DOI: 10.1111/cgf.70214
High-Performance Graphics 2025
A. Knoll and C. Peters
(Guest Editors)

COMPUTER GRAPHICS *forum*
*Volume 44 (2025), Number 8*

# Splatshop: Efficiently Editing Large Gaussian Splat Models

Markus Schütz[1], Christoph Peters[2], Florian Hahlbohm[3], Elmar Eisemann[2], Marcus Magnor[3], Michael Wimmer[1]

[1] TU Wien, [2] TU Delft, [3] TU Braunschweig

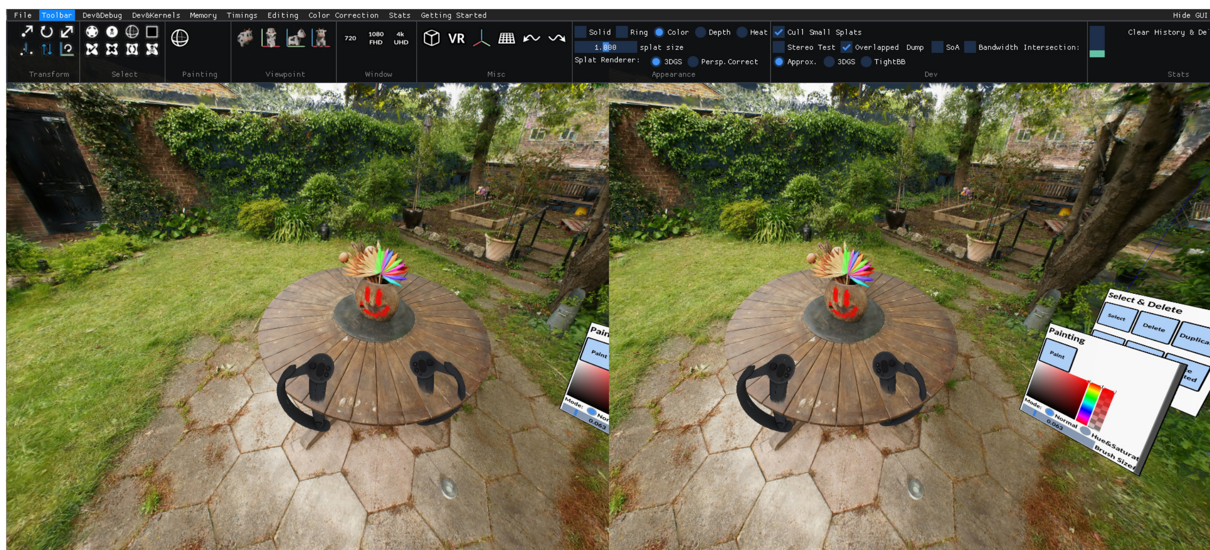



**Figure 1:** *Our editor supports selection, deletion, painting, duplication, and asset libraries for 100M splats (10M at 90 Hz in VR).*

**Abstract**

*We present Splatshop, a highly optimized toolbox for interactive editing (selection, deletion, painting, transformation, ...) of 3D Gaussian Splatting models. Utilizing a comprehensive collection of heuristic approaches, we carefully balance between exact and fast rendering to enable precise editing without sacrificing real-time performance. Our experiments confirm that Splatshop achieves these goals for scenes with up to 100 million primitives. We also show how our proposed pipeline can be extended for use with head-mounted displays. As such, Splatshop is the first VR-capable editor for large-scale 3D Gaussian Splatting models and a step towards a "Photoshop for Gaussian Splatting."*

## 1. Introduction

3D Gaussian Splatting (3DGS) [KKLD23] is a recent breakthrough in novel view synthesis that has rapidly gained traction due to its impressive visual quality and rendering performance. Given a set of input photographs, 3DGS automatically generates a 3D model by fitting and optimizing a collection of colored point primitives, each with varying opacity and 3D Gaussian extent. This approach has proven particularly effective in representing fuzzy or volumetric content, such as hair, fur, or vegetation, where traditional triangle-based reconstruction methods often fall short. Moreover, thanks to the integration of spherical harmonics into the model, Gaussian splats can capture view-dependent material properties like glossiness.

Compared to neural radiance fields (NeRFs) [MST*21], Gaussian splats typically achieve significantly faster rendering times. While NeRFs rely on extensive neural computations along viewing rays through a volumetric scene, 3DGS strategically places anisotropic 3D Gaussians in space and renders them using screen-aligned 2D billboards. Consequently, 3DGS is also suitable for older 3D graphics APIs. Another advantage over NeRFs is that the 3D Gaussians constitute tangible geometric primitives, which makes them intu-

© 2025 The Author(s). Computer Graphics Forum published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org
diglib.eg.org

itive to analyze and easier to edit compared to neural volumetric representations.

At the same time, consumer devices – especially smartphones – now offer high-resolution cameras, making it easier than ever for casual users to capture scenes and generate 3D reconstructions. As with digital photography, where tools like *Adobe Photoshop* have long enabled post-processing before images are shared on social media or stored in personal archives, there is a growing demand for similar workflows for 3D content. Early efforts, such as Pointshop [ZPKG02] for point-based rendering, explored this idea, but were limited by the complexity of creating high-quality models as well as the absence of widely adopted GPGPU frameworks.

With 3DGS, it is now possible to generate high-quality 3D models given a few smartphone images. In the future, such models could be used to enable immersive experiences in virtual reality (VR) or integrated as assets for games and interactive media. However, to unlock the full potential of these representations, accessible and efficient editing tools are needed for both professional users and casual creators. Tools like SuperSplat [Ltd25] have demonstrated the practicality of 3DGS editing, even within the constraints of web-based graphics APIs.

In this work, we present Splatshop, a post-processing tool for 3DGS-based scene representations that enables both essential tasks like splat cleaning as well as optional enhancements such as painting. Our tool is designed with a strong focus on performance, usability, and support for large-scale models with tens of millions of splats. Furthermore, we introduce functionality for direct editing in VR, enabling an immersive and intuitive 3D content creation workflow. Specifically, our contributions are:

- An efficient and modern CUDA-based splat editing system.
- A fast splat sorting algorithm that splits the 48-bit radix sort of $m$ splat fragments into a 32-bit radix sort of $n$ splats and a 16-bit radix sort of $m$ fragments,
- A staged-fragment (per-touched-tile duplicated splats) reduction through tighter AABBs and an approximate tile-ellipse intersection test (concurrent with Speedy-Splat [HTL*25]),
- Undo/redo functionalities including a discussion for efficiently addressing some tasks and workarounds for open problems,
- A study of workload imbalances of tiles with large numbers of splats, including VR improvements.

## 2. Related Work

In 2023, 3DGS became a popular geometric primitive for 3D reconstruction from a set of images [KKLD23]. They offer similar quality as neural radiance fields (NeRFs) [MST*21] while greatly improving rendering times. In addition, splats are geometric primitives that can be reasoned about and freely edited [Ltd25], while NeRFs are often considered to be opaque neural representations with limited editability.

### 2.1. Rendering

Based on elliptic Gaussian kernels [ZPvBG01], 3DGS renders 3-dimensional Gaussians by first projecting them to two-dimensional, screen-facing Gaussians, and then blending them in a front-to-back order. For large Gaussians, this may lead to popping artifacts as their order suddenly changes during camera rotations. StopThePop [RSP*24] addresses this shortcoming by essentially bending the splats in a way that makes the order in which they are blended consistent under camera rotation while also introducing tile-based culling to prevent unnecessary fragment creations during rasterization. EVER [MHK*24] switches from rasterization to ray tracing constant-density ellipsoids, which features more accurate blending, consistent shapes under rotation, and also eliminates popping. StochasticSplats [KVK*25] uses Monte Carlo estimators that maintain higher frame rates than exact ray tracing. It offers the option to trade off quality and performance via a choice of samples per pixel and enables volumetric intermixing. Hahlbohm et al. propose efficient perspective-correct Gaussian rasterization that is view-consistent [HFW*25] and incorporates hybrid transparency solutions to mitigate popping [MCTB13; Wym16]. Speedy-Splat [HTL*25] introduces optimizations to the 3DGS splat rendering pipeline, reducing the number of tiles that are processed for each splat to exactly those that intersect its screen-space ellipse. To reconstruct city-scale scenes with tens of millions of Gaussians, Kerbl et al. introduced a hierarchical representation enabling efficient LOD selection and interpolation [KMK*24]. Long before 3DGS emerged as a means of novel-view synthesis, Weyrich et al. already developed a hardware system that implements EWA-splatting in FPGA and ASICs [WHA*07].

Recent VR-focused approaches propose foveated rendering techniques for Gaussian splats that adapt the workload based on the perceivable details. Franke et al. [FFS25] render the center-region with a high-quality, neural point-based renderer, and the periphery with low-resolution Gaussians. Tu et al. [TRS*25] propose lower shading rates in the periphery, switching from the typical 16×16 pixel tiles to 32×32 pixel tiles, combining and averaging groups of 2×2 pixels.

### 2.2. Editing

Editing of 3D representations encompasses a range of user-driven manipulations, such as selecting, removing, inpainting, relighting, re-texturing, and restyling elements of a scene. Early work on interactive editing of point-based datasets includes Pointshop 3D [ZPKG02], which supports operations like selection, brushing, and sculpting on surfel-based data. Subsequent advances by Wand et al. [WBB*07] and Scheiblauer et al. [SW11] introduced multi-resolution octree structures to facilitate editing of large-scale point clouds, scaling to billions of points. Scheiblauer et al. additionally proposed a selection octree to define and apply selection volumes in an out-of-core fashion. The PS4 game Dreams [Eva15] is especially notable for its capability to efficiently paint and sculpt large scenes on modest hardware, using a custom atomic-based software rasterizer and a structure described as "a cloud of clouds of point clouds". Neural radiance fields can be edited with NeRFshop [JKK*23], which provides volumetric deformations based on cages.

For Gaussian Splatting, manual editing capabilities remain relatively limited. The most notable system is SuperSplat [Ltd25], a browser-based open-source editor. It enables splat selection through a screen-brushing interaction, followed by selection refinement, based on intersected pixels. Selected splats can then be

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

duplicated and transformed. Beyond that, editing approaches for splat-based datasets have predominantly relied on prompt-driven paradigms, where users specify desired modifications via text or image input, and systems execute the corresponding edits automatically [WFZ*24; ZKC*24; BBM*24].

Segmentation-based approaches aim to cluster splats with similar attributes, thereby simplifying the selection of meaningful scene components. Once identified, these regions can be manipulated through operations such as translation or removal. Inpainting techniques complement this process by seamlessly reconstructing missing content, whether due to object deletion or imperfections in the original capture. To enable more precise control, several works integrate segmentation directly into the editing pipeline. Gaussian Grouping [YDYK25] introduces a general-purpose segmentation method for editing arbitrary objects, while Point'n Move [HYZN24] leverages segmentation alongside inpainting to enable object manipulation. Most recently, Chen et al. [CCZ*24] combine fast localized editing with segmentation-aware tools for an enhanced flexibility and control.

A growing body of work explores advanced editing paradigms for Gaussian splatting. Jambon et al. [JCZ*24] introduce a generative approach to scene manipulation. InFusion [LOW*24] leverages depth-completion via diffusion priors for inpainting tasks. Wang et al. [WYW*24] proposed style transfer and Xu et al. [XHL*25] texture manipulation. Text-based editing has become increasingly prominent, as demonstrated in GaussCtrl [WBL*24], GaussianEditor [WFZ*24]. TIP-Editor [ZKC*24] even supports both text- and image-driven instructions.

## 3. Method

In the following, we will cover our method for Gaussian Splat editing. We first cover the editing capabilities before focusing on the efficient rendering system to enable even VR use.

### 3.1. Editing

Our editor provides the following editing functionalities for geometry and colors, each accompanied by respective undo/redo operations (Section 3.1.5): selection/deletion via brushes, translation, rotation, scaling, duplication, and painting (splat color changes using a spherical brush).

All splat data reside in GPU memory, and the respective operations are implemented inside custom CUDA kernels. Operations are applied in a brute-force fashion, i.e., intersection tests or other conditionals are evaluated on all splats before applying them to the ones that pass. We originally used spatial acceleration structures, but the marginal increase in performance was outweighed by the substantial increase in implementation effort (Section 4).

In order to represent states such as *highlighted*, *selected*, or *deleted*, we store a 32-bit integer for each splat, where each potential state is encoded inside a single bit. This serves as an acceleration for the transformation of selected splats: We first quickly check the flags before loading an order of magnitude more of additional splat data. In the following, we first describe how we use virtual memory to deal with large data sets, then we give details on the individual editing operations.

#### 3.1.1. Dynamically Resizeable Buffers

While editing, splats may be removed from or added to individual splat models, which requires a form of dynamically-sized arrays. We opted to use CUDA's virtual memory management (VMM) [PS20] functionality for two reasons: we can store data in a single consecutive memory address range, even when it physically grows or shrinks, and it is as fast as a regular buffer allocation. A potential disadvantage of VMM is that the granularity of physical memory that we can add or remove to our buffers is fairly large – 2 MB on an RTX 4090. In case of positions, each chunk of allocated physical memory holds up to $\frac{2\,097\,152\text{ byte}}{12\text{ byte}} = 174\,762$ splats, i.e. tiny models may leave a sizeable amount of memory unused. Since we are focusing on models with millions to tens of millions of splats, this issue becomes negligible.

The way virtual memory works is that we first reserve a buffer with a virtual address range with a modestly large capacity ($\geq$100 MB). At this point, the buffer does not yet use up any of the GPU's actual memory until we start physically committing parts of it. When splats are added to a model (e.g., during merging), we "commit" as much physical memory as needed (with the aforementioned granularity of 2 MB). If the required physical memory exceeds the initially reserved virtual memory, we simply reserve a larger range of virtual memory and remap the previously allocated physical memory to the new virtual memory range without the need of copying the data.

In the same fashion, we can also "uncommit" physical memory that is no longer required, e.g., after deleting splats. To do so, splats are sorted, such that deleted splats are at the end of the buffer, and we can then safely de-allocate those chunks of physical memory.

#### 3.1.2. Selection and Deletion

SuperSplat [Ltd25] uses an indirect approach for selection and deletion, where artists first brush the screen, and the system then computes the splats that intersect with that selection. In contrast, our approach instantly highlights all splats that intersect with the brush during selection. The indirect approach is computationally inexpensive, which is especially useful for large splat models on lower-end hardware, and to deal with limitations in WebGL (such as the lack of compute shaders). The advantage of the direct approach is that users get to see the selected splats immediately, which can be useful to avoid mistakes. For virtual reality, in particular, direct selection is vital, as we cannot rely on a static viewpoint during an ongoing brushing operation.

Selection/deletion brushes may be either circular (screen space) or spherical (world space). Users may choose selections based on splat centers or their border. In case of circular brushes and splat-border intersection tests, we apply an approximation that checks whether the center of the brush circle is inside an ellipse that is enlarged by the radius of the brush (Figure 13).

The use of virtual memory and a sorting step allow us to remove deleted splats from memory at any time, without the need for additional intermediate/target buffers. We first sort all splats by their deletion flag, putting the deleted splats towards the end of the buffer. Since our editor uses virtual memory for splats, we can then unmap and release physical allocations at the buffer's end in steps of the

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

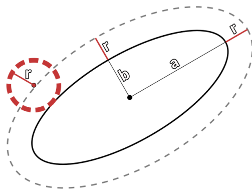



**Figure 2:** *Approximate circle-ellipse intersection test by adding the circle's radius to the ellipse's basis vectors, then checking whether the circle's position is inside the enlarged ellipse.*

GPU's granularity. (Note: Instead of using a radix sort, developers may want to implement this functionality with GPU-friendly stream compaction [BOA09]).

#### 3.1.3. Duplication and Merging

Besides populating the scene with multiple copies of an object, duplication also allows us to fill holes, similar to clone-stamp tools in 2D photo manipulation (Figure 3). Duplication itself is easily realized by copying all selected splats to a new scene node.

**Figure 3:** *Removal of an object, followed by hole-filling via duplication and translation of several patches of grass.*

Merging nodes with capacity A and B back together may pose a minor challenge since we want to avoid having to create an entirely new target buffer with sufficient capacity $A + B = C$ for the sum of all splats, thereby doubling the amount of memory that is needed during the merging operation. Virtual memory proves again beneficial since we can simply append A to B by increasing the capacity of B and then memcpy from source A, eliminating the need for an entirely new target buffer C. Further optimizations that we did not implement could merge source node A into target B progressively, reducing the memory usage of the merging process to as little as 2 MB: Instead of increasing the capacity of B by the size of A, we could increase B's capacity by CUDA's physical allocation granularity (2 MB, RTX 4090), then memcpy 2 MB over from A to B, release 2 MB from A, and repeat until all splats are transferred to the target buffer.

#### 3.1.4. Painting

Painting covers the need for recolorization (e.g., ancient statues whose pigments faded over time) or shadow additions (in the absence of proper illumination models, e.g., after adding assets). To implement a basic spherical brush, we check whether a splat center intersects with the brush's sphere, and, if it does, adjust its color

**(a)** *Painting graffiti on a wall*

**(b)** *Family Statue* **(c)** *Hue-and-saturation brush*

**Figure 4:** *(a) Painting in RGB color space. (c) Painting by adjusting only hue and saturation to the brush's color value, thereby preserving the texture and ambient occlusion of the original model.*

value. In order to support brushes with a smooth falloff and low opacity, we opted to use 16-bit color channels.

An issue we experienced is that splat models often do not lend themselves well to painting detailed features, such as text, lines, and symbols, since we are essentially vertex-painting sparse geometry. Figure 1 had a sufficient density to add a smiley face, but individual splats and undersampling are already visible. This issue is especially prevalent with models constructed from images for two reasons: They make an effort to generate as few splats as possible and focus splats on regions with high-frequency color patterns. The latter means that clean, single-colored surfaces, such as walls, use only few splats. Therefore, we cannot add detailed color changes there. In contrast, Fig. 4a shows a splat model that was generated from a textured triangle mesh [3dh17] using Mesh2Splat [Sco24]. It features detailed splats with uniform density and size, which is optimal for painting. Figure 4c proved sufficiently dense for a brush that modifies hue and saturation – demonstrating the use case of colorizing old Greek statues – but a certain amount of cherry-picking of the viewpoint is sometimes necessary since state-of-the-art Gaussian splat reconstructions rarely feature clean and opaque surfaces. In this case, it turns out that some splats within the statue still contribute significantly to the surface's appearance. This works for the original statue with homogeneous colors, but changes in appearance may lead to color leaks into differently colored parts of the statue's surface.

#### 3.1.5. Undo and Redo

The arguably most important feature of any editor is correcting mistakes and trying operations without permanent consequences. Keeping a growing history of changes and being able to apply them quickly poses challenges, such as memory usage and applying them

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

fast enough to avoid stutters in VR. The major question is how to compactly store differences for hundreds of editing actions?

For many actions, an identifier of said action along with a bitmask of the modified splats suffices to create a fairly compact diff. For example, selection and deletion are efficiently encoded by a single bit per splat that tells us whether the selection/deletion flag has changed during the action, allowing us to undo or redo it. Adding a single selection or deletion step to the history requires 1 MB state information per 8 million splats. If less than 1/32 of the splats are affected, we could think of encoding the diff more efficiently in the form of an array of 32-bit indices, but in light of the already low memory requirements of bit masks, we did not explore this option.

Undoing painting actions requires additional information. In our case, we append a list of each modified splat's index and previous color value to the history, requiring 4 (index) + 8 (color) bytes of memory per modified splat. At the start of a painting operation however – before creating the compact diff – we duplicate the model's entire color buffer and freely modify the duplicate's values. After the user finishes a paint stroke, we then check the original and duplicated color buffers to identify modified values, and create a new compact diff that is added to our undo/redo history. Note that at this point, the diff only contains the splat's previous color values – we can go back in time but not forward. To fix this, during the undo operation, we swap the previous color values from the diff with the modified color values in the model, converting undo into redo data.

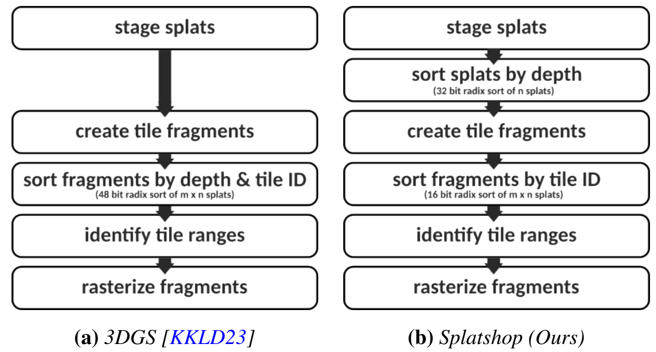
Diffs supporting lossless undo/redo of transformations require large amounts of memory since, like painting, we need to store the modified splat's previous values. For geometry, this amounts to position (12) + scale (12) + rotation (16) = 40 bytes of memory per modified splat. In the presence of spherical harmonics, this may grow by another 180 bytes per splat. To avoid that explosion of memory, we opted for lossy undo/redo of transformations at the current stage. We store the transformation matrix that led to the change and during undo, we apply the inverse (which may not recover the exact previous values). This works well for smaller scenes, where 32-bit floats provide sufficient precision for coordinates, as well as rotations with highly precise floating-point quaternion values. Scaling is the main concern, as extreme scaling factors may degrade both the resulting positions and scaling values of each splat.

### 3.2. Rendering

The rendering pipeline builds on the original 3DGS paper but introduces important differences to increase efficiency: The separation of the 48-bit tileID+depth sort into a 32-bit depth-sort of the smaller amount of visible splats, followed by a 16-bit tileID sort of the larger amount of generated duplicates/fragments; an approximate splat-tile intersection test that avoids creating fragments for tiles that are not touched by a splat; and discarding small splats to avoid slowdowns when zooming out. In the following, we recap the 3DGS system and detail our differences.

#### 3.2.1. Staging Splats

This pass creates a per-visible-splat list of depth values, their location on screen, the two basis vectors that describe each splat's elliptical shape on the screen, their view- and state-dependent colors,




**(a)** *3DGS [KKLD23]* **(b)** *Splatshop (Ours)*

**Figure 5:** *In comparison to 3DGS, our rendering pipeline first sorts visible splats by depth before creating per-tile lists of overlapping splats. This improves performance significantly as the majority of splats will contribute to the rendered image across multiple tiles.*

and the number of tiles each splat touches. Similar to the implementation of Kerbl et al. [KKLD23], each thread processes a single 3D Gaussian. First, the 3D covariance matrix of each Gaussian is constructed from its rotation and scaling information. By locally approximating the perspective projection with an affine transformation [ZPvBG01], the 3D Gaussian is then projected onto the image plane, resulting in a 2D Gaussian. Given its 2D covariance matrix, we compute the elliptical extent of each splat in screen space.

**Evaluating Tile-Splat Intersections.** Contrary to 3DGS, we reduce the number of tiles per splat; from all tiles that intersect with the splat's bounding-circle's bounding box, down to all tiles whose bounding-circle intersects with the splat. To compute these intersections, we use approximate circle-ellipse intersection tests shown in Figure 13 and Appendix B. Inaccuracies in the test mostly occur near the side of a splat's sharp end, where its potential impact on the rendered image is minimal due to the low remaining opacity (see Figure 6).

A simpler solution is to work with a tight AABB around the ellipse. Using the 2D basis vectors $a, b$ shown in Figure 13, the extent of this AABB along x and y is $2\sqrt{a_0^2 + b_0^2}$ and $2\sqrt{a_1^2 + b_1^2}$, respectively (see Appendix A). Hanson et al. concurrently published a similar formula that also produces the tight AABB around an ellipse [HTL*25]. However, our formula is more thoroughly simplified, which saves a few arithmetic instructions. In Section 4.6, we evaluate all variants shown in Figure 6. We note that exact tile-based culling has also been explored by Radl et al. in StopThePop [RSP*24]. For gradient descent-based optimization, it is important to use an intersection test that avoids false-negatives as this might negatively influence convergence of the optimization. In contrast, our editor operates only on fully-trained models, which justifies our choice of using an approximate but more efficient approach for performing false-positive-free tile list creation.

**Progressively Updating View-Dependent Colors.** High-degree spherical harmonics are a major performance bottleneck due to their pressure on memory bandwidth. For example, SH degree 3 adds 45 additional floating-point values (16 coefficients times three for

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

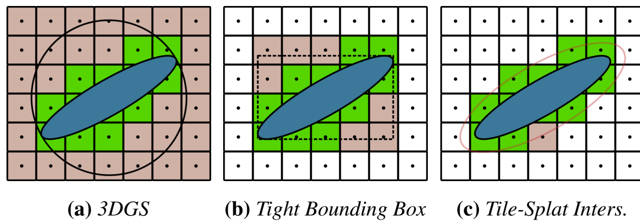



**Figure 6:** *Created duplicates/fragments for intersected tiles. (a) 3DGS duplicates splats for each tile within the bounding box of the ellipse's bounding circle, leading to numerous false-positives. (b) A tight-fitting bounding box removes the majority of false-positives. (c) We further reduce the amount of fragments with an approximate tile-ellipse intersection test, with occasional false-negatives that do not affect perceived results (Section 4.6).*

RGB, minus three since the base color is handled separately) that need to be evaluated during rendering to obtain a view-dependent color value. Assuming 32-bit floating point storage, this adds a total of $4 \times 45 = 180$ bytes per splat that are loaded during the staging pass (more than triple the 52 bytes used for splat position, scale, orientation, color, and flags). To alleviate this issue, we perform progressive evaluation of the spherical harmonics over multiple, e.g., ten frames and write the results in a "baked" view-dependent color buffer. Under sufficiently high frame rates, which we target, the view-dependent color values are updated quickly enough to cause little to no apparent lag, even under motion.

In VR, progressive updates need to account for the different camera direction vectors of splats towards the left and right eye. We found it sufficient to progressively bake color values for only one eye, and use the same colors for the other. Alternatively, developers could evaluate SHs based on the direction towards the center of both eyes, or create separate baked-color buffers for each eye.

**Sorting Splats by Depth.** We sort the list of staged splats by their depth values, using a 32-bit radix sort [b0n25], taking depth as key and staged-splat indices as values to be sorted. In the original 3DGS pipeline, sorting is done after the splats were split into fragments – one for each tile they touch (refered to as "duplicateWithKeys" in 3DGS). The reason is that we need a list of tile fragments that are grouped by tile ID and sorted by depth within each tile. 3DGS achieves this by assigning a 48-bit key to each fragment, with the 16-bit tile ID in the most significant bits and the 32-bit depth in the least significant bits. Radix sort is then applied to the combined 48-bit of each fragment. In practice, the number of fragments tends to be about 1.5 to 4× higher than the number of visible splats, thus, sorting visible splats by depth greatly reduces the sorting costs.

**Creating Tile Fragments.** This stage creates one tile fragment item for each tile a splat touches. Since we already sorted splats by depth, we can now create a list of fragments that are also sorted by depth. To do so, we first compute the exclusive prefix sum for all staged splats' numbers of touched tiles, which will give us the offsets into which we store the fragments that correspond to a splat. For example, if three splats touch 3, 2, 4 tiles, respectively, the resulting prefix sum is 0, 3, 5. As a result, the three tile fragments of the first splat are stored starting at offset 0, the two tile fragments of the second one are stored starting at offset 3, and for the third splat we start storing fragments at offset 5. In doing so, all created fragments inherit the splat's previous ordering by depth.

**Sorting Depth-Sorted Tile Fragments by Tile ID.** For rendering, we need a per-tile list of depth-sorted fragments. Since fragments are already sorted by depth, we can easily group them into tiles by using a stable sort with the tile ID as the key. Fortunately, most state-of-the-art radix sort algorithms for the GPU are based on bottom-up, least-significant-bit-first sorting, where being stable is a part of the algorithm's requirements. For sorting, we compute a 16-bit tile ID that we use as a key for a 16-bit radix sort.

As the last step before rasterizing the fragments, we compute the indices of the first and the last splat for each tile in the same way as 3DGS. We launch a kernel with one thread per fragment where each thread checks if the previous and the current fragment have the same tile ID. If the tile IDs differ, we found a jump between two tiles. The thread then updates the previous tile's *lastIndex* and the current tile's *firstIndex*.

**Rasterizing Tile Fragments.** Rasterization closely follows the original 3DGS approach. For each tile, we launch one CUDA thread-block, comprising of one thread per pixel (16×16 threads), which blends the splats assigned to that tile in front-to-back order. The iteration process ping-pongs between utilizing 128 threads to load 128 splats into shared memory and then having each of the 256 threads iterate through the 128 loaded splats to blend them together.

#### 3.2.2. Multi-View Coherent VR Rendering

While interacting with splats in VR, we found that there are two prominent perceptual issues during immersion: popping between splats under view rotation, and the shape of splats not being consistent across different viewpoints. Both problems are a result of the approximate projection of the 3D Gaussians onto the image plane, effectively rendering them as screen-facing billboards. Prior work proposes, for example, to use hierarchical sorting [RSP*24] or OptiX-based ray tracing [MMP*24] to achieve multi-view consistency. Both, however, result in significantly slower rendering, which is a major disadvantage for their usage in VR applications, as these require high frame rates to mitigate cybersickness.

As an alternative solution, Hahlbohm et al. [HFW*25] propose to use hybrid transparency [MCTB13] to accelerate view-consistent rendering of 3D Gaussians. We integrate their approach into our editor as a second rendering mode that can be toggled on or off. During splat staging, we adopt their plane-fitting approach for computing exact bounding boxes of each 3D Gaussian in screen space. As hybrid transparency allows locally resolving depth ordering during blending, we can omit the depth ordering step and directly create unordered per-tile fragment lists. During blending, exact ray casting is used to evaluate each Gaussian at the point of maximum contribution along pixel-specific viewing rays directly in 3D. The foremost $K = 16$ contributions in each pixel are then alpha-blended and composited with an order-independent tail that combines all remaining $N - K$ contributions.

Compared to the original implementation by Hahlbohm et

© 2025 The Author(s).

al. [HFW*25], we make two changes to improve performance. First, we apply load balancing during tile fragment creation similar to Radl et al. [RSP*24] and second, each thread uses a more compact 64-bit data structure to store core fragments during blending. The latter integrates well with our internal color representation while reducing the number of required registers and swaps during the insertion sort thus increasing performance. Although slightly slower than our optimized 3DGS-based rendering, the absence of popping artifacts and perspective-correct rendering of 3D Gaussians are of great benefit to the VR viewing and editing experience.

### 3.3. Imbalanced Workloads & VR Rendering

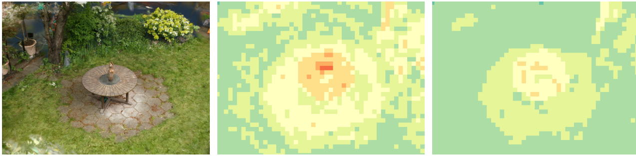


**Figure 7:** *Heatmap depicting the number of splats per tile. Without culling small splats (middle), and with culling small splats (right).*

We observed that tiles with a large number of splats are a major rendering bottleneck. Since we launch one thread-block per tile during rasterization, blocks assigned to tiles with only few splats will finish quickly, while blocks assigned to tiles with many splats take much longer. Figure 7 demonstrates such imbalances in the number of splats per tile. Towards the end of the rasterization stage, only few thread-blocks will remain active, which causes severe under-utilization of GPU resources as shown in Figure 8.

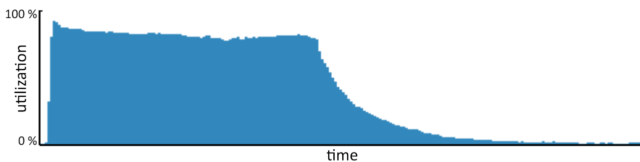
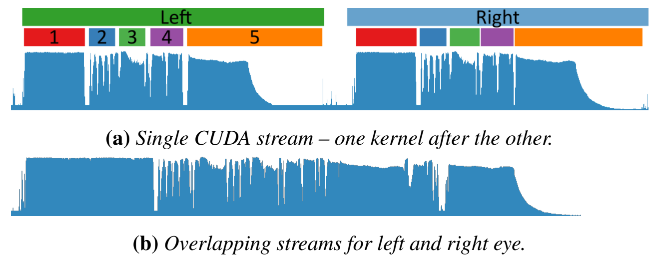



**Figure 8:** *GPU-utilization of the splat rasterization kernel.*

VR rendering provides us with the opportunity to fill some of the gaps in utilization by overlapping the rendering pipelines for the left and the right eye, which we can realize by launching kernels in separate, non-blocking CUDA streams. In Fig. 9a, we observe that significant under-utilization is mainly limited to the rasterization kernel, where some gains were obtained by overlapping the staging of splats for the left and right eye (fusing both mostly-orange parts). We did not overlap the left and the right eye's sorting stages since this would require creating separate intermediate sorting buffers, thus increasing memory usage. However, we did overlap the right eye's tile ID sorting and rasterization stages with the left eye's rasterization stage, which is responsible for the majority of improvements in utilization.

## 4. Evaluation

Our editor was implemented in C++ and CUDA, with OpenGL interop to display the results. Performance was evaluated on a system




**(a)** *Single CUDA stream – one kernel after the other.*

**(b)** *Overlapping streams for left and right eye.*

**Figure 9:** *GPU utilization in VR. (1) Splat staging (2) depth sorting (3) creating tile fragments (4) tile ID sorting (5) rasterization. Captured with Nsight Graphics GPU Trace Profiler.*

with Windows 11, an RTX 4090 GPU, an AMD Ryzen 9 7950X CPU, and a Valve Index HMD. The framebuffer size was set to 1920×1080 pixels for desktop rendering, and to 2016×2240 per eye for virtual reality rendering.

We used the test data sets shown in Figure 10. The Garden scene is provided as a pre-trained model from 3DGS [KKLD23], while the Campus scene is a pre-trained model from VastGaussians [LLT*24]. The Kinuta and Berlin scenes were reconstructed by us using Postshot and the 3DGS implementation by Kerbl et al. [KKLD23] respectively. The images from the Berlin scene are taken from Barron et al. [BMV*23]. The Berlin scene is of interest as it features three rooms, of which two are occluded at any given time. Timings are captured with CUDA events and computed as the median over 50 frames. Compaction and undo (Section 4.1) are an exception and computed as the median over five repetitions.

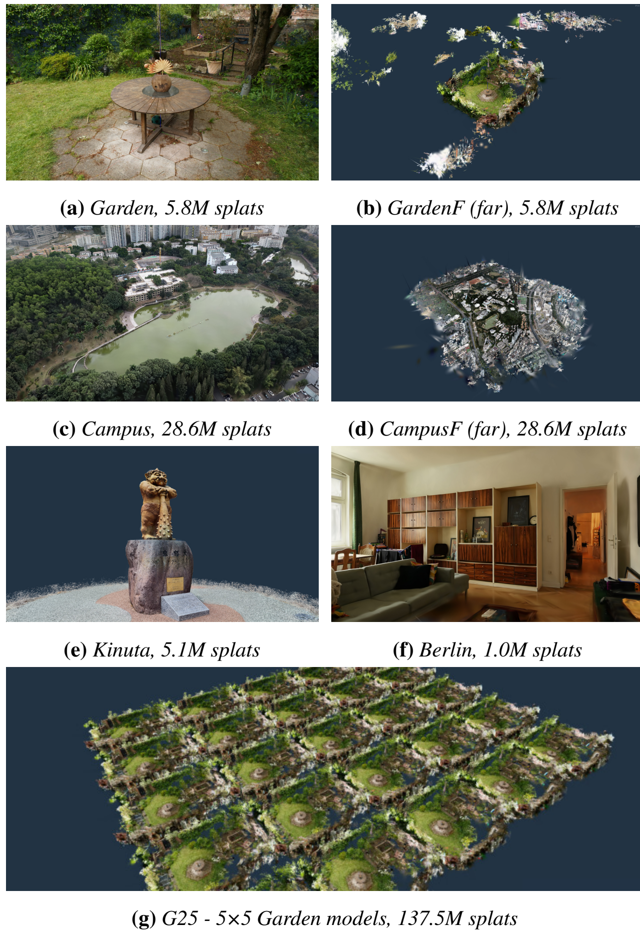
### 4.1. Painting

**Table 1:** *Painting Performance (Spherical Brush)*

| | splats | painting | compaction | undo |
|---|---|---|---|---|
| Berlin | 1.0 M | 0.03 - 0.05 ms | 0.32 - 0.91 ms | 0.08 ms |
| Kinuta | 5.1 M | 0.01 - 0.23 ms | 0.47 - 2.16 ms | 0.30 ms |
| Garden | 5.8 M | 0.14 - 0.27 ms | 0.50 - 2.60 ms | 0.34 ms |
| Campus | 28.6 M | 0.50 - 1.70 ms | 1.60 - 9.20 ms | 2.29 ms |
| G25 | 137.5 M | 2.48 - 6.47 ms | 8.36 - 51.63 ms | 8.64 ms |

Painting adds a certain amount of GPU utilization in each frame, and requires compaction whenever a stroke is finished (button release). Minimum times are observed with tiny brushes that affect tens of splats, while maximum times and undo times are measured with massive brushes that affect all splats. Since all splats are brute-force evaluated, a certain small overhead for intersection tests is always added to the frame time during painting. Compaction adds another overhead, as the host syncs with the device, and the number of affected splats is retrieved to then allocate new GPU buffers that are sufficiently large to accommodate the compacted diff between original and modified colors. Undo/redo buffers grow linearly with the number of affected splats that are stored in the compacted diff.

© 2025 The Author(s).



(a) *Garden, 5.8M splats* (b) *GardenF (far), 5.8M splats*
(c) *Campus, 28.6M splats* (d) *CampusF (far), 28.6M splats*
(e) *Kinuta, 5.1M splats* (f) *Berlin, 1.0M splats*
(g) *G25 - 5×5 Garden models, 137.5M splats*

**Figure 10:** *Our test data sets.*

### 4.2. Transformation

**Table 2:** *Splat Transformation Performance*

| | splats | SH Degree | duration |
|---|---|---|---|
| Berlin | 1.0 M | 3 | 0.01 - 1.65 ms |
| Kinuta | 5.1 M | 3 | 0.03 - 6.88 ms |
| Garden | 5.8 M | 3 | 0.03 - 7.82 ms |
| Campus | 28.6 M | 0 | 0.16 - 2.63 ms |
| | | 1 | 0.16 - 9.49 ms |
| | | 2 | 0.16 - 21.57 ms |
| | | 3 | 0.16 - 37.42 ms |
| G25 | 137.5 M | 0 | 0.76 - 13.74 ms |

The minimum duration is experienced while selecting tens of splats, and the maximum duration applies when selecting all splats. Although the transformation kernel brute-force processes all splats, the overhead for non-selected splats remains small since we exit early upon checking a splat's selection flag. Undo and redo utilize the same CUDA kernels as the main transformation logic, and thus exhibit the same performance characteristics. Spherical harmonics have a major impact on transformation performance: Rotating degree 3 harmonics reduces the performance by a factor of up to 14. However, translation and scaling operations do not need to update harmonics and may therefore implement a fast path.

### 4.3. Separating Depth and Tile Sorting

We are not able to directly measure the original 3DGS 6-digit / 48-bit radix sorting since our editor is based on the CUDA driver API, but the CUDA CUB used by 3DGS requires the CUDA runtime API instead. However, radix sort's runtime scales linearly with the number of bits in the sorting key, and we can assume that the 48-bit radix sort is three times as expensive as the 16-bit sort, which we added to GPUSorting [b0n25] by simply stopping after sorting two 8-bit digits.

**Table 3:** *Comparing 32-bit sort of visible splats + 16-bit sort of duplicate fragments to a traditional 48-bit sort of duplicate fragments. With small-splat-culling (SSC) and without.*

| | | depth (32 bit) | tileID (16 bit) | total | 48 bit |
|---|---|---|---|---|---|
| w/ SSC | Berlin | 0.11 ms | 0.13 ms | 0.24 ms | 0.39 ms |
| w/ SSC | Kinuta | 0.24 ms | 0.20 ms | 0.44 ms | 0.60 ms |
| w/ SSC | Garden | 0.15 ms | 0.16 ms | 0.31 ms | 0.48 ms |
| w/ SSC | Campus | 0.38 ms | 0.62 ms | 1.00 ms | 1.86 ms |
| w/ SSC | CampusF | 0.45 ms | 0.41 ms | 0.86 ms | 1.23 ms |
| w/ SSC | G25 | 0.52 ms | 0.39 ms | 0.91 ms | 1.17 ms |
| | Berlin | 0.12 ms | 0.14 ms | 0.26 ms | 0.42 ms |
| | Kinuta | 0.27 ms | 0.24 ms | 0.51 ms | 0.72 ms |
| | Campus | 0.41 ms | 0.53 ms | 0.94 ms | 1.59 ms |
| | CampusF | 2.74 ms | 1.89 ms | 4.63 ms | 5.67 ms |

Table 3 shows the time it takes to sort a smaller number of visible splats by 32-bit depth values first, and then the larger number of tile-wise fragments by 16-bit tileIDs. The sorting time of our approach is compared to 3DGS's combined tileID+depth 48-bit sort by comparing our total duration to sorting by 3× tileID.

### 4.4. Single vs. Concurrent CUDA Streams (VR)

Imbalanced workloads, like the rasterization of tiles with varying amounts of splats, under-utilize the GPU, as the stream waits until the kernel's longest-running thread-blocks are finished (see Section 3.3). In VR, we have the opportunity to concurrently run CUDA kernels for the left and right eye in multiple streams, leading to a better utilization (Figure 9). In doing so, we observed the performance improvements shown in Table 4.

**Table 4:** *Concurrent Stream Performance*

| | splats | single stream | concurrent streams |
|---|---|---|---|
| Berlin | 1.0 M | 6.3 ms | 5.2 ms |
| Kinuta | 5.1 M | 9.0 ms | 7.7 ms |
| Garden | 5.8 M | 8.1 ms | 6.5 ms |
| Campus | 28.6 M | 16.2 ms | 13.8 ms |

© 2025 The Author(s).

### 4.5. Progressive Spherical Harmonics

Higher-degree spherical harmonics can increase the time to stage splats due to the large amount of memory they require and the related bandwidth bottlenecks. Updating the harmonics progressively over multiple frames and storing the results in "baked" color buffers leads to significant performance improvements with typically barely noticeable artifacts, especially with quick convergence settings, such as fully updating all visible splats within 10 frames. Table 5 shows the differences in performance for these approaches.

**Table 5:** *Progressive Spherical harmonics - Time to* stageSplats *without SHs; with deg. 3 SHs (+45 floats) of all visible splats each frame; and progressively evaluating a fraction of SHs each frame.*

| | none | all | 1 / 10th | 1 / 30th |
|---|---|---|---|---|
| Berlin | 0.1 ms | 0.4 ms | 0.2 ms | 0.1 ms |
| Kinuta | 0.4 ms | 1.6 ms | 0.6 ms | 0.5 ms |
| Garden | 1.8 ms | 2.6 ms | 1.9 ms | 1.8 ms |
| Campus | 1.1 ms | 4.4 ms | 1.5 ms | 1.1 ms |

### 4.6. Approximate Tile-Wise Culling

To justify our intersection test with false-negatives during rendering (Section 3.2.1 and Figure 6), we computed image difference metrics between a rendering with and without missing tile fragments:

**Table 6:** *Image Difference of Approximate Culling*

| | PSNR↑ | SSIM↑ | FLIP↓ | LPIPS↓ |
|---|---|---|---|---|
| Berlin | 41.83 dB | 0.9996 | 0.0010 | 0.0021 |
| Kinuta | 82.86 dB | 1.0000 | 0.0000 | 0.0000 |
| Garden | 75.82 dB | 1.0000 | 0.0002 | 0.0000 |
| Campus | 49.89 dB | 0.9999 | 0.0002 | 0.0002 |

The particularly extreme values for perceptual metrics SSIM, FLIP, and LPIPS [ZIE*18] suggest that renderings with correct and approximate intersection tests are virtually indistinguishable. Likewise, the PSNR values are as high or higher than typical JPEG-compressed images at highest quality settings, also indicating that approximate culling is imperceptible.

With respect to rendering performance, tight bounding boxes and approximate tile-wise culling affect the amount of duplicated fragments and rendering times as follows:

**Table 7:** *Per-tile fragments created during splat rendering with 3DGS intersection method, tight bounding box around the ellipse, and approximate tile-ellipse intersection tests (See Figure 6).*

| | #Fragments (M) | | | Rendering time (ms) | | |
|---|---|---|---|---|---|---|
| | 3DGS | Tight | Appr. | 3DGS | Tight | Appr. |
| Berlin | 8.1 | 6.3 | 5.9 | 3.0 | 2.7 | 2.6 |
| Kinuta | 9.2 | 7.1 | 6.7 | 3.6 | 3.2 | 3.1 |
| Garden | 8.2 | 5.1 | 4.4 | 2.7 | 2.0 | 2.0 |
| Campus | 16.8 | 10.8 | 9.6 | 6.2 | 5.3 | 5.2 |
| Virtual Reality | | | | | | |
| Berlin | 30.3 | 12.8 | 9.8 | 9.3 | 5.6 | 5.2 |
| Kinuta | 22.6 | 16.0 | 14.5 | 9.7 | 8.1 | 7.7 |
| Garden | 20.8 | 12.1 | 9.9 | 6.2 | 4.3 | 4.0 |
| Campus | 43.6 | 28.5 | 25.7 | 16.5 | 14.1 | 13.7 |

Table 7 shows that using the tight bounding box has a major impact on the number of splats and rendering times. Approximate tile-wise intersection tests further reduce the amount of staged fragments but show little impact when using the default framebuffer size of 1920×1080 pixels, because even though it slightly reduces the workload for the sorting and rasterization stages, it also slightly increases the duration of the fragment creation stage. However, the impact of approximate culling increases with the framebuffer size and is especially prominent in VR rendering, where we are rendering a total of 4.36 times as many pixels as in desktop mode.

### 4.7. Culling Small Splats

**(a)** *No Culling; 9.7 ms* **(b)** *Culling; 2.1 ms*

**Figure 11:** *Close-up of GardenF scene and viewpoint.*

Unlike reconstruction applications, which need to maximize rendering quality in order to construct high-quality splat data sets, we can afford to (optionally) reduce visual quality in order to maintain real-time frame rates. During staging, we discard splats that are smaller than a certain threshold as their contribution to the image are likely to be smaller. Figure 11 depicts an inset of GardenF (Figure 10b) that demonstrates the differences between no culling and culling enabled. Without culling, 5.79 out of 5.84 million splats are staged, and 8.16 million fragments are created. With culling, 1.25 million splats are staged and 1.99 million fragments are created, thus significantly reducing the workload. The time to render the

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

scene decreases from 9.7 ms to 2.1 ms. To capture high-quality screenshots, users can always disable culling.

This approach tends to work well for 3DGS-generated splat data sets that comprise a mixture of wildly different splat sizes. Even at greater distances, where the majority of splats are discarded, many larger semi-representative splats remain. It does not, however, work with scenes comprising solely small splats, such as models constructed via Mesh2Splat [Sco24].

## 5. Discussion and Future Work

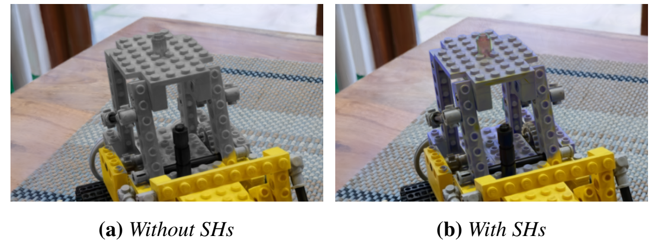
**Memory-Efficient Spherical Harmonics.** Wiederien and Sloan [TYL24] propose an efficient quantization of spherical harmonics that could be used as an alternative or in addition to progressive evaluations.

**Lossless Undo/Redo of Transformations.** Diffs of transformations are currently lossy, potentially degrading the model over multiple undo/redo iterations. Future work may consider memory-efficient lossless diffs, potentially incorporating state-of-the-art compression algorithms.

**Spatial Acceleration Structures.** Initially we attempted to use a spatial acceleration structure that groups, e.g., 256 consecutive, Morton-ordered splats, and computes their bounding box. The idea was that we could use this structure for frustum-culling and to reduce the effort of finding intersections during brushing, painting, selection, etc. We eventually gave up on this and opted for a brute-force approach instead for three reasons: The performance gains were small compared to the cost of rendering; the development efforts raised greatly; and editing would occasionally affect most splats, when increasing brute-force performance is necessary either way. As the performance evaluation of painting and transformations show, brute-forcing over up to a 100 million splats is a viable strategy. To improve performance with reasonable development effort, we would like to try integrating acceleration-structures on-demand instead, e.g., spending a millisecond or two to create a structure at the start of specific operations, such as painting in VR.

**Hierarchical Level of Detail.** Although brute-forcing over all splats has shown to be viable for up to around 100 million splats, LODs will eventually be necessary to support higher amounts of splats with better quality on less efficient hardware. Editing is challenging in that regard since the geometry frequently changes, but projects like Dreams [Eva15] have shown this to be possible.

**Painting and Spherical Harmonics.** Our painting functionality currently only modifies the base splat colors, but not the spherical harmonics. Theoretically, the unmodified spherical harmonics could lead to deviations from the desired results of painting or color correction operations. In practice, we did not observe notable issues, except when strongly reducing the saturation of the model's base colors, as shown in Figure 12. At some point, the typically subtle spherical harmonics take over and impose unintended changes to saturation and hue.



**(a)** *Without SHs* **(b)** *With SHs*

**Figure 12:** *Painting currently does not modify spherical harmonics. In special cases such as reducing saturation, the unaltered spherical harmonics may cause undesired changes in hue and saturation.*

## 6. Conclusion

We presented a new system, Splatshop, which offers a highly optimized solution for interactively editing (selection, deletion, painting, transformation, ...) 3D Gaussian Splatting models. Our method enables real-time performance in scenes with up to 100 million primitives. It is the first solution to support VR for editing large-scale GS scenes and, hereby, marks a significant step towards their use in immersive applications and, along with alternatives like Supersplat, a further step towards Photoshop-like editing of Gaussian splat models.

The source code for this paper is available at https://github.com/m-schuetz/splatshop.

## 7. Acknowledgements

The authors wish to thank Kerbl and Koppanas et al. for the splat model of the Garden data set [KKLD23]; Baron et al. for the Garden [BMV*22] and Berlin [BMV*23] data sets; Lin and Li et al. for the Campus data set [LLT*24]; Knapitsch et al. for the family statue data set [KPZK17]; and Sketchfab user 3dhdscan for the Ruin scan [3dh17].

This research has been funded by WWTF project *ICT22-055 - Instant Visualization and Interaction for Large Point Clouds* and been partially supported by Convergence AI's Immersive AI and Tech Lab at TU Delft.

## References

[3dh17] 3DHDSCAN. *Ruin - InteriorScanChallenge*. 2017. URL: https://sketchfab.com/3d-models/ruin-interiorscanchallenge-76b02dc9a081491797733e9a89a00bb5 4, 10.

[b0n25] (B0NES164), THOMAS SMITH. *GPUSorting*. github. Accessed 2025.01.02. 2025. URL: https://github.com/b0nes164/GPUSorting/ 6, 8.

[BBM*24] BARTHEL, FLORIAN, BECKMANN, ARIAN, MORGENSTERN, WIELAND, et al. *Gaussian Splatting Decoder for 3D-aware Generative Adversarial Networks*. 2024. arXiv: 2404.10625 [cs.CV] 3.

[BMV*22] BARRON, JONATHAN T., MILDENHALL, BEN, VERBIN, DOR, et al. "Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields". *CVPR* (2022) 10.

[BMV*23] BARRON, JONATHAN T., MILDENHALL, BEN, VERBIN, DOR, et al. "Zip-NeRF: Anti-Aliased Grid-Based Neural Radiance Fields". *ICCV* (2023) 7, 10.

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

[BOA09] BILLETER, MARKUS, OLSSON, OLA, and ASSARSSON, ULF. "Efficient stream compaction on wide SIMD many-core architectures". *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, 159–166. ISBN: 9781605586038. DOI: 10.1145/1572769.1572795. URL: https://doi.org/10.1145/1572769.1572795 4.

[CCZ*24] CHEN, YIWEN, CHEN, ZILONG, ZHANG, CHI, et al. "GaussianEditor: Swift and Controllable 3D Editing with Gaussian Splatting". *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2024, 21476–21485 3.

[Eva15] EVANS, ALEX. "Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game". *ACM SIGGRAPH 2015 Courses, Advances in Real-Time Rendering in Games*. https://advances.realtimerendering.com/s2015/ [Accessed 14-April-2025]. 2015 2, 10.

[FFS25] FRANKE, LINUS, FINK, LAURA, and STAMMINGER, MARC. "VR-Splatting: Foveated Radiance Field Rendering via 3D Gaussian Splatting and Neural Points". *Proc. ACM Comput. Graph. Interact. Tech.* 8.1 (May 2025). DOI: 10.1145/3728302. URL: https://doi.org/10.1145/3728302 2.

[HFW*25] HAHLBOHM, FLORIAN, FRIEDERICHS, FABIAN, WEYRICH, TIM, et al. "Efficient Perspective-Correct 3D Gaussian Splatting Using Hybrid Transparency". *Computer Graphics Forum* 44.2 (2025). DOI: 10.1111/cgf.70014 2, 6, 7.

[HTL*25] HANSON, ALEX, TU, ALLEN, LIN, GENG, et al. "Speedy-Splat: Fast 3D Gaussian Splatting with Sparse Pixels and Sparse Primitives". *CVPR*. 2025 2, 5.

[HYZN24] HUANG, JIAJUN, YU, HONGCHUAN, ZHANG, JIANJUN, and NAIT-CHARIF, HAMMADI. "Point'n Move: Interactive scene object manipulation on Gaussian splatting radiance fields". *IET Image Processing* 18.12 (2024), 3507–3517. DOI: https://doi.org/10.1049/ipr2.13190. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/ipr2.13190. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/ipr2.13190 3.

[JCZ*24] JAMBON, CLÉMENT, CHOI, CHANGWOON, ZHANG, DONGSU, et al. *Interactive Scene Authoring with Specialized Generative Primitives*. 2024. arXiv: 2412.16253 [cs.CV]. URL: https://arxiv.org/abs/2412.16253 3.

[JKK*23] JAMBON, CLÉMENT, KERBL, BERNHARD, KOPANAS, GEORGIOS, et al. "Nerfshop: Interactive editing of neural radiance fields". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 6.1 (2023) 2.

[KKLD23] KERBL, BERNHARD, KOPANAS, GEORGIOS, LEIMKÜHLER, THOMAS, and DRETTAKIS, GEORGE. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". *ACM Transactions on Graphics* 42.4 (July 2023). URL: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/ 1, 2, 5, 7, 10.

[KMK*24] KERBL, BERNHARD, MEULEMAN, ANDREAS, KOPANAS, GEORGIOS, et al. "A Hierarchical 3D Gaussian Representation for Real-Time Rendering of Very Large Datasets". *ACM Transactions on Graphics* 43.4 (July 2024). URL: https://repo-sam.inria.fr/fungraph/hierarchical-3d-gaussians/ 2.

[KPZK17] KNAPITSCH, ARNO, PARK, JAESIK, ZHOU, QIAN-YI, and KOLTUN, VLADLEN. "Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction". *ACM Transactions on Graphics* 36.4 (2017) 10.

[KVK*25] KHERADMAND, SHAKIBA, VICINI, DELIO, KOPANAS, GEORGE, et al. *StochasticSplats: Stochastic Rasterization for Sorting-Free 3D Gaussian Splatting*. 2025. arXiv: 2503.24366 [cs.CV]. URL: https://arxiv.org/abs/2503.24366 2.

[LLT*24] LIN, JIAQI, LI, ZHIHAO, TANG, XIAO, et al. "VastGaussian: Vast 3D Gaussians for Large Scene Reconstruction". *CVPR*. 2024 7, 10.

[LOW*24] LIU, ZHIHENG, OUYANG, HAO, WANG, QIUYU, et al. "InFusion: Inpainting 3D Gaussians via Learning Depth Completion from Diffusion Prior". *arXiv preprint arXiv:2404.11613* (2024) 3.

[Ltd25] LTD., PLAYCANVAS. *SuperSplat*. github. Accessed 2025.01.02. 2025. URL: https://github.com/playcanvas/supersplat/ 2, 3.

[MCTB13] MAULE, MARILENA, COMBA, JOÃO, TORCHELSEN, RAFAEL, and BASTOS, RUI. "Hybrid transparency". *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '13. Orlando, Florida: Association for Computing Machinery, 2013, 103–118. ISBN: 9781450319560. DOI: 10.1145/2448196.2448212. URL: https://doi.org/10.1145/2448196.2448212 2, 6.

[MHK*24] MAI, ALEXANDER, HEDMAN, PETER, KOPANAS, GEORGE, et al. *EVER: Exact Volumetric Ellipsoid Rendering for Real-time View Synthesis*. 2024. arXiv: 2410.01804 [cs.CV]. URL: https://arxiv.org/abs/2410.01804 2.

[MMP*24] MOENNE-LOCCOZ, NICOLAS, MIRZAEI, ASHKAN, PEREL, OR, et al. "3D Gaussian Ray Tracing: Fast Tracing of Particle Scenes". *ACM Trans. Graph.* 43.6 (Nov. 2024). ISSN: 0730-0301. DOI: 10.1145/3687934. URL: https://doi.org/10.1145/3687934 6.

[MST*21] MILDENHALL, BEN, SRINIVASAN, PRATUL P, TANCIK, MATTHEW, et al. "Nerf: Representing scenes as neural radiance fields for view synthesis". *Communications of the ACM* 65.1 (2021), 99–106 1, 2.

[PS20] PERRY, CORY and SAKHARNYKH, NIKOLAY. *Introducing Low-Level GPU Virtual Memory Management*. NVIDIA Developer Blog. Accessed 2025.01.02. 2020. URL: https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/ 3.

[RSP*24] RADL, LUKAS, STEINER, MICHAEL, PARGER, MATHIAS, et al. "StopThePop: Sorted Gaussian Splatting for View-Consistent Real-time Rendering". *ACM Transactions on Graphics* 4.43 (2024) 2, 5–7.

[Sco24] SCOLARI, STEFANO. "Mesh2Splat: Gaussian Splatting from 3D Geometry and Materials". MA thesis. KTH Royal Institute of Technology, 2024. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-359582 4, 10.

[SW11] SCHEIBLAUER, CLAUS and WIMMER, MICHAEL. "Out-of-Core Selection and Editing of Huge Point Clouds". *Computers & Graphics* 35.2 (Apr. 2011), 342–351. ISSN: 0097-8493. URL: https://www.cg.tuwien.ac.at/research/publications/2011/scheiblauer-2011-cag/ 2.

[TRS*25] TU, XUECHANG, RADL, LUKAS, STEINER, MICHAEL, et al. "VRsplat: Fast and Robust Gaussian Splatting for Virtual Reality". *Proc. ACM Comput. Graph. Interact. Tech.* 8.1 (2025) 2.

[TYL24] TYLER WIEDERIEN, PETER-PIKE SLOAN. *A Tighter Quantization Bound for the Spherical Harmonic Projection of Non-Negative Functions*. Technical Memo ATVI-TR-24-02. 2024. URL: https://www.activision.com/cdn/research/Tighter_Quantization_Bound.pdf 10.

[WBB*07] WAND, MICHAEL, BERNER, ALEXANDER, BOKELOH, MARTIN, et al. "Interactive Editing of Large Point Clouds." *PBG@ Eurographics*. 2007, 37–45 2.

[WBL*24] WU, JING, BIAN, JIA-WANG, LI, XINGHUI, et al. "GaussCtrl: Multi-View Consistent Text-Driven 3D Gaussian Splatting Editing". *ECCV* (2024) 3.

[WFZ*24] WANG, JUNJIE, FANG, JIEMIN, ZHANG, XIAOPENG, et al. "GaussianEditor: Editing 3D Gaussians Delicately with Text Instructions". *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2024, 20902–20911 3.

[WHA*07] WEYRICH, TIM, HEINZLE, SIMON, AILA, TIMO, et al. "A hardware architecture for surface splatting". *ACM Transactions on Graphics (TOG)* 26.3 (2007), 90–es 2.

[Wym16] WYMAN, CHRIS. "Exploring and expanding the continuum of OIT algorithms". *Proceedings of High Performance Graphics*. HPG '16. Dublin, Ireland: Eurographics Association, 2016, 1–11. ISBN: 9783038680086 2.

© 2025 The Author(s).

[WYW*24] WANG, YUXUAN, YI, XUANYU, WU, ZIKE, et al. “View-Consistent 3D Editing with Gaussian Splatting”. *arXiv preprint arXiv:2403.11868* (2024) 3.

[XHL*25] XU, TIAN-XING, HU, WENBO, LAI, YU-KUN, et al. “Texture-GS: Disentangling the Geometry and Texture for 3D Gaussian Splatting Editing”. *Computer Vision – ECCV 2024*. Ed. by LEONARDIS, ALEŠ, RICCI, ELISA, ROTH, STEFAN, et al. Cham: Springer Nature Switzerland, 2025, 37–53. ISBN: 978-3-031-72698-9 3.

[YDYK25] YE, MINGQIAO, DANELLJAN, MARTIN, YU, FISHER, and KE, LEI. “Gaussian Grouping: Segment and Edit Anything in 3D Scenes”. *Computer Vision – ECCV 2024*. Ed. by LEONARDIS, ALEŠ, RICCI, ELISA, ROTH, STEFAN, et al. Cham: Springer Nature Switzerland, 2025, 162–179. ISBN: 978-3-031-73397-0 3.

[ZIE*18] ZHANG, RICHARD, ISOLA, PHILLIP, EFROS, ALEXEI A., et al. “The Unreasonable Effectiveness of Deep Features as a Perceptual Metric”. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018 9.

[ZKC*24] ZHUANG, JINGYU, KANG, DI, CAO, YAN-PEI, et al. “TIP-Editor: An Accurate 3D Editor Following Both Text-Prompts And Image-Prompts”. *ACM Trans. Graph.* 43.4 (July 2024). ISSN: 0730-0301. DOI: 10.1145/3658205. URL: https://doi.org/10.1145/3658205 3.

[ZPKG02] ZWICKER, MATTHIAS, PAULY, MARK, KNOLL, OLIVER, and GROSS, MARKUS. “Pointshop 3D: An interactive system for point-based surface editing”. *ACM Transactions on Graphics (TOG)* 21.3 (2002), 322–329 2.

[ZPvBG01] ZWICKER, MATTHIAS, PFISTER, HANSPETER, van BAAR, JEROEN, and GROSS, MARKUS. “EWA volume splatting”. *Proceedings of the Conference on Visualization ’01*. VIS ’01. San Diego, California: IEEE Computer Society, 2001, 29–36. ISBN: 078037200X 2, 5.


## Appendix A: Tight Ellipse AABB

To derive the tight ellipse AABB, we turn the two basis vectors $a, b \in \mathbb{R}^2$ into a symmetric $2 \times 2$-matrix $D := aa^\top + bb^\top$ using outer products. Then we consider the inverse matrix $C := D^{-1}$ such that the ellipse is given by the implicit equation

$$f(x,y) := (x,y)C(x,y)^\top - 1 = 0.$$

Points $(x,y)^\top$ on the vertical edge of its AABB are points on the ellipse, where the gradient $\nabla f$ is horizontal, i.e.:

$$\begin{aligned}
& f(x,y) = 0 \quad \wedge \quad \frac{\partial f}{\partial y}(x,y) = 0 \\
\Leftrightarrow\ & (x,y)C(x,y)^\top = 1 \quad \wedge \quad 2C_{0,1}x + 2C_{1,1}y = 0 \\
\Leftrightarrow\ & C_{0,0}x^2 + 2C_{0,1}xy + C_{1,1}y^2 = 1 \quad \wedge \quad y = -\frac{C_{0,1}}{C_{1,1}}x \\
\Rightarrow\ & C_{0,0}x^2 - 2\frac{C_{0,1}^2}{C_{1,1}}x^2 + \frac{C_{0,1}^2}{C_{1,1}}x^2 = 1 \\
\Leftrightarrow\ & x = \pm\sqrt{\frac{C_{1,1}}{C_{0,0}C_{1,1} - C_{0,1}^2}} = \pm\sqrt{\frac{C_{1,1}}{\det C}} = \pm\sqrt{D_{0,0}}
\end{aligned}$$

The same derivation with x/y and 0/1 swapped applies to the horizontal edge. Then our end result for the AABB bounds is

$$\begin{aligned}
x &= \pm\sqrt{D_{0,0}} = \pm\sqrt{a_0^2 + b_0^2}, \\
y &= \pm\sqrt{D_{1,1}} = \pm\sqrt{a_1^2 + b_1^2}.
\end{aligned}$$

## Appendix B: Ellipse Intersection Tests

```
bool intersection_point_splat(
  vec2 pos_point,
  vec2 pos_splat,
  vec2 a,
  vec2 b
){
  vec2 pos = pos_point - pos_splat;
  float sT = dot(normalize(a), pos) / length(a);
  float sB = dot(normalize(b), pos) / length(b);
  float w = sqrt(sT * sT + sB * sB);

  return w < 1.0f;
}
```

**Listing 1:** *Point-Ellipse Intersection Test in CUDA*

```
bool intersection_circle_splat(
  vec2 pos_circle,
  float r,
  vec2 pos_splat,
  vec2 a,
  vec2 b
){
  // add circle radius to length of basisvector.
  vec2 ar = (length(a) + r) * a / length(a);
  vec2 br = (length(b) + r) * b / length(b);

  return intersection_point_splat(pos_circle,
    pos_splat, ar, br);
}
```

**Listing 2:** *Approximate Circle-Ellipse Intersection Test in CUDA*

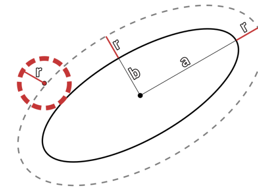



**Figure 13:** *Enlarged ellipse used in approximate intersection test.*

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.