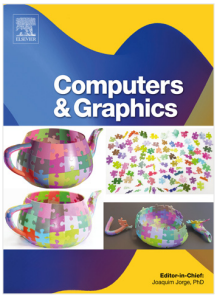
Journal Pre-proof



Virtual memory for 3D Gaussian Splatting

Jonathan Haberl, Philipp Fleck, Clemens Arth

PII: S0097-8493(26)00069-5
DOI: https://doi.org/10.1016/j.cag.2026.104598
Reference: CAG 104598

To appear in: *Computers & Graphics*

Received date : 9 January 2026
Revised date : 6 April 2026
Accepted date : 17 April 2026

Please cite this article as: J. Haberl, P. Fleck and C. Arth, Virtual memory for 3D Gaussian Splatting. *Computers & Graphics* (2026), doi: https://doi.org/10.1016/j.cag.2026.104598.

Graphical Abstract

Virtual Memory for 3D Gaussian Splatting

3D Gaussian Splatting of the well known truck scene [1], render using the original implementation *vs.* ours running on mobile together with our generated proxy mesh to determine visibility.

Highlights

Virtual Memory for 3D Gaussian Splatting

- a novel approach to virtual memory and Level of Detail (LOD) within 3D Gaussian Splatting (3DGS)
- an implementation for preprocessing and real-time rendering using a modern rendering API
- an efficient method for rendering large scenes on mobiles
- a detailed evaluation of our approach on desktop and mobile devices

Virtual Memory for 3D Gaussian Splatting

Abstract

3D Gaussian Splatting represents a breakthrough in the field of novel view synthesis. It establishes Gaussians as core rendering primitives for highly accurate real-world environment reconstruction. Recent advances have drastically increased the size of scenes that can be created. In this work, we present a method for rendering large and complex 3D Gaussian Splatting scenes using virtual memory. By leveraging well-established virtual memory and virtual texturing techniques, our approach efficiently identifies visible Gaussians and dynamically streams them to the GPU just in time for real-time rendering. Selecting only the necessary Gaussians for both storage and rendering results in reduced memory usage and effectively accelerates rendering, especially for highly complex scenes. Furthermore, we demonstrate how level of detail can be integrated into our proposed method to further enhance rendering speed for large-scale scenes. With an optimized implementation, we highlight key practical considerations and thoroughly evaluate the proposed technique and its impact on desktop and mobile devices.

Keywords: Gaussian Splatting, Computer Graphics, Mobile Computing

1. Introduction

In Novel View Synthesis (NVS), a set of pictures of a scene is taken and used to create new images. These images may be generated from arbitrary viewpoints, beyond those included in the original set of pictures. The objective is to make these synthetic images convincing; seemingly taken in the original scene.

3DGS uses neither mesh nor texture. Its only primitive is a 3D Gaussian, while the collectivity of 3D Gaussians represents a scene accessible in CPU memory. As the scales of 3DGS scenes increase, so does the memory and work required to store and render them. Depending on the actual size of the scene, the amounts of data can be huge, multiple gigabytes or terabytes even. Handling 3DGS reconstructions requires graphics engines with high memory bandwidth along with huge amounts of memory, which seriously limits the application of 3DGS and puts its use on even modern mobile computers out of reach. Therefore, novel ideas and the invention of new methods to tackle these problems are of high relevance.

The method introduced in this work is concerned with these aspects of efficient data handling and rendering of large scenes. We propose the use of virtual memory and LOD which help to achieve virtually unlimited scene scale and efficient rendering, even on low-end mobile devices.

At a glance, we make use of the concept of **virtual memory applied to Gaussian splatting**. To overcome the sheer amount of data (3D Gaussians) in large-scale reconstructions, we group Gaussians by their proximity to place them on virtual pages (similar to virtual memory [2]) of fixed sizes. Furthermore, we can link geometrically linked pages together, to guarantee preemptive loadings of content to the GPU. This approach goes beyond traditional frustum culling, since neighboring regions are already present in memory. By parameterizing page sizes regarding Gaussians and physical memory bandwidth, such as the preferred block size, we achieve a performance similar to that of higher-end devices.

Figure 1: 3D Gaussian Splatting of the well known truck scene [1], render using the original implementation *vs.* ours running on mobile together with our generated proxy mesh to determine visibility.

Our approach is split into two parts: a **pre-processing** and a **real-time rendering** part. In the pre-processing stage, an existing 3DGS scene is analyzed offline to create a mesh (namely, the *proxy mesh*) that approximates the structure present in the scene. Gaussians are separated into *pages* (pages as in virtual memory) that group them by their proximity. Each face on the proxy mesh is marked with the corresponding page ID. Where pages overlap, links between them are established. Finally, lower LOD levels for each page are provided. The proxy mesh has **not contributed** on *how* and *where* the Gaussians are rendered, but dictated which Gaussians are assigned to which pages (Figure 1), and which pages are linked to others. Moreover, the accuracy of the proxy mesh is only of a secondary nature, since it only determines the Gaussians placed together.

The second part of our method is concerned with rendering scenes in real time. The preprocessed scene is imported and for each frame, we first render the page IDs on the proxy mesh to a visibility buffer. The sole purpose of this buffer is to determine the IDs of the pages in view of the camera. The page IDs contained in the image, as well as their established links, allow us to determine which pages of Gaussians will likely be visible. The pages are then transferred to GPU memory and rendered

using a modern rendering API.

Note that **we do not modify the process of scene creation**. Any scene in the format described by Kerbl *et al*. [1] is compatible with the approach presented. This includes the original implementation, similar derivative implementations, and at least a subset of more recent implementations that focused on increasing the size of reconstructed scenes. Several solutions have been proposed to reduce memory usage or modify the attributes of 3DGS scenes. Many of these modifications may generally be used in combination with our work in the future. However, we limit our exploration of such extensions to a discussion of 3DGS compression in related work.

In the following, we provide details about the concepts involved, from preprocessing to per-frame operations. We explore several alterations to the basic idea to improve quality and performance or limit memory usage and highlight the resulting trade-offs. An end-to-end implementation is provided, together with an in-depth evaluation. Our contributions can thus be summarized as follows:

- a novel approach to virtual memory and LOD within 3DGS;
- an implementation for preprocessing and real-time rendering using a modern rendering API;
- an efficient method for rendering large scenes on mobiles; and
- a detailed evaluation of our approach on desktop and mobile devices.

2. Related Work

In this section we focus on the related work in 3DGS, which is relevant to our approach. Due to the vast amount of continuously published work, compiling a comprehensive overview is impossible. We also abstain from discussing the mathematical basics of 3DGS for brevity and refer the interested reader to the work of Kerbl *et al*. [1] for details. Because we introduce virtual memory to 3DGS in our work, we first briefly introduce the basics and then focus on related work required to understand the concepts. Thus, we bridge the gaps between 3DGS and virtual texturing.

2.1. 3D Gaussian Splatting

In NVS, research is divided mainly into two state-of-the-art approaches: Neural Radience Fields (NeRF) and 3DGS. While there are similarities, the latter was introduced by Kerbl *et al*. [1] to render radiance fields, representing scenes using an explicit scene representation. 3D Gaussians, stored as their ellipsoid equivalents, are defined by position, rotation, scale, opacity, and view-dependent color in the form of spherical harmonics. This representation allows them to produce accurate images that can be rendered in real time on desktop hardware and edited easily. The images are rendered with a tiled software renderer implemented in Compute Unified Device Architecture (CUDA).

Taking inspiration from the initial proposal of 3DGS, researchers have explored a variety of further avenues and details, some of which are relevant to the concept proposed in this work.

2.1.1. Mesh Extraction

There are multiple reasons to create a triangle mesh from 3DGS scenes. The mesh may replace the scene representation entirely, or triangle meshes may support an application such as providing a foundation for physics interactions. The conventional method to extract a triangle mesh from a 3DGS reconstruction is to apply the Marching Cubes algorithm, proposed by Lorensen and Klein [3]. The algorithm converts 3D data to triangle meshes depicting constant-density surfaces.

Chen *et al*. [4] introduce a regularization term to 3DGS, encouraging thin Gaussians to better align with surfaces. Gaussians, as well as a neural network learning the scene's Signed Distance Function (SDF), are jointly optimized to do this. Guédon *et al*. [5] introduce a similar regularization term. Waczyńska *et al*. [6] use a parameterization that defines Gaussians by their position on a mesh. Both the mesh and its Gaussians are optimized together. The idea of a joint representation for mesh and Gaussians is especially promising for deformation and has also been explored in that context by Yuan *et al*. [7] and Gao *et al*. [8].

We don't aim to advance the field in this direction, rather we resort to using the Marching Cubes algorithm as a baseline for our work to also create a mesh for our purposes.

2.1.2. Compression

The concept of 3DGS implies the storage and handling of large amounts of data that increases significantly with the reconstruction scale. Kerbl *et al*. [1] use 248 bytes to represent a single Gaussian. Most of this storage is consumed by the spherical harmonics that represent colors.

Pranckevičius [9] creates groups of Gaussians by their positions, whose properties are stored in patches in 2D textures. Morgenstern *et al*. [10] present an algorithm to efficiently sort millions of Gaussians by their properties in seconds to create 2D textures. Later, Pranckevičius [11] proposes using vector quantization with k-Means on spherical harmonics. Fan *et al*. [12] determine the importance of a Gaussian in a scene and use vector quantization on the less important Gaussians. Niedermayr *et al*. [13] determine the sensitivity of images to changes in properties to perform sensitivity-aware quantization. Vector quantization generates codebooks with an index to the respective entry for each Gaussian. Navaneet *et al*. [14] uses a form of Run-Length Encoding (RLE) on these indices, while Niedermayr *et al*. [13] compress them with the DEFLATE algorithm, which also employs a variation of RLE.

Girish *et al*. [15] quantize Gaussian properties, then decode those with a Multi-layer Perceptron (MLP) to recover the initial properties. Li *et al*. [16] train an MLP to return a final color in dynamic scenes based on a base color, a view direction and time. Lee *et al*. [17] determine a feature and a view direction from positions with a hash grid and use those in an MLP to determine the color.

We do not employ any such compression, in general, but our approach can be used alongside several of those extensions.

Figure 2: Scene preprocessing and real-time rendering steps: A preprocessing stage prepares the scene offline. Real-time rendering steps are performed for every rendered frame.

2.1.3. Large Scene Reconstruction

3DGS gains most of its popularity through its scalability to large scenes and the visual appeal of rendered imagery, albeit Kerbl *et al*. [18], complain about the lack of available huge datasets for further enhancements. Certainly, increasing the scale of reconstructions introduces a new range of challenges related to the amount of data that can be adequately stored in physical memory, making it essential to partition the reconstruction at some stage.

Lin *et al*. [19] propose a dataset and an approach to partitioning large scenes, splitting a scene into chunks based on the distribution of camera positions and the respective Structure from Motion (SfM) reconstruction. Liu *et al*. [20] split scenes into uniform chunks and prune chunks with low contribution to rendering. Kerbl *et al*. [18] define a low-resolution skybox on a sphere surrounding each chunk, defining the surrounding scene as background. Zhao *et al*. [21] follow a different approach, distributing Gaussians across multiple GPUs, assigning pixels to be rendered to GPUs in tiles, and transferring the Gaussians between them as needed. Dynamic load balancing can make distributed training efficient and scale to up to 32 GPUs. Later, Meulemann *et al*. [22] introduce *on-the-fly* to efficiently process unposed data in an SfM manner and pre-estimate 3D Gassians while training, leading in a more efficient overall performance. Both Liu *et al*. [20] and Kerbl *et al*. [18] make their code available for scene reconstruction. However, the scenes created by Liu *et al*. [20] are not available, and Kerbl *et al*. [18] introduce a new file format to incorporate their LOD implementation. We use a selection of the scenes from Lin *et al*. [19] for evaluation in Section 5. However, because of the differences in formats, direct comparisons to other implementations are generally challenging.

2.1.4. Streaming

In their initial work, Kerbl *et al*. [1], only perform frustum culling during rendering. A tiled compute-based software rasterizer determines whether a Gaussian overlaps with a tile before sorting Gaussians for each tile individually. To do this, the data for Gaussians need to be in GPU memory. Jiang *et al*. [23] divide a scene into voxels. They determine which voxels lie in the view frustum using a KD-Tree and subsequently transfer these to the GPU. Kerbl *et al*. [18] asynchronously transfer high-detail Gaussians to replace those with lower detail as part of their LOD solution.

The use of streaming, or on-demand transfers, is still somewhat limited in current research. If it is used, the decisions for which Gaussians to copy is generally made based on the view-frustum, instead of any more sophisticated visibility determinations. In contrast, we perform visibility determination using a visibility buffer. Culling based on both the view frustum and occlusions are the result.

2.1.5. Level of Detail

Reducing the overall number of Gaussians to be rendered can significantly boost performance, while the most common approach is to merge multiple Gaussians. Yan *et al*. [24] propose combining Gaussians to solve aliasing problems in 3DGS. Gaussians that are too small during rendering are culled and replaced with larger ones. Kerbl *et al*. [18] introduce a hierarchy of Gaussians, merging them based on the contribution of each one to its parent. Their hierarchy allows them to choose which layer and therefore LOD level to render at runtime.

Liu *et al*. [20] do not reduce the number of Gaussians but compress those further away from the camera. Vector quantization proposed by Fan *et al*. [12] reduces the visual fidelity of less important Gaussians. Lu *et al*. [25] create anchors in a scene, spawning neural Gaussians around each that can be dynamically adapted to the distance. Ren *et al*. [26] create an octree for a scene, where each level represents an LOD level, each containing anchors spawning additional neural Gaussians.

We base our LOD solution on Yan *et al*. [24]. However, unlike most of the described methods, we do not modify Gaussians during scene reconstruction.

Related to our approach, Wang and Sin [27] propose a method binding 3D Gaussians in texture space to a proxy mesh and re-projecting them back to world space through implicit shell mapping. Their work focuses on animation, in contrast to our approach, which puts an emphasis on memory footprint for mobile devices.

Timely coherent with our work, RTGS [28] and the work from Papantonakis *et al*. [29] were published. RTGS achieves real-time performance on mobile devices by pruning of Gaussians to improve rendering, paired with foveated rendering to reduce the overall amount of Gaussians rendered. In contrast, we focus on a more fundamental problem, namely improved memory management. A combination of RTGS with our method would further push the community forward. On the other hand, Papantonakis *et al*. reduce the memory footprint by removing redundancy through pruning. While their work focuses on compression, it can be considered a supplement to our work, since we focus on the underlying structure. A mixture of said methods can achieve extremely large scenes to be rendered on low-end devices, reaching a new sense of scale, for example, in games.

In summary, for the current state of mobile devices, we provide a runtime applicable solution to the hard memory limitations on such device, enabling large scale Gaussian splatting. The underlying novelty in addition to the application of virtual memory is the adaptability in how memory is managed, providing a potential good fit for many devices.

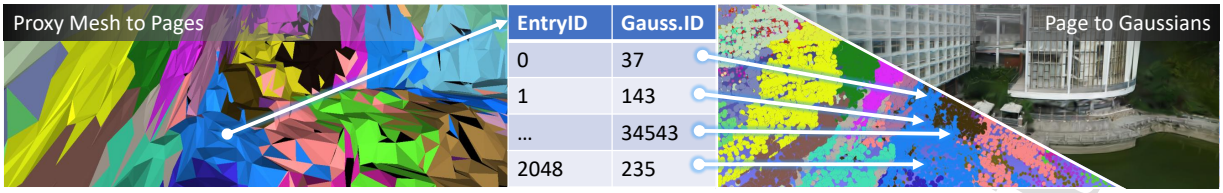


Figure 3: *Mesh to Render:* The proxy mesh is used to determine the visibility of Gaussians using pages (virtual memory), which is are used to render the final representation.

2.2. Virtual Texturing

Virtual texturing is the GPU version of virtual memory, using GPU-Textures as buffers instead of CPU memory. The concept of virtual memory separates virtual and physical address spaces. Memory addresses used by a process in userspace do not hint at actual locations in system memory but must be translated first. These translations query a page table, which maps pages of addresses in virtual memory to respective regions in physical memory once a process uses them. These pages are in regions of equal size in memory. The same system is used in modern GPU architectures. Applications can create buffers or textures that are not bound to any physical memory initially. Regions of these can be manually bound once they are required. This functionality is exposed to applications through modern APIs as sparse residency (Vulkan) or reserved resources (Direct3D 12); however, support is platform dependent.

Barrett [30] demonstrates how the concept can be applied to texturing, splitting textures into tiles (pages) of equal size. These tiles are generally not resident in GPU memory if they are not visible. At runtime, the scene is first rendered with tile IDs only, and visible tiles are copied to the GPU if necessary. An indirection texture points to the physical tile containing the required page. Mayer [31] provides a detailed overview and analysis of virtual texturing methods.

Virtual texturing forms the foundation for our work. We use these ideas to determine which pages are visible and transfer them to the GPU if necessary. Similarly to how mipmapping is integrated into virtual texturing, we integrate an LOD solution, and apply both concepts for the first time to improve 3DGS.

3. Preprocessing for 3DGS using Virtual Memory

Our virtual memory solution, inspired by virtual texturing, uses a proxy mesh to approximate a 3DGS scene. Gaussians in the scene are grouped into pages with a respective ID. Rendering the proxy mesh quickly identifies which Gaussian pages are visible to the camera.

In Figure 2 we provide a visualization of the steps involved with the method, while this Section is concerned with the left part only. The right part of Figure 2 is discussed in Section 4.

3.1. Mesh Extraction

A number of different approaches for mesh extraction have emerged over recent years and decades. Some modern approaches have already been directly integrated into Gaussian splatting algorithms. Recently, Gaussian Opacity Fields [32], Sorted Opacity Fields [33] and Sugar [34] have taken advantage of Gaussians and their representation to create depth maps and meshes as part of the training process. Other methods, such as Triangle Splatting [35, 36] omit the idea of quads and render Gaussians as triangles of native 3D engine support. A biproduct is a mesh-like representation which can be used for collision detection in real-time applications, such as games. In contrast, more traditional approaches dating back to the 2000s and earlier have been integrated in *e.g.* AliceVision [37] for SfM reconstruction, such as the Marching Cubes [38] algorithm, for example.

To render a scene, we first need a simplified mesh with fewer primitives for faster processing while maintaining enough faces to prevent page occlusion. Since we use default methods, such as Kerbl *et al.* [1] to create the Gaussians, those serve as input to the mesh extraction, as shown in Figure 2. The resulting proxy mesh just has to be ”good enough” to determine the page affiliations of Gaussians and to facilitate the page linking of related pages. As long as the generated proxy mesh remains within a usable range, the method to create it is interchangeable and of lesser importance for our proposed pre-processing pipeline.

We employ Marching Cubes for efficient surface reconstruction due to its memory efficiency and its high potential for parallelism, which allows processing of very large scenes on even mid-tier hardware. The process is inspired by medical devices, which create images depicting slices of physical objects. These medical applications were the first use case of the marching cubes algorithm. Following the method of Hartmann [39], we define a plane that is normal to the Z-axis to create a slice of the scene. This plane is intersected with the scene and is rendered as an image. A pixel receives a value based on whether or not its corresponding position lies inside a Gaussian (using its equivalent ellipsoid representation). We repeat this process while moving the plane along the Z-axis in fixed steps for subsequent slices. This process can also be thought of as sampling a uniform 3D grid of points in the scene and determining whether they lie inside or outside of any Gaussians in the scene. The data produced can be used as input for the marching cubes algorithm for surface reconstruction. Before we reconstruct the surface, we perform morphological operations. The intent is to make the final mesh coarser, close small holes, and reduce unnecessary details. These operations and kernels are tailored per scene.

3.2. Page Assignment

To minimize the overhead of checking Gaussian visibility, we group them into pages. A proxy mesh face can correspond

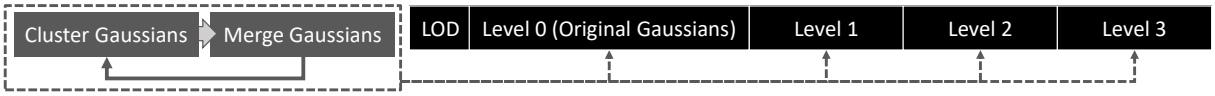


Figure 4: LOD Generation: Visualization of LOD generation for four total levels (including the original Gaussians as level zero). The number of Gaussians per page is halved for each subsequent level. Merging Gaussians is done with an average for all properties but their scale.

to none or multiple pages, while a page can cover multiple faces (Figure 3). Each page holds Gaussians up to a specified maximum. Initially, each mesh face gets a separate page, and Gaussians are assigned to the page of the face nearest to their mean. If a face gets more Gaussians than the page size allows, it is subdivided by adding a vertex at an edge's midpoint, forming two triangles. This is repeated until each face has no more Gaussians than the target page size.

To prevent thin triangles from repeated subdivisions, we rotate the order of vertices and choose different edges for subsequent subdivisions. This method is not optimal because shared edges may create vertices on adjacent triangles, leading to T-junctions and rendering artifacts due to floating point inaccuracies. However, it is fast, sufficiently simple to implement and can easily be replaced by a more sophisticated algorithm.

Initial page assignments group nearby Gaussians but create small pages. We merge neighboring pages to approach the target size without exceeding it, using a greedy breadth-first method. Pages are first merged with neighboring faces. If a page remains under the target size and no neighbors are left, a nearby page is chosen. If adding a page exceeds the target size, it becomes the basis for a new merged page to which more pages are added.

The initial pages, one per face, are greatly reduced. A triangular mesh face maps to one or no page, marked with a page ID for rendering if included. Unmarked faces lack a Gaussian mean match during initial assignment. Gaussians are grouped in memory once assigned to pages, with zero-property Gaussians added as padding to reach page size, easily culled during rendering.

3.3. Page Linking

In 3DGS scenes, millions of overlapping ellipsoids don't align with surfaces like textures do with mesh faces. We assign Gaussians to pages based on their means, without considering nearby interactions. Ellipsoids extending into surrounding pages can incorrectly be marked as invisible. We address this by using page linking, identifying overlapping pages after initial assignment to establish uni- or bi-directional links. When a page is visible, **all linked pages must also be present in GPU memory**. To naively find pages that need linking, each ellipsoid is tested for intersections with others. Intersecting ellipsoids on different pages require links. Lacking acceleration structures, this method has $O(n^2)$ complexity, leading to trillions of tests even for small scenes.

Instead, we generate random points within an ellipsoid and identify the nearest face on the proxy mesh for each. A link is needed if the face's page is different from the ellipsoid's assigned page. This approximate method tests only the closest face on the proxy mesh. This step is part of our algorithm for initial mesh assignment, allowing optimized page assignment and linking. Random points are generated uniformly in a unit sphere using spherical coordinates and converted to Cartesian coordinates. Applying Gaussian scale, rotation, and translation transforms these points to lie within an ellipsoid. The final distribution is non-uniform due to initial sampling in the unit sphere, not accounting for the ellipsoid's properties.

After assigning Gaussians to pages, a face on the proxy mesh may match one page. If the closest face for a sample belongs to a different page from its Gaussian counterpart, a link is established. If a mesh face isn't yet assigned to a page, it is marked with the page having the most sample points close to it. The page with the most overlap takes precedence, while other overlapping pages are linked. The number of page links indicates the quality of the page assignments, which should be compact and minimize overlaps. This algorithm has $O(n)$ time complexity; however, future work may further refine assignments based on these links.

3.4. LOD Generation

In mesh rendering, a mipmap is a texture with multiple precomputed downscaled versions. As surfaces move further from the camera, lower resolution levels are used, improving performance by reducing texel fetches and offering anti-aliasing through bilinear filtering. Early virtual texturing, such as Mittring *et al*. [40], relies on mipmaps. The mip level is chosen when transferring a new texture to the GPU.

Mip level selection is performed on the basis of derivatives. These are computed by rendering in groups of two by two pixels (quads). All pixels in such a quad are rendered for a primitive, even if the primitive only overlaps a single pixel. Pixels that do not overlap are helper pixels, executed in helper lanes, and are subsequently discarded. This can cause an overhead of up to three additional pixels per pixel rendered to the output image. LOD offers a solution with a mesh having various complexity levels, switchable based on camera distance, minimizing small primitives and reducing helper lanes, thus reducing the impact on performance.

Yan *et al*. [24] propose a method for 3DGS using mipmapping and LOD, merging Gaussians during training to create detail levels and reduce aliasing. Our LOD solution follows a similar idea (Fig. 4), but focuses on minimizing the Gaussians to be copied, stored in GPU memory, sorted, and rendered, without altering the scene creation stage. The objective is to merge Gaussians while minimally impacting their distant appearance, while only Gaussians on the same page can be merged. Their attributes are points in multidimensional space, making the task one of clustering these points by similar attributes such as position, rotation, and color. We identify clusters using the k-means algorithm, while scaling attributes adjusts their influ-

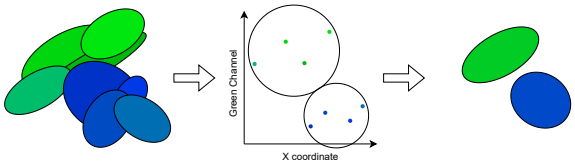


Figure 5: Gaussians are clustered based on their properties, then averaged. This illustration clusters ellipses based on only their x coordinates and green channels. The ellipses are then averaged such that each cluster becomes one new ellipse.

ence. Fig. 5 illustrates the concept by clustering and averaging ellipses based on two properties. The resulting clusters represent new Gaussians, which are directly used in LOD loading.

Merging multiple Gaussians is straightforward for most attributes using the arithmetic mean for translation, rotation, opacity, and spherical harmonics. However, as Yan *et al*. [24] note, merging scales can cause the resulting Gaussian to be too small. To remedy this, additional Gaussians are adjusted using average pixel coverage during training. We scale up the merged Gaussian by a fixed factor to fill gaps. In clustering, we prioritize position attributes to prevent dispersed clusters.

3.5. Scene Storage

The original 3DGS implementation exports generated data in Polygon File Format (PLY). Other applications, such as Nerfstudio [41], have followed suit. The data describing such a scene consist of a list of Gaussians with 62 properties each. Originally, the encoding ensures quality despite numerical errors. Properties must be decoded from disk to display.

Our virtual memory solution streams Gaussians to GPU memory when needed, possibly loading just in time. Thus, we eliminate encoding and adjust the file format to stream data directly to the GPU. These modifications to the original format include (a) removing obsolete normal properties, (b) reordering spherical harmonics data for better cache-locality, (c) decoding the opacity property, (d) decoding the scale properties, and (e) normalizing the quaternion describing the rotation. We defer compression, such as converting spherical harmonics to half floats, to future work. Coordinate conversions, if needed, can be done by adjusting position, scale, and rotation. We limit the 3DGS scene to a central cube, excluding Gaussians with means outside it.

The mesh is stored alongside the modified scene file, including a list of page IDs (with each entry corresponding to a mesh face), a buffer containing page links, and metadata such as page size, number of LOD levels, and more.

4. Real-Time Rendering

Modern applications require at least 60 Frames per Second (FPS), or frame times below a certain threshold, *e.g.* 16 ms. Virtual reality demands even higher rates to prevent nausea. Integrating rendering with other processes (*e.g.* physics simulations, pathfinding) can reduce the time available for each frame. Our real-time rendering concept depicted in Figure 7 allows for low-level optimizations. Parallelizable tasks are handled in thread pools or with compute shaders.

Starting with the scene description for our renderer stored in memory, the proxy mesh loads instantly, but Gaussians use a memory-mapped file placed in virtual memory, with physical memory accessed on a page fault. Although physical access is slower and generally undesirable, this method allows 3DGS scene size to exceed system memory limits.

4.1. Page Determination

Before rendering a frame, we must determine which pages are visible from the camera. The aim is to cull Gaussians lying outside the view frustum and those occluded by other Gaussians. The Gaussians assigned to these visible pages need to be copied if they are not yet in GPU memory.

For virtual texturing, Barrett [30] suggests rendering scenes with page IDs. As described in the previous section, we adopt this idea by marking our proxy mesh faces with page IDs, generating a list of visible pages from the image. Inspired by operating systems, we use a page table in which each entry maps to a physical memory page. After listing the required pages, new ones replace the unused physical pages, prioritizing the least recently used. Once visible pages are loaded into GPU memory to render the scene, we sort visible Gaussians by camera distance for alpha blending using RadixSort.

Page Links. We request pages via page-links (Section 3.3) where we preprocess scenes to identify overlapping pages. Each page's visibility buffer includes its overlapping pages, and marking them as required minimizes visible artifacts.

Level of Detail. We add the distance of the nearest pixel to the visible pages list with its page ID. As the page table updates and new pages are copied, a suitable LOD is chosen based on this distance. This resembles choosing a mip level in virtual texturing.

Barrett [30] segments the mip levels into uniform-sized pages. In contrast, we reduce Gaussians with each LOD, allowing multiple virtual pages within one physical page. We mark page table entries with their current LOD for this reason. Thresholds, which can be dynamically updated based on memory usage, move away from the camera when usage exceeds a limit and move closer when memory is available.

However, setting the initial threshold in fixed increments is problematic if the step size is too large (a single step may affect memory usage drastically) overshooting the targeted region. This can lead to rapid flipping between two levels, causing noticeable artifacts, such as popping and flickering. Similarly, differences in scene scale can make a step size too large, leading to the above mentioned issue, or too small, making the system slow to adapt. We therefore dynamically adjust the step size based on previous moves.

4.1.1. Page Table Management

To track pages in GPU memory, we use a page table with entries describing the physical page contents, detail level, last use,

Figure 6: Simplified overview of our pipeline used to render Gaussians. A vertex shader is supplied with a sorted index buffer, which it uses to retrieve properties for its respective Gaussian from a storage buffer. If the device is capable of a geometry stage, a geometry shader is used to construct a rectangular quad. Alternatively, a point primitive is used. A fragment shader draws an ellipse onto the quad. Finally, the quads are combined via alpha blending to create the final image.

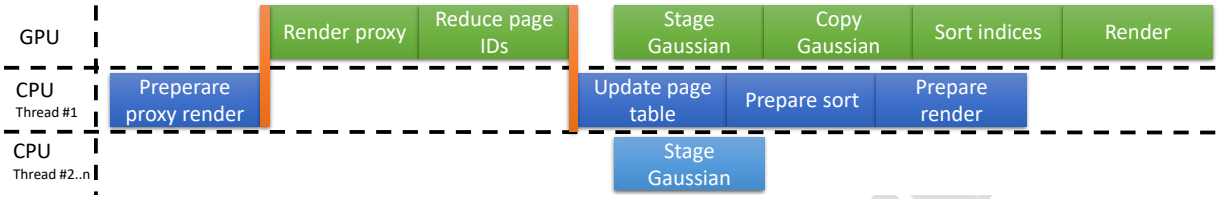


Figure 7: High-level overview of the stages of rendering with virtual memory. Orange bars indicate synchronization between CPU and GPU. Each step additionally depends on the previous step in the same lane. The tasks are not scaled with the time it takes to complete them.

necessity for the frame, and a LOD-dependent list of contained pages. We selected a single array structure over a hierarchical one for better memory locality.

Based on the list of required pages from page determination, the page table is updated in two steps: First, its entries are traversed and physical pages that are required to render the frame are marked. Free physical pages, referring to physical pages that contain no valid page or pages that are not required, are stored along with the frame in which they were last used. The list of required pages is simultaneously updated, removing pages already present in the page table at the correct LOD. Second, the system checks the updated list of required pages, which contains page IDs to render the frame, none of which are in GPU memory. If there is space, these pages are copied to a staging buffer, updating the page table. Free physical pages are chosen based on the least-recently-used order, and the contents of the staging buffer are copied into a Gaussian buffer accordingly. The list of physical pages is written to a buffer to indicate required pages to the renderer. The compute shader is adjusted to convert these pages into sortable indices based on camera distance.

We use a regular buffer, skipping the sparse residency feature described in Section 2, for three main reasons:

- in virtual texturing, textures are addressed using UV coordinates, typically needing translation via an indirection texture, adding a lookup overhead. We avoid this lookup and use an index buffer to indicate Gaussians for drawing, which must be sorted regardless of sparse buffers;
- only a few newer high-end desktop GPUs support it. The hardware capability database [42] indicates that less than a third of GPUs permit the `sparseResidencyBuffer` feature. For Android devices, support is just over 10%. MoltenVK [43], which bridges Vulkan and Metal to support Apple devices, generally lacks this feature; and
- even on recent hardware with sparse residency, performance seems to be unsatisfactory as mentioned by Richermoz [44].

4.2. Rendering

There are generally two approaches to rendering 3DGS scenes. Kerbl *et al*. [1] use CUDA to create a software renderer. The image is split into tiles, determining overlapping Gaussians. The Gaussians of each tile are sorted and then drawn to the output image. In contrast, our implementation sorts Gaussians globally in a OpenGL Shading Language (GLSL) compute shader before splatting the Gaussians onto quads rendered using the hardware rasterizer.

We follow the pipeline depicted in Figure 6 to create quads representing Gaussians. By default, commands are recorded and submitted to draw point primitives. This dispatches a vertex shader for every Gaussian, performing the necessary computations, and geometry shaders create the quads. The fragment shader discards pixels outside the splatted Gaussian. However, if geometry shaders are unavailable, we can fall back to a pipeline, omitting the geometry stage. The quads are then the result of the point primitives, passed to the rasterizer. Their square shape can lead to a significant increase in fragment shader invocations, leading to further discards.

Depth Sorting. Gaussians with opacity require alpha blending and must be sorted by distance from the camera. The rendering order is determined by constructing and globally sorting the index buffer with one index per Gaussian. Compute shaders are used, where one calculates distances from the camera, and another uses a radix sort to reorder indices, and finally, indices are sent to the vertex shader to access attributes from a storage buffer.

Shaders. The math for drawing each Gaussian remains substantially the same as presented by Kerbl *et al*. [1]. The majority of the required computations, are performed once per Gaussian. They can therefore be performed in the vertex shader. The fragment shader is then responsible for shading the projected Gaussian within a quad based on a 2D covariance matrix.

4.3. Limitations and Options for Optimization

In the following, we discuss some limitations and options for improvement of our proposed method. Where applicable, we also highlight how to overcome them.

Proxy Mesh Rendering. Once the proxy mesh image is rendered, it is reduced to a buffer indicating the visibility of each page. Hable [45] discusses the negative performance impact of `gl_PrimitiveID` and suggests using the leading vertex to determine triangle primitive IDs. A compute shader reduces the visibility buffer to a list of IDs. To communicate additional LOD levels, the distance to the closest pixel is stored. Since finding this minimum can lead to race conditions, atomic operations are employed with an adverse performance impact on concurrent thread updates. To enhance performance, atomic operations should be avoided in favour of using shared memory in workgroups.

We mitigate these shortcomings by rendering the visibility buffer at low resolution. Mayer [31] demonstrates how smaller visibility buffers improve performance with virtual texturing. Although a low resolution can cause distant pages to be missed in virtual texturing, in our renderer overlapping Gaussians cause nearby pages to remain memory-bound.

Pagetable Array. Our page table is a single array of entries, each managing a physical page, which may include multiple pages depending on the LOD level. For efficiency reasons, we avoid hierarchical page tables found in operating systems to reduce indirections and potential page misses.

More work is needed to shrink the page entries and test the scalability with a larger buffer size for rendering. The page table entries, among other parts of the system, would also require modification to allow blending between LOD levels. During the transition, both levels are needed in memory simultaneously.

LOD Transitions. Both LOD transitions and adding linked pages are less urgent than rendering visible Gaussian pages. The copy budget is limited by the staging buffer size and the time required.

To maintain quality under high load, we prioritize important copies and delay LOD transitions to later frames.

Frame Synchronization. Strict synchronization during frame rendering causes pipeline stalls. The GPU could do more work but waits for task completion.

Efforts to separate steps and perform tasks asynchronously could speed up rendering, especially data copying from CPU to GPU.

Low-End Hardware. Mobile devices, including those by Apple, generally contain integrated GPUs nowadays. Memory is shared between CPU and GPU.

To take advantage of this properly, modifying our concept could avoid streaming any data overall. Instead, memory could be mapped from the file directly to the buffer and used by the GPU to render Gaussians.

CUDA. Finally, Gaussians are rendered using the GPU hardware rasterizer. Kerbl *et al*. [1] uses a software renderer in CUDA, breaking cross-platform portability in turn. Our concept does not use CUDA and is therefore less restrictive. Nevertheless, using compute shaders or custom CUDA kernels[1] can noticebly improve the performance.

5. Evaluation

This section evaluates the method and implementation described in this work. In particular, we explore our results regarding visuals, memory use, and performance.

5.1. Details on Prototypical Implementation

To better understand the visualization, we initially present important insight to our Vulkan-based implementation. The interested reader is referred to Haberl [46] for a more in-depth description.

5.1.1. Page determination details

We render the proxy mesh to a visibility buffer using a single-channel 32-bit unsigned integer image. This image can be much smaller than the final rendering while still delivering effective results. Page linking enhances this by connecting nearby pages, minimizing the effects of a smaller rendered image.

The vertex shader receives vertex positions and applies model and camera transformations used later in the scene. The fragment shader gets a buffer of page IDs for each face, with the input variable supplying the face index to output the corresponding page ID. Background and unassigned faces result in a zero page, ignored in later steps.

After rendering the proxy mesh, we reduce the data by creating a buffer indicating the required pages. A compute shader updates each pixel's list entry in parallel, encoding depth as integers that decrease with greater depth. Using the `atomicMax()` function, only the closest depth value is written to the buffer. We iterate through linked pages per pixel to set buffer entries. Once transferred to CPU memory, the array shows zeroes for unneeded and depth values for needed pages. Selection is based on depth value LOD. Transitions between detail levels are adjusted at run-time based on used-to-free memory ratio, with the goal of 50-80% usage. Thresholds are moved if outside this range, increasing step size by 1% after successive similar adjustments or decreasing it by 1% if direction changes quickly.

5.1.2. Transfer

The buffer of Gaussians is memory-mapped on the Central Processing Unit (CPU), enabling buffer access while file reads are managed by the OS. Discrete Graphics Processing Unit (GPU)s' large memory segments are not directly accessible from the CPU. We copy these to host-visible memory for writing, despite its small size and slower GPU access. This staging buffer is vital for discrete GPUs, as they have independent memory separate from the CPU, connected typically via Peripheral Component Interconnect Express (PCIe). Integrated GPUs, used in mobile devices, share memory with the

[1] Note that CUDA is only available on a very low number of consumer mobile devices overall.

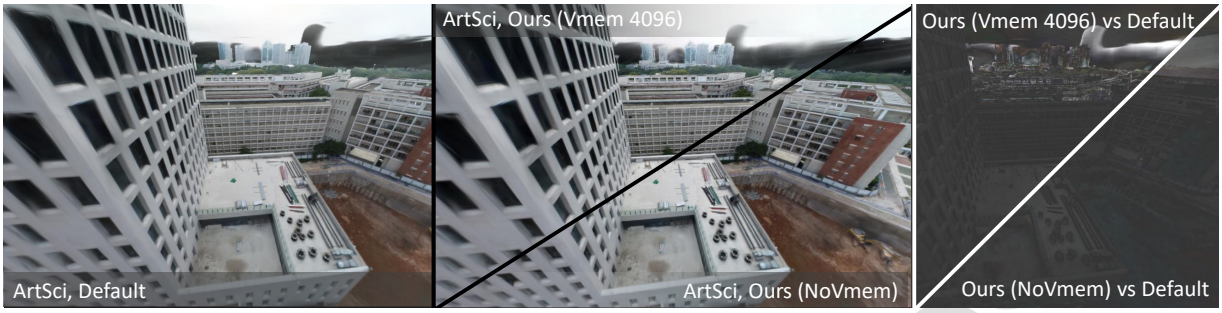


Figure 8: *Baseline comparison:* We compare our method without virtual memory and with virtual memory (4096 Gaussians per page) against the default version, resulting into minimal discrepancies (visually negligible) as show in the diff images on the right. To make small differences visible we feature scaled the difference by 40%.

	Alameda	Berlin	Residence	ArtSci
↑ PSNR[dB] (D *vs* NV)	21.75	48.97	34.18	29.42
↑ SSIM (D *vs* NV)	0.7123	0.9947	0.9864	0.9767
↑ PSNR[dB] (D *vs* 4096)	21.75	49.03	26.97	26.59
↑ SSIM (D *vs* 4096)	0.7124	0.9947	0.9153	0.9623
↑ PSNR[dB] (4096 *vs* NV)	47.93	57.43	13.28	25.91
↑ SSIM (4096 *vs* NV)	0.9989	0.9996	0.5269	0.9722

D .. Default NV .. No Virtual Memory 4096 .. number of Vmem pages

Table 1: Baseline comparison to the algorithm of Kerbl *et al*. [1], who implemented shading in CUDA, while our implementation is based on a Vulkan shader. The Peak Signal-to-Noise Ratio (PSNR) measure is therefore less meaningful, while the Structural Similarity Index Measure (SSIM) shows high values consistently.

CPU, making all accessible memory host-visible but with lower throughput. The implementation completes all memory transfers before further steps, with potential enhancements through separate GPU queues for reduced framerate impact.

5.1.3. Synchronisation

Work is largely performed on the GPU. Steps in Fig. 7 benefit from parallel execution but require synchronization. The page update begins with recording commands to draw the proxy mesh and reduce page IDs, separated by a pipeline barrier to ensure memory is flushed before processing results. These commands are submitted to a graphics and compute-capable queue. We use a fence to wait before updating the page table on the CPU. Data transfer to the staging buffer starts during the page table update, with multi-threaded transfers handled by the drivers. After updating, Gaussians are copied to the second command buffer, also containing depth sorting and drawing, separated by a barrier. Commands are recorded during data transfer in separate threads. Depth sorting via compute shaders requires nine dispatches with metadata in a uniform buffer, defined by a push constant. Lean barriers prevent pipeline stalls. Post-dispatch, depth results move to the index buffer. Rendering the final 3DGS scene uses the updated index buffer. After updating the page table and completing transfers, command buffer submission triggers a semaphore, presenting the image. Two frames can be processed simultaneously, with work on the CPU while the GPU presents the previous frame.

5.1.4. Adaptations for Mobiles

We port our real-time renderer to iOS to test its viability on mobile devices. We rely on MoltenVK [47], which serves as compatibility layer to run Vulkan on the Metal API allowing our existing Vulkan code to run on macOS and iOS. Despite some extension support differences with Windows, the essential extensions required are supported. The main differences between desktop OS (Windows, macOS) and iOS relate to file handling. An iOS app bundle includes an executable and resources, while it can read from and write to an app-specific, sandboxed directory. We package all shaders and test scenes into the app bundle, using the POSIX function `mmap(...)` for memory mapping. Apple devices fully support POSIX, and system interactions are feasible via the C++ standard library. While we adjust Vulkan and file-related code to ensure our implementation compiles and runs, we do not optimize for mobile devices or iOS specifically.

5.1.5. Build Details and Test Hardware

As mentioned, our renderer is implemented in C++ using Vulkan, which uses the hardware rasterizer. The evaluation is done on Windows computer and we compile our application with Microsoft Visual C++ (MSVC). The build type is set to `RelWithDebInfo` to optimize the binary for best performance. We disable any Vulkan validation layers during our evaluation.

Our testing hardware includes an NVIDIA GeForce GTX 1070 GPU and Intel Core i7-4770k CPU, using Double Data Rate 3.0 (DDR3) system memory to transfer Gaussian via PCIe 3.0. Though not state of the art, this setup is expected to remain common among consumers [50]. Thus, good performance on such devices is crucial for 3DGS's adoption. We also test our method on an Apple iPad Pro, model A2377 with an Apple M1 System-On-Chip (SoC), featuring CPU and integrated GPU, to assess performance impacts on mobile devices and guide future development regarding integrated GPUs.

5.2. Qualitative Analysis

For a better understanding of the achieved rendering quality, we perform a baseline evaluation against the default rendering, similar to the original 3DGS [1] implementation. We also test on a set of large scenes with motion paths, and discuss the impact of page linking.

Figure 9: Impact of the spatial resolution of the proxy mesh (low to high from left to right) on the ArtSci scene. The medium proxy mesh (not shown to provide a full view of the scene) is inbetween both extremes in terms of the numbers of faces.

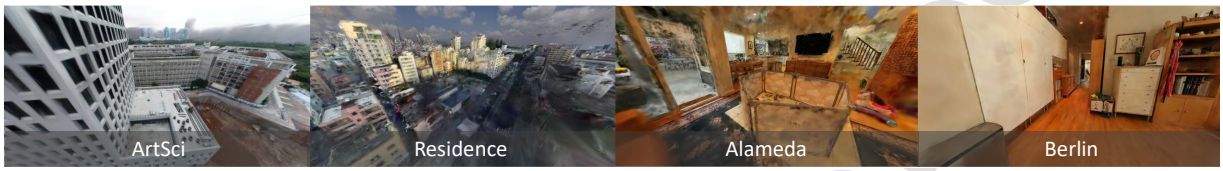


Figure 10: Example images of the evaluated scenes: *ArtSci* and *Residence* are taken from the UrbanScene3D [48] dataset and trained with VastGaussian [19]. *Alameda* and *Berlin* are from Zip-NeRF [49] and trained with Nerfstudio [41].

5.2.1. Baseline Evaluation

In Figure 8, a rendering using the baseline implementation is compared to renderings using our method, without (noVmem) and with virtual memory (Vmem). There is no obvious difference between them; however, once there is not enough virtual memory available, not all visible Gaussians can be loaded at the same time.

To simulate different mesh qualities, our marching cubes implementation was repeatedly run with different spatial X,Y and Z resolutions. In Figure 9 the impact of the spatial resolution of the proxy mesh on the rendering quality is shown. Since all different generated meshes resemble the underlying geometry properly, the visibility of Gaussians is correctly determined, and the differences in visual quality are negligible. A numerical evaluation shown in Table 1 reveals that slight differences in the rendering result in a high PSNR, but that the SSIM remains very high. This provides proof that our baseline behaves similar to the original 3DGS implementation in terms of quality.

5.2.2. Motion through Large Scenes

Large scenes are expected to make extensive use of LOD, while complex scenes with potential for occlusions allow us to eliminate many Gaussians from rendering. We therefore choose a selection of large-scale scenes trained with the methods introduced by Lin *et al*. [19]. Kerbl *et al*. [18] present a similar dataset in their work on large-scale rendering, but it is not available at the time of writing. Smaller scenes with obstructions are created from images by Barron *et al*. [49] and trained using Nerfstudio [41]. The selected scenes are listed in Table 2, with an image of each in Fig. 10. Datasets in novel view synthesis often consist of images of a single central object captured from all angles. They contain negligible amounts of occlusion and, therefore, generally do not benefit significantly from virtual memory. We omit testing for these types of scenes.

We defined a camera path for each using a perspective camera with a 90° Field Of View (FoV). The camera moves at a fixed speed, interpolating its position and orientation between checkpoints. Paths are designed to encompass diverse scenarios, including close-ups and distant overviews, thoroughly navigating the scene to engage multiple Gaussian pages at varying detail levels, thereby showcasing the method's strengths and weaknesses. We track key statistics per frame along the camera path: memory used by active Gaussians, the time to render the proxy mesh, list pages, update the page table, copy data, sort, and render Gaussians.

Scene	Initial Size [MiB]	Pages	Links	File Size [MiB]	Pre processing [s]
ArtSci	804.9	3778	49576	1505.5	2600
Residence	2214.6	4954	65856	4281.5	4579
Alameda	353.9	730	10664	630.9	1169
Berlin	236.5	492	6444	425.2	921

Table 2: Evaluation statistics for scenes with a page size of 2048 include initial sizes post-reconstruction, followed by updates such as padding and added LOD levels, affecting final sizes. Preprocessing times for each scene are also provided.

Table 2 presents statistics per scene with a 2048 page size. Page numbers correlate with Gaussian counts despite padding. Adaptive LOD allows the same buffer for scenes with varying pages, aiming to maintain stable frame times and memory. When pages exceed the buffer, higher LOD levels decrease quality. Alternatively, positioning the camera to capture fewer pages permits rendering at lower LOD levels, improving results. Creating a 3DGS scene involves steps like SfM, training, and preprocessing, which are not deterministic and may lead to different statistics on recreation. We consistently find an average of 13-15 links per page across scenes. Similar to mipmaps, halving the page size with each LOD level nearly doubles the final size. The overhead we introduce is negligible compared to the size of the original scene. We store the proxy mesh with its page IDs, metadata, and page links in a single file. This file stays below 20 MiB in all listed scenes.

For completion, the pre-processing times excluding slice rendering are given in the last column in Table 2. Factors such as

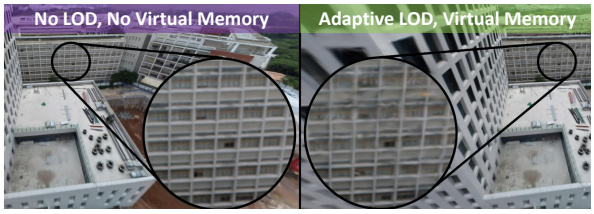


Figure 11: Similar quality, smaller memory footprint: To avoid creating holes due to lack of available memory, the scene is rendered entirely without virtual memory. Virtual memory with adaptive LOD is enabled. Close to the camera, the original Gaussians are rendered.

size, scene structure, and hardware impact these times. The largest scene, residence, requires more than an hour to pre-process, with 85% spent on LOD generation. Its structure makes k-means converge slowly. For most scenes, page assignment, linking, and LOD generation each make up half the pre-processing time. Excluding residence, each page adds about half a second. LOD generation, being highly parallel, can be accelerated by reducing iterations or using a faster CPU with more cores.

5.2.3. Impact of Page Linking and LOD

To illustrate the impact of page linking and LOD, we chose the ArtSci scene. Page assignment of Gaussians is based solely on position, so Gaussians at *e.g.* the building's top, likely facing upward, should still be rendered even if they're not in the visibility buffer. With page links, pages at the front and top are linked due to overlap. Fig. 11 contrasts rendering without virtual memory with using it with LOD. Gaussians further from the camera naturally use higher LOD levels. During pre-processing, Gaussians within a page are combined to halve those of the lower level. At runtime, memory use determines the distance from the camera where LOD levels change. Higher levels are applied further away, slightly lowering visual quality, but clustering such pages in a render buffer boosts performance.

To determine the impact of individual contributions proposed, we enable the methods separately (linking, LOD, etc.). In Table 3 we calculate PSNR and SSIM between an image rendered with and without virtual memory, resembling various ablations of our method. Fig. 12 contains all image variations and their differences from the scene rendered without virtual memory. None of the configurations are memory-constrained; all required pages fit in memory.

When LOD is disabled but page links are used, almost all of the 500 available pages are used. With buffer usage above 80%, adaptive LOD reduces the distances for level transitions. As a result, image quality suffers, with a sharp drop in PSNR. The ranges LOD adapts to satisfy conditions on memory use are not deterministic. When page links are not used, noticable artifacts appear in the image, which is reflected in the image comparison scores. These scores listed in Table 3, as well as the right image in Fig. 12, indicate the best quality with page links enabled and LOD disabled. This makes sense since page links increase the number of required pages to produce better results while LOD decreases quality to regain memory. Disabling page linking (Fig. 12 (b) and (d)) leads to the expected artifacts.

ArtSci	PSNR [dB]	SSIM
With page links, with LOD	42.89	0.99
Without page links, with LOD	35.02	0.97
With page links, without LOD	47.19	0.99
Without page links, without LOD	35.02	0.97

Table 3: Comparison between images rendered without virtual memory and with ablations of our virtual memory solution.

Page Size	Page Table Update [ms]	Depth Sort [ms]	Render [ms]
2048	0.9	2.0	8.0
4096	0.5	3.0	9.7
8192	0.3	6.7	10.6

Table 4: Comparison of median time taken for selected rendering stages with different page sizes. In all tests, the camera follows the same predefined path. As the page size increases, page table updates shorten, while depth sorting and rendering take longer.

Enabling both methods can be thought of as balancing quality and memory usage. Page links fulfill their part of contributing pages not visible on the proxy mesh but overlapping with pages that are. Moreover, when those pages are distant, a different LOD level is used, slightly reducing the quality.

Limitations. There are mainly two adversary effects of our current page link method. First, the reduction in the number of page overlaps is not guarded during pre-processing. This might cause the occlusion culling effect achieved by the visibility buffer during rendering to be completely negated, leading to increased memory overhead (Figure 13 (b)). As LOD might mitigate overhead, higher LOD levels may lower quality. As LOD reduces Gaussian page quality, artifacts appear when merging over distances. Although minor when far from the camera, these artifacts become significant near the camera at high LOD levels (right side of Figure 13 (c)). Reducing page overlaps and adjusting Gaussian scales can decrease artifacts.

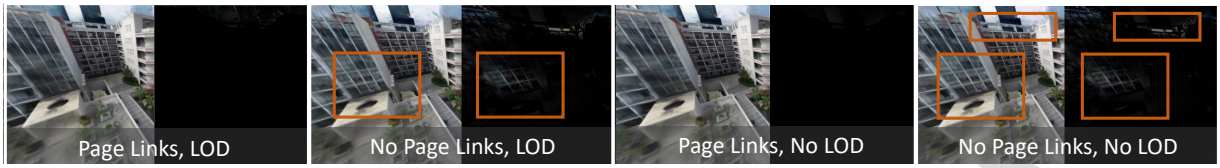
5.3. Quantitive Analysis

In order to closer assess our approach, several factors were investigated in more detail.

5.3.1. Cache Locality

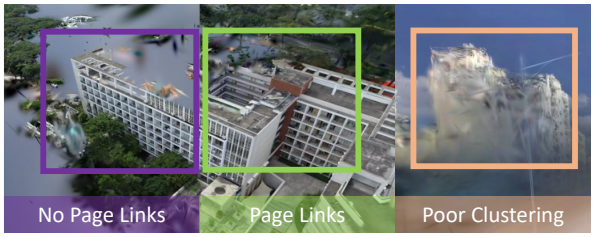
Grouping Gaussians together enhances cache locality, increasing cache hits during rendering, especially with virtual memory. Although page order does not reflect Gaussian proximity, streaming visible pages to a smaller buffer brings pages closer. Larger page sizes raise position variance and cause more Gaussians to be falsely marked visible due to a less atomic visibility buffer, while smaller sizes add overhead to page table management and memory transfers.

Table 4 compares timings of selected rendering stages for different page sizes. Other stages are largely unaffected. Notably, the time gained updating the page table at larger page sizes does not seem to outweigh the penalty of unnecessary Gaussians and reduced cache locality, at least for large scenes. To reduce the impact of cache locality, future implementations may additionally choose to reorder Gaussians within each page based on the Morton order. Table 4 indicates, that a 2048 page size is appropriate to balance cache locality with page table management



(a) (b) (c) (d)

Figure 12: Images with the same camera position and settings, rendered with ablations of our method (left). Images on the right compare these to the same scene rendered virtual memory. Black indicates no difference, pixel brightness indicates how large the difference is.



(a) (b) (c)

Figure 13: No Page Links leading only the pages of Gaussians determined to be visible based on the visibility buffer are transferred and rendered. Enabled page links, causing significantly more pages to be rendered based on page overlaps. Poor clustering of Gaussians to merge can create artifacts, which becomes obvious when high LOD levels are used close to the camera.

Buffer Size [Pages]	Without LOD		With LOD	
	Render [ms]	Missing Pages	Render [ms]	Missing Pages
250	3.8	549	5.0	0
500	6.6	263	6.7	0
1000	8.1	0	8.4	0

Table 5: Comparison of median render times and number of missing pages for different buffer sizes with and without LOD. Buffer sizes are specified as a multiple of the page size. Both with and without LOD, the render time increases with the buffer size. However, LOD reduces the necessary memory to contain the required pages.

overhead. The size of the staging and rendering buffers is crucial. A larger staging buffer permits longer delays, as copying Gaussians blocks depth-sorting and rendering. Conversely, if the buffer is too small for all required pages yet fails to present pages in GPU, artifacts like holes and popping occur. Our staging buffer is set to a size of ∼18.4MiB, equal to 40 pages of 2048 Gaussians.

The rendering buffer size significantly affects performance. The buffer must be large enough to hold visible pages. LOD addresses performance issues from rendering numerous Gaussians: **Our implementation adjusts LOD levels based on memory usage. In other words, it aims to fit all required pages into the buffer to avoid holes.** As a side effect, large buffers allow many Gaussians to be rendered at low LOD levels (more details), negatively impacting performance. Table 5 examines the link between buffer size, render times, and missing pages. Without LOD, smaller buffer sizes are too small to hold all visible pages. As a result, render times increase when increasing the buffer size. When using adaptive LOD, distances are chosen to avoid any holes due to missing pages. **As a result, a smaller buffer can render the same scene without artifacts.**

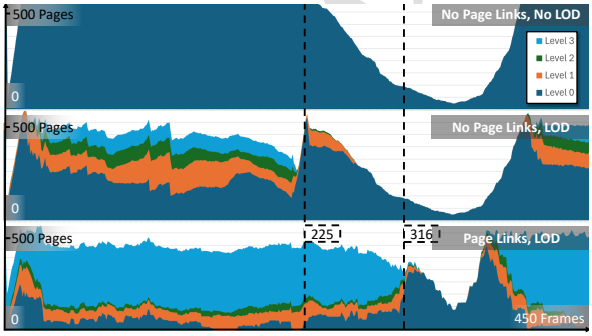


Figure 14: Memory usage during a fly-through in the scene ”Residence” with different ablations. This scene contains almost 5000 pages of Gaussians with the first LOD level taking up more than 2 GiB. With a 500 page buffer only ∼ 10% of Gaussians can be rendered at a time.

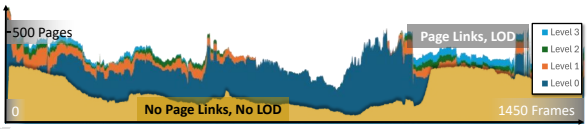


Figure 15: Memory usage during a fly-through in the scene ”Alameda” with different ablations. This indoor scene contains many occlusions but only 730 pages in all.

5.3.2. *Memory Usage*

We assess the memory usage for two scenes in more detail. The camera follows a path through each, with different ablations. As the camera moves, the number of rendered pages changes. In Residence (Fig. 14), a large open scene, memory is quickly saturated without LOD enabled. In contrast, Alameda (Fig. 15), which features many occlusions but is limited in size, memory use never exceeds 50%.

Upon enabling LOD, memory use in Residence is reduced drastically. A single physical page can store up to 2, 4, or 8 pages, respectively, at higher levels. Adaptive LOD attempts to keep memory use between 50% and 80% and modifies the distances at which levels transition to achieve this. Alameda, which uses limited memory, is not affected by enabling LOD. All required pages fit within the 80% limit in memory at all times, resulting in the use of level zero throughout.

When page links are used, memory usage is increased drastically. A large number of physical pages are populated with level three pages in Fig. 14. This indicates adaptive LOD needs to set the distances for level transitions close to the camera in

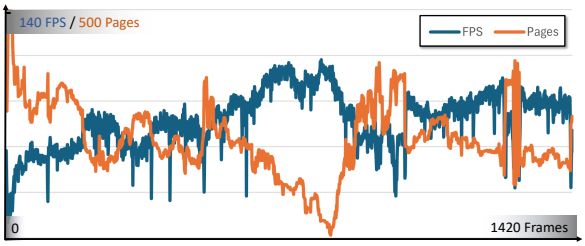


Figure 16: FPS observed along a fixed camera path, compared to the number of rendered pages. FPS are not smoothed; they are determined by the time between subsequent frames. Performance clearly correlates with the number of Gaussians rendered. Frame times decrease as the number of pages decreases.

GPU	FP32 [TFLOPS]	Bandwidth [GB/s]
NVIDIA GTX 1070	6.5 [51]	256.3 [51]
Apple M1 SoC	2.6 [52]	68.3 [53]

Table 6: Basic comparison of the iPad Pro's integrated M1 GPU to the dedicated GPU used for evaluation on desktop. FP32 measures the number of 32-bit floating point operations per second (in trillions).

order to keep memory use within the targeted range. A similar increase in rendered pages can be observed in Fig. 15, where multiple levels of detail are now required.

Page linking can indeed degrade image quality by increasing memory needs, leading to poorer rendering LOD. However, without page links scenes may show artifacts, as seen in Fig. 13. As a tradeoff, careful page assignments and setting a threshold to ignore small overlaps can minimize page links without causing significant artifacts. Overall, using virtual memory significantly reduces memory usage and the count of Gaussians rendered. Without virtual memory, full scenes equate to 4417 and 724 pages. In the "Alameda" scene, visible Gaussians are at most 35% of the total due to occlusions.

5.3.3. *Performance*

Performance, measured in FPS, is closely correlated with the number of Gaussians rendered (Fig. 16). As the number of pages increases, the FPS drop. This is indicative of the effectiveness of reducing the number of rendered Gaussians to improve performance. In the following, we discuss selected frames from the Berlin scene with a certain impact:

- **Most Pages.** One of the frames with the highest number of physical pages in active use.
- **Median.** The median time taken to complete each step over the entire path. This is not a real frame but rather a reconstruction meant to approximate the median frame.
- **Shortest.** The frame with the smallest sum of the time taken by each step.
- **Largest Transfer.** The frame with the most bytes of Gaussian data copied to GPU memory.

Rendering the visibility buffer and reducing it to a list of page IDs takes an almost constant amount of time. Page table updates are nearly constant, with a slight delay when adding many new pages. Most frames have no waiting overhead for the staging buffer copy. For the frame with the largest transfer, this is a minor issue. Sorting by depth and rendering time scale with

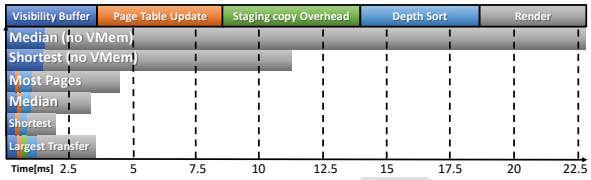


Figure 17: The time taken for various steps with virtual memory as well as the median and shortest frame rendered without virtual memory. The overhead associated with virtual memory is negligible when compared to the time gain for depth sort and render stages.

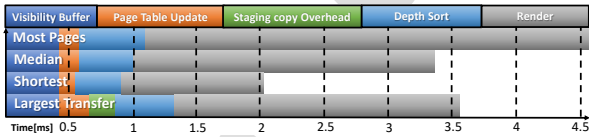


Figure 18: Comparison of the time taken for various steps when rendering a frame with virtual memory. Specific frames are selected for further analysis. The median frame is not an actual frame but a combination of the median time taken by each step.

the number of Gaussians, with rendering time being crucial to minimize.

The effectiveness of virtual memory can be seen when rendering the scene without virtual memory and comparing the timings (Fig. 17). The graph contains the timings shown in Fig. 18 but adds additional data without the use of virtual memory. The rendered scene contains the same Gaussians as the base level of Gaussians with virtual memory. Gaussians are not grouped into pages, rearranged, or padded. Therefore, it does not benefit from visibility determination or improved memory locality.

Rendering the Berlin scene uses 489 pages in virtual memory, comfortably fitting in a 500-page buffer. Excluding hidden pages significantly improves frame times. Frames without virtual memory take longer than previously analyzed ones, with the median frame taking even more time. Without virtual memory, page management overhead is eliminated. Timing shows the benefits of these preparatory steps, as even the depth sort stage is faster. Rendering speeds up significantly due to improved memory locality and fewer Gaussians to process.

5.3.4. *Mobile*

Our concept renders discrete and integrated GPUs identically under the same configuration and swapchain image size. However, based on Table 6, performance varies across devices. The iPad's M1 is a SoC with CPU and GPU on a single chip, sharing the same memory. Vulkan reports only a single memory heap where memory may be accessed by both the device and the host. Our application does not make use of this, instead keeping the same data in memory multiple times. In an optimization file, backed memory can be used by the GPU directly and no manual streaming is necessary. After measuring timings, we can therefore make some limited assumptions about performance without streaming.

With the same configuration and camera paths, only performance varies between desktop and mobile. Initially, comparing baseline performance without virtual memory, Fig. 19 shows

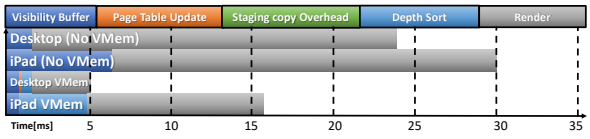


Figure 19: Comparison of the time taken to render frames on desktop and mobile. All frames are a collection of the median times for each stage along a path. The first two bars are a baseline, without virtual memory enabled.

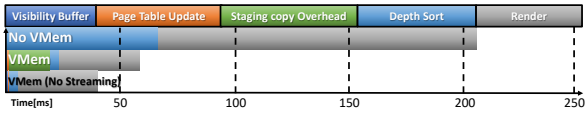


Figure 20: Time taken to render a frame with the median of each stage on mobile. The last bar displays a fictional frame that is the same as the one above it but with the streaming overhead removed.

frames take longer on mobile devices due to a fourfold increase in Gaussian sorting time, indicating slower memory access as expected with an integrated GPU. Enabling virtual memory reduces the number of Gaussians that need to be sorted. The overhead of this method is justified, as depth sorting and rendering are significantly faster.

Fig. 20 shows the measured mobile times for a larger scene with frame times greater than 200 ms without virtual memory. Our method reduces time by culling Gaussians, though streaming overhead remains due to slow memory access and lack of optimization for integrated GPUs. We simulate median times without streaming overhead, potentially possible with file-backed memory. However, even then, it is not interactive. Large scenes need further optimization, like reducing buffer sizes or software rendering.

In general, mobile performance is mainly affected by memory accesses. This is especially noticeable during our global depth sort. Our virtual memory method, apart from the unoptimized streaming, is not greatly impacted by the move to mobile.

5.4. Discussion

Our initial assumption is that the number of rendered Gaussians directly affects performance. For an approach to be effective, an induced reduction in render time must exceed the overhead of identifying visible Gaussians. This is clearly confirmed by our results. We expect this to also hold with different rendering approaches, including the software renderer from Kerbl *et al*. [1].

Our proxy mesh strategy effectively reduces memory in large scenes and relies on acceleration structures for fast Gaussian determination, as mentioned in Section 2. However, unlike other methods, it handles occlusions well, boosting performance in occluded environments such as indoor and city street-level datasets. Our approach also allows virtually unlimited scene size during rendering at a reasonable scale and speed.

Published datasets and implementations for large-scale scenes are inadequate. Common 3DGS training implementations poorly handle occlusions, creating artifacts. Datasets often originate from Unmanned Aerial Vehicle (UAV)s with few occlusions. The dataset from Kerbl *et al*. [18] is promising but uses a non-standard file format.

Gaussians can't map directly to surface textures, so we use page links, which help avoid rendering artifacts but increase memory usage. We plan to enhance link creation to use fewer pages. LOD integrates with our solution like mipmapping with virtual texturing, crucial for maintaining a strict memory budget without significant quality loss. Our LOD system has flaws in creation (incorrect scale) and rendering (lack of level blending). Integrating recent works from Section 2 can address these issues.

6. Conclusion and Future Work

In this work, we demonstrate the viability of applying the concepts used in virtual texturing to cutting-edge research in 3DGS. A proxy mesh is generated from an existing 3DGS scene, grouping Gaussians into pages with IDs and linking them if they overlap, followed by forming several LOD levels. Rendering this mesh to a visibility buffer enables fast visibility checks. Pages are managed in GPU memory, transferring visible ones to the GPU just in time, replacing unnecessary ones. LOD levels are selected based on the camera distance and memory. The effectiveness of our approach is evaluated and demonstrated on several large scenes on both desktop and mobile hardware.

In contrast to related works, we avoid compression in order to test the proposed methods in isolation. Reconstruction of large and huge scenes is still an active research topic, and proper tools for this are not readily available yet. Not compressing Gaussians allows us to test our method at its limits to effectively find shortcomings. We demonstrate the use of virtual texturing in this field, effectively reducing memory and enhancing performance.

Future work will address several shortcomings and potential optimizations mentioned throughout this work. Both the page linking and the LOD solution can greatly benefit from several improvements also denoted in Section 2, as recent research in LOD for 3DGS has gained interest. Currently, our method avoids scene reconstruction changes, starting with a complete scene for pre-processing. Future work could integrate virtual memory into the training phase, regularly pre-processing and rendering with visibility determination.

References

[1] B. Kerbl, G. Kopanas, T. Leimkuehler, G. Drettakis, 3D Gaussian Splatting for Real-Time Radiance Field Rendering, ACM Transactions on Graphics 42 (4) (2023) 1–14. doi:10.1145/3592433.

[2] P. J. Denning, Virtual memory, ACM Computing Surveys (CSUR) 2 (3) (1970) 153–189.

[3] W. E. Lorensen, H. E. Cline, Marching cubes: A high resolution 3d surface construction algorithm, in: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87, Association for Computing Machinery, New York, NY, USA, 1987, p. 163–169. doi:10.1145/37401.37422.
URL https://doi.org/10.1145/37401.37422

[4] H. Chen, C. Li, G. H. Lee, Neusg: Neural implicit surface reconstruction with 3d gaussian splatting guidance (arXiv:2312.00846), arXiv:2312.00846 [cs] (Dec. 2023).
URL http://arxiv.org/abs/2312.00846

[5] A. Guédon, V. Lepetit, Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering (arXiv:2311.12775), arXiv:2311.12775 [cs] (Dec. 2023).
URL http://arxiv.org/abs/2311.12775

[6] J. Waczyńska, P. Borycki, S. Tadeja, J. Tabor, P. Spurek, Games: Mesh-based adapting and modification of gaussian splatting (arXiv:2402.01459), arXiv:2402.01459 [cs] (Feb. 2024).
URL http://arxiv.org/abs/2402.01459

[7] Y. Yuan, X. Li, Y. Huang, S. De Mello, K. Nagano, J. Kautz, U. Iqbal, Gavatar: Animatable 3d gaussian avatars with implicit mesh learning (arXiv:2312.11461), arXiv:2312.11461 [cs] (Dec. 2023).
URL http://arxiv.org/abs/2312.11461

[8] L. Gao, J. Yang, B.-T. Zhang, J.-M. Sun, Y.-J. Yuan, H. Fu, Y.-K. Lai, Mesh-based gaussian splatting for real-time large-scale deformation (arXiv:2402.04796), arXiv:2402.04796 [cs] (Feb. 2024).
URL http://arxiv.org/abs/2402.04796

[9] A. Pranckevičius, Making gaussian splats smaller, https://aras-p.info/blog/2023/09/13/Making-Gaussian-Splats-smaller/, accessed: 2024-06-04 (Sep 2023).

[10] W. Morgenstern, F. Barthel, A. Hilsmann, P. Eisert, Compact 3d scene representation via self-organizing gaussian grids (arXiv:2312.13299), arXiv:2312.13299 [cs] (Dec. 2023).
URL http://arxiv.org/abs/2312.13299

[11] A. Pranckevičius, Making gaussian splats more smaller, https://aras-p.info/blog/2023/09/27/Making-Gaussian-Splats-more-smaller/, accessed: 2024-04-06 (Sep 2023).

[12] Z. Fan, K. Wang, K. Wen, Z. Zhu, D. Xu, Z. Wang, Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps (arXiv:2311.17245), arXiv:2311.17245 [cs] (Feb. 2024).
URL http://arxiv.org/abs/2311.17245

[13] S. Niedermayr, J. Stumpfegger, R. Westermann, Compressed 3d gaussian splatting for accelerated novel view synthesis (arXiv:2401.02436), arXiv:2401.02436 [cs] (Jan. 2024).
URL http://arxiv.org/abs/2401.02436

[14] K. Navaneet, K. Pourahmadi Meibodi, S. Koohpayegani, H. Pirsiavash, Compact3D: Compressing Gaussian Splat Radiance Field Models with Vector Quantization, 2024.

[15] S. Girish, K. Gupta, A. Shrivastava, Eagles: Efficient accelerated 3d gaussians with lightweight encodings (arXiv:2312.04564), arXiv:2312.04564 [cs] (Dec. 2023).
URL http://arxiv.org/abs/2312.04564

[16] Z. Li, Z. Chen, Z. Li, Y. Xu, Spacetime gaussian feature splatting for real-time dynamic view synthesis (arXiv:2312.16812), arXiv:2312.16812 [cs] (Dec. 2023).
URL http://arxiv.org/abs/2312.16812

[17] J. C. Lee, D. Rho, X. Sun, J. H. Ko, E. Park, Compact 3d gaussian representation for radiance field (arXiv:2311.13681), arXiv:2311.13681 [cs] (Nov. 2023).
URL http://arxiv.org/abs/2311.13681

[18] B. Kerbl, A. Meuleman, G. Kopanas, M. Wimmer, A. Lanvin, G. Drettakis, A hierarchical 3d gaussian representation for real-time rendering of very large datasets, ACM Transactions on Graphics 43 (4) (2024) 1–15. doi:10.1145/3658160.

[19] J. Lin, Z. Li, X. Tang, J. Liu, S. Liu, J. Liu, Y. Lu, X. Wu, S. Xu, Y. Yan, W. Yang, Vastgaussian: Vast 3d gaussians for large scene reconstruction (arXiv:2402.17427), arXiv:2402.17427 [cs] (Feb. 2024). doi:10.48550/arXiv.2402.17427.
URL http://arxiv.org/abs/2402.17427

[20] Y. Liu, H. Guan, C. Luo, L. Fan, N. Wang, J. Peng, Z. Zhang, Citygaussian: Real-time high-quality large-scale scene rendering with gaussians (arXiv:2404.01133), arXiv:2404.01133 [cs] (Jul. 2024). doi:10.48550/arXiv.2404.01133.
URL http://arxiv.org/abs/2404.01133

[21] H. Zhao, H. Weng, D. Lu, A. Li, J. Li, A. Panda, S. Xie, On scaling up 3d gaussian splatting training (arXiv:2406.18533), arXiv:2406.18533 [cs] (Jun. 2024).
URL http://arxiv.org/abs/2406.18533

[22] A. Meuleman, I. Shah, A. Lanvin, B. Kerbl, G. Drettakis, On-the-fly reconstruction for large-scale novel view synthesis from unposed images, ACM Trans. Graph. 44 (4) (Jul. 2025). doi:10.1145/3730913.
URL https://doi.org/10.1145/3730913

[23] P. Jiang, G. Pandey, S. Saripalli, 3dgs-reloc: 3d gaussian splatting for map representation and visual relocalization (arXiv:2403.11367), arXiv:2403.11367 [cs] (Mar. 2024).
URL http://arxiv.org/abs/2403.11367

[24] Z. Yan, W. F. Low, Y. Chen, G. H. Lee, Multi-scale 3d gaussian splatting for anti-aliased rendering (arXiv:2311.17089), arXiv:2311.17089 [cs] (Nov. 2023).
URL http://arxiv.org/abs/2311.17089

[25] T. Lu, M. Yu, L. Xu, Y. Xiangli, L. Wang, D. Lin, B. Dai, Scaffold-gs: Structured 3d gaussians for view-adaptive rendering (arXiv:2312.00109), arXiv:2312.00109 [cs] (Nov. 2023).
URL http://arxiv.org/abs/2312.00109

[26] K. Ren, L. Jiang, T. Lu, M. Yu, L. Xu, Z. Ni, B. Dai, Octree-gs: Towards consistent real-time rendering with lod-structured 3d gaussians (arXiv:2403.17898), arXiv:2403.17898 [cs] (Mar. 2024).
URL http://arxiv.org/abs/2403.17898

[27] X. E. Wang, Z. P. T. Sin, 3DGM: Deformable and Texturable 3D Gaussian Model via Level-of-Detail Proxy, Computer Graphics Forum (2025). doi:10.1111/cgf.70223.

[28] W. Lin, Y. Feng, Y. Zhu, RTGS: Enabling real-time gaussian splatting on mobile devices using efficiency-guided pruning and foveated rendering, arXiv e-prints (2024) arXiv–2407.

[29] P. Papantonakis, G. Kopanas, B. Kerbl, A. Lanvin, G. Drettakis, Reducing the memory footprint of 3d gaussian splatting, Proceedings of the ACM on Computer Graphics and Interactive Techniques 7 (1) (2024) 1–17.

[30] S. Barrett, Sparse virtual textures, https://silverspaceship.com/src/svt/, accessed: 2024-06-04 (2008).

[31] A. J. Mayer, Virtual texturing, Master's Thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria (Oct. 2010).
URL https://www.cg.tuwien.ac.at/research/publications/2010/Mayer-2010-VT/

[32] Z. Yu, T. Sattler, A. Geiger, Gaussian opacity fields: Efficient adaptive surface reconstruction in unbounded scenes, ACM Transactions on Graphics (2024).

[33] L. Radl, F. Windisch, T. Deixelberger, J. Hladky, M. Steiner, D. Schmalstieg, M. Steinberger, Sof: Sorted opacity fields for fast unbounded surface reconstruction, arXiv preprint arXiv:2506.19139 (2025).

[34] A. Guédon, V. Lepetit, Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering, CVPR (2024).

[35] J. Held, R. Vandeghen, A. Deliege, A. Hamdi, S. Giancola, A. Cioppa, A. Vedaldi, B. Ghanem, A. Tagliasacchi, M. Van Droogenbroeck, Triangle splatting for real-time radiance field rendering, arXiv preprint arXiv:2505.19175 (2025).

[36] J. Held, R. Vandeghen, S. Son, D. Rebain, M. Gadelha, Y. Zhou, M. C. Lin, M. Van Droogenbroeck, A. Tagliasacchi, Triangle splatting+: Differentiable rendering with opaque triangles, arXiv preprint arXiv:2509.25122 (2025).

[37] C. Griwodz, S. Gasparini, L. Calvet, P. Gurdjos, F. Castan, B. Maujean, G. D. Lillo, Y. Lanthony, Alicevision Meshroom: An open-source 3D reconstruction pipeline, in: Proceedings of the 12th ACM Multimedia Systems Conference - MMSys '21, ACM Press, 2021. doi:10.1145/3458305.3478443.

[38] W. E. Lorensen, H. E. Cline, Marching cubes: A high resolution 3d surface construction algorithm, in: Seminal graphics: pioneering efforts that shaped the field, 1998, pp. 347–353.

[39] E. Hartmann, Computerunterstützte darstellende und konstruktive geometrie, https://www2.mathematik.tu-darmstadt.de/~ehartmann/cdg-skript-1998.pdf, accessed: 2024-09-09 (1997).

[40] M. Mittring, C. GmbH, Advanced virtual texture topics, in: ACM SIGGRAPH 2008 Games, SIGGRAPH '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 23–51. doi:10.1145/1404435.1404438.
URL https://doi.org/10.1145/1404435.1404438

[41] M. Tancik, E. Weber, E. Ng, R. Li, B. Yi, J. Kerr, T. Wang, A. Kristoffersen, J. Austin, K. Salahi, A. Ahuja, D. McAllister, A. Kanazawa, Nerfstudio: A modular framework for neural radiance field development, in: ACM SIGGRAPH 2023 Conference Proceedings, SIGGRAPH '23, 2023.

[42] S. Willems, Hardware capability database for vulkan, http://vulkan.gpuinfo.org, accessed: 2024-09-24 (2016).

[43] The Brenwill Workshop Ltd., Moltenvk, https://github.com/KhronosGroup/MoltenVK, accessed: 2024-09-24 (2015).

[44] A. Richermoz, Sparse texture binding is painfully slow, https://forums.developer.nvidia.com/t/sparse-texture-binding-is-painfully-slow, accessed: 2024-09-01 (2023).

[45] J. Hable, Variable rate shading with visibility buffer rendering, https://advances.realtimerendering.com/s2024/index.html#hable, accessed: 2024-08-07 (2024).

[46] J. Haberl, Virtual memory for 3d gaussian splatting, Master's Thesis, Institute of Visual Computing, Graz University of Technology, Inffeldgasse 16/2, A-8010 Graz, Austria (Dec. 2024). doi:10.3217/xstz7-q3q02.

[47] T. K. Group, Khronos vulkan portability initiative, https://www.khronos.org/vulkan/portability-initiative, accessed: 2024-10-15 (2024).

[48] L. Lin, Y. Liu, Y. Hu, X. Yan, K. Xie, H. Huang, Capturing, reconstructing, and simulating: the urbanscene3d dataset (arXiv:2107.04286), arXiv:2107.04286 [cs] (Jul. 2022).
URL http://arxiv.org/abs/2107.04286

[49] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, P. Hedman, Zip-nerf: Anti-aliased grid-based neural radiance fields, ICCV (2023).

[50] N. Tatarchuk, Welcome and introduction – trends in games and rendering, https://advances.realtimerendering.com/s2024/index.html#intro, accessed: 2024-08-20 (2024).

[51] techpowerup.com, Nvidia geforce gtx 1070, https://www.techpowerup.com/gpu-specs/geforce-gtx-1070.c2840, accessed: 2024-10-18 (2024).

[52] NanoReview.net, Apple ipad pro 11" (3rd gen, 2021), https://nanoreview.net/en/tablet/apple-ipad-pro-11-3rd-gen-2021?m=r.1, accessed: 2024-10-18 (2021).

[53] A. Frumusanu, The 2020 mac mini unleashed: Putting apple silicon m1 to the test, https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested, accessed: 2024-10-18 (2020).

Virtual Memory for 3D Gaussian Splatting

Jonathan Haberl,
jonathan.haberl@alumni.tugraz.at,
Graz University of Technology,
Inffeldgasse 16/2,
Graz, 8010, Styria,
Austria

Philipp Fleck,
philipp.fleck@tugraz.at
Graz University of Technology,
Inffeldgasse 16/2,
Graz, 8010, Styria,
Austria

Clemens Arth
clemens.arth@tugraz.at
Graz University of Technology,
Inffeldgasse 16/2,
Graz, 8010, Styria,
Austria

Declaration of interests

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Clemens Arth reports financial support was provided by European Union. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.