HANDLING HIDDEN HOLES IN VIEW-BASED 3D SCENE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

3D scene generation from a single image is a challenging task with many applications in areas such as AR/VR, gaming, and embodied AI. One popular family of methods for this task is iterative view-based 2D inpainting, which expands 3D scene coverage by projecting known geometry from existing views into a novel view and inpainting pixels with missing depth. These methods typically assume that all occluded geometry in existing views appears as holes of missing depth in the novel view, and that filling these visible holes completes the scene. However, this assumption is often incorrect: although visible holes necessarily mean missing geometry, the lack of visible holes does not necessarily mean no geometry is missing. It is possible that there are no visible holes among pixels corresponding to potentially missing geometry because some other object also projects to these pixels in the novel view. We refer to these pixels as hidden holes, which can prevent inpainting methods from properly completing occluded objects because they are never flagged for inpainting. To address this, we introduce QuadMask, our geometric occlusion detection system that identifies both visible and hidden holes in novel views, given existing views and camera poses as input. QuadMask outputs can aid downstream RGB pipelines by signaling areas that could be missing visually important geometry.

1 INTRODUCTION

3D scene generation from a single image is an important problem with many real-world applications. In AR/VR and gaming, it enables creation of immersive and realistic content. For embodied AI, it provides agents with a better understanding of their surroundings, aiding navigation and interaction. Reconstructing 3D scenes from a single image makes these systems more practical and deployable.

One common approach to this problem is view-based 2D inpainting, which iteratively expands scene coverage by synthesizing novel views. Starting from a single view with RGB and depth information, these methods unproject per-view pixels into 3D space and reproject the result into a novel camera view. This process naturally produces holes in the novel view with no valid depth for two reasons: (1) occluded geometry in input views remains unseen and is therefore missing in the novel view; and (2) the novel camera's field of view extends beyond the coverage of the input views. We refer to these as **visible holes** (see Fig. 1). Standard pipelines use 2D inpainting models to fill visible holes, generating completions for the missing regions.

However, these approaches assume that all regions requiring inpainting will appear as visible holes—a false assumption that can lead to incomplete scene generation. The key issue is that although visible holes necessarily mean missing geometry, the lack of visible holes does not necessarily mean no geometry is missing. For example, consider generating a 3D scene from a single image of a room with a chair viewed from the side (see Fig. 1). In this input image, only one side of the chair is visible, and the geometry of the other side is missing and needs to be completed through inpainting. However, when geometry from the input view is projected into a new camera placed behind the chair's back, there are no visible holes among the pixels that correspond to the missing chair geometry, because some other object, in this case the background wall, also projects to these pixels in the new view. We refer to these pixels that require inpainting but do not appear as visible holes as **hidden holes**. Hidden holes form because pixels in a new view can be occupied by objects

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



Figure 1: Demonstration of **visible vs. hidden holes** in view-based 2D inpainting. Visible holes appear as white gaps in the novel view and lack depth. Hidden holes, such as the missing back of the chair (top row), appear filled because they incorrectly inherit depth from background surfaces.

located behind the missing 3D geometry. Hidden holes are irrecoverable by current view-based 3D generation methods, as they are not flagged for inpainting.

In this work, we introduce QuadMask, a geometric occlusion detection system that produces augmented inpainting masks identifying both visible and hidden holes, helping 3D scene generation pipelines recover occluded regions and achieve more accurate scene completions. Given single- or multi-view RGB images and poses, QuadMask first predicts per-view depth maps and converts them into watertight triangulated surfaces. It then computes the 3D regions occluded in input views and intersects them with the viewing frustums of novel view pixels. This enables QuadMask to identify all visible and hidden holes in the novel view. When integrated into existing view-based generation pipelines, QuadMask improves object completion and overall scene fidelity across single-image 3D scene generation methods. In summary, our contributions are:

- We identify the problem of hidden holes, which prevent proper object completion in view-based 2D inpainting pipelines for 3D scene generation.
- We introduce QuadMask, a geometric occlusion detection method that finds both visible and hidden holes in novel views given single- or multi-view RGB images and poses.
- We provide a prior for more visually coherent downstream view completion by correctly flagging all areas where new geometry can potentially be added.

2 Related Work

Direct 3D Synthesis. Some 3D scene generation methods (Meng et al., 2025; Szymanowicz et al., 2025a;b; Dahnert et al., 2024) predict hidden 3D scene structure directly without iteratively building an intermediate representation. Unlike these methods, QuadMask separates 3D scene generation into two parts: where hidden structure can exist, versus downstream generation appearance.

Joint novel-view generation followed by 3D reconstruction. Another class of 3D scene generation methods instead generate unseen content in a coordinated image (Gao et al., 2024), panorama (Li et al., 2024), video (Müller et al., 2024), or latent space (Liang et al., 2025), and then use the resulting observations to reconstruct the scene in 3D. The generated novel views are not perfectly 3D consistent, which is usually addressed during the 3D reconstruction or fusion step. QuadMask instead provides an explicit geometric indication of where unobserved foreground structure could occur, which could serve as an additional cue helpful for self-consistent iterative inpainting.

Iterative view-based scene expansion. The class of 3D scene generation methods that are most related to QuadMask is iterative view-based inpainting methods. Höllein et al. (2023) and Lei et al. (2023) maintain a centralized textured mesh representation that is iteratively expanded. Other works instead use a point cloud representation, which is then used for scene reconstruction via 3D Gaussian splatting (3DGS) (Chung et al., 2023; Kerbl et al., 2023) or text-guided outpainting (Yu et al., 2024). Yu et al. (2025) uses Gaussian surfels to represent the foreground, background, and sky layers of each view. Compared to QuadMask, these methods do not explicitly consider the existence of hidden holes during their inpainting steps, which can lead to degraded performance.

Occluded-space reasoning and volumetric guidance for 3D scene generation. Some 3D scene generation methods reason about occluded-space volumes in their generation process. Shriram et al. (2025) constructs an occupancy grid around an initial point cloud and uses reference-camera ray traversal to identify voxel centers hidden behind the first occupied surface. This occlusion volume is then used to initialize additional 3D Gaussians and to serve as a conditioning mask for an inpainting diffusion model, ultimately allowing for the learning of occluded structures within a 3DGS framework. However, Shriram et al. (2025) is not built for iterative scene inpainting pipelines, whereas QuadMask is. Salimi et al. (2025) uses estimated input view depth and cameras to construct structures like reference meshes and silhouette-driven shadow volumes that can be projected into a target view as cues for occlusion-aware inpainting. However, their method takes in user-specified inpainting masks and only uses occlusion information to improve inpainting in these regions, whereas QuadMask uses occlusion information to identify potential inpainting regions for scene completion.

Graphics foundations for QuadMask geometry primitives. QuadMask's occlusion prism construction and target pixel frustums are related to classical computer graphics. Crow (1977) and McCool (2000) work on representing source-occluded space as explicit volumetric polygonal geometry. Heckbert & Hanrahan (1984) performs visibility analysis by replacing ray tracing with beam tracing. These works establish QuadMask's underlying graphics primitives.

Hierarchical visibility culling and quadtree acceleration. Glassner (1984) uses spatial subdivision to reduce expensive ray–object intersection tests; Samet (1984) surveys quadtrees and related structures based on recursive decomposition; and Greene et al. (1993) combines object-space and image-space hierarchies to cull visibility computations. QuadMask applies a similar hierarchical-culling principle to accelerate its search for potential hidden holes.

3 METHODOLOGY

Given single- or multi-view RGB images, our method produces augmented inpainting masks that identify both visible holes and hidden holes in novel views. We first preprocess input images by predicting per-view depth maps and converting them into watertight triangulated surfaces with proper surface normals and occlusion boundary preservation (§ 3.1). We then apply QuadMask, our geometric occlusion detection system that identifies novel view pixels potentially observing geometry occluded in the input views (§ 3.2). QuadMask represents occluded 3D regions as triangular prisms and efficiently tests for intersections with novel view pixel frustums through combinatorial vertex enumeration and hierarchical quadtree acceleration. Finally, we integrate QuadMask into existing view-based 3D scene generation pipelines, enabling improved scene completion (§ 3.3).

3.1 PREPROCESSING IMAGES AND 3D REPRESENTATION

To enable geometric occlusion detection, we first convert input RGB images into per-view 3D geometry in a standardized form: oriented, watertight triangulated meshes with connectivity that respects occlusion boundaries. This preprocessing step provides the information needed for QuadMask to perform occlusion reasoning.

We first predict a point map using depth from a monocular depth estimation model, and known camera pose and intrinsics. We then identify occlusion boundaries on this map using depth discontinuities between neighboring pixels. Each point map and occlusion boundary set is converted to a watertight mesh, by using bilinear interpolation on same-surface points to estimate pixel-corner positions. This process constructs a watertight triangulated mesh without connecting distinct surfaces. See our supplementary for details.

3.2 QUADMASK: GEOMETRIC OCCLUSION DETECTION

Given the triangle sets $\mathcal{S}_v$ for each input view and a target novel viewpoint, QuadMask computes an augmented inpainting mask that identifies all pixels requiring inpainting due to occlusion. We first represent the 3D regions occluded in the input views as triangular prisms constructed from $\mathcal{S}_v$ (§ 3.2.1). We then formulate occlusion detection as a polytope intersection problem (§ 3.2.2) and solve it efficiently through combinatorial vertex enumeration (§ 3.2.3). Because we must compute the intersection problem over all novel view pixels, we can further accelerate it via hierarchical

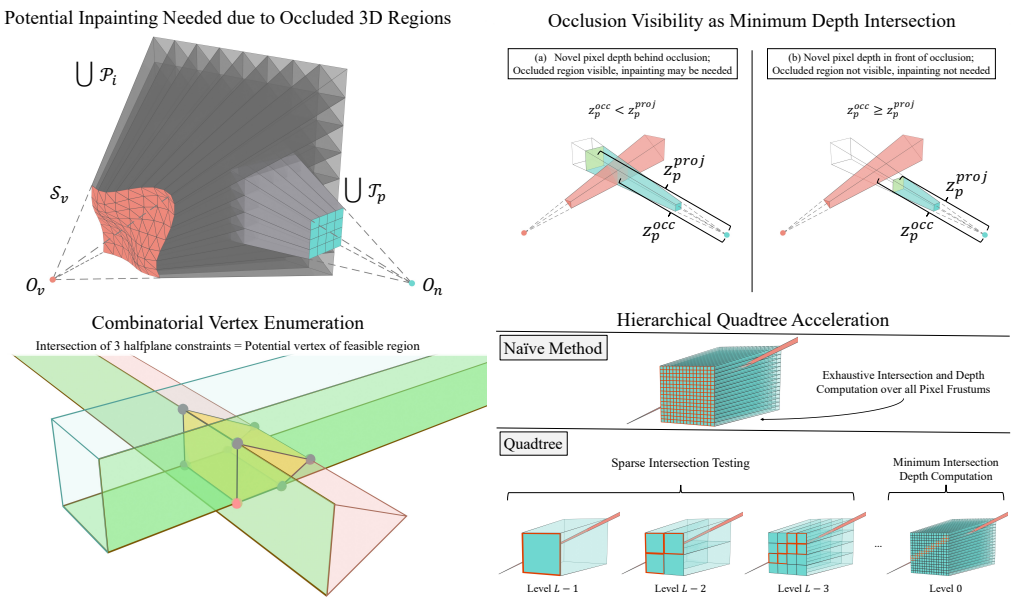


Figure 2: *(Top left)* Occluded 3D region $\cup_i \mathcal{P}_i$, formed by extruding $\mathcal{S}_v$. *(Top right)* Potential inpainting pixels identified by comparing novel pixel depth with polytope intersection depth. *(Bottom left)* Finding minimum intersection depth is sped up on GPU by testing all combinations of 3 faces and checking vertex validity. *(Bottom right)* Quadtree acceleration reduces the number of intersection and depth computations needed.

quadtree spatial culling methodology (§ 3.2.4). Finally, we integrate results across multiple input views and apply backface culling to produce the complete augmented mask in (§ 3.2.5).

3.2.1 Occluded 3D Regions Representation

For each input view v, the triangle set $\mathcal{S}_v$ represents surfaces visible from that view. The 3D space behind these surfaces, away from the input camera, is therefore the occluded 3D region for that view. To represent this occluded region, we extrude each triangle $t_i \in \mathcal{S}_v$ into a triangular prism $\mathcal{P}_i$ by casting rays from the input camera center through each triangle vertex and extending them to a far depth z_{far}. Since the triangles form watertight surfaces for continuous surface regions, the union of all triangular prisms $\bigcup_i \mathcal{P}_i$ forms a volume without gaps that precisely represents the entire 3D region occluded in that input view, as shown in the top left of Fig. 2. We transform all triangular prisms into the novel view's coordinate frame and represent each prism by its 6 vertices and 5 halfplane constraints corresponding to its faces.

3.2.2 Occlusion Detection via Polytope Intersection

To construct the augmented inpainting mask, we must determine which novel view pixels might observe occluded geometry from the input views. For now, consider the case where there is only a single input view v. Then, for each pixel p in the novel view, the problem reduces to determining if the 3D occluded region represented by the union of triangular prisms $\bigcup_i \mathcal{P}_i$ is visible along the pixel's viewing frustum and appears at a depth less than the pixel's depth. If the occluded region is beyond the pixel's depth, the occlusion is not visible and no inpainting is needed.

Novel View Depth Map Generation. We generate a depth map for the novel view by rasterizing all triangles $\mathcal{S}_v$ from all input views. For each novel view pixel p, we store the depth z_p^{proj} and surface normal of the triangle with minimum depth that projects onto p. To achieve high-quality rasterization, we supersample each pixel with a grid of sub-pixel samples. We implement this rasterizer in a custom CUDA kernel; see our supplementary for details.

Pixel Frustums as Trapezoidal Prisms. The viewing frustum for a pixel p in the novel view is formed by casting rays from the novel camera center through the four corners of the pixel. Bounding these rays between near and far depth planes z_{near} and z_{far} produces a trapezoidal prism $\mathcal{T}_p$. Each prism can be represented by its 8 vertices and 6 halfplane constraints corresponding to its faces.

Problem Formulation: Minimum Depth Intersection. To determine if the occluded region $\bigcup_i \mathcal{P}_i$ from an input view v is visible to pixel p, we compute the minimum depth at which the pixel's trapezoidal prism $\mathcal{T}_p$ intersects any triangular prism $\mathcal{P}_i$. If this minimum intersection depth is smaller than the pixel p's novel view depth, then the occluded region is visible.

To compute this, we can first find the minimum z-coordinate within each intersection $\mathcal{T}_p \cap \mathcal{P}_i$. Then, the overall minimum depth can be written as:

$$z_p^{\text{occ}} = \min_i \min_{\mathbf{x} \in \mathcal{T}_p \cap \mathcal{P}_i} z(\mathbf{x})$$

If $z_p^{\text{occ}} < z_p^{\text{proj}}$, occluded geometry from the input view is visible for pixel p, and the pixel will be flagged as a potential hidden hole requiring inpainting to recover potential occluded foreground geometry. An example of this intersection between a pixel trapezoidal prism and a mesh face prism is shown on the top right of Fig. 2, in the case of flagging a hidden hole and not flagging, respectively.

3.2.3 Combinatorial Vertex Enumeration

To efficiently compute the minimum depth of intersection within $\mathcal{T}_p \cap \mathcal{P}_i$, we can formulate the problem as linear programming: minimize z-coordinate subject to 11 halfplane constraints from the two polytopes. A fundamental property of linear programs is that the optimal solution must occur at a vertex of the feasible region.

In 3D, a vertex of a convex polytope is defined as a point that cannot be expressed as a strict convex combination of other points in the polytope. Every such vertex is uniquely determined by the intersection of exactly three linearly independent active constraints—that is, three halfplane constraints $\{\mathbf{n}_i \cdot \mathbf{x} + d_i = 0\}$ whose normal vectors are linearly independent. Since the intersection $\mathcal{T}_p \cap \mathcal{P}_i$ is defined by 11 halfplane constraints—6 from the trapezoidal prism and 5 from the triangular prism—we can enumerate all possible vertices by testing all $\binom{11}{3} = 165$ combinations of three halfplanes and checking if the result satisfies all constraints simultaneously. This approach yields a fixed-cost algorithm ideal for GPU parallelization. For each triplet, we do:

1. **Form linear system:** $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a 3×3 matrix of plane normals
2. **Check for unique intersection:** If $|\det(\mathbf{A})| > \epsilon$ with $\epsilon = 10^{-8}$, the planes meet at a unique point
3. **Solve:** Compute $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ using the cofactor method for closed-form 3×3 inversion
4. **Verify feasibility:** Check if $\mathbf{x}$ satisfies all 11 constraints
5. **Track minimum:** If feasible, update $z_p^{\text{occ}} = \min(z_p^{\text{occ}}, z(\mathbf{x}))$

For degenerate cases where $|\det(\mathbf{A})| \leq \epsilon$, the plane triplets are parallel, coincident, or meet along a line. However, all vertices of the intersection polytope can be discovered by non-degenerate triplet combinations, so these cases can be safely skipped. See our supplementary for the proof.

In addition to testing all halfplane triplets, we also test the 8 vertices of $\mathcal{T}_p$ and 6 vertices of $\mathcal{P}_i$ against the opposing polytope's constraints as potential vertices for a more numerically stable and cheap check. Because our algorithm has fixed complexity, with exactly 165 triplet checks plus 14 vertex checks per pixel-triangle pair, there is no data-dependent branching. This enables efficient GPU execution, where each CUDA thread independently processes one pixel-triangle pair in parallel.

3.2.4 Hierarchical Quadtree Acceleration

Although the vertex enumeration algorithm efficiently solves individual intersection queries, naively testing all pixel-triangle pairs requires $O(HW \cdot N_{\text{tri}})$ queries. In our experiments with 512×512 images and triangulated depth maps containing 500K triangles, this amounts to over 130 billion tests per novel view. Even with GPU parallelization, this is prohibitively expensive. We address this through hierarchical spatial culling using a quadtree structure (Glassner, 1984).

Algorithm 1 Hierarchical Occlusion Detection

```
Initialize: z_p^occ ← ∞ for all pixels p
for each triangle t_i ∈ S_v do
    Transform t_i to novel view, extrude to form P_i
    active ← {root prism}
    for level ℓ = L − 1 down to 0 do
        GPU: intersect all prisms in active with P_i
        Filter to feasible intersections, update z_p^occ
        if ℓ > 0 then
            Subdivide feasible prisms into 4 children
            active ← children
        end if
    end for
end for
```

Quadtree Structure. We organize pixel frustums in a quadtree with L levels. Level 0 contains individual pixel frustums at full resolution, totaling HW prisms. Each coarser level ℓ groups 2×2 child prisms into a single parent prism, yielding $\frac{HW}{4^\ell}$ prisms at level ℓ. Each parent is a trapezoidal frustum that exactly bounds its four children.

Top-Down Traversal with Pruning. We process each triangular prism independently, traversing the quadtree from coarse to fine. At each level, we test active trapezoidal prisms against the triangular prism using vertex enumeration. If a coarse prism has no intersection with a triangular prism, none of its descendant pixels intersect, allowing us to prune the entire subtree. This reduces queries from $O(HW \cdot N_{\text{tri}})$ to approximately $O(k \cdot N_{\text{tri}})$, for some $k \ll HW$. See Alg. 1 for our quadtree algorithm, and the bottom right of Fig. 2 for an illustration of the recursive structure.

3.2.5 Multi-View Integration and Backface Culling

Initial Detection of Visible Holes and Backfaces. Before running computationally expensive quadtree intersection tests, we perform two fast checks to identify pixels that definitively require inpainting, allowing us to skip quadtree processing for these pixels entirely.

Visible holes: Pixels with no valid rasterized depth, where $z_p^{\text{proj}} = \infty$.

Backface pixels: For pixels with valid depth, we check if they observe the back side of geometry from the input views by computing:

$$\cos\theta = \mathbf{n}_p \cdot \mathbf{r}_p \tag{1}$$

where $\mathbf{n}_p$ is the surface normal of the closest rasterized triangle at pixel p, and $\mathbf{r}_p$ is the normalized viewing ray direction for pixel p. If $\cos\theta > 0$, the normal points away from the camera, indicating the pixel observes a surface that was facing away from all input view cameras. This represents self-occlusion: the back of an object that was only seen from the front, and thus requires inpainting.

These pixels are marked for inpainting and excluded from subsequent quadtree traversal, providing significant speedup by reducing the number of intersection tests.

Merging Results Across Multiple Input Views. We first identify visible holes and backface pixels as requiring inpainting. For the remaining pixels with valid rasterized depth, we must determine if they are hidden holes by reasoning across multiple input views.

For each input view v, the quadtree algorithm produces an occlusion depth map $z_p^{\text{occ},v}$ representing the minimum depth of occluded geometry visible to each pixel p. A pixel is flagged as potentially occluded with respect to view v if the occluded region from view v appears in front of the pixel's rasterized depth. A pixel is marked as a possible hidden hole if it is occluded with respect to any input view, i.e., QuadMask flags any pixel that could potentially be a hidden hole. We compute:

$$\text{is_hidden_hole}(p) = \bigvee_v \left(z_p^{\text{occ},v} < z_p^{\text{proj}} \right) \tag{2}$$

This logical OR operation ensures that every possible hidden hole is marked as such. In this way, the mask is a superset of all pixels where adding objects would be consistent with the existing geometry, and the choice of whether to do so depends on the particular scene. QuadMask's augmented inpainting mask finally combines three categories: visible holes, backface pixels, and hidden holes identified through this multi-view reasoning.

3.3 Integration with Reconstruction Pipelines

QuadMask integrates naturally with existing novel view synthesis and 3D reconstruction pipelines by providing an occlusion-aware mask that replaces the naive render mask in downstream processing. See Section 3 of the Supplement for further details.

Occlusion Mask Output. For a novel view at resolution $H \times W$, QuadMask outputs a depth map $\{z_p^*\}$ indicating the minimum occlusion depth at each pixel, along with a binary mask $\{m_p\}$ indicating whether occlusion occurs ($m_p = 1$ if $z_p^* < \infty$, else $m_p = 0$). Additionally, pixels viewing back-facing surfaces or having no valid projected depth from input views also require inpainting.

Integration with Inpainting. The occlusion mask directly indicates which regions require inpainting or object completion. Rather than relying on heuristics, we provide geometric certainty about where information is missing. This enables targeted application of inpainting methods specifically to occluded regions while leaving well-observed regions untouched.

Iterative Refinement. In iterative reconstruction pipelines, geometry is progressively refined over multiple iterations. QuadMask can be re-invoked at each iteration with updated geometry, providing increasingly accurate occlusion masks as reconstruction quality improves. This creates a feedback loop: better geometry $\rightarrow$ better occlusion detection $\rightarrow$ better inpainting $\rightarrow$ better geometry.

Multi-View Consistency. When synthesizing multiple novel views, QuadMask can be applied independently to each view. The resulting occlusion masks are geometrically consistent across views, as they are derived from the same underlying 3D geometry. This consistency is valuable for multi-view refinement strategies and for ensuring temporal coherence in video novel view synthesis.

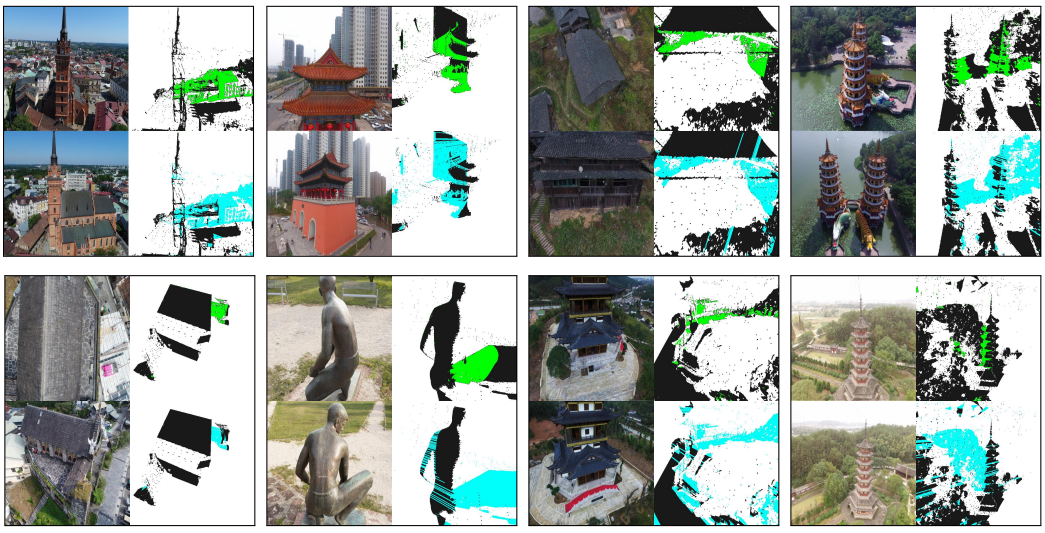


Figure 3: Gallery of qualitative results on view pairs sampled from BlendedMVS scenes. In each example, top left is input view RGB; bottom left is novel view RGB; top right is ground truth mask; and bottom right is QuadMask-generated mask. In each mask, black pixels are not masked, white pixels are visible holes, and colored pixels are hidden holes. See our supplement for more examples.

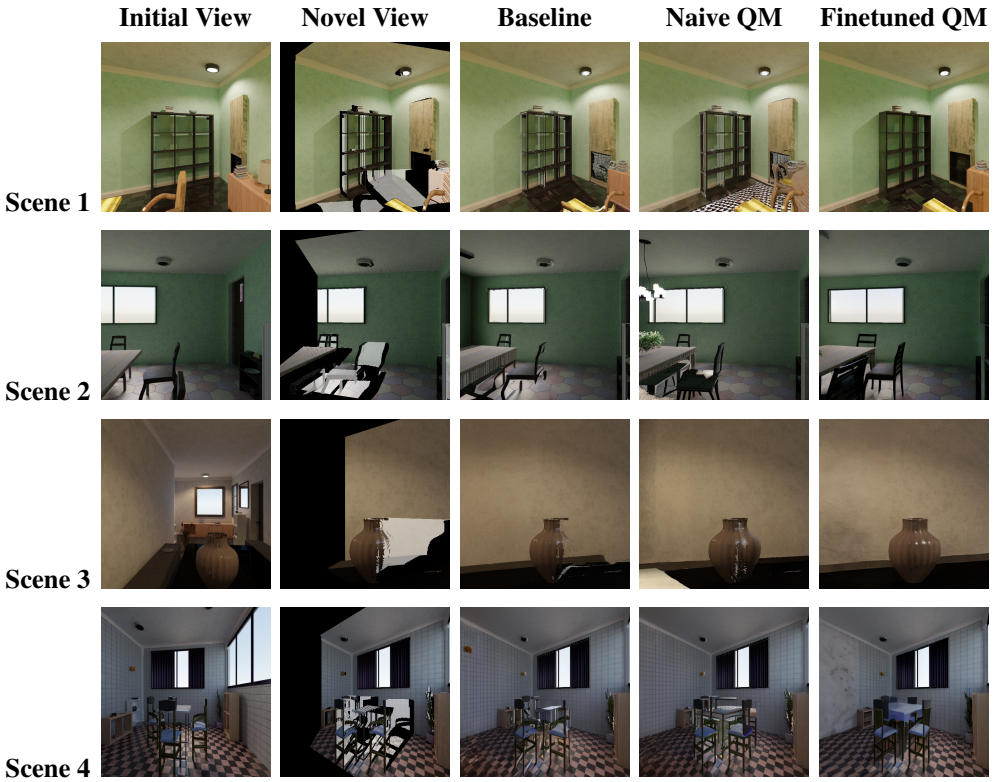


Figure 4: Comparison of inpainting results on four scenes. The first column shows the original image, and the second shows the novel view render with visible holes in black and hidden holes overlaid in white. The Baseline, Naive QM, and Finetuned QM columns follow our experiment naming convention. For each scene and setting, we have chosen the best out of 8 random trials.

4 Experiment Results

Flagging Hidden Holes. First, we show that QuadMask successfully flags hidden holes. We construct ground truth masks of visible and hidden holes for 4.2K+ view pairs sampled across 194 scenes from BlendedMVS+. We project input view depths into the novel view and compare the resulting depth map with the full ground truth novel view depth map. Missing depths in the reprojected map are visible holes, and areas where the reprojected depth is greater than ground truth depth are hidden holes. See Fig. 3 for a qualitative gallery, and our supplement for quantitative metrics.

Iterative Inpainting Pipelines. We demonstrate the modularity of QuadMask by showing quantitative results in conjunction with two iterative inpainting methods: WonderJourney (Yu et al., 2024) and Text2Room (Höllein et al., 2023). While finetuning a generalized hidden-hole aware inpainter is left to future work, for this evaluation, we finetuned a hidden-hole aware SD Inpaint model on simple procedurally generated rooms.

Data Generation. Our data consists of Infinigen rooms with randomized walls, floors, windows, and bookshelves (Raistrick et al., 2024). We randomize bookshelf location, and paired camera poses. Each pose is sampled from a ball of random radius around the center of the bookshelf. The training dataset consists of 64 pose pairs.

Hidden Hole-Aware Model. The model is given the novel pose render of an old view, supervised against a GT view of the novel pose. It is given the usual binary mask, but also hidden holes denoted by 0.5 in the same mask. It was trained for 1500 epochs, with LR 1.0e-5, batch size 64, and a linear warmup of 45 steps.

Evaluation. We evaluate on 166 scene pairs drawn from the same randomized distribution, giving each method the initial view and two camera poses. We tested the following configurations. For

Table 1: Iterative pipeline results with GT initial view depth.

	Text2Room					WonderJourney				
Config	FID ↓	LPIPS ↓	PSNR ↑	Mean CD ↓	Median CD ↓	FID ↓	LPIPS ↓	PSNR ↑	Mean CD ↓	Median CD ↓
Baseline	117.3	0.455	16.1	0.058	5.0×10^{-3}	132.0	0.483	16.3	0.059	6.64×10^{-3}
Naive QM	114.9	0.460	15.9	0.065	7.78×10^{-3}	143.6	0.520	15.1	0.078	1.15×10^{-2}
Finetuned QM	**112.5**	**0.453**	**16.9**	**0.054**	**4.37** $\times 10^{-3}$	**117.6**	**0.481**	**17.1**	**0.051**	**4.40** $\times 10^{-3}$

Table 2: Iterative pipeline results without given depth.

	Text2Room					WonderJourney				
Config	FID ↓	LPIPS ↓	PSNR ↑	Mean CD ↓	Median CD ↓	FID ↓	LPIPS ↓	PSNR ↑	Mean CD ↓	Median CD ↓
Baseline	139.2	0.499	15.7	**0.274**	2.08×10^{-2}	168.3	0.523	16.0	0.527	2.77×10^{-2}
Naive QM	**128.6**	0.501	15.6	0.298	2.07×10^{-2}	157.6	0.539	15.0	0.636	3.25×10^{-2}
Finetuned QM	131.2	**0.498**	**16.3**	0.404	**1.59** $\times 10^{-2}$	**148.0**	**0.518**	**16.6**	**0.468**	**1.95** $\times 10^{-2}$

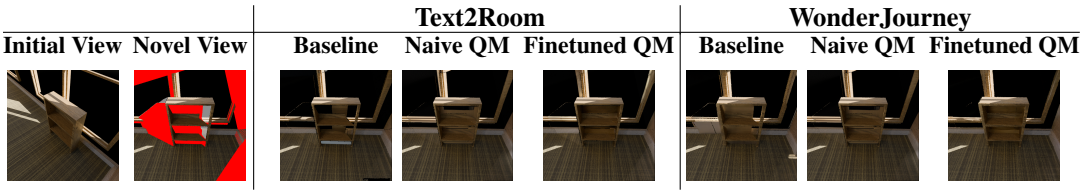


Figure 5: The first image is the starting view for this scene, and the second shows the novel-view render with visible holes in red and hidden holes overlaid in white. The remaining images show the completion attempts from Text2Room and WonderJourney under the baseline, naive QM, and finetuned QM configurations.

Baseline, we allowed the pipeline to inpaint only visible holes (the default setting). For Naive QuadMask, we used off-the-shelf SD Inpaint on the combined visible and hidden hole mask. For Finetuned QuadMask, we used the finetuned model with the combined visible and hidden hole mask. Figure 4 shows examples of these inpainting settings on four synthetic scenes.

In our evaluation, we modify the scene generation methods Text2Room and WonderJourney. For Baseline, the pipelines iteratively add visible hole content to their 3D structures (the usual behavior). For Naive QuadMask and Finetuned QuadMask, we use the inpainting procedures above, and add new content from both visible and hidden holes into the 3D structure. For Table 1, the methods get ground truth depth at the initial view. For Table 2, we replaced the baseline depth predictors with MoGe and Marigold-DC. A visual example of the given-depth pipelines on one scene can be seen in Figure 5.

To evaluate, we render the 3D structure from each method at the novel view poses. We compare the rendered novel views and the overall GT distribution for FID. We compare the novel view renders to their corresponding GT views for LPIPS and PSNR. Finally, for CD, we compare the generated 3D structures to ground truth 3D based on the two views, subsampling each structure to 100k points or vertices. We discount sky depth outliers, defined as points at least 20 meters away from the cameras.

We find that QuadMask with a hidden hole-aware inpainter helps iterative pipeline performance. While Naive QuadMask often suffers from removal of context, accounting for hidden holes when inpainting allows perceptually plausible completions. Median CD furthermore shows that using the hidden hole-aware inpainter tends to have a geometry gain.

5 Future Work

Our results demonstrate the use of QuadMask in hidden hole-aware iterative inpainting pipelines. A promising line of future work is to train a more generalized inpainter, which would allow large-scale 3D generation using iterative inpainting methods.

AI USE STATEMENT

In this work, we used generative AI tools for assistance of the implementation of parts of QuadMask's GPU kernel, as well as finetuning code, functions to run the procedural data generator, and code to run experiments. We have not used generative AI tools for generating synthetic data sets, assisting in the writing of proofs, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, supporting qualitative and thematic data analysis, or interpreting results. Helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting with translation, and cleaning or reformatting datasets are not applicable to this work.

Additionally, we used generative AI tools for assisting with creating paper figures. We have reviewed all AI-assisted work. Specifically, we used generative AI tools to assist with writing Blender code to create 3D visualizations for some paper figures. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Jaeyoung Chung, Suyoung Lee, Hyeongjin Nam, Jaerin Lee, and Kyoung Mu Lee. Luciddreamer: Domain-free generation of 3d gaussian splatting scenes, 2023. URL https://arxiv.org/abs/2311.13384.

Franklin C. Crow. Shadow algorithms for computer graphics. In James George (ed.), *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1977, San Jose, California, USA, July 20-22, 1977*, pp. 242–248. ACM, 1977. doi: 10.1145/563858.563901. URL https://doi.org/10.1145/563858.563901.

Manuel Dahnert, Angela Dai, Norman Müller, and Matthias Nießner. Coherent 3d scene diffusion from a single rgb image, 2024. URL https://arxiv.org/abs/2412.10294.

Ruiqi Gao, Aleksander Holynski, Philipp Henzler, Arthur Brussee, Ricardo Martin-Brualla, Pratul Srinivasan, Jonathan T. Barron, and Ben Poole. Cat3d: Create anything in 3d with multi-view diffusion models, 2024. URL https://arxiv.org/abs/2405.10314.

Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–24, 1984. doi: 10.1109/MCG.1984.6429331. URL https://doi.org/10.1109/MCG.1984.6429331.

Ned Greene, Michael Kass, and Gavin S. P. Miller. Hierarchical z-buffer visibility. In Mary C. Whitton (ed.), *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1993, Anaheim, CA, USA, August 2-6, 1993*, pp. 231–238. ACM, 1993. doi: 10.1145/166117.166147. URL https://doi.org/10.1145/166117.166147.

Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In Hank Christiansen (ed.), *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*, pp. 119–127. ACM, 1984. doi: 10.1145/800031.808588. URL https://doi.org/10.1145/800031.808588.

Lukas Höllein, Ang Cao, Andrew Owens, Justin Johnson, and Matthias Nießner. Text2room: Extracting textured 3d meshes from 2d text-to-image models, 2023. URL https://arxiv.org/abs/2303.11989.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023. URL https://arxiv.org/abs/2308.04079.

Jiabao Lei, Jiapeng Tang, and Kui Jia. Rgbd2: Generative scene synthesis via incremental view inpainting using rgbd diffusion models, 2023. URL https://arxiv.org/abs/2212.05993.

Ming-Feng Li, Yueh-Feng Ku, Hong-Xuan Yen, Chi Liu, Yu-Lun Liu, Albert Y. C. Chen, Cheng-Hao Kuo, and Min Sun. Genrc: Generative 3d room completion from sparse image collections, 2024. URL https://arxiv.org/abs/2407.12939.

Hanwen Liang, Junli Cao, Vidit Goel, Guocheng Qian, Sergei Korolev, Demetri Terzopoulos, Konstantinos N. Plataniotis, Sergey Tulyakov, and Jian Ren. Wonderland: Navigating 3d scenes from a single image, 2025. URL https://arxiv.org/abs/2412.12091.

Michael D. McCool. Shadow volume reconstruction from depth maps. *ACM Trans. Graph.*, 19(1): 1–26, 2000. doi: 10.1145/343002.343006. URL https://doi.org/10.1145/343002.343006.

Yanxu Meng, Haoning Wu, Ya Zhang, and Weidi Xie. Scenegen: Single-image 3d scene generation in one feedforward pass, 2025. URL https://arxiv.org/abs/2508.15769.

Norman Müller, Katja Schwarz, Barbara Roessle, Lorenzo Porzi, Samuel Rota Bulò, Matthias Nießner, and Peter Kontschieder. Multidiff: Consistent novel view synthesis from a single image, 2024. URL https://arxiv.org/abs/2406.18524.

Alexander Raistrick, Lingjie Mei, Karhan Kayan, David Yan, Yiming Zuo, Beining Han, Hongyu Wen, Meenal Parakh, Stamatis Alexandropoulos, Lahav Lipson, Zeyu Ma, and Jia Deng. Infinigen indoors: Photorealistic indoor scenes using procedural generation, 2024. URL https://arxiv.org/abs/2406.11824.

Ahmad Salimi, Tristan Aumentado-Armstrong, Marcus A. Brubaker, and Konstantinos G. Derpanis. Geometry-aware diffusion models for multiview scene inpainting, 2025. URL https://arxiv.org/abs/2502.13335.

Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984. doi: 10.1145/356924.356930. URL https://doi.org/10.1145/356924.356930.

Jaidev Shriram, Alex Trevithick, Lingjie Liu, and Ravi Ramamoorthi. Realmdreamer: Text-driven 3d scene generation with inpainting and depth diffusion, 2025. URL https://arxiv.org/abs/2404.07199.

Stanislaw Szymanowicz, Eldar Insafutdinov, Chuanxia Zheng, Dylan Campbell, João F. Henriques, Christian Rupprecht, and Andrea Vedaldi. Flash3d: Feed-forward generalisable 3d scene reconstruction from a single image, 2025a. URL https://arxiv.org/abs/2406.04343.

Stanislaw Szymanowicz, Jason Y. Zhang, Pratul Srinivasan, Ruiqi Gao, Arthur Brussee, Aleksander Holynski, Ricardo Martin-Brualla, Jonathan T. Barron, and Philipp Henzler. Bolt3d: Generating 3d scenes in seconds, 2025b. URL https://arxiv.org/abs/2503.14445.

Hong-Xing Yu, Haoyi Duan, Junhwa Hur, Kyle Sargent, Michael Rubinstein, William T. Freeman, Forrester Cole, Deqing Sun, Noah Snavely, Jiajun Wu, and Charles Herrmann. Wonderjourney: Going from anywhere to everywhere, 2024. URL https://arxiv.org/abs/2312.03884.

Hong-Xing Yu, Haoyi Duan, Charles Herrmann, William T. Freeman, and Jiajun Wu. Wonderworld: Interactive 3d scene generation from a single image, 2025. URL https://arxiv.org/abs/2406.09394.