WorldGen: From Text to Traversable and Interactive 3D Worlds

Supplementary Material

6. Videos

We encourage readers to view the videos for a better sense of the scale and complexity of the scenes generated by WorldGen. In particular, we import the generated scenes into a game engine and demonstrate a character traversing each environment. We decimate the meshes to cap the total number of vertices at 400K, as imposed by the game engine, but we do not compress the textures.

7. Stage I: Scene Planning Details

7.1. Procedural Blockout Generation

Traditional procedural generation (PG) produces coherent and functional environments based on hand-crafted rules and procedures, but it can handle only a narrow range of environments and cannot be controlled using natural-language prompts. Inspired by recent text-conditioned procedural systems [48, 62], we extend a PG system with a language interface. An LLM parses the user prompt y into a structured JSON specification of parameters such as terrain type, object density, verticality, and placement regularity. These parameters configure a modular PG pipeline that procedurally constructs a blockout B aligned with the user's intent.

In more detail, our procedural generation (PG) pipeline constructs the blockout in three steps: *terrain generation*, *spatial partitioning*, and *hierarchical asset placement*. First, *terrain generation* constructs a base landscape that defines the large-scale geometry of the scene—such as elevation, slopes, and flat regions—providing a base for where structures and traversal paths can exist. Next, *spatial partitioning* divides the terrain into distinct regions that serve different scene purposes (e.g., open areas, clusters of structures, or transition zones). This step provides a high-level organizational layout, ensuring that the scene has variation in density and structure while maintaining overall navigability. Finally, *hierarchical asset placement* populates each region with 3D assets in multiple passes: large landmark assets are placed first to establish structure and focal points, followed by smaller objects and decorative elements that add realism and detail. This multi-level placement strategy produces consistent yet varied scenes, maintaining both functional organization and visual diversity.

(1) Terrain Generation. We synthesize the terrain using either a Perlin-noise generator [60] or a rule-based height map configured by the parsed JSON specification. The JSON parameters further define terrain attributes such as type (e.g., "flat", "steep"), surface roughness, and elevation range, which together control the overall topography and structural variation of the scene.

(2) Spatial Partitioning. Spatial partitioning divides the terrain into distinct regions that provide structural organization for the scene. This step determines where dense clusters, open areas, and transitional zones appear. For structured environments (e.g., "urban," "grid village"), we employ binary space partitioning [17], uniform grids, or k-d trees [5] to produce regular, orthogonal layouts. For natural or irregular landscapes (e.g., "archipelago," "jungle"), we use Voronoi diagrams, noise-based partitions, or Drunkard's Walk [59] to create organic, non-uniform boundaries. This process defines the macro-level organization of the environment, balancing structured regions with open space to ensure both navigability and visual diversity.

(3) Hierarchical Asset Placement. Finally, we populate the layout with blocks, which serve as placeholders for different categories of elements, in three passes to reflect structural hierarchy and spatial semantics. (i) *Hero assets* (major landmarks or buildings) are placed first. (ii) *Medium-scale elements* such as trees, walls, or bridges are distributed relative to hero assets. (iii) *Small decorative assets* fill residual spaces according to density and clustering parameters. A final terrain-smoothing step prevents asset collisions, improving realism and playability of the blockout geometry. While we use categories (i-iii) to generate a reasonable distribution of volumes, we do not make hard decisions on what these represent at this point; instead, we let the image generator decide what they are.

The resulting blockout B above is a 3D mesh composed of simple primitives (ground plane and boxes) that encode the essential geometry of the scene. It serves as an editable structural scaffold from which we subsequently derive the navmesh S and reference image $\mathbf{R}$.

7.2. Planning Stage Results

In Figure 12, we show several representative examples generated by our text-conditioned layout module. Each example illustrates different combinations of terrain types, verticality and object density—the primary factors governing scene structure and downstream difficulty. The explicit procedural generation process guarantees that the entire area is navigable.

The first column depicts low-density layouts with various terrain types, producing open and easily traversable spaces. The second column introduces medium-density object placement with diverse verticality changes, leading to a richer spatial composition that includes both open grounds and dense activity areas. The third column features a dense asset distribution, which results in a complex environment.

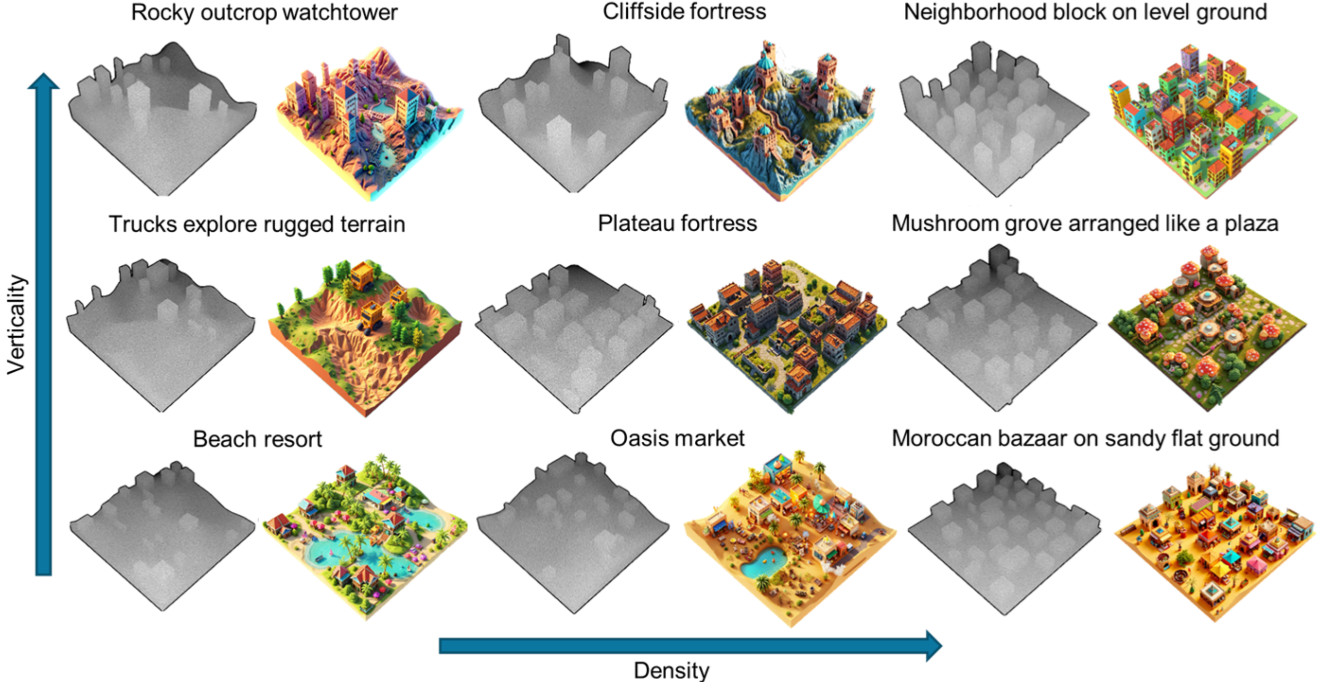


Figure 12. **Depth-conditioned generation across density (columns) and verticality range (rows).** In each grid cell, we show the input depth map (left) and the corresponding generated image conditioned on that depth (right). Columns are ordered by increasing density from left to right; rows are ordered by increasing verticality range from low to high.

8. Stage II: Scene Reconstruction Details

8.1. Image-to-3D Base Model

Our shape generator adopts the popular *VecSet* [86] representation for 3D diffusion modeling, where a scene or object is represented as an unordered set of latent vectors. The diffusion model learns to denoise these vector sets to reconstruct signed distance fields (SDFs) conditioned on the input image.

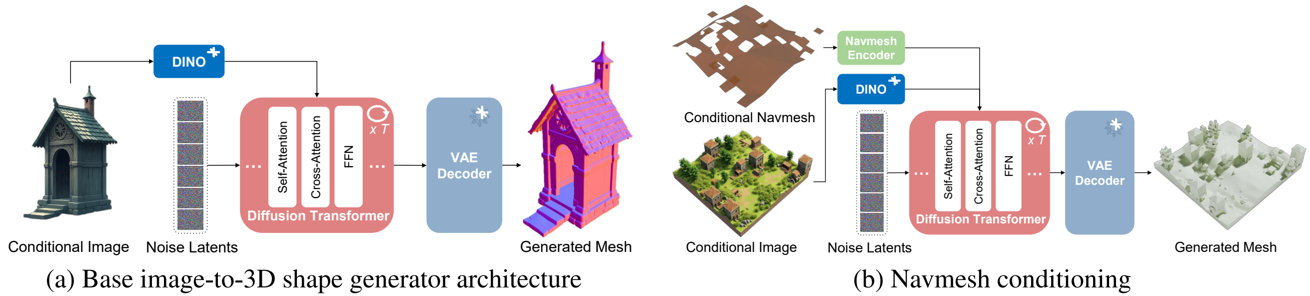
VecSet Latent Representation. VecSet learns a 3D latent space for compact object representation using an autoencoder. Given a 3D object $\boldsymbol{x}$ represented by a point cloud $\mathcal{P} = \{(p_i, n_i)\}_{i=1}^{M}$, with points $p_i \in \mathbb{R}^3$ and normals $n_i \in \mathbb{S}^2$, the encoder $\mathcal{E}$ maps $\mathcal{P}$ to a latent code $\boldsymbol{z} \in \mathbb{R}^{K \times D}$ consisting of K D-dimensional tokens: $\boldsymbol{z} = \mathcal{E}(\mathcal{P})$. The decoder $\mathcal{D}$ reconstructs the signed distance function (SDF) of the object, assigning each query location $q \in \mathbb{R}^3$ with an SDF value $\mathcal{D}(q \mid \boldsymbol{z}) \in \mathbb{R}$. The encoder starts by randomly downsampling the input point cloud to K points, one per token, extracting a subset $\hat{\mathcal{P}} = \text{FPS}(\mathcal{P} \mid K) = \{\hat{p}_1, \ldots, \hat{p}_K\}$ using farthest point sampling (FPS). The encoder then projects the full point clouds (with normals) $\mathcal{P}$ to the selected points $\hat{\mathcal{P}}$ using sinusoidal spatial encoding followed by cross-attention and several standard transformer layers until the final code $\boldsymbol{z}$ is obtained. The decoder operates in 'reverse', taking as input a query point q in order to compute the corresponding SDF value. The resulting latent tokens form a compact, permutation-invariant 3D representation suitable for diffusion-based generation.

Image-to-3D Latent Diffusion Model. Our shape generator learns a diffusion model that generates 3D object latents conditioned on an input image $\mathbf{I}$. Specifically, it models the conditional distribution of latent codes as $p(\boldsymbol{z} \mid \mathbf{I}; \Phi)$ and learns it via a denoising diffusion process parameterized by a transformer Φ. The resulting latent $\boldsymbol{z}$ defines a signed distance field (SDF), from which a watertight triangular mesh is extracted using Marching Cubes [47] at 512^3 grid. An overview of the model architecture is shown in Figure 13(a).

Training. We train the VecSet autoencoder and the image-to-3D diffusion model in our shape generator using an internal dataset of artist-authored 3D assets.

8.2. Navmesh Conditioning

We include an ablation study using a baseline image-to-3D model trained on our curated dataset of scene triplets. The quantitative comparisons demonstrate that our method achieves stronger spatial alignment with the input navigation conditions, validating the importance of combining the curated triplets with the navmesh condition. Moreover, when evaluated on 256 unseen procedurally generated 3D scenes, **our output 3D scene mesh achieves a Chamfer distance of 0.036, compared to 0.038 for the image-to-**



(a) Base image-to-3D shape generator architecture (b) Navmesh conditioning

Figure 13. **Base shape generator and Navmesh architectures.** Left: Overview of the base shape generation architecture. Right: Our Navmesh conditioned scene mesh generation (Stage II) based on cross-attention.

Table 3. **Quantitative evaluation of navmesh alignment on various scene types.** Our model shows stronger spatial alignment with the input navigation conditions across various types of scenes compared to the image-to-3D shape generator baseline.

Model	Flat Terrain	Mountainous	Sparse Objects
Baseline	0.036	0.045	0.032
Ours	**0.021**	**0.038**	**0.026**

3D baseline. This result indicates that our approach not only ensures better navmesh alignment but also yields superior overall mesh quality.

Moreover, we evaluate our model against the baseline image-to-3D model on a diverse curated scene benchmark generated via our procedural approach, as shown in Tab. 3. This benchmark comprises three distinct scene categories: (1) 50 low-verticality scenes (flat ground) with high object density (10–30 items); (2) 40 high-verticality scenes (complex mountainous terrain) with sparse object placement (average 10 items); and (3) 50 low-verticality scenes with sparse density ($<$ 10 items). Quantitative evaluation demonstrates that our approach consistently outperforms the baseline across these varying scene complexities.

9. Stage IV: Scene Enhancement Details

9.1. Per-object Image Enhancement with LLM-VLM

To inform the LLM-VLM image generator of the global scene context, we render a top-down view of the entire environment from M, highlighting the target object in red. This visualization, together with the global reference image $\mathbf{R}$, is input to a large language-vision model which identifies the object's location and analyzes its visual attributes, such as material and color palette. Then, for each object $\hat{\boldsymbol{x}}_i$, we render a view $\hat{\mathbf{I}}_i$ that captures its coarse geometry and low-resolution texture, serving as input for image enhancement (Figure 7, third row).

To evaluate the impact of our image-enhancement strategy, we present ablation results without the top-down view in Fig. 14. In this case, the LLM-VLM only receives $\mathbf{R}$ and $\hat{\mathbf{I}}_i$ from each object to enhance. As shown in Figure 14, the LLM-VLM struggles to generate style-consistent or reference-faithful object images when conditioned solely on the global reference image. This highlights the importance of including a top-down view with the target object highlighted, as it provides essential context about the object's location, semantics, and surroundings within the scene.

9.2. Per-Object Mesh Enhancement

We provide a quick summary of the mesh refinement model [57], its architecture (see Figure 15). Given a coarse object mesh $\hat{\boldsymbol{x}}_i$ and a high-resolution image $\mathbf{I}_i$ from the *Image Enhancement* step as input conditions, our *Mesh Refinement Model* is trained to generate a high-resolution object mesh $\boldsymbol{x}_i$ that preserves the orientation of the coarse mesh while adding fine geometric details.

9.3. Per-Object Texture Enhancement

We finally generate high-resolution textures for each object $\boldsymbol{x}_i$ based on its enhanced image $\mathbf{I}_i$ and the super-resolved geometry.

Following an established paradigm for texture generation [4], we fine-tune a pretrained text-to-image latent diffusion model to produce 3D-consistent multi-view renderings of the object conditioned on normal and position maps for the target views, as well as the delighted version of an enhanced image $\mathbf{I}_i$. We backproject the generated multi-view images to UV map to get the final texture image.

Delighting the Conditioning Image. Since the enhanced object image $\mathbf{I}_i$ contains baked-in lighting and shading effects, it often exhibits complex illumination patterns, shadows, and specular highlights. To mitigate this, we train a delighting model by fine-tuning a text-to-image latent diffusion model, where the latent representation of the shaded input image is provided as an in-context conditioning signal for generation.

Generating Multi-view Images. We build upon Meta 3D TextureGen [4] with the following design choices. First, we

Figure 14. **Per-object image enhancement without using the top-down view.** Without access to a top-down view of the entire scene—which gives the LLM-VLM important information about object location and surrounding context—the model has difficulty generating object images that are visually consistent with the scene's style or faithful to the reference image. As a result, the generated images may not match the overall style of the scene or may differ from the appearance of the object in the reference image.

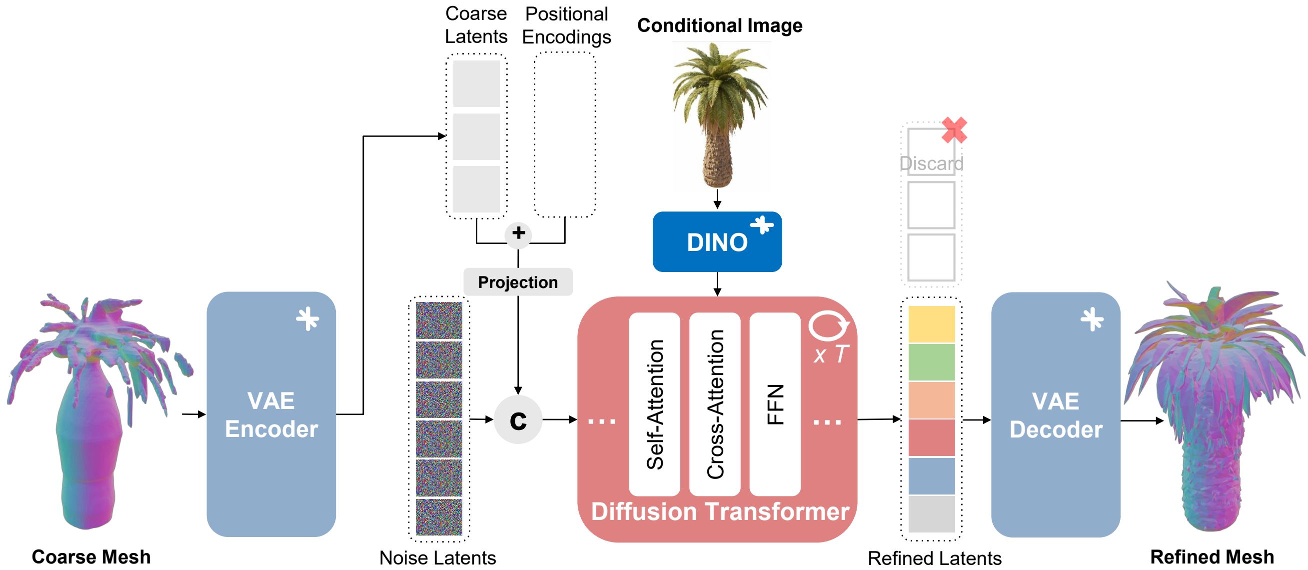


Figure 15. **Per-object mesh refinement**. Given a coarse object mesh and a high-resolution image, we feed them to our mesh refinement model which outputs a refined high-quality mesh that adheres to the orientation and shape of the coarse input yet incorporates fine details from the image input.

make the generator conditioned on the image. The latent of the condition image is supplied as an in-context input to guide the generation process. Second, we generate ten orthographic multi-view images, including eight side views

Figure 16. Illustration of multi-view texture generation. Given a reference image as input, we sequentially generate: (1) frontal views, (2) side views conditioned on the frontal view, and (3) top and (4) bottom views conditioned on all previously generated views.

evenly spaced at 45° around the object at 0° elevation, along with top and bottom views (see Figure 16). Third, we employ sequential generation strategy, where we first generate the frontal view, then side views, and finally the top and bottom views. We empirically find that this improves the cross-view coherence and reduces geometric distortion.

Disentangled Multi-View Attention. We employ disentangled attention, where the self-attention block is decomposed into three attention modules—in-plane self-attention, reference attention, and multi-view attention. The first is *in-plane self-attention*, where each view independently attends to its own spatial features, preserving local coherence and detail within individual renderings. The second is *reference attention*, where the generated views (i.e., views 1 to $N-1$) attend to the reference view (i.e., view 0) via cross-attention, ensuring all synthesized views remain consistent with the input enhanced image. The third is *multi-view attention*, where the generated views attend to each other, promoting global 3D consistency across different viewpoints while maintaining strong adherence to the reference image. This factorization enables more structured feature interactions across views.

Texture Post-processing. Once the ten views are generated, we initialize the UV texture by back-projecting the multi-view images onto the object's surface. This step yields sharp and well-aligned textures for all regions visible in at least one generated view. Finally, we apply an inpainting algorithm in UV space to fill small gaps and un-

Figure 17. **Comparison with SynCity [14].** Left: WorldGen. Right: SynCity. Our method produces scenes with higher geometric detail, improved texture fidelity, and more consistent global structure.

observed areas, producing complete, high-quality textures ready for scene assembly.

10. Comparison with SynCity

To further demonstrate the capabilities of our approach compared to recent scene-level generation baselines, we provide a qualitative comparison with SynCity [14] in Fig. 17.

Our method differs from SynCity in several aspects related to scene structure, usability, and rendering representation. First, SynCity adopts a tile-based generation paradigm that does not explicitly consider navigability, which is an important requirement for game-ready environment synthesis. In contrast, our pipeline is designed to generate navigable scenes, enabling the creation of large-scale environments that can be explored in a coherent manner.

Second, the tile-based formulation in SynCity can introduce boundary artifacts and makes it challenging to generate objects that extend across multiple tiles. Our approach does not impose such constraints and is able to synthesize scene elements that span larger spatial regions while maintaining structural continuity.

Third, SynCity generates content primarily at the tile level without explicit object-level decomposition. This limits the ability to perform fine-grained scene editing and manipulation. In comparison, our method preserves object-level structure, which facilitates flexible editing operations such as modifying geometry, appearance, and layout.

Fourth, due to its patch-based generation process, SynCity does not explicitly enforce global consistency across the entire scene. Our pipeline incorporates mechanisms to promote global coherence, resulting in environments with more consistent geometry, spatial organization, and visual appearance.

Finally, the rendering representations of the two approaches differ substantially. SynCity produces Gaussian splatting representations, which are not yet directly compatible with standard real-time rendering engines. Our method instead outputs explicit textured 3D meshes that can be readily integrated into conventional graphics pipelines.

Because SynCity operates under different assumptions and representations, a direct output-to-output comparison from identical inputs is not feasible. To enable a meaningful qualitative comparison, we start from a scene generated by SynCity, render a holistic image of the scene, and then apply our pipeline to reconstruct the environment from this image. As shown in Figure 17, our approach produces scenes with improved geometric detail, texture fidelity, and overall structural coherence.

11. Style consistency

To further demonstrate the advantages of our method in terms of stylistic coherence, we conduct a quantitative evaluation. Since prior approaches do not provide explicit object-wise representations, direct object-level comparison is not feasible. Instead, we evaluate style consistency at the scene level by computing CLIP similarity between generated frames and both the input text prompt and reference images.

We use frames from the *medieval village* scene, and measure (i) text–image alignment and (ii) image–image similarity as proxies for overall stylistic consistency. Higher similarity indicates better alignment with the intended style.

As shown in Table 4, our method consistently outperforms state-of-the-art text/image-to-3D approaches (Models A and B) across both metrics.

Table 4. **Scene-level style consistency comparison.**

Method	Text Similarity ↑	Image Similarity ↑
Ours	**0.1515**	**0.6860**
Model A	0.1349	0.6455
Model B	0.1242	0.6230

These results indicate that our approach generates scenes that are more faithful to both the textual description and visual references, demonstrating stronger global style consistency without relying on explicit object-level supervision.

12. Limitations and Future Work

As briefly discussed in the main paper, our current pipeline relies on a single reference view, which inherently limits generation to bounded, predominantly single-story environments. This restricts applicability to more complex scenarios such as unbounded outdoor worlds or multi-level structures with significant vertical interactions.

In addition, our holistic scene planning and generation strategy, while effective for producing globally coherent results, becomes a bottleneck when scaling to larger environments. Extending to unbounded scenes will likely require a transition to more scalable paradigms, such as sliding-window or chunk-based generation, where local regions are

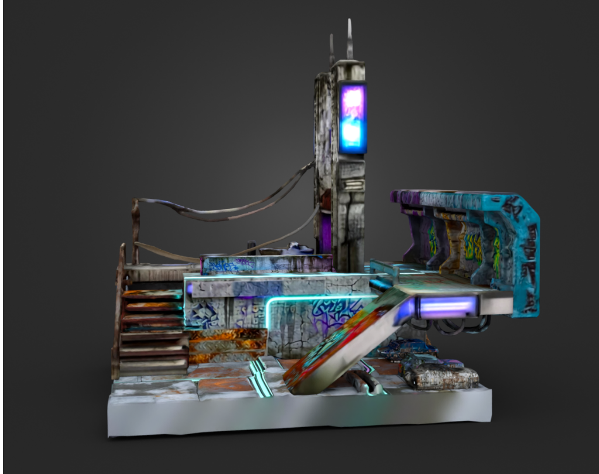


Figure 18. **Multi-level platforms example.**

synthesized incrementally while maintaining global consistency.

For large environments, however, the primary bottleneck shifts from generation itself to efficient representation. In particular, geometry and texture reuse is largely unexplored in current generative pipelines and becomes critical for both memory efficiency and real-time rendering. The absence of asset instancing (i.e., reuse of repeated structures) further exacerbates this issue, especially in dense scenes, leading to redundant geometry and increased rendering cost.

Future work should address these challenges along multiple directions. First, incorporating incremental or streaming generation strategies [14] can enable scalable synthesis of large environments. Second, introducing explicit mechanisms for material and geometry reuse, such as instance-aware generation or library-based asset retrieval, could significantly improve both efficiency and visual consistency.

Finally, to broaden the applicability of our approach, we are actively extending the pipeline to indoor environments, which introduce additional challenges such as tighter spatial constraints and more intricate structural relationships. In Figure 18, we present preliminary results demonstrating early progress in generating complex indoor layouts, including structures such as stairs and multi-level platforms.