PROCFUNC: FUNCTION-ORIENTED ABSTRACTIONS FOR PROCEDURAL 3D GENERATION IN PYTHON

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce ProcFunc, a library for Blender-based procedural 3D generation in Python. ProcFunc provides a library of easy-to-use Python functions, which streamline creating, combining, analyzing, and executing procedural generation code. ProcFunc makes it easy to create large-scale diverse training data, by combinatorial composition of semantic components. VLMs can use ProcFunc to edit procedural material and geometry code and can create new procedural code with significantly fewer coding errors. Finally, as an example use case, we use ProcFunc to develop a new procedural generator of indoor rooms, which includes a collection of new compositional procedural materials. We demonstrate the detail, runtime efficiency, and diversity of this room generator, as well as its use for 3D synthetic data generation.

1 INTRODUCTION

Procedural 3D generation refers to the creation of 3D assets—objects, scenes, materials, textures, animations—through algorithms and compact mathematical rules, in other words, human-understandable code. For example, the 3D mesh of a tree can be created by randomized recursive branching starting from a base cylinder; the texture map of the tree bark can be created as a simple function of a grid of random 2D vectors (Perlin, 1985).

The primary advantages of procedural generation are control and composition. In essence, procedural generation represents 3D assets as code; it thus allows precise and fine-grained control, from the curvature of a tree leaf to the layout of a forest. Because procedural generators are code, they can be modular and composable. By composing modules and varying control parameters, procedural generators can create unlimited, diverse, varied, and photorealistic 3D assets, as demonstrated by Infinigen (Raistrick et al., 2023; 2024), a Blender-based procedural system. Procedural generation has proven useful in various ways, including creating synthetic training data (Yan et al., 2026), evaluation (Nugent et al., 2025; Wang et al., 2025), and AI-assisted content creation (Huang et al., 2024; Lu et al., 2025).

Because procedural generators are code, the development of procedural generators can benefit greatly from well-designed and easy-to-use programming abstractions, just as deep learning has benefited greatly from libraries such as TensorFlow, PyTorch, and JAX. A good programming abstraction can reduce implementation complexity, improve development efficiency, and enable new applications.

A good programming abstraction for procedural generation should allow efficient creation and composition of procedural generators in the form of compute graphs that define shape or texture, similar to how PyTorch can define and combine neural modules through a few lines of code. Also similar to PyTorch, it would be highly useful to be able to easily access, modify, analyze the control parameters to enable precise control and domain randomization, which would be useful in both content creation and training data generation.

However, existing open-source programming interfaces for procedural generation fall short of these desiderata. For example, composing mathematical functions is a basic operation in procedural generation, but often requires lengthy code to implement using Blender's native Python API. It takes multiple steps to create a gadget and connect its input and output sockets, and particular sockets may appear or disappear depending on the gadget state and may have dynamic indices and duplicated names. To be clear, this does not mean that the Blender API is flawed, just that it is designed for a

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.

Material Color – Blender API

```
mix = ng.nodes.new(type="ShaderNodeMixRGB")
ng.links.new(
    base_color_1.outputs["Color"], mix.inputs[6]
)
ng.links.new(
    value.outputs["Value"], mix.inputs[0]
)
mix.inputs[7].default_value = (1.0, 1.0, 1.0, 1.0)
mix.data_type = "RGBA"
```

Material Color – Infinigen API

```
mix = nw.new_node(
    Nodes.Mix,
    input_kwargs = {
        0: value.outputs["Value"],
        6: base_color.outputs["Color"],
        7: (1.0, 1.0, 1.0, 1.0)
    }
)
```

Material Color – ProcFunc

```
mix = pf.nodes.func.mix_rgb(
    factor=value,
    a=base_color,
    b=(1.0, 1.0, 1.0)
)
```

Mesh Extrude – Blender API

```
# SELECT OBJECT
for obj in bpy.data.objects:
    obj.select_set(False)
bpy.data.objects["Cube"].select_set(True)

# SELECT POLYGONS
for i, p in enumerate(target.data.polygons):
    p.select = mask[i]
bpy.ops.object.mode_set(mode="EDIT")

# EXTRUDE
bpy.ops.mesh.extrude_region_move(
    TRANSFORM_OT_translate={"value": (0, 0, 1)}
)
bpy.ops.object.mode_set(mode="OBJECT")
```

Mesh Extrude – Infinigen API

```
# SELECT OBJECT
with butil.SelectObjects(cube):

    # SELECT POLYGONS
    for i, p in enumerate(cube.data.polygons):
        p.select = mask[i]

    with butil.ViewportMode(cube, "EDIT"):
            bpy.ops.mesh.extrude_region_move(
                TRANSFORM_OT_translate={
                    "value": (0, 0, 1)
                }
            )
```

Mesh Extrude – ProcFunc

```
pf.ops.mesh.extrude_faces(
    cube, (0, 0, 1), mask=mask
)
```

Figure 1: **ProcFunc Primitives API** - ProcFunc provides a new API for low-level graphics operations which eliminates all non-essential or GUI-oriented complexity. Typical usage of the Blender and Infinigen API requires separate selection and action steps to manipulate Blender's global background state. Blender and Infinigen require unclear runtime keys such as "0, 6, 7" in the top row, which change dynamically depending on other arguments or program state. ProcFunc instead provides each operation as an atomic Python function with all inputs as explicit arguments.

different purpose, namely to facilitate a GUI-based workflow: these steps are intuitive for artists and GUI-centric plugins, but can be significantly simplified for Python-centric development.

Infinigen's code streamlines some of these steps, but does not fundamentally reduce their complexity. Each operation still requires multiple steps (albeit wrapped by new functions), and still depends on dynamically changing UI strings (Fig. 1). This leads to complex procedural code, where one function can fail silently due to the global state configured by a previous function, and modules are fragile and hard to combine. In addition, the control parameters lack standardization and are hard to access, making it difficult to perform arbitrary customization and randomization.

In this work, we introduce ProcFunc, a library for efficient creation, composition, analysis and execution of Blender-based procedural generators. ProcFunc automates common tasks in creating procedural generators, and makes generators easy to combine and re-parameterize. Procedural generators using ProcFunc support many research tasks without complex per-asset implementation: we provide tools which can automatically analyze generators to access their parameters and composition, access intermediate values, and analyze or optimize their overall random distribution. We aim to streamline the overall process of procedural dataset generation, much in the same way that deep learning libraries (Abadi et al., 2016; Paszke et al., 2019; Bradbury et al., 2018) have streamlined the process of creating and using deep learning models.

ProcFunc is particularly useful in increasing *compositionality*. Compositionality is the degree to which a system can be decomposed into a hierarchy of reusable parts, and recombined (composed) into valid and useful combinations. This is helpful for procedural generation because recombining components can drastically increase visual diversity and coverage (Fig. 2).

Compositionality and ease of use depend on newly designed APIs at three levels. First is simple, explicit functions for all graphics operations in Blender, which significantly simplify Python-oriented procedural generation, and can be used for individual asset generation or content creation. Second is a design pattern for module-level composition of procedural generators, which provides individual assets as off-the-shelf functions with either precise control or automatic randomization, and in some cases can be statically analyzed using a compute graph tracer. Finally, building on top of ProcFunc,

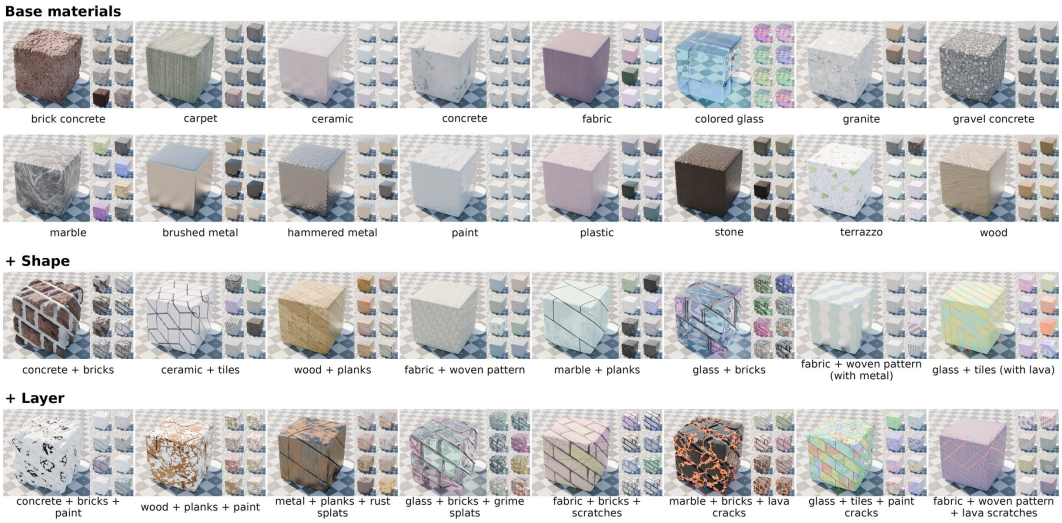


Figure 2: **Compositional Procedural Generation** - ProcFunc enables reusable procedural generators which can form countless semantically meaningful combinations. For materials, one can create bricks, tiles, paint and scratches combined with any material, forming tens of thousands of combinations.

we provide a library of high-quality procedural generators. It features compositional materials (Fig. 2), as well as simple tools for object arrangement and collisions which can be used to efficiently create full rooms (Fig. 3) or user-designed scenes.

We evaluate ProcFunc's effectiveness for its two primary applications: procedural generation for content creation, and useful procedural data *distributions* to be sampled for synthetic data. For content creation, we use BlenderGym (Gu et al., 2025)'s material and geometry tasks. ProcFunc is helpful in both the default parameter-editing setting and a new from-scratch generation setting, albeit with a non-standard harness. For synthetic dataset generation, we demonstrate our library of components via a new diverse and high-quality compositional indoor room generator. We find it provides greater analytical diversity and semantic coverage due to composition, offers a favorable performance-complexity tradeoff, and is useful in creating large synthetic datasets.

In summary, we provide ProcFunc, a system which substantially reduces the complexity of creating and combining procedural generators. Using it, we create an indoor room generator which has diverse, detailed and compositional materials, and is efficient and useful for dataset generation.

2 Related Work

Procedural Dataset Generation Computer vision has benefited from procedural generation of 3D assets at various levels, including scene-level procedural arrangement of fixed objects (Deitke et al., 2022), procedural variation within objects (Eppner et al., 2025; Sun et al., 2025c), and fully procedural scenes (Raistrick et al., 2023; 2024). These existing procedural generators are useful and complementary to our goal of making further development of new procedural generators more efficient. Our APIs enable the creation of new material and object generators, which is distinct from systems which easily create datasets from existing assets (Denninger et al., 2019; Greff et al., 2022).

Domain-specific Languages for Procedural Generation Languages for scene generation and procedural content creation vary in scope. The Scene Language (Zhang et al., 2025) and Procedural Scene Programs (Gumin et al., 2025) create useful scene-level arrangements, including control flow and symmetry. However, these rely on pre-made non-procedural object and material assets, whereas ProcFunc's APIs can be composed to create entire scenes using similar tools as would be used by an artist. Other works propose languages (Ganeshan et al., 2024; Jones et al., 2026) or generative grammars (Müller et al., 2006) for shapes at the object level, especially for object editing or abstract shape generation. ProcFunc has a broader scope of primitives, by including every procedural Blender

operation (materials, extrusion, vertex positioning, bezier surfaces, etc.), and can tackle from-scratch procedural generation. These graphics operations can implement scenes more compactly than large numbers of shape primitives.

Most similar to our tools are the operator and geometry node tools introduced in Infinigen (Raistrick et al., 2023), which are also used for material and geometry editing tasks in BlenderGym (Gu et al., 2025). Compared to these tools, ours requires less knowledge of the Blender UI (Fig. 1), as it is fully specified as static Python functions, and is more concise and less error-prone (Tab. 2).

Automated Graphics Editing Many works combine LLMs and VLMs with Blender or other graphics systems. This is often for scene arrangement of non-procedural objects and materials (Huang et al., 2025; Yang et al., 2024; Sun et al., 2025b; Yang et al., 2025; Feng et al., 2023), but has also been applied for high-level configuration of existing procedural generators (Sun et al., 2025a; Liu et al., 2025; Zhang et al., 2026; Zhou et al., 2025). Most directly related to our work is automated graphics editing at the object and material level (Huang et al., 2024; Lu et al., 2025; Li et al., 2025). However, ProcFunc's primitives are a complementary contribution to all LLM/VLM systems: we provide a new graphics API and high quality procedural generator examples, which can improve any LLM/VLM system, rather than advancing agent or LLM design.

3 PROCFUNC

ProcFunc's overall goal is to make procedural generators easy to create and use in Python, especially for content creation and synthetic dataset generation. This includes creating procedural generators from scratch, quickly combining existing generators, customizing dataset distributions (Wen et al., 2025b) and meeting task requirements such as paired synthetic data (Nugent et al., 2025) or providing full parameter access for inverse graphics (Dax et al., 2025).

3.1 PRIMITIVES API

We maximize the simplicity of combining primitive graphics operations while ensuring they are capable of generating any procedural scene. This results in a carefully designed set of 497 Python functions which express the same functionality as commonly used procedural tools in Blender, such as material and mesh editing operations.

We design each primitive to expose a single graphics algorithm as an atomic function, even if the corresponding Blender construct is multiple functions, or is many algorithms combined into a single function. Instead of a single "Math" shader function we provide many arithmetic functions, or for a single "Noise" function we provide a separate function with different arguments for each type of noise. We also combine functions, so that users do not need different functions for different data types or for materials vs geometry. We ensure each function exposes all dependencies as explicit function arguments, rather than implicit background state, and returns all its effects as return values, including any active faces as NumPy arrays. ProcFunc's function and argument names are similar to Blender's operation names, but are changed for clarity and disambiguation (e.g. duplicate "Vector" arguments in Fig. 1).

Each primitive function is executed by configuring the correct context and executing the operation through the Blender Python API. We optimize this invocation where possible, e.g. by using a vectorized operation to select items, or configuring an isolated Blender context for the operation to run in, even if this would not normally be done by a Blender API user. The exact implementation of each primitive can also evolve in future versions of ProcFunc without affecting its interface. At execution-time we also perform runtime type inference of "data_type" arguments (see Fig. 1), and resolve operators "$+ - / * / /$" to shader or other arithmetic with that datatype.

Finally, we provide a *transpiler*, which converts procedural generators from Blender Node graphs to Python code to be used with LLMs or other code. Our transpiler is similar to Infinigen's, but adapted to ProcFunc's interface. It also inlines short expressions, uses operator overloads, and can post-process graphs to use HSV colors and interpolate example parameters. This allows Python code to assist in GUI-based editing, and is essential for LLMs to edit pre-existing Blender node graphs in ProcFunc. See Appendix B.4 for example ProcFunc material programs.

3.2 Module Composition Framework

All asset generators in ProcFunc are simple functions which are easily controllable by setting their input arguments. These functions generate and return a useful asset with no other steps or configuration. We define standardized function interfaces for categories of generators (*Masks, Materials, Objects, Scenes*), which guarantee certain input / output arguments will be present. For example, Materials must have a "vector" coordinate system argument, and return a "surface", "displacement" and "volume" to define shading and texture. This enables widespread compositionality, as generators can be substituted while providing the same interface. Treating outputs as return values is different to the Blender API, which writes them into a hidden scene state which must be retrieved with a string name. Explicit outputs are useful for composition, e.g. allowing downstream effects like scratches to access the "surface" then emit their own modified scratched surface. Our primitives API is also essential to this, as otherwise there are implicit inputs and outputs which are challenging to capture in an interface.

Random Generation We also distinguish between random generators and deterministic generators. Each class of generator (e.g. Wood) has one or more *random sampler functions*, which sample parameters then create and return the asset. Random samplers are useful for dataset generation or any case in which the user doesn't wish to specify exact parameters. There can be multiple samplers for different distributions of a single deterministic asset. All random functions require a "np.random.Generator" (RNG) input, so randomness is explicitly declared and reproducible from a random seed.

Static Analysis with Program Tracing Our design, so far, is simple but not accessible for inspection or optimization, as random parameters are local to the functions that use them. We address this with a tracer, similar to JAX and especially PyTorch's `fx.symbolic_trace`, which executes a program in a special context to capture its compute graph without executing any expensive operations. Our tracer can produce both *Instance-level Graphs* and *Distribution-level Graphs*. Instance-level graphs capture only the control-flow path and numeric parameters that affected a single output. *Distribution-level* graphs capture distributions of parameters (uniform, normal, etc.) and capture all possible discrete combinations in the graph, to allow the weights and parameters of random choices to be inspected and changed, e.g. changing the chance of a material or range of a color hue.

Both types of graphs can also be converted into Python code corresponding to a particular asset, with all parameters and public function calls specified - see Appendix C.1 for examples. Overall, each random procedural sampler can produce unlimited specific instance-level graphs with concrete parameters corresponding to an image. We anticipate these will be useful for precise per-asset control or as training supervision for image-to-program / inverse procedural generation tasks.

3.3 Pre-implemented Procedural Generators

We provide a high-quality library of pre-made procedural generators implemented using ProcFunc, as well as tools to easily customize and make datasets using these generators. We use these to create an off-the-shelf indoor room generator, shown in Fig. 3.

We provide 35 newly-created high-quality procedural material components:

- *Material Base* - Concrete, Ceramic, Metal, Fabric, Glass, Granite, Gravel, Marble, Paint, Plastic, Stone, Terrazzo and Wood.
- *Material Shapes* - Tiles (10 variants), Bricks (4 variants), Masonry, Planks.
- *Material Masks* - Cracks, Flakes, Smudges, Dirt, Scratches, Edge Wear

Random samples and compositions are shown in Fig. 2 and Appendix B.3. Each category has significant internal diversity. ProcFunc's interfaces enable arbitrary compositions of these modules, with thousands or tens of thousands of useful semantic combinations depending on the desired granularity of categories. We also provide altered versions of 14 object-level procedural generators adapted from Infinigen Indoors, which are refactored to meet our interface by accepting procedural parameters and externally provided materials as function arguments.



Figure 3: **Example Room Generator** - We use ProcFunc materials and scene generation tools to create procedural indoor rooms.

Scene and Dataset-Generation Tools We provide several easy-to-use tools which enable simple Python scripts to create new scene arrangements and dataset generation pipelines:

- *Collision Checking* - We use Python-FCL (Pan et al., 2012) but efficiently re-use bounding-volume-hierarchies for any duplicate objects or rigid motion between queries.
- *Iterative Snapping* - we align objects front-to-back or side-to-side with simple function calls, and provide a framework for concise rejection sampling. This enables easy creation of new scene types.
- *UV Surface Cutouts* - we create walls by aligning UV-surface handles to windows/door edges instead of via mesh boolean. This preserves quadrilateral mesh topology, which then allows efficient subdivision to create dense meshes, rather than requiring volumetric marching cubes.
- *Camera Trajectories* - we implement efficient *Rapidly-exploring Random Tree* (RRT*) (Karaman & Frazzoli, 2011; Cheng & LaValle, 2002) trajectories, same as in TartanAir (Wang et al., 2020).
- *Rendering and Ground Truth* - we provide simple functions for Blender Cycles & EEVEE rendering and ground truth. We support any annotation type and NumPy output format. We also optionally compress ground truth storage cost by 75% via bit-casting and video compression (FFmpeg Developers, 2026).

We use these tools to create a procedural indoor room generator (Fig. 3) and augment it with floating objects and lights for stereo datasets (Sec. 4.5). See Appendix C.2 for discussion of each tool.

4 Experiments

We evaluate ProcFunc's ease of use for creating and editing procedural generators. We use VLM editing due to the availability of standardized quantitative benchmarks. We expect the results to indicate usefulness of content creation or LLM-assisted coding, and serve as a proxy for human-understandability and similarity to the general distribution of Python code. We also demonstrate ProcFunc's usefulness for creating new procedural dataset generators. Particularly, we investigate their semantic coverage, analytical diversity, runtime efficiency, level of ground truth detail, and use for model training.

4.1 Procedural Parameter Editing

We compare the relative success of VLMs when using ProcFunc to solve BlenderGym's material and geometry tasks, as opposed to the original benchmark files which use Infinigen. We use a simplified editing process, which accepts starter code and a reference image, and performs 4 rounds of sequential editing and rendering to produce a final set of renders, different to the tree of editing and verification in BlenderAlchemy (Huang et al., 2024). We intend only to demonstrate the relative usefulness of APIs, not the absolute performance of any LLM system — state-of-the-art harnesses would likely be superior. See Appendix D.1 for details and prompts used.

Table 1: **Procedural Parameter Editing** - We use both Infinigen and ProcFunc's APIs as representations for VLMs to edit procedural generators. ProcFunc slightly improves visual metrics, despite being used zero-shot, as opposed to Infinigen which is included in model training data.

		Materials				Geometry				
model	api	PL ↓	NCLIP ↓	Error % ↓	Cost ↓	PL ↓	NCLIP ↓	CD ↓	Error % ↓	Cost ↓
gemini-2.5-flash	infinigen	4.64	10.69	4.1%	$5.30	6.53	11.77	2.98	12.80%	$0.98
gemini-2.5-flash	procfunc	4.14	10.18	3.5%	$6.48	6.87	12.13	1.97	9.40%	$1.03
gemini-2.5-pro	infinigen	8.22	12.43	0.9%	$7.63	5.98	8.57	1.65	6.10%	$3.52
gemini-2.5-pro	procfunc	3.14	7.52	2.0%	$5.72	4.63	7.07	1.43	6.10%	$4.09
gpt5.2-mini	infinigen	5.18	12.61	0.0%	$1.56	6.99	12.19	2.81	2.20%	$1.21
gpt5.2-mini	procfunc	3.95	10.18	2.8%	$1.31	7.54	10.54	2.03	4.40%	$1.19
gpt5.2	infinigen	2.93	9.38	0.0%	$6.59	5.61	8.44	1.55	0.60%	$5.01
gpt5.2	procfunc	3.93	9.82	0.0%	$4.97	5.39	8.29	1.02	1.70%	$5.01
Averaged	infinigen	5.24	11.28	**1.3%**	$5.27	6.28	10.24	2.25	5.43%	**$2.68**
Averaged	procfunc	**3.79**	**9.43**	2.1%	**$4.62**	**6.11**	**9.51**	**1.61**	**5.40%**	$2.83

We compute metrics as in BlenderGym, as well as error rate and cost. *Photometric Loss* (PL) is the mean squared error between final renders and goal images. *Negative CLIP Distance* (N-CLIP) is 1 minus the cosine-similarity of feature vectors extracted by applying the `openai/clip-vit-base-patch32` pretrained ViT encoder. *Chamfer Distance* is computed using 10k-point pointclouds extracted from the generated and goal geometries. *PL, NCLIP and CD* are scaled by 10^2 as in BlenderGym. We also add *Error Rate*, the percentage of code editing steps which crash, and *Cost*, the estimated API dollar cost of each method.

ProcFunc slightly improves visual metrics (Tab. 1) despite significant disadvantages in the standard BlenderGym setting. BlenderGym primarily focuses on editing existing numerical parameters of procedural generators, which makes adding new code and providing clear input arguments less necessary. ProcFunc is also used in zero-shot, as opposed to Infinigen's API which we believe was included in commercial model pretraining. gpt-5.2-mini often completely re-writes the tasks into Infinigen's format, despite having been instructed to make minor edits to ProcFunc code, and despite Infinigen's API not actually being available. Despite these factors, gains from ProcFunc may stem from more concise code, or from better alignment with typical Python code in pretraining. ProcFunc represents the tasks with on average 30% reduced code characters, but this does not yet translate to cost savings due to increased model reasoning tokens or commenting when out of distribution.

We also create a challenging new setting of creating materials from scratch by removing the starter code from BlenderGym, presented in Tab. 2. This experiment could equally be conducted with real images, but we use BlenderGym goal images as they are known to be exactly recreatable with procedural nodes. We perform 8 rounds to compensate for increased difficulty, and additionally provide in-context documentation and 5 completed in-context examples to compensate for the lack of starter code. VLMs make overwhelming coding errors when using Infinigen (30-44%) but much fewer when using our interface.

Table 2: **Procedural Material Creation** - We tackle BlenderGym without starter code, which requires using the APIs to add nodes and create materials from scratch. ProcFunc significantly reduces editing errors and improves visual quality and API cost.

model	api	PL ↓	NCLIP ↓	Error % ↓	Cost ↓
gemini-2.5-pro	infinigen	4.37	10.56	30.3%	$28.60
gemini-2.5-pro	procfunc	3.95	10.14	4.9%	$15.08
gpt5.2	infinigen	5.09	11.92	44.7%	$38.81
gpt5.2	procfunc	4.49	11.51	21.4%	$34.21

4.2 Procedural Generator Efficiency

We aim to evaluate the resource-efficiency of the example procedural room generator (Sec. 3.3). Efficiency is generally a tradeoff with scene detail and complexity: higher triangle-count or object-count scenes require both more memory and render time on the GPU. Thus, we evaluate both high-detail and low-detail scenes, and report both the efficiency and complexity of each result.

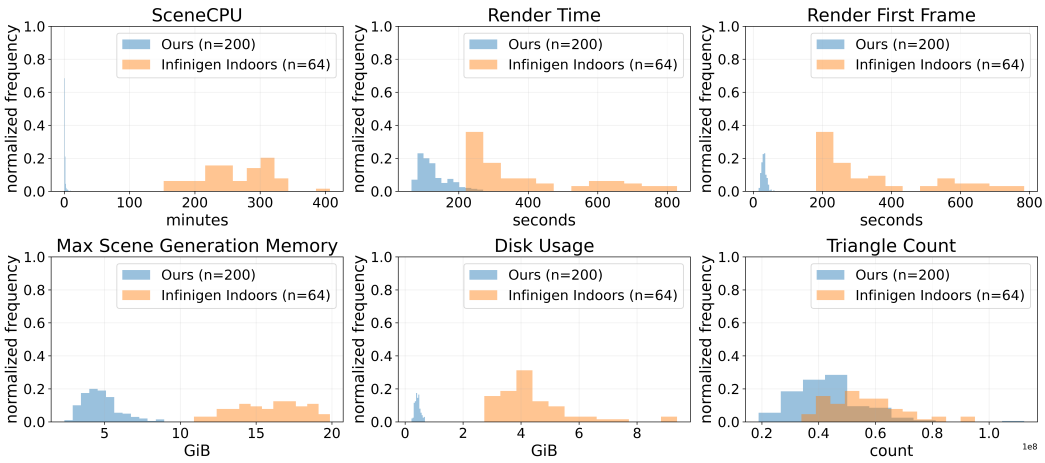


Figure 4: **Resource Usage**: Our system provides a new and useful tradeoff of performance and complexity. It matches the object count and triangle density of Infinigen-Indoors, while having significantly lower mean and variance of resource usage.

Table 3: **Room Generation Efficiency**. ProcFunc's included pre-implemented generators create simple room backgrounds at very low cost. Scene-CPU is the time required to generate a 3D scene prior to rendering or export. Render $t_1 + t_2$ means that the first frame takes t_1 seconds to generate, including setup time, and the remaining frames take t_2 seconds per frame.

Settings			Complexity		Performance			
Method	Obj?	Hipoly?	#Tris	#Objs.	SceneCPU ↓	Mem. ↓	Render ↓	Storage ↓
InfIndoors	×	×	0.37M	22.9	0.46min	1.2G	**7.6**+**2.67s**	0.35G
Ours	×	×	0.02M	18.5	**0.02min**	**0.41G**	10.4+8.01s	0.16G
Ours Eevee	×	×	0.02M	18.5	**0.02min**	**0.41G**	71.9+4.44s	**0.15G**
InfIndoors	×	✓	49.1M	22.7	276min	15.07G	938.1+3.82s	3.78G
Ours	×	✓	35.4M	18.5	**0.02min**	**2.53G**	**25.8**+**8.49s**	**0.16G**
InfIndoors	✓	×	6.92M	31.9	17.91min	**2.67G**	196.1+**3.27s**	0.77G
Ours	✓	×	3.13M	60.1	0.77min	4.11G	**13.1**+8.19s	0.42G
Ours Eevee	✓	×	3.22M	59.3	**0.69min**	4.11G	105.7+5.46s	**0.41G**
InfIndoors	✓	✓	55.7M	30.6	278.98min	15.94G	356.6+**3.66s**	4.39G
Ours	✓	✓	43.4M	60.1	**1.1min**	**4.743G**	**30.7**+8.79s	**0.43G**

We benchmark our system by generating many scenes and rendering 12 monocular camera views in each scene. We report hardware usage (CPU time, RAM, GPU time, Storage) so that individuals can determine how this maps to throughput or monetary cost. SceneCPU refers to the CPU-only duration to create a 3D scene file (which can be used for any purpose), and assumes access to 4 Intel(R) Xeon(R) Gold 5320 CPU cores. GPU time is measured on 1 NVIDIA L40 GPU.

We provide results in Tab. 3 and Fig. 4. Our system has very short CPU runtime (less than 2 minutes, even for detailed scenes). Render time is comparable to or slightly slower than Infinigen Indoors, due to the cost of executing more detailed shader programs. This could be reduced by pre-evaluating shaders to image textures. We also reduce scene and ground truth storage cost by 40% to 90%.

4.3 Procedural Generator Diversity

Composition increases the semantic coverage of any given set of procedural generators. Without composition each attribute is present by itself, whereas with composition, all combinations are present without being implemented one by one. This is best measured within a single system, with the set of attributes held fixed, rather than between systems with different sets or granularity of attributes.

Table 4: **Analytical Diversity** - Composition increases the mean number of discrete choices and continuous parameters per generator, increases the entropy of the distribution (assuming continuous parameters discretized to 3 bits), and increases Cyclomatic / McCabe complexity, which measures the number of linearly independent control flow paths.

Strategy	Continuous Params.	Discrete Params.	Cyclomatic Complexity	Entropy (bits)
Base Materials	25.5	1.80	45	81.3
Multi-level Composition	41.7	3.71	448	134

Table 5: **Stereo Dataset Comparison**: Our system is able to generate highly effective stereo data. Midd (F) and (H) refer to full and half resolution of the Middlebury 2014 validation set, respectively. We evaluate on quarter resolution Booster.

Dataset	Size	Midd (F)		Midd (H)		KITTI-15		ETH3D		Booster	
		EPE	2px	EPE	2px	EPE	3px	EPE	1px	EPE	2px
Sceneflow	40k	4.37	13.72	1.71	9.27	2.59	6.03	0.30	3.83	3.77	15.34
CREStereo	200k	5.28	17.24	2.09	13.81	1.19	6.27	**0.28**	3.85	2.67	14.17
TartanAir	306k	10.7	18.24	2.17	8.92	1.09	5.49	0.35	4.3	3.89	14.01
FSD	1M	**1.67**	**7.79**	**0.83**	**5.93**	**1.03**	5.00	0.29	2.98	2.04	10.04
ProcFunc	50k	2.23	9.05	1.16	6.86	1.06	**4.92**	**0.28**	**2.95**	**1.42**	**9.23**

We use our program tracer to extract compute graphs with and without composition and analytically calculate their diversity (Tab. 4, Fig. 40). Our procedural implementations avoid code which samples random values that are not used, especially by avoiding invoking modules or control flow branches that wont be included in the final asset. This ensures entropy and other statistics are as meaningful as possible, rather than counting randomness that does not affect the asset output.

4.4 Procedural Material Geometry Detail

ProcFunc's provided material generators are designed to create detailed meshes by compositionally combining per-vertex displacement shaders. Besides providing excellent visual realism (Fig. 3), this provides very detailed ground truth annotations for tasks such as depth estimation or surface normal estimation. In Fig. 5 we compute the detailedness of these annotations via average normal variation $V_i = \sum_{j \in N(i)} \angle(n_i, n_j)$ for square neighborhoods of size 15×15 centered at pixel i. We aggregate valid pixels from 500 images in each dataset.

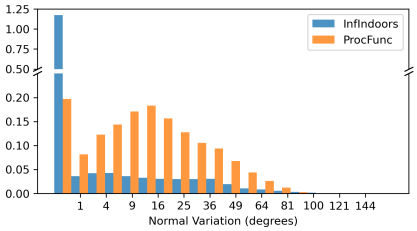


Figure 5: **Surface Detail** - ProcFunc materials have significantly more varied surface normals than in Infinigen Indoors, which are primarily flat with no geometry displacement.

4.5 Stereo Dataset Generation

We provide a baseline result for the effectiveness of stereo datasets generated using our minimal base room generator. We generate a dataset of 50k stereo pairs using our room generator combined with random floating objects and no reflective materials, following WMGStereo (Yan et al., 2026). Compared to the WMGStereo scene generator, which builds on an optimized version of Infinigen Indoors, our stereo scenes were 31x faster to generate (0.41 min vs. 13 min).

We train RAFT-Stereo (Lipson et al., 2021) on a short training schedule of 75k steps using standard hyper-parameters and data augmentation using 2 NVIDIA A6000 GPUs. We provide results on standard benchmarks in Tab. 5. Training on ProcFunc data yielded better performance than several widely used stereo datasets, and is competitive with FSD's dataset (Wen et al., 2025a), despite being drastically smaller. Notably, we do not claim an advance in state-of-the-art stereo dataset generation. Rather, we show that a simple ProcFunc program serves as a strong baseline for stereo data.

AI USE STATEMENT

In this work, we used generative AI tools to create and edit software code and implement methods for some parts of our system. This consisted of interactive editing using Claude Code and OpenAI Codex to achieve short human-directed code editing tasks.

We did not use generative AI to propose or refine hypotheses, design or provide feedback on research methodology or experiments, or interpret results. We also did not use it to develop conceptual frameworks; the design of our procedural generation interfaces and primitives is created entirely by us.

The other required disclosure tasks are not applicable to this work: formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, assist with translation, support qualitative and thematic data analysis. Although our work creates synthetic data using computer graphics, this synthetic data comes from computer graphics rendering, not direct generation by an AI, and the development of this graphics code is discussed above.

We have reviewed all AI-assisted work, namely by interactively approving and later re-reviewing all code edits. We take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We include a .zip file of our source code in our supplementary material. This includes code for all procedural generators, which can produce assets and images from the same distribution as our procedural generator qualitative figures. We also include experiment code for LLM API evaluation on BlenderGym, which can evaluate the same models on the same tasks as reported in our experiments, and thereby reproduce them to the greatest extent possible. However, these experiments rely on propriety LLM APIs, and like all such experiments they are not ultimately exactly reproducible from the source code alone.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695, 2016. URL http://arxiv.org/abs/1605.08695.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Peng Cheng and Steven M. LaValle. Resolution complete rapidly-exploring random trees. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation, ICRA 2002, May 11-15, 2002, Washington, DC, USA*, pp. 267–272. IEEE, 2002. doi: 10.1109/ROBOT.2002.1013372. URL https://doi.org/10.1109/ROBOT.2002.1013372.

Maximilian Dax, Jordi Berbel, Jan Stria, Leonidas Guibas, and Urs Bergmann. Synthesizing 3d abstractions by inverting procedural buildings with transformers, 2025. URL https://arxiv.org/abs/2501.17044.

Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Kiana Ehsani, Jordi Salvador, Winson Han, Eric Kolve, Aniruddha Kembhavi, and Roozbeh Mottaghi. Procthor: Large-scale embodied AI using procedural generation. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/27c546ab1e4f1d7d638e6a8dfbad9a07-Abstract-Conference.html.

Maximilian Denninger, Martin Sundermeyer, Dominik Winkelbauer, Youssef Zidan, Dmitry Olefir, Mohamad Elbadrawy, Ahsan Lodhi, and Harinandan Katam. Blenderproc. *CoRR*, abs/1911.01911, 2019. URL http://arxiv.org/abs/1911.01911.

Clemens Eppner, Adithyavairavan Murali, Caelan Reed Garrett, Rowland O'Flaherty, Tucker Hermans, Wei Yang, and Dieter Fox. scene_synthesizer: A python library for procedural scene generation in robot manipulation. *J. Open Source Softw.*, 10(105):7561, 2025. doi: 10.21105/JOSS.07561. URL https://doi.org/10.21105/joss.07561.

Weixi Feng, Wanrong Zhu, Tsu-Jui Fu, Varun Jampani, Arjun R. Akula, Xuehai He, Sugato Basu, Xin Eric Wang, and William Yang Wang. Layoutgpt: Compositional visual planning and generation with large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/3a7f9e485845dac27423375c934cb4db-Abstract-Conference.html.

FFmpeg Developers. FFmpeg, 2026. Software, available from https://ffmpeg.org.

Aditya Ganeshan, Ryan Y. Huang, Xianghao Xu, R. Kenny Jones, and Daniel Ritchie. Parsel: Parameterized shape editing with language. *ACM Trans. Graph.*, 43(6):197:1–197:14, 2024. doi: 10.1145/3687922. URL https://doi.org/10.1145/3687922.

Klaus Greff, Francois Belletti, Lucas Beyer, Carl Doersch, Yilun Du, Daniel Duckworth, David J. Fleet, Dan Gnanapragasam, Florian Golemo, Charles Herrmann, Thomas Kipf, Abhijit Kundu, Dmitry Lagun, Issam H. Laradji, Hsueh-Ti Derek Liu, Henning Meyer, Yishu Miao, Derek Nowrouzezahrai, A. Cengiz Öztireli, Etienne Pot, Noha Radwan, Daniel Rebain, Sara Sabour, Mehdi S. M. Sajjadi, Matan Sela, Vincent Sitzmann, Austin Stone, Deqing Sun, Suhani Vora, Ziyu Wang, Tianhao Wu, Kwang Moo Yi, Fangcheng Zhong, and Andrea Tagliasacchi. Kubric: A scalable dataset generator. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pp. 3739–3751. IEEE, 2022. doi: 10.1109/CVPR52688.2022.00373. URL https://doi.org/10.1109/CVPR52688.2022.00373.

Yunqi Gu, Ian Huang, Jihyeon Je, Guandao Yang, and Leonidas J. Guibas. Blendergym: Benchmarking foundational model systems for graphics editing. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 18574–18583. Computer Vision Foundation / IEEE, 2025. doi: 10.1109/CVPR52734.2025.01731. URL https://openaccess.thecvf.com/content/CVPR2025/html/Gu_BlenderGym_Benchmarking_Foundational_Model_Systems_for_Graphics_Editing_CVPR_2025_paper.html.

Maxim Gumin, Do Heon Han, Seung Jean Yoo, Aditya Ganeshan, R. Kenny Jones, Kailiang Fu, Rio Aguina-Kang, Stewart Morris, and Daniel Ritchie. Procedural scene programs for open-universe scene generation: Llm-free error correction via program search. In Taku Komura, Michael Wimmer, and Hongbo Fu (eds.), *Proceedings of the SIGGRAPH Asia 2025 Conference Papers, SA Conference Papers 2025, Hong Kong, December 15-18, 2025*, pp. 141:1–141:11. ACM, 2025. doi: 10.1145/3757377.3763930. URL https://doi.org/10.1145/3757377.3763930.

Ian Huang, Guandao Yang, and Leonidas J. Guibas. Blenderalchemy: Editing 3d graphics with vision-language models. In Ales Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol (eds.), *Computer Vision - ECCV 2024 - 18th European Conference, Milan, Italy, September 29-October 4, 2024, Proceedings, Part LXXXIX*, volume 15147 of *Lecture Notes in Computer Science*, pp. 297–314. Springer, 2024. doi: 10.1007/978-3-031-73024-5_18. URL https://doi.org/10.1007/978-3-031-73024-5_18.

Ian Huang, Yanan Bao, Karen Truong, Howard Zhou, Cordelia Schmid, Leonidas J. Guibas, and Alireza Fathi. Fireplace: Geometric refinements of LLM common sense reasoning for 3d object placement. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 13466–13476. Computer Vision Foundation / IEEE, 2025. doi: 10.1109/CVPR52734.2025.01257. URL https://openaccess.thecvf.com/

content/CVPR2025/html/Huang_FirePlace_Geometric_Refinements_of_LLM_Common_Sense_Reasoning_for_3D_CVPR_2025_paper.html.

R. Kenny Jones, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapelib: Designing a library of programmatic 3d shape abstractions with large language models, 2026. URL https://arxiv.org/abs/2502.08884.

Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *Int. J. Robotics Res.*, 30(7):846–894, 2011. doi: 10.1177/0278364911406761. URL https://doi.org/10.1177/0278364911406761.

Beichen Li, Rundi Wu, Armando Solar-Lezama, Changxi Zheng, Liang Shi, Bernd Bickel, and Wojciech Matusik. Vlmaterial: Procedural material generation with large vision-language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=wHebuIb6IH.

Lahav Lipson, Zachary Teed, and Jia Deng. Raft-stereo: Multilevel recurrent field transforms for stereo matching. In *International Conference on 3D Vision, 3DV 2021, London, United Kingdom, December 1-3, 2021*, pp. 218–227. IEEE, 2021. doi: 10.1109/3DV53792.2021.00032. URL https://doi.org/10.1109/3DV53792.2021.00032.

Xinhang Liu, Chi-Keung Tang, and Yu-Wing Tai. Worldcraft: Photo-realistic 3d world creation and customization via LLM agents. *CoRR*, abs/2502.15601, 2025. doi: 10.48550/ARXIV.2502.15601. URL https://doi.org/10.48550/arXiv.2502.15601.

Sining Lu, Guan Chen, Nam Anh Dinh, Itai Lang, Ari Holtzman, and Rana Hanocka. Ll3m: Large language 3d modelers, 2025. URL https://arxiv.org/abs/2508.08228.

Zeyu Ma, Alexander Raistrick, Lahav Lipson, and Jia Deng. Mesh extraction for unbounded scenes using camera-aware octrees. In *International Conference on 3D Vision, 3DV 2025, Singapore, March 25-28, 2025*, pp. 845–854. IEEE, 2025. doi: 10.1109/3DV66043.2025.00083. URL https://doi.org/10.1109/3DV66043.2025.00083.

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006. doi: 10.1145/1141911.1141931. URL https://doi.org/10.1145/1141911.1141931.

Jack Nugent, Siyang Wu, Zeyu Ma, Beining Han, Meenal Parakh, Abhishek Joshi, Lingjie Mei, Alexander Raistrick, Xinyuan Li, and Jia Deng. Evaluating robustness of monocular depth estimation with procedural scene perturbations. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/129033c7c08be683059559e8d6bfd460-Abstract-Conference.html.

Jia Pan, Sachin Chitta, and Dinesh Manocha. FCL: A general purpose library for collision and proximity queries. In *IEEE International Conference on Robotics and Automation, ICRA 2012, 14-18 May, 2012, St. Paul, Minnesota, USA*, pp. 3859–3866. IEEE, 2012. doi: 10.1109/ICRA.2012.6225337. URL https://doi.org/10.1109/ICRA.2012.6225337.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Ken Perlin. An image synthesizer. In Pat Cole, Robert Heilman, and Brian A. Barsky (eds.), *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1985, San Francisco, California, USA, July 22-26, 1985*, pp. 287–296. ACM, 1985. doi: 10.1145/325334.325247. URL https://doi.org/10.1145/325334.325247.

Alexander Raistrick, Lahav Lipson, Zeyu Ma, Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han, Yihan Wang, Alejandro Newell, Hei Law, Ankit Goyal, Kaiyu Yang, and Jia Deng. Infinite photorealistic worlds using procedural generation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pp. 12630–12641. IEEE, 2023. doi: 10.1109/CVPR52729.2023.01215. URL https://doi.org/10.1109/CVPR52729.2023.01215.

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. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*, pp. 21783–21794. IEEE, 2024. doi: 10.1109/CVPR52733.2024.02058. URL https://doi.org/10.1109/CVPR52733.2024.02058.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3d-gpt: Procedural 3d modeling with large language models. In *International Conference on 3D Vision, 3DV 2025, Singapore, March 25-28, 2025*, pp. 1253–1263. IEEE, 2025a. doi: 10.1109/3DV66043.2025.00119. URL https://doi.org/10.1109/3DV66043.2025.00119.

Fan-Yun Sun, Weiyu Liu, Siyi Gu, Dylan Lim, Goutam Bhat, Federico Tombari, Manling Li, Nick Haber, and Jiajun Wu. Layoutvlm: Differentiable optimization of 3d layout via vision-language models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 29469–29478. Computer Vision Foundation / IEEE, 2025b. doi: 10.1109/CVPR52734.2025.02744. URL https://openaccess.thecvf.com/content/CVPR2025/html/Sun_LayoutVLM_Differentiable_Optimization_of_3D_Layout_via_Vision-Language_Models_CVPR_2025_paper.html.

Jianhua Sun, Yuxuan Li, Jiude Wei, Longfei Xu, Nange Wang, Yining Zhang, and Cewu Lu. Artipg: A toolbox for procedurally synthesizing large-scale and diverse articulated objects with rich annotations. In *IEEE/CVF International Conference on Computer Vision, ICCV 2025, Honolulu, HI, USA, October 19-25, 2025*, pp. 6396–6405. IEEE, 2025c. doi: 10.1109/ICCV51701.2025.00603. URL https://doi.org/10.1109/ICCV51701.2025.00603.

Haoming Wang, Qiyao Xue, and Wei Gao. Infinibench: Infinite benchmarking for visual spatial reasoning with customizable scene complexity, 2025. URL https://arxiv.org/abs/2511.18200.

Wenshan Wang, Delong Zhu, Xiangwei Wang, Yaoyu Hu, Yuheng Qiu, Chen Wang, Yafei Hu, Ashish Kapoor, and Sebastian A. Scherer. Tartanair: A dataset to push the limits of visual SLAM. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, pp. 4909–4916. IEEE, 2020. doi: 10.1109/IROS45743.2020.9341801. URL https://doi.org/10.1109/IROS45743.2020.9341801.

Bowen Wen, Matthew Trepte, Joseph Aribido, Jan Kautz, Orazio Gallo, and Stan Birchfield. Foundationstereo: Zero-shot stereo matching. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 5249–5260. Computer Vision Foundation / IEEE, 2025a. doi: 10.1109/CVPR52734.2025.00495. URL https://openaccess.thecvf.com/content/CVPR2025/html/Wen_FoundationStereo_Zero-Shot_Stereo_Matching_CVPR_2025_paper.html.

Hongyu Wen, Yiming Zuo, Venkat Subramanian, Patrick Chen, and Jia Deng. Seeing and seeing through the glass: Real and synthetic data for multi-layer depth estimation. In *IEEE/CVF International Conference on Computer Vision, ICCV 2025, Honolulu, HI, USA, October 19-25, 2025*, pp. 6715–6725. IEEE, 2025b. doi: 10.1109/ICCV51701.2025.00632. URL https://doi.org/10.1109/ICCV51701.2025.00632.

David Yan, Alexander Raistrick, and Jia Deng. What makes good synthetic training data for zero-shot stereo matching?, 2026. URL https://arxiv.org/abs/2504.16930.

Yandan Yang, Baoxiong Jia, Shujie Zhang, and Siyuan Huang. Sceneweaver: All-in-one 3d scene synthesis with an extensible and self-reflective agent. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/cd2b3c429c8a2ca57656970e010b4b60-Abstract-Conference.html.

Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, Chris Callison-Burch, Mark Yatskar, Aniruddha Kembhavi, and Christopher Clark. Holodeck: Language guided generation of 3d embodied AI environments. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*, pp. 16277–16287. IEEE, 2024. doi: 10.1109/CVPR52733.2024.01536. URL https://doi.org/10.1109/CVPR52733.2024.01536.

Yi Zhang, Yunshuang Wang, Zeyu Zhang, and Hao Tang. Code2worlds: Empowering coding llms for 4d world generation. *CoRR*, abs/2602.11757, 2026. doi: 10.48550/ARXIV.2602.11757. URL https://doi.org/10.48550/arXiv.2602.11757.

Yunzhi Zhang, Zizhang Li, Matt Zhou, Shangzhe Wu, and Jiajun Wu. The scene language: Representing scenes with programs, words, and embeddings. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 24625–24634. Computer Vision Foundation / IEEE, 2025. doi: 10.1109/CVPR52734.2025.02293. URL https://openaccess.thecvf.com/content/CVPR2025/html/Zhang_The_Scene_Language_Representing_Scenes_with_Programs_Words_and_Embeddings_CVPR_2025_paper.html.

Mengqi Zhou, Xipeng Wang, Yuxi Wang, and Zhaoxiang Zhang. Roomcraft: Controllable and complete 3d indoor scene generation. *CoRR*, abs/2506.22291, 2025. doi: 10.48550/ARXIV.2506.22291. URL https://doi.org/10.48550/arXiv.2506.22291.

A Appendix

B Qualitative Results

B.1 Room Generation Random Sample

Fig. 6 and Fig. 7 show random livingroom samples from our procedural scene generator paired with surface-normal ground truth.

B.2 Stereo Dataset Random Sample

We provide a random sample of left-camera images from our stereo training dataset in Fig. 8

B.3 High-Realism Materials and Material Composites

We provide 36 random samples for each high-level pre-made material generator in Figures. 9-30.

B.4 Material Implementation Examples

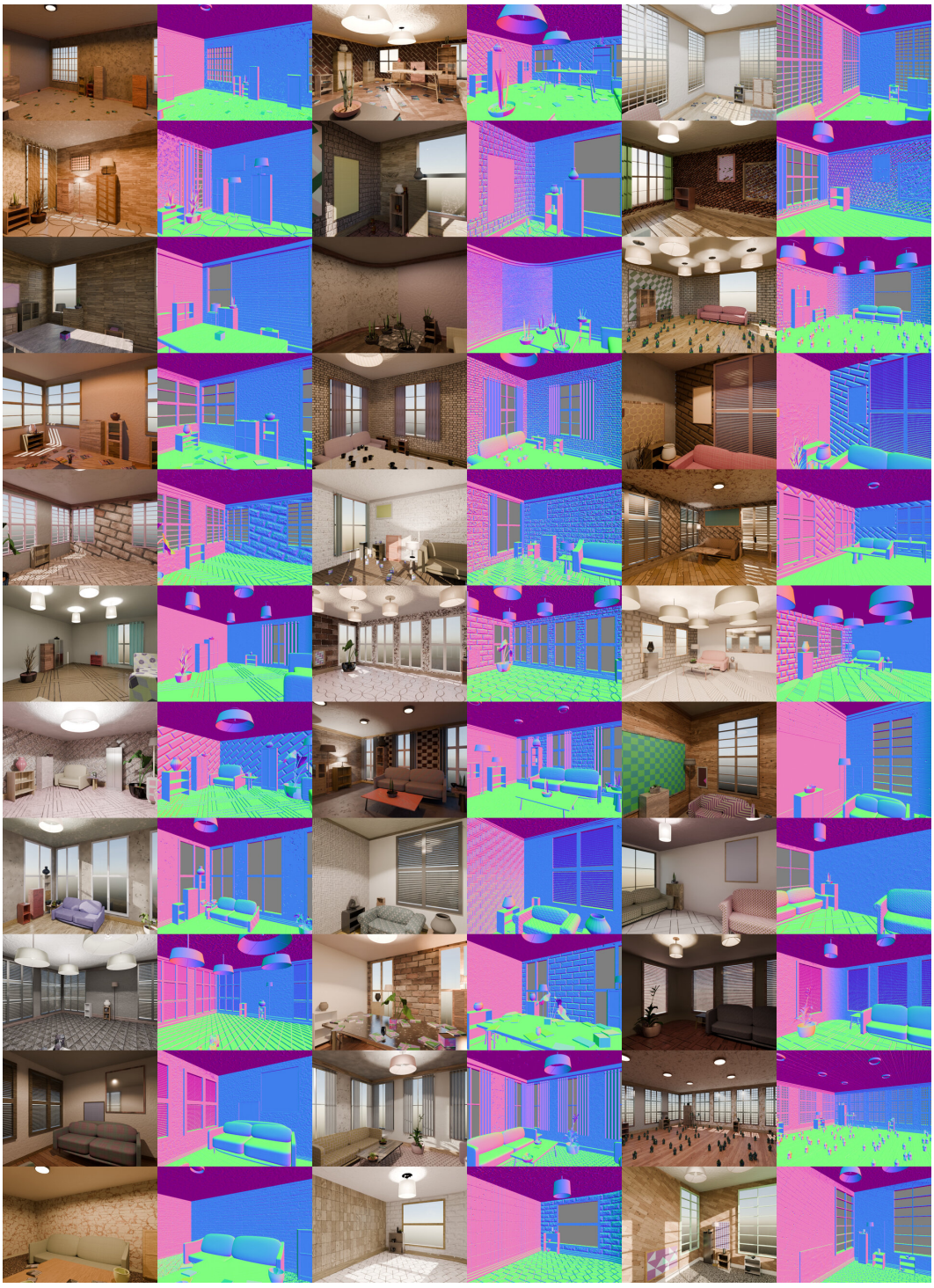


Figure 6: Sample living rooms from our procedural scene generator paired with surface normal ground-truth 1/2.

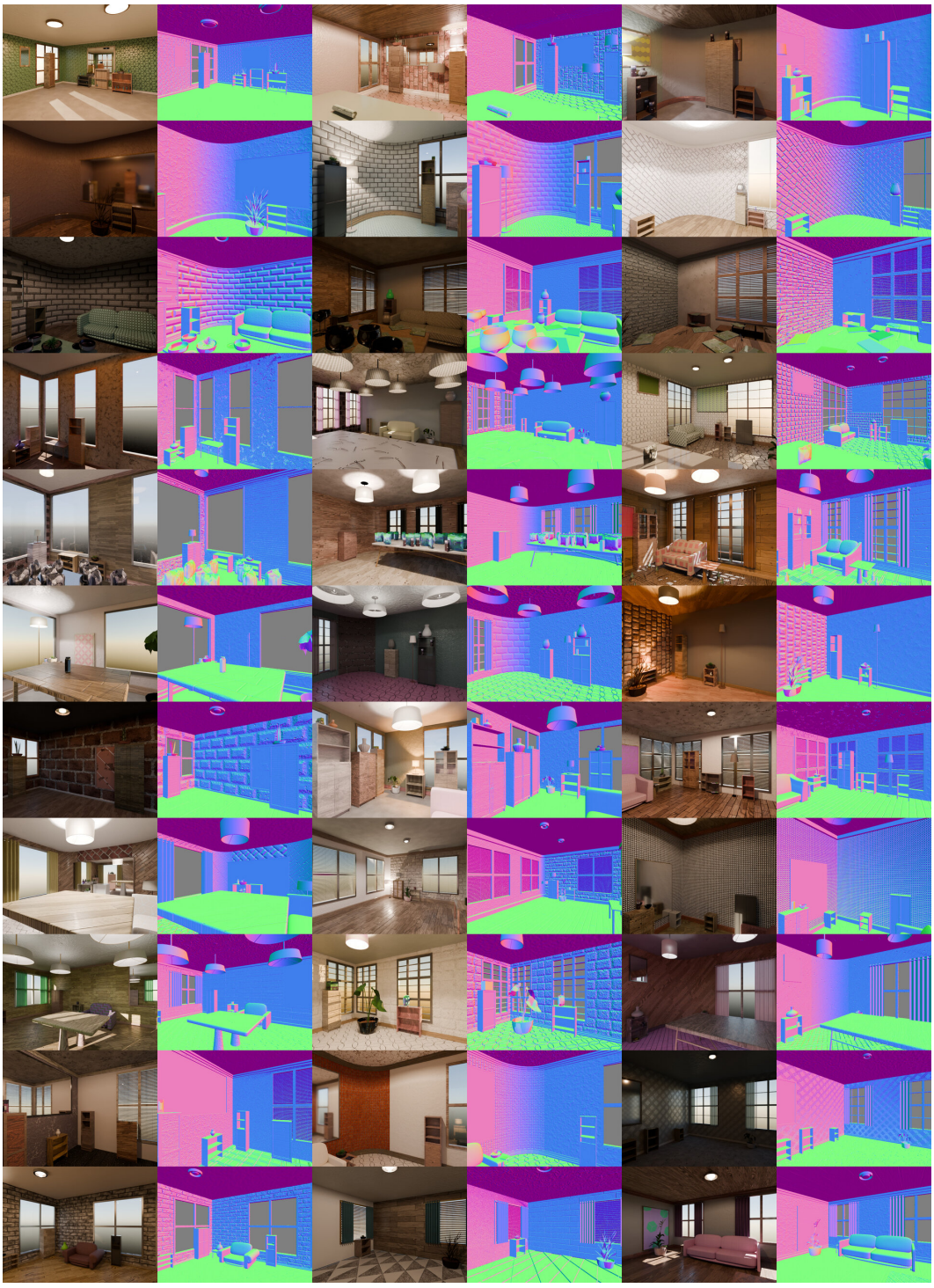


Figure 7: Sample living rooms from our procedural scene generator paired with surface normal ground-truth 2/2.



Figure 8: Raw non-preprocessed stereo dataset samples. We show one random image from each of a random subset of scenes in the dataset. In practice, we render stereo images from 12 unique camera views per scene. We also apply filtering before using them as training data, to remove cases with objects too close to the camera.

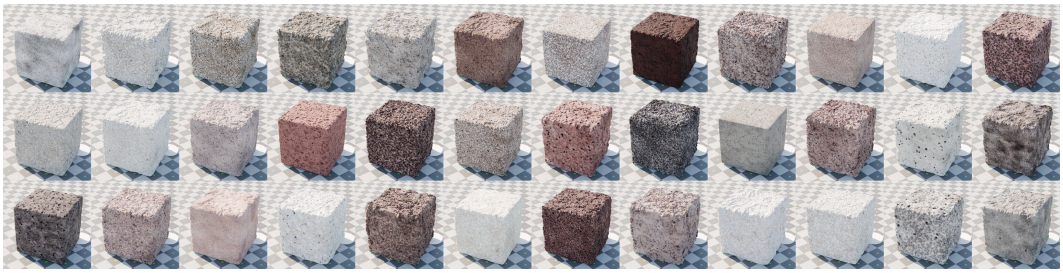
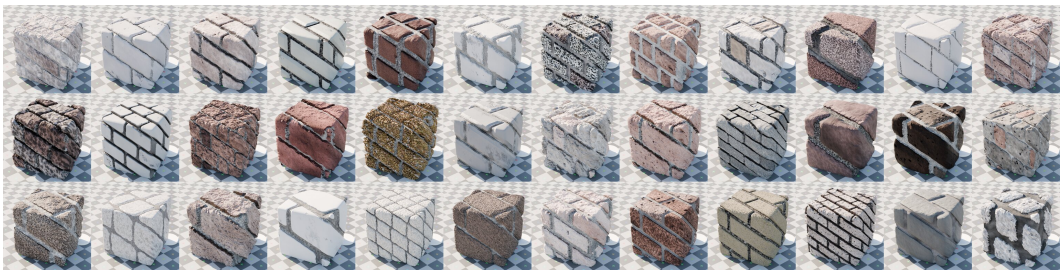
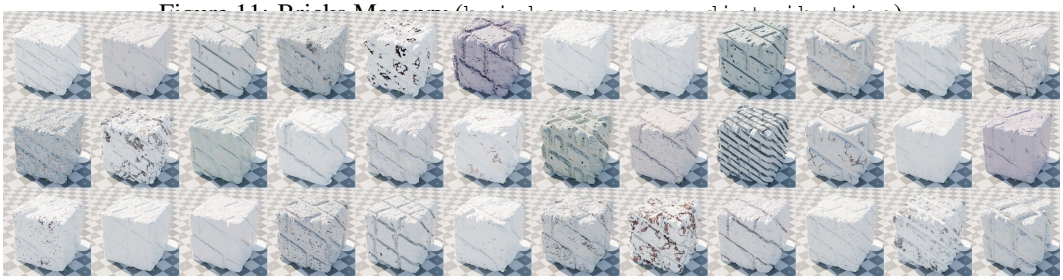
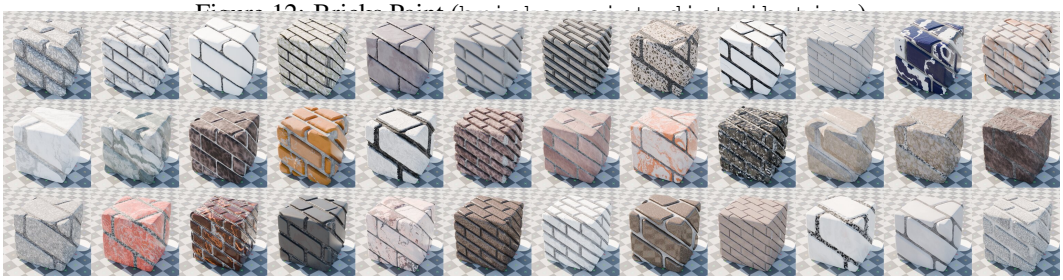


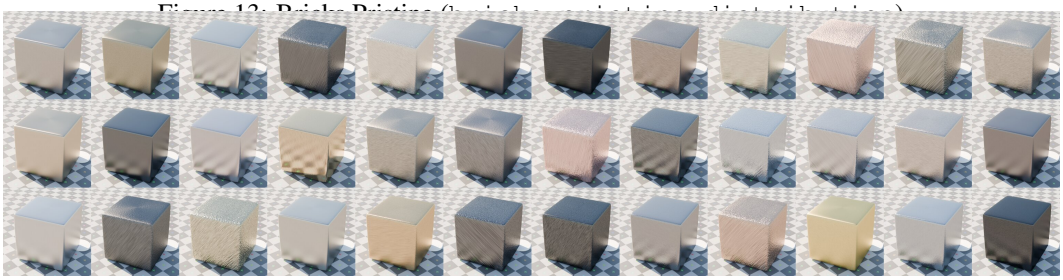
Figure 9: Brick Concrete (`brick_concrete_distribution`)











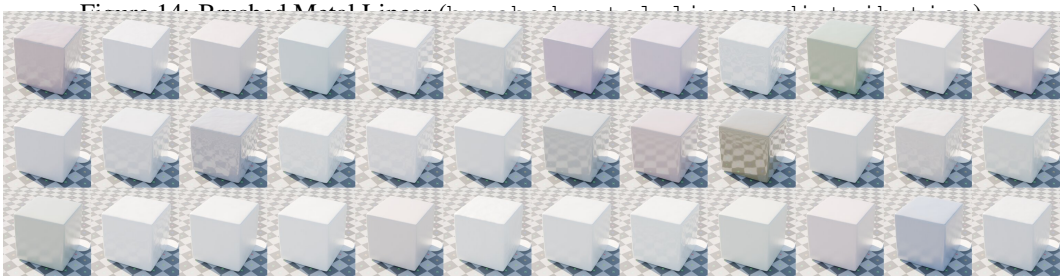
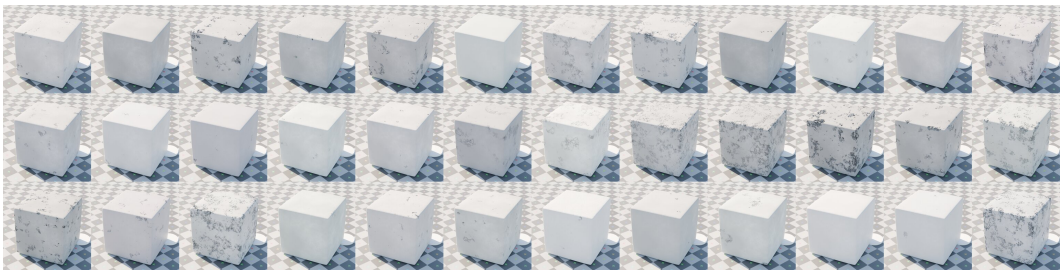
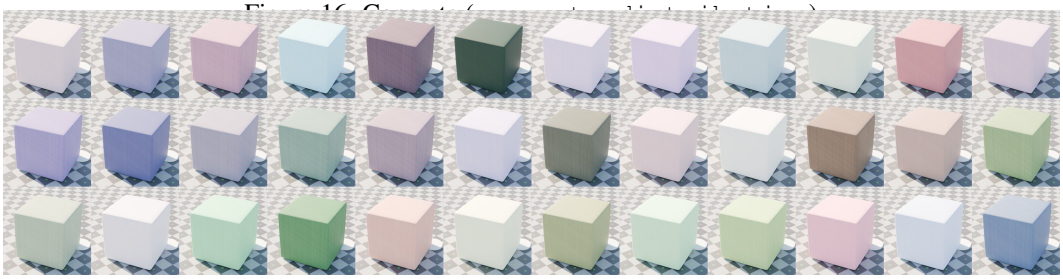
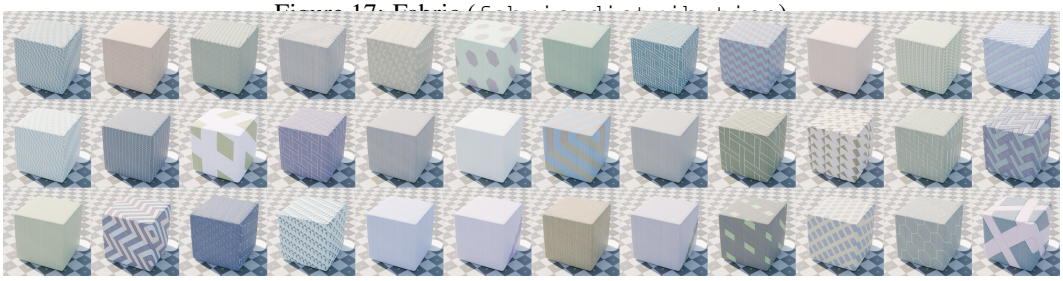
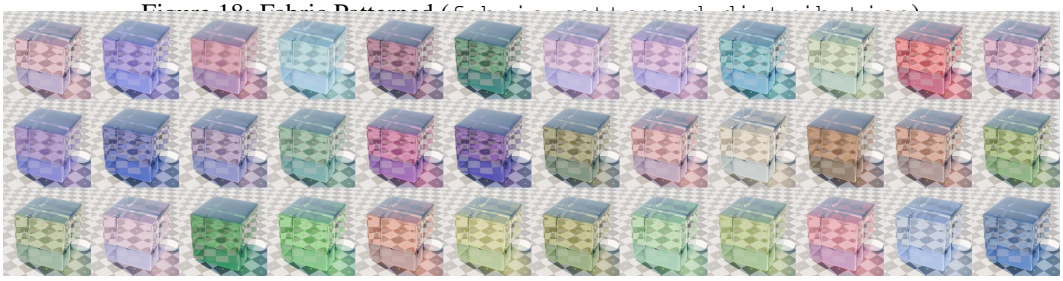


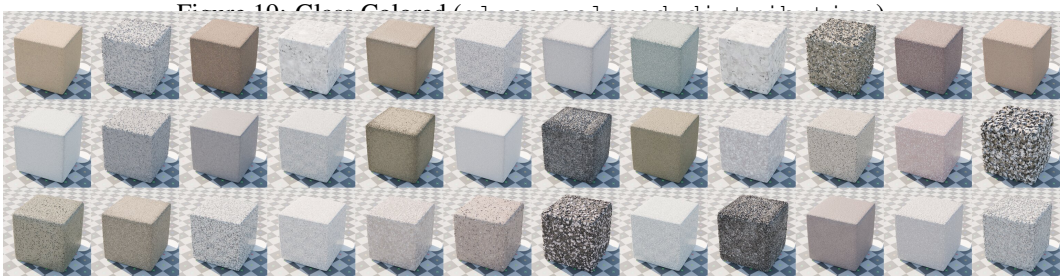
Figure 15: Ceramic (`ceramic_distribution`)











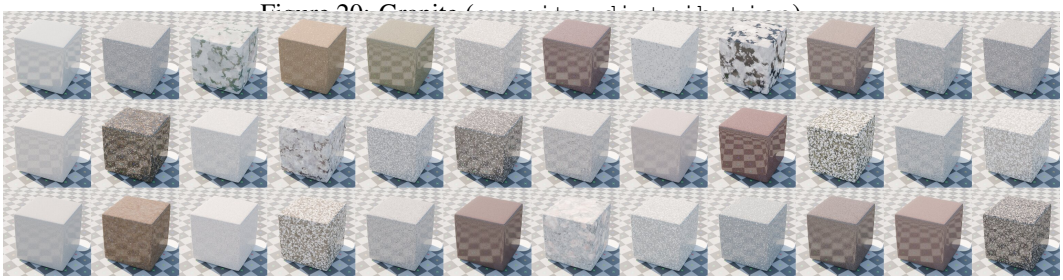
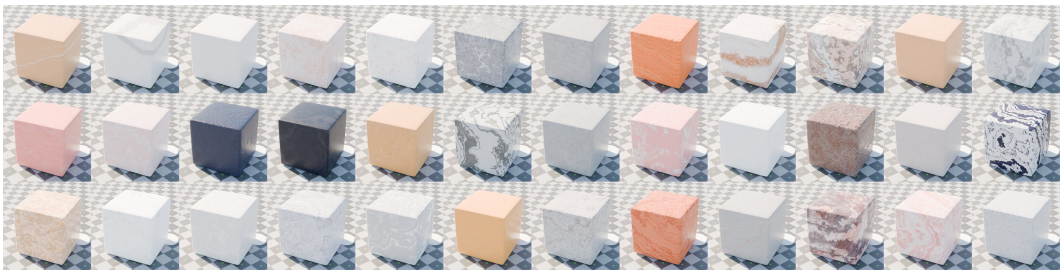
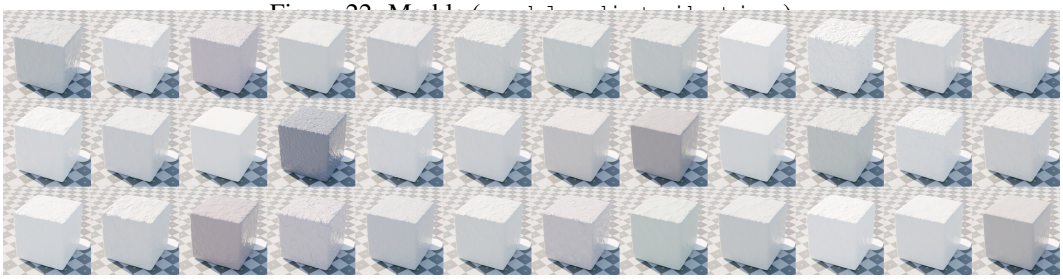
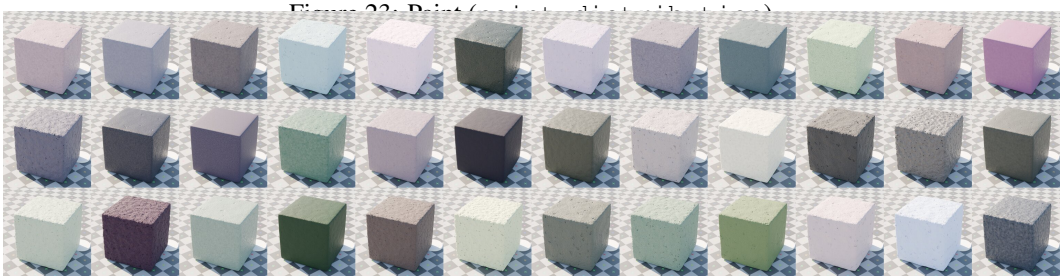
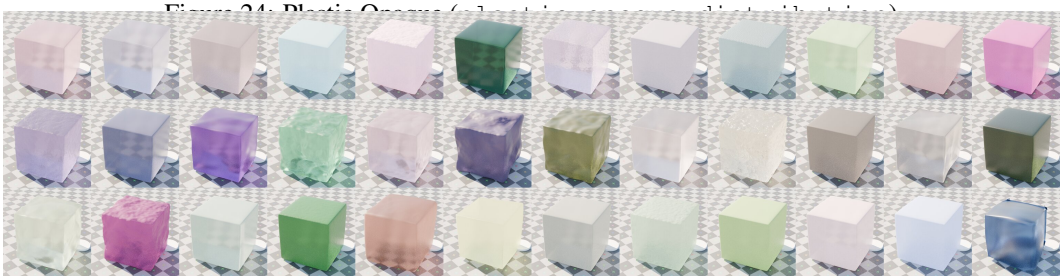


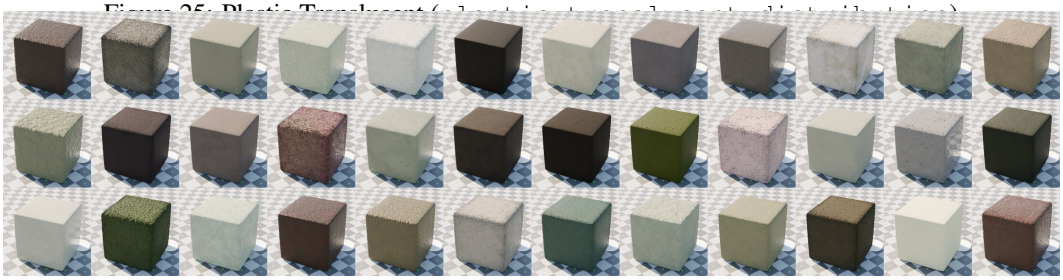
Figure 21: Granite Smooth (`granite_smooth_distribution`)











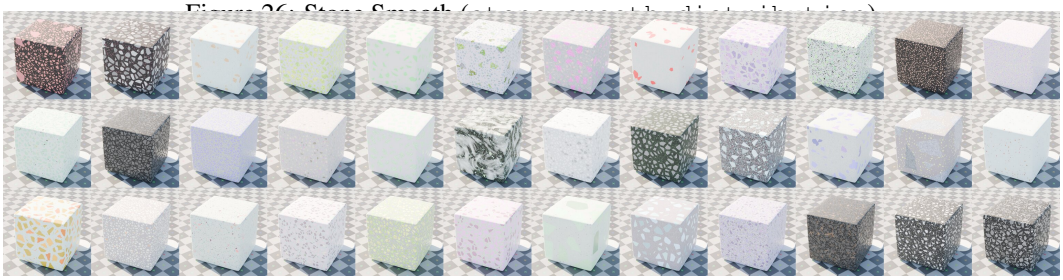


Figure 27: Terrazzo (`terrazzo_distribution`)

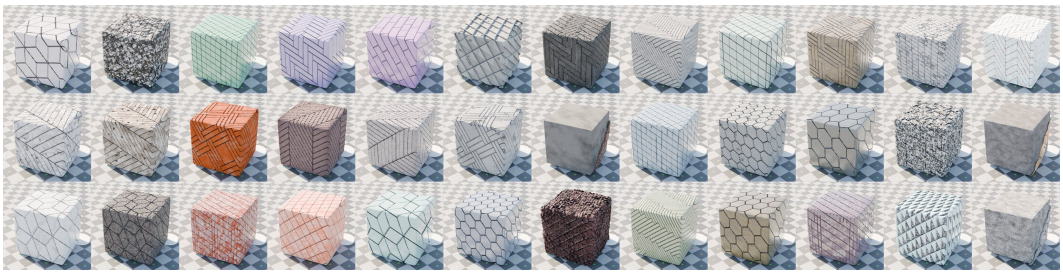


Figure 28: Tile (`tile_distribution`)

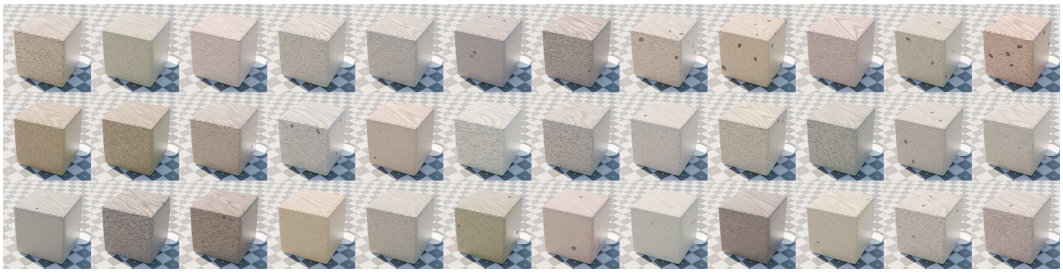


Figure 29: Wood Grain (`wood_grain_distribution`)

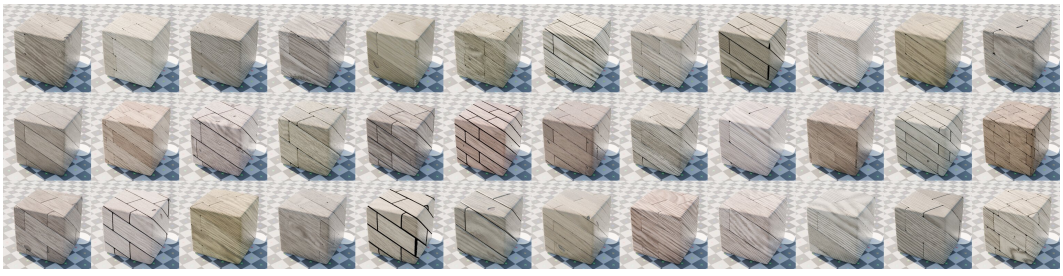


Figure 30: Wood Planks (`wood_planks_distribution`)

Figure 31: Infinigen vs ProcFunc representations of a “Bone" material shader used by BlenderGym (Gu et al., 2025) from Infinigen (Raistrick et al., 2023). ProcFunc’s is significantly shorter (73 vs 127 LOC), and uses function keyword arguments, rather than dictionary keys and Enum entries, to indicate inputs and outputs.

Infinigen

```python
def shader_bone(nw: NodeWrangler):
    # Code generated using version 2.4.3 of the
     node_transpiler

    texture_coordinate =
     nw.new_node(Nodes.TextureCoord)

    mapping = nw.new_node(
        Nodes.Mapping,
        input_kwargs={
            "Vector":
     texture_coordinate.outputs["Object"],
            "Location": (
                1.7 ,
                0.29999999999999999 ,
                0.0,
            ),
        },
    )

    noise_texture_2 = nw.new_node(
        Nodes.NoiseTexture,
        input_kwargs={
            "Vector": mapping,
            "Scale": 10.800000000000001,
            "Detail": 15.0,
            "Roughness": 0.76670000000000005,
        },
    )

    voronoi_texture_1 = nw.new_node(
        Nodes.VoronoiTexture,
        input_kwargs={"Vector":
     noise_texture_2.outputs["Fac"], "Scale":
     10.0},
    )

    colorramp_2 = nw.new_node(
        Nodes.ColorRamp, input_kwargs={"Fac":
     voronoi_texture_1.outputs["Color"]}
    )
    colorramp_2.color_ramp.elements[0].position =
     0.4364
    colorramp_2.color_ramp.elements[0].color =
     (0, 0, 0, 1.0)
    colorramp_2.color_ramp.elements[1].position =
     0.58
    colorramp_2.color_ramp.elements[1].color =
     (1, 1, 1, 1.0)

    mapping_2 = nw.new_node(
        Nodes.Mapping, input_kwargs={"Vector":
     texture_coordinate.outputs["Object"]}
    )

    noise_texture = nw.new_node(
        Nodes.NoiseTexture,
        input_kwargs={
            "Vector": mapping_2,
            "Scale": 98.900000000000006,
            "Detail": 15.0,
            "Roughness": 0.76670000000000005,
        },
    )

    voronoi_texture = nw.new_node(
        Nodes.VoronoiTexture,
        input_kwargs={
            "Vector":
     noise_texture.outputs["Fac"],
            "Scale": 10.0,
        },
    )

    colorramp = nw.new_node(
        Nodes.ColorRamp, input_kwargs={"Fac":
     voronoi_texture.outputs["Color"]}
    )
    colorramp.color_ramp.elements[0].position =
     0.3089
    colorramp.color_ramp.elements[0].color =
     (0.0, 0.0, 0.0, 1.0)
    colorramp.color_ramp.elements[1].position =
     0.673
    colorramp.color_ramp.elements[1].color =
     (1.0, 1.0, 1.0, 1.0)

    multiply = nw.new_node(
        Nodes.VectorMath,
        input_kwargs={0:
     colorramp_2.outputs["Color"], 1:
```

ProcFunc

```python
def shader_bone():
    # Code generated by procfunc

    coord = pf.nodes.shader.coord()

    mapping = pf.nodes.shader.mapping(
        vector=coord.object, location=(1.6554,
     0.3241, -0.007)
    )

    noise = pf.nodes.shader.noise(
        vector=mapping, scale=12.100835,
     detail=15.0, roughness=0.7667
    )

    voronoi = pf.nodes.shader.voronoi(
        vector=noise.fac.astype(dtype=pf.Vector),
     scale=10.0
    )

    color_ramp = pf.nodes.shader.color_ramp(
        fac=voronoi.color.astype(dtype=float),
        points=[(0.4, (0.0, 0.0, 0.0, 1.0)),
     (0.573, (1.0, 1.0, 1.0, 1.0))],
        interpolation="LINEAR",
    )

    mapping_1 =
     pf.nodes.shader.mapping(coord.object)

    noise_1 = pf.nodes.shader.noise(
        vector=mapping_1, scale=102.481506,
     detail=15.0, roughness=0.7667
    )

    voronoi_1 = pf.nodes.shader.voronoi(

     vector=noise_1.fac.astype(dtype=pf.Vector),
     scale=10.044822
    )

    color_ramp_1 = pf.nodes.shader.color_ramp(
        fac=voronoi_1.color.astype(dtype=float),
        points=[(0.334, (0.0, 0.0, 0.0, 1.0)),
     (0.717, (1.0, 1.0, 1.0, 1.0))],
        interpolation="LINEAR",
    )

    mix_rgb_factor = color_ramp.color.astype(
        dtype=pf.Vector
    ) * color_ramp_1.color.astype(dtype=pf.Vector)

    mapping_2 =
     pf.nodes.shader.mapping(vector=coord.uv,
     scale=(1.0, 1.0, 0.0))

    noise_2 =
     pf.nodes.shader.noise(vector=mapping_2,
     scale=7.187254)

    color_ramp_2 = pf.nodes.shader.color_ramp(
        fac=noise_2.fac,
        points=[
            (0.413, (0.381, 0.238, 0.118, 1.0)),
            (0.742, (0.497, 0.503, 0.468, 1.0)),
        ],
        interpolation="LINEAR",
    )

    mix_rgb = pf.nodes.shader.mix_rgb(
        factor=mix_rgb_factor.astype(dtype=float),
        a=pf.Color((0.191, 0.045, 0.01)),
        b=color_ramp_2.color,
    )

    principled = pf.nodes.shader.principled_bsdf(
        base_color=mix_rgb,
        roughness=0.4409,
        ior=1.45,
        subsurface_method="RANDOM_WALK_SKIN",
        subsurface_ior=1.4,
        subsurface_anisotropy=0.0,
        distribution="GGX",
        emission_color=pf.Color((0.0, 0.0, 0.0)),
        emission_strength=1.0,
    )
```

C Extended Method

C.1 Tracer Implementation

A tracer is a system which executes a program, but upon reaching certain functions, will simply record that the function would have been executed, rather than perform the actual computation. The return value of a skipped function is instead replaced with a “Proxy" value, which records its origin point and can be passed to subsequent functions as usual. The Proxy object allows as many operations as possible, such as arithmetic and property access, so that programs which applied these operations to the non-proxy return value will still execute them correctly, albeit simply recording the operations rather than executing them. Once a Proxy has propagated all the way through the program, encountering many skipped functions along the way, it has recorded a compute graph where each yet-to-be-executed function is a graph node, and any proxy inputs and outputs are graph edges.

ProcFunc uses this framework for procedural code by decorating/marking all of its primitives and modules as functions which may need to be captured into the compute graph instead of executed. We also decorate all functions in NumPy while tracing, similarly to JAX (Bradbury et al., 2018). This means that a procedural generator can be analyzed without executing any graphics operations, so tracing can complete usually in less than a second.

ProcFunc decorates all primitives, modules, random values and distribution functions, however it represents these as distinct levels of granularity of computation. Every dataset distribution is composed of random values and modules and primitives, and every module is composed of primitives. Ultimately then, a given dataset `_distribution` function can be represented as either a single entity, or a composition of many primitives, and any one of these granularities may be useful as compute graphs.

ProcFunc’s tracer allows access to any granularity via a `-trace_level` command-line or Python argument. Its effects are demonstrated via code examples. Fig. 32 shows the coarsest level, which can be used to generate useful starter scripts. Fig. 33 shows one step more granular, which has combined the random choices and generator calls of many distributions into a single large branching graph, including random control flow. Fig. 34 shows a more granular graph, which has removed all control flow to show only the path of execution that would actually be taken by a single run of the generator. Finally, Fig. 35 is the most granular, as it executes every random function instead of capturing it in the graph, and therefore has concrete values for every random choice, which provides a concrete and editable representation of a specific run of the procedural generator. These four graphs all represent the same computation, but at different levels of detail, and with different degrees of randomness either executed or left unexecuted.

Figure 32: **Grammar trace.** ProcFunc's Tracer can auto-generate a compute-graph which lists simply the highest level functions invoked in a computation. Here, the `bricks distribution`, `material cube` and `render cycles` functions directly correspond to the user's input command. This is the simplest case of tracing, as it does not inspect the internals of any function, but is still useful to create starting scripts which can be customized to produce randomized datasets.

```python
# uv run procgen bricks_distribution material_cube render_cycles --codegen print
    --trace_level GRAMMAR

def execute_generators():
    # Code generated by procfunc v0.18.0-ifg
    rng = np.random.default_rng()

    rngs = rng.spawn(3)

    vector_vector = pf.nodes.shader.geometry()

    mapping = pf.nodes.shader.mapping(
        vector=vector_vector.position,
        rotation=(-0.7853981633974483, 0, 0),
    )

    bricks_distribution = bricks.bricks_distribution(rng=rngs[0], vector=mapping)

    material_cube = asset_demo.material_cube(rng=rngs[1], material=bricks_distribution)

    render_result = render_cycles.render_cycles(
        objects=material_cube.all_objects,
        camera=material_cube.cameras[0],
        output_folder=Path("outputs/myresult"),
        render_passes=[
            RenderPass(
                type=ExportType.IMAGE,
                path=Path("%c/%f.png"),
                data_type=np.dtype("uint8"),
            )
        ],
        frame_start=0,
        frame_end=0,
        resolution=(512, 512),
        min_samples=8,
        film_exposure=2.0,
    )
    return render_result
```

Figure 33: **Control trace.** ProcFunc's Tracer can auto-generate a compute-graph which contains all control paths, even if they would not have been executed for the current random seed. This example includes parameters and function calls for `granite`, even though a single invocation of "bricks" only uses one. The true path that would be taken is instead recorded via `chosen idx`. This graph is useful for computing metrics without having to execute every possible path, or for transforming the distribution (e.g. replacing every gaussian with uniform, removing certain materials, or other effects). Ellipses (...) are code omitted for brevity.

```
# uv run procgen bricks_distribution material_cube render_cycles --codegen print
    --trace_level RANDOM_CONTROL

def execute_generators():
    # Code generated by procfunc v0.18.0-ifg

    ...

    brick_x = pf.random.clip_gaussian(rng=rngs_2[0], mean=0.3, std=0.1, low=0.1, high=0.5)
    brick_y = pf.random.clip_gaussian(rng=rngs_2[0], mean=0.4, std=0.1, low=0.2, high=1.0)
    brick_cutter_brick_width_vector = pf.nodes.func.combine_xyz(x=brick_x, y=brick_x *
     brick_y)
    brick_cutter_brick_width = brick_cutter_brick_width_vector
    brick_cutter_brick_scaling = pf.random.uniform(rng=rngs_2[2], low=1, high=2)
    brick_cutter_irregularity_choice_options_1 = pf.random.uniform(rng=rngs_2[2], low=0.3,
     high=0.45)
    brick_cutter_grout_bevel = pf.random.uniform(rng=rngs_2[2], low=0.4, high=1.0)
    ...
    brick_cutter_result = brick_cutter.brick_cutter(
        vector=mapping,
        brick_width=brick_cutter_brick_width.x,
        brick_height=brick_cutter_brick_width.y,
        brick_scaling=brick_cutter_brick_scaling,
        brick_bevel_concavity=brick_cutter_brick_bevel_concavity,
        ...
    )

    granite_result = granite.granite(
        ... random params ...
    )
    concrete_result = concrete.concrete(
        ... random params ...
    )

    surface = pf.control.choice(
        choice_options=[
            (brick_concrete_result.surface, 1.0),
            (granite_result.surface, 0.5),
            (concrete_result.surface, 1.0),
        ],
        chosen_idx=2,
    )
    ...
    return render_cycles.render_cycles(...)
```

Figure 34: **Params trace.** ProcFunc's Tracer can auto-generate a compute-graph which contains only the *active* parameters for a particular execution of a randomized generator. Unlike Fig. 33, this code does not contain `granite`, since granite is not used in this particular execution of the bricks. The params trace of a material or object can be used to know the continuous distribution of that object, or to optimize the parameters of the object to meet a target while remaining within the support of the distribution. Ellipses (...) are code omitted for brevity.

```
# uv run procgen bricks_distribution material_cube render_cycles --codegen print
    --trace_level RANDOM_PARAMS

def execute_generators():
    # Code generated by procfunc v0.18.0-ifg

    ...

    brick_x = pf.random.clip_gaussian(rng=rngs_2[0], mean=0.3, std=0.1, low=0.1, high=0.5)
    brick_y = pf.random.clip_gaussian(rng=rngs_2[0], mean=0.4, std=0.1, low=0.2, high=1.0)
    brick_cutter_brick_width_vector = pf.nodes.func.combine_xyz(x=brick_x, y=brick_x *
     brick_y)
    brick_cutter_brick_width = brick_cutter_brick_width_vector
    brick_cutter_brick_scaling = pf.random.uniform(rng=rngs_2[2], low=1, high=2)
    brick_cutter_irregularity_choice_options_1 = pf.random.uniform(rng=rngs_2[2], low=0.3,
     high=0.45)
    brick_cutter_grout_bevel = pf.random.uniform(rng=rngs_2[2], low=0.4, high=1.0)
    ...
    brick_cutter_result = brick_cutter.brick_cutter(
        vector=mapping,
        brick_width=brick_cutter_brick_width.x,
        brick_height=brick_cutter_brick_width.y,
        brick_scaling=brick_cutter_brick_scaling,
        brick_bevel_concavity=brick_cutter_brick_bevel_concavity,
        ...
    )
    rngs_3 = rngs_1[2].spawn(2)

    concrete_base_color_hue = pf.random.uniform(rng=rngs_1[1], low=np.array([0., 0.]),
     high=np.array([1. , 0.1]))
    concrete_base_color_value = pf.random.clip_gaussian(rng=rngs_1[1], mean=0.5, std=0.2,
     low=0.2, high=0.95)
    concrete_base_color = pf.color.hsv_color(
        hue=concrete_base_color_hue[0],
        saturation=concrete_base_color_hue[1],
        value=concrete_base_color_value,
    )
    concrete_roughness = pf.random.uniform(rng=rngs_1[1], low=0.5, high=1.0)
    ...
    surface = concrete.concrete(
        vector=brick_cutter_result.vector,
        base_color=concrete_base_color,
        roughness=concrete_roughness,
        ...
    )
    ...
    return render_cycles.render_cycles(...)
```

Figure 35: **Specific trace.** ProcFunc's Tracer can auto-generate a compute-graph or Python file which expresses the specific numerical parameters which correspond to a particular execution of a randomized procedural generator. Ellipses (...) are code omitted for brevity. This means that randomized procedural generators can automatically emit code for non-randomized procedural generators for each sample. This is useful for editing specific examples, either via the generated Python code or the underlying compute graph it was generated from. It is also plausibly useful for LLM training, by creating many diverse image-text pairs, even with differing code structure due to random control paths.

```python
# uv run procgen bricks_distribution material_cube render_cycles --codegen print
    --trace_level GENERATORS

def execute_generators():
    # Code generated by procfunc v0.18.0-ifg

    ...

    brick_cutter_brick_width = pf.nodes.func.combine_xyz(
        x=0.14953272082744012, y=0.05129953938152119
    )
    brick_cutter_result = brick_cutter.brick_cutter(
        vector=mapping,
        brick_width=brick_cutter_brick_width.x,
        brick_height=brick_cutter_brick_width.y,
        brick_scaling=1.9920081006344654,
        brick_bevel_concavity=0.11666851003155587,\
        ...
    )
    ...
    concrete_base_color = pf.color.hsv_color(
        hue=0.6258491463323542,
        saturation=0.0245507562345461,
        value=0.4491897512413083,
    )

    concrete_micro_grain = pf.control.choice(
        choice_rng=rngs_1[1],
        choice_options=[(0.5572780986583625, 1.0), (0.0, 1.0)],
        chosen_idx=0,
    )
    surface = concrete.concrete(
        vector=brick_cutter_result.vector,
        base_color=concrete_base_color,
        roughness=0.8040807062831756,
        ...
    )
    ...
    return render_cycles.render_cycles(...)
```

C.2 Base Generator

C.2.1 Efficient Wall Geometry

The walls, floor and ceiling of an indoor room present a very large surface area which must be densely subdivided to achieve high quality mesh displacement and accurate geometric ground truth. Whereas Infinigen Indoors (Raistrick et al., 2024) uses either no dense geometry or dense geometry from volumetric marching cubes, our system can create dense mesh geometry only through mesh surface subdivision. Infinigen-Indoors uses mesh booleaning to create openings for doors and windows, which leads to complex, non-convex, non-quadrilateral polygons, which are not amenable to mesh subdivision. To avoid this, we treat room planes as parametric UV surfaces, and create holes for windows or other protrusions without any booleaning. To add a set of windows at a set of locations, we calculate the U and V parameters for the corners of the windows, and simply decline to generate any polygons within that U and V range. This results in mesh geometry which has even quadrilateral topology while still leaving holes for the windows. We then add a Blender subdivision operation, but leave it in an unapplied state, where it is then efficiently executed and displaced by the shader graph upon render startup.

C.2.2 Object Generators

We derive the current library of object-level procedural generators via refactoring generators already present in Infinigen-Indoors (Raistrick et al., 2024). This is *unlike* our material assets and scene-arrangement tools, which are wholly original. So far, we have refactored *Bookcases, Cabinets, Ceiling lights, Desks, Drawers, Flowers, Lamps, Sofas, Tables, Triangle Shelves, Vases, Wall Art, Mirrors and Windows* to use ProcFunc Primitives and Module Interfaces. This involves significant engineering and improvement to the generators - we partially automate the process for the non-randomized components by using the transpiler, but have manually re-implemented and adjusted the random parameter sampling code to use the new traceable ProcFunc standard. We also create weighted lists of materials to be used for various purposes within these objects (e.g. a tabletop may use metal, wood, planks, marble). We also create the needed UV unwrappings to apply our materials.

Besides refactored procedural generators, we also demonstrate that ProcFunc can import arbitrary external assets and use them efficiently in scene arrangement. Particularly we do this for small object assets, which we pre-generate with Infinigen Indoors (Raistrick et al., 2024) from a variety of existing asset classes: *Books, Bottles, Bowls, cans, Chopsticks, Cups, Food Bags, Boxes, Forks, Jars, Knives, Pans, Plates, Pots, Spoons, Towels, Vases, Wine glasses, Large Plants, Small Plants*.

To do this, we generated and exported a library of these assets in obj format. We then filtered out the objects that were over 100MB and only imported ones that were under the limit. Our final library consisted of 279 pre-generated assets.

Infinigen Indoors (Raistrick et al., 2024) does not come with functionality to import and place external assets. Therefore, we implemented asset importing and instancing utilities. The object importer takes a folder of pre-generated assets and imports them into the scene, supporting `dae`, `abc`, `usd`, `obj`, `ply`, `stl`, `fbx`, `glb`, `gltf` and `blend` formats. If the object consists of a tree, the importer collapses it into a single mesh. The user defines an external asset distribution using the following API, which samples a random pre-generated asset from the folder:
`object_distribution = distribution_from_asset_glob(folder)`

The instancing utility takes a parent object and a child object, and instances the child object on top of the parent object using a Poisson disk distribution. This allows us to distribute a large number of small objects while keeping the memory usage down. It finally randomizes the pose of each instance and checks for collisions.

C.2.3 Collision

We implement a collision API that allows for efficient collision testing while being simple from a user perspective. A minimal example of the API is as follows:

```
col = collision_set(objs, existing_set)
collides = intersection_test(col, query_obj):
hits_xyz, index_ray, index_tri = raycast(col, ray_origins, ray_dirs)
```

We use the Trimesh and the FCL libraries to manage the collisions. Every mesh builds an FCL BVH once and every instance is mapped to its source mesh. Thus, we do not store separate meshes for instances. This is crucial in terms of efficient runtime and memory since most small objects are instanced.

C.2.4 GROUND TRUTH

Infinigen's (Raistrick et al., 2023) ground-truth system is tightly coupled with its scene generation code. This means that the user cannot take an arbitrary Blender scene and export the ground-truth from it. Furthermore, Infinigen does not allow the user to customize the ground-truth exports in terms of file type, path, and which types of ground-truth to choose.

Motivated by this, we implement our own ground-truth API, which decouples ground-truth exports from scene generation and allows the user to customize what type of ground-truth they want. Here is an example usage of our API:

```
objects, camera = configure_scene()

render_passes = [
    RenderPass(ExportType.DEPTH, Path("depth.npy"), np.dtype(
        np.float32)),
    RenderPass(ExportType.SURFACE_NORMAL, Path("normal.npy"),
        np.dtype(np.float32)),
    RenderPass(ExportType.OPTICAL_FLOW, Path("flow.npy"), np.
        dtype(np.float32)),
]

if method == "cycles":
    results = render_cycles_ground_truth(
        objects=objects,
        camera=camera,
        output_folder=tmp_path,
        render_passes=render_passes,
        frame_start=1,
        frame_end=10,
        resolution=(256, 256),
    )
elif method == "eevee":
    results = render_eevee_ground_truth(
        objects=objects,
        camera=camera,
        output_folder=tmp_path,
        render_passes=render_passes,
        frame_start=1,
        frame_end=10,
        resolution=(256, 256),
    )
```

Note that our API can be called in any Blender scene, not just ones generated from our system. The filetypes we support include `.png, .exr, .npy, .npz`.

Infinigen uses the Cycles rendering engine to obtain most of its ground-truth types and to render the RGB images. Since Cycles is a physically-based rendering engine, this process is slow. Thus, we implement an additional API to render and export ground-truth using the EEVEE engine, which is a rasterization-based rendering engine. EEVEE is much faster than Cycles, especially for scenes with simple materials. The user can choose which engine to use for their renders and ground-truth.

C.2.5 RRT* CAMERA TRAJECTORY

Infinigen (Raistrick et al., 2023) uses a random-walk-based camera trajectory generation algorithm. The issue is that this algorithm lacks diversity and does not generate interesting trajectories. Hence,

we implement an RRT* (Cheng & LaValle, 2002; Karaman & Frazzoli, 2011)-based camera trajectory generation algorithm. Our algorithm takes a start and a goal point as input. At each iteration, it samples a random point, finds the nearest neighbor in the existing tree, and steers towards the random point from the neighbor. It chooses a distance-minimizing parent from the neighborhood of the new step, and adds the new point to the tree. It also uses raycasting to test if new nodes are valid points in the sense that there are no obstacles in the way. RRT* allows us to generate much more diverse and dynamic trajectories.

D Extended Experiments

D.1 LLM Editing

We provide exact prompts used for our parameter editing (Fig. 37, 36) and from-scratch procedural generator creation settings (Fig. 39,38). These are based on BlenderGym(Gu et al., 2025) with as little adaptation as possible to allow use with ProcFunc and with our new setting.

```
The following code uses the `procfunc` API to produce a material in
    Blender:
```python
{code}
```

Your job is to edit the PARAMETERS of the code to make the render more
    like the goal material. Only edit the parameters, not the structure
    or args.

Answer the following questions:
1) What is the SINGLE most visually obvious difference between the two
    materials shown below?
2) Look at the code. Which fields/variables which are set to numerical
    values are most likely responsible for the obvious visual difference
     in your answer to question 1?
3) Copy the code above (COPY ALL OF IT) and replace the assignments of
    such fields/variables accordingly!

- MAKE SURE YOUR CODE IS RUNNABLE. You do not need to add your material
     to an object, it will be auto-rendered so long as you assign it to
    `shader` at the end of the file
- Only use functions inside `pf.nodes`, not any other way of accessing
    blender. These functions have a similar interface to standard
    blender nodes.
- DO NOT BE BRIEF IN YOUR CODE. DO NOT ABBREVIATE YOUR CODE WITH "..."
    -- TYPE OUT EVERYTHING.
- Return your code in a ```python code block.
```

Figure 36: Prompt template for the `procfunc` backend in the **parameter-edit** setting. The `{code}` placeholder is replaced with the current code at inference time. Two images are provided: the current render and the goal render. This prompt is closely based on BlenderGym's, but has been modified for use with our evaluation setup.

```
The following Blender code was used to produce a material:
```python
{code}
```

Your job is to edit the PARAMETERS of the code to make the render more
    like the goal material. Only edit the parameters, not the structure
    or args.

Answer the following questions:
1) What is the SINGLE most visually obvious difference between the two
    materials shown below?
2) Look at the code. Which fields/variables which are set to numerical
    values are most likely responsible for the obvious visual difference
     in your answer to question 1?
3) Copy the code above (COPY ALL OF IT) and replace the assignments of
    such fields/variables accordingly!

- MAKE SURE YOUR CODE IS RUNNABLE. MAKE SURE TO ASSIGN THE FINAL
    MATERIAL TO `material_obj` (through `apply(material_obj)`) AS THE
    LAST LINE OF YOUR CODE.
- DO NOT BE BRIEF IN YOUR CODE. DO NOT ABBREVIATE YOUR CODE WITH "..."
    -- TYPE OUT EVERYTHING.
- Return your code in a ```python code block.
```

Figure 37: Prompt template for the `infinigen` backend in the **parameter-edit** setting. The `{code}` placeholder is replaced with the current code at inference time. Two images are provided: the current render and the goal render. This prompt is closely based on BlenderGym's, but has been modified for use with our evaluation setup.

```
The following code uses the `procfunc` API to produce a material in
    Blender:
```python
{code}
```

The image shows the desired goal material. Please edit the code above
    to produce a material that matches this goal.

Pay special attention to the base color of the materials, as well as
    any patterns, textures, or details.

- MAKE SURE YOUR CODE IS RUNNABLE. You do not need to add your material
     to an object, it will be auto-rendered so long as you assign it to
    `shader` at the end of the file
- Only use functions inside `pf.nodes`, not any other way of accessing
    blender. These functions have a similar interface to standard
    blender nodes.
- DO NOT BE BRIEF IN YOUR CODE. DO NOT ABBREVIATE YOUR CODE WITH "..."
    -- TYPE OUT EVERYTHING.
- Return your code in a ```python code block.
```

Figure 38: Prompt template for the `procfunc` backend in the **from-scratch** setting. The `{code}` placeholder is replaced with starter code at inference time. A single image (the goal render) is provided. This prompt is closely based on BlenderGym's, but has been modified for use with our evaluation setup.

````
The following Blender code was used to produce a material:
```python
{code}
```
The final material is assigned to the object `material_obj`, a sphere.

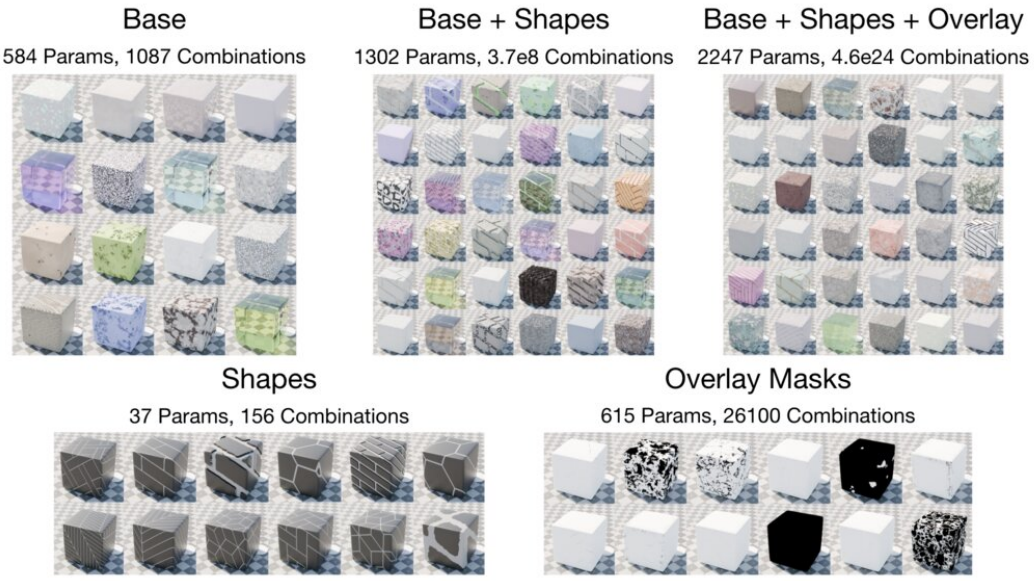
The image shows the desired goal material. Please edit the code above
    to produce a material that matches this goal.

Pay special attention to the base color of the materials, as well as
    any patterns, textures, or details.

MAKE SURE YOUR CODE IS RUNNABLE. MAKE SURE TO ASSIGN THE FINAL MATERIAL
     TO `material_obj` (through `apply(material_obj)`) AS THE LAST LINE
    OF YOUR CODE.
DO NOT BE BRIEF IN YOUR CODE. DO NOT ABBREVIATE YOUR CODE WITH "..." --
     TYPE OUT EVERYTHING.
Return your code in a ```python code block.
````

Figure 39: Prompt template for the `infinigen` backend in the **from-scratch** setting. The `{code}` placeholder is replaced with starter code at inference time. A single image (the goal render) is provided. This prompt is closely based on BlenderGym's, but has been modified for use with our evaluation setup.

Figure 40: **Maximum Diversity Composition** - We provide a random sampling function which combines together all possible arrangements of ProcFunc's material components, rather than just the more ordinary combinations in Fig.9 to 30. This enables a very large number of combinations, including the particular combination of any two materials with any shape and including or not including every type of wear and tear or added layer.



D.2 Procedural Generator Diversity

In Figure 40 we display samples from a material generator constructed to be extremely random and compositional. Unlike the previous material generators tuned for realism, this generator aims to demonstrate the composition modes of base materials, shapes, and overlay masks. We use the tracer to generate a compute graph of the generator down to the level of individual random choices. From this graph, we calculate the total number of continuous parameters and the number of possible combinations from the discrete choices. For example, when combining base materials with shapes, we can have bricks with an arbitrary "brick" material and an arbitrary "grout" material. This creates more than 1 million combinations within bricks alone. Overall, we have 4.6×10^{24} possible combinations after composing base materials, shapes, and overlay masks.

D.3 Procedural Generator Efficiency

The efficiency experiments measure performance and complexity across 10 different settings:

- **Infinigen Indoors (Raistrick et al., 2024) without objects and lowpoly:** We only generate rooms and disable object generation. We use the default settings of Infinigen Indoors and do not use OcMesher (Ma et al., 2025).
- **Our system without objects and lowpoly:** We only generate rooms and do not place objects. We disable subdivisions in room generation.
- **Our system without objects, lowpoly, and rendered with EEVEE:** Same as the previous setting, except we render with the rasterization-based EEVEE rendering engine and use EEVEE ground truth instead of the raytracing-based Cycles. We realize meshes before rendering.
- **Infinigen Indoors (Raistrick et al., 2024) without objects and hipoly:** We only generate rooms and disable object generation. We use the default settings of Infinigen Indoors and use OcMesher (Ma et al., 2025) to get higher detail rooms.

Table 6: Ground-truth render runtimes for Infinigen Indoors and our system. GT Render $t_1 + t_2$ means that the first frame takes t_1 seconds to generate and the remaining frames take t_2 seconds per frame.

Settings			Performance
Method	Obj?	Hipoly?	GT Render ↓
InfIndoors	×	×	**7.65**+5.23s
Ours	×	×	10.46+**2.40s**
Ours Eevee	×	×	71.99+4.15s
InfIndoors	×	✓	938.13+8.74s
Ours	×	✓	**25.86**+**2.91s**
InfIndoors	✓	×	196.17+6.48s
Ours	✓	×	**13.12**+**2.91s**
Ours Eevee	✓	×	105.79+5.21s
InfIndoors	✓	✓	356.68+9.19s
Ours	✓	✓	**30.76**+**3.80s**

- **Our system without objects and hipoly:** We only generate rooms and do not place objects. We enable subdivisions during room generation.
- **Infinigen Indoors (Raistrick et al., 2024) with objects and lowpoly:** We generate full livingrooms in infinigen and enable object generation. We use default settings and do not use OcMesher (Ma et al., 2025).
- **Our system with objects and lowpoly:** We generate livingrooms with objects and disable subdivisions. The subdivisions are disabled for both room generation and object generation.
- **Our system with objects and lowpoly, rendered with EEVEE:** We generate livingrooms with objects and disable subdivisions. The subdivisions are disabled for both room generation and object generation. We realize meshes before rendering and use the EEVEE rendering engine instead of Cycles.
- **Infinigen Indoors full:** We generate Infinigen Indoors livingrooms with objects and use OcMesher (Ma et al., 2025) to subdivide the rooms.
- **Our system full:** We generate livingrooms with objects and enable subdivisions. We render with the Cycles rendering engine.

Our system can run as a single job in a Slurm pipeline. We use 4 Intel(R) Xeon(R) Gold 5320 CPU cores, 48 GB of memory, and 1 NVIDIA L40 GPU per run, i.e. a single livingroom generation.

Infinigen Indoors (Raistrick et al., 2024) runs in multiple jobs. We use 4 Intel(R) Xeon(R) Gold 5320 CPU cores and 24 GB of memory for the scene generation stage, 8 CPU cores, 48 GB memory, and 1 L40 GPU for rendering. There is also a backup rendering stage if initial rendering fails which uses 16 CPU cores, 96 GB of memory, and 2 L40 GPUs. The ground-truth stage uses 8 CPU cores and 48 GB of memory.

In Table 6 we report ground-truth generation times. Our full system is much faster than full Infinigen Indoors ground-truth generation.