Generation

Preceding shaders act as a prerequisite for the actual DVF precomputation work carried out in compute shaders 4 and 5. In this shader, occlusion and visibility masks are created and written to an image array. For this purpose, first, the number of points to be sampled on the triangle is calculated. If this number is larger than a certain limit (meaning that the area of the triangle is significantly large), the workload is distributed among local invocations. Then, points are generated on the triangle with the Hammersley sequence or a hash function. For each point, N_r rays are generated using the Hammersley sequence. Again, the workload of tracing N_r rays is distributed among local invocations to speed up the process. If there is a

ray intersection, *imageAtomicOr()* function is used to modify the value in the corresponding location in the DVF. The details of these operations are provided below.

Uniform point sampling on mesh surface. A triangular mesh is comprised of multiple triangles. To sample points on a mesh surface uniformly, triangle areas must be taken into consideration. The number of points to sample per triangle can be found as:

$$N(T_k) = \left\lceil N_P \times \frac{A(T_k)}{\sum_{i=1}^K A(T_i)} \right\rceil, \quad (1)$$

where $N(T_k)$ is the number of points that must be sampled on triangle T_k , $A(T_k)$ is the area of triangle T_k , N_P is the number of total point samples, and K is the triangle count. $N(T_k)$ is calculated by dividing $A(T_k)$ by the total surface area and rounding the number up. By rounding the number up, at least 1 point sample is generated on each triangle. This leads to better coverage of the mesh surface than simply rounding the number, which may cause tiny triangles to be missed. After $N(T_k)$ is calculated, the coordinates of the sample points must be determined.

Random point generation on a triangle. A random point on the surface of a triangle with vertices (A, B, C) can be created by generating uniformly random values r_1 & r_2 in range $[0, 1)$ and applying Eqn. (2) [47].

$$P = (1 - \sqrt{r_1})A + \sqrt{r_1}(1 - r_2)B + \sqrt{r_1}r_2C \quad (2)$$

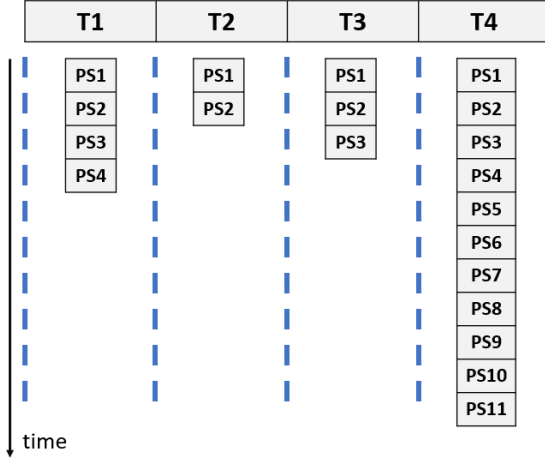


Fig. 9: Problem with large triangles. The workgroup operating on point samples PS1-PS11 of T4 will spend the longest time.

1 *Hash functions for generating random values.* To implement
 2 the method in the previous section, random numbers must be
 3 generated in the compute shader. One of the methods for gener-
 4 ating random numbers in GPU is using hash functions [48].
 5 Hash functions are favored for several reasons. They generate
 6 sufficiently different results for adjacent seed values, they are
 7 capable of handling multidimensional inputs and outputs, and
 8 they are efficient to compute. In our work, we used the *hash-*
 9 *withoutsine33* function provided by [48].

10 *Hammersley sequence for generating quasirandom values.* Al-
 11 ternative to hash functions, a low discrepancy sequence such as
 12 the Hammersley sequence can be used to generate quasirandom
 13 points on the surface of the triangle ensuring uniform cover-
 14 age [49]. The performance of hash functions vs. Hammersley
 15 sequence is compared in the results section.

16 *Distributing the workload for point samples.* Since the num-
 17 ber of points to be sampled depends on the triangle area, a
 18 considerable variation among triangle sizes in the mesh would
 19 cause significantly different workloads for different shader in-
 20 vocations. This leads to an imbalance in the execution times of
 21 different workgroups: the workgroups operating on the triangle
 22 with the large area will continue executing while other work-
 23 groups have completed their tasks and are idle. This problem
 24 can be observed in Figure 9 in which the size of triangle T4 is
 25 significantly larger than T1, T2, and T3.

26 To circumvent this problem, local invocations are utilized as
 27 illustrated in Figure 10. Local invocations work on the same
 28 triangle and the maximum allowed number of point samples
 29 per invocation is set as 3 for illustration purposes. When hash
 30 functions are used to generate random numbers, each invocation
 31 uses different index values as seed values. When the Ham-
 32 mersley sequence is used to generate quasirandom numbers, each
 33 invocation starts from its assigned place in the sequence. Thus,
 34 each invocation produces different point samples.

35 As shown in Figure 10, the total number of point samples for
 36 large triangles may slightly increase. However, this does not

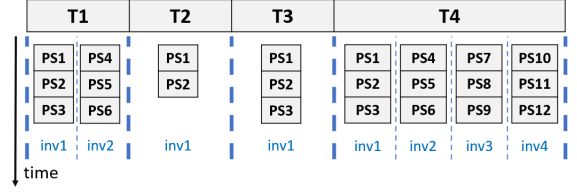


Fig. 10: Local invocations for large triangles.

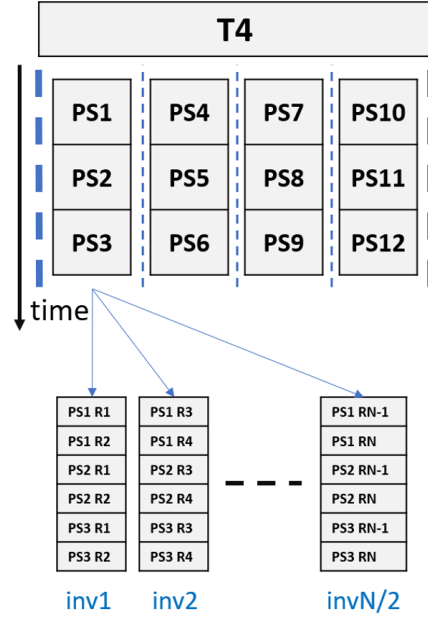


Fig. 11: Local invocations for ray tracing.

affect the outcome negatively since more point samples only in- 37
 crease the accuracy of the DVF without causing a performance 38
 difference. 39

Distributing the workload for ray samples. Each point sam- 40
 ple requires N_r ray samples generated with the Hammersley se- 41
 quence. If one invocation works on all N_r rays, the computa- 42
 tion would be highly serialized. Therefore, iterating over N_r 43
 rays must be avoided in compute shaders. This issue is solved 44
 by distributing the N_r rays among local invocations as shown in 45
 Figure 11. 46

Atomic operations for constructing DVF masks. The *DVF* 47
Masks Image Array is created by the CPU according to the oc- 48
 tahedral map size and grid dimensions. This array is cleared 49
 before Compute Shader 4 is dispatched. When invocations of 50
 Compute Shader 4 produce ray intersection results, they need 51
 to write them to the corresponding locations in this array. How- 52
 ever, when multiple invocations attempt to access the same lo- 53
 cation race conditions may occur. To avoid this problem *im-* 54
ageAtomicOr() function is utilized [46]. *DVF Masks Image Ar-* 55
ray is created in *VK_FORMAT_R32_UINT* format to be able to 56
 use this function. In fact, we only need two bits from this data 57
 type: bit 6 is used to represent the occlusion map and bit 7 is 58

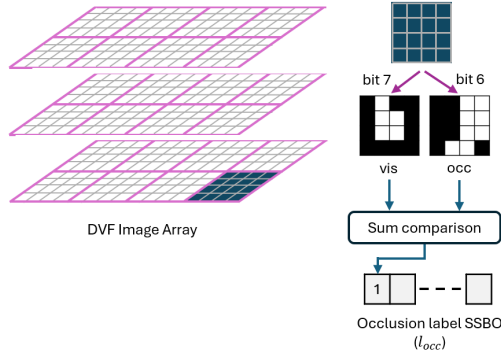


Fig. 12: Occlusion label SSBO construction.

NVIDIA GeForce RTX 4060 GPU, Intel i9-13900H CPU, and 16GB RAM.

4.1. Static Scene Results

In this section, we use the Sponza [51] and Amazon Luminary Bistro [52] scenes for presenting the results of our algorithm. Sponza is a relatively complex scene with 211,410 vertices and 262,267 triangles. The ground-truth version of the Sponza scene is generated using 128 ray samples per fragment without generating and using the DVF structure. Under this configuration, it can be rendered in our test system with 9 FPS. Amazon Bistro has significantly more geometric complexity with 2,892,045 vertices and 2,829,226 triangles, which are more than 10 times the corresponding values of the Sponza scene. The ground-truth version of the Amazon Bistro scene is generated using 64 ray samples per fragment without generating and using the DVF structure. Under this configuration, it can be rendered in our test system with 12 FPS. In the following, we first analyze the performance of our algorithm under different parameter configurations using the Sponza scene. We then compare our results with Yang et al. [18] in terms of computational performance. Finally, we provide a detailed timing analysis of our approach for both the Sponza and Amazon Bistro scenes.

4.1.1. Number of Point Samples

In this section, the rendering results are presented for scenes that are sampled with different numbers of point samples. As presented in Section 3.1.3, we propose to use the *ceiling* function to determine the number of point samples to generate per triangle. This ensures that each triangle is represented with at least one point sample. As a side-effect, it also increases the total number of samples. The method in [18] is more accurately replicated with the *round* function, which may cause small triangles to be skipped. We therefore show the rendering results using both approaches. As can be seen from Figure 13, the use of *ceiling* function improves image quality without imposing a significant performance cost. The difference is more noticeable for small sample counts such as 80K as this leads to a higher number of extra samples. It is also shown in this figure that the use of the hash function produces better results than the Hammersley sequence at approximately the same computational cost.

4.1.2. DVF Grid Resolution

The resolution of the DVF grid is one of the key parameters that affect the quality of the visibility approximation around surface points. Our evaluation of this parameter is shown in Figure 14. As can be seen from this figure, the performance in terms of FPS is lower for higher resolution grids – but it also drop for lower resolution ones. Also, the quality improves with reduced grid resolutions. These results may appear somewhat counter-intuitive, but they can be explained as below.

A higher grid resolution better approximates the visibility around each point – however, it also leads to a higher computational cost. Also, somewhat counter-intuitively, a higher resolution grid does not necessarily improve rendering quality.

used for the visibility map as shown in Figure 12.

3.1.4. Compute Shader 5: Calculation of Occlusion Labels

Occlusion labels are grid cell attributes and they can only be calculated once all occlusion and visibility masks are constructed. This shader is dispatched per grid cell and it performs parallel sum reduction to calculate the sum of occlusion masks and the sum of visibility masks of the cell. These values are then compared to calculate occlusion label l_{occ} . Specifically, if the number of occluded directions is larger than the number of visible directions 1 is written for the corresponding grid cell location in the *Occlusion label SSBO*. The process for one grid cell is illustrated in Figure 12.

3.2. Graphics Pipeline

Our graphics pipeline includes vertex and fragment shaders. Similar to Yang et al. [18], we use Vulkan ray tracing extension VK_KHR_ray_query to perform ray tracing within the fragment shader. The only difference from the method in [18] is the way the DVF is accessed in the fragment shader: In [18] the DVF is accessed through a texture whereas in our method it is accessed from an image array (*DVF Masks Image Array*) and occlusion labels are accessed from an SSBO (*Occlusion label SSBO*).

We use two types of synchronization mechanisms namely semaphores and pipeline barriers. Semaphores are used for synchronization between compute and graphics command buffer submissions to queues. Pipeline barriers are used for synchronization between different compute shaders and also to ensure that the fragment shader accesses *DVF Masks Image Array* and *Occlusion label SSBO* after their constructions are finalized.

4. Results & Validation

In this section, the results of our implementation are provided for both static and dynamic scenes. Similar to Yang et al. [18], we assumed all object materials are diffuse. The results are evaluated with respect to several metrics such as the FLIP [50], the root mean square error (RMSE), and the frame per second (FPS). The provided FPS values reflect the sum of precomputation and rendering durations. For error visualizations, *viridis* color map is used where the error is represented in purple - yellow scale. We performed all tests on a laptop system with an


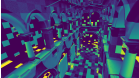
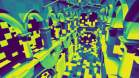

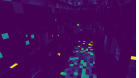





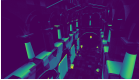

























	Our Result	Absolute Error	\mathcal{H} LIP	Our Result	Absolute Error	\mathcal{H} LIP	Our Result	Absolute Error	\mathcal{H} LIP
80K	 73 FPS	 RMSE: 0.330940	 Mean: 0.499832	 58 FPS	 RMSE: 0.113394	 Mean: 0.090513	 61 FPS	 RMSE: 0.109425	 Mean: 0.145041
400K	 66 FPS	 RMSE: 0.209978	 Mean: 0.215490	 56 FPS	 RMSE: 0.013245	 Mean: 0.022700	 58 FPS	 RMSE: 0.028231	 Mean: 0.054061
2M	 59 FPS	 RMSE: 0.060940	 Mean: 0.043503	 55 FPS	 RMSE: 0.005040	 Mean: 0.006104	 56 FPS	 RMSE: 0.012378	 Mean: 0.017519
10M	 54 FPS	 RMSE: 0.003100	 Mean: 0.002501	 54 FPS	 RMSE: 0.002604	 Mean: 0.001792	 54 FPS	 RMSE: 0.003515	 Mean: 0.003755
	Round with hash			Ceiling with hash			Ceiling with Ham.		

Fig. 13: Left: Rounding approach used with the hash function. Middle: Ceiling approach with the hash function. Right: Ceiling with the Hammersley sequence. The left-most column shows the number of point samples.





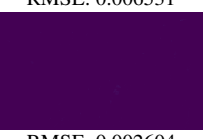
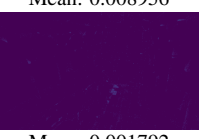

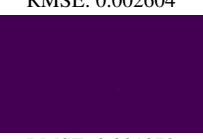
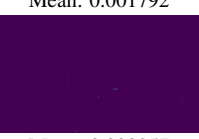


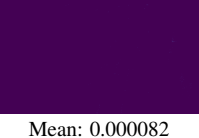


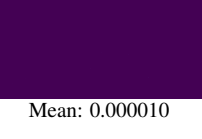
	Our Result	Absolute Error	\mathcal{H} LIP
155x96x70	 30 FPS	 RMSE: 0.006531	 Mean: 0.008956
77x48x34	 54 FPS	 RMSE: 0.002604	 Mean: 0.001792
40x25x18	 57 FPS	 RMSE: 0.001072	 Mean: 0.000357
21x14x10	 49 FPS	 RMSE: 0.000423	 Mean: 0.000082
12x8x6	 37 FPS	 RMSE: 0.000120	 Mean: 0.000010

Fig. 14: The effect of different grid dimensions. Triangle point sampling is performed using the hash function. 10M point samples are used.

tions are also occluded, all 128 rays will have to be traced. This means the DVF loses its ability to reduce the number of rays that must be traced for each point, causing both a drop in performance and a rise in quality.

Furthermore, for high resolution grids, memory management will be more complex due to the limited cache size of compute units. Even though the number of compute shaders that try to access the same pixel will be lower, each compute shader will try to access different parts of the image array that are far from each other in terms of their memory location. When grid dimensions are smaller, cache hits will be more probable but as each compute shader will try to access the similar parts of the image array atomic operations may cause serialization as explained in Section 3.1.3.

This means that for each scene there could be a different ideal grid resolution and this resolution should be neither too low nor too high. For the Sponza scene shown in Figure 14, the $40 \times 25 \times 18$ resolution appears to yield the best trade-off.

4.1.3. Number of Ray Samples

As expected and can be seen from Figure 15, the rendering accuracy increases in lockstep with the number of ray samples. The performance drop is not linear, however, and with 128 ray samples high fidelity renderings can be obtained.

4.1.4. Number of Invocations per Point Sample

In this section rendering results are presented for cases where different numbers of invocations per point sample are invoked. If N_r ray samples have to be traced for a point sample, N_{inv} local invocations work on the different rays of the same point sample so that N_{inv} rays can be traced simultaneously. As shown in Figure 16, the rendering speed increases with the number of parallel invocations per point sample. The accuracy of the result is similar in all cases – only a minor variation is observed due to the pseudo-random number generation process.

4.2. Comparison with Yang et al. [18]

In [18], the authors perform precomputation on the CPU, which precludes the possibility of animation or geometry

This can be understood if we consider a $1 \times 1 \times 1$ grid, whose sole voxel encompasses the entire scene. In such a case, the number of occluded and visible directions will be very large and most likely equal as they are computed from all point samples scattered throughout the scene. The fragment shader uses the DVF to reduce the number of rays it needs to trace. It traces rays in occluded directions when the surface point is mostly visible, and it traces rays in visible directions when the surface point is mostly occluded (see Figure 1). In the extreme case, the occlusion label is determined to be 0, which means the fragment shader will trace rays in occluded directions. Since all direc-

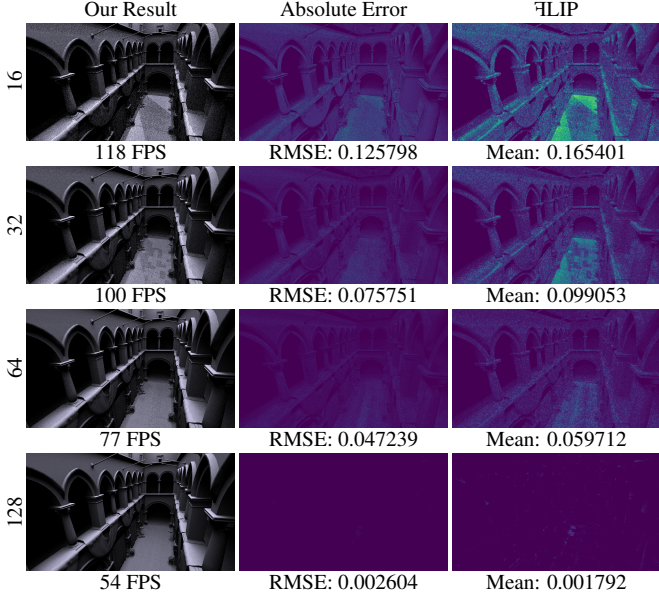


Fig. 15: Different number of ray samples. Triangle point sampling is performed using the hash function. Results are reported for 10M point samples.

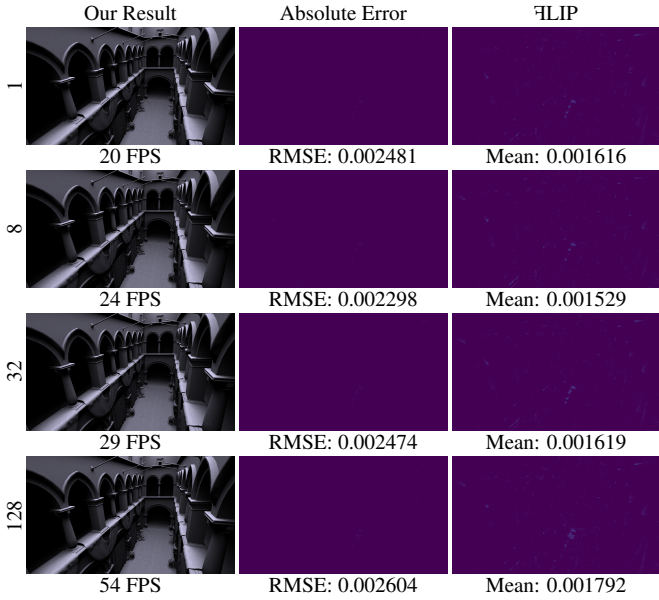


Fig. 16: The number of parallel execution of ray samples for each point sample. The total amount of ray samples is the same for each case. Triangle point sampling is performed using the hash function. Results are reported for 10M point samples.

Table 2: Frame durations (ms) with different number of point samples, including both precomputation and rendering for the Sponza scene.

Point sample count	80K	400K	2M	10M
Yang et al. - Precomputation [18]	1.651	3.279	11.187	50.558
Yang et al. - Rendering [18]	11.41	13.07	14.69	15.74
Ours - Precomputation	3.89	3.87	3.82	3.79
Ours - Rendering	13.35	13.97	14.42	14.72
Ours - Total	17.24	17.84	18.24	18.51

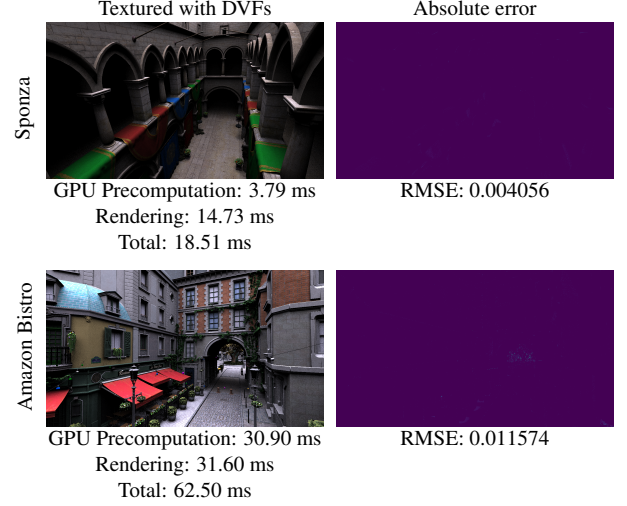


Fig. 17: Left: textured versions of the Sponza and Amazon Bistro scenes rendered using our per-frame computation of DVFs. Right: corresponding RMSE maps. 10M point samples are used.

DVF is accessed. In [18], the DVF and occlusion labels are accessed through textures, which are computed offline and uploaded to the GPU memory. In our case, the DVF is created for each frame in real-time and its visibility and occlusion masks are accessed through the *DVF masks image array*. The occlusion labels are read from the *occlusion label SSBO*.

It can be seen from these results that the CPU implementation of the DVF algorithm is time consuming. It takes about 1.5 seconds for 80K samples and it goes up to 50.5 seconds for 10M samples. We note that this high number of samples is required for producing accurate rendering results (see Figure 13). As for the run-time performance, Yang et al.'s algorithm takes approximately 15 ms for this high sample count, whereas our algorithm takes about 18 ms including precomputation.

4.2.1. Detailed Timing Analysis

In this section, we further analyze the computational complexity of our algorithm by showing the time taken by each compute shader stage. For accurate time measurement, the compute pipelines are flushed and synchronized. This causes the total time to be more than what it would be without these additional synchronization points. The time taken by each compute shader stage for Sponza and Amazon Bistro scenes with 10M point samples are shown in Table 3. For Sponza, it can be seen that CS5 takes the majority of the time. The fact that CS5 takes more time than CS4 can be explained by the fact that CS5 is executed per grid cell and is therefore parallelized to a lesser

changes during the rendering process. By comparing our timing results with this work, we aim to shed light on (i) how much CPU work is avoided and (ii) how much run-time cost is incurred as a result of doing the precomputation for each frame. These results are summarized in Table 2 for different numbers of point samples. The precomputation timings for the reference work are taken from [18], who used Intel Core i9-10980XE CPU to implement their algorithm on the CPU. The run-time algorithm of [18] is implemented using the graphics pipeline on NVIDIA GeForce RTX 3080. The only difference between this implementation from our run-time approach is the way the

Table 3: Time taken by each shader stage (in ms) for Sponza and Amazon Bistro scenes using 10M samples.

Shader Stage	Sponza (ms)	Amazon Bistro (ms)
CS1: Area and Normal Calculation	0.084	0.862
CS2: Adaptive Prefix Sum (Upsweep)	0.583	6.486
CS3: Adaptive Prefix Sum (Downsweep)	0.582	6.253
CS4: DVF Generation	0.683	10.765
CS5: Calculation of Occlusion Labels	1.858	6.533
Total	3.793	30.899

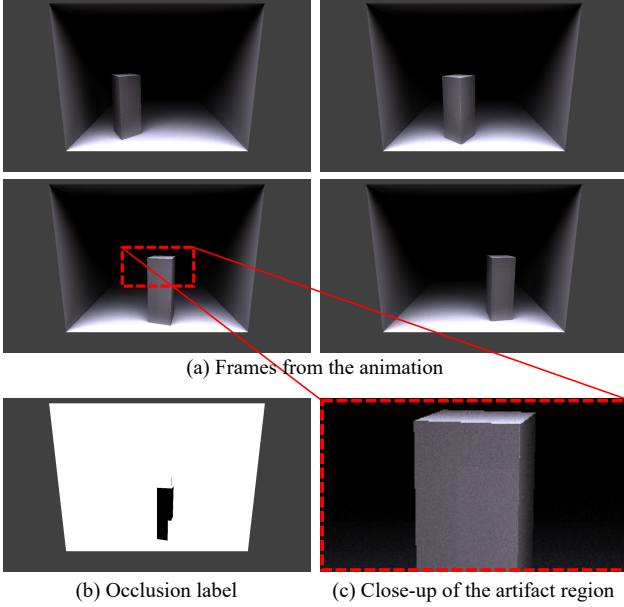


Fig. 18: Several frames from the dynamic Cornell box scene. The bottom-left frame from the animation is shown together with its occlusion label. Note the small discontinuity artifact on the object's surface due to the change of the occlusion label. See text for details.

degree than CS4, which is executed per triangle. However, for the Amazon Bistro scene which has 2.8M triangles compared to 262K in Sponza, CS4 takes the majority of the time. Although the grid resolution is also increased to $111 \times 118 \times 34$ from $77 \times 48 \times 34$ of Sponza, the higher increase in triangle count appears to dominate the computation times. The computation times of other shaders such as CS1, CS2, and CS3 that also depend on the triangle count appear to increase linearly with the triangle count as well. This evaluation indicates that for a very complex scene such as Amazon Bistro, we can perform the entire precomputation process with more than 30 frames per second.

The textured versions of these scenes rendered using our per-frame computation of DVFs together with their error maps are shown in Figure 17 for 10M point samples.

4.3. Dynamic scene performance

As our algorithm is designed to enable real-time environment lighting for dynamic scenes, we show the applicability of our algorithm for such a setup using a Cornell box-like scene in which a rectangular prism orbits around the center of the scene. The reference rendering for this scene is obtained without using the DVF structure with 128 environment lighting rays per

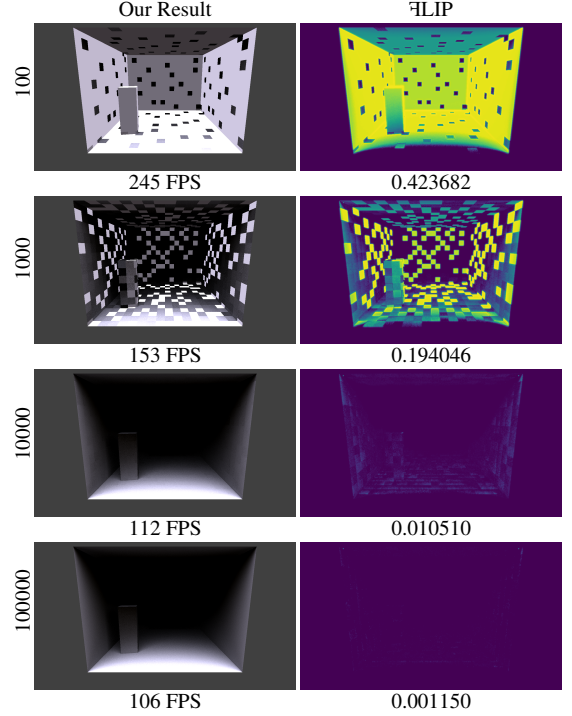


Fig. 19: The effect of different number of point samples. Hammersley sequence is used for generating the point samples.

fragment. This baseline implementation yields 44 FPS, which can be considered high enough for real-time rendering. However, we show that much higher frame rates can be obtained using our approach. For the results presented in this section, we use the Hammersley low discrepancy sequence for generating random numbers. This is because we experimentally observed that the hash function produces inferior results when used with meshes with low triangle count (see Discussion). Our rendering results for different positions of the orbiting prism are shown in Figure 18. Several recorded animations can be seen in supplementary materials.

The effect of different numbers of point samples is illustrated in Figure 19. Here, artifacts can be clearly observed for low sample counts as this leaves many surface regions to be under-represented. High quality results can be obtained with a mean FLIP value of 0.001150 using 1M point samples, in which case the corresponding FPS becomes 106. This can be considered a noticeable improvement over the ground-truth frame rate of 44 with negligible effect on image quality. These results are obtained with a DVF grid resolution of $16 \times 16 \times 16$, with 128 ray samples per point sample, and 128 parallel ray invocations per point sample. As the results of evaluation with respect to these parameters follow a similar trend to the static case, they are left out for brevity in this paper. A detailed timing analysis of the shader stages is provided in Table 4.

4.4. Animation Artifacts

One problem that we observed during animation was occasional artifacts that occurred when a part of an animated object moved between two grid cells that have different occlusion labels. This changes the irradiance computation from visible

Table 4: Time taken by each shader stage (in ms) for Cornell box scene using 100K samples.

Shader Stage	Cornell Box (ms)
CS1: Area and Normal Calculation	0.006
CS2: Adaptive Prefix Sum (Upsweep)	0.005
CS3: Adaptive Prefix Sum (Downsweep)	0.005
CS4: DVF Generation	0.909
CS5: Calculation of Occlusion Labels	0.056
Total	0.9306

irradiance (if the grid cell is mostly occluded) to subtracted irradiance (if the grid cell is mostly visible). Because smooth continuity between two such irradiance values cannot be ensured, this can lead to discontinuity and flickering artifacts (see Figure 18 (c)). These artifacts can be mitigated by increasing the number of point samples as shown by example videos in supplementary materials. We also note that when textures and more complex models and animations are used, as opposed to purely diffuse objects with low polygon count, the visibility of such artifacts is likely to be visually masked [53].

5. Discussion

After examining the effect of various parameters, the following overall conclusions can be drawn.

Hammersley Sequence vs. Hash Function. For the Sponza scene, which consists of many small triangles, the difference between the hash function and the Hammersley sequence was negligible with a slight advantage for the hash function. This could be attributed to the fact that the Hammersley sequence is not random: it is a well-distributed low discrepancy sequence, but when only a few samples are selected on each triangle they tend to be selected from the same relative positions – which is not a problem if more samples are selected per triangle as in the Cornell box scene. In fact, the Cornell box scene confirms this observation for which the hashing based randomization is inferior to the low-discrepancy one. This is because the hash function requires seed values to generate random numbers. In this study, these seed values are harvested from the vertex indices of the triangles. Since simple scenes, such as the Cornell box, have relatively few vertices with repeated indices, generating random points using the hash function does not produce well-distributed point samples. It can therefore be concluded that the Hammersley sequence is overall a better choice for generating the point samples over the surfaces, especially for low polygon models.

Ideal Grid Resolution. As discussed in Section 4.1.2, the DVF grid resolution should neither be too small nor too large. The best results are obtained with intermediate grid resolutions, which need to be tuned for each scene. We found the optimal grid resolution as $40 \times 25 \times 18$ for the Sponza scene and $16 \times 16 \times 16$ for the Cornell box scene.

Compute Shader Subdivision. As discussed in Section 3.1.3, Compute Shader 4 carries out many different operations and it can be considered as our most complex shader. We therefore investigated whether its performance can be improved by subdividing it into two shaders as A and B. In this

setup, Compute Shader 4A would generate the point samples on triangles and write their coordinates to an SSBO. Compute Shader 4B would be dispatched per point sample with N_r (number of ray samples) local invocations to trace the rays and write the results into *DVF Masks Image Array*. While we considered this as a logical subdivision, experimental results produced inferior performance with this approach. This is because all invocations of Compute Shader 4B were trying to access the same image array locations simultaneously using atomic OR operations, leading to congestion.

When this subdivision is not done, the congestion is reduced because invocations first generate the point samples and then trace rays before writing the result to the image array. Since Compute Shader 4 is dispatched per triangle and point sample generation duration is different for each triangle, the invocations get jittered on the time domain reducing conflicts.

Pipelining Compute and Graphics Tasks. In our implementation, compute and graphics tasks are executed sequentially (Figure 3). In practice, their execution can be pipelined by ensuring that compute shaders always work one frame ahead of graphics shaders, potentially leading to a better performance.

Culling. In the original DVF algorithm, precomputation has to be done for the entire scene as the scene can be viewed from different camera positions and orientations during runtime. However, in our dynamic DVF implementation, it is possible to limit the precomputation to only visible objects in the current frame. This can be achieved by applying frustum and back-face culling inside compute shaders to reduce the number of processed triangles. We leave the exploration of this approach as future work.

GPU Based Animation. In this study, we animate the Cornell box model using a simplified approach for a proof-of-concept implementation: the geometry of the orbiting box is updated on the CPU and the results are written to the *Vertices SSBO*. Consequently, the top- and bottom-level acceleration structures (TLAS and BLAS) are also updated since they contain the scene geometry information (see Figure 4). This approach does not affect the performance evaluation since both the ground-truth and our results are obtained in the same manner. To achieve more complex animations in real-time, a compute shader can be utilized to update the vertices in the GPU.

6. Conclusion

We presented a GPU-based precomputation algorithm that improves upon the work presented in [18] to support rendering of dynamic scenes. The key contribution of our work was to show how the relatively complex precomputation tasks can be divided into different compute shader stages. We show that this is not a trivial task and different parallelization schemes and parameters yield different performance. We conducted additional tests to evaluate the performance-quality trade-off of our approach for different parameter configurations. Finally, we examined the performance of real-time rendering of dynamic scenes by utilizing a proof-of-concept animation using a Cornell box-like scene. We believe that the improved runtime performance afforded by our algorithm can make realistic

Algorithm 2: Precomputation of the DVF [18].

Input : surface mesh M , N_r ray samples $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_{N_r}$
Output: precomputed DVF \mathcal{V}

```

1 create  $N_p$  point samples  $s_1, s_2, \dots, s_{N_p}$  on  $M$ 
2 create octahedral map  $O_s^V(s)$  storing  $V_s(s, \omega)$  for each  $s$ 
3 create octahedral map  $O_s^O(s)$  storing  $O_s(s, \omega)$  for each  $s$ 
4 set all pixels of  $O_s^V(s)$  and  $O_s^O(s)$  as 0
  // Create masks of point samples
5 foreach point sample  $s = (p, n)$  do
6   foreach ray  $\mathcal{R} = (p, \omega)$  do
7     randomly rotate  $\mathcal{R}$  along  $n$ 
8     project  $\omega$  to unit square using octahedral mapping
9     if  $\mathcal{R}$  intersects scene geometry then
10      set the corresp. pixel in  $O_s^V(s)$  as 1
11    else
12      set the corresp. pixel in  $O_s^O(s)$  as 1
13    end if
14  end foreach
15  dilate  $O_s^V(s)$  by one pixel
16  dilate  $O_s^O(s)$  by one pixel
17 end foreach
18 construct a uniform grid  $\mathcal{G}$  for  $M$ 
19 create octahedral map  $O_c^V(c)$  for each grid cell  $c$ 
20 create octahedral map  $O_c^O(c)$  for each grid cell  $c$ 
21 set all pixels of  $O_c^V(c)$  and  $O_c^O(c)$  as 0
  // Merge masks inside a grid cell
22 foreach point sample  $s = (p, n)$  do
23   locate the grid cell  $c$  by  $p$ 
24   foreach pixel in the octahedral maps of  $c$  do
25      $V_c(c, \omega) \leftarrow V_c(c, \omega) \vee V_s(s, \omega)$ 
26      $O_c(c, \omega) \leftarrow O_c(c, \omega) \vee O_s(s, \omega)$ 
27   end foreach
  // Compute occlusion labels
28 foreach grid cell  $c$  in  $\mathcal{G}$  do
29    $r_{occ}(c) = \frac{N_{occ}(c)}{N_{occ}(c) + N_{vis}(c)}$  if  $r_{occ}(c) > 0.5$  then
30      $l_{occ}(c) = 1$ 
31   else
32      $l_{occ}(c) = 0$ 
33   end if
34 end foreach

```

1 and real-time environment lighting for complex and dynamic
2 scenes a tangible possibility.

3 Appendix A. Precomputation and Runtime Algorithms 4 used by Yang et al. [18]

5 In this section, we provide both of the algorithms used by
6 Yang et al.'s work for the sake of completeness. In our work,
7 we replace the precomputation algorithm with a GPU-based im-
8 plementation while using the runtime algorithm as-is.

9 References

- 10 [1] Blinn, JF, Newell, ME. Texture and reflection in computer generated
11 images. Communications of the ACM 1976;19(10):542–547.
12 [2] Pharr, M, Jakob, W, Humphreys, G. Physically based rendering: From
13 theory to implementation. MIT Press; 2023.
14 [3] Lehtinen, J, Zwicker, M, Turkkanen, J, Durand, F, Sillion, FX, et al. A meshless hierarchical representation for light transport. ACM
15 Trans Graph 2008;27(3):1–9.
16 [4] Greger, G, Shirley, P, Hubbard, P, Greenberg, D. The irradiance vol-
17 ume. IEEE Computer Graphics and Applications 1998;18(2):32–43.
18 [5] McGuire, M, Mara, M, Nowrouzezahrai, D, Luebke, D. Real-time
19 global illumination using precomputed light field probes. In: Proc. of
20 the 21st ACM SIGGRAPH symposium on interactive 3D graphics and
21 games. 2017, p. 1–11.
22 [6] Seyb, D, Sloan, PP, Silvennoinen, A, Iwanicki, M, Jarosz, W. The
23 design and evolution of the uberbake light baking system. ACM Trans
24 Graph 2020;39(4).

Algorithm 3: Rendering using the precomputed DVF [18].

Input : precomputed DVF \mathcal{V} , incoming radiance of environment map $L_i(\omega)$,
fragment position x , fragment normal n , N_r randomly rotated ray
samples $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_{N_r}$
Output: incoming irradiance $E(x)$

```

1  $E(x) \leftarrow 0$ 
2  $E_{occ}(x) \leftarrow 0$ 
3 locate the grid cell  $c$  by  $x$ 
4 fetch occlusion label  $l_{occ}(c)$  from  $\mathcal{V}$ 
5 foreach ray  $\mathcal{R} = (x, \omega)$  do
6   project  $\omega$  to unit square using octahedral mapping
7   fetch visibility and occlusion masks  $V_c(c, \omega)$ ,  $O_c(c, \omega)$ 
  // More than half pixels are occluded
8   if  $l_{occ}(c) = 1$  then
9     // Trace inside visible region
10    if  $V_c(c, \omega) = 1$  then
11      trace  $\mathcal{R}$ 
12      if  $\mathcal{R}$  does not intersect scene geometry then
13         $E(x) \leftarrow E(x) + \frac{1}{N_r} L_i(\omega)$ 
14      end if
15    end if
  // Trace inside occluded region
16    if  $O_c(c, \omega) = 1$  then
17      trace  $\mathcal{R}$ 
18      if  $\mathcal{R}$  intersects scene geometry then
19         $E_{occ}(x) \leftarrow E_{occ}(x) + \frac{1}{N_r} L_i(\omega)$ 
20      end if
21    end if
22  end if
23  if  $l_{occ}(c) = 0$  then
24    evaluate  $E_{unshad}(n)$  by  $n$  and  $\mathcal{E}$ 
25     $E(x) \leftarrow E_{unshad}(n) - E_{occ}(x)$ 
26  end if
27 end foreach

```

- [7] Sloan, PP, Kautz, J, Snyder, J. Precomputed radiance transfer for real-
time rendering in dynamic, low-frequency lighting environments. ACM
Trans Graph 2002;21(3):527–536.
[8] Zhou, K, Hu, Y, Lin, S, Guo, B, Shum, HY. Precomputed shadow
fields for dynamic scenes. ACM Trans Graph 2005;24(3):1196–1201.
[9] Sloan, PP, Hall, J, Hart, J, Snyder, J. Clustered principal components for
precomputed radiance transfer. ACM Trans Graph 2003;22(3):382–391.
[10] Jendersie, J, Kuri, D, Grosch, T. Precomputed illuminance composition
for real-time global illumination. In: Proc. of the 20th ACM SIGGRAPH
Symposium on Interactive 3D Graphics and Games. 2016, p. 129–137.
[11] Öztürk, B, Akyüz, AO. Realistic lighting for interactive applica-
tions using semi-dynamic light maps. The Computer Games Journal
2020;9(4):421–452.
[12] Clarberg, P, Akenine-Möller, T. Exploiting visibility correlation in direct
illumination. In: Computer Graphics Forum; vol. 27. Wiley Online
Library; 2008, p. 1125–1136.
[13] Airey, JM, Rohlf, JH, Brooks, FP. Towards image realism with inter-
active update rates in complex virtual building environments. In: Pro-
ceedings of the 1990 Symposium on Interactive 3D Graphics. I3D '90;
New York, NY, USA: Association for Computing Machinery. ISBN
0897913515; 1990, p. 41–50.
[14] Durand, F, Drettakis, G, Thollot, J, Puech, C. Conservative visibility
preprocessing using extended projections. In: Proceedings of the 27th An-
nual Conference on Computer Graphics and Interactive Techniques. SIG-
GRAPH '00; USA: ACM Press/Addison-Wesley Publishing Co. ISBN
1581132085; 2000, p. 239–248.
[15] Popov, S, Georgiev, I, Slusallek, P, Dachsbacher, C. Adaptive quanti-
zation visibility caching. In: Computer Graphics Forum; vol. 32. Wiley
Online Library; 2013, p. 399–408.
[16] Guo, JJ, Eisemann, M, Eisemann, E. Next event estimation++: visibility
mapping for efficient light transport simulation. In: Computer Graphics
Forum; vol. 39. Wiley Online Library; 2020, p. 205–217.
[17] Schaffler, G, Dorsey, J, Decoret, X, Sillion, FX. Conservative volumetric
visibility with occluder fusion. In: Proc. of the 27th annual conference
on Computer graphics and interactive techniques. 2000, p. 229–238.

- [18] Xu, Y, Jiang, Y, Wang, C, Li, K, Zhou, P, Geng, G. Precomputed Discrete Visibility Fields for Real-Time Ray-Traced Environment Lighting. In: Ghosh, A, Wei, LY, editors. Eurographics Symposium on Rendering. The Eurographics Association. ISBN 978-3-03868-187-8; 2022,doi:10.2312/sr.20221158.
- [19] Burgess, J. Rtx on—the nvidia turing gpu. IEEE Micro 2020;40(2):36–44.
- [20] Group, K. Opengl wiki. <https://www.khronos.org/blog/ray-tracing-in-vulkan>; 2020. (Last accessed on 05/01/2024).
- [21] Greene, N. Environment mapping and other applications of world projections. IEEE computer graphics and Applications 1986;6(11):21–29.
- [22] Heidrich, W, Seidel, HP. View-independent environment maps. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware. 1998, p. 39–ff.
- [23] Debevec, P. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In: SIGGRAPH 98 Conf. Proc. New York, NY, USA: ACM; 1998, p. 189–198.
- [24] Kollig, T, Keller, A. Efficient illumination by high dynamic range images. Fachbereich Informatik, Univ.; 2002.
- [25] Debevec, P. Light probe image gallery. <https://www.pauldebevec.com/Probes/>; 2004. (Last accessed on 19/12/2023).
- [26] Schäfer, H, Süßmuth, J, Denk, C, Stamminger, M. Memory efficient light baking. Computers & Graphics 2012;36(3):193–200.
- [27] Jakob, W. Mitsuba renderer. 2010. Accessed 22 April 2023.
- [28] Pharr, M, Jakob, W, Humphreys, G. Physically Based Rendering: From Theory to Implementation. 4 ed.; The MIT Press; 2023.
- [29] Öztürk, B, Akyüz, AO. Semi-dynamic light maps. In: ACM SIGGRAPH 2017 Posters. 2017, p. 1–2.
- [30] Seyb, D, Sloan, PP, Silvennoinen, A, Iwanicki, M, Jarosz, W. The design and evolution of the uberbake light baking system. ACM Transactions on Graphics (TOG) 2020;39(4):150–1.
- [31] Sillion, FX, Arvo, JR, Westin, SH, Greenberg, DP. A global illumination solution for general reflectance distributions. In: Proceedings of the 18th annual conference on Computer graphics and interactive techniques. 1991, p. 187–196.
- [32] Ramamoorthi, R, Hanrahan, P. An efficient representation for irradiance environment maps. In: Proc. of the 28th annual conference on Computer graphics and interactive techniques. 2001, p. 497–500.
- [33] Tsai, YT, Shih, ZC. All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation. ACM Transactions on graphics (TOG) 2006;25(3):967–976.
- [34] Kristensen, AW, Akenine-Möller, T, Jensen, HW. Precomputed local radiance transfer for real-time lighting design. In: ACM SIGGRAPH 2005 Papers. ACM; 2005, p. 1208–1215.
- [35] Scherzer, D, Wimmer, M, Purgathofer, W. A survey of real-time hard shadow mapping methods. In: Computer graphics forum; vol. 30. Wiley Online Library; 2011, p. 169–186.
- [36] Ben-Artzi, A, Ramamoorthi, R, Agrawala, M. Efficient shadows for sampled environment maps. Jnl of Graphics Tools 2006;11(1):13–36.
- [37] Debevec, P. A median cut algorithm for light probe sampling. In: ACM SIGGRAPH 2008 classes. ACM; 2008, p. 1–3.
- [38] Guo, JJ, Eisemann, M, Eisemann, E. Next event estimation++: visibility mapping for efficient light transport simulation. Computer Graphics Forum 2020;39(7):205–217.
- [39] Hammersley, JM. Monte carlo methods for solving multivariable problems. Annals of the New York Academy of Sciences 1960;86(3):844–874.
- [40] Cigolle, ZH, Donow, S, Evangelakos, D, Mara, M, McGuire, M, Meyer, Q. A survey of efficient representations for independent unit vectors. Journal of Computer Graphics Techniques 2014;3(2).
- [41] Peters, C. Free blue noise texture. 2016. Accessed 28 April 2023.
- [42] Koch, D, Hector, T, Barczak, J, Werness, E. Ray tracing in vulkan. <https://www.khronos.org/blog/ray-tracing-in-vulkan>; 2020. (Last accessed on 05/01/2024).
- [43] Group, K. Shader storage buffer object. https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object; 2020. (Last accessed on 31/12/2023).
- [44] Blelloch, GE. Prefix sums and their applications. 1990,URL: <https://api.semanticscholar.org/CorpusID:60459178>.
- [45] Harris, M. Parallel prefix sum (scan) with cuda. GPU Gems 2007;3.
- [46] Group, K. Khronos registry. <https://registry.khronos.org/>; 2022. (Last accessed on 01/01/2024).
- [47] Osada, R, Funkhouser, T, Chazelle, B, Dobkin, D. Shape distributions. ACM Transactions on Graphics (TOG) 2002;21(4):807–832.
- [48] Jarzynski, M, Olano, M. Hash functions for gpu rendering. Journal of Computer Graphics Techniques (JCGT) 2020;9(3):20–38.
- [49] Dalal, IL, Stefan, D, Harwayne-Gidansky, J. Low discrepancy sequences for monte carlo simulations on reconfigurable platforms. In: 2008 International Conference on Application-Specific Systems, Architectures and Processors. IEEE; 2008, p. 108–113.
- [50] Andersson, P, Nilsson, J, Akenine-Möller, T. Visualizing and Communicating Errors in Rendered Images. In: Marrs, A, Shirley, P, Wald, I, editors. Ray Tracing Gems II; chap. 19. 2021, p. 301–320.
- [51] Group, K. gltf sample models, sponza. <https://github.com/KhronosGroup/gltf-Sample-Models/tree/master/2.0/Sponza>; 2023.
- [52] Lumberyard, A. Amazon lumberyard bistro, open research content archive (orca). 2017. URL: <http://developer.nvidia.com/orca/amazon-lumberyard-bistro>.
- [53] Ferwerda, JA, Shirley, P, Pattanaik, SN, Greenberg, DP. A model of visual masking for computer graphics. In: Proc. of the 24th annual conference on Computer graphics and interactive techniques. 1997, p. 143–152.