DOI: 10.1111/cgf.70538
Eurographics Symposium on Rendering 2026
I. Gkioulekas and A. Jarabo
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 45 (2026), Number 4

CuRast: Cuda-Based Software Rasterization for Billions of Triangles

M. Schütz, L. Lipp, E. Kristmann, M. Wimmer

TU Wien

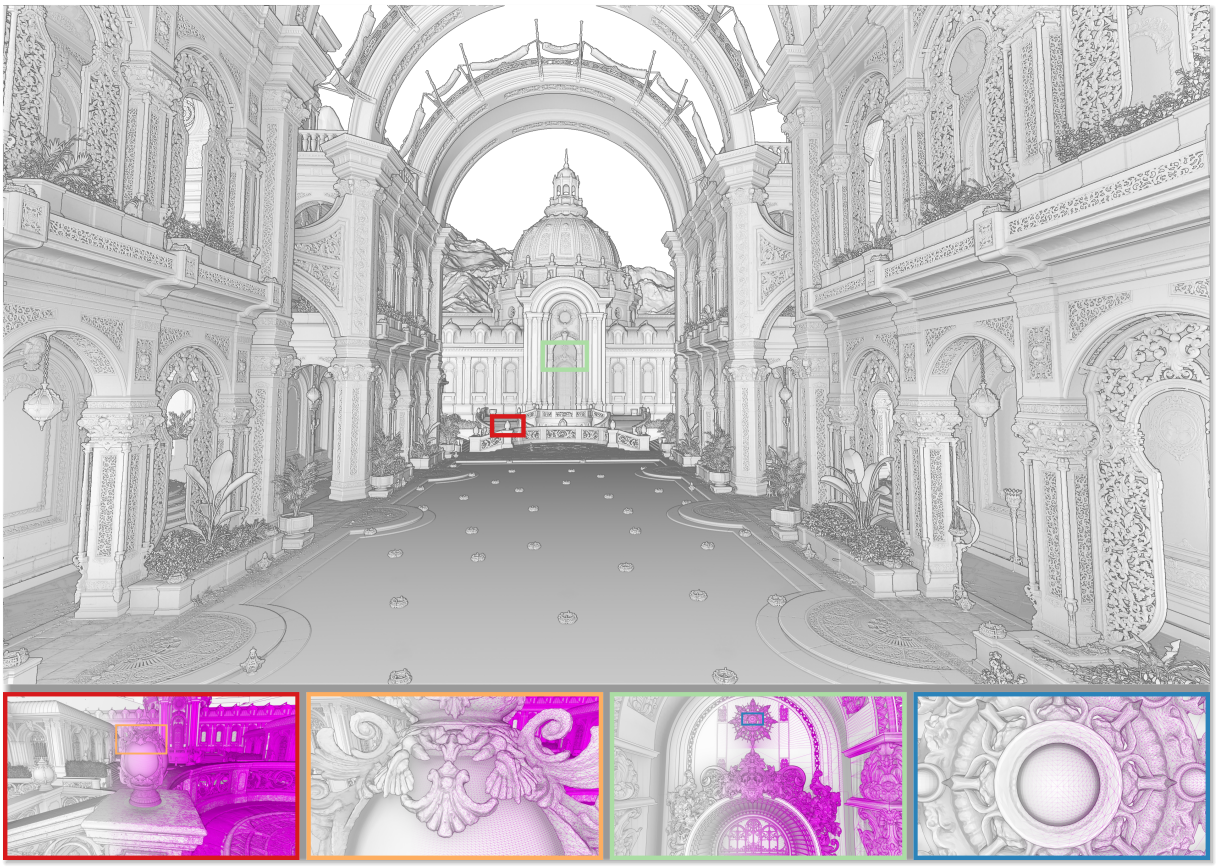


Figure 1: *Brute-force rendering the Zorah data set on an RTX 5090. 38.8GB of geometry loaded from an SSD, compressed to 21.7 GB, transferred to GPU and ready to render in 6.6 seconds. 18.9 billion triangles total; 13.5 billion triangles visible after frustum culling. Rendered in 67.3 milliseconds into a 3840×2160 framebuffer with screen space ambient occlusion and eye-dome lighting enabled.*

Abstract

This paper presents a CUDA-based software rasterizer capable of rendering up to a billion unique triangles, or up to 4 billion instanced triangles, in real time at 60 fps on an RTX 5090. By specifically targeting dense, opaque meshes, our approach is able to outperform the native GPU rasterization pipeline in these scenarios. The resulting performance enables rapid loading and visualization of massive triangle datasets without requiring precomputed spatial acceleration or level-of-detail structures, and supports applications such as efficient editing of large-scale geometry. While the method is primarily designed for dense meshes that generate pixel-sized triangles, we additionally introduce a three-stage pipeline to handle larger primitives.
The source code is available at: https://github.com/m-schuetz/CuRast

CCS Concepts
• ***Computing methodologies*** → ***Rasterization;***

1. Introduction

Dedicated hardware rendering pipelines have been the dominant approach for visualizing three-dimensional triangle meshes for decades. However, prior work on micropolygon and triangle cluster rendering, such as Nanite [KSW21], have demonstrated that hardware rasterization can be outperformed in certain scenarios, particularly when rendering highly detailed geometry composed of pixel-sized triangles. On average, Nanite's software rasterizer achieves roughly three times the throughput of the hardware pipeline, with even larger gains for pure micropolygon geometry. For larger triangles or triangle clusters, Nanite falls back to traditional hardware rasterization.

In this paper, we further explore the potential of GPGPU-accelerated software rasterization and investigate the limits achievable through brute-force processing of unstructured triangle meshes. By *unstructured*, we mean that no spatial acceleration structures or hierarchical levels of detail (LOD) are precomputed. We deliberately avoid such preprocessing for two reasons: first, constructing these structures requires a costly preprocessing step, and second, they typically must be recomputed whenever the underlying mesh is modified.

GPU-based software rasterizers typically also aim to support a wide range of features, including blending and support for transparent geometry. These features can be a defining aspect in the architecture of a rasterization pipeline and potentially slow down the processing of opaque geometry. In this work, we omit blending and focus on maximizing performance for opaque geometry. Transparent geometry could, in theory, be implemented in subsequent passes that execute after our pipeline.

Our contributions to the state of the art are as follows:

- A software rasterizer capable of rendering up to a billion (unique geometry) to 4 billion (instanced geometry) of triangles in real time, without the need to create spatial acceleration structures or hierarchical levels of detail in advance.
- A 3-stage pipeline that is highly optimized for small, but also handles medium and large triangles.
- Adapting visibility buffer indexing for an arbitrary mix of nodes/instances/triangles, rather than supporting a fixed number of nodes and a fixed maximum of triangles per node.

To manage expectations about the scope of this work, following limitations and observations apply:

- We specifically focus on dense and opaque geometry. Blending is not supported.
- Our implementation currently focuses on triangle-dense meshes and scales poorly with tens of thousands of low-density meshes.
- Likewise, instancing is targeted towards meshes with several hundreds of thousands to millions of triangles.
- Larger triangles are handled in a "good enough" fashion with glaring potential for improvement.

Given these benefits and limitations, we believe the results of this work are particularly relevant for applications involving dense and editable or animated meshes, as our approach does not rely on acceleration structures that require recomputation after modifications. Representative use cases include photogrammetry reconstruction and processing pipelines that generate and curate models containing tens of millions to billions of triangles. Additional examples include content creation tools such as Blender, where users perform arbitrary editing operations on complex geometry. It is not yet suitable for games as these work largely with non-editable geometry that benefits from pre-constructed LODs, and because they typically comprise scenes with tens of thousands of low-density meshes, which our implementation is currently not made for.

2. Related Work

2.1. Software Rasterization

Software rasterization has a long history that predates dedicated GPU hardware, and it continues to be of interest as modern GPUs provide powerful and fast compute pipelines for custom rendering algorithms. One of the most fundamental concepts of efficient rasterization pipelines – the depth buffer – dates all the way back to 1974, long before GPUs became commonplace [Cat74]. Since GPUs became widely available, software rasterization was primarily motivated by education and research, but in the recent decade the interest has shifted towards targeting specialized use cases in which custom programs can outperform the more generalized hardware pipeline that has to support a wide range of scenarios.

2.2. Triangles

FreePipe [LHLW10] introduces the concept of atomic-min to efficiently store the closest fragment inside pixels without the need for inter-thread communication and sorting stages. Since CUDA was limited to 32 bit atomics back then, which did not allow atomically writing depth and color values with a single atomic operation, they suggested performing two 32 bit atomic-min operations with the same 20 bit depth value but different 12 bits of the color value into the same pixel that is separated into two buffers. Afterwards, they extract the 12 bit parts located in different buffers, and fuse them back into a 24 bit color value. Brunhaver et al. [BFH10] design a hardware system targeted towards micropolygons that is able to process triangles at the rate of an GTX 480, but with less than 1% of the die size and power usage. CudaRaster [LK11] employs a hierarchical approach where triangles are first queued into bins, then tiles, and finally rasterized in pixels. They also structure their pipeline to honor the order of triangles during rendering via a sort-middle [MCEF94] approach, enabling order-dependent algorithms such as front-to-back blending operations of transparent triangles. Weber built a software rasterization approach for micropolygons, where they first lock a pixel sample, update the color value, then unlock it [Web15]. Kenzel et al. [KKSS18] propose cuRE, a streaming architecture that performs multiple rasterization stages in parallel rather than one after another. This approach avoids amassing large workloads in queues between stages, as the queues are quickly processed in a timely manner. They also aim for a rich set of features comparable to DirectX 9, and evaluate with a large set of more than 100 test scenes captured from video games, with about 0.3 to 8 million triangles. cuRE compares to Piko [PTSO15], FreePipe, CUDARaster and OpenGL, and finds that CUDARaster generally performs fastest within a factor of 4-6 to OpenGL, followed by cuRE. cuRE, on the other hand, scales better with multiple draw

calls and higher screen resolutions. FreePipe is slower than either for most scenes due to utilizing only one thread per triangle, which becomes an issue for scenes with large triangles.

2.3. Points and Splats

Software rasterization has seen particular success in the field of point-based rendering, which poses fewer challenges as points are typically always small, often pixel-sized. Due to this, we rarely run into load-balancing issues caused by processing different-sized primitives (with the notable exception of Gaussian Splats). Günther et al. [GKLR13] proposed a compute-based rendering pipeline based on spin loops: Each thread trying to draw a point to a pixel first attempts to lock it with atomic-CAS, then updates and subsequently unlocks it. Marrs et al. [MWH18] use 32 bit atomic-min operations to efficiently construct multiple depth maps. Schütz et al. [SKW21] use 64 bit atomic-min operations to efficiently write interleaved depth and color values into a framebuffer, then extract the color values for display on screen. Neural point-based renderers ADOP [RFS22] and NePO [LRSF] use atomic-min-based rasterization to quickly construct multi-resolution framebuffers that are then fed to a neural renderer that fills holes and shades the final image. One of the biggest success stories of software rasterization and point-based rendering might be 3D Gaussian Splatting (3DGS) [KKLD23]. 3DGS proposes using translucent gaussian primitives for scene reconstruction from photos, resulting in high-quality models of the real world. The 3-dimensional gaussians are then rendered by approximating them through 2-dimensional gaussians in screen space, sorting them by depth, and blending them from front to back. To efficiently process splats with wildly different sizes, they are partitioned into 16×16 pixel tiles, and they then rasterize all splats in a tile with one CUDA thread block. Since then, a plethora of research has proposed optimizations to various aspects of the training and rasterization pipeline. We refer to Hahlbohm et al. [HFEM26] as a recent summary of a variety of these optimizations.

Besides 3DGS, Dreams (PS4) [Eva15] and Nanite [KSW21] are two notable software rasterizers that demonstrate the practical use of custom rendering pipelines. The developers of Dreams discovered that they could rasterize small splats faster with a custom atomics-based renderer than the standard PS4 hardware pipeline. Nanite is primarily an LOD system for massive meshes that renders clusters of 128 triangles with a desired level of detail. However, they discovered that rasterizing these fine-grained clusters with near pixel-sized triangles is often faster through custom compute shaders, than through the hardware rasterizer. For large triangles, they fall back to hardware rasterization.

2.4. Visibility Buffers

Visibility buffers were initially introduced under the term *item buffers* in the context of ray tracing [WHG84]. For each pixel, they store the id of the element that corresponds to the first hit of a ray. They were later re-introduced and formalized for triangle-primitives under the term *visibility buffer*, as an alternative to deferred shading [BH13]. Instead of constructing feature-rich G-Buffers, visibility buffers store triangle and model/instance indices of the visible triangle for each pixel, which simplifies the rasterization stage and allows us to look up all necessary data in the shading stage. Alternatively, visibility buffers could also store references to shading values [SD15], but in the context of this paper, we assume they reference individual triangles. Visibility buffers have gained popularity for software rasterization as they allow us to address all attributes of a visible triangle, despite the limited amount of information we can write to a framebuffer with 64 bit atomic-min-based rasterizers.

2.5. Geometry Optimization

The order of vertices and indices has a substantial impact on rendering performance due to factors like access to coalesced data in memory and caching, as well as potential vertex reuse mechanisms in GPUs. Kenzel et al. [KKT*18] and Kerbl et al. [KKI*18] studied the behaviour of vertex caching and reuse on various modern GPUs, and propose strategies to optimize the meshes for better reuse, as well as strategies to implement reuse in software rasterization pipelines. Schütz et al. [SKW21] rearrange point clouds such that some spatially close points are also close in memory to promote coalesced memory access, but avoid too much locality that leads to contention when rendering thousands of overlapping points to the same pixel. Bene and Valasek [BV26] investigate various triangulation strategies for polygons to find those that render the fastest.

Meshoptimizer is an open source project that implements strategies that promote locality and vertex reuse [Kap26]. In this paper, we use it to obtain fair comparisons with a Vulkan-based renderer that greatly benefits from it, and also observe its impact on our own software rasterization pipeline.

3. Preliminaries

In this section, we recap important basics/prior work we build on in more detail, particularly atomic-min-based software rasterization and visibility buffer rendering pipelines.

3.1. Atomics-Based Depth Testing

In 2010, FreePipe [LHLW10] proposed using atomic min operations to efficiently write the closest depth value to each pixel in a massively parallel rasterizer. Because global atomics implicitly synchronize concurrent writes, this approach eliminates the need for inter-thread communication or fragment queuing and sorting. At the time, however, the method was limited to 32-bit atomics, which made it difficult to attach information in addition to the mandatory depth value. With the introduction of 64-bit atomic min/max operations, it became possible to attach additional payload such as color values or primitive IDs, enabling atomic updates to an interleaved depth+payload framebuffer:

```
u32 udepth = __float_as_uint(depth);
u32 payload = ...; // color or primitive ID
u64 fragment = u64(udepth) << 32 | u64(payload);
atomicMin(framebuffer[pixelID], fragment);
```

The PS4 game "Dreams" utilized that additional payload to create a highly efficient brush-stroke/splat based rasterizer that was already able to outperform the hardware rasterization pipeline of

the PlayStation 4 [Eva15]. While 64 bit atomics are fast, the still fairly small payload is not suitable for rendering algorithms that rely on multiple render targets, such as deferred rendering with rich G-Buffers. To work around this limitation, we can either (A) compute each fragment's final shaded color value directly during triangle rasterization, which is prohibitively expensive, or (B) we can store primitive indices instead to access all of the visible triangle's attributes in a deferred resolve pass. The second approach is what is commonly referred to as "visibility buffers" and popularized by Nanite [KSW21].

3.2. Visibility Buffers

Visibility buffer rendering and deferred shading with G-Buffers are closely related. Both approaches aim to rasterize all geometry first, and defer the shading to a full-screen resolve pass in order to apply expensive shading operations only to visible fragments. The difference is that deferred rendering creates multiple (or packed) render targets that contain attributes that are necessary for shading (e.g., albedo, normals, material id/flags, roughness, etc.), while visibility buffers only store the ID of the visible mesh and triangle in each pixel. With these IDs, we are then able to fetch all of the data that we need for shading in the full-screen resolve pass.

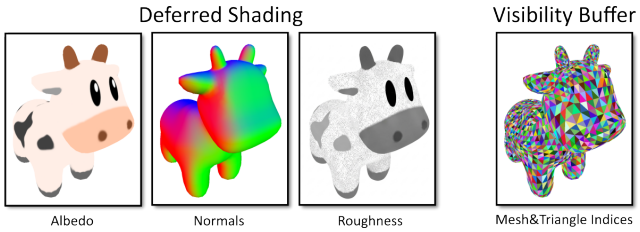


Figure 2: *Deferred shading creates G-Buffers with various attributes that may be needed for the shading pass. Visibility buffers only store triangle indices.*

Nanite [KSW21] demonstrated the efficiency and viability of software-rasterization with visibility buffers in a practical and widely deployed gaming engine. It assigns 30 bit for depth values and a payload of 34 bit for the triangle ID. The triangle ID itself is composed of 27 bit for the cluster index and 7 bit to address the triangle within the cluster. With this assignment of bits, they are able to address 134 million clusters and 128 triangles per cluster.

A major advantage of visibility buffers is that they significantly reduce memory bandwidth pressure during geometry rasterization, since we only need to load vertex indices and positions. Other attributes, such as uv-coordinates, normals, textures, etc., are not needed, unless they affect vertex positions (e.g. bump maps or height maps). This further accelerates the geometry processing stage compared to deferred rendering, which still fetches data for numerous fragments that will not be visible in the final image. Another advantage of visibility buffers is that we get object picking for free.

A disadvantage of visibility buffers is that the resolve pass is significantly more expensive: We now need to load the triangle geometry, project it again, compute interpolated uv coordinates, etc. Visibility buffers may also suffer from caching issues because if triangles are roughly pixel-sized, adjacent pixels load vertex data from different and quasi-random regions in memory [Hab21]. Another disadvantage of visibility buffers is that computing the mip map level can be tricky for large triangles that intersect with the near plane. If one or two vertices are behind the near plane, we can not inter- or extrapolate uv-coordinates in adjacent pixels in screen space, which would allow us to cheaply compute the pixel's footprint in texture space. We therefore shade triangles in world space (see Section 4.4).

4. Method

Our method draws massive triangle models entirely in CUDA and supports following functionality: Indexed meshes, instancing, textures or vertex colors, mip mapping, and massive amounts of small but also large triangles. We furthermore experiment with compressed/quantized indices between 8 to 32 bit and 16-bit fixed-precision coordinates.

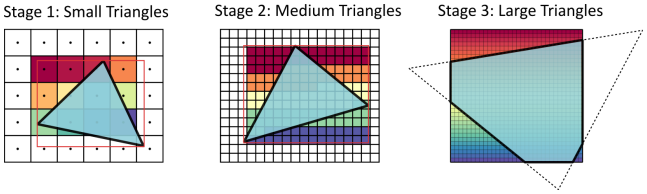


Figure 3: *Rasterization Stages. Pixels colored by iteration counter.* ***Stage 1****: A single thread iterates over all sample positions inside the triangle's bounding box.* ***Stage 2****: A warp (32 threads) iterates over all samples inside the bounding box.* ***Stage 3****: A workgroup (64 threads) iterates over all samples in a 64x64 tile to rasterize a portion of the triangle.*

As we primarily focus on massive models with hundreds of millions to billions of small triangles, our method first launches a CUDA kernel with one thread per triangle, under the assumption that triangles can easily be rasterized by a single thread, and multiple threads in a warp have balanced workload as they process similar-sized triangles. This assumption will, of course, be frequently violated so we implement two additional stages that handle medium-sized and large triangles:

- **Stage 1**: Launch 1 thread per triangle, rasterizing it directly if it is small, or adding it to a global queue for the next stage if it is either too large or intersects the near plane.
- **Stage 2**: Launch a warp (32 threads) per triangle that was queued by stage 1, rasterizing it directly if it is medium-sized, or splitting it into parts and adding each part to a queue if it is either very large or intersects the near plane.
- **Stage 3**: Launch 64 threads for each part of a triangle that was queued by stage 2.

Small triangles were experimentally determined as those whose screen-space bounding box covers less than 128 pixels, and *medium* triangles as those covering less than 4096 pixels.

On the host side, before launching the rasterization stages, we first perform frustum culling and assemble a list of meshes, each comprising transformation matrices and pointers to vertex and index

buffers. Crucially, we also compute a cumulative triangle count for each mesh and instance (i.e. the exclusive prefix sum), which we will use to store the absolute triangle ID over all visible meshes in the visibility buffer. This data is then copied to the GPU, and we are ready to launch the 3 stages of our rasterization kernels.

Experienced readers will notice that we perform naïve boundary constraints by processing all fragments inside the triangle's bounding box, despite well-known approaches that only traverse fragments within the triangle [Pin88]. We initially tried to do so but found that the additional effort that was required to avoid wasted work is often more expensive than unnecessary but cheap work, especially for massive, dense data sets that produce pixel-sized triangles.

4.1. Stage 1: Rasterizing Small Triangles

The premise of this stage is that each triangle is small enough such that a single thread can render it in a timely manner. If they are too large, we instead put them in a global queue for the next stage.

Looping Through Batches of 256 Triangles: We perform a persistent-kernel launch using cooperative groups since we are rendering numerous meshes with varying numbers of triangles, and therefore cannot easily map a global thread index to the corresponding triangle. A limited set of workgroups with 256 threads keeps looping until all triangles are rendered. Each specific workgroup obtains the next set of triangles it renders by atomically incrementing a global work counter: *atomicAdd(numTrianglesRendered, 256)*. If the returned triangle index is outside of the current mesh, the workgroup linearly advances through the list of meshes until it encounters the mesh that contains the corresponding triangles.

Preparing a Batch of 256 Triangles for Rasterization: Each thread loads the vertex data of its triangle and transforms it to view space. Unlike classical graphics APIs, we do not use a projection matrix. Instead, we perform perspective projection through an element-wise multiplication of the view-space coordinates with following projection-vector:

$$f = \frac{1}{\tan\left(\frac{\text{fovy}}{2}\right)} \tag{1}$$

$$\mathbf{P} = \begin{bmatrix} \frac{f}{\text{aspect}} & f & -1 \end{bmatrix} \tag{2}$$

Afterwards, the x and y components are divided by the depth (z component). The result is a normalized-device-coordinate with x and y between -1 and 1, and z storing the positive, linear and non-normalized depth value, starting from the camera position.

We continue to perform frustum culling, discarding/skipping any triangle that is entirely outside the visible volume. For the remaining triangles, we compute the screen-space bounding box, or mark it as *nontrivial* if it intersects the near plane. The latter is done because triangles that intersect the near plane require clipping to obtain accurate screen-space bounding boxes, but we want to keep this stage as simple as possible and avoid increased register pressure and expensive branches, which would slow down the processing of nicely behaved triangles. To optimize for scenarios with massive amounts of micro-polys, we also discard/skip triangles whose bounding box does not intersect with any sample position (pixel center or super sampling location), followed by performing backface culling. Finally, we put all triangles that either intersect with the near plane or whose bounding box covers more than 128 pixels into the queue for the next stage. The remaining triangles are then rasterized.

Rasterization: Rasterization follows a straight-forward bounding-box-rasterization scheme. Each thread loops over the y (outer loop) and x (inner loop) coordinates inside the bounding box, computes the barycentric coordinates of the triangle, and checks if the barycentric coordinates lie within the triangle boundary. If the fragment lies inside the triangle, we continue to perform a 64-bit atomicMin to update the framebuffer's pixel with a combination of depth (most significant bits) and global triangle index (least significant bits). To support scenes with tens of billions of triangles, we assign 28 bit to the depth value, and 36 bit to the global triangle ID. Since depth values are always positive, one bit is taken from the sign, and three bits are taken away from the mantissa. As the 28 bit depth value is stored in the most significant bits, the fragment with the smallest depth will prevail in the resulting framebuffer. The encoded global triangle index is later used in a resolve/screen pass in order to find and process the triangle that occupies that pixel.

For efficient perspective-correct interpolation of depth values, we precompute the inverse depth $\frac{1}{z}$ for each vertex outside of the loop, then interpolate them according to the barycentric coordinates s,t,v and a fast division intrinsic inside the loop:

```
float depthI = v * z0I + s * z1I + t * z2I;
float depth = __fdividef(1.0f, depthI);
```

Of note is that __fdividef is not just a faster approximate division; it also significantly reduces the kernel's register usage from 55 to 48, and therefore improves its occupancy.

For barycentric coordinates, we precompute their changes across the x and y directions outside of the loop, and then increment them as we loop over the pixels of the bounding box.

4.2. Stage 2: Rasterizing Medium Triangles

We now launch 32 threads per triangle that was queued by stage 1, so that we can process these more demanding ones with additional compute power. For any triangle whose bounding box covers up to 4096 pixels, the 32 threads perform a 1-dimensional loop over the number of pixels, incrementing the loop counter by 32 each iteration. The 1D index is then mapped to the 2D coordinate within the bounding box via $x = i \bmod \text{width}$ and $y = \frac{i}{\text{width}}$, but otherwise the rasterization logic remains the same as in stage 1.

A major difference from the first stage is that we now also perform an accurate and expensive screen-space bounding box calculation for each triangle by clipping them via the Sutherland-Hodgman [SH74] algorithm. However, we do not actually split triangles that intersect the frustum into multiple smaller ones. Our intention is to (1) identify and measure huge triangles (which often intersect the near plane), so that we can partition them into a suitable number of tiles for stage 3, and (2) to obtain the bounding box of the visible part of the triangle, which may be a fraction of the bounding box of the entire triangle.

Triangles that are larger than 4096 pixels are split into smaller parts according to 64×64 pixel tile boundaries. For each part, we add one entry into a global queue comprising the triangle ID, mesh ID, and the tile coordinate.

4.3. Stage 3: Rasterizing Large Triangles

For each part of a triangle that was queued in stage 2, we launch one workgroup comprising 64 threads. Each workgroup then processes the pixels inside the 64 x 64 tile line-by-line. For each processed pixel, the corresponding thread performs a ray cast in view space in order to determine whether the triangle is hit, and at which location. Depth and total triangle index are then stored inside the pixel using a 64-bit atomicMin.

The reason we render triangles in this stage with ray-triangle intersections in view space is because in addition to large triangles, stages 1 and 2 also forwarded triangles that intersect the view plane at $z = 0$. The screen-space approach with division by the depth as used in stages 1 and 2 would require us to clip these triangles, while ray-triangle intersections allow us to avoid clipping and operate on the original triangle.

4.4. Screen-Space Resolve/Shading Pass

In this pass, we texture and shade the triangles whose global indices are stored in the pixels of the visibility buffer. We launch a 2-dimensional CUDA kernel with one thread per pixel. For each pixel, we decode depth (28 bit) and global triangle index (36 bit).

One of the challenges we faced was how these 36 bit of the visibility buffer would be able to address individual triangles in scenes with tens of billions of triangles, tens of thousands of meshes, and up to tens of millions of triangles per mesh. Traditionally, visibility buffers assign a fixed number of bits to mesh indices and triangle indices, so we can either increase the number of supported meshes, or the number of triangles per mesh. This was not feasible for Zorah, which required much of both. So instead of storing a combination of mesh and triangle index inside the visibility buffer, we store the cumulative triangle index and now perform a binary search inside the resolve pass to find the mesh that this triangle belongs to. A 64-bit integer array storing the exclusive prefix sum of triangle counts accelerates the search for the correct mesh's index.

Shading and Mip Mapping: Having identified the mesh that corresponds to the global triangle index, we proceed to load the triangle's vertex data and perform shading in world space. This choice is primarily motivated by the need to correctly determine the mipmap level for large triangles intersecting the near plane, for which we can not easily compute deltas to adjacent pixels in screen-space. Estimating a suitable mipmap level requires computing the pixel footprint in texture space. To achieve this, we cast rays from the camera through the current pixel as well as its right, top, and top-right pixels, and intersect these rays with the triangle's plane, as shown in Figure 4.

Using the resulting world-space intersection points together with the triangle's vertex positions, we compute barycentric coordinates for each intersection. These are then used to interpolate or extrapolate the corresponding UV coordinates. Note that some intersections, and thus their UV coordinates, may lie outside the triangle, which is acceptable for mipmap level estimation. Once the UV coordinates of all four samples are obtained, we compute their extent (delta) in texture space and derive the appropriate mipmap level.

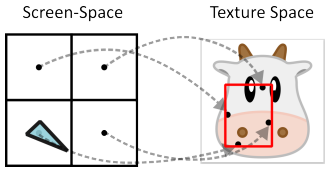


Figure 4: *Finding an appropriate mip map level by intersecting the triangle's plane at the current and three adjacent pixels, extrapolating uv coordinates, and computing the extent in texture space.*

4.5. Instancing

Memory bandwidth is one of the major bottlenecks during rasterization, particularly when rendering massive amounts of small triangles that require fewer compute resources. Instancing allows us to reduce this bandwidth bottleneck because instead of loading and rasterizing a set of triangles several times, we only need to load it once to rasterize it several times.

We adapt stage 1 such that each workgroup loads indices and position of a triangle once, then loops through all instances and draws the triangle with the corresponding transformation matrices. Stages 2 and 3 remain unchanged because most instances of a triangle are already likely to be finished by stage 1, so the kernel-overhead of handling instancing does not benefit the later stages. Unfortunately, this overhead does affect the stage 1 kernel, i.e., it increases the amount of required registers and thus reduces occupancy. We therefore suggest compiling and using two variations of the stage 1 kernel, one without and one with instancing.

4.6. Compression

Massive data sets such as Zorah [NVI25b; NVI25a] do not fit into GPU memory and thus need to be streamed or compressed. In order to fit the original 38.8GB data set into the 24GB VRAM of an RTX 4090, we perform following two compression schemes.

Compressing Indices: While loading the indices of a mesh, we compute the minimum and maximum index and the resulting $span = \max - \min$. The number of bits required to store indices in that span is given by the ceiling of the base-2 logarithm:

$$bitsPerIndex = \lceil \log_2(span) \rceil \tag{3}$$

For example, if an index buffer comprises index values between 2500 and 3000, we must represent 500 distinct values, which can be encoded in 9 bits.

Compressing Positions: For positions, we encode them as 16-bit fixed-precision integers. 16 bits afford 65 536 distinct values, meaning a mesh spanning 65 meters retains a vertex precision of 1 mm. The conversion to fixed-precision coordinates relative to the mesh's bounding box is:

```
uint16_t X = 65536.0 * (pos.x - min.x) / size.x;
```

With these two lightweight compression schemes, and by dropping the 2 GB of UV coordinates which only exist for a fraction of meshes, we reduced the Zorah data set from 38.8 GB to 21.7 GB, sufficiently small to fit on an RTX 4090. Similar to instancing, we found that compiling separate CUDA kernels for uncompressed and compressed data slightly improves performance for the uncompressed case. The presence of a branch inside the kernel that checks if data is compressed would otherwise add up to 1-2% to the runtime in scenes with billions of triangles.

5. Evaluation

The performance of CuRast is compared to a Vulkan-based rendering path, using data sets and configurations as described in Table 1 and Section 5.1. We focus mainly on large and dense data sets – the target of our rasterization approach – but include Sponza as a widely known reference scene.

Benchmarks were captured on following test systems:

- **4070**: CPU: AMD Ryzen 7 5800X; GPU: RTX 4070 12GB
- **4090**: CPU: AMD Ryzen 9 7950X; GPU: RTX 4090 24GB
- **5090**: CPU: AMD Ryzen 9 7950X; GPU: RTX 5090 32GB

We compare to two Vulkan-based render paths:

- **VK-ID** (Indexed Draw): Performs one *vkCmdDrawIndexed* per unique mesh, drawing all of its instances. Uses vertex buffers and index buffers.
- **VK-PIP** (Programmable Index Pulling): Performs one *vkCmdDraw* per unique mesh, drawing all of its instances. Instead of vertex and index buffers, this approach uses programmable index pulling and vertex pulling. This approach is needed to draw meshes whose index buffers are compressed to arbitrary bit rates other than 16 or 32 bit.

We omitted a visibility-buffer-based Vulkan pipeline as we observed no difference between these two approaches in our test data sets, except for being slightly slower due to an additional resolve pass. I.e., omitting triangle attributes other than position and indices did not significantly accelerate the geometry rasterization pass in our scenarios.

During benchmarking of the CUDA and Vulkan approaches, we let each allocate buffers through their own API. Reported timings are the mean over 60 frames.

5.1. Data Sets and Test Configurations

Table 1 depicts the test data sets we use for evaluation. Aside from the amount of geometry, these data sets present a range of challenges that stress different aspects of triangle rasterization pipelines.

Sponza [DMC*26] comprises 103 low-density meshes that produce large visible triangles. Our rasterizer is not targeted towards this scenario, but it is included due to being a widely used reference scene.

Lantern is a photogrammetry reconstruction that was simplified to about 1 million triangles. It also serves as a case study for creating thousands of instances (50×60) of a moderately detailed mesh, resulting in a scene with 3 billion triangles.

Komainu Kobe [Gil20] is a photogrammetry reconstruction of one of two lion statues in Kobe / Ikuta-jinja. After a high-resolution reconstruction in Reality Scan, resulting in 284 million triangles, the right Komainu was isolated and simplified to 60 million triangles for use in this evaluation.

Venice [ItF26] features twelve 16k JPEG textures (3.2 billion pixels, 909MB compressed) that, after decompression and mip map construction, did not fit in GPU memory alongside the geometry. Because the textures are distributed in JPEG format, we decided to keep them in JPEG format for rendering [KWS25]. A lightweight indexing table is constructed so that we are able to randomly access and decode visible JPEG blocks during the resolve pass. Since our Vulkan render path does not support JPEG textures, we also evaluate a scenario with downscaled textures to be able to compare CUDA and Vulkan performances.

Zorah [NVI25b] is our largest stress-test, featuring a massive 18.9 billion triangles in total, of which 1.6 billion are unique. We use the original v1 release, which has since been replaced by a compressed v2 by NVIDIA. The original release takes up 38.8GB disk space – 19.7GB for indices, 17.1GB for vertex coordinates, and 2GB for a partial set of uv-coordinates. Since this model has no textures and because most meshes are without uv-coordinates, we ignore the 2GB of uvs. To fit the data set on an RTX 4090, we apply the compression scheme as outlined in Section 4.5. After compression, index and vertex buffers require 13.1GB and 8.6GB of memory. We filter out a couple of billboards for aesthetic reasons, particularly those labeled "FogCard" and "Plane".

To ensure a fair comparison between our software rasterizer and Vulkan indexed draw calls, we also apply meshoptimizer/gltfpack [Kap26] to some scenarios. Meshoptimizer rearranges vertices and triangles to improve locality and helps GPUs to reuse vertices.

Since indexed draw calls in Vulkan only support 16 or 32 bit precision index buffers, test scenarios with compression are only evaluated in CUDA and VK-PIP, but not in VK-ID. VK-PIP supports compressed indices by loading and decoding them inside the vertex shader.

CUDA timings are measured with CUDA events, and Vulkan timings are measured via timestamp queries. CUDA timings in Table 2 cover stages 1, 2, and 3, and the resolve pass. Vulkan timings cover the vkCmdDraw and vkCmdDrawIndexed commands.

5.2. Results

Table 2 shows the benchmarking results of various scenarios on three GPUs. Figure 5 illustrates the density of triangles and the stages used to rasterize them. In Sponza, Vulkan indexed draws are 7 to 13 times faster than our approach. The Lantern scene shortens that gap to a factor of about 4. For the remaining scenes, CuRast performs significantly faster than Vulkan. The biggest increases are seen in scenes with numerous instances (Lantern Instances, Zorah), where CuRast only loads triangle geometry once to rasterize all instances of these triangles. CUDA also performs multiple times faster than Vulkan indexed draws in scenarios without meshoptimized geometry, although that gap to indexed draws shortens to a factor of 2× after meshoptimization.

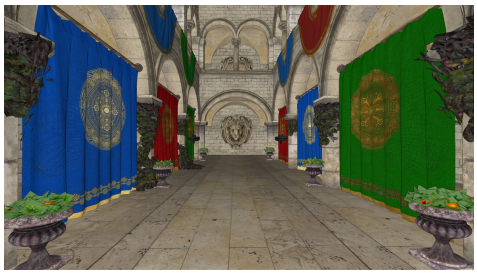
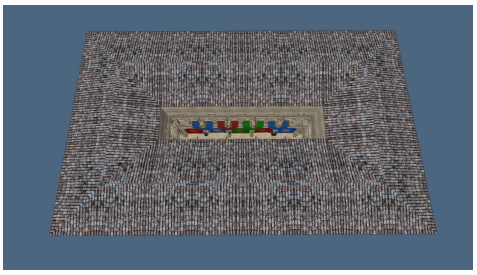
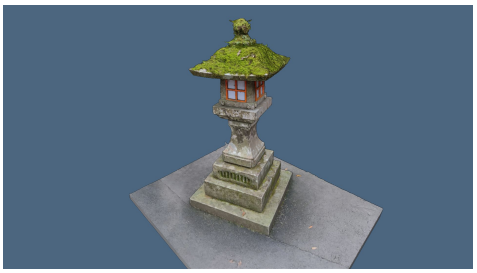
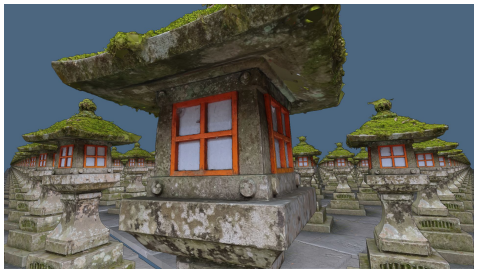
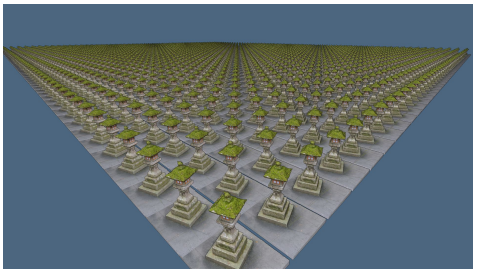
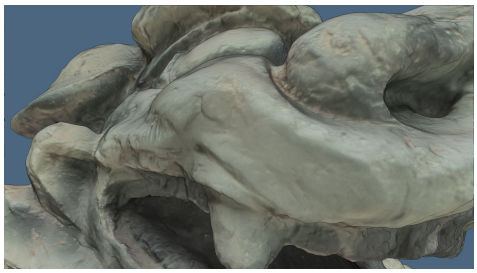
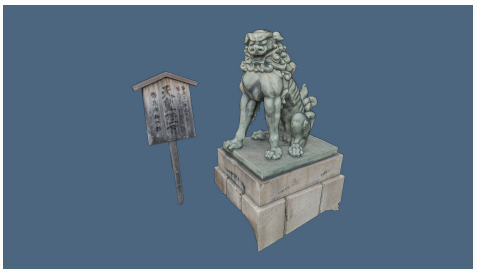
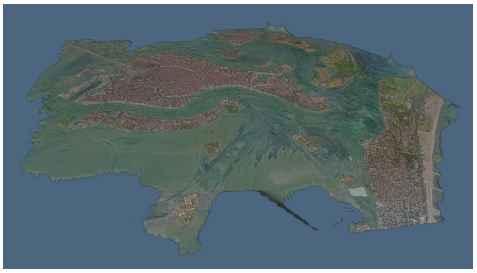
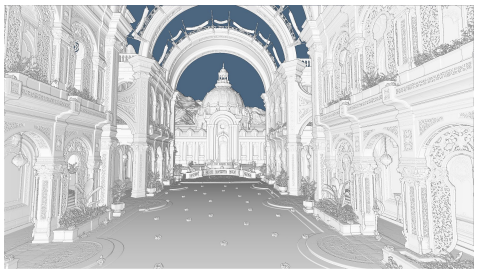
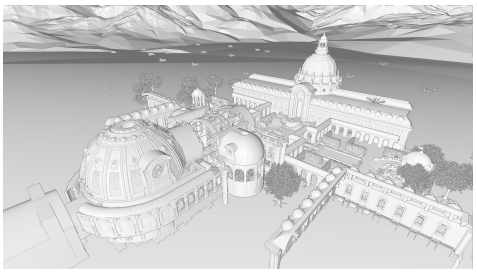
View 1 (Close-up)	View 2 (Overview)	Description	
		Name	Sponza
		Unique triangles	262k
		Triangles	262k
		Unique meshes	103
		Total meshes	103
		Textures	64 x 1k x 1k 67M Pixels
		Name	Lantern
		Unique triangles	1 million
		Triangles	1 million
		Unique meshes	1
		Total meshes	1
		Textures	1 x 8k x 8k 67M Pixels
		Name	Lantern Instanced
		Unique triangles	1 million
		Triangles	3 billion
		Unique meshes	1
		Total meshes	3000
		Textures	1 x 8k x 8k 67M Pixels
		Name	Komainu Kobe
		Unique triangles	60 million
		Triangles	60 million
		Unique meshes	9
		Total meshes	9
		Textures	9 x 8k x 8k 604M Pixels
		Name	Venice
		Unique triangles	400 million
		Triangles	400 million
		Unique meshes	12
		Total meshes	12
		Textures	12 x 16k x 16k 3.2B Pixels
		Name	Zorah
		Unique triangles	1.6 billion
		Triangles	18.9 billion
		Unique meshes	3880
		Total meshes	26170
		Textures	None

Table 1: *Test data sets. We evaluated scenes ranging from 260k to 18.9 billion triangles, each captured from a close-up and an overview viewpoint.*

Table 2: *Benchmark timings in milliseconds.* ***c****: Compressed coordinates and index buffers.* ***m****: Meshes optimized with meshoptimizer/gltfpack [Kap26].* ***j****: JPEG-compressed textures.* ***h****: Resized texture to half resolution.* **–***: Out of memory.* ×*: Unsupported test configuration. E.g., Vulkan indexed draws do not support compressed index buffers, and JPEG-compressed textures are only supported in our CUDA rasterizer.*

	Scene	vis. tris		RTX 4070			RTX 4090			RTX 5090		
				CuRast	VK-ID	VK-PIP	CuRast	VK-ID	VK-PIP	CuRast	VK-ID	VK-PIP
closeup	Sponza	197.0k		0.580	0.068	0.071	0.271	0.040	0.052	0.251	0.022	0.025
	Lantern	1.0M	m	0.460	0.121	0.140	0.208	0.056	0.094	0.162	0.042	0.049
	Lantern Instances	3.1B	m	41.351	241.833	367.267	17.325	142.001	282.385	9.951	125.677	142.147
	Komainu Kobe	28.7M		1.855	3.975	3.627	0.865	2.791	2.678	0.644	2.413	1.365
	Komainu Kobe	28.7M	m	1.656	1.524	3.466	0.825	1.380	2.640	0.544	1.220	1.323
	Venice	399.9M	c m h	–	–	–	6.929	×	37.571	4.681	×	18.892
	Venice	399.9M	c h	–	–	–	10.126	×	37.763	7.288	×	19.048
	Venice	399.9M	j	–	–	–	10.732	×	×	7.692	×	×
	Venice	399.9M	m j	–	–	–	9.548	×	×	5.242	×	×
	Venice	399.9M	m h	–	–	–	9.570	19.710	37.575	5.252	17.106	18.888
	Venice	399.9M	h	–	–	–	10.724	49.616	37.668	8.051	42.387	19.333
	Zorah	13.6B	c	–	–	–	74.906	×	1778.303	57.569	×	633.081
	Zorah	13.6B	c m	–	–	–	69.764	×	1250.210	53.395	×	631.095
overview	Sponza	262.3k		0.439	0.042	0.065	0.224	0.029	0.052	0.239	0.018	0.022
	Lantern	1.0M	m	0.358	0.113	0.130	0.157	0.056	0.094	0.128	0.042	0.050
	Lantern Instances	3.1B	m	36.550	245.304	367.131	15.516	141.747	282.331	9.614	125.680	142.131
	Komainu Kobe	60.0M		3.376	8.338	7.482	1.568	6.100	5.870	1.134	5.026	2.800
	Komainu Kobe	60.0M	m	2.963	5.158	7.293	1.456	2.881	5.863	0.932	2.560	2.766
	Venice	399.9M	c m h	–	–	–	7.271	×	37.571	4.985	×	18.890
	Venice	399.9M	c h	–	–	–	9.955	×	37.516	7.258	×	19.018
	Venice	399.9M	j	–	–	–	10.948	×	×	7.757	×	×
	Venice	399.9M	m j	–	–	–	9.759	×	×	5.377	×	×
	Venice	399.9M	m h	–	–	–	9.766	19.690	37.580	5.380	17.103	18.882
	Venice	399.9M	h	–	–	–	10.946	49.596	37.595	7.789	42.344	19.097
	Zorah	18.8B	c	–	–	–	99.698	×	2388.751	76.532	×	873.428
	Zorah	18.8B	c m	–	–	–	93.043	×	1729.774	70.667	×	872.477

Summary of notable observations based on the benchmark results:

- CuRast thrives with large models and numerous instances of large models.
- Meshoptimization has the biggest impact on Vulkan instanced draw calls, often doubling the performance when rendering dense meshes. The impact on CuRast or Vulkan programmable index pulling is significant but comparatively modest.
- The RTX 5090 shows major improvements for programmable index pulling, making it nearly twice as fast in all cases compared to the RTX 4090. Improvements for indexed draws have been smaller. This makes index pulling significantly faster than indexed draw in some non-meshoptimized scenarios. After mesh optimization, indexed draws are always faster than index pulling, but the difference is now small.

5.2.1. Performance of Individual Stages

Table 3 breaks down the timings for the individual stages, measured on an RTX 4090.

5.2.2. Culling tiny triangles

In massively dense scenes, a large number of triangles end up in between pixel samples without intersecting them. Culling these early affects the performance of the stage 1 kernel as shown in Table 4. Our takeaway here is that it substantially benefits the most massive of data sets, without negatively affecting small data sets that do not benefit from this optimization.

Table 3: *Timings (ms) of individual rasterization stages and resolve on an RTX 4090.*

		Stage 1	Stage 2	Stage 3	Resolve
Sponza (closeup)		0.048	0.054	0.063	0.098
Venice (closeup)	h	10.162	0.033	0.006	0.517
Venice (overview)	h	10.186	0.013	0.007	0.734
Zorah (closeup)	c	74.109	0.193	0.183	0.416
Zorah (overview)	c	99.067	0.130	0.210	0.254

Table 4: *Early-culling triangles that do not intersect a sample.*

		enabled	disabled
Sponza (closeup)		0.048 ms	0.048 ms
Venice (overview)	h	10.1 ms	10.2 ms
Zorah (closeup)	c	74.1 ms	102.5 ms
Zorah (overview)	c	99.1 ms	140.2 ms

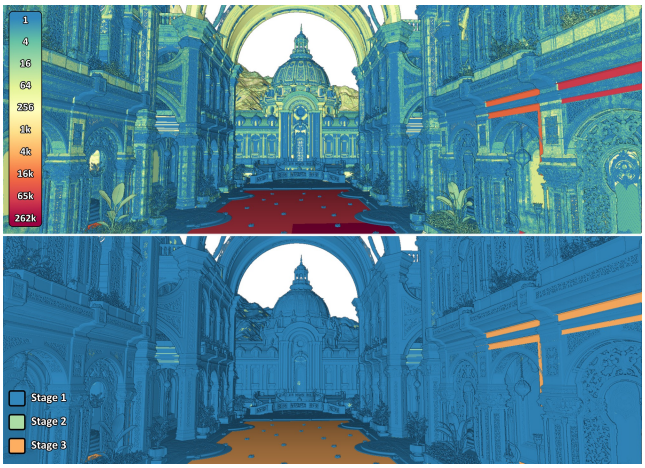


Figure 5: *Top: Colored by size (px) of the bounding box of a pixel's triangle. Bottom: Pixels colored by the rasterization stage that created the fragment. In Zorah, the vast majority of triangles are rasterized in stage 1.*

5.3. Anti-Aliasing (Super Sampling)

We support anti-aliasing via super sampling, which affects the performance as shown in Table 5. Rendering times for Sponza increase linearly with the sample count because it is already bottle-necked by compute, and super sampling multiplies the compute work load proportionally. The larger data sets, on the other hand, scale sub-linearly because they are mainly bottle-necked by memory bandwidth. Increasing the samples gradually moves that bottleneck from memory to compute as we essentially increase the size of the framebuffer, and thus the size and fragment count of each triangle.

Table 5: *Super Sampling performance (ms) on an RTX 5090.*

		No SS	2x2 SS	4x4 SS
Sponza (closeup)		0.251	0.847	2.968
Venice (overview)	h	7.789	8.845	12.261
Zorah (closeup)	c	57.569	64.913	84.008

6. Discussion

6.1. Watertighness and the Top-Left Rule

While we have not observed any issues in our tests, our implementation does not currently guarantee watertightness. Numerical precision errors could lead to situations where a pixel sample located on the shared edge of two adjacent triangles is rejected by both, resulting in visible cracks between triangles. Such issues may be particularly plausible when transitioning from screen-space barycentric rasterization in Stages 1 and 2 to view-space ray–triangle intersections in Stage 3, as the use of different coordinate spaces and algorithms introduces differing precision characteristics.

Conversely, instead of neither triangle processing a shared sample, both triangles may process it. In this case, the final result of a rasterizer could depend on the order in which the triangles are processed, potentially causing inconsistencies between frames, or even multiple render targets. The commonly used top-left [Mic20b] rule resolves this ambiguity by assigning ownership of a shared sample to the triangle with the top-left edge, thereby ensuring a deterministic outcome independent of processing order.

Atomic-min-based rasterizers achieve deterministic behavior without relying on the top-left rule. If a sample on a shared edge is processed by both triangles and produces the same depth value, the triangle with the lower triangle ID implicitly takes ownership because the ID is part of the atomic comparison.

6.2. Mesh Shaders

In this paper we compared render times to the traditional triangle rasterization pipeline based on vertex shaders. In recent years, mesh shaders emerged as an alternative that address several limitations and inefficiencies of vertex shaders through a programming model that more closely resembles compute shaders [Kub18; OKM23]. Rather than following the one-thread-per-vertex paradigm of vertex shaders, mesh shaders operate on groups of threads that cooperatively process batches of vertices and triangles, commonly referred to as meshlets. One advantage of mesh shaders is that they make vertex reuse an inherent part of meshlets. In traditional graphics pipelines, vertex reuse is handled by the fixed-function input assembler, which must dynamically analyze the index buffer at runtime to identify vertices shared by multiple triangles, whose vertex-shader results could be reused. With mesh shaders, reuse within a meshlet is an inherent part of the system. A meshlet containing 64 vertices requires exactly 64 vertex transformations.

The gl_vk_meshlet_cadscene [NVI18] repository demonstrates that mesh shaders can perform twice as fast as vertex shaders on our RTX 4090. The demo's vertex shader pipeline achieves a triangle throughput of about 20 million triangles per millisecond, which matches the best-case results for VK-ID on an RTX 4090 in Table 2. The mesh shader renderer increases this throughput to about 40 million triangles per millisecond. However, mesh shaders rely on meshlets, an optimized geometry representation that must first be precomputed, and which may also benefit CuRast. This will be explored in future work.

Instancing has shown to be a particularly advantageous situation for CuRast compared to the traditional vertex shader pipeline. A set of triangles can be loaded into shared memory once and then reused to rasterize multiple instances, significantly reducing global memory traffic. The Meshlet Instancing sample in DirectX Graphics Samples [Mic20a] suggests that mesh shaders can exploit a similar strategy to some extent. A mesh shader workgroup can load a single meshlet and emit multiple instances, for example five, depending on the meshlet size and the maximum number of vertices and triangles that the mesh shader is permitted to emit. However, mesh shaders require additional shared memory to store per-instance data, whereas CuRast draws any number of instances directly without allocating additional shared memory. Nevertheless, even the ability to emit only two instances from a single meshlet effectively halves the required global memory fetches, potentially removing memory bandwidth as the primary performance bottleneck.

7. Conclusion and Future Work

In this paper, we have shown that a software rasterization pipeline specifically targeted towards dense, opaque triangle meshes can outperform the native hardware rasterization pipeline by factors of 2-12. CuRast mainly benefits from scenarios such as fewer meshes with large triangle counts producing pixel-sized triangles; instancing of dense meshes; and, to a smaller extent, from geometry compression. Vulkan, on the other hand, remains the champion when it comes to dealing with numerous small meshes.

Based on the results, we believe that software rasterization is at a stage where it is already a viable option for use cases such as reconstruction of massive photogrammetry data sets with users in the loop for manual editing, or generally editing large, dense meshes.

For more general graphics applications, and games in particular, future work will need to improve performance for large numbers of small meshes and add support for features like blending and transparency. Given the success of 3DGS, the latter should be easily doable in CUDA. We also believe that past and ongoing research on clustered approaches and/or hierarchical LODs [KSW21; KKG*26; Kap26] is the way forward for games, and integrating these approaches into CuRast is likely to result in major improvements in rendering performance. Nanite, taking advantage of clustered geometry, also implements an interesting mechanism akin to vertex-reuse. Instead of simply launching one thread per triangle and independently processing potentially shared vertices multiple times as we do, they first process vertices in a cluster using one thread per vertex, store the results in shared memory, and then process the triangles using one thread per triangle [Kar26]. Kuth et al. [KOK*25] propose a meshlet compression approach that can reduce index buffer sizes down to 9 bit per triangle, thus reducing pressure on memory bandwidth. Finally, limiting processing to fragments within the triangle instead of the triangle's screen-space bounding box is a likely way to improve performance. While we did not yet have success in doing so, this may be a matter of crafting the right implementation. It may also be more advantageous for less dense geometry than the data sets we were targeting in this work.

8. Acknowledgments

The authors wish to thank following data set providers: Keenan Crane for the *Spot* model [CPS13]; Marko Dabrovic, Frank Meinl, Hans-Kristian Arntzen, Morgan McGuire, Crytek and Ludicon for *Sponza* in glb format with lossless-compressed textures [DMC*26]; Gildas Sidobre and the NRHK for the images of the Komainu Kobe [Gil20]; Iconem and the Fondazione Musei Civici di Venezia for the Venice data set [ItF26]; NVIDIA for the *Zorah* data set [NVI25b].

This research has been funded by WWTF project *ICT22-055 - Instant Visualization and Interaction for Large Point Clouds* and WWTF project *ICT25-084 - Instant Visualization and Editing of Arbitrarily Large 3D Data Sets*.

Open Access funding provided by Technische Universitat Wien.

References

[BFH10] BRUNHAVER, JOHN S, FATAHALIAN, KAYVON, and HANRAHAN, PAT. "Hardware implementation of micropolygon rasterization with motion and defocus blur." *High Performance Graphics*. 2010, 1–9 2.

[BH13] BURNS, CHRISTOPHER A. and HUNT, WARREN A. "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading". *Journal of Computer Graphics Techniques (JCGT)* 2.2 (Aug. 2013), 55–69. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/02/04/ 3.

[BV26] BENE, ROBERT and VALASEK, GÁBOR. *Helper-Lane Optimized Triangulation of Polygons*. short paper. 2026 3.

[Cat74] CATMULL, EDWIN EARL. *A subdivision algorithm for computer display of curved surfaces*. The University of Utah, 1974 2.

[CPS13] CRANE, KEENAN, PINKALL, ULRICH, and SCHRÖDER, PETER. "Robust fairing via conformal curvature flow". *ACM Transactions on Graphics (TOG)* 32.4 (2013), 1–10 11.

[DMC*26] DABROVIC, MARKO, MEINL, FRANK, CRYTEK, et al. *Sponza*. NVIDIA nvpro-samples. Sponza has undergone several adjustments by different authors over the years. Originally created by Marko Dabrovic, then re-modelled by Frank Meinl at Crytek, Morgan McGuire, Hans-Kristian Arntzen and Ludicon. 2026. URL: https://github.com/ludicon/sponza-gltf 7, 11.

[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/AlexEvans_SIGGRAPH-2015-sml.pdf [Accessed 23-April-2026]. 2015 3, 4.

[Gil20] GILDAS SIDOBRE, NRHK. *Komainu Kobe Ikuta-jinja*. Distributed by Open Heritage 3D. 2020. DOI: 10.26301/1wv3-9775 7, 11.

[GKLR13] GÜNTHER, CHRISTIAN, KANZOK, THOMAS, LINSEN, LARS, and ROSENTHAL, PAUL. "A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds". *J. WSCG* 21 (2013), 153–161 3.

[Hab21] HABLE, JOHN. *Visibility Buffer Rendering with Material Graphs*. 2021. URL: http://filmicworlds.com/blog/visibility-buffer-rendering-with-material-graphs/ 4.

[HFEM26] HAHLBOHM, FLORIAN, FRANKE, LINUS, EISEMANN, MARTIN, and MAGNOR, MARCUS. *Faster-GS: Analyzing and Improving Gaussian Splatting Optimization*. 2026. arXiv: 2602.09999 [cs.CV]. URL: https://arxiv.org/abs/2602.09999 3.

[ItF26] ICONEM and the FONDAZIONE MUSEI CIVICI DI VENEZIA. *Venice*. 2026. URL: https://iconem.com/ 7, 11.

[Kap26] KAPOULKINE, ARSENY. *Meshoptimizer / gltfpack v1.1*. https://github.com/zeux/meshoptimizer. Apr. 2026 3, 7, 9, 11.

[Kar26] KARIS, BRIAN. *Nanite + Reyes*. 2026. URL: https://graphicrants.blogspot.com/2026/02/nanite-reyes.html 11.

[KKG*26] KUBISCH, CHRISTOPH, KNOWLES, PYARELAL, GAUTRON, PASCAL, et al. *NVIDIA RTX Mega Geometry Now Available with New Vulkan Samples*. 2026. URL: https://developer.nvidia.com/blog/nvidia-rtx-mega-geometry-now-available-with-new-vulkan-samples/ 11.

[KKI*18] KERBL, BERNHARD, KENZEL, MICHAEL, IVANCHENKO, ELENA, et al. "Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the modern GPU". *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (Aug. 2018). DOI: 10.1145/3233302. URL: https://doi.org/10.1145/3233302 3.

[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/ 3.

[KKSS18] KENZEL, MICHAEL, KERBL, BERNHARD, SCHMALSTIEG, DIETER, and STEINBERGER, MARKUS. "A High-Performance Software Graphics Pipeline Architecture for the GPU". *ACM Trans. Graph.* 37.4 (Nov. 2018). DOI: 10.1145/3197517.3201374 2.

[KKT*18] KENZEL, MICHAEL, KERBL, BERNHARD, TATZGERN, WOLFGANG, et al. "On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing". *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (Aug. 2018). DOI: 10.1145/3233303 3.

[KOK*25] KUTH, BASTIAN, OBERBERGER, MAX, KAWALA, FELIX, et al. "Real-time meshlet decompression". *Computers & Graphics* 131 (2025), 104292. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2025.104292. URL: https://www.sciencedirect.com/science/article/pii/S0097849325001335 11.

[KSW21] KARIS, BRIAN, STUBBE, RUNE, and WIHLIDAL, GRAHAM. *A Deep Dive into Nanite Virtualized Geometry*. SIGGRAPH 2021 Course: Advances in Real-Time Rendering in Games. Industry talk. 2021. URL: https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf 2–4, 11.

[Kub18] KUBISCH, CHRISTOPH. *Introduction to Turing Mesh Shaders*. accessed 2026-06-08. Sept. 2018. URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/ 10.

[KWS25] KRISTMANN, ELIAS, WIMMER, MICHAEL, and SCHÜTZ, MARKUS. *Variable-Rate Texture Compression: Real-Time Rendering with JPEG*. 2025. arXiv: 2510.08166 [cs.GR]. URL: https://arxiv.org/abs/2510.08166 7.

[LHLW10] LIU, FANG, HUANG, MENG-CHENG, LIU, XUE-HUI, and WU, EN-HUA. "FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects". *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: Association for Computing Machinery, 2010, 75–82. ISBN: 9781605589398. DOI: 10.1145/1730804.1730817. URL: https://doi.org/10.1145/1730804.1730817 2, 3.

[LK11] LAINE, SAMULI and KARRAS, TERO. "High-performance software rasterization on GPUs". *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, 79–88. ISBN: 9781450308960. DOI: 10.1145/2018323.2018337. URL: https://doi.org/10.1145/2018323.2018337 2.

[LRSF] LEWIS, NOAH, RÜCKERT, DARIUS, STAMMINGER, MARC, and FRANKE, LINUS. "NePO: Neural Point Octrees for Large-Scale Novel View Synthesis". *Computer Graphics Forum* (), e70287. DOI: https://doi.org/10.1111/cgf.70287. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.70287. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.70287 3.

[MCEF94] MOLNAR, STEVEN, COX, MICHAEL, ELLSWORTH, DAVID, and FUCHS, HENRY. *A Sorting Classification of Parallel Rendering*. 1994. DOI: 10.1109/38.291528 2.

[Mic20a] MICROSOFT. *Meshlet Instancing*. accessed 2026-06-08. 2020. URL: https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12MeshShaders/src/MeshletInstancing 10.

[Mic20b] MICROSOFT. *Rasterization Rules*. accessed 2026-06-07. Aug. 2020. URL: https://learn.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-rasterizer-stage-rules 10.

[MWH18] MARRS, ADAM, WATSON, BENJAMIN, and HEALEY, CHRISTOPHER. "View-warped Multi-view Soft Shadows for Local Area Lights". *Journal of Computer Graphics Techniques (JCGT)* 7.3 (2018), 1–28 3.

[NVI18] NVIDIA NVPRO-SAMPLES. *Vulkan OpenGL CAD Mesh Shader Sample*. accessed 2026-06-08. 2018. URL: https://github.com/nvpro-samples/gl_vk_meshlet_cadscene 10.

[NVI25a] NVIDIA. *Visualizing Next-Gen Games With RTX Neural Rendering and Unreal Engine 5*. GDC 2025. 2025. URL: https://www.youtube.com/watch?v=udqApkIqZmQ 6.

[NVI25b] NVIDIA. *Zorah*. NVIDIA nvpro-samples. Export of NVIDIA RTX Kit - Zorah Sample as presented in "NVIDIA RTX Advances with Neural Rendering and Digital Human Technologies at GDC 2025". 2025. URL: https://github.com/nvpro-samples/vk_lod_clusters/blob/main/README.md#zorah-demo-scene 6, 7, 11.

[OKM23] OBERBERGER, MAX, KUTH, BASTIAN, and MEYER, QUIRIN. *From Vertex Shader to Mesh Shader*. accessed 2026-06-08. Dec. 2023. URL: https://gpuopen.com/learn/mesh_shaders/mesh_shaders-from_vertex_shader_to_mesh_shader/ 10.

[Pin88] PINEDA, JUAN. "A parallel algorithm for polygon rasterization". *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, 17–20. ISBN: 0897912756. DOI: 10.1145/54852.378457. URL: https://doi.org/10.1145/54852.378457 5.

[PTSO15] PATNEY, ANJUL, TZENG, STANLEY, SEITZ Kerry A., JR., and OWENS, JOHN D. "Piko: a framework for authoring programmable graphics pipelines". *ACM Trans. Graph.* 34.4 (July 2015). ISSN: 0730-0301. DOI: 10.1145/2766973. URL: https://doi.org/10.1145/2766973 2.

[RFS22] RÜCKERT, DARIUS, FRANKE, LINUS, and STAMMINGER, MARC. "Adop: Approximate differentiable one-pixel point rendering". *ACM Transactions on Graphics (ToG)* 41.4 (2022), 1–14 3.

[SD15] SCHIED, CHRISTOPH and DACHSBACHER, CARSTEN. "Deferred attribute interpolation for memory-efficient deferred shading". *Proceedings of the 7th Conference on High-Performance Graphics*. 2015, 43–49 3.

[SH74] SUTHERLAND, IVAN E. and HODGMAN, GARY W. "Reentrant polygon clipping". *Commun. ACM* 17.1 (Jan. 1974), 32–42. ISSN: 0001-0782. DOI: 10.1145/360767.360802. URL: https://doi.org/10.1145/360767.360802 5.

[SKW21] SCHÜTZ, MARKUS, KERBL, BERNHARD, and WIMMER, MICHAEL. "Rendering Point Clouds with Compute Shaders and Vertex Order Optimization". *Computer Graphics Forum* 40.4 (July 2021), 115–126. ISSN: 1467-8659. DOI: 10.1111/cgf.14345. URL: https://www.cg.tuwien.ac.at/research/publications/2021/SCHUETZ-2021-PCC/ 3.

[Web15] WEBER, THOMAS. "Micropolygon Rendering on the GPU". MA thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, Jan. 2015. URL: https://www.cg.tuwien.ac.at/research/publications/2015/WEBER-2015-PRA1/ 2.

[WHG84] WEGHORST, HANK, HOOPER, GARY, and GREENBERG, DONALD P. "Improved Computational Methods for Ray Tracing". *ACM Trans. Graph.* 3.1 (Jan. 1984), 52–69. ISSN: 0730-0301. DOI: 10.1145/357332.357335. URL: https://doi.org/10.1145/357332.357335 3.