Chunked Caching for Disocclusion Computation and Frame Extrapolation

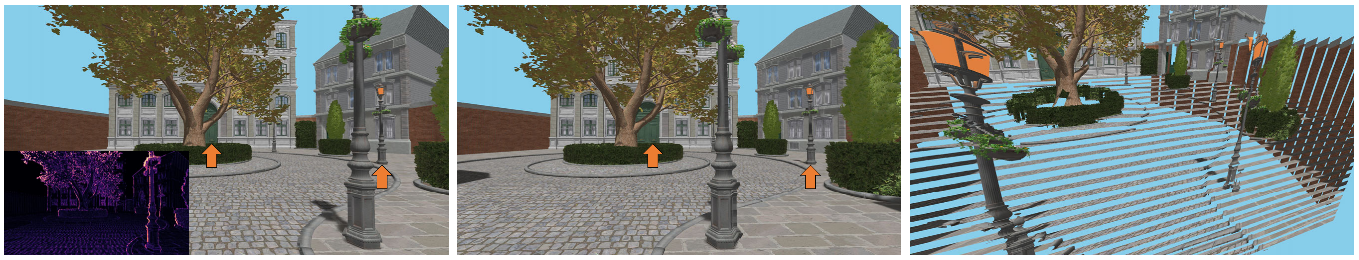

Fig. 1. The images on the left and in the middle show two views extrapolated from the same chunked cache. The cache contains the shading for a potentially visible set computed for an extremely large viewcell with a diameter of more than two meters. The orange arrows show how the tree and the lantern move in the image plane, leading to strong disocclusions successfully filled in from our cache. The image on the right visualizes the cache layers without reprojection. The inset (bottom-left) demonstrates the high quality of frame extrapolation by comparing the extrapolated image to the ground truth with the FLIP metric [2].

We introduce a chunked caching architecture that replaces a conventional linear buffer (e.g., a framebuffer, G-buffer or visibility buffer) with a set of chunks, i.e., sparse "pixel tiles" organized in a layered image space. The cache can represent arbitrary portions of the scene. Consequently, we can use chunked caching for efficient scene processing algorithms without using the original complex scene geometry. We demonstrate how this approach can be used to generate a from-region potentially visible set (PVS) by explicitly computing disocclusions (rather than occlusions) in the scene. Our PVS algorithm is order-independent and 6.9× faster than the state of the art at the same level of accuracy. Shading needs to be computed only for the PVS and can be directly used for frame extrapolation without having to worry about disocclusions. Chunked caching runs as a pure software implementation on the GPU without requiring any hardware extensions. We demonstrate how our work outperforms previous PVS and frame extrapolation methods in speed and extrapolation range without compromising quality.

ACM Reference Format:

. 2025. Chunked Caching for Disocclusion Computation and Frame Extrapolation. In *Proceedings of SIGGRAPH 2025*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The demand for graphics throughput rises continuously. For example, current-generation game consoles operating at 4K/60 Hz require 8× more pixels than the previous generation operating at 1080p/30 Hz. GPU improvements face cost and power constraints and cannot trivially keep up with this increase in demand. Exploiting spatiotemporal coherence is a possible way out of this dilemma. Therefore, various forms of spatiotemporal upsampling, such as DLSS [38], are now commonly used in practice.

As a conceptual model for exploiting spatiotemporal coherence, Ragan-Kelly et al. [39] proposed choosing separate sampling frequencies [33] for visibility and shading. Shading sampling is by far the most expensive task, but visibility sampling needs to be determined at much higher spatial and temporal rates. For example, deferred rendering [41] restricts shading computations to only visible samples of a frame, and reverse reprojection caching [36] combines the previous frame buffer with the current one to amortize supersampled shading computations. Both methods rely on a conventional linear (i.e., flat and dense) buffer to cache the visibility/shading computations for later reuse.

Unfortunately, a conventional framebuffer does not store enough information about the scene geometry to support more demanding applications, such as extrapolation over multiple frames. With novel viewpoints located at increasing distances from the reference viewpoint, disocclusions revealing previously unseen scene portions increase to the point that image quality becomes unacceptable.

Almost 30 years ago, *Talisman* [50] proposed the radical idea of replacing the linear framebuffer with a cache of layered framebuffer "chunks" and creating target images by warping and compositing chunks on the fly. Talisman hardware was never built, and using linear buffers for caching between graphics pipeline invocations is still the dominant approach today. This preference for linear buffers may be attributed to their importance for the fixed-function stages of the GPU, especially the raster operations. However, entire graphics pipelines, such as *Nanite* [23], are increasingly created entirely with compute shaders, as many previous dependencies on fixed-function units have been relaxed or removed.

In this paper, we explore how a chunked cache in the vein of Talisman can be implemented in software using modern GPU techniques (Figure 1). We make the following contributions:

Chunked cache for visibility and shading. We describe a chunked cache implemented entirely in GPU shaders. It can be built,

Author's Contact Information:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGGRAPH 2025, Vancouver, Canada
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

refreshed and queried with a similar level of efficiency as found in a standard deferred rendering system. Our cache has several desirable properties: All samples on or near a view ray are retrievable in constant time and without having to refer back to the (potentially complex) scene geometry. The memory footprint is proportional to the set of samples considered relevant for downstream applications, rather than to the overall size of the scene. No precomputation is required; hence, cache refreshes work with fully dynamic scenes.

Potentially visible set via disocclusion computation. As a first application, we show how to generate a from-region potentially visible set (PVS) 6.9× faster on average than the previously fastest method. This improvement is possible because the PVS is computed on the chunked cache rather than on the geometric scene description. Consequently, we can generate a conservative PVS by considering per-chunk disocclusions, rather than geometric occlusions as used in previous work. Moreover, the use of layers makes our method order-independent, avoiding costly synchronization and sorting.

Frame extrapolation with extended range. As a second application, we demonstrate frame extrapolation with extended range compared to traditional framebuffer extrapolation. After the samples contained in the PVS are shaded, new frames can be extrapolated by reprojecting the shading cache at a rate that depends only on the target resolution but not on the geometric scene complexity. Since the PVS anticipates disocclusions, extrapolation is possible over more frames than is feasible with methods that work only with linear buffers. We demonstrate extrapolation of novel views at 4K resolution in 2-6 ms without the need for costly inpainting or neural networks.

2 Related work

We discuss related relevant work in the three areas of visibility computation, upsampling and shading caching.

2.1 Visibility computation

Deferred rendering improves over traditional forward rendering by resolving from-point visibility before entering the shading pass. At minimum, it suffices to generate a visibility buffer [9] with depth and primitive ID information to capture visible pixels. However, for scenes with high geometric complexity, it may pay off to prepend a from-point PVS generation [6] to the visibility buffer generation. The PVS should be conservative – it should report only too much to be visible (false positives), but never too little (false negatives).

Identifying significant occluders can help prune the scene to be considered for visibility sampling. Determining such an occlusion requires fusion of many small and medium-sized occluders [13, 43, 55]. Usually, from-point occluder fusion is implemented by aggregating occluders into a hierarchical shadow volume [5] or a depth pyramid [15]. Various enhancements have been proposed for using the depth buffer for occlusion queries, but they either require careful scheduling of scene traversal [30] or GPU hardware extensions [1].

A key technique in all these PVS generation methods is traversing the scene front to back, so that the shadow volumes of occluders close to the camera lead to early culling of occluded parts of the scene. Durand et al. [13] use rasterization to reproject an occlusion mask oriented parallel to the image plane to successively increasing

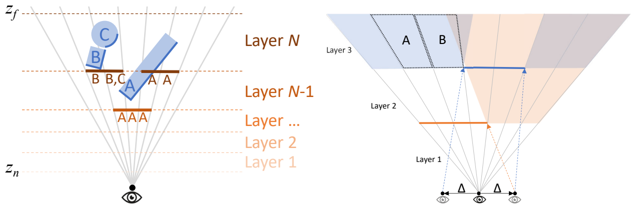

Fig. 2. (left) Object A maps to three chunks of layer $N - 1$ and two chunks of layer N, whereas objects B and C are fused into two chunks of layer N. (right) If we allow a viewpoint movement by up to Δ, the area marked in orange is disoccluded in layer 1, while the area marked in blue is disoccluded in layer 2. In layer 3, only the area disoccluded both by layer 1 and layer 2 must be included in the PVS. In particular, we can determine that B is disoccluded, but A is occluded because it is not visible from layer 1.

distances from the viewpoint. Voglreiter et al. [53] traverse the scene by peeling layers off an octree and retain the current visibility mask as a k-buffer. Our technique is inspired by these works but differs in two important ways: First, it explicitly considers disocclusion rather than occlusion. Second, it is order-independent, avoiding the synchronization costs that limited previous methods.

Unfortunately, a from-point PVS is not sufficient for frame extrapolation over large distances or multi-fragment effects [51]. These applications require a from-region PVS operating in a 4D or even 5D ray-space [12]. Naturally, the algorithmic complexity of from-region visibility is significantly higher than the one of from-point visibility with its 2D ray-space. Hence, the computation of from-region visibility tends to be costly, often in the range of hundreds of milliseconds [19, 25], with the fastest method to date [53] still requiring 20 ms on a recent GPU. As a consequence, the PVS is either precomputed for a scene partitioned into discrete regions [14], or a from-point PVS is computed from one or more predicted future viewpoints [22, 27, 34, 40] as a substitute for the from-region PVS. The former limits the benefits to a static scene, whereas the latter depends on correct prediction and is potentially error-prone if not all visible primitives are caught in the sample views.

2.2 Spatiotemporal upsampling

Spatiotemporal upsampling can help amortize the shading load by exploiting spatial and temporal coherence [17]. If we want to synthesize entirely novel views, the classic approach is 3D warping [29] of a conventional framebuffer or G-buffer. Reprojecting via a depth or motion vector buffer inevitably leads to undesirable disocclusions, limiting the possible extent of extrapolation. Even with sophisticated strategies for warping [7] or adaptive refresh [47], disocclusions are hard to hide for ultra-high resolutions and framerates.

One way to mitigate the disocclusion problem is by frame interpolation, where additional upsampled frames are inserted to stabilize framerates between a previous and predicted frame, usually with the help of motion vectors [58]. Better image quality can be obtained by predicting or refining motion vectors with neural networks [8]. A similar form of frame interpolation based on neural networks is now available through commercial solutions such as DLSS3 [38], although technical details are not revealed. Despite its popularity,

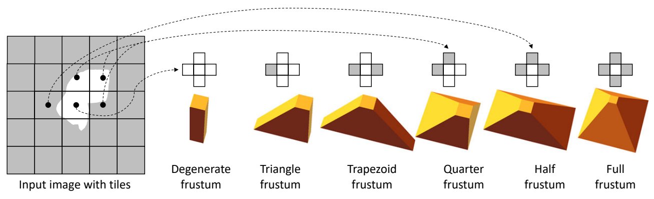

Fig. 3. (left) A grid of 4×4 chunks partially covered by scene geometry (in gray). We consider the occupation status of neighboring chunks (gray/white pixels in cross) for disocclusion computations in the scene. Partially filled or empty chunks are classified as "open," whereas chunks completely filled with samples (gray) are "full." (right) Patterns of edges between open and closed chunks lead to unique disocclusion frusta (up to rotation).

frame interpolation suffers from additional latency, making it unsuitable for longer predictions or streaming applications that require keeping strict temporal bounds.

In contrast, frame extrapolation methods generate novel views purely from past frames. Extrapolation thereby avoids the additional latency, but it must handle disocclusions without the ability to peek into future frames. This lack of disocclusion information can be compensated for to some extent by careful processing of motion vectors and additional G-buffer channels [59], possibly with the help of neural networks [16, 56].

A shading cache in the form of a single 2.5D framebuffer can never provide all the samples needed to fill in disocclusions revealed from a sufficiently displaced viewpoint. Therefore, some frame extrapolation schemes try to opportunistically collect samples from a cache consisting of multiple past frames [57]. However, without explicitly seeking for future changes in visibility, cache misses caused by disocclusions can never be avoided. Our approach improves over this work by using a PVS to reliably avoid cache misses.

2.3 Shading caching in 3D

Only a cache layout capable of storing samples at arbitrary locations in 3D space can ensure that all relevant samples are present in the cache. Such alternative cache layouts can be roughly categorized into view-independent and view-dependent organizations. The view-independent category uses either a hash table filled with individual shading samples [20, 28, 39] or some form of texture space. The texture-space methods can be categorized into three groups: The scene is either pre-charted as a whole [18], split into pre-charted primitive groups which are dynamically allocated at runtime [3, 34], or individual primitives are packed into a kind of atlas on the fly [10, 21]. Such texture-space organizations have been demonstrated to be beneficial for applications such as radiance caching [48, 49] or streaming [21, 34]. However, they perform less favorably for frame extrapolation. One reason is that frame extrapolation benefits from the lower distortion of storing view-dependent sample patterns when reprojecting from a nearby view with a similar perspective [21, 35] rather than a canonical view.

Therefore, most of the literature relevant to our work organizes the shading cache using view-dependent methods. Early techniques relied on rendering individual objects to textures [44] or proposed GPU hardware extensions [11, 50]. Other approaches use per-pixel linked lists [19, 45], depth peeling [24, 27], *k*-buffering [4] or reprojection into a peeled-layer-like structure [57]. Our cache also falls into this view-dependent category, but its chunked organization better exploits spatial coherence, making it drastically more efficient than previous view-dependent 3D shading caches. Moreover, our cache uses layers with regular depth spacing, which makes it similar to other image-based models, such as impostors [42], pop-up light fields [46] or multi-plane images [32].

3 Method

We propose replacing a linear buffer (specifically, a conventional visibility buffer [9]) with a chunked cache in a modified deferred rendering system. The cache is organized into rectangular groups of samples, sparsely located in an array of chunk planes parallel to the image plane. We call the slab between two chunk planes a *layer*. The chunks in the cache represent the scene geometry, but resampled into a sparse regular grid in normalized device coordinates. We first explain how to generate the cache, then we describe how to use the cache to compute a PVS and extrapolate novel frames.

3.1 Chunked cache generation

To build the cache, we rasterize the scene from the current viewpoint and write primitive ID and depth per fragment into the cache. We retain separate depth buffers for each layer. From two fragments in the same layer that map to the same location in image space, the depth test retains only the closer one. This rule ensures that the fragment closest to the viewpoint is always present in the cache, although it may not be found in the frontmost layer. Fragments that are hidden from the current viewpoint will be contained in the cache only if they fall into different layers (see Figure 2, left). Fragments of primitives that span two layers will be assigned either to a chunk in the first layer or to a chunk in the second layer, depending on their depth values.

3.2 PVS generation by disocclusion computation

As a first application of the chunked cache, we describe an efficient algorithm for PVS computation.

After resampling and quantizing the scene geometry to one entry per layer and pixel location, disocclusions can occur only between layers. This observation is the key to implementing a drastically simplified PVS algorithm. Unlike other PVS algorithms that must propagate occlusions front-to-back through the scene, we can explicitly compute *disocclusions*. A critical advantage of our disocclusion computation is its ability to work without access to geometric primitives and in an order-independent manner.

The PVS computation operates at the granularity of chunks. Let us assume two chunks of the same layer, but in different image locations (Figure 2, right): A is disoccluded from layer 2, but not from layer 1. B is disoccluded from both layer 1 and layer 2. For inclusion in the PVS, a chunk must be disoccluded in *all* previous layers, which applies only to B. Our goal is therefore to efficiently compute disocclusion for all chunks and all previous layers. We will make use of the fact that the disocclusion information of all chunks of all layers can be updated in parallel through scattered writes without enforcing a particular order of traversing the scene.

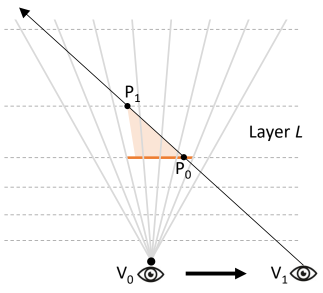

Fig. 4. Frame extrapolation marches a ray from a novel viewpoint V_1 through the layers. The ray enters layer L at P_0 and leaves it at P_1. Consequently, we must seek for an intersection with the chunks between P_0 and P_1, which are marked in orange.

We must only ensure that the disocclusion information affecting a chunk can be collected for the final classification of the chunk.

To ensure a conservative algorithm, a chunk completely filled with samples is classified as *closed*, whereas a chunk is *open* if it is partially filled with samples or completely empty. A disocclusion occurs if the viewpoint is moved such that the scene behind a closed chunk becomes visible through an open chunk. Therefore, we inspect all open chunks and classify their edges with adjacent chunks in the same layer. The resulting patterns of edges can be classified into six cases (Figure 3) corresponding to *disocclusion frusta* with characteristic shapes.

The six shapes are chosen to have minimal volume. Only edges between an open and a closed chunk generate a sloped pyramid surface. Edges between two open chunks need not be included in the frustum, since the corresponding disoccluded areas are already handled by the chunk's neighbors. Hence, the trivial sixth case of an open chunk surrounded by four open chunks leads to a degenerate frustum that only covers the chunk itself.

The disocclusion computation algorithm produces a conservative PVS. False negatives can occur only if individual samples are suppressed by the layer-wise depth tests during cache generation, but never for entire chunks during PVS generation. Our results demonstrate that these per-sample false negatives are similar in frequency to previous PVS methods and negligible in practice.

3.3 Viewcell definition

Before the actual PVS computation starts, clipping of large portions of the scene is already taken care of by hardware rasterization filling the cache. This rasterization requires specifying a conventional view frustum with a pinhole camera model. In contrast, computing a PVS requires first defining a viewcell. The viewcell can be seen as a generalization of a camera specification. It defines a (usually convex) subspace of 5D ray-space that contains all eligible viewpoints and view directions. The viewcell determines the range of possible frame extrapolation. The chunked cache must contain all portions of the scene that can be reached from any ray contained in the viewcell.

To fulfill this requirement, we adopt a method similar to the one described by Voglreiter et al. [53], who adjust the camera and increase the subtended angle of the view frustum to enable additional rotations. Voglreiter et al. show that, for a frustum with a field of view of 2α and a viewcell allowing lateral motion of Δ, the PVS is guaranteed to be valid for a viewpoint displacement along the optical axis by up to $\Delta/\tan\alpha$. In our experiments, we found that the range of extrapolation often exceeds this conservative limit by a large amount. Therefore, we manually adjusted the number of extrapolated frames per scene based on the observed image quality.

3.4 Frame extrapolation

Our second application builds on the first one and uses the PVS for efficient frame extrapolation. It consists of a shading task and an extrapolation task that operate concurrently. The shading task fills the shading cache for all chunks contained in the PVS. The extrapolation task creates extrapolated frames from shading stored in the cache. Only frames extrapolated from the cache are shown. Since the shading task is amortized over multiple frames, we double-buffer two shading caches. A "front" cache with finished shading is used for frame extrapolation, while a "back" cache contains a new PVS and is filled with new shading information. This new PVS is created for a predicted future viewpoint that corresponds to the time when the shading is expected to be complete. We predict the viewpoint and the time taken by the shading with linear extrapolation.

PVS shading. For every chunk in the PVS and every covered sample in the chunk, we run a shader to determine color and alpha values. The shader employs the standard procedure for rendering with visibility buffers [9]: The shader input consists of the fragment's screen coordinates and the primitive ID, which are used to determine barycentric coordinates and query per-fragment attributes.

Novel view generation. We use a variant of reverse reprojection, but without motion vectors. For every pixel of the extrapolated frame, a view ray traverses the layers of the shading cache in front-to-back order, skipping over non-existing chunks in the process. For each visited layer, we determine the portion of the view ray that intersects the layer. The chunk locations along that ray are enumerated by a DDA algorithm and inspected sequentially (Figure 4). For every chunk that may have an intersection with the ray, we determine the exact intersection point by projecting the view ray into the chunk's depth buffer and searching with a DDA variant [31].

The search reports an intersection for the first sample location where the depth sample has a value smaller than the depth of the ray. Only if an intersection is found, this sample is alpha-blended with the accumulated color. If no sample is found or the opacity of the ray is not yet saturated, the traversal advances to the next layer. If no transparent surfaces are present in the scene, the ray can be terminated on the first hit.

4 Implementation

In this section, we provide selected implementation details. Upon acceptance, we will make our code available as open source to encourage further exploration of chunked caching.

4.1 Chunked cache

We empirically determined that a chunk size of $K \times K$ with $K = 8$ (for 1080p and 1440p) or $K = 16$ (for 4K) works well in our experiments. Higher resolutions, such as 4K, benefit from larger chunk

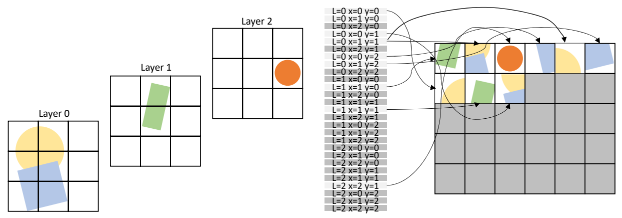

Fig. 5. (left) The scene is subdivided into three layers with 3×3 chunks per layer. (right) Page table and chunk buffer corresponding to the scene.

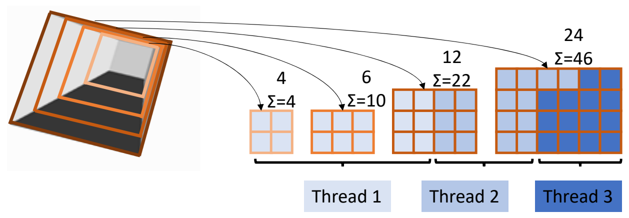

Fig. 6. (left) Intersecting a disocclusion frustum with subsequent layers produces one rectangle per layer (four rectangles in this example). (right) A prefix sum of the rectangle sizes determines that 46 chunks must be marked as disoccluded. The work is evenly distributed across a workgroup consisting of three threads, as indicated by shades of blue.

sizes, likely because of better cache coherence. Smaller resolutions do not benefit from larger chunk sizes, because the average number of empty samples in the chunk (not covered by any geometric primitive) becomes too high, leading to fragmentation. A similar consideration applies to the number of layers. We found that $N = 64$ layers already sufficiently cover all relevant samples needed for our applications without losing too much by the depth quantization that keeps only the frontmost sample per view-ray and layer. More layers provide small improvements in image quality, but have much higher computational costs (see the results section).

We index in the cache using the quantized coordinates of a sample point $\mathbf{p} = [x, y, z]$, where x, y are in screen coordinates, and z is the linear depth between the near clip plane z_n and the far clip plane z_f, i.e., $z_n < z < z_f$. The corresponding chunk coordinate C is

$$C(\mathbf{p}) = [c_x, c_y, c_z] = \left[\lfloor x/K \rfloor, \lfloor y/K \rfloor, \left\lfloor N \frac{\log(z - z_n + 1)}{\log(z_f - z_n + 1)} \right\rfloor \right]$$

for a cache with N layers. In other words, the depth interval associated with a layer is linearly related to the disparity of a sample, i.e., the distance the sample travels across the scene when it is reprojected to a new viewpoint from the original viewpoint used to create the cache. This property makes it easy to guarantee error bounds when reprojecting the cached samples.

We use a standard vertex shader to invoke the fixed-function rasterizer, but our method differs from a conventional fragment pipeline in two important aspects:

First, we do not use standard raster operations (ROP) to write fragment shader results to a linear framebuffer. Instead, the fragment shader writes its results to a shader storage buffer object (SSBO), i.e., a random access buffer. This strategy follows the suggestion of Clarberg et al. [10] to leverage "the capability of modern GPUs to perform unordered memory accesses from within shaders."

Second, as proposed by Karis et al. [23], the fragment shader uses 64-bit atomic minimum operations to implement depth buffering. Each visibility sample is a 64-bit value, with the depth occupying the most significant 32 bits and the primitive ID occupying the least significant 32 bits. Consequently, an atomic comparison determines the smaller depth value and writes the entire sample back to the cache without a race condition. Giving up ROP lets us replace a linear framebuffer with our own, more efficient data structure.

The memory layout of the visibility cache is organized as a two-level hierarchy [26], consisting of a dense page table that contains pointers to a sparse chunk buffer (Figure 5). We index into the page table using the chunk location c_x, c_y and the chunk's layer c_z. Initially, the chunk buffer is empty (all fragments have an invalid primitive id and a depth of z_f), and the page table entries are cleared. To write a sample at position $\mathbf{p}$, the fragment shader looks up the chunk address in the page table at the entry with the index $C(\mathbf{p})$.

If that entry is invalid, the chunk is allocated on the fly using a fast combination of atomic operations. For that purpose, the page table contains two entries: an address that is initialized with an invalid value to indicate an empty page, and a ticket. The ticket is initialized to zero and can be drawn only by the first thread (using an atomic increment). The thread drawing the ticket then acquires an empty page via atomically incrementing the page count and writes the new address to the page table. Threads that did not draw a ticket probe the page table entry until the address becomes available (i.e., a valid address is returned). Probing relies on atomically adding zero to the address. Adding zero does not change the state of the address, but the atomic operation forces cache synchronization and minimizes the latency until the new address becomes available.

4.2 PVS generation

The objective of the PVS generation is to determine the visibility (occluded or disoccluded) for each non-empty chunk. To store the results of the disocclusion computation, we allocate for each non-empty chunk a *visibility mask* with one bit per layer indicating if the chunk is disoccluded from a given layer. Layer 1 is stored in the least significant bit. The mask is initialized to zero, indicating that all chunks are occluded by default. Setting bits in the visibility mask is performed using atomic operations, so all chunks can be processed in parallel without any ordering or sorting. Therefore, we avoid the drawback of synchronized back-to-front traversal that limits many previous methods [30, 53].

A first compute shader processes all non-empty, open chunks in parallel. The sample counter in the page table entry indicates if a non-empty chunk is open or closed. We retrieve the open/closed classification of the chunk's 4-connected neighbors and use the resulting 4-bit pattern to identify the shape of the disocclusion frustum (Figure 3). We write a corresponding record describing the disocclusion frustum to a per-chunk array in global memory. For a degenerate frustum (open chunk with open neighbors) of a chunk at layer L, we set a single bit L in the visibility mask to indicate trivial disocclusion.

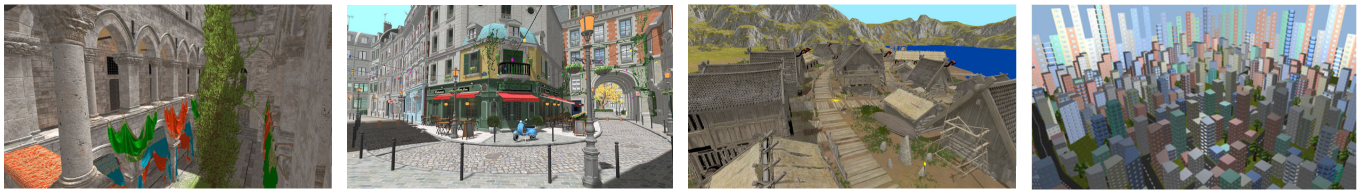
Fig. 7. Test scenes from left to right, with triangle count in parentheses: Sponza (21.9M), Bistro Exterior (3.1 M), Viking Village (6.5 M), Big City (31.4 M)

Scene	Bistro	Sponza	Viking	Bistro	Sponza	Viking
Viewcell	30 cm	30 cm	10 cm	30 cm	30 cm	10 cm
Resolution	1080p	1080p	1080p	4K	4K	4K
Extrap.frames	10	11	7	12	14	15

Table 1. Number of frames that can be extrapolated

A second compute shader assigns one workgroup to each disocclusion frustum. For a disocclusion frustum spawned in layer L, the workgroups responsibility is to set the disocclusion bit L in the visibility masks of all chunks contained in the frustum. The frustum is intersected with all subsequent layers $i \in [L+1 \dots N]$. Every intersection with subsequent layers yields a rectangle of increasing size (i.e., number of chunks covered). The rectangle coordinates are stored in an array in shared memory (Figure 6). We run a prefix scan over the rectangle areas to obtain a searchable index into the array. For load balancing, the total number of chunks covered by the disocclusion frustum is distributed equally between the threads of the workgroup. Then, each thread uses binary search in the index to find the chunk location it is responsible for and marks the corresponding entry in the chunk's visibility mask. The page table location of a chunk is implicitly known from the rectangle index and the thread's position inside the rectangle.

We obtain visibility masks that store if a chunk in layer L is disoccluded in all previous layers $j \in [1 \dots L-1]$. To determine PVS inclusion, a bit operation on the chunk's visibility mask ensures that all bits in previous layers are set. This query operation is formed in a third compute shader to obtain the final PVS. Please see the supplementary material for pseudocode of the algorithm.

4.3 PVS shading

For PVS shading, a compute shader invokes one workgroup per chunk in the PVS. One thread per (potentially) visible pixel executes the shading pass and writes the resulting shading values back to the cache. At this point, we do not require the primitive ID anymore and overwrite it with the RGBA value determined by the shading. This in-place conversion of a chunk from storing visibility samples to storing shading samples avoids allocating additional memory for the shading cache and even provides a small speed-up. After shading is complete, chunks that are marked as part of the PVS in the page table contain colors in the lower 32 bits of the sample. Chunks that are not included in the PVS still contain primitive IDs, but these chunks will no longer be required in downstream applications. When a new cache (a new PVS) is started, both the page table and the chunk buffer are cleared.

4.4 Frame extrapolation

During frame extrapolation, we follow view rays emitted from the new viewpoint. This procedure also relies on compute shaders. Note that we do not use GPU ray-tracing hardware units, since we only perform a variant of screen-space ray-marching.

During the intersection of a ray with the chunked cache, empty chunks along the ray are immediately detected because they lack a page table entry. For non-empty chunks, we compare the min/max depth values of a chunk with the ray depth to determine whether the chunk can have an intersection with the ray. This test allows for an early exit before loading the actual chunk data.

5 Results

In this section, we present performance evaluation results for a set of test scenes and comparisons to state-of-the-art methods. Our results were collected on a desktop computer (CPU: Intel i7-13700T with 64 GB RAM, GPU: NVidia GeForce RTX 4090, OS: Windows 11). In all experiments, the horizontal field of view of the target images was set to 60°, and the extended field of view used to sample the viewcell was set to 90° which corresponds to a view cell of 30 cm. We compared three target frame resolutions, 1080p (1920×1080), 1440p (2560×1440) and 4K (3840×2160).

The test scenes used in our evaluation are shown in Figure 7. Big City is challenging because of its size (31.7M polygons), while the modified Sponza scene (21.9 M polygons) is even more challenging because it houses a tree with high depth complexity resulting from very small polygons forming the foliage. In each scene, an animated camera path (600 frames) was used to generate a reproducible image sequence. Measurements were averaged over all frames.

5.1 Cache generation

Figure 8 shows the memory consumed to generate the chunked cache (without PVS computation or shading), and Figure 9 shows the runtime. For comparison, we list two simpler cache data structures: a dense multi-layered buffer (implemented as an array texture with the same number of layers as the chunked cache) and a dense single-layer buffer (storing only a single visible sample per view ray). The chunked cache requires 4-6× the memory of a single layer and takes about 2.5× longer to generate; a multi-layered buffer has excessive memory requirements and generation times. Cache size is rather insensitive to the geometric scene complexity (Figure 10). Cache generation for small/medium scenes ranges from 2–4 ms (Figure 11).

We compared cache configurations with 32, 64, and 128 layers. The memory requirements (Figure 13), the generation time (Figure 14) and the chunk count (Figure 14) for these layer numbers

Method	Ref.	PSNR↑	SSIM↑	LPIPS↓
Extranet	[16]	28-31	0.83-0.95	-
GFFE	[57]	24	0.86	9.7
Mob-FGSR	[59]	25-30	0.84-0.92	-
Cunked cache	**ours**	36-42	0.87-0.95	2.1-6.2

Table 2. Image quality measurements of frame extrapolation methods.

all show an increase that is sub-linear relative to the layer number. This fact makes it inexpensive to have more layers in the cache for a better scene approximation. However, PVS generation and frame extrapolation are more sensitive to the layer number. Therefore, we settled for 64 layers as the best compromise in our implementation.

5.2 PVS evaluation

For the evaluation of the PVS, we compare our method with Trim Regions (TR), the previously fastest method for from-region PVS computation, using the original code provided by the authors [53]. TR uses a two-level hierarchy for efficient front-to-back sorting of fragments, where the first level uses octree traversal, and the second level sorts fragments in a k-buffer. Occlusion is propagated via an extension of occluder shrinking [55].

We report on the runtime of PVS generation with the chunked cache in Figure 17. It ranges from about 1.1 ms (Bistro, 1080p, 10 cm viewcell) to 10.6 ms (Big City, 4K, 60 cm viewcell). The chunked cache outperforms Trim Regions on average by a factor of 6.9×.

We state the PVS size as a fraction of the primitive count of the entire scene to a ground truth PVS. As in previous work [19, 27, 53], we approximate ground truth by dense sampling from 576 viewpoints uniformly distributed inside the viewcell. For the chunked cache, a scene primitive is considered visible if its primitive ID is contained in at least one chunk marked as disoccluded in the PVS. Note that our method of image generation though frame extrapolation does not use the scene geometry directly. Therefore, we added an extra sub-pass for counting the primitives referenced in the cache, which is not considered in the reported timings.

Figure 18 shows the *false positive geometry rate*, i.e., the number of primitives our algorithm reports as belonging to our PVS method divided by the size of the ground truth PVS. The overhead ranges from about 50% to 160%. This rate is 3–10× better than for TR (except for Sponza). Note that the overall PVS size is always between 0.5% and 7% of the total scene size, so the overheads coming from reported false positives do not have a significant effect on the runtime.

Figure 19 reports *pixel errors* [37], i.e., the average number of false negative pixels per frame for which the wrong depth or primitive ID is present in the visibility buffer.. This number is reported as a fraction of the screen resolution. Pixel error is considered to be most indicative of the image quality when rendering from a PVS [25, 53]. To suppress false negatives coming from spurious micro-geometry (triangles smaller than a pixel), we use the approach described by Voglreiter et al. [53]. The implementation of TR that we used in the comparison relies on a more effective micro-geometry filtering which is described in the dissertation of Voglreiter [52, section 5.3.5]. Therefore, our false negative rate is comparable to TR but slightly worse. However, the superior micro-geometry filtering method is independent of the algorithm and could be added to our method.

5.3 Frame extrapolation

To test the frame extrapolation, we modified the number of light-sources so that the resulting frame rate was roughly 60 Hz using a standard frame-by-frame deferred renderer. For this experiment, we used 128 layers and tested in two resolutions (1080p and 4K).

We characterize the performance of the frame extrapolation by measuring the time it takes to shade the PVS and the time it takes to extrapolate a shaded frame (Figure 20). Table 1 lists the extrapolated frames successfully accommodated in our experiments. Frame extrapolation by itself takes about 1 ms for 1080p and 3–6 ms for 4K. The combined time for PVS shading amortized over the extrapolated frames until the animated viewpoint leaves the viewcell (Table 1) is about 8–9 ms for both resolutions. This corresponds to an average framerate of just under 120 Hz presented to the user, effectively doubling the original framerate. Note that the camera moves constantly at a fast running speed of 3 m/s. The observed part of the scene changes in every frame, exercising maximum pressure on the view extrapolation. Extrapolation while standing or looking around could continue over much longer periods before PVS re-computation.

Figure 21 characterizes the image quality of extrapolated frames relative to a ground truth image from a deferred renderer. We show four metrics (PSNR, SSIM [54], FLIP [2], LPIPS [60]). Table 2 compares our results to measurements of three state-of-the-art frame extrapolation methods. Results should not be directly compared, because all these methods (including ours) use different scenes and settings for evaluation. However, our method generates a similar range of results and has the added benefit of being able to extrapolate multiple frames ahead of time.

6 Conclusions and future work

The chunked shading cache demonstrates how to implement graphics pipelines with new properties. It works at similar levels of efficiency as conventional deferred shading, but allows frame extrapolation far beyond what is possible with single-layer framebuffer (or G-buffer) warping. We show that our pipeline runs efficiently at resolutions of up to 4K and framerates up to 120 Hz.

The flexibility of a chunked cache suggests many avenues for future work. We currently consider layers with uniform spacing (in log space), but a generalization to adaptive spacing is straightforward and would allow handling even more complex scenes efficiently. In scenes with moving objects, the cache organization could be extended to consider a model-view transformation rather than just a view transformation. For virtual reality, stereo image pairs could be extrapolated, and individual scaling of chunks would enable variable rate shading and efficient foveated rendering.

The chunked cache also lends itself to temporally adaptive shading. For example, chunk-specific refresh rates could be determined from shading gradients, as proposed by Müller et al. [33]. The combination of shading gradients with motion vectors (as used in conventional temporal anti-aliasing) or per-object chunks for moving objects could help extrapolate shading in time for animated scenes.

References

[1] Magnus Andersson, Jon Hasselgren, and Tomas Akenine-Möller. 2015. Masked depth culling for graphics hardware. *ACM Transactions on Graphics* 34, 6, Article 188 (2015), 9 pages. doi:10.1145/2816795.2818138

[2] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. 2020. FLIP: A Difference Evaluator for Alternating Images. *Proceedings of Computer Graphics and Interactive Techniques* 3, 2, Article 15 (2020), 23 pages. doi:10.1145/3406183

[3] Daniel Baker and Mark Jarzynski. 2022. Generalized Decoupled and Object Space Shading System. In *Eurographics Symposium on Rendering*. doi:10.2312/sr.20221163

[4] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. 2007. Multi-fragment Effects on the GPU Using the k-buffer. In *Proceedings of I3D*. Association for Computing Machinery, New York, 97–104. doi:10.1145/1230100.1230117

[5] J. Bittner, V. Havran, and P. Slavik. 1998. Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International*. IEEE Comput. Soc, 207–219. doi:10.1109/CGI.1998.694268

[6] Jiri Bittner and Peter Wonka. 2003. Visibility in Computer Graphics. *Environment and Planning B: Planning and Design* 30, 5 (2003), 729–755. doi:10.1068/b2957

[7] Huw Bowles, Kenny Mitchell, Robert W. Sumner, Jeremy Moore, and Markus Gross. 2012. Iterative Image Warping. *Computer Graphics Forum* 31 (2012), 237–246. doi:10.1111/j.1467-8659.2012.03002.x

[8] Karlis Martins Briedis, Abdelaziz Djelouah, Raphaël Ortiz, Mark Meyer, Markus Gross, and Christopher Schroers. 2023. Kernel-Based Frame Interpolation for Spatio-Temporally Adaptive Rendering. In *Proceedings of SIGGRAPH*. Association for Computing Machinery, New York, Article 59, 11 pages. doi:10.1145/3588432.3591497

[9] Christopher A Burns and Warren A Hunt. 2013. The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading. *Journal of Computer Graphics Techniques* 2 (2013), 55–69. Issue 2. http://jcgt.org/published/0002/02/04/

[10] Petrik Clarberg and Jacob Munkberg. 2014. Deep Shading Buffers on Commodity GPUs. *ACM Transactions on Graphics* 33, 6 (2014), 227:1–227:12. doi:10.1145/2661229.2661245

[11] Joachim Diepstraten, Daniel Weiskopf, Martin Kraus, and Thomas Ertl. 2004. Vragments and Raxels – Relocatability as an Extension to Programmable Rasterization Hardware. In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*. 181–188.

[12] Frédo Durand, George Drettakis, and Claude Puech. 2002. The 3D visibility complex. *ACM Transactions on Graphics* 21, 2 (2002), 176–206. doi:10.1145/508357.508362

[13] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. 2000. Conservative Visibility Preprocessing Using Extended Projections. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*. ACM Press, New York, NY, USA, 239–248. doi:10.1145/344779.344891

[14] Epic Games. 2022. Precomputed Visibility Volumes. https://docs.unrealengine.com/5.1/en-US/precomputed-visibility-volumes-in-unreal-engine/ Last visited: 2022-12-21.

[15] Ned Greene, Michael Kass, and Gavin Miller. 1993. Hierarchical Z-buffer visibility. In *Proceedings of SIGGRAPH (SIGGRAPH '93)*. Association for Computing Machinery, New York, 231–238. doi:10.1145/166117.166147

[16] Jie Guo, Xihao Fu, Liqiang Lin, Hengjun Ma, Yanwen Guo, Shiqiu Liu, and Ling-Qi Yan. 2021. ExtraNet: real-time extrapolated rendering for low-latency temporal supersampling. *ACM Transactions on Graphics* 40, 6, Article 278 (2021), 16 pages. doi:10.1145/3478513.3480531

[17] Robert Herzog, Elmar Eisemann, Karol Myszkowski, and H.-P. Seidel. 2010. Spatio-temporal upsampling on the GPU. In *Proceedings of I3D*. Association for Computing Machinery, New York, 91–98. doi:10.1145/1730804.1730819

[18] K E Hillesland and J C Yang. 2016. Texel Shading. *Proceedings of Eurographics (Short Papers)* (2016). doi:10.2312/egsh.20161018

[19] Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019. The Camera Offset Space: Real-Time Potentially Visible Set Computations for Streaming Rendering. *ACM Transactions on Graphics* 38, 6, Article 231 (2019), 14 pages. doi:10.1145/3355089.3356530

[20] J. Hladky, H. P. Seidel, and M. Steinberger. 2019. Tessellated Shading Streaming. *Computer Graphics Forum* 38, 4 (2019), 171–182. doi:10.1111/cgf.13780

[21] Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2021. SnakeBinning: Efficient Temporally Coherent Triangle Packing for Shading Streaming. *Computer Graphics Forum* (2021). doi:10.1111/cgf.142648

[22] Jozef Hladky, Michael Stengel, Nicholas Vining, Bernhard Kerbl, Hans-Peter Seidel, and Markus Steinberger. 2022. QuadStream: A Quad-Based Scene Streaming Architecture for Novel Viewpoint Reconstruction. *ACM Transactions on Graphics* 41, 6, Article 233 (2022), 13 pages. doi:10.1145/3550454.3555524

[23] Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. SIGGRAPH Course on Advances in Real-Time Rendering in Games. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf Last visited: 2025-09-25.

[24] Janghun Kim and Sungkil Lee. 2023. Potentially Visible Hidden-Volume Rendering for Multi-View Warping. *ACM Transactions on Graphics* 42, 4, Article 86 (2023), 11 pages. doi:10.1145/3592108

[25] Thomas Koch and Michael Wimmer. 2021. Guided Visibility Sampling++. *Proceedings of Computer Graphics and Interactive Techniques* 4, 1, Article 4 (2021), 16 pages. doi:10.1145/3451266

[26] Martin Kraus and Thomas Ertl. 2002. Adaptive Texture Maps. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Eds.). The Eurographics Association, Goslar, Germany, 7–15. doi:10.2312/EGGH/EGGH02/007-015

[27] Jaroslav Kravec, Martin Kácerik, and Jirí Bittner. 2023. PVLI: Potentially Visible Layered Image for Real-Time Ray Tracing. *The Visual Computer* 39, 8 (2023), 3359–3372. doi:10.1007/s00371-023-03007-5

[28] Gábor Liktor and Carsten Dachsbacher. 2012. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH symposium on interactive 3D graphics and games*. 143–150.

[29] William R. Mark, Leonard McMillan, and Gary Bishop. 1997. Post-rendering 3D warping. In *Proc. SIGGRAPH Symposium on Interactive 3D Graphics*.

[30] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. 2008. CHC++: Coherent Hierarchical Culling Revisited. *Computer Graphics Forum* 27, 2 (2008), 221–230. doi:10.1111/j.1467-8659.2008.01119.x

[31] Morgan McGuire and Michael Mara. 2014. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques* 3, 4 (2014), 73–85. http://jcgt.org/published/0003/04/04/

[32] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. 2019. Local light field fusion: practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics* 38, 4, Article 29 (2019). doi:10.1145/3306346.3322980

[33] Joerg H. Mueller, Thomas Neff, Philip Voglreiter, Markus Steinberger, and Dieter Schmalstieg. 2021. Temporally Adaptive Shading Reuse for Real-Time Rendering and Virtual Reality. *ACM Transactions on Graphics* 40, 2, Article 11 (2021), 14 pages. doi:10.1145/3446790

[34] Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading Atlas Streaming. *ACM Transactions on Graphics* 37, 6, Article 199 (Dec. 2018), 16 pages. doi:10.1145/3272127.3275087

[35] Thomas Neff, Joerg Mueller, Markus Steinberger, and Dieter Schmalstieg. 2022. Meshlets and How to Shade Them: A Study on Texture-Space Shading. *Computer Graphics Forum* 41, 2 (2022), 277–287. doi:10.1111/cgf.14474

[36] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. 2007. Accelerating real-time shading with reverse reprojection caching. In *Proceedings of SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Eurographics Association, Goslar, 25–35. doi:doi/10.5555/1280094.1280098

[37] S. Nirenstein and E. Blake. 2004. Hardware accelerated visibility preprocessing using adaptive sampling. In *Proceedings of Eurographics Conference on Rendering Techniques (EGSR'04)*. The Eurographics Association, Goslar, Germany, 207–216. doi:10.2312/EGWR/EGSR04/207-216

[38] NVIDIA-Cooperation. 2024. DLSS Frame Generation. https://www.nvidia.com/en-us/geforce/technologies/dlss/ Last visited: 2024-01-06.

[39] Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, and Frédo Durand. 2011. Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics* 30, 3, Article 17 (2011), 17 pages. doi:10.1145/1966394.1966396

[40] Bernhard Reinert, Johannes Kopf, Tobias Ritschel, Eduardo Cuervo, David Chu, and Hans-Peter Seidel. 2016. Proxy-guided Image-based Rendering for Mobile Devices. *Computer Graphics Forum* 35 (2016), 353–362.

[41] Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible Rendering of 3-D Shapes. *Proc. SIGGRAPH* 24, 4 (1990), 197–206. doi:10.1145/97880.97901

[42] Gernot Schaufler. 1998. Per-Object Image Warping with Layered Impostors. In *Eurographics Rendering Techniques*, George Drettakis and Nelson Max (Eds.). Springer Vienna, 145–156. doi:10.1007/978-3-7091-6453-2_14

[43] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. 2000. Conservative Volumetric Visibility with Occluder Fusion. In *Proceedings of SIGGRAPH*. Association for Computing Machinery/Addison-Wesley Publishing Co., New York, 229–238. doi:10.1145/344779.344886

[44] Gernot Schaufler and Wolfgang Stürzlinger. 1996. A Three Dimensional Image Cache for Virtual Reality. *Computer Graphics Forum* 15, 3 (1996), 227–235. doi:10.1111/1467-8659.1530227

[45] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. 1998. Layered Depth Images. In *Proceedings of SIGGRAPH*. Association for Computing Machinery, New York, 231––242. doi:10.1145/280814.280882

[46] Heung Yeung Shum, Jian Sun, Shuntaro Yamazaki, Yin Li, and Chi Keung Tang. 2004. Pop-up light field. *ACM Transactions on Graphics* 23, 2 (2004), 143–162. doi:10.1145/990002.990005

[47] Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V. Sander, and Diego Nehab. 2008. An Improved Shading Cache for Modern GPUs. In *Proceedings of Graphics Hardware*, David Luebke and John Owens (Eds.). The Eurographics Association. doi:/10.2312/EGGH/EGGH08/095-101

[48] Wolfgang Tatzgern, Alexander Weinrauch, Pascal Stadlbauer, Joerg H. Mueller, Martin Winter, and Markus Steinberger. 2024. Radiance Caching with On-Surface Caches for Real-Time Global Illumination. *Proceedings of Computer Graphics and Interactive Techniques* 7, 3, Article 38 (2024), 17 pages. doi:10.1145/3675382

[49] Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg. 2002. Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics* 21, 3 (2002), 537–546. doi:10.1145/566654.566613

[50] Jay Torborg and James T. Kajiya. 1996. Talisman: commodity realtime 3D graphics for the PC. In *Proceedings of SIGGRAPH*. Association for Computing Machinery, New York, 353–363. doi:10.1145/237170.237274

[51] Andreas Alexandros Vasilakis, Konstantinos Vardis, and Georgios Papaioannou. 2020. A Survey of Multifragment Rendering. *Computer Graphics Forum* 39, 2 (2020), 623–642. doi:10.1111/cgf.14019

[52] Philip Voglreiter. 2024. *Handling real-world data in visual computing: algorithms and applications*. PhD dissertation, Graz University of Technology. doi:10.3217/ewbbg-6pp26

[53] Philip Voglreiter, Bernhard Kerbl, Alexander Weinrauch, Joerg Hermann Mueller, Thomas Neff, Markus Steinberger, and Dieter Schmalstieg. 2023. Trim Regions for Online Computation of From-Region Potentially Visible Sets. *ACM Transactions on Graphics* 42, 4, Article 85 (2023), 15 pages. doi:10.1145/3592434

[54] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image Quality Assessment: from Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612. doi:10.1109/TIP.2003.819861

[55] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. 2000. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. In *Proceedings of EUROGRAPHICS Workshop on Rendering*. Springer Vienna, Vienna, Austria, 71–82.

[56] Songyin Wu, Sungye Kim, Zheng Zeng, Deepak Vembar, Sangeeta Jha, Anton Kaplanyan, and Ling-Qi Yan. 2023. ExtraSS: A Framework for Joint Spatial Super Sampling and Frame Extrapolation. In *Proceedings of SIGGRAPH Asia*. Association for Computing Machinery, New York, Article 92, 11 pages. doi:10.1145/3610548.3618224

[57] Songyin Wu, Deepak Vembar, Anton Sochenov, Selvakumar Panneer, Sungye Kim, Anton Kaplanyan, and Ling-Qi Yan. 2024. GFFE: G-buffer Free Frame Extrapolation for Low-latency Real-time Rendering. *ACM Transactions on Graphics* 43, 6, Article 248 (2024), 15 pages. doi:10.1145/3687923

[58] Lei Yang, Yu-Chiu Tse, Pedro V. Sander, Jason Lawrence, Diego Nehab, Hugues Hoppe, and Clara L. Wilkins. 2011. Image-based bidirectional scene reprojection. *ACM Transactions on Graphics* 30, 6 (2011), 1–10. doi:10.1145/2070781.2024184

[59] Sipeng Yang, Qingchuan Zhu, Junhao Zhuge, Qiang Qiu, Chen Li, Yuzhong Yan, Huihui Xu, Ling-Qi Yan, and Xiaogang Jin. 2024. Mob-FGSR: Frame Generation and Super Resolution for Mobile Real-Time Rendering. In *Proceedings of SIGGRAPH*. Association for Computing Machinery, New York, Article 64, 11 pages. doi:10.1145/3641519.3657424

[60] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. 2018. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. In *Proceedings of CVF*. 586–595. doi:10.1109/CVPR.2018.00068

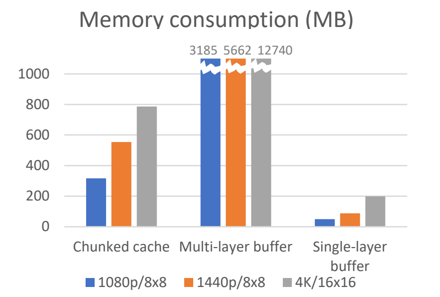

Fig. 8. Memory comparison of chunked cache, multi-layer buffer and a single-layer buffer (only frontmost samples). The chunked cache needs 4-6× more than a single layer, but stays <800 MB for 4K. A 4K multi-layer buffer needs 12 GB.

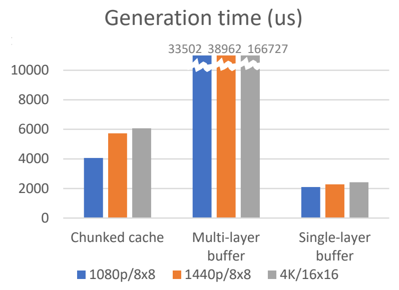

Fig. 9. Generating the chunked cache takes about 4-6 ms, which is 2.5× longer than generating a single-layer buffer. In comparison, generating a dense multi-layer buffer takes 33-166 ms.

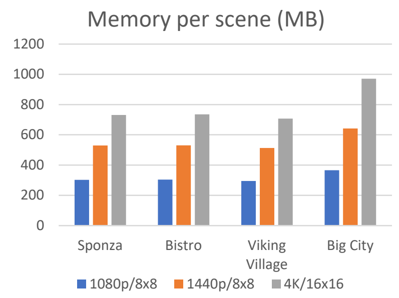

Fig. 10. Memory consumption of the chunked cache is rather insensitive to scene geometric complexity.

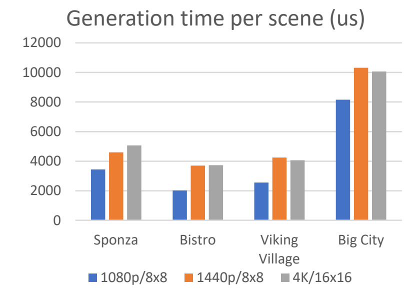

Fig. 11. Smaller scenes needs 2-4 ms to generate the chunked cache. Note that, depending on the frame extrapolation rate, even longer generation times can be amortized.

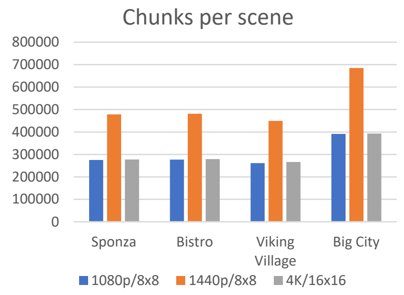

Fig. 12. Chunks per scene depend on the chunk size and the screen resolution. 1440/p8×8 requires twice the chunks than 1080p/8×8. 4K/16×16 requires the same chunks as 1080p/8×8.

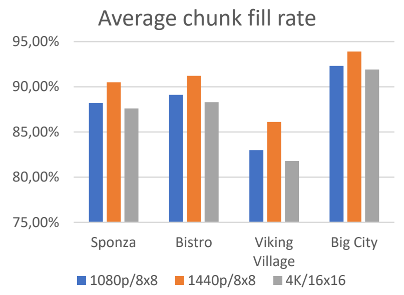

Fig. 13. The chunk fill rate, i.e., the number of filled samples per chunk is overall very high (about 80-92%) and does not vary much. Less than every fourth sample is left empty.

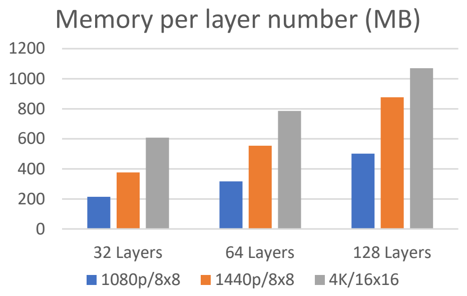

Fig. 14. Memory requirements of the chunked cache are sub-linear in the layer number. Doubling the layers increases memory only by 29-58%.

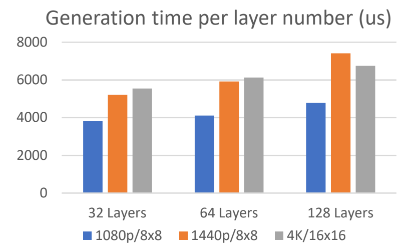

Fig. 15. Generation times of the chunked cache are sub-linear in layer number. Doubling the layers increases generation time by 8-16%.

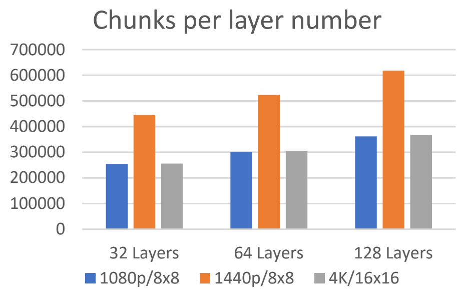

Fig. 16. The chunk number is sub-linear in layer number. Doubling the layers increases the number of chunks by 17-20%.

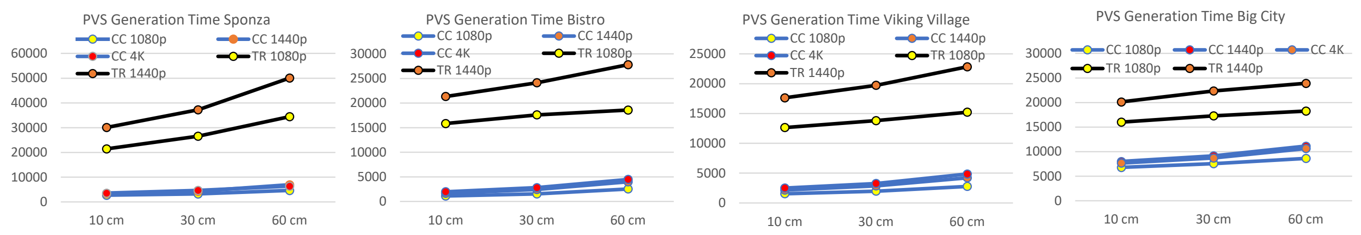

Fig. 17. PVS generation times, grouped by scene and organized by viewcell size (10, 30 and 60 cm). We compare our Chunked Cache (CC) to the state-of-the-art method Trim Regions (TR). TR could not support 4K resolution.

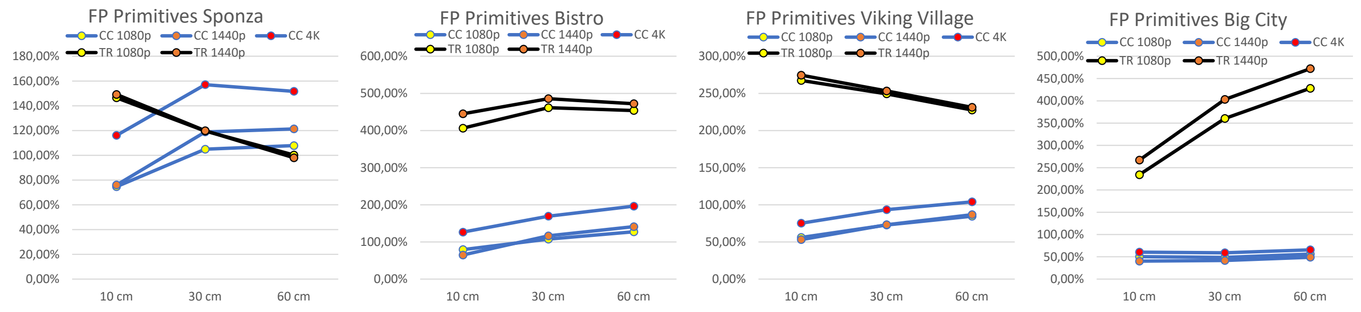

Fig. 18. False positive primitive rate in the PVS: We report the fraction of primitives unnecessarily included in our PVS compared to the ground truth PVS, averaged over all scenes and animation paths.

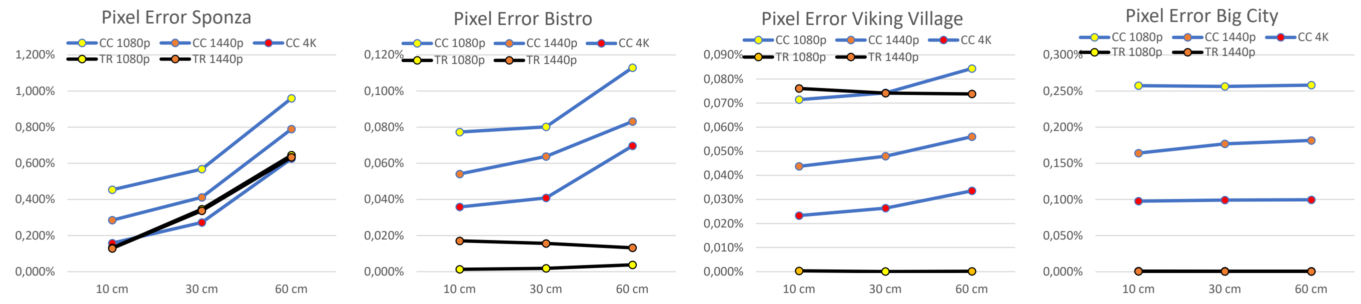

Fig. 19. Pixel error rate: We report the fraction of incorrect pixels observed in all scenes and animations paths compared to the screen resolution.

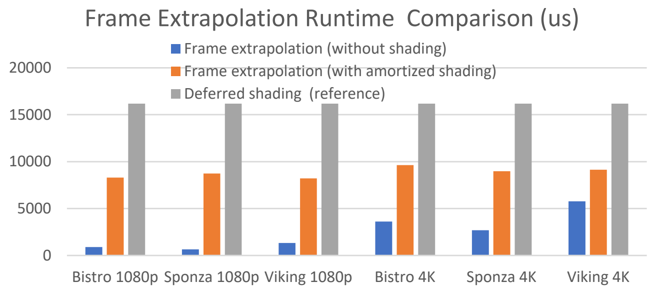

Fig. 20. Frame extrapolation time. We compare frame extrapolation (without and with amortized shading) to a standard deferred renderer. The shading load was chosen so the deferred renderer reached 60 Hz for each target resolution.

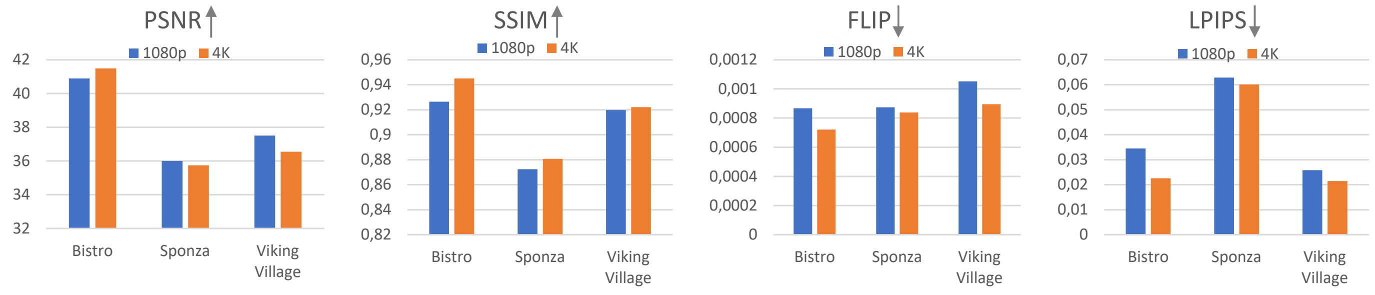

Fig. 21. Image quality of extrapolated frames. We report four metrics: PSNR, SSIM, FLIP and LPIPS. Higher values are better for PSNR and SSIM; lower values are better for FLIP and LPIPS.