Proxy-GS: Unified Occlusion Priors for Training and Inference in Structured 3D Gaussian Splatting

Supplementary Material

Table 1. Partition information in MatrixCity.

Block	$x_{\min}$	$x_{\max}$	$y_{\min}$	$y_{\max}$
1	-9.80	-2.64	0	3.9
2	-2.64	0.44	0	3.9
3	0.44	3.52	0	3.9
4	3.52	8.70	0	3.9
5	-6.90	6.90	3.9	7.4

Table 2. **The FPS of different depth acquisition methods on Block 5.**

Method	Nvdiffrast	3DGS	Ours
FPS	32	54	151

Figure 2. **Visualization on different safety margins.**

1. Appendix

1.1. Division detail on MatrixCity

We divide all the Horizon street scenes in MatrixCity's small city into five blocks (eg. Block 1, Block 2), The partition margin details is in Tab. 1

1.2. Combine with Hardware 3DGS

We combine our method with Hardware 3DGS [1] in Tab. 3 and Tab. 4. As observed, the FPS improves across all datasets, but due to the precision settings used, there is a noticeable decline in rendering quality.

1.3. Comparison with different depth-acquisition approaches

In Tab. 2, To demonstrate the effectiveness of our depth acquisition strategy, we also evaluate two alternative approaches: directly rendering depth using nvdiffrast [7], and extracting depth from a pre-trained 3DGS model. These two depth-generation baselines allow us to compare our method against commonly used depth sources.

1.4. Average decoded anchor number on all the datasets

. In Tab. 5, we report the average number of anchors used during training and inference across all datasets. It can be observed that our method consistently reduces the decoding burden, although the degree of improvement varies across different scenes.

Safety margin of the occlusion culling. In Tab. 6, we report results on the Small City dataset by varying the depth culling threshold γ in Eq. **??**. We observe that $\gamma = 0.3$ yields the best trade-off between rendering quality and speed. As can be seen in Fig. 2, when the threshold is too small $\gamma = 0.1$, it leads to rendering artifacts in nearby regions. However, setting γ too large is also undesirable: a larger threshold introduces excessive anchors, which increases structural redundancy and reduces FPS, while a too small threshold restricts anchor growth and degrades rendering quality.

1.5. Visualization on different Vertex Noise

As illustrated in Fig. 3, increasing noise introduces spurious protrusions on the ground, which smear the image and progressively degrade rendering quality.

1.6. Mesh extraction on different datasets

1.6.1. Indoor and Outdoor Scenes with Dense Point Clouds

We describe the mesh extraction process when dense point clouds are available for both indoor and outdoor environments. This category includes real-world datasets that provide LiDAR point clouds (e.g., [10]), where mesh generation can be directly performed using surface reconstruction methods, such as [4]. In addition, for synthetic datasets such as MatrixCity, ground-truth depth maps are available, which can be fused via TSDF to obtain high-quality meshes.

Table 3. Combine with Hardware 3DGS [1], quantitative results on MatrixCity [8]

Methods	Block 1&2				Block 3&4				Block 5			
	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑
Proxy-GS	**22.11**	**0.751**	**0.330**	**126**	**21.06**	**0.751**	**0.348**	**134**	**21.68**	**0.744**	**0.362**	**151**
+Hardware 3DGS [1]	**22.05**	**0.747**	**0.338**	**167**	**20.86**	**0.743**	**0.357**	**174**	**21.58**	**0.735**	**0.372**	**196**

Table 4. Combine with Hardware 3DGS [1], quantitative results on real world Outdoor and Indoor datasets [2, 6, 10].

Methods	CUHK-LOWER				Berlin				Small City			
	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑
Proxy-GS	26.44	0.795	0.262	239	27.85	0.912	0.216	275	23.09	0.736	0.344	139
+Hardware 3DGS [1]	26.28	0.787	0.265	280	27.78	0.906	0.210	325	22.91	0.732	0.343	163

1.6.2. Indoor Scenes with Sparse COLMAP Point Clouds

We describe the workflow of mesh extraction in indoor scenes where only sparse COLMAP reconstructions are available. Directly relying on COLMAP to generate dense point clouds in indoor environments is often unreliable, as such scenes frequently contain large textureless regions. To address this challenge, To address this challenge, we leverage a foundation-model approach similar to VGGT [9]. Our experiments show that VGGT-style models exhibit strong robustness in indoor scenes. However, since our pipeline requires alignment with COLMAP poses, we adopt MapAnything [5] instead, using RGB images together with COLMAP inputs to obtain a dense point cloud. This hybrid strategy enables the recovery of reasonable indoor meshes despite the limitations of sparse COLMAP input.

1.6.3. Outdoor Scenes with Sparse COLMAP Point Clouds

For outdoor environments where the reconstruction relies solely on sparse COLMAP point clouds, the abundance of feature points generally mitigates the issue of sparse textures. However, due to the large spatial extent, many 3DGS-based indoor reconstruction methods encounter out-of-memory (OOM) problems when applied to outdoor scenes. To address this, we employ CityGS-X [3], a state-of-the-art large-scale geometric reconstruction framework, which leverages multi-GPU parallelism to achieve scalable mesh generation with competitive performance.

1.6.4. Mesh visualization

As shown in Fig. 4, we visualize all the lightweight proxies. Our method does not require highly accurate meshes; an approximate geometry is sufficient. Thanks to the anchor-based filtering, the subsequent growth of Gaussians introduces offsets that provide additional tolerance, thereby ensuring that our approach maintains a certain degree of robustness to mesh inaccuracies.

1.7. Fast Depth Acquisition

1.7.1. Overview.

We follow a modern real-time rendering pipeline to obtain high-quality depth maps at minimal latency. The key ideas are: (i) *preprocess* the reconstructed mesh into compact *clusters*; (ii) perform fully GPU-resident *frustum* and *hierarchical-Z (Hi-Z)* occlusion culling at *cluster* granularity each frame; (iii) emit a *depth-only* pass that leverages Early-Z; and (iv) *zero-copy* the resulting depth buffer into the learning runtime (PyTorch) via Vulkan–CUDA interop, avoiding CPU round trips. This section details each component.

1.7.2. Preprocessing: from reconstructed mesh to clusters.

Given a triangle mesh $\mathcal{M} = (\mathcal{V}, \mathcal{F})$ obtained by the reconstruction routine above, we apply the following:

1. **Topology-preserving simplification.** We reduce face count with a quadric-error-metric (QEM) style simplifier while enforcing feature and boundary preservation. For a vertex in homogeneous coordinates $\tilde{\mathbf{x}} = (x, y, z, 1)^\top$ and its incident face planes $\{\mathbf{p}_f = (a, b, c, d)^\top\}$ (with $\|(a, b, c)\|_2 = 1$ for all f), the local quadric is

$$Q = \sum_f \mathbf{p}_f \mathbf{p}_f^\top,$$

These per-vertex quadrics are *accumulated* and then used by an edge-collapse procedure to decide the contraction position and cost, which removes superfluous micro-triangles commonly produced by reconstruction and improves cache locality and GPU occupancy.

Edge-collapse simplification with QEM. For each vertex v, accumulate $Q_v = \sum_{f \in N(v)} \mathbf{p}_f \mathbf{p}_f^\top$. To collapse an edge (i, j), combine quadrics

$$Q' = Q_i + Q_j, \qquad E(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}}^\top Q' \tilde{\mathbf{x}}.$$

Table 5. Average anchor number used to decode all the datasets

Method	Block 1&2	Block 3&4	Block 5	Berlin	CUHK-LOWER	Small City
Proxy-GS	190k	190k	80k	40k	110k	350k
Octree-GS	800k	1040k	720k	60k	120k	840k

GT Vertex Noise 0% Vertex Noise 5% Vertex Noise 10% Vertex Noise 20%

Figure 3. Quantitative visualization of different Vertex noise on Block 5.

Table 6. **Ablations of different safety margin of depth culling γ trained on Small City [6].**

γ	PSNR↑	SSIM↑	LPIPS↓	FPS↑
0.1	22.94	0.734	0.349	142
0.3	23.09	0.736	0.344	139
0.6	23.02	0.735	0.348	135
1.0	23.05	0.736	0.345	128

Partition Q' as $Q' = \begin{bmatrix} A & \mathbf{b} \\ \mathbf{b}^\top & c \end{bmatrix}$ with $A \in \mathbb{R}^{3\times3}$, $\mathbf{b} \in \mathbb{R}^3$, $c \in \mathbb{R}$. The optimal contraction position is

$$\begin{aligned} \mathbf{x}^* &= \arg\min_{\mathbf{x}\in\mathbb{R}^3} \ \mathbf{x}^\top A\,\mathbf{x} + 2\,\mathbf{b}^\top\mathbf{x} + c \\ &= -\,A^{-1}\mathbf{b} \quad \text{(if } A \text{ is invertible).} \end{aligned} \tag{1}$$

with cost $\delta = E([\mathbf{x}^{*\top},\, 1]^\top)$.
If A is singular, evaluate $\{\mathbf{x}_i, \mathbf{x}_j, (\mathbf{x}_i + \mathbf{x}_j)/2\}$ and pick the one with minimal E. We maintain a priority queue keyed by δ and iteratively collapse the lowest-cost edge, updating connectivity and setting the new vertex quadric to Q'. Collapses that would break manifoldness or flip triangle orientations are forbidden.

Boundary/feature preservation. For a boundary or sharp-crease edge with unit tangent $\mathbf{t}$ and unit average normal $\hat{\mathbf{n}}$, add two *constraint planes* whose intersection is the edge line,

$$\mathbf{p}_1 = (\hat{\mathbf{n}},\ -\hat{\mathbf{n}}^\top\mathbf{x}_0)^\top, \quad \mathbf{p}_2 = (\widehat{\mathbf{t}\times\hat{\mathbf{n}}},\ -\widehat{\mathbf{t}\times\hat{\mathbf{n}}}^\top\mathbf{x}_0)^\top,$$

and augment incident vertex quadrics by

$$Q_v \leftarrow Q_v + \lambda_b\,\mathbf{p}_1\mathbf{p}_1^\top + \lambda_b\,\mathbf{p}_2\mathbf{p}_2^\top,$$

with a large weight λ_b. Alternatively, restrict collapses so boundary vertices only collapse along the boundary, and forbid collapses across edges whose dihedral angle exceeds a feature threshold.

2. **Cluster construction.** We partition the simplified mesh into triangle sets $\{\mathcal{L}_k\}_{k=1}^K$ such that $\bigsqcup_k \mathcal{L}_k = \mathcal{F}$ and $\tau_{\min} \le |\mathcal{L}_k| \le \tau_{\max}$. For each cluster we precompute: (a) an object-space axis-aligned bounding box (AABB) $\mathrm{AABB}_k = [\mathbf{b}_k^{\min}, \mathbf{b}_k^{\max}]$; and (b) a conservative screen-space bounding rectangle at level-0, $R_k^{(0)}$, for any given view. Project the AABB's eight corners $\{\mathbf{x}_{k,j}\}_{j=1}^8$ with the view-projection PV:

$$\mathbf{y}_{k,j} = PV\begin{bmatrix}\mathbf{x}_{k,j}\\ 1\end{bmatrix}, \qquad \mathbf{u}_{k,j}^{\mathrm{ndc}} = \left(\tfrac{y_{k,j}^x}{y_{k,j}^w},\ \tfrac{y_{k,j}^y}{y_{k,j}^w}\right).$$

Let the viewport be $W \times H$ (origin at the top-left). Map to pixels

$$\mathbf{s}_{k,j} = \Big(\tfrac{W}{2}(\,u_x^{\mathrm{ndc}} + 1),\ \tfrac{H}{2}(u_y^{\mathrm{ndc}} + 1)\Big),$$

then take an outward-rounded, padded box (padding $\Delta \in \{0, 1\}$) and clip to the screen:

$$\begin{aligned} R_k^{(0)} = \Big[\,\lfloor \min_j \mathbf{s}_{k,j} \rfloor - \Delta,\ \lceil \max_j \mathbf{s}_{k,j} \rceil + \Delta\,\Big] \\ \cap\ [0, W-1] \times [0, H-1]. \end{aligned} \tag{2}$$

Such cluster construction helps us to do cluster-level culling, increasing granularity compared to per-triangle culling while retaining high selectivity.

Per-frame visibility: frustum and Hi-Z occlusion. Let $\{\Pi_i\}_{i=1}^6$ be the frustum planes with *inward* normals $\mathbf{n}_i$ and offsets d_i. A cluster $\mathcal{L}_k$ with AABB_k corners $\{\mathbf{x}_j\}_{j=1}^8$ is frustum-culled if

$$\exists\, i \text{ s.t. } \max_j\big(\mathbf{n}_i^\top\mathbf{x}_j + d_i\big) < 0. \tag{3}$$

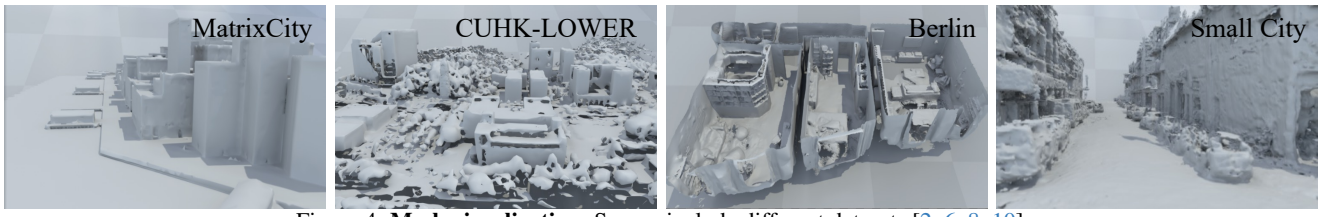

Figure 4. **Mesh visualization**. Scenes include different datasets [2, 6, 8, 10].

Let $Z^{(0)}(u, v)$ be the base depth. The Hi-Z pyramid for standard depth is

$$Z^{(\ell+1)}(u, v) = \max_{\delta_x, \delta_y \in \{0,1\}} Z^{(\ell)}(2u + \delta_x,\, 2v + \delta_y). \quad (4)$$

Level snapping and conservative depth. Given $R_k^{(0)}$, choose a pyramid level ℓ (e.g., $\ell = \mathrm{clamp}(\lfloor \log_2(\max(\mathrm{width}(R_k^{(0)}), \mathrm{height}(R_k^{(0)})))\rfloor - c,\ 0,\ L_{\max})$ with a small constant $c \in \{1, 2\}$), and snap the rectangle to level ℓ by outward rounding:

$$R_k^{(\ell)} = \left[\ \left\lfloor \frac{R_{k,\min}^{(0)}}{2^\ell} \right\rfloor,\ \left\lceil \frac{R_{k,\max}^{(0)}}{2^\ell} \right\rceil\ \right].$$

Let $\mathbf{y}_{k,j} = PV\big[\mathbf{x}_{k,j}^\top,\, 1\big]^\top$ denote the clip-space 4-vectors of the eight AABB corners introduced above (the same ones used to build $R_k^{(0)}$). A conservative near-depth estimate for the cluster is

$$\hat{z}_k \;=\; \min_{j=1,\ldots,8} \left(\max \left(z_{\text{near}}^{\text{ndc}},\ \frac{y_{k,j}^z}{y_{k,j}^w} \right) \right),$$

If any $y_{k,j}^w \le 0$, the near-plane clamp above makes the estimate conservative; alternatively, one may skip the occlusion test for full safety.

Given the screen-space bounding box R_k of $\mathcal{L}_k$ snapped to level ℓ, and a conservative near depth $\hat{z}_k$ of $\mathcal{L}_k$, the occlusion test is

$$\text{occluded}(\mathcal{L}_k) \iff \hat{z}_k \ge \max_{(u,v) \in R_k^{(\ell)}} Z^{(\ell)}(u, v). \quad (5)$$

Depth-only pass with early-Z. After visibility, we render only the surviving clusters in a *solid, depth-only* pipeline (color writes disabled, depth writes enabled). A minimal fragment shader lets the rasterizer perform early-depth testing. This produces the depth map $D \in \mathbb{R}^{H \times W}$ used downstream.

Zero-copy interop to PyTorch. In order to obtain the depth every frame efficiently, a naive path would be to read back the GPU depth buffer to host memory and then upload it to CUDA, introducing synchronization and PCIe traffic. Instead, we adopt a fully GPU-resident path: we render with Vulkan and export the depth image's memory as an *external file descriptor* (FD). On the CUDA side, we import that FD as external memory and map it to a device pointer; the pointer is then wrapped as a PyTorch CUDA tensor without a copy.