Infinite Photorealistic Worlds using Procedural Generation

Alexander Raistrick*, Lahav Lipson*, Zeyu Ma*, (*equal contribution; alphabetical order)
Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han,
Yihan Wang, Alejandro Newell,† Hei Law,† Ankit Goyal,† Kaiyu Yang,† Jia Deng
Department of Computer Science, Princeton University

arXiv:2306.09310v1 [cs.CV] 15 Jun 2023

Abstract

We introduce Infinigen, a procedural generator of photorealistic 3D scenes of the natural world. Infinigen is entirely procedural: every asset, from shape to texture, is generated from scratch via randomized mathematical rules, using no external source and allowing infinite variation and composition. Infinigen offers broad coverage of objects and scenes in the natural world including plants, animals, terrains, and natural phenomena such as fire, cloud, rain, and snow. Infinigen can be used to generate unlimited, diverse training data for a wide range of computer vision tasks including object detection, semantic segmentation, optical flow, and 3D reconstruction. We expect Infinigen to be a useful resource for computer vision research and beyond. Please visit infinigen.org for videos, code and pre-generated data.

1. Introduction

Data, especially large-scale labeled data [13, 53], has been a critical driver of progress in computer vision. At the same time, data has also been a major challenge, as many important vision tasks remain starved of high-quality data. This is especially true for 3D vision, where accurate 3D ground truth is difficult to acquire for real images.

Synthetic data from computer graphics is a promising solution to this data challenge. Synthetic data can be generated in unlimited quantity with high-quality labels. Synthetic data has been used in a wide range of tasks [4,12,46,48,55,58,71], with notable successes in 3D vision, where models trained on synthetic data can perform well on real images zero-shot [28, 54, 82–85, 89].

Despite its great promise, the use of synthetic data in computer vision remains much less common than real images. We hypothesize that a key reason is the limited diversity of 3D assets: for synthetic data to be maximally useful, it needs to capture the diversity and complexity of the real world, but existing freely available synthetic datasets are mostly restricted to a fairly narrow set of objects and shapes, often driving scenes (e.g. [32, 71]) or human-made objects in indoor environments (e.g. [21, 56]).

In this work, we seek to substantially expand the coverage of synthetic data, particularly objects and scenes from the natural world. We introduce Infinigen, a procedural generator of photorealistic 3D scenes of the natural world. Compared to existing sources of synthetic data, Infinigen is unique due to the combination of the following properties:

- *Procedural:* Infinigen is not a finite collection of 3D assets or synthetic images; instead, it is a *generator* that can create infinitely many distinct shapes, textures, materials, and scene compositions. Every asset, from shape to texture, is entirely procedural, generated from scratch via randomized mathematical rules that allow infinite variation and composition. This sets it apart from datasets or dataset generators that rely on external assets.

- *Diverse:* Infinigen offers a broad coverage of objects and scenes in the natural world, including plants, animals, terrains, and natural phenomena such as fire, cloud, rain, and snow.

- *Photorealistic:* Infinigen creates highly photorealistic 3D scenes. It achieves high photorealism by procedurally generating not only coarse structures but also fine details in geometry and texture.

- *Real geometry*: unlike in video game assets, which often use texture maps to fake geometrical details (e.g. a surface appears rugged but is in fact flat), all geometric details in Infinigen are real. This ensures accurate geometric ground truth for 3D reconstruction tasks.

- *Free and open-source:* Infinigen builds on top of Blender [11], a free and open-source graphics tool. Infinigen's code is released for free under the GPL license, same as Blender. Anyone can freely use Infinigen to obtain unlimited assets and renders [‡].

†work done while a student at Princeton University

‡The output of GPL code is generally not covered by GPL. See www.gnu.org/licenses/gpl-faq.en.html#WhatCaseIsOutputGPL

Figure 1. Randomly generated, *non cherry-picked* images produced by our system. From top left to bottom right: Forest, River, Underwater, Caves, Coast, Desert, Mountain and Plains. See Appendix for more samples.

Infinigen focuses on the natural world for two reasons. First, accurate perception of natural objects is demanded by many applications, including geological survey, drone navigation, ecological monitoring, rescue robots, agriculture automation, but existing synthetic datasets have limited coverage of the natural world. Second, we hypothesize that the natural world alone can be sufficient for pretraining powerful "foundation models"—the human visual system was evolved entirely in the natural world; exposure to human-made objects was likely unnecessary.

Infinigen is useful in many ways. It can serve as a generator of unlimited training data for a wide range of computer vision tasks, including object detection, semantic segmentation, pose estimation, 3D reconstruction, view synthesis, and video generation. Because users have access to all the procedural rules and parameters underlying each 3D scene, Infinigen can be easily customized to generate a large variety of task-specific ground truth. Infinigen can also serve as a generator of 3D assets, which can be used to build simulated environments for training physical robots as well as virtual embodied agents. The same 3D assets are also useful for 3D printing, game development, virtual reality, film production, and content creation in general.

We construct Infinigen on top of Blender [11], a graphics system that provides many useful primitives for procedural generation. Utilizing these primitives we design and implement a library of procedural rules to cover a wide range of natural objects and scenes. In addition, we develop utilities that facilitate creation of procedural rules and enable all Blender users including non-programmers to contribute; the utilities include a transpiler that automatically converts Blender node graphs (intuitive visual representation of procedural rules often used by Blender artists) to Python code. We also develop utilities to render synthetic images and extract common ground truth labels including depth, occlusion boundaries, surface normals, optical flow, object category, bounding boxes, and instance segmentation. Constructing Infinigen involves substantial software engineering: the latest main branch of the Infinigen codebase consists of 40,485 lines of code.

In this paper, we provide a detailed description of our procedural system. We also perform experiments to validate the quality of the generated synthetic data; our experiments suggest that data from Infinigen is indeed useful, especially for bridging gaps in the coverage of natural objects. Finally, we provide an analysis on computational costs including a detailed profiling of the generation pipeline.

We expect Infinigen to be a useful resource for computer vision research and beyond. In future work, we intend to make Infinigen a living project that, through open-source collaboration with the whole community, will expand to cover virtually everything in the visual world.

2. Related Work

Synthetic data from computer graphics have been used in computer vision for a wide range of tasks [56, 71]. We refer the reader to [65] for a comprehensive survey. Below we categorize existing work in terms of application domain, generation method, and accessibility. Tab. 1 provides detailed comparisons.

Application Domain. Synthetic datasets or dataset generators have been developed to cover a variety of domains. The

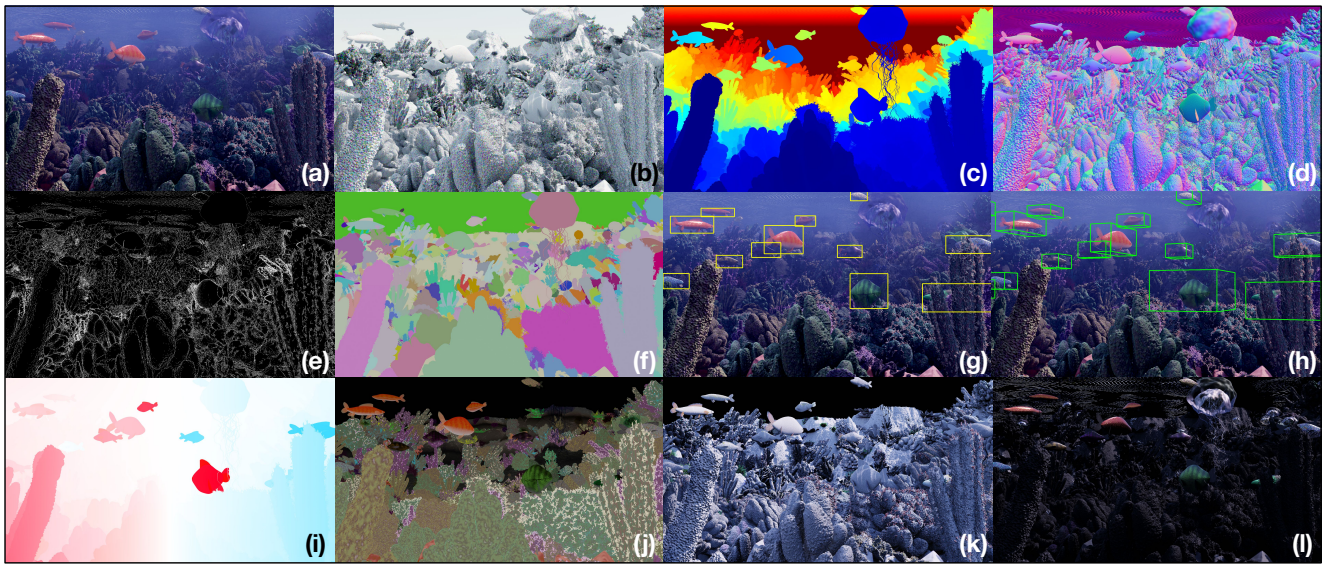


Figure 2. For each image (a), we have a high-res mesh (b), which readily yields Depth (c), Surface Normals (d), Occlusion Boundaries (e), Instance Segmentation masks (f), and 2D / 3D bounding boxes (g/h). From rendering metadata, we obtain Optical Flow (i), and material parameters such as Albedo (j), Lighting Intensity (k) and Specular Reflection (l).

Synthetic Dataset	Domain	# Triangles Per-Scene	# Scenes in Total	# Assets in Total	Free Assets	Procedural Arrangement	Procedural Assets	Provides Procedural Code	External Asset Source
GTA-V [71]	Driving, Urban	-	-	-	No	No	No	N/A	Grand Theft Auto
MOTSynth [18]	Urban	-	-	-	No	No	No	N/A	Grand Theft Auto
MVS-Synth [32]	Driving, Urban	-	-	-	No	No	No	N/A	Grand Theft Auto
DeformingThings4D [50]	Animals/Humanoids	-	2K	2K	Yes	No	No	N/A	Adobe Mixamo [34]
DeepFurniture [56]	Indoor	-	20K	-	No	No	No	N/A	Professional Designers
Robotrix [21]	Indoor	-	16	-	No	No	No	N/A	UE4Arch, UnrealEngine Marketplace [36]
SUNCG [76] (+ [51, 75, 95])	Indoor	-	46K	2.6K	No	No	No	N/A	Planner5D [1]
TartanAir [90]	In/Outdoor, Natural/Urban	-	30	-	No	No	No	N/A	UnrealEngine Marketplace [36]
Hypersim [72]	Indoor	100K-11M	461	59K	No ($6000)	No	No	N/A	Evermotion Architectures [17]
OpenRooms [52]	Indoor	1M	1.3K	3K	No ($500)	No	No	N/A	Scan2CAD [3], ShapeNet [10], Adobe Stock [35]
Sintel [9]	Medieval, Natural	300K	27	-	No ($12)	No	No	N/A	Blender Foundation [11]
Spring [63]	Natural	-	47	-	No ($12)	No	No	N/A	Blender Foundation [11]
Structured3D [96]	Indoor	-	22K	472K	No	No	No	N/A	Professional Designers
SceneNet-RGBD [61]	Indoor	420K	57	5.1K	Yes	No	No	N/A	ShapeNet [10], SceneNet [26]
3D-Front [19]	Indoor	60K	19K	13K	Yes	No	No	N/A	3D-FUTURE [20]
Jiang et al. [39]	Indoor	-	∞	54K	No	Yes	No	No	ShapeNet [10], Planner 5D [1]
InteriorNet [49]	Indoor	-	22M	1M	No	Yes	No	No	Manufacturers / Kujiale [44]
FaceSynthetics [91]	Faces	7.4K	-	∞	No	N/A	Partial	No	Artist-Created Faces (textures, hair, clothing)
Meta-Sim2 [14]	Driving, Urban	-	∞	∞	No	Yes	Partial	No	-
Synscapes [87, 92]	Driving, Urban	-	25K	-	No	Yes	Partial	No	7D-Labs [45]
ProcSy [41]	Driving, Urban	-	∞	∞	Yes	Yes	Partial	No	CityEngine, OpenStreetMap [25], Manual Annotation
ProcTHOR [12]	Indoor	-	∞	1.6K	Yes	Yes	No	Yes	AI2-THOR [43], Professional Designers
Kubric [24]	Scattered Objects	161K	∞	52K	Yes	Yes	No	Yes	ShapeNet [10], Google Scanned Objects [16]
Infinigen (Ours)	Natural	Dynamic (16M @ 1080p)	∞	∞	Yes	Yes	Yes	Yes	None

Table 1. Comparison to existing synthetic datasets or generators. Ours is entirely procedural, relying on no external assets, and can produce infinite original assets and scenes. Many existing datasets use external, static asset libraries. Procedural generation is often limited to object placement or a subset of objects. The vast majority of datasets are also restricted to the built environment, especially indoor scenes. In terms of accessibility, many do not provide free assets or make code available. Many works do not report average triangles per scene; where possible, we calculate this using generous assumptions from the numbers they do report. Dashes represent numbers we were not able to obtain or estimate. In counting the number of assets, we exclude trivial modifications like re-lighting and re-scaling.

built environment has been covered by the largest amount of existing work [9, 18, 24, 48, 50, 60, 86, 90] especially indoor scenes [12, 21, 39, 47, 51, 52, 56, 72, 75, 76, 80, 95, 96] and urban scenes [14, 18, 32, 41, 71, 87, 92]. A significant source of synthetic data for the built environment comes from simulated platforms for embodied AI, such as AI2-THOR [43], Habitat [81], BEHAVIOR [78], SAPIEN [93], RLBench [37], CARLA [15]. Some datasets, such as TartanAir [90] and Sintel [9], include a mix of built and natural environments. There also exist datasets such as FlyingThings [60], FallingThings [86] and Kubric [24] that do not render realistic scenes and instead scatter (mostly artificial) objects against simple backgrounds. Synthetic humans are another important application domain, where high-quality synthetic data have been generated for understanding faces [91], pose [2, 88], and activity [40, 77].

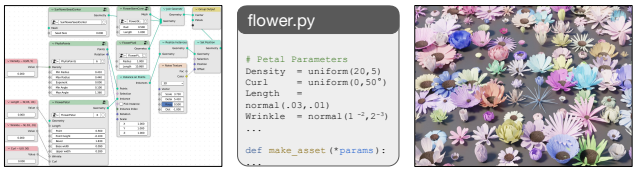


Figure 3. Our *Node Transpiler* converts artist-friendly Node-Graphs (left) to procedural code (middle) which produces assets (right).

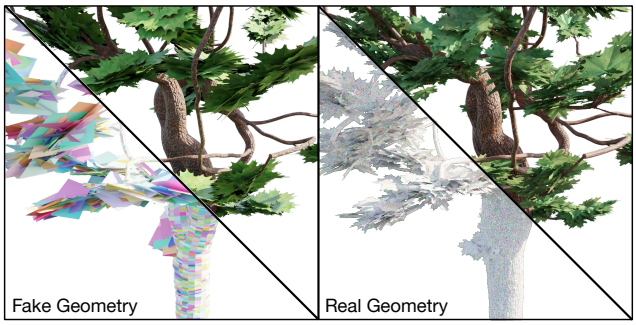


Figure 4. Real-time optimized assets (left) often use low res. geometry in conjunction with shading tricks and alpha-masked image textures to give the illusion of geometric detail. Infinigen assets (right) instead model objects in full geometric detail. Bottom left triangles show a random color per mesh face.

Figure 5. Examples of a subset of our material generators. Columns 1-4 are for terrain, 5-7 are for creatures, and 8 is miscellaneous.

Some datasets focus on objects, not full scenes, to serve object-centric tasks such as non-rigid reconstruction [50], view synthesis [64], and 6D pose [31].

We focus on natural objects and natural scenes, which have had limited coverage in existing work. Even though natural objects do occur in many existing datasets such as urban driving, they are mostly on the periphery and have limited diversity.

Generation Method. Most synthetic datasets are constructed by using a *static* library of 3D assets, either externally sourced or made in house. The downside of a static library is that the synthetic data would be easier to overfit. Procedural generation has been involved in some existing datasets or generators [12, 24, 29, 39, 49], but is limited in scope. Procedural generation is only applied to either object arrangement or a subset of objects, e.g. only buildings and roads but not cars [41, 92]. In contrast, Infinigen is entirely procedural, from shape to texture, from macro structures to micro details, without relying on any external asset.

Asset Type	Num. Generators	Interpretable DOF
Terrain	26	17
Materials	50	271
Weather, Fluid	19	61
Rocks	4	12
Small Plants	30	258
Trees	3	26
Creatures	39	315
Scattering	11	110
Total	182	1070

Table 2. Approximate degrees of freedom, as a proxy of overall diversity. We count only distinct human-understandable parameters with useful ranges of interpolation, with the caveat that this could be an overestimate as not all parameters are fully independent. Some asset classes (e.g terrain) are based on physics simulation and have many more *internal* degrees of freedom not counted here. See Appendix D for a full list of named parameters.

Accessibility. A synthetic dataset or generator is most useful if it is maximally accessible, i.e. it provides free access to assets and code with minimum use restrictions. However, few existing works are maximally accessible. Often the rendered images are provided, but underlying 3D assets are unavailable, not free, or have significant use restrictions. Moreover, the code for procedural generation, if any, is often unavailable.

Infinigen is maximally accessible. Its *code* is available under the GPL license. Anyone can freely use Infinigen to generate unlimited assets.

3. Method

Procedural Generation. Procedural generation refers to the creation of data through generalized rules and simulators. Where an artist might manually create the structure of a single tree by eye, a procedural system creates infinite trees by coding their structure and growth in generality. Developing procedural rules is a form of world modeling using compact mathematical language.

Blender Preliminaries. We develop procedural rules primarily using Blender, an open-source 3D modelling software that provides various primitives and utilities. Blender represents scenes as a hierarchy of posed objects. Users modify this representation by transforming objects, adding primitives, and editing meshes. Blender provides import/export for most common 3D file-formats. Finally, all operations in Blender can be automated using its Python API, or by inspecting its open-source code.

For more complex operations, Blender provides an intuitive node-graph interface. Rather than directly edit shader code to define materials, artists edit *Shader Nodes* to compose primitives into a photo-realistic material. Similarly, *Geometry Nodes* define a mesh using nodes representing

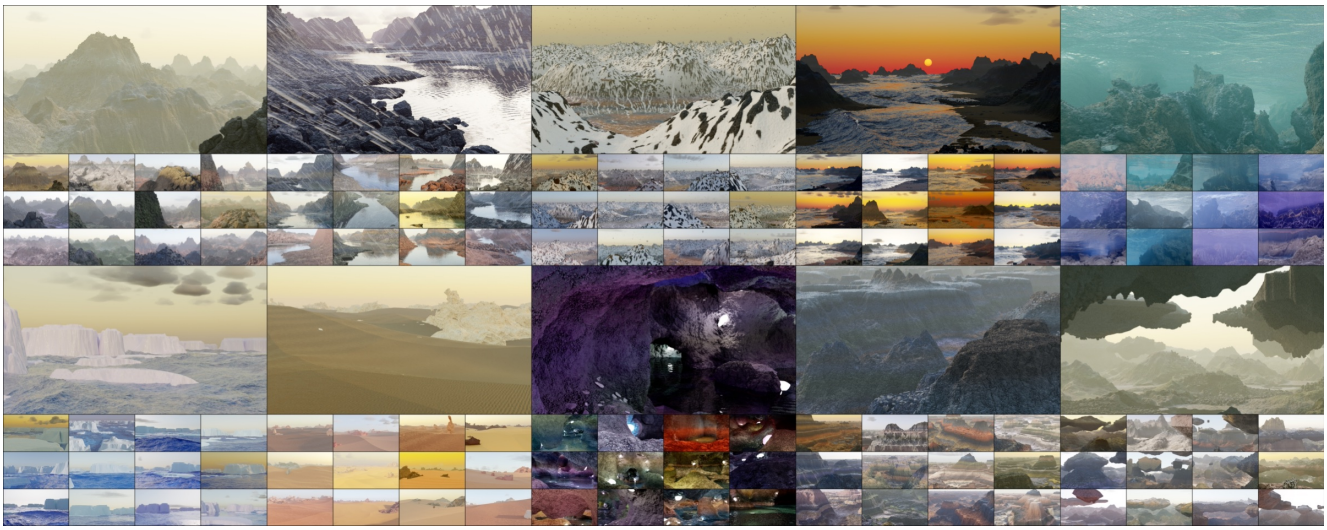


Figure 6. Random, *non cherry-picked* terrain-only scenes. We sample 13 images for various natural scene types. From top left to bottom right; Mountains, Rainy river, Snowy mountains, Coastal sunrise, Underwater, Arctic icebergs, Desert, Caves, Canyons and Floating islands. See Appendix for more samples.

Figure 7. Random, *non cherry-picked* images of simulated fire, smoke, waterfalls, and volcano eruptions.

Figure 8. Random, *non cherry-picked* leaves, flowers, mushrooms and pinecones.

operators such as Poisson disk sampling, mesh boolean, extrusion etc. A finalized Geometry Node Tree is a generalized parametric CAD model, which produces a unique 3D object for each combination of its input parameters. These tools are intuitive and widely adopted by 3D artists.

Although we use Blender heavily, not all of our procedural modeling is done using node-graphs; a significant portion of our procedural generation is done outside Blender and only loosely interacts with Blender.

Node Transpiler. As part of Infinigen, we develop a suite of new tools to speed up our procedural modeling. A notable example is our *Node Transpiler*, which automates the process of converting node-graphs to Python code, as shown in Fig. 3. The resulting code is more general, and allows us to randomize graph *structure* not just input parameters. This tool makes node-graphs more expressive and allows easy integration with other procedural rules developed directly in Python or C++. It also allows non-programmers to contribute Python code to Infinigen by making node-graphs. See Appendix E for more details.

***Generator* Subsystems**. Infinigen is organized into *generators*, which are probabilistic programs each specialized to produce one subclass of assets (e.g. mountains or fish). Each has a set of high-level parameters (e.g. the overall height of a mountain), which reflect the external degrees of freedom controllable by the user. By default, we randomly sample these parameters according to distributions tuned to mirror the natural world, with no input from the user. However,

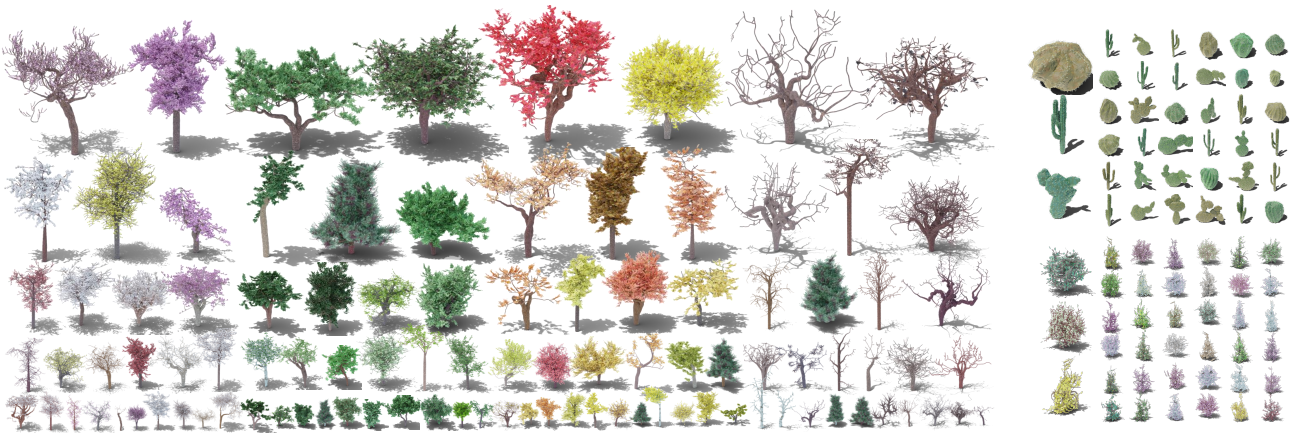


Figure 9. Random, *non cherry-picked* procedural trees (left), cacti (top right) and bushes (bottom right).

Figure 10. Random, *non cherry-picked* underwater objects.

Figure 11. Random, *non cherry-picked* surface scatters. Dense coverage with procedural assets turns any surface into a convincing *grassland*, *sea floor* or *forest floor* environment.

users can also override any parameter using our Python API to achieve fine grained control of data generation.

Each probabilistic program involves many additional internal, low-level degrees of freedom (e.g. the heights of every point on a mountain). Randomizing over both the internal and external degrees of freedom leads to a distribution of assets which we sample from for unlimited generation. Tab. 2 summarizes the number of human-interpretable degrees of freedom in Infinigen, with the caveat that the numbers could be an over-estimation because not all parameters are fully independent. Note that it is hard to quantify the internal degrees of freedom, so the external degrees of freedom serve as a lower bound of the total degrees of freedom for our system.

Material Generators. We provide 50 procedural material generators (Fig. 5). Each is composed of a randomized shader, specifying color and reflectance, and a local geometry generator, which generates corresponding fine geometric details.

The ability to produce accurate ground-truth geometry is a key feature of our system. This precludes the use of many common graphics techniques such as Bump Mapping and Phong Interpolation [7, 70]. Both manipulate face normals to give the illusion of detailed geometric textures, but do so in a way that cannot be represented as a mesh. Similarly, artists often rely on image textures or alpha channel masking to give the illusion of high res. meshes where none exist. All such shortcuts are excluded from our system. See Fig. 4 for an illustrative example of this distinction.

Terrain Generators. We generate terrain (Fig. 6) using SDF elements derived from fractal noise [68] and simulators [6,30,33,62,68]. We evaluate these to a mesh using marching cubes [57]. We generate boulders via repeated extrusion, and small stones using Blender's built-in addon. We simulate dynamic fluids (Fig. 7) using FLIP [8], sun/sky light using the Nishita sky model [66], and weather with Blender's particle system.

Plants & Underwater Object Generators. We model tree growth with random walks and space colonization [73], resulting in a system with diverse coverage of various trees, bushes and even some cacti (Fig. 9). We provide generators for a variety of corals (Fig. 10) using Differential Growth [67], Laplacian Growth [42], and Reaction-Diffusion [23]. We produce Leaves (Fig. 8), Flowers [38], Seaweed, Kelp, Mollusks and Jellyfish using geometry node-graphs.

Surface Scatter Generators. Some natural environments are characterized by a dense coverage of smaller objects. To this end, we provide several scatter generators, which

Figure 12. Creature Generation. Our system automatically generates genomes (a), parts (b), assembly (c), materials (d) and animation rigs (e). On the right, we show random, *non cherry-picked* samples from our *Carnivore*, *Herbivore*, *Bird*, *Beetle*, and *Fish* generators.

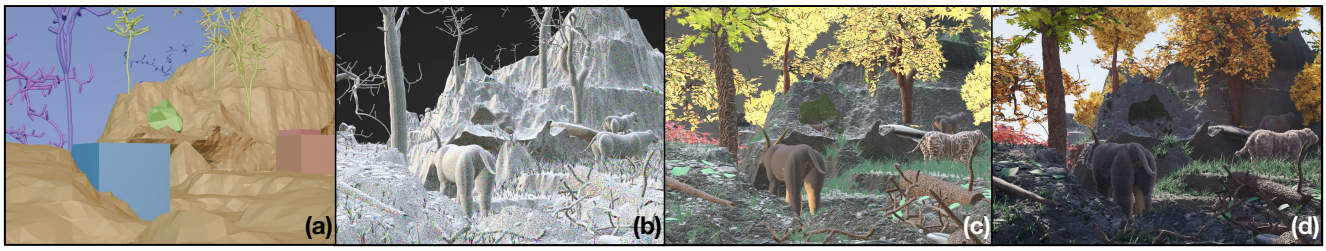


Figure 13. Data Generation Pipeline. We procedurally compose a scene layout (a) with random camera poses. We generate all necessary assets (b, showing a color per mesh face), and apply procedural materials and displacement (c). Finally, we render a photo-real image (d).

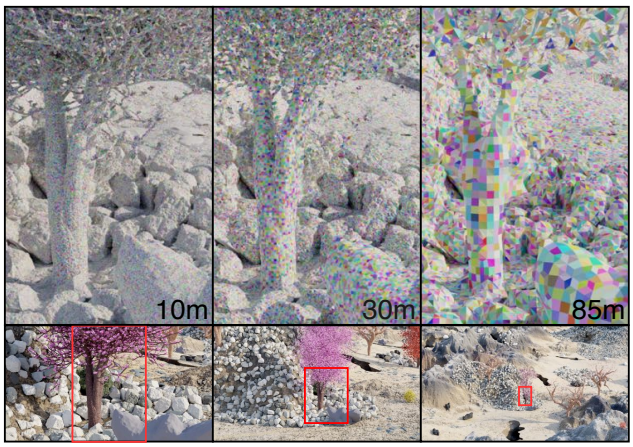


Figure 14. Dynamic Resolution Scaling. We show close-up mesh visualizations (top) of the same content for three different camera distances. Despite differing mesh resolution, no changes are visible in the final images (bottom).

combine one or more existing assets in a dense layer (Fig. 11). In the forest floor example, we generate fallen tree logs by procedurally fracturing entire trees from our tree system.

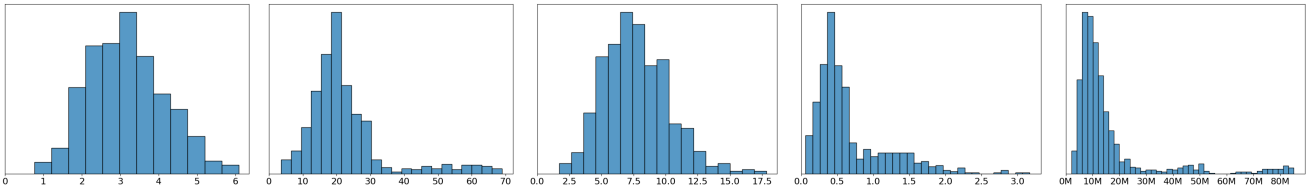
Due to space constraints, all specific implementation details of the above are available in Appendix G.

Creature Generators. The genome of each creature is represented as a tree data-structure (Fig. 12 a). This reflects the topology of real creatures, whose limbs don't form closed loops. Nodes contain part parameters, and edges specify part attachment. We provide generators for 5 classes of realistic creature genomes, shown in Fig. 12. We can also combine creature parts at random, or interpolate similar genomes. See Appendix G.6 for details.

Each part generator is either a transpiled node-graph, or a non-uniform rational basis spline (NURBS). NURBS parameter-space is high-dimensional, so we randomize NURBS parameters under a factorization inspired by lofting, composed of deviations from a center curve. To tune the random distribution, we modelled 30 example heads and bodies, and ensured that our distribution supports them.

Our system produces high-quality animation rigs, and optionally simulates realistic surface folding, sagging and motion of creature skin using cloth simulation. For hair, we use the transpiler to automate the process of grooming hairs, as usually performed by human character artists.

Dynamic Resolution Scaling. With the camera location fixed, we evaluate our procedural assets at precisely the level of detail such that each face is $<$ 1px in size when rendered. This process is visualized in Fig. 14. For most assets, this entails evaluating a parametric curve at the given pixel size, or using Blender's built-in subdivision or re-meshing. For terrain, we perform Marching Cubes on SDF points in *spherical coordinates*. For densely scattered assets (incl. all assets in Fig. 11) we use *instancing* - that is, we generate a fixed number of assets of each type, and reuse them with random transforms within a scene. Even with this effort in optimization, the average complete scene has 16M polygons.



(a) Wall Time (Hours) (b) Memory (GB) (c) CPU Hours (d) GPU Hours (e) # Triangles per scene

Figure 15. Resource requirements to create a pair of stereo 1080p images using Infinigen.

Training Dataset	Adirondack	**Jadeplant**	Motorcycle	Piano	Pipes	Playroom	Playtable	Recycle	Shelves	Vintage	Avg
FallingThings [86]	8.3	43.3	**12.3**	18.2	25.3	29.7	50.0	10.4	43.3	45.6	28.6
Sintel-Stereo [9]	35.7	62.9	31.1	24.1	31.9	41.7	60.1	30.8	55.8	76.1	45.0
HR-VS [94]	43.5	43.2	17.0	29.6	32.1	34.6	68.4	24.7	57.4	34.9	38.5
Li et al. [48]	23.9	80.2	40.7	32.0	40.3	49.1	67.5	36.6	51.7	42.3	46.4
SceneFlow [60]	**7.4**	41.3	14.9	16.2	33.3	**18.8**	**38.6**	**10.2**	**39.1**	29.9	**25.0**
TartanAir [90]	15.5	45.1	18.1	**12.9**	28.4	25.6	51.0	20.9	49.1	**28.2**	29.5
InStereo2K [5]	17.1	59.7	21.3	23.8	35.8	33.9	**36.4**	20.0	**33.4**	44.1	32.5
Ours (Infinigen 30K)	**7.4**	**35.2**	15.2	20.7	**24.7**	29.3	50.0	12.6	55.1	46.9	29.7

Table 3. Bad 3.0 (%) ↓ error on the Middlebury [74] validation set. Infinigen generalizes well to natural objects (e.g. Jadeplant). However, natural objects contain very few planar or textureless surfaces; models trained exclusively on natural objects generalize less well on Middlebury's indoor scenes.

Image Rendering & Ground Truth Extraction. We render images using Cycles, Blender's physically-based path tracing renderer. We provide code to extract ground truth for common tasks, visualized in Fig. 2.

Cycles individually traces photons of light to accurately simulate diffuse and specular reflection, transparent refraction and volumetric effects. We render at 1920×1080 resolution using $10,000$ random samples per-pixel, which is standard for blender artists and ensures almost no sampling noise in the final image.

Prior datasets [9, 24, 27, 29, 48] rely on blender's built-in render-passes to obtain dense ground truth. However, these rendering passes are a byproduct of the rendering pipeline and not intended for training neural networks. Specifically, they are often incorrect due to translucent surfaces, volumetric effects, motion blur, focus blur or sampling noise. See Appendix C.2 for examples of these issues.

Instead, we provide OpenGL code to extract surface normals, depth and occlusion boundaries from the mesh directly without relying on blender. This solution has many benefits in addition to its accuracy. Users can exclude objects not relevant to their task (e.g. water, clouds, or any other object) independently of whether they are rendered. Many annotations like occlusion boundaries are also plainly not supported by Blender. Finally, our implementation is modular, and we anticipate that users will generate task-specific ground truth not covered above via simple extensions to our codebase.

Runtime. We benchmark our Infinigen on 2 *Intel(R) Xeon(R) Silver 4114 @ 2.20GHz* CPUs and 1 NVidia-GPU across 1000 independent trials. The wall time to produce a pair of 1080p images is 3.5 hours. Statistics are shown in Fig. 15.

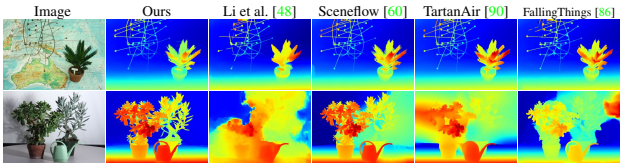


Figure 16. Qualitative results on the *Plant* and *Australia* Middlebury [74] test images. RAFT-Stereo trained using Infinigen generalizes well to images with natural objects.

4. Experiments

To evaluate Infinigen, we produced 30K image pairs with ground truth for rectified stereo matching. We train RAFT-Stereo [54] on these images from scratch and compare results on the Middlebury validation (Tab. 3) and test sets (Fig. 16). See Appendix for qualitative results on in-the-wild natural photographs.

5. Contributions & Acknowledgements

Alexander Raistrick, Lahav Lipson and Zeyu Ma contributed equally, and are ordered alphabetically by first name; each has the right to list their name first in their CV. Alexander Raistrick performed team coordination, and developed the creature system, transpiler and scene composition. Lahav Lipson trained models and implemented dense annotations and rendering. Zeyu Ma developed the terrain system and camera selection. Lingjie Mei created coral, sea invertebrates, small plants, boulders & moss. Mingzhe Wang developed materials for creatures, plants and terrain. Yiming Zuo developed trees and leaves. Karhan Kayan created liquids and fire. Hongyu Wen & Yihan Wang developed creature parts and materials. Beining Han created ferns and small plants. Alejandro Newell designed the tree & bush system. Hei Law created weather and clouds. Ankit Goyal developed terrain materials. Kaiyu Yang developed an initial prototype with randomized shapes. Jia Deng conceptualized the project, led the team, and set the directions.

We thank Zachary Teed for helping with the early prototypes. This work was partially supported by the Office of Naval Research under Grant N00014-20-1-2634 and the National Science Foundation under Award IIS-1942981.

References

[1] Planner 5D. Planner 5d. https://planner5d.com/. 3

[2] Mykhaylo Andriluka, Leonid Pishchulin, Peter Gehler, and Bernt Schiele. 2D human pose estimation: New benchmark and state of the art analysis. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3686–3693, 2014. 3

[3] Armen Avetisyan, Manuel Dahnert, Angela Dai, Manolis Savva, Angel X Chang, and Matthias Nießner. Scan2CAD: Learning CAD model alignment in RGB-D scans. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 3

[4] Shaojie Bai, Zhengyang Geng, Yash Savani, and J Zico Kolter. Deep equilibrium optical flow estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 620–630, 2022. 1

[5] Wei Bao, Wei Wang, Yuhua Xu, Yulan Guo, Siyu Hong, and Xiaohu Zhang. Instereo2k: A large real dataset for stereo matching in indoor scenes. *Science China Information Sciences*, 63(11):1–11, 2020. 8, 13, 18

[6] Katherine R Barnhart, Eric WH Hutton, Gregory E Tucker, Nicole M Gasparini, Erkan Istanbulluoglu, Daniel EJ Hobley, Nathan J Lyons, Margaux Mouchene, Sai Siddhartha Nudurupati, Jordan M Adams, et al. Landlab v2. 0: a software package for earth surface dynamics. *Earth Surface Dynamics*, 8(2):379–397, 2020. 6, 22

[7] James F. Blinn. Simulation of wrinkled surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):286–292, aug 1978. 6

[8] J. U. Brackbill, D. B. Kothe, and H. M. Ruppel. Flip: A low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, 48(1):25–38, Jan. 1988. 6, 22

[9] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In *European Conference on Computer Vision (ECCV)*, pages 611–625, 2012. 3, 8, 13, 18, 36

[10] Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. ShapeNet: An information-rich 3D model repository. *arXiv preprint arXiv:1512.03012*, 2015. 3

[11] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. 1, 2, 3, 22

[12] Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Jordi Salvador, Kiana Ehsani, Winson Han, Eric Kolve, Ali Farhadi, Aniruddha Kembhavi, et al. ProcTHOR: Large-scale embodied AI using procedural generation. *arXiv preprint arXiv:2206.06994*, 2022. 1, 3, 4

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 1

[14] Jeevan Devaranjan, Amlan Kar, and Sanja Fidler. Meta-Sim2: Unsupervised learning of scene structure for synthetic data generation. In *European Conference on Computer Vision (ECCV)*, pages 715–733. Springer, 2020. 3

[15] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Conference on Robot Learning (CoRL)*, pages 1–16, 2017. 3

[16] Laura Downs, Anthony Francis, Nate Koenig, Brandon Kinman, Ryan Hickman, Krista Reymann, Thomas B McHugh, and Vincent Vanhoucke. Google Scanned Objects: A high-quality dataset of 3d scanned household items. *arXiv preprint arXiv:2204.11918*, 2022. 3

[17] Evermotion. Evermotion architectures. https://evermotion.org/shop. 3

[18] Matteo Fabbri, Guillem Brasó, Gianluca Maugeri, Orcun Cetintas, Riccardo Gasparini, Aljoša Ošep, Simone Calderara, Laura Leal-Taixé, and Rita Cucchiara. MOTSynth: How can synthetic data help pedestrian detection and tracking? In *International Conference on Computer Vision (ICCV)*, 2021. 3

[19] Huan Fu, Bowen Cai, Lin Gao, Ling-Xiao Zhang, Jiaming Wang, Cao Li, Qixun Zeng, Chengyue Sun, Rongfei Jia, Binqiang Zhao, et al. 3D-FRONT: 3D furnished rooms with layouts and semantics. In *International Conference on Computer Vision (ICCV)*, pages 10933–10942, 2021. 3

[20] Huan Fu, Rongfei Jia, Lin Gao, Mingming Gong, Binqiang Zhao, Steve Maybank, and Dacheng Tao. 3D-FUTURE: 3D furniture shape with texture. *International Journal of Computer Vision (IJCV)*, 129(12):3313–3337, 2021. 3

[21] Alberto Garcia-Garcia, Pablo Martinez-Gonzalez, Sergiu Oprea, John Alejandro Castro-Vargas, Sergio Orts-Escolano, Jose Garcia-Rodriguez, and Alvaro Jover-Alvarez. The robotrix: An extremely photorealistic and very-large-scale indoor dataset of sequences with robot trajectories and interactions. In *International Conference on Intelligent Robots and Systems (IROS)*, 2018. 1, 3

[22] Ryan Geiss. Generating complex procedural terrains using the gpu. *GPU gems*, 3(7):37, 2007. 22

[23] Peter. Gray and Stephen K. Scott. *Chemical Oscillations and Instabilities: Non-linear Chemical Kinetics*. International series of monographs on chemistry. Clarendon Press, 1994. 6, 26

[24] Klaus Greff, Francois Belletti, Lucas Beyer, Carl Doersch, Yilun Du, Daniel Duckworth, David J Fleet, Dan Gnanapragasam, Florian Golemo, Charles Herrmann, et al. Kubric: A scalable dataset generator. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 3, 4, 8, 18, 36

[25] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008. 3

[26] Ankur Handa, Viorica Patraucean, Vijay Badrinarayanan, Simon Stent, and Roberto Cipolla. Understanding real world indoor scenes with synthetic data. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4077–4085, 2016. 3

[27] Yana Hasson, Gül Varol, Dimitris Tzionas, Igor Kalevatykh, Michael J. Black, Ivan Laptev, and Cordelia Schmid. Learning joint reconstruction of hands and manipulated objects. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 8, 18, 36

[28] Rasmus Laurvig Haugaard and Anders Glent Buch. Surfemb: Dense and continuous correspondence distributions for object pose estimation with learnt surface embeddings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6749–6758, 2022. 1

[29] Ju He, Enyu Zhou, Liusheng Sun, Fei Lei, Chenyang Liu, and Wenxiu Sun. Semi-synthesis: A fast way to produce effective datasets for stereo matching. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 4, 8, 18, 36

[30] Daniel EJ Hobley, Jordan M Adams, Sai Siddhartha Nudurupati, Eric WH Hutton, Nicole M Gasparini, Erkan Istanbulluoglu, and Gregory E Tucker. Creative computing with landlab: an open-source toolkit for building, coupling, and exploring two-dimensional numerical models of earth-surface dynamics. *Earth Surface Dynamics*, 5(1):21–46, 2017. 6, 22

[31] Tomáš Hodaň, Martin Sundermeyer, Bertram Drost, Yann Labbé, Eric Brachmann, Frank Michel, Carsten Rother, and Jiří Matas. BOP challenge 2020 on 6D object localization. *European Conference on Computer Vision Workshops (ECCVW)*, 2020. 4

[32] Po-Han Huang, Kevin Matzen, Johannes Kopf, Narendra Ahuja, and Jia-Bin Huang. DeepMVS: Learning multi-view stereopsis. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2821–2830, 2018. 1, 3

[33] Eric Hutton, Katy Barnhart, Dan Hobley, Greg Tucker, Sai Nudurupati, Jordan Adams, Nicole Gasparini, Charlie Shobe, Ronda Strauch, Jenny Knuth, Margaux Mouchene, Nathan Lyons, David Litwin, Rachel Glade, Giuseppecipolla95, Amanda Manaster, Langston Abby, Kristen Thyng, and Francis Rengers. landlab, 4 2020. 6, 22

[34] Adobe Inc. Adobe mixamo. https://www.mixamo.com. 3

[35] Adobe Inc. Adobe stock. https://stock.adobe.com/3d-assets. 3

[36] Epic Games Inc. Unreal engine marketplace. https://www.unrealengine.com/marketplace/en-US/store. 3

[37] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. RLBench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020. 3

[38] Roger V Jean. Introductory review: Mathematical modeling in phyllotaxis: The state of the art. *Mathematical Biosciences*, 64(1):1–27, 1983. 6

[39] Chenfanfu Jiang, Siyuan Qi, Yixin Zhu, Siyuan Huang, Jenny Lin, Lap-Fai Yu, Demetri Terzopoulos, and Song-Chun Zhu. Configurable 3D scene synthesis and 2D image rendering with per-pixel ground truth using stochastic grammars. *International Journal of Computer Vision (IJCV)*, 126(9):920–941, 2018. 3, 4

[40] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, et al. The kinetics human action video dataset. *arXiv preprint arXiv:1705.06950*, 2017. 3

[41] Samin Khan, Puu Phan, Rick Salay, and Krzysztof Czarnecki. ProcSy: Procedural synthetic dataset generation towards influence factor studies of semantic segmentation networks. In *CVPR Workshops*, 2019. 3, 4

[42] Ryo Kobayashi. Modeling and numerical simulations of dendritic crystal growth. *Physica D: Nonlinear Phenomena*, 63:410–423, 3 1993. 6, 26

[43] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An interactive 3D environment for visual AI. *arXiv preprint arXiv:1712.05474*, 2017. 3

[44] Kujiale. Kujiale.com. https://www.kujiale.com/. 3

[45] 7D Labs. 7d labs. https://www.7dlabs.com/. 3

[46] Hei Law and Jia Deng. Label-free synthetic pretraining of object detectors. *arXiv preprint arXiv:2208.04268*, 2022. 1

[47] Chengshu Li, Fei Xia, Roberto Martín-Martín, Michael Lingelbach, Sanjana Srivastava, Bokui Shen, Kent Vainio, Cem Gokmen, Gokul Dharan, Tanish Jain, et al. IGibson 2.0: Object-centric simulation for robot learning of everyday household tasks. *arXiv preprint arXiv:2108.03272*, 2021. 3

[48] Jiankun Li, Peisen Wang, Pengfei Xiong, Tao Cai, Ziwei Yan, Lei Yang, Jiangyu Liu, Haoqiang Fan, and Shuaicheng Liu. Practical stereo matching via cascaded recurrent network with adaptive correlation. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 1, 3, 8, 13, 18, 33, 34, 36

[49] Wenbin Li, Sajad Saeedi, John McCormac, Ronald Clark, Dimos Tzoumanikas, Qing Ye, Yuzhong Huang, Rui Tang, and Stefan Leutenegger. InteriorNet: Mega-scale multi-sensor photo-realistic indoor scenes dataset. *arXiv preprint arXiv:1809.00716*, 2018. 3, 4

[50] Yang Li, Hikari Takehara, Takafumi Taketomi, Bo Zheng, and Matthias Nießner. 4DComplete: Non-rigid motion estimation beyond the observable surface. In *International Conference on Computer Vision (ICCV)*, pages 12706–12716, 2021. 3, 4

[51] Zhengqi Li and Noah Snavely. Cgintrinsics: Better intrinsic image decomposition through physically-based rendering. In *Proceedings of the European conference on computer vision (ECCV)*, pages 371–387, 2018. 3

[52] Zhengqin Li, Ting-Wei Yu, Shen Sang, Sarah Wang, Meng Song, Yuhan Liu, Yu-Ying Yeh, Rui Zhu, Nitesh Gundavarapu, Jia Shi, et al. OpenRooms: An open framework for photorealistic indoor scene datasets. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 3

[53] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014. 1

[54] Lahav Lipson, Zachary Teed, and Jia Deng. RAFT-Stereo: Multilevel recurrent field transforms for stereo matching. In *International Conference on 3D Vision (3DV)*, 2021. 1, 8, 13

[55] Lahav Lipson, Zachary Teed, Ankit Goyal, and Jia Deng. Coupled iterative refinement for 6d multi-object pose estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6728–6737, 2022. 1

[56] Bingyuan Liu, Jiantao Zhang, Xiaoting Zhang, Wei Zhang, Chuanhui Yu, and Yuan Zhou. Furnishing your room by what you see: An end-to-end furniture set retrieval framework with rich annotated benchmark dataset. *arXiv preprint arXiv:1911.09299*, 2019. 1, 2, 3

[57] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987. 6

[58] Zeyu Ma, Zachary Teed, and Jia Deng. Multiview stereo with cascaded epipolar raft. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXI*, pages 734–750. Springer, 2022. 1

[59] Benjamin Mark, Tudor Berechet, Tobias Mahlmann, and Julian Togelius. Procedural generation of 3d caves for games on the gpu. In *FDG*, 2015. 22

[60] N. Mayer, E. Ilg, P. Häusser, P. Fischer, D. Cremers, A. Dosovitskiy, and T. Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. arXiv:1512.02134. 3, 8, 13, 18, 33, 34

[61] John McCormac, Ankur Handa, Stefan Leutenegger, and Andrew J Davison. SceneNet RGB-D: Can 5m synthetic images beat generic imagenet pre-training on indoor segmentation? In *International Conference on Computer Vision (ICCV)*, pages 2678–2687, 2017. 3

[62] Nick McDonald. Soilmachine. https://github.com/weigert/SoilMachine, 2022. 6, 22

[63] Lukas Mehl, Jenny Schmalfuss, Azin Jahedi, Yaroslava Nalivayko, and Andrés Bruhn. Spring: A high-resolution high-detail dataset and benchmark for scene flow, optical flow and stereo. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4981–4991, 2023. 3

[64] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 4

[65] Sergey I Nikolenko. *Synthetic data for deep learning*, volume 174. Springer, 2021. 2

[66] Tomoyuki Nishita, Takao Sirai, Katsumi Tadamura, and Eihachiro Nakamae. Display of the earth taking into account atmospheric scattering. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, page 175–182, New York, NY, USA, 1993. Association for Computing Machinery. 6, 23

[67] Boris Okunskiy. Introducing differential growth addon for blender. https://boris.okunskiy.name/posts/blender-differential-growth. 6, 25

[68] Jordan Peck. Fastnoise lite. https://github.com/Auburn/FastNoiseLite, 2022. 6, 22

[69] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985. 22

[70] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, jun 1975. 6

[71] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision (ECCV)*, pages 102–118. Springer, 2016. 1, 2, 3

[72] Mike Roberts, Jason Ramapuram, Anurag Ranjan, Atulit Kumar, Miguel Angel Bautista, Nathan Paczan, Russ Webb, and Joshua M. Susskind. Hypersim: A photorealistic synthetic dataset for holistic indoor scene understanding. In *International Conference on Computer Vision (ICCV)*, 2021. 3

[73] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. *NPH*, 7(63-70):6, 2007. 6, 24

[74] Daniel Scharstein, Heiko Hirschmüller, York Kitajima, Greg Krathwohl, Nera Nesic, Xi Wang, and Porter Westling. High-resolution stereo datasets with subpixel-accurate ground truth. In *GCPR*, 2014. 8, 13, 18, 34

[75] Soumyadip Sengupta, Jinwei Gu, Kihwan Kim, Guilin Liu, David W Jacobs, and Jan Kautz. Neural inverse rendering of an indoor scene from a single image. In *International Conference on Computer Vision (ICCV)*, pages 8598–8607, 2019. 3

[76] Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. Semantic scene completion from a single depth image. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. 3

[77] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. UCF101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012. 3

[78] Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, Karen Liu, et al. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Conference on Robot Learning (CoRL)*, pages 477–490, 2022. 3

[79] Stereolabs. Zed 2. https://www.stereolabs.com/zed-2/. 13, 33

[80] Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J Engel, Raul Mur-Artal, Carl Ren, Shobhit Verma, et al. The Replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019. 3

[81] Andrew Szot, Alex Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Chaplot, Oleksandr Maksymets, Aaron Gokaslan, Vladimir Vondrus, Sameer Dharur, Franziska Meier, Wojciech Galuba, Angel Chang, Zsolt Kira, Vladlen Koltun, Jitendra Malik, Manolis Savva, and Dhruv Batra. Habitat 2.0: Training home assistants to rearrange their habitat. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021. 3

[82] Zachary Teed and Jia Deng. RAFT: Recurrent all-pairs field transforms for optical flow. In *European Conference on Computer Vision (ECCV)*, pages 402–419. Springer, 2020. 1

[83] Zachary Teed and Jia Deng. DROID-SLAM: Deep visual SLAM for monocular, stereo, and RGB-D cameras. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021. 1

[84] Zachary Teed and Jia Deng. Raft-3d: Scene flow using rigid-motion embeddings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8375–8384, 2021. 1

[85] Zachary Teed, Lahav Lipson, and Jia Deng. Deep patch visual odometry. *arXiv preprint arXiv:2208.04726*, 2022. 1

[86] Jonathan Tremblay, Thang To, and Stan Birchfield. Falling Things: A synthetic dataset for 3D object detection and pose estimation. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 3, 8, 13, 18, 33, 34

[87] Apostolia Tsirikoglou, Joel Kronander, Magnus Wrenninge, and Jonas Unger. Procedural modeling and physically based rendering for synthetic data generation in automotive applications. *arXiv preprint arXiv:1710.06270*, 2017. 3

[88] Timo Von Marcard, Roberto Henschel, Michael J Black, Bodo Rosenhahn, and Gerard Pons-Moll. Recovering accurate 3D human pose in the wild using imus and a moving camera. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 601–617, 2018. 3

[89] Wenshan Wang, Yaoyu Hu, and Sebastian Scherer. Tartanvo: A generalizable learning-based vo. In *Conference on Robot Learning*, pages 1761–1772. PMLR, 2021. 1

[90] Wenshan Wang, Delong Zhu, Xiangwei Wang, Yaoyu Hu, Yuheng Qiu, Chen Wang, Yafei Hu, Ashish Kapoor, and Sebastian Scherer. TartanAir: A dataset to push the limits of visual SLAM. In *International Conference on Intelligent Robots and Systems (IROS)*, 2020. 3, 8, 13, 18, 33, 34

[91] Erroll Wood, Tadas Baltrušaitis, Charlie Hewitt, Sebastian Dziadzio, Thomas J Cashman, and Jamie Shotton. Fake it till you make it: face analysis in the wild using synthetic data alone. In *International Conference on Computer Vision (ICCV)*, pages 3681–3691, 2021. 3

[92] Magnus Wrenninge and Jonas Unger. Synscapes: A photorealistic synthetic dataset for street scene parsing. *arXiv preprint arXiv:1810.08705*, 2018. 3, 4

[93] Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, et al. SAPIEN: A simulated part-based interactive environment. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 3

[94] Gengshan Yang, Joshua Manela, Michael Happold, and Deva Ramanan. Hierarchical deep stereo matching on high-resolution images. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 8, 13, 18

[95] Yinda Zhang, Shuran Song, Ersin Yumer, Manolis Savva, Joon-Young Lee, Hailin Jin, and Thomas Funkhouser. Physically-based rendering for indoor scene understanding using convolutional neural networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5287–5295, 2017. 3

[96] Jia Zheng, Junfei Zhang, Jing Li, Rui Tang, Shenghua Gao, and Zihan Zhou. Structured3D: A large photo-realistic dataset for structured 3D modeling. In *European Conference on Computer Vision (ECCV)*, 2020. 3

Appendix

A. Figures Extended

First, we provide a large sample of random, non-cherry-picked RGB images from our dataset generator (Figs. A, B, C, D). Both this sample and Fig. 1 in the main paper are randomly selected, however unlike in Fig. 1, here we do not group the images by scene type. We limit the sample to 576 JPEG images of resolution 960x540 only due to space constraints - in practice we can generate infinitely many such images as 1080P PNGs, with a full suite of accompanying ground truth data.

Second, we show an extended random sample of our terrain system (Fig. E F), grouped by scene type as in Fig. 6 of the main paper.

Please visit infinigen.org for videos, code, and extended hi-res. random samples.

B. Experiments

To validate the usefulness of the generated data, we produce 30K image pairs for training rectified stereo matching. We train RAFT-Stereo [54] on these images from scratch and compare against the same architecture trained on other synthetic datasets. Models are trained for 200k steps using the same hyper-parameters from [54].

Real Images of Natural Scenes. Because images from Infinigen consist of entirely of natural scenes and are devoid of any human-made objects, we expect models trained on Infinigen data to perform better on images with natural environments and worse on images without (e.g. indoor scenes). However, quantitative evaluation on real-world natural scenes is currently infeasible because there does not exist a real-world benchmark that evaluates depth estimation for natural scenes. Existing real-world benchmarks consist almost entirely of images of indoor environments dominated by artificial objects. In addition, it is challenging to obtain 3D ground truth for real-world natural scenes, because real-world natural scenes are often highly complex and non-static (e.g. moving tree leaves and animals), making high-resolution laser-based 3D scanning impractical.

Due to the difficulty of obtaining 3D ground truth for real images of natural scenes, we perform qualitative evaluation instead. We collected high-resolution rectified stereo images of real-world natural scenes using the ZED 2 Stereo Camera [79] and visualize the predicted disparity maps from RAFT-Stereo [54] in Fig. G. Our results show that a model trained entirely on synthetic scenes from Infinigen can perform well on real images of natural scenes zero-shot. The model trained on Infinigen data produces noticeably better results than models trained on existing datasets, suggesting that Infinigen is useful in that it provides training data for a domain that is poorly covered by existing datasets.

Middlebury Dataset. We evaluate our trained model on the Middlebury Dataset [74], which is a standard evaluation benchmark for stereo matching. It consists of megapixel image-pairs of cluttered indoor spaces, 10 with public ground-truth and 10 without. This benchmark is challenging due to its abundance of objects, textureless surfaces, and thin structures. In Tab. B, we see that our Infinigen-trained model struggles on images with exclusively artificial objects but performs well on the only image with natural objects (Jadeplant). In Fig H, we qualitatively evaluate our model on Middlebury images without public ground-truth and observe that our model generalizes well to the natural scenes.

Training Dataset	Bad 3.0 (%) ↓	# Image Pairs
InStereo2K [5]	25.282	2K
FallingThings [86]	12.199	62K
Sintel-Stereo [9]	10.253	2K
HR-VS [94]	9.296	780
Li et al. [48]	9.271	177K
SceneFlow [60]	7.837	35K
TartanAir [90]	6.504	296K
Ours (Infinigen 30K)	**5.527**	31K

Table A. Performance on 400 independent Infinigen evaluation scenes. No assets are shared between our Infinigen training and evaluation scenes.

Synthetic Images of Natural Scenes. Although quantitative evaluation is not currently feasible on real-world natural scenes, it can be done using synthetic natural scenes from Infinigen, with the caveat that we rely on the assumption that performance on Infinigen images is a good proxy to real-world performance, as suggested by the qualitative results in Fig. G. In Tab. A, we evaluate our Infinigen-trained model on an independent set of 400 image pairs from Infinigen. No assets are shared between our training and evaluation sets. Tab. A also compares the model trained on Infinigen to models trained on other datasets. We see that the model trained on Infinigen data has a significant lower error than those trained on other datasets. These quantitative results suggest that the distribution of Infinigen images is significantly different from existing datasets and that Infinigen can serve as a useful supplement to existing datasets.

C. Dataset Generation

C.1. Image Rendering

We render images using Cycles, Blender's physically-based path tracing renderer. Cycles individually traces photons of light to accurately simulate diffuse and specular reflection, transparent refraction and volumetric effects. We render at 1920×1080 resolution using $10,000$ random samples per-pixel.

Figure A. 576 randomly generated, non-cherry-picked images produced by our system (Part 1 of 4). Images are compressed due to space constraints - please see infinigen.org

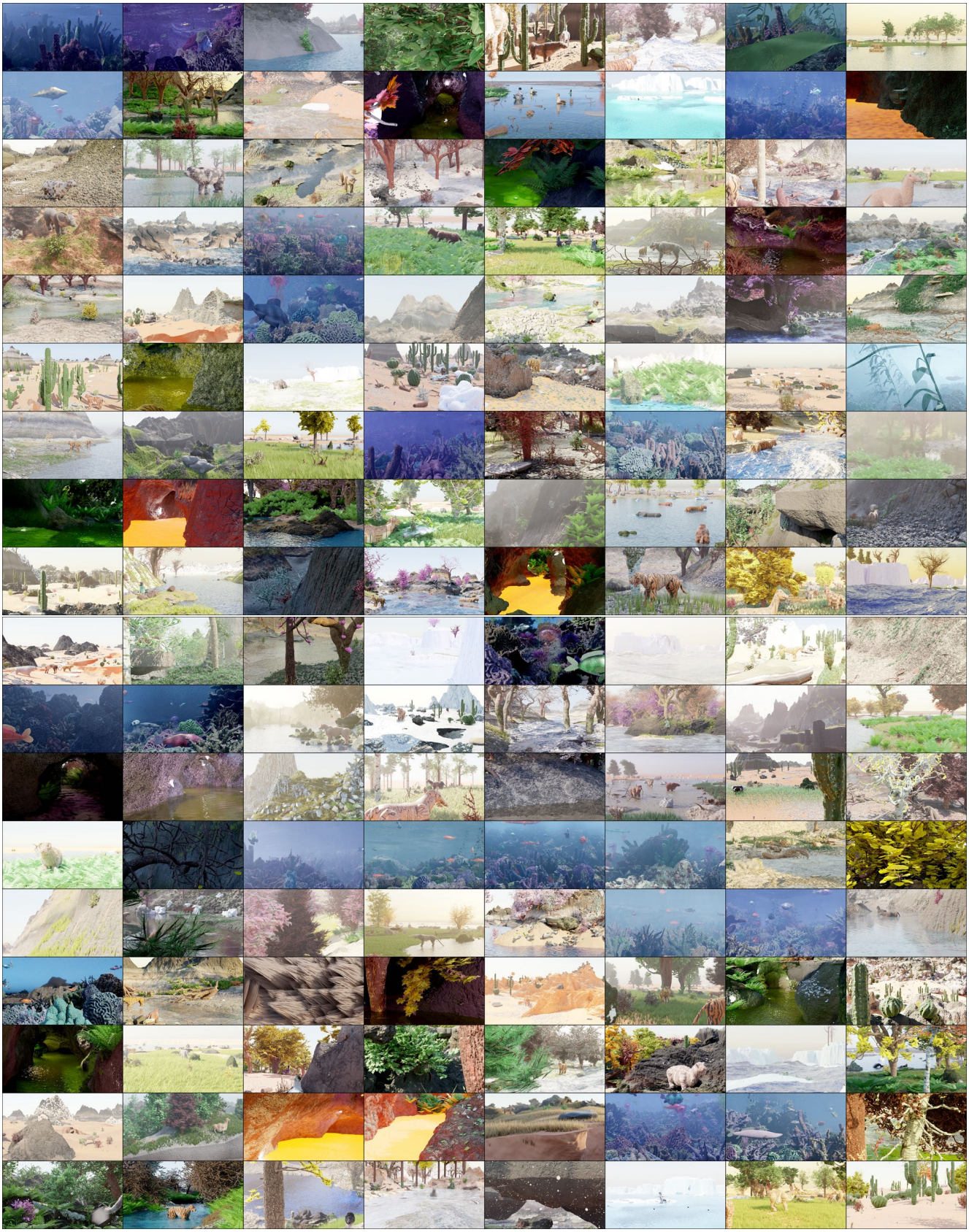


Figure B. 576 randomly generated, non-cherry-picked images produced by our system (Part 2 of 4). Images are compressed due to space constraints - please see infinigen.org

Figure C. 576 randomly generated, non-cherry-picked images produced by our system (Part 3 of 4). Images are compressed due to space constraints - please see infinigen.org

Figure D. 576 randomly generated, non-cherry-picked images produced by our system (Part 4 of 4). Images are compressed due to space constraints - please see infinigen.org

Training Dataset	Adirondack	**Jadeplant**	Motorcycle	Piano	Pipes	Playroom	Playtable	Recycle	Shelves	Vintage	Avg
FallingThings [86]	8.3	43.3	**12.3**	18.2	25.3	29.7	50.0	10.4	43.3	45.6	28.6
Sintel-Stereo [9]	35.7	62.9	31.1	24.1	31.9	41.7	60.1	30.8	55.8	76.1	45.0
HR-VS [94]	43.5	43.2	17.0	29.6	32.1	34.6	68.4	24.7	57.4	34.9	38.5
Li et al. [48]	23.9	80.2	40.7	32.0	40.3	49.1	67.5	36.6	51.7	42.3	46.4
SceneFlow [60]	**7.4**	41.3	14.9	16.2	33.3	**18.8**	**38.6**	**10.2**	**39.1**	29.9	**25.0**
TartanAir [90]	15.5	45.1	18.1	**12.9**	28.4	25.6	51.0	20.9	49.1	**28.2**	29.5
InStereo2K [5]	17.1	59.7	21.3	23.8	35.8	33.9	**36.4**	20.0	**33.4**	44.1	32.5
Ours (Infinigen 30K)	**7.4**	**35.2**	15.2	20.7	**24.7**	29.3	50.0	12.6	55.1	46.9	29.7

Table B. Bad 3.0 (%) ↓ error on the Middlebury [74] validation set. Infinigen helps models generalize to images with natural objects (e.g. Jadeplant). On the other hand, natural objects contain very few planar or texture-less surfaces; models trained exclusively on natural objects can generalize less well on indoor datasets like Middlebury.

C.2. Ground Truth Extraction

We develop custom code for extracting ground-truth directly from the geometry. Prior datasets [9,24,27,29,48] rely on blender's built-in render-passes to obtain dense ground truth. However, these rendering passes are a byproduct of the rendering pipeline and not intended for training ML models. Specifically, they are incorrect for translucent surfaces, volumetric effects, or when motion blur, focus blur or sampling noise are present.

We contribute OpenGL code to extract surface normals, depth, segmentation masks, and occlusion boundaries from the mesh directly without relying on blender.

Depth. We show several examples of our depth maps in Fig. I. In Fig. K, we visualize the alternative approach of naively producing depth using blender's built-in render passes.

Occlusion Boundaries. We compute occlusion boundaries using the mesh geometry. Blender does not natively produce occlusion boundaries, and we are not aware of any other synthetic dataset or generator which provides exact occlusion boundaries.

Surface Normals. We compute surface normals by fitting a plane to the local depth map around each pixel. Sampling the geometry directly instead can lead to aliasing on high-frequency surfaces (e.g. grass).

The size of the plane used to fit the local depth map is configurable, effectively changing the resolution of the surface normals. We can also configure our sampling operation to exclude values which cannot be reached from the center of each plane without crossing an occlusion boundary; planes with fewer than 3 samples are marked as invalid. We show these occlusion-augmented surface normals in Fig. I. These surface normals appear only surfaces with sparse occlusion boundaries, and exclude surfaces like grass, moss, lichen, etc.

Segmentation Masks. We compute instance segmentation masks for all objects in the scene, shown in Fig. I. Object meta-data can be used to group certain objects together arbitrarily (e.g. all grass gets the same label, a single tree gets one label, etc).

Customizable. Since our system is controllable and fully open-source, we anticipate that users will generate countless task-specific ground truth not covered above via simple extensions to our codebase.

C.3. Runtime

We benchmark Infinigen on 2 *Intel(R) Xeon(R) Silver 4114 @ 2.20GHz* CPUs and 1 NVidia-GPU (one of GTX-1080, RTX-[2080, 6000, a6000] or a40) across 1000 independent trials. We show the distribution in Fig. L. The average wall time to produce a pair of 1080p images is 3.5 hours. About one hour of this uses a GPU, for rendering specifically. More CPUs per-image-pair will decrease the wall-time significantly as will faster CPUs. Our system also uses about 24Gb of memory on average.

Pre-generated Infinigen Data. To maximize the accessibility of our system we will provide a large number of pre-generated assets, videos, and images from Infinigen upon acceptance.

D. Interpretable Degrees of Freedom

We attempted to estimate the complexity of our procedural system by counting the number of human-interpretable parameters, as shown in the per-category totals in Table 2 of the main paper. Here, we provide a more granular break down of what named parameters contributed to these results.

Counting Method We seek to provide a conservative estimate of the expressive capacity of our system. We only count distinct human-understandable parameters. We also only include parameters that are *useful*, that is if it can be randomized within some neighborhood and produce noticeably different but still photorealistic assets. Each row of Tabs. C–J gives the names of all *Intepretable DOF* that are relevant to some set of *Generators*.

We exclude trivial transformations such as scaling, rotating and translating an asset. We include absolute sizes such as 'Length' or 'Radius' as parameters only when their *ratio* to some other part of the scene is significant, such as the leg to body ratio of a creature, or the ratio of a sand dune's height to width.

Many of our material generators involve randomly generating colors using random HSV coordinates. This has three degrees of freedom, but out of caution we treat each color as one parameter. Usually, one or more HSV coordinates are restricted to a relatively narrow range, so this one parameter represents the value of the remaining axis. Equivalently, it can be imagined as a discrete parameter specifying some named color-palette to draw the color from. Some generators also contain compact functions or parametric mapping curves, each usually with 3-5 control points. We treat each curve as one parameter, as the effect of adjusting any one handle is subtle.

Results In total, we counted 182 procedural asset generators with a sum of 1070 distinct interpretable parameters / Degrees-of-Freedom. We provide the full list of these named parameters as Tables C–J, placed at the end of this document as they fill several pages.

E. Transpiler

Code Generation In the simplest case, the transpiler is a recursive operation which performs a post-order traversal of Blender's internal representation of a node-graph, and produces a python statement defining each as a function call of it's children. We automatically handle and abstract away many edge cases present in the underlying node tree, such as enabled/disabled inputs, multi-input sockets and more. This procedure also supports all forms of blender nodes, including *shader nodes, geometry nodes, light nodes* and *compositor nodes*.

Doubly-recursive parsing Blender's node-graphs support many systems by which node-graphs can contain and depend on one another. Most often this is in the form of a node-group, such as the dark green boxes titled *SunflowerSeedCenter* and *PhylloPoints* in Fig. M. Node-groups are user-defined nodes, containing an independent node-tree as an implementation. These are equivalent to functions in a typical programming languages, so whenever one is encountered, the transpiler will invoke itself on the node-graph implementation and package the result as a python function, before calling that function in the parent node-graph. In a similar fashion, we often use *SetMaterial* nodes that reference a shader node-graph to be transpiled as a function.

Probability Distribution Annotation Node-graphs contain many internal parameters, which are artistically tuned by the user to produce a desired result. We provide a minimal interface for users to also specify the distribution of these parameters by writing small strings in the node name. See, for example, the red nodes in Fig. M. When annotations of this format are detected, the transpiler automatically inserts calls to appropriate random number generators into the resulting code.

F. Scene Composition Details

Our scenes are not individually staged images - for each image, we produce a map of an expansive and view-consistent world. One can select any camera pose or sequence of poses, which allows for video and other multi-view data generation.

To allow this, we start by sampling a full-scene ground surface. This is low-resolution, but is sufficient to approximate the surface of the terrain for the purpose of placing objects. We determine surface points using Poisson-Disc Sampling. This avoids the majority of asset-asset intersections. We modulate point density using procedural masks based on surface normals, Perlin noise and terrain attributes. Asset rotations are determined uniformly at random. Our final coarse global map is represented as a lightweight blender file with intuitive editable placeholders to represent where assets will be spawned in later steps of the pipeline.

We provide a library of 11 optional configuration files to modify scene composition, namely *Arctic, Coast, Canyon, Cave, Cliff, Desert, Forest, Mountain, Plains, River* and *Underwater*. Each encodes simple natural priors such as "Cacti often grow in deserts" or "Trees are less dense on mountains", expressed as modifications to these mask and density parameters. More complex relations, like predators and prey avoiding each other, or plants not growing in shaded areas, are not currently captured.

F.1. Camera Selection

We select camera viewpoints with simple heuristics designed to match the perspective of a creature or person, which are as follow:

Height above Ground In order to match the perspective of a creature or person, we sample the camera height above ground from a Gaussian distribution. (with the exception that in terrain-only scenes sometimes this height is higher to highlight some landscape features)

Minimum Distance To avoid being blocked by a close-up object and over-subdividing the geometry (which is expensive), we select camera views with a minimum distance threshold to all objects.

Coverage In order to avoid overly barren images and to highlight interesting features, we may select views such that a certain terrain component, e.g. a river, is visible. Specifically, we may require that the camera view has pixels from a specific terrain component or object type within a certain range.

Standard Deviation of Depth We compute the variance of pixel-wise depth values, and choose the viewpoint out of ten random samples with the largest variance to favor more interesting content.

F.2. Dynamic Resolution

In Fig. J, we show a visualization of triangle sizes in cm^2 and in pixels as viewed from the camera. Face size in meters increases proportionally to depth, whereas face size in pixels remains approximately constant.

F.2.1 Spherical Marching Cubes

To generate a mesh for a specific camera view, we must extract a mesh representation of the terrain which contains dense pixel-size faces. Classical *Marching Cubes* struggles to achieve this, as it evaluates the SDF at fixed intervals, which results in too-sparse geometry near the camera and too-dense geometry in the far distance. Spherical Marching Cubes is our novel adaptation of this classic algorithm to operate in spherical coordinates, which automatically creates dense geometry near the camera where it is most needed, thereby preventing waste and drastically improving performance.

Spherical Marching Cube algorithm works as follows:

Low Resolution Step We divide the visible 3D space (within the frustrum camera and a certain distance range (d_{min}, d_{max})) into $M \times N \times R$ blocks in spherical coordinates, uniform in θ and ϕ and logarithmically in r. We convert these to cartesian coordinates and evaluate the SDF as usual. This yields a tensor of SDF values where grid cells near d_{max} represent larger regions of space than those close to the camera, thereby saving space.

Visible Block Search Step We use this SDF tensor to find the closest block for each pixel with an SDF zero crossing, resulting in an approximate terrain-only depth-map. These blocks are low-resolution and may contain holes, so we check them again with dense SDF queries to determine any farther away visible blocks.

High Resolution Step Finally, we evaluate dense pixel-size SDF queries for all visible blocks, and produce a finalized mesh with Marching Cubes. Theoretically the final size of this mesh can still be proportional to cube of the resolution, but in practice we find it is proportional to square. This step and the previous step can be performed iteratively to prevent all potential holes.

Out-of-view Part The out-of-view part of the terrain is needed for lighting effects but done with low resolution.

F.2.2 Parametric Surface Resolution Scaling

NURBS and other parametric surfaces support evaluation at any mesh resolution. This is achieved by specifying some du, dv as step sizes in parameter space. We provide heuristics to compute appropriate values for these step sizes such that the resulting mesh achieves a given max pixel size.

F.2.3 Subdivision and Remeshing

All other assets rely on established Subdivision and Voxel-Remeshing algorithms to create dense pixel-size geometry. We provide heuristics to compute appropriate subdivision levels. Voxel Remeshing is time intensive but is especially useful for creatures and trees whose geometry can otherwise self-intersect or contain stretched faces.

G. Asset Implementation Details

G.1. Materials

Our materials are composed of a shader and a local geometry template. The shader procedurally generates realistic color, roughness, specularity, metallicity, subsurface scattering, and translucence parameters. The local geometry template generates corresponding geometric detail. Most often, the local geometry template simply computes a scalar field over the underlying mesh vertices, and displaces them along their normals to form a rough texture.

G.1.1 Terrain Materials

The majority of our terrain materials (including *Mountain, Granite, Snow, Stone, Ice)* operate by combining many octaves of Perlin Noise to form geometric texture, before applying a mostly flat color. Some, such as some random variations of *Mountain*, create layered color masks as a function of the world Z coordinate. Others such as *Sand* follow a similar scheme, but with a procedural Wave texture instead of Perlin Noise.

Our *Mud* and *Sandstone* materials are particular expressive. *Mud* procedurally generates puddles and slick ground by altering color, displacement and roughness jointly. Sandstone generates realistic layered sedimentary rock using noise functions and modular arithmetic based on the world Z coordinate.

Lava uses displacement from Perlin noise, F1-smooth Voronoi texture, and wave texture with varying scales. The shader uses Perlin noise, Voronoi texture to model hot and rocky parts, mixed with blackbody emission and a principled BSDF.

Fire and *Smoke* are comprised of multiple volumetric shaders. The first shader uses a principled volume shader with blackbody radiation whose intensity is sampled based on the amount of flame and smoke density. The smoke

density is randomly sampled. The second shader imitates a high detail image captured with fast shutter speed and low exposure. The detail is brought out based on contrasting different regions of temperature and adding Perlin noise. The colors are shades of orange.

All our water materials feature glass-like surfaces with physically accurate Index-of-Refraction, combined with a volumetric scattering shader to simulate realistic underwater light bounces. The *Water* shader uses these effects with geometry untouched (for use with simulators), wherase *Water Surface* assumes the base geometry is a plane and adds geometric ripples.

G.1.2 Plant Materials

Our plant materials feature geometric and color variation using procedural Stucci, Marble and Shot Noise textures. We provide color pallettes for realistic plant and coral colors, and produce variations on these using Musgrave noise. All leafy plants feature realistic transmission and roughness properties, to simulate light filtering through translucent waxy leaves. Our *Bark* and **Bark Birch** simulate bark expansion and fracturing using voronoi and perlin noise to generate displacements.

G.1.3 Creature Material

Fish We create two fish materials, a goldfish material and a regular fish material. Each material consists of two parts, a fish body material and a fish fin material. The fish body material creates the displacement of fish scales. To build the pattern of fish scales, we create a grid and fill every two adjacent grids in one column with a half circle. Then we move up the half circles in the even columns by one grid. The colors of regular fish are guided by one wave texture and two noise textures. The colors of goldfish are guided by two noise textures and sampled between red and yellow. The colors of fish bellies are usually white. A fish fin is created by adding periodical bumps to round planes. The weights of bump displacement are decreasing away from the fish body. Dorsal fins are sometimes serrated. The goldfish fins are translucent by mixing a principled BSDF shader, a transparent shader and a translucent shader.

Bird Since our generated birds have particle fur, the bird material need only create a colormap. We create masks that highlight different body parts, including heads, necks, upper bodies, lower bodies and wings. Each part is assigned two similar colors guided by a noise texture. Our Bird material generator has two modes of colors, one with light colored heads and dark color bodies (emulating bald eagles and falcons), one with dark color heads and light color bodies (emulating ducks and common birds).

Carnivore and Herbivore Carnivores and Herbivores are also equipped with particle fur and therefore color-only materials. We provide a *Tiger* material, which makes short stripes by cutting a small-scale wave texture with another larger-scale wave texture. Our *Giraffe* material builds spots by subtracting a F1-smooth voronoi texture from a F1 voronoi texture with the same scale. Three other spot materials build scattered and sometimes overlapped spots using noise textures and voronoi textures. Our three reptile materials build colormaps inspired by different kinds of reptiles. We pick their colors to mimic the reference reptile images, and then randomize in a small neighborhood.

Beetle We provide a *Chitin* material emulating the material of real beetles and other insects. It uses a computed "Pointiness" attribute to highlight the boundaries with sharp curvature. We color the insect head and sharp boundaries black, and other areas dark red or brown.

Bone, Beak & Horns *Bone* is most often used for animal claws and teeth. It starts wit a white to light gray color, before creating small pits in two different scales using noise and voronoi textures. Our *Horn* material samples light brown to dark brown colors, with a similar mechanism for pits and scratches. Our *Beak* material replicates realistic bird beaks by sampling a random yellow/orange/black color gradient along. Noise textures are used to add some small pits on it. Principled BSDF shaders are added to make the beaks more shiny.

Slime We provide a material titled *Slimy*, which replicates folded shiny flesh similar to a "Blobfish". It builds thin and distorted stripe displacement using a noise texture followed by a wave texture. We create the shiny appearance by assigning high specularity and subsurface scattering parameters.

G.1.4 Other material

Metal We build a silver material and an aluminium material. These use Blender's Principled BSDF shader with *Metallic* set to 1. They also have sparse sunken displacement from a noise texture.

G.2. Terrain

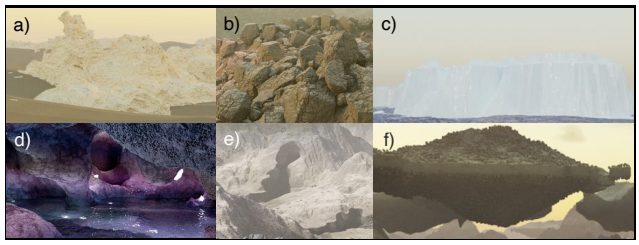


Figure N. Terrain elements, including a) wind-eroded rocks, b) Voronoi rocks, c) Tiled landscapes, d/e) Caves, and f) Floating Islands.

G.2.1 Terrain Elements

The main part of terrain (*Ground Surface*) is composed of a set of terrain elements represented by Signed Distance Functions (SDF). Using SDF has the following advantages:

- SDF is written in C/C++ language and can be compiled either in CPU (with OpenMP speedup) or CUDA to achieve parallelization.
- SDF can be evaluated at arbitrary precision and range, producing a mesh with arbitrary details and extent.
- SDF is flexible for composition. Boolean operations are just minimum and maximum of SDF. To put cellular rocks onto a mountain, we just query whether each corresponding Voronoi center has positive SDF of the mountain.

Terrain elements include:

Wind Eroded Rocks They are made out of Perlin noise [69] from FastNoise Lite [68] with domain warping adapted from an article [22]. See Fig. N(a) for an example.

Voronoi Rocks These are made from Cellular Noise (Voronoi Noise) from FastNoise Lite. They take another terrain element as input and generate cells that are on the surface of the given terrain element and add noisy gaps to the cells. See Fig. N(b) for an example. We utilize this element to replicate fragmented rubble, small rocks and even tiny sand grains.

Tiled Landscape Complementary to above, we generate heterogeneous terrain elements as finite domain tiles. First, we generate a primitive tile using the A.N.T. Landscape Add-on in Blender [11], or a function from FastNoise Lite. This tile has a finite size, so we can simulate various natural process on it, e.g., erosion by SoilMachine [62], snowfall by diffusion algorithm in Landlab [33] [6] [30]. Finally, various types of tiles can be used alone or in combination to generate infinite scenes including mountains, rivers, coastal beaches, volcanos, cliffs, and icebergs. Fig. N(c) shows an example of a iceberg tile. Tiles are combined by repeating them with random rotations and smoothing any boundaries. The resulting terrain element is still represented as an SDF.

Caves Our terrain includes extensive cave systems. These are cut out from other SDF elements before the mesh is created. The cave passages are generated procedurally using an Lindenmayer-System with probabilistic rules, where each rule controls the direction and movement of a virtual turtle [59]. These rules include turns, elevation changes, forks, and others. These passages have varying cross section shapes and are unioned with each other, leading to complicated cave systems with features from small gaps to large caverns. One can intuitively tune the cavern size, tunnel frequency and fork frequency by adjusting the likelihoods of various random rules. See Fig. N(d) for an interior view of a cave and Fig. N(e) for how it cut from mountains.

Floating islands Besides natural scenes, we also have fantastical terrain elements, e.g., floating islands (Fig. N(f)) by gluing mountains and upside down mountains together.

G.2.2 Boulders

We start off with a mesh built from convex hull of around 32 vertices. We randomly select some faces that are large enough, so that they can be extruded and scaled. We repeat this process for two levels of extrusions: large and small. Finally we bevel the mesh, and add a displacement based on high- and low-frequency Voronoi textures. After generating the base mesh, boulders are given a rock surface and optionally a rock cover surface. See Sec. G.1 for details. Boulders are placed on the terrain mesh as placeholders.

G.2.3 Fluid

Most water and lava in our scenes form relatively static pools and lakes — these are handled by generating a flat plane and applying a *Surface Water* or *Lava* material from Sec. G.1.

Ocean We simulate dyhnamic oceans Blender's built-in modifier to generate displacements on top of a water plane. This simulation is finite domain, so we tile it as described in *Tiled Landscape* above.

Dynamic Fluid Simulation We generate dynamic water and lava simulations using Blender's built-in Fluid-Implicit-Particle (FLIP) plugin [8]. They can either simulated on a small region of the terrain or work together with Tiled Landscape, e.g., Volcanos to be reused as instances. The

simulations are parameterized by sampled values of vorticity, viscosity, surface tension, flow amount, and other liquid parameters. We simulate fire and smoke simulations using Blender's particle simulator. Our system allows for 1) Simulating fire and smoke on small random regions on the terrain or 2) Choosing arbitrary meshed assets on the scene to be set on fire or emit smoke. The simulations interact with turbulent and laminar wind flows added on the scene. While these simulators are provided with Blender, we contribute significant engineering effort to automate their use. Typically, users manually set up individual simulations and execute them through the UI - we do so programmatically and at large scale.

G.2.4 Weather

We provide procedural SDF functions for 4 realistic categories of clouds, each implemented as node-graphs. We implement rain and snow using Blender's particle system and wind simulation. We also apply atmospheric volume scattering to the entire scene to create haze, fog etc.

G.2.5 Lighting

The majority of scenes are lit only by the sun and sky. We simulate these using the built in *Nishita* [66] sky model, with randomized parameters for the Sun's position and brightness, as well as atmospheric parameters. In cave scenes, we place glowing gemstones as natural proxies for point lights. In underwater scenes, ray-traced refractive caustics are too costly at render-time, so we substitute textured spot lights. Finally, we provide an option to attach a virtual flash light or area light to the camera, to simulate a human or robot with an attached light.

G.3. Plants

G.3.1 Leaves, Flowers & Pinecones

Pinecones Pinecones are the woody seed-bearing organs for conifers, which features scales and bracts arranged around a central axis, as shown in Fig. U b). Pinecones are made from individual buds. Each buds are sculpted from a mesh circle, with its left most point chosen as the origin. The vertices are displaced along the axis to the origin as well as along the Z axis, with scale designated by the direction from the origin to that point. We then create a mesh line along the Z-axis to form the stem of the pinecone. Pinecone buds are distributed on the axis from bottom to up with a decreasing scale and changing rotation. The rotation is composed of two parts: one along the Z axis that spread the buds around the pinecone, another along the X axis the gradually point the buds upwards. Pine shaders are made from Principled BSDF of a single color. Pinecones are scatters on the terrain mesh surfaces.

Leaves Our leaf generation system covers common leaf types including oval-shaped (which covers most of the broad-leaved trees), maple, ginkgo, and pine twigs.

For oval-shaped leaves, we start by subdividing a 2D plane mesh finely into grids, and evaluate each grid location with various noise functions. The leaf boundary defined by a set of control points of a Blender curve node, which specifies the width of the leaf at each location along the main stem. We then delete the unused geometry to get a rough shape of the leaf. To create veins, we use a 1D *Voronoi Texture* node on a rotated coordinate system, to model the angle between the veins and the main stem. We extrude the veins and pass the height values into the Shader Node Tree to assign them different colors. We then add jigsaw-like patterns on the boundaries of the leaf, and create the cell structures using a 2D *Voronoi Texture* node. We finally add wrapping effect to the leaf with another curve node.

The maple leaves and ginkgo leaves are created in a very similar way as the oval-shaped leaves, except that we use polar coordinates to model rotational symmetric patterns, and the shape of the leaves is defined by a curve node in the polar coordinates.

Pine twigs are created by placing pine needles on a main stem, whose curvature and length are randomized.

We use a mixture of translucent, diffuse, glossy BSDF to represent the leaf materials. The base colors are randomly sampled in the HSV space, and the distribution is tuned for each season (*e.g.*, more yellow and red colors for autumn).

FlowerPlant We create the stem of a flower plant with a curve line together with an cylindrical geometry. The radius of the stem gradually shrinks from the bottom to the top. Leaves are randomly attached to resampled points on the curve line with random orientations and scales within a reasonable range. Each leaf is sampled from a pool of leaf-like meshes. Additionally, we add extra branches to the main stem to mimic the forked shape of flower plants. Flowers are attached at the top of the stem and the top of the branches. Additionally, the stem is randomly rotated w.r.t the top point along all axes to generate curly looks of natural flower plants.

G.3.2 Trees & Bushes

We create a tree with the following steps: 1) *Skeleton Creation* 2) *Skinning* 3) *Leaves Placement*.

Skeleton Creation. This step creates a directed tree-graph to represent the skeleton of a tree. Starting from a graph containing a single root node, we apply *Recursive Paths* to grow the tree. Specifically, in each growing step, a new node is added as the child of the current leaf node. We computed the growing direction of the new node as the weighted sum of the previous direction (representing the momentum) and

a random unit vector (representing the randomness). We further add an optional *pulling direction* as a bias to the growing direction, to model effects such as gravity. We also specify the nodes where the tree should be branching, where we add several child nodes and apply Recursive Paths for all of them. Finally, given the skeleton created by Recursive Paths, we use the *space colonization* algorithm [73] to create dense and natural-looking branches. We scatter attraction points uniformly in a cube around the generated skeleton, and run the space colonization for a fixed amount of steps.

Skinning. We convert the skeleton into a Blender curve object, and put cylinders around the edges, whose radius grows exponentially from the leaf node to the root node. We then apply a procedural tree bark material to the surface of the cylinders. Instead of using a UV, we directly evaluate the values of the bark material in the 3D coordinate space to avoid seams. Since the bark patterns are usually anisotropic (*e.g.*, strip-like patterns along the principal direction of the tree trunks), we use the local coordinate of the cylinders, up to some translation to avoid seams in the boundaries.

Leaves Placement. We place leaves on twigs, and then twigs on trees. Twigs are created using the same skeleton creation and skinning methods, with smaller radius and more branches. Leaves are placed on the leaf nodes of the twig skeleton, with random rotation and possibly multiple instances on the same node. We use the same strategy to place the twigs on the trees, again with multiple instances to make the leaves very dense. For each tree we create 5 twig templates and reuse them all over the tree by doing *instancing* in Blender, to strike a balance between diversity and memory cost.

Compared to existing tree generation systems such as the *Sapling-Tree-Gen*[*] addon in Blender and *Speed-Tree*[†] in UE4, our tree generation system creates leaves and barks using real geometry with much denser polygons, and thus provides high-quality ground-truth labels for segmentation and depth estimation tasks. We find this generation procedural very general and flexible, whose parameter space can cover a large number of tree species in the real world.

Bushes We also use this system to create other plants such as bushes, which have smaller heights and more branching compared to trees. Our system models the landscaping bushes that are pruned to different shapes by specifying the distributions of the attraction points in the space colonization step. Our bushes can be of either cone, cube or ball shaped, as shown in Fig. O.

G.3.3 Cactus

Globular Cactus is modeled after cactus from genus *Ferocactus*, as shown in Fig. Pa). It features a barrel-like base shape and tentacles growing from the pointy vertices of the cactus body. We implement globular cactus by first creating a star-like 2D mesh as its cross section. We then use geometry nodes to rotate, translate and scale it along the Z-axis at the same time, converting it to a 3D mesh. The rotation of the cross section mesh contributes to the desired tilt of the cactus body, and the scale determines the general shape of cactus. Finally the cactus is deformed and scale along the X and Y axis. For Globular Cactus, spikes are distributed on the the pointy vertices generated from the star and on the top most part of the cactus.

Columnar Cactus is modeled after cactus from genus *Cereus*, as shown in Fig. Pb). It features an elongated body with a torch-like shape. We first generate the skeleton using our tree skeleton generation method. This time we choose a configuration with only two levels, both with a smaller momentum in path generation and a large drag towards the positive Z-axis that finally makes the cactus pointing up. From the cactus skeleton, we convert it into a 3D mesh with geometry nodes that moves a star mesh along all the splines in the tree-like skeleton, with the top end of each path having a smaller radius. For Columnar Cactus, spikes are distributed on the the pointy vertices generated from the star and on the top most part of the cactus.

Pricky pear Cactus is modeled after cactus from genus *Opuntia*, as shown in Fig. Pc). It features pear-like cactus part extruded from another one. We create individual cactus parts similar to the one in Globular cactus. We it down in Y-direction and rotated it along the Z-axis so that it becomes almost planar and pear-shaped. These individual cactus parts are stacked on top of each other with an angle recursively to make the whole cactus branching like a tree. For Pricky Pear Cactus, we distribute spikes on both the front and the back faces of the cactus.

Cactus spikes After the main body of a cactus is created, we apply medium-frequency displacement on its surface and add the spike according to specifications. The low-poly spikes are made from several straight skeletons generated by the tree generation system, and are distributed on the selected areas of the cactus with some minimal distance between two instances.

G.3.4 Fern

We create 2-pinnate fern (fern) in Infinigen, as it is common in nature. Each fern is composed of a random number of pinnae with random orientations above the ground. Several instances of fern pinnae are illustrated in Fig.Q.

[*] https://docs.blender.org/manual/en/latest/addons/add_curve/sapling.html
[†] https://store.speedtree.com

Pinnae Composition Each pinnae consists of a main stem and a random number of pinna attached on each side of the stem. The length of the pinnae (main stem) is controlled by a parameter *age*. The main stem is first created with a mesh line with its points set along the z-axis. Then, the mesh line is randomly rotated w.r.t x, y axis around the top point to generate the curly look of a fern pinnae. The pinnae's z-axis rotation is slightly different with x, y axis rotation, as we use its curvature to represent the *age* of the fern. In nature, young fern pinnae has more curly stem and grown-up fern pinnae is usually more stretched and flat. Therefore, we choose the scale of the pinnae's z-axis rotation inverse proportion to the *age* of the pinnae. The external geometry of the main stem is a cylindar with its radius gradually shrinks to 0 at the top. Noticing that the bottom point is always set to world origin despite of the random rotations of the mesh line.

Before merging multiple pinnae into a fern, each pinnae is further curved along the z-axis towards the ground according to the desired orientation of the pinnae. We refer this as the gravitational rotation induced on the geomtery of pinnae. The scale of the gravitational rotation at each mesh line point is also proportion to its distance to the world origin, i.e., the longer the larger.

For 2-pinnate fern, the geometry of each pinna is similar to pinnae, i.e., a stem and leaves attached on each side. Similar to the main stem, we also curve the pinna stem randomly w.r.t x, y axis and inverse proportion to the *age* w.r.t the z-axis. In our fern, the leaves are created with simple leaves-like geometry.

The whole pinnae is generated by adding leaf instances on pinna stem and then adding pinna instances on the main stem. The scale of each leaf instance is scaled to form a desired contour shape of the pinna, which is defined to grow linearly from tip to bottom with additional random noise. For pinna instances on the pinnae, we generate multiple distinct pinna versions and then randomly select one for each mesh line point. In this way, we can enough irregularity and asymmetricity on the pinnae. Furthermore, the pinna instances are also scaled according to the desired pinnae contour. In our asset, two contour modes are use. One grows linearly from top to bottom with additional random noise and the other grows linearly from top to $\frac{1}{6}$ from the bottom and then decrease linearly till the bottom of the pinnae.

Moreover, after all components are joined together, additional texture noise is added on the mesh to create more irregularities.

Fern Composition Each fern is a mixture of pinnae with random orientations. In nature, these fern pinnae typically bend down towards the ground. We also add young fern pinnae standing rigidly in the center.

G.3.5 Mushroom

Mushrooms are modeled after real mushrooms from genus *Agaricus* and *Phallus*, as shown in Fig. U a). A mushroom is composed of its cap, its stem that supports the cap and optionally the skirt that grows from underneath the cap. The cap is made from moving a star-like mesh along the Z-axis with radius specified by multiple Bezier curves. The star-like mesh forms the pleats on the cap surface and gills underneath the caps. The stem is made from a Bezier curve skeleton and converted to a 3D mesh via geometry nodes, with its top end sticking to the cap at an angle. For the mushroom skirt, we create an invisible mesh P underneath the cap that have a similar cross section as the cap. Following the same technique in the Brain Coral (See Appendix G.3.6), we shrinkwrap a reaction-diffusion pattern from a icosphere S onto P, which also mapped the A field onto P. This time, we remove the vertices whose A field is below a certain threshold from the mesh P, so P would have honeycomb-like holes. The skirt is placed underneath the cap. Mushroom have similar material as corals, but have more white spots and lower roughness on its surface. Mushrooms are scattered on the terrain surface.

G.3.6 Corals

Corals are marine invertebrates that live mostly on the seafloor, and are prevalent in underwater scenes. In Infinigen, we provide a library of 8 templates for generating different classes of corals. Examples of individual coral classes are provided in Fig. R. We elaborate these templates for the main coral bodies as follows:

Leather Coral is modeled after corals from genus *Sinularia*, as shown in Fig. Ra). It features curvy surfaces that folds on it self. We implement Leather Coral using a iterative mesh generation technique named Differential Growth from [67]. The mesh starts off as a mesh circle. At each iteration, a force is applied to all of its points. The force at each point is composed of an attraction force from its graph neighbors, a repulsive force from vertices that are close in 3D position, a global growth direction, and a noise vector. All of these forces can be specified by a set of parameters. Such force is applied on all vertices, and the vertices would be displaced by a distance proportional to the force. More concretely, for all vertices v

$$\Delta \mathbf{x}_v = \mathbf{x}'_v - \mathbf{x}_v \propto f_{attr} + f_{rep} + f_{grow} + f_{noise}$$

If the displacement has moved two vertices so that the edge connecting these two vertices has its length above a certain threshold, such edge would subdivided so that their lengths would fall below the threshold, creating new vertices on the edges. The aforementioned process defines a growing

mesh when it is repeated for multiple iterations. In specific, for Leather Corals, we choose the parameters so that the noise force function and growth force function are large, and the growth iterations stop when there's 1k faces. To convert this mesh to the final coral mesh, we apply smooth and subsurface operation on the mesh, followed by a solidify operation that gives the planar mesh some width, converting it into a 3D mesh.

Table Coral is modeled after corals from genus *Acropora*, as shown in Fig. Rb). It features a flat table with curvy surfaces near the boundary. For Table Corals, we use the same Differential Growth method as the Leather Coral. However, we choose a different set of parameters for force application. More concretely, we use a larger repulsion force, apply less displacement on boundary vertices, and stop the process after there is 400 faces in the mesh.

Cauliflower Coral is modeled after corals from genus *Pocillopora*, as shown in Fig. Rc). It features wart-like growth on its surface. We implement Cauliflower Coral after [42]'s simulation of dendritic crystal growth. In this growth simulation setup, we have two density fields A and B over 3D space. The simulation follows the following PDE:

$$
\begin{aligned}
m &= \alpha \arctan\left(\gamma(T-b)/\pi\right) \\
\frac{\partial A}{\partial t} &= \varepsilon^2 \nabla A + A(1-A)(A-1/2+m)/\tau \\
\frac{\partial B}{\partial t} &= \nabla B + k\frac{\partial A}{\partial t}
\end{aligned}
$$

where $\alpha, \gamma, T, \varepsilon, \tau, k$ are pre-specified parameters. We run this PDE simulation on a 3D grid space with forward Euler method for 800 iterations. The resulting density map A, is used to generate the 3D mesh for the coral via the marching cube mesh conversion method.

Brain Coral is modeled after corals from genus *Diploria*, as shown in Fig. Rd). It features groovy surface and intricate patterns on the surface of the coral. We first create the surface texture with reaction-diffusion system simulation. In particular, we start off from a mesh icosphere, and run Gray-Scott reaction-diffusion model on its vertex graph, where edges in the mesh are the edges of the graph. This simulation has two fields A, B on individual vertices, following the equation of:

$$
\begin{aligned}
\frac{\partial A}{\partial t} &= r_A \nabla A - A^2 B + f(1-A) \\
\frac{\partial B}{\partial t} &= r_B \nabla B - A^2 B - (f+k)B
\end{aligned}
$$

where r_A, r_B, f, k are pre-specified parameters. After 1k iterations, the field A is stored to the mesh sphere S. Then we build another polygon mesh P, which follows simple deformation by geometry nodes. We apply shrinkwrap object modifier from S to P, which also maps the field A onto mesh P. The shrinkwrap object modifiers 'wraps' the surface of A onto P by finding the projected points of A along A's normal direction. We displace P's surface using the projected field A, which forms the grooves on the mesh surface. For Brain Corals, we choose the parameters so that $f = \sqrt{k}/2 - k$ for some specific k, i.e. the kill rate and feed rate of system are on the saddle-node bifurcation boundary, so that the surface is groovy.

Honeycomb Coral is modeled after corals from genus *Favia* or *Mussismilia*, as shown in Fig. Re). It features honeycomb-shaped holes on the surface of the coral. We use the same Gray-Scott reaction-diffusion model [23] as the Brain Corals. We choose the parameters so that $f = \sqrt{k}/2 - k - 0.001$ for some specific k, i.e. the kill rate and the feed rate of the system are between the saddle-node bifurcation boundary and the Hopf bifurcation boundary, which yields the honeycomb-shaped holes.

Bush Coral is modeled after corals from genus *Acropora*, which includes the Staghorn coral, as shown in Fig. Rf). We implement bush coral using our tree skeleton generator as discussed. In particular, the skeleton of tree coral has three levels of configuration, with each level of configuration specifying how a branch would grow in terms of directions and length, as well as where new branches would emerge. After the tree skeleton is generated, we convert the 1D skeleton to 3D mesh by specifying radius at each vertex of the skeleton, with the vertices closer to endpoints having a smaller radius.

Twig Coral is modeled after corals from genus *Oculina*, as shown in Fig. Rg). We use the same tree skeleton generator as in Bush Corals. We choose a separate set of parameters so that Twig Corals are more low-lying and less directional than Bush Corals.

Tube Coral is modeled after not corals, but sponges from genus *Aplysina*, which none-the-less lives in the same habitat as corals, as shown in Fig. Rh). We first generate the base mesh as the dual mesh of a deformed icosphere, whose faces

have between 5-6 vertices. The optionally extrude some its upward facing faces along the direction that is approximately along the positive Z-axis. This types of extrusion happens multiple times with different multiple extrusion length. The extruded mesh will have its topmost face removed and will be applied with the solidify object modifier so that the mesh would become a hollow tube.

Coral tentacles After the main body of a coral is created, we add a high frequency noise onto the coral surface. We then add tentacles to the coral mesh. Each tentacle has a low-poly mesh generated from the tree generator system that sprawls outside the coral body. Tentacles are distributed on certain parts of the coral body, like on top-facing surfaces or outermost surfaces.

G.3.7 Other sea plants

Besides corals, we also provide assets of other sea plants, like kelps and seaweeds. Both kelps and seaweeds observe the global oceanic current, which is unique for the entire scene. We show examples of kelps and seaweeds in Fig. S.

Kelps Kelps are large brown algae that lives in shallow waters, as shown in Fig. Sa). To build kelps, we first build meshes for individual kelp leaves. To do so, we first create a planar mesh which is bounded by two sinusoid functions from two sides. It is then deformed along the Z axis an subsurfaced and make a wavy shape. For the kelp stem, we first plot a Bezier curve with its control points following a Brownian process with drift towards the sum of the positive Z direction and the oceanic current drift. The curve is turned into a mesh with a small radius, and this becomes the mesh of the kelp stem. Along the stem we scatter points with fixed intervals, where we place kelp leaves along its normal direction. Finally, we rotate the kelp leaves towards somewhere lower than the oceanic current vector, so that they are affected by both the oceanic current and gravity.

Seaweeds Seaweeds are another class of marine algae that are shorter than kelps, as shown in Fig. Sb). We create seaweed assets using the Differential Growth model as in the Leather Coral assets (See Section Appendix G.3.6). In particular, we choose the parameters so that it has a large growth force towards the positive Z axis. We apply smoothing and subsurfacing to the 2D mesh and solidify it into a 3D mesh. Then the seaweed is bent towards the direction of the oceanic current by a varying degree using the simple deform object modifier. Seaweeds are scattered on the surface of the terrain mesh.

G.4. Surface Scatters

Moss forms dense green clumps or mats on a flat surface. We create individual moss using Bezier curves as skeletons, and then turning them into 3D mesh. We distribute individual moss instance on the boulder surfaces with the Z axis aligned at an angle with the surface normal. Such angle of rotation is guided by a Musgrave texture so that mosses in a neighbourhood have similar rotations. Moss instances have shaders that is built from a mixture of several yellowish to greenish colors, with the color variation determined by a Musgrave texture. Moss can grow on three different places of the boulder, from the faces with a higher Z coordinate than a threshold, or those whose face normal is within a threshold of the positive Z axis, or near edges where there are concave edges.

Lichen is a composite organism that arises from algae that forms a mat on rock surfaces. Individual lichens are made from Differential Growth specified in Appendix G.3.6. The color of lichens comes from a mixture of yellowish-greenish color and white colors, with the mixing ratio guided by a Musgrave texture. Lichen can grow on either boulders or on tree trunks. On boulders, lichens are distributed on all faces of the boulder, or the lower portions of tree trunks with a minimal distance between two instances.

Slime mold are organisms shaped like gelatinous slime that lives on decaying plant materials. We first designate around 20 initial seedling vertices where the slime mold can grow from, and assign a random weight proportional to the local convexity to all edges in the chosen area of the mesh. Then we use geometry nodes to compute the shortest path from each vertex to any of the seedling vertices, and connect these shortest path. These shortest paths from the skeleton of the slime mold. Slime mold only grows on the lower portion of tree trunks

Pine needles Pine needles are the leaves of the pine tree that have fallen onto the ground, as shown in Fig. V. Pine needles are made from a segment of a ellipsoid and are typically of brownish and greenish colors. Pine needles are scattered on certain parts of the terrain surface based on noise texture. Pine needles are scattered on different heights so that pine needles of a certain color are above pine needles of another color.

G.5. Marine Invertebrates

G.5.1 Mollusk

Mollusk is the collection of animals that includes most snails and shells, as shown in Fig. W.

Snails is modeled after animals mostly from the class *Gastropoda*. It features snails of following shape: Conch (Fig. W a), from family *Strombidae*, Auger (Fig. W b), from family *Terebridae*), Volute (Fig. W c),from family *Volutidae*) and Nautilus (Fig. W d),from a different class *Cephalopod*). All these class of animals share the commonality that they live in a shell that grows and rotates at a constant angle as the soft body of the animal grows, which can be modeled using the array object modifier in Blender. We first build the cross section of these snails from the interpolation of a start and a ellipsoid, which gives the space for the soft body to live in, as well as the pointy spikes on some snails' surface. Then we apply the array object modifier onto the cross section so that it is rotated with a constant angle, scaled at a constant ratio, and displaced at a constant interval. These series of cross section can uniquely define the cross section at each stage of the growth, with which we can bridge the edge loops to finally form a 3D mesh for the whole snail. Parameters in this process includes the displacement of cross section along and orthogonal to the axis, the total number cycles of rotations, the ratio that the scale of cross section shrinks, and the other parameters with regard to the shape of the base cross section. The parameters are set differently for individual class of snails: Conch and Auger have large displacement along their axis while Volute and Nautilus have almost none; Conch is made from more spiky cross section mesh, and has less overlapping chambers than Auger; Nautilus has a faster-shrinking cross section, almost no displacement along the axis, and have less over lapping chambers than Volute.

Shells is modeled after animals mostly from the class *Bivalvia*. It features shells of the following shape: Scallop (Fig. W e), from family *Pectinidae*), Clam (Fig. W f), from family *Veneridae* and Mussel (Fig. W g), from family *Mytilidae*). These animals share the commonality that they are covered by two symmetrical shells joining at a point that folds around the soft body of the individual, with both shells growing gradually from inside the shell. To build assets for these shells, we first generate individual shells. We first create a mesh circle and select one point on the circle as its origin. For all points on the circle, we scale its distance from the origin, with the ratio determined by the direction from the origin to the target point. Then we choose a point above the XY-plane, and interpolate between the previous mesh and the newly selected point. The interpolation ratio is determined by a point's distance to the boundary, so that the boundary points are the XY plane, which creates the convex shape of an individual shell. We mirror the shell at an angle, and now we have the 3D mesh of the shell. We have different designs for distinct class of shells: Scallops are given a wavy pattern depending on vertices' direction to the origin, and have girdles near the origin; Clams are the most basic shells with no alternations; Mussels are made from shells that are similar to ellipsoids with large eccentricities.

Mollusk material Both snails and shells grows along a certain direction and leaves a changing color pattern along its growing direction. We define a 2D coordinate (U, V) the mollusks surface for mapping textures, whether U is the growth direction and V is orthogonal to it. For snails, U is the direction of displacement for its cross section mesh, and V is along the boundary of the cross section mesh. For shells, U is the direction from the center of the shell to the boundary of shell, and V is along the boundary of the shell. We design a saw-like wavy texture that progresses along either U or V directions, which creates interchanging color patterns along or orthogonal to the direction of growth. Both snails and shells are given a low-frequency surface displacement, and scatter on the terrain mesh.

G.5.2 Other marine invertebrates

Jellyfish is composed of its cap and tentacles. For the cap, we first generate two mesh uv spheres with one above another, then scale and deform both spheres. Then we subtract the sphere below from the the sphere above, creating the cap with two surfaces, one facing towards the positive Z axis and another facing towards the negative Z axis. Tentacles are made from ribbons along the Z axis, which are later deformed along the X axis and tapered, and finally rotated around the Z axis. Tentacles of varying sizes are placed around the lower surface of the cap. The jellyfish shader is made from a mixture of colored emission, a principled BSDF with transmission, and a colored transparent shader, whose mixing ratio is guided by Fresnel coefficients. We use a more transparent material for outer surface of the cap and shorter tentacles, which are more peripheral parts of the body, and a more opaque material for inner surface of the cap and longer tentacles, which are core organs of a jellyfish. Jellyfish are scattered with a random offset above the ground mesh.

Urchin is a spiny, globular animal living on the sea floor. For modeling urchin assets, we first start with an icosphere. For each face of the icosphere, we extrude it outwards by a small distance, scale it down, and extrude it inwards to form the girdle. We then extrude the faces outwards by a varying but large distance, and scale it down to zero so that they form the spikes that ground on the urchin. The bases of an urchin are from a darker color and the spikes are from a lighter color between purple and yellow, with the girdle's color somewhere in between.

G.6. Creatures

G.6.1 Creature Construction

Each creature genome is a tree of parameters, with nodes specifying parts and edges specifying attachment.

Each node contains a dictionary of named input parameters for one of our part templates (Sec. G.6.4). We compute all parts in isolation before proceeding to attach them. This part template must produce 1) a mesh and 2) a skeleton line. The skeleton line is a 3D parametric curve specifying the center line of the part, and is used for attachment and rigging. Requiring part templates to produce a center line is not a limitation - for NURBS and most node-graph parts it is trivial to obtain. Additionally, this output can be omitted for any part not intended to have further children attached to it. Each part template may also produce additional metadata for use in the animation and material stages.

Each edge contains a coordinate (u, v, r) to determine the attachment location. $(u, v) \in [0, 1]^2$ specifies a location on the parent mesh's surface. For arbitrary meshes, this is computed by travelling u percent of the way along the parent's skeleton and raycasting orthogonally to it, with angle $360° * v$. If the parent part is a NURBS, one can instead query (u, v) on it's parametric surface. Finally, we use r to interpolate between the found surface point, and the corresponding skeleton point, which has the effect of controlling how close to the surface the part is mounted.

Finally, each edge specifies a relative rotation used to pose the part. Optionally, this rotation can be specified relative to the parent part's skeleton tangent, or the attachment surface normal.

G.6.2 Creature Animation

As an optional extra output, each part template may specify where and how it articulates, by specifying some number of *joints*. Each *joint* provides specifies rotation constraints as min/max euler angles, and a parameter $t \in [0, 1]$, specifying how far along the skeleton curve it lies. If a part template specifies no joints, it's skeleton is assumed to be rigid, with only a single joint at $t = 0$. We then create animation bones spanning between all the joints, and insert additional bones to span between parts.

Any joint may also be tagged as a named inverse kinematics (IK) target, which are automatically instantiated. These targets provide intuitive control of creature pose - a user or program can translate / rotate them to specify the pose of any tagged bones (typically the head, feet, shoulders, hips and tail), and inverse kinematics will solve for all remaining rotations to produce a good pose.

We provide simple walk, run and swim animations for each creature. We procedurally generate these by constructing parametric curves for each IK target. These looped curves specify the position of each foot as a function of time. They can be elongated or scaled in height to achieve a gallop or trot, or even rotated to achieve a crab walk or walking in reverse. Once the paths are determined, we further adjust gait by choosing overall RPM, and offsets for how synchronized each foot will be in it's revolution.

G.6.3 Genome Templates

In the main paper, we show the results of our realistic *Carnivore, Herbivore, Bird, Beetle* and *Fish* templates. These templates contain procedural rules to determine tree structure by adding legs, feet, head and appropriate details to create a realistic creature. Each template specifies distributions over all attachment parameters specified above, which provides additional diversity ontop of that of the parts themselves. Tree topology is mostly fixed for each template, although some elements like the quantity of insect legs and presence of horns or fish fins are random.

Our creature system is modular by design, and supports infinite combinations besides the realistic ones provided above. For example, we can randomly combine various creature bodies, heads and locomotion types to form a diverse array of creatures shown in Fig X. As we continue to implement more independent creature parts and templates, the possible combinations of these random creature genomes will exponentially increase.

Creature genomes also support a semi-continuous interpolation operation. Interpolation is trivial for creatures with identical tree structure and part types - one can perform part-wise linear interpolation of node and edge parameters. When tree topology or part types don't match, we recursively compute a matching for each node's children which minimizes the difference of edge attachment parameters, then perform linear interpolation on any node parameters with matching names. To interpolate between a present and missing genome node, we scale the part down from its original size to 0, which results in small vestigial arms or tails on the intermediate creatures. When part types do not align exactly, there is a discrete transition halfway through interpolation, so interpolation is not continuous in all cases.

G.6.4 Creature Parts

NURBS Parameterization Many of our creature body and head templates are comprised of non-uniform rational B-splines (NURBS). NURBS are the generalized 3D analog of a Bézier curve. In order to form a closed shape, we pinch each NURBS surface closed at its ends, and loop it's handles in the V direction to form a closed cylinder as a starting point. We set the U and V knot-vectors to be *Pinned Uniform*, and instead rely on densely-spaced or coincident handles to create sharp edges where necessary.

By default, a NURBS cylinder is represented as an $N \times M$ array of 3D handle locations. We find the space of all NURBS handle configurations too high dimensional and unstructured to randomize directly. Adding Gaussian noise to handle locations produces lumpy, unrealistic creatures, and is unlikely to coordinate to create phenomena like bent limbs or widened midsections. Instead, we randomize under a factored representation. Specifically, we start with a $Nx3$ array of radii and relative angles, which stores a center line for the part as polar-coordinate offsets. Accumulating these produces an $Nx3$ skeleton line. We arrange N *profile shapes* around this center line, each stored as M 3D points centered about the origin. This representation has just as many parameters as the original, but randomizing it produces better results. Adding noise to the polar skeleton angles and radii produces macro-scale changes in body or head shape, and multiplying the profiles by random scalars can easily change the radius or cross section, independent of where along the skeleton that profile is located.

As a starting point for this randomization, we determined skeleton and profile values which visually match a large number of reference animal photos, including *Ducks, Gulls, Robins, Cheetahs, Housecats, Tigers, Wolves, Bluefish, Crappie Fish, Eels, Pickernel Fish, Pufferfish, Spadefish, Cows, Giraffe, Llama* and *Goats*. Rather than make a discrete choice of which mean values to use, we take a random convex combination of all values of a certain category, before applying the extensive randomization described above.

Horns are modeled by a transpiled Blender node graph. The 2D shapes of horns are based on spiral geometry node, supporting adjustable rotation, start radius, end radius and height. The 3D meshes then are constructed from 2D shapes by curve-to-mesh node, along with density-adjustable depth-adjustable ridges. Along with the model, we provide three parameter sets to create goats, gazelles and bulls' horn templates.

Hooves are modeled by NURBS. We start with a cylinder, distributing control points on the side surface evenly. For the upper part of the cylinder, we scale it down and make it tilted to the negative X axis, which makes its shape closed to the horseshoe. The model also has an option to move some control points toward the origin, in order to create cloven hooves for goats and bison. Along with the model, we provide two parameter sets to create horses' and goats' horns templates.

Beaks are modeled by NURBS. Bird beaks are composed of two jaws, generally known as the upper mandible and lower mandible. The upper part starts with a half cone. We use the exponential curve instead of the linear curve of the side surface of the cone to obtain the natural beak shape. The model also has parameters that control how much the tip of the beak hooks and how much the middle and bottom of the beak bulge, to cover different types of beaks. The lower part is modeled by the same model of the upper part with reversed Z coordinates. Along with the model, we provide four parameter sets to create eagles, herons, ducks and sparrows' beaks templates.

Node-Graph Creature Parts All part templates besides those mentioned above are implemented as node-graphs. We provide an extensive library of node-groups to ease the construction of creature parts. The majority of these involve placement and querying of parameterized tubes, which we use to build muscular legs, arms and head parts. For example, our *Tiger Head* and *Quadruped Leg* templates contain nodes to construct the main central form of each part, followed by placement of several tubes along their length to create muscles and detailed forms. This results in a randomizable representation of face and arm musculature, which produces the detailed carnivore heads and legs shown in the main paper. These node-graph tools can also be layered ontop of NURBS as a base representation.

Figure E. 144 randomly generated, non-cherry-picked images of terrain produced by our system (Part 1 of 2). Images are compressed due to space constraints - please see infinigen.org

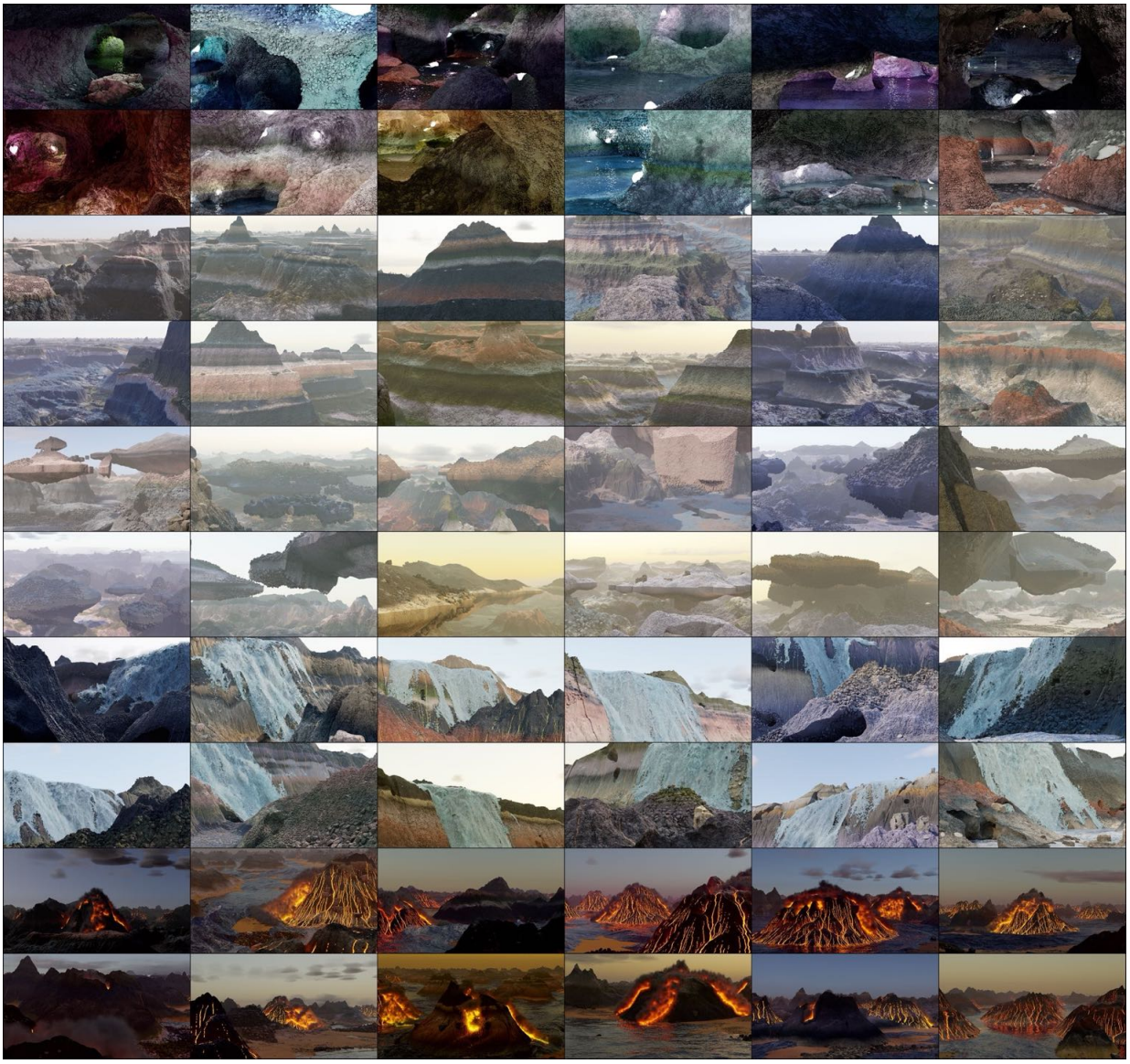


Figure F. 144 randomly generated, non-cherry-picked images of terrain produced by our system (Part 2 of 2). Images are compressed due to space constraints - please see infinigen.org

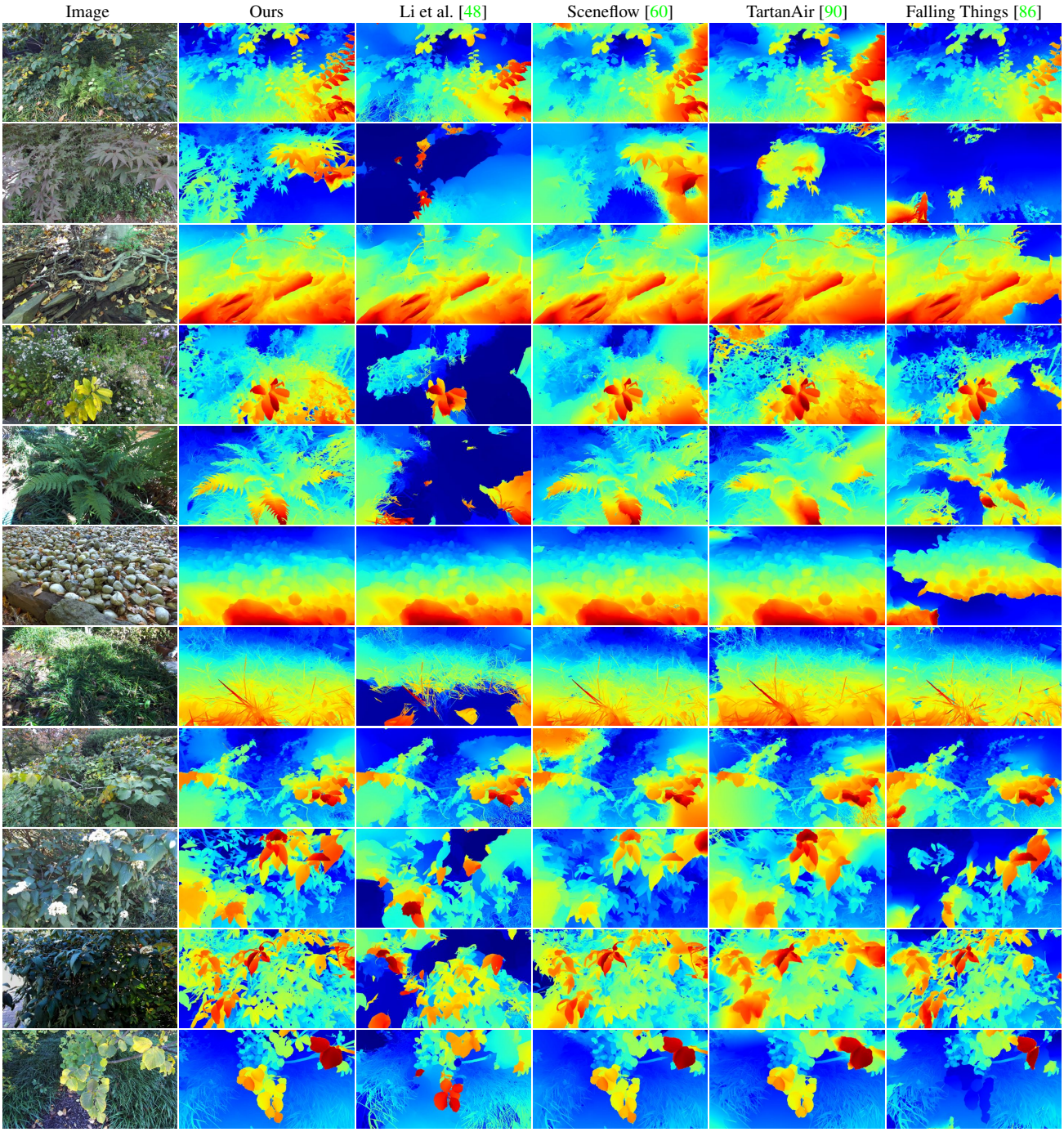


Figure G. Qualitative results on natural stereo photographs. Rectified input images are captured at 2208×2484 resolution using a calibrated ZED 2 stereo camera [79]. Our data generator helps RAFT-Stereo generalize well to real images of natural objects.

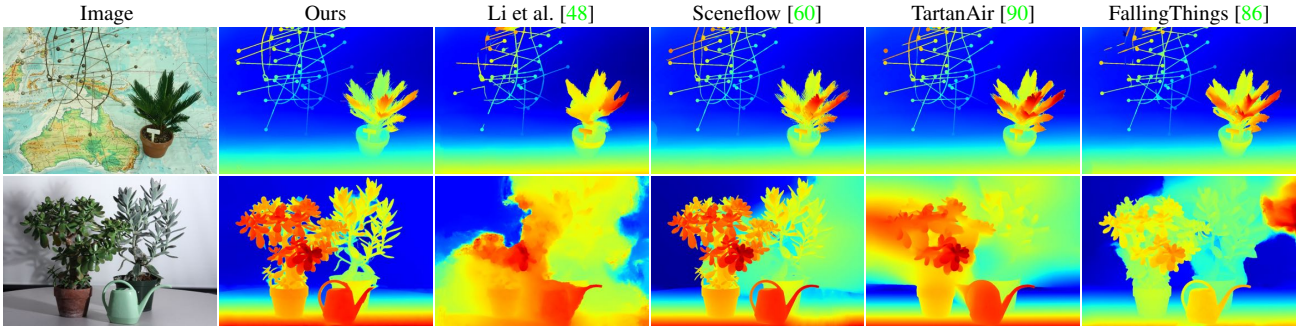


Figure H. Qualitative results on the *Plant* and *Australia* Middlebury [74] test images. RAFT-Stereo trained using Infinigen generalizes well to images with natural objects.

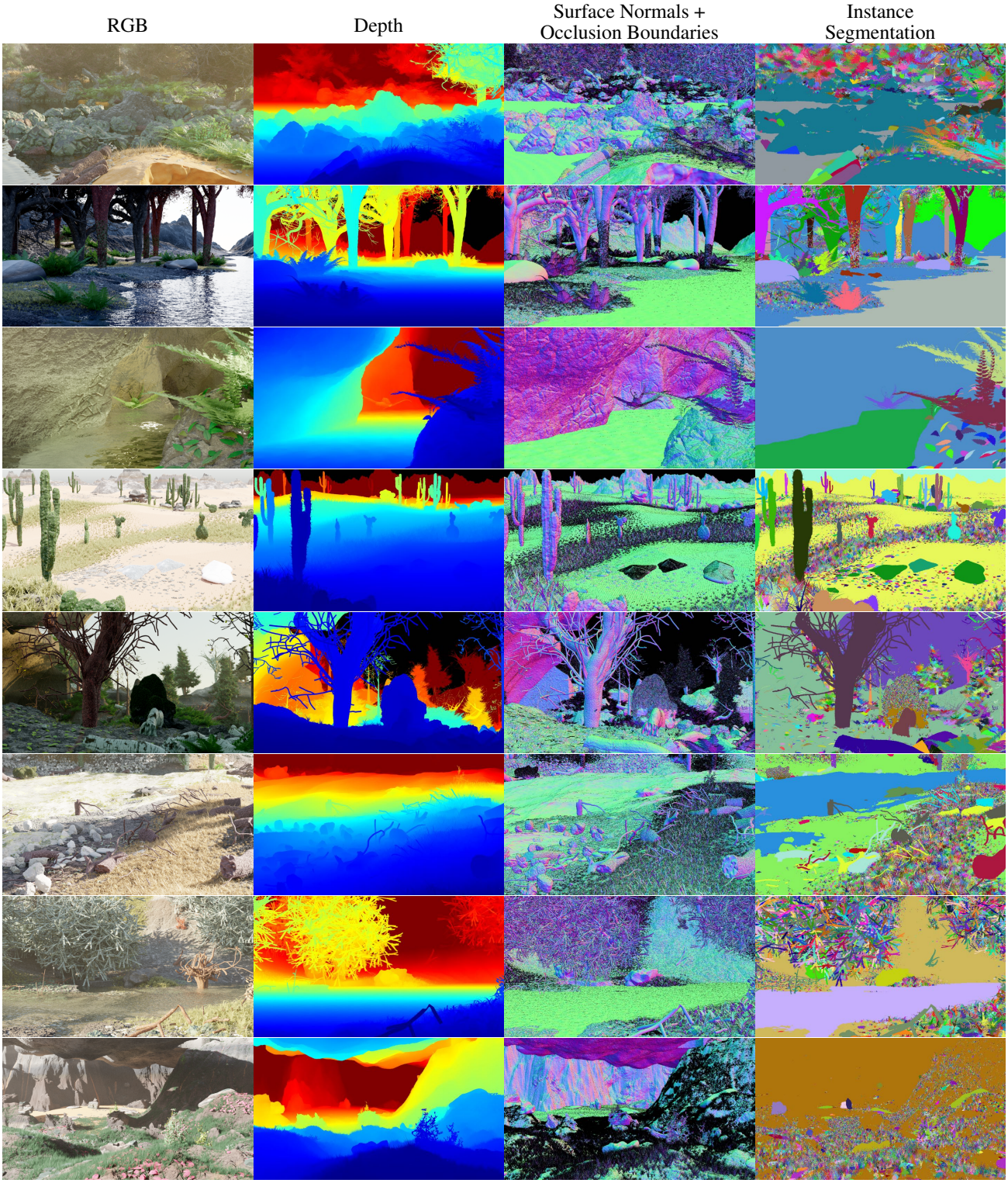
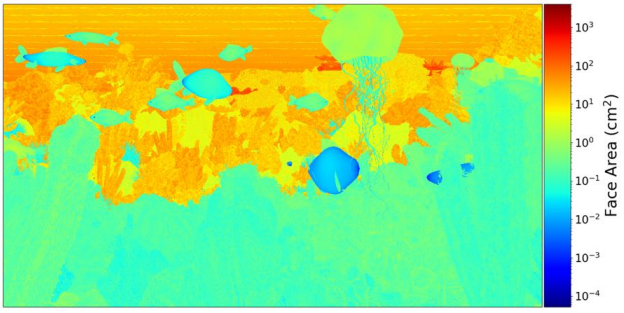
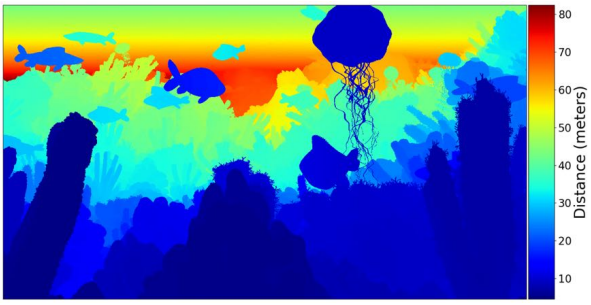


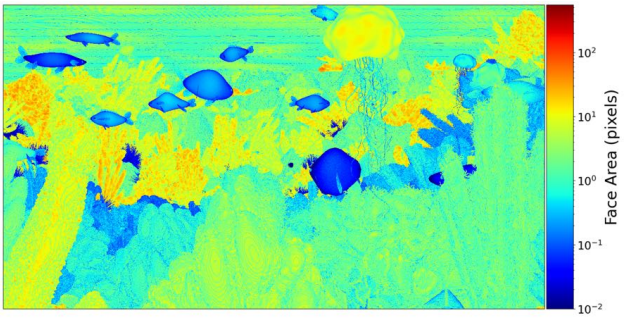
Figure I. High-Resolution Ground Truth Samples. We show select ground truth maps for 8 example Infinigen images. For space reasons, we show only Depth, Surface Normals / Occlusion and Instance Segmentation. Our instance segmentation is highly granular, but classes can be grouped arbitrarily using object metadata. See Sec.C.2 for a full explanation.



(a) The area of mesh faces in cm^2. Our dynamic-resolution scaling causes faces closer to the camera to be smaller.

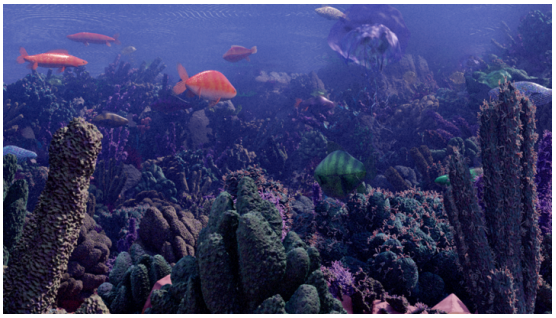


(b) Distance of faces from the camera (i.e. depth). Distance is proportional to the area of faces.

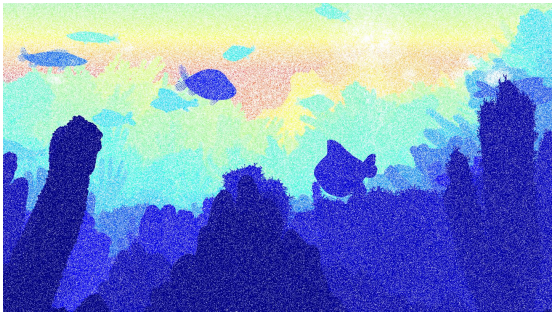


(c) Face area measured in pixels. Our dynamic resolution scaling causes individual mesh faces to appear approximately one pixel across.

Figure J. Dynamic Resolution Scaling. Faces further from the camera are made smaller (a) such that they appear to be the same size from the camera's perspective (c). We show the depth map for reference (b).

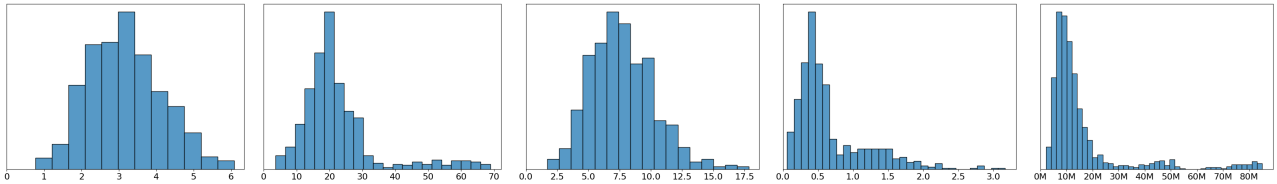


(a) Input Image for reference.



(b) Depth from Blender's built-in render passes.

Figure K. Our ground truth is computed directly from the underlying geometry and is always exact. Prior methods [9, 24, 27, 29, 48] generate ground-truth from Blender's render-passes, which leads to noisy depth for volumetric effects such as water, fog, smoke, and semi-transparent objects.



(a) Wall Time (Hours) (b) Memory (GB) (c) CPU Hours (d) GPU Hours (e) # Triangles per scene

Figure L. Resource requirements for creating a pair of stereo 1080p images using Infinigen. Our mesh resolutions scale with the output image resolution, such that individual mesh faces are barely visible. As a result, these statistics will change for different image resolutions.

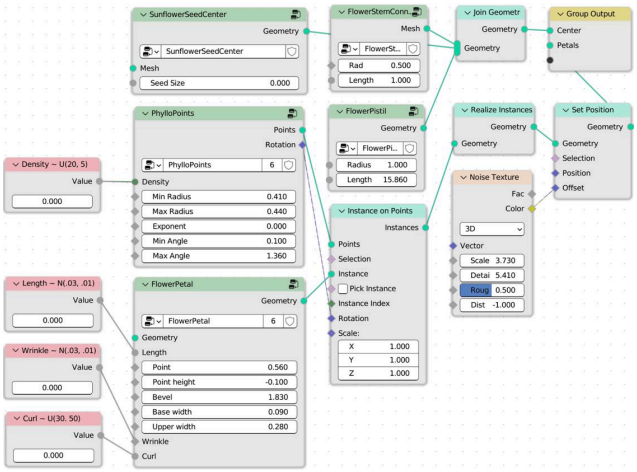


Figure M. An example node-graph, as shown as input to the transpiler in Fig. 3 of the main paper. Dark green nodes are *node groups*, containing user-defined node-graphs as their implementations. Red nodes show tuned constants, with annotations for their distribution.

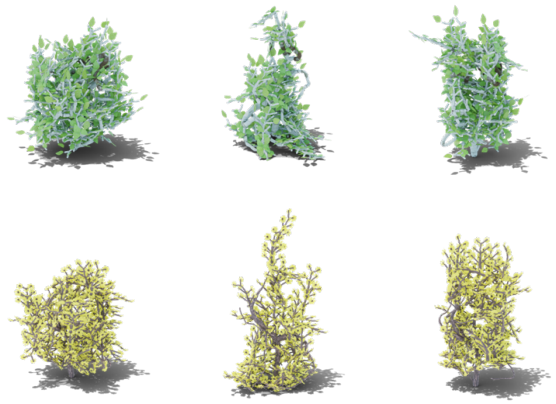


Figure O. We control the shape of bushes by by specifying the distributions of the attraction points. Each row are the same bush species with different shapes (left to right: ball, cone, cube).

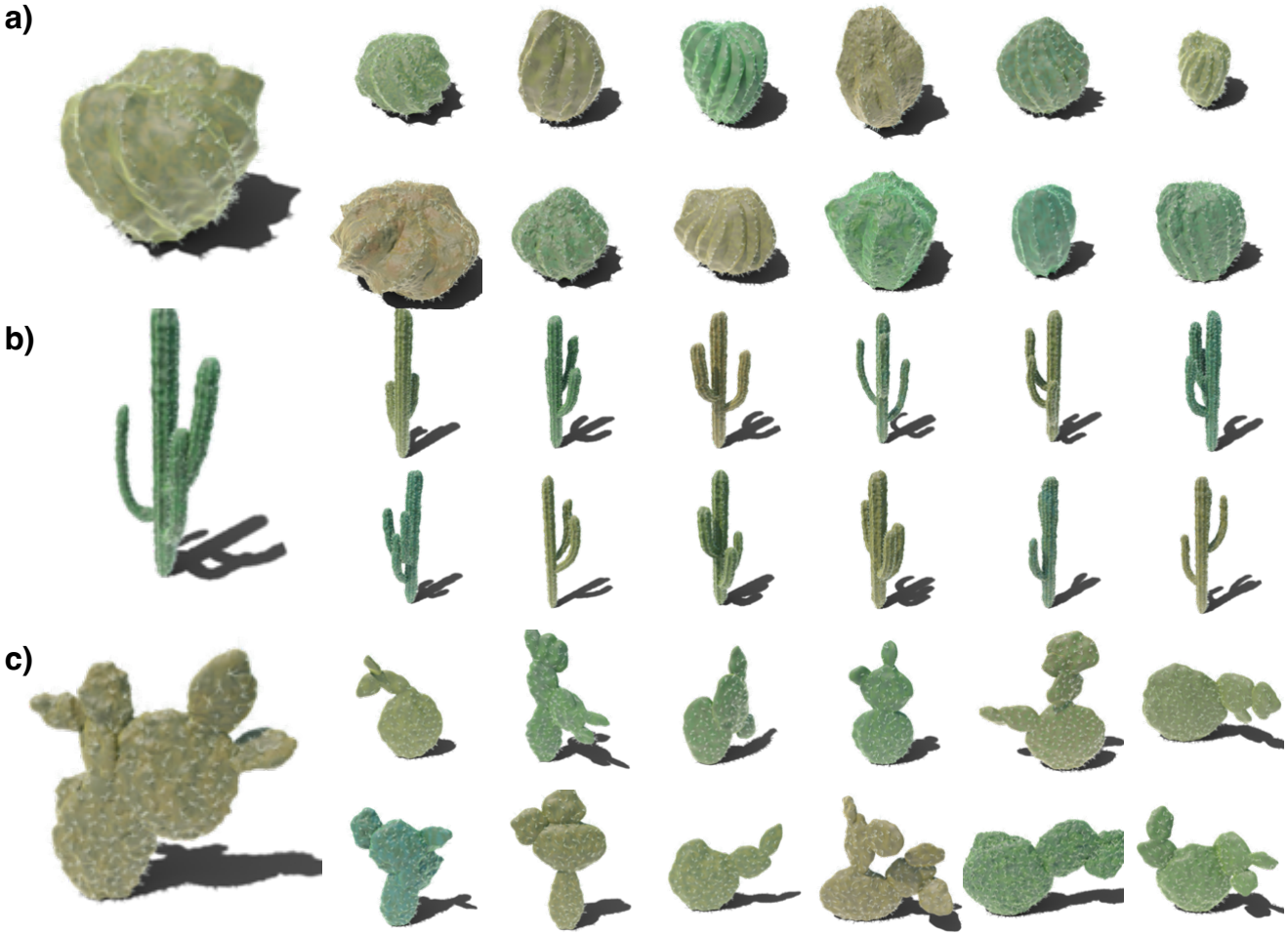


Figure P. All classes of cacti included in Infinigen. Each row contains one class of cactus: **a)** Globular Cactus; **b)** Columnar Cactus; **c)** Pricky pear Cactus.

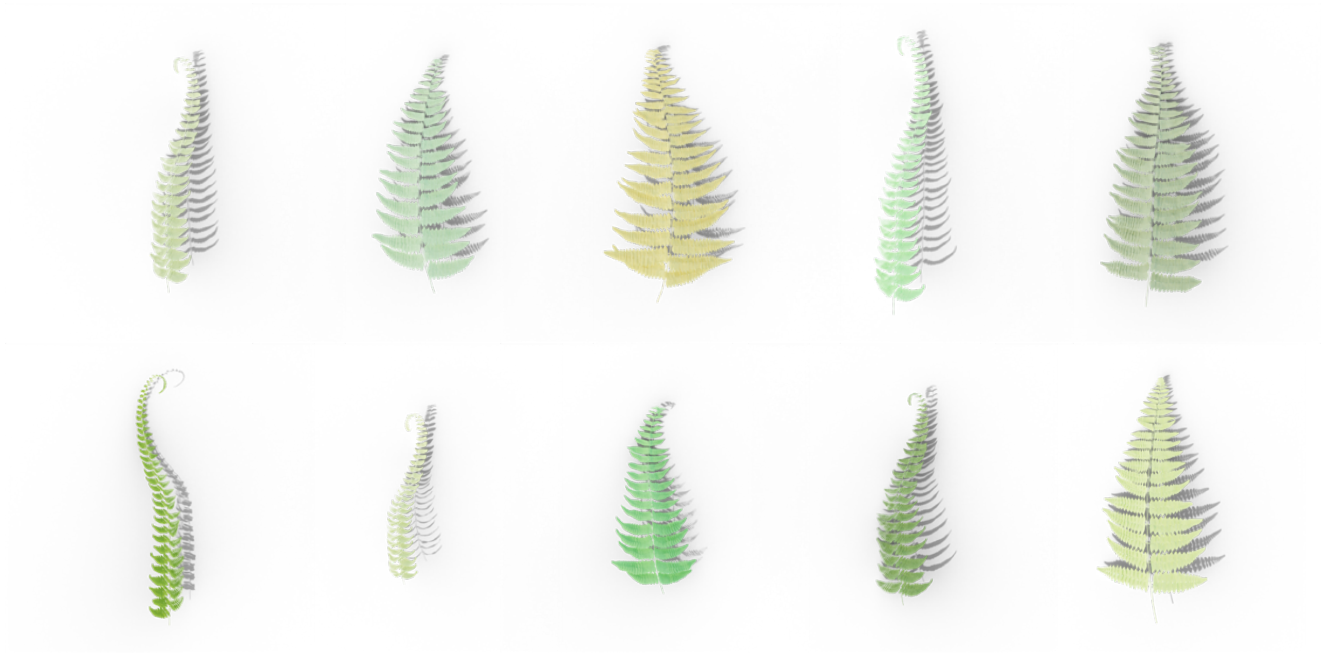


Figure Q. Assets of fern pinnae included in Infinigen. A fern consists of a random number of pinnae in the same color with random orientations.

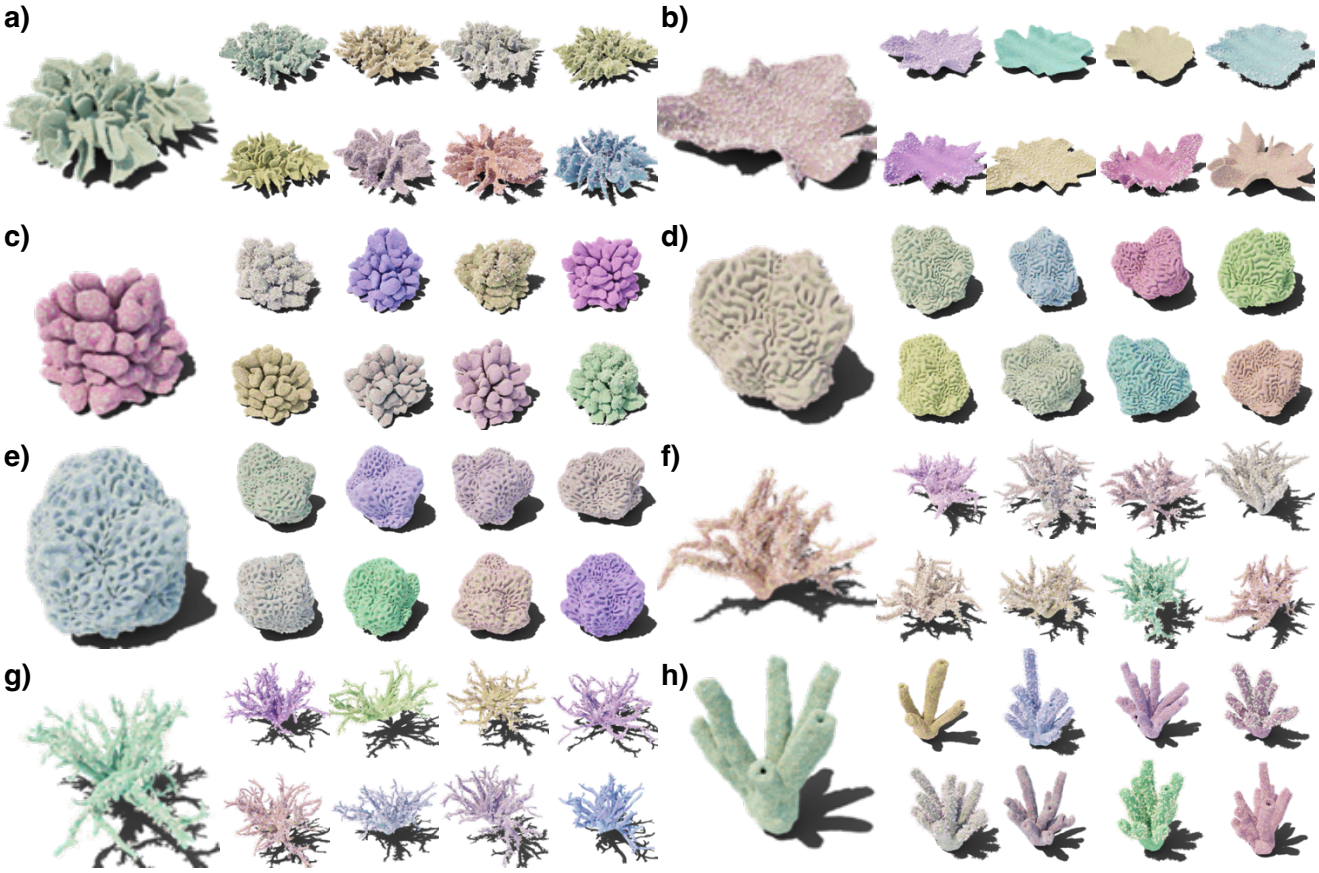


Figure R. All classes of corals included in Infinigen. Each block contains one class of coral: **a)** Leather Coral; **b)** Table Coral; **c)** Cauliflower Coral; **d)** Brain Coral; **e)** Honeycomb Coral; **f)** Bush Coral; **g)** Twig Coral; **h)** Tube Coral.

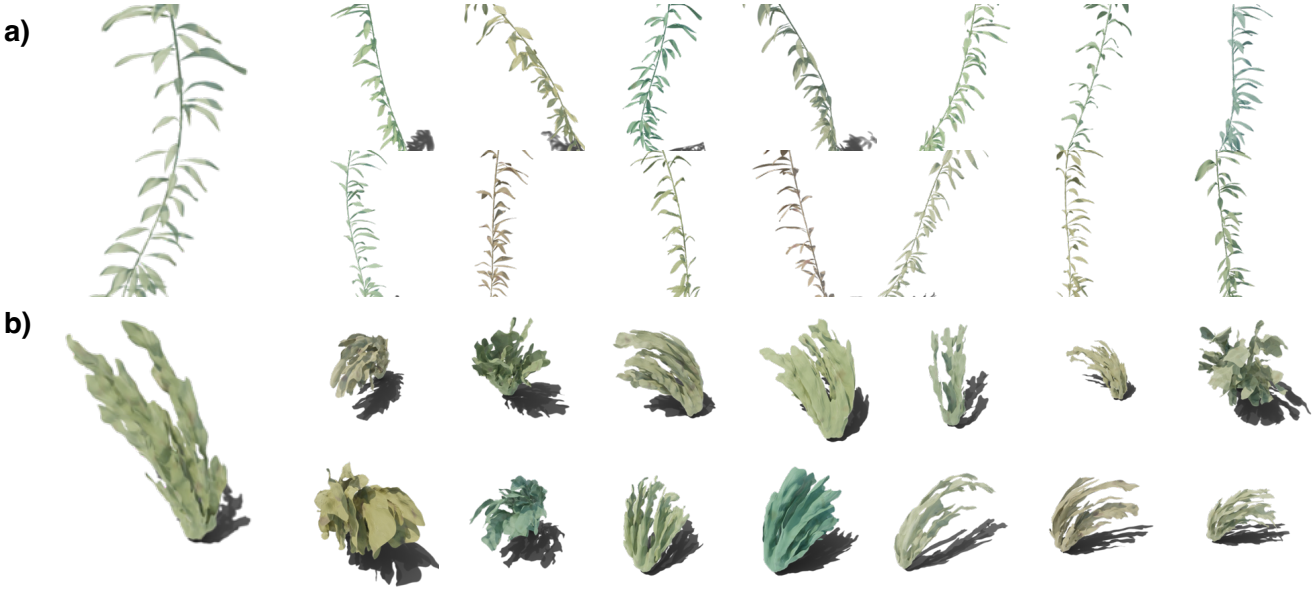


Figure S. Kelps **a)** and seaweeds **b)** examples, each occupying one row.

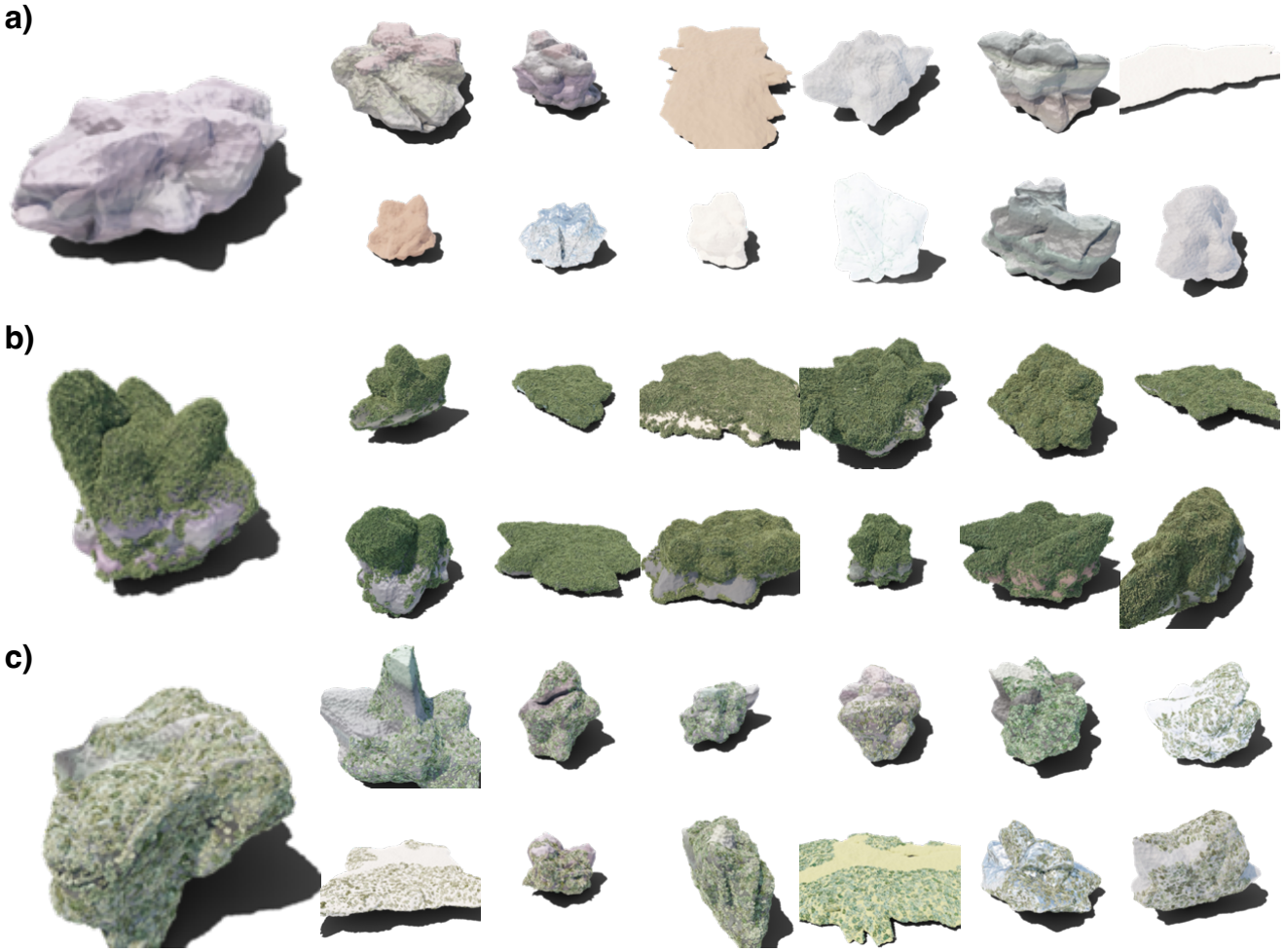


Figure T. Boulder assets with different rock cover surfaces. In particular, each row of boulders are under **a)** no surface; **b)** moss surface; **)** lichen surface.

Figure U. Mushrooms **a)** and pinecones **b)**.

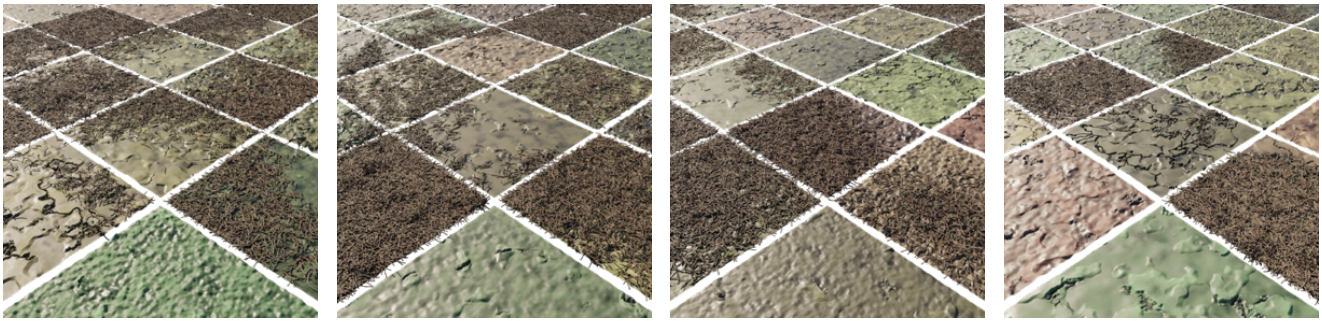


Figure V. Pine needles scattered onto the ground with varying density.

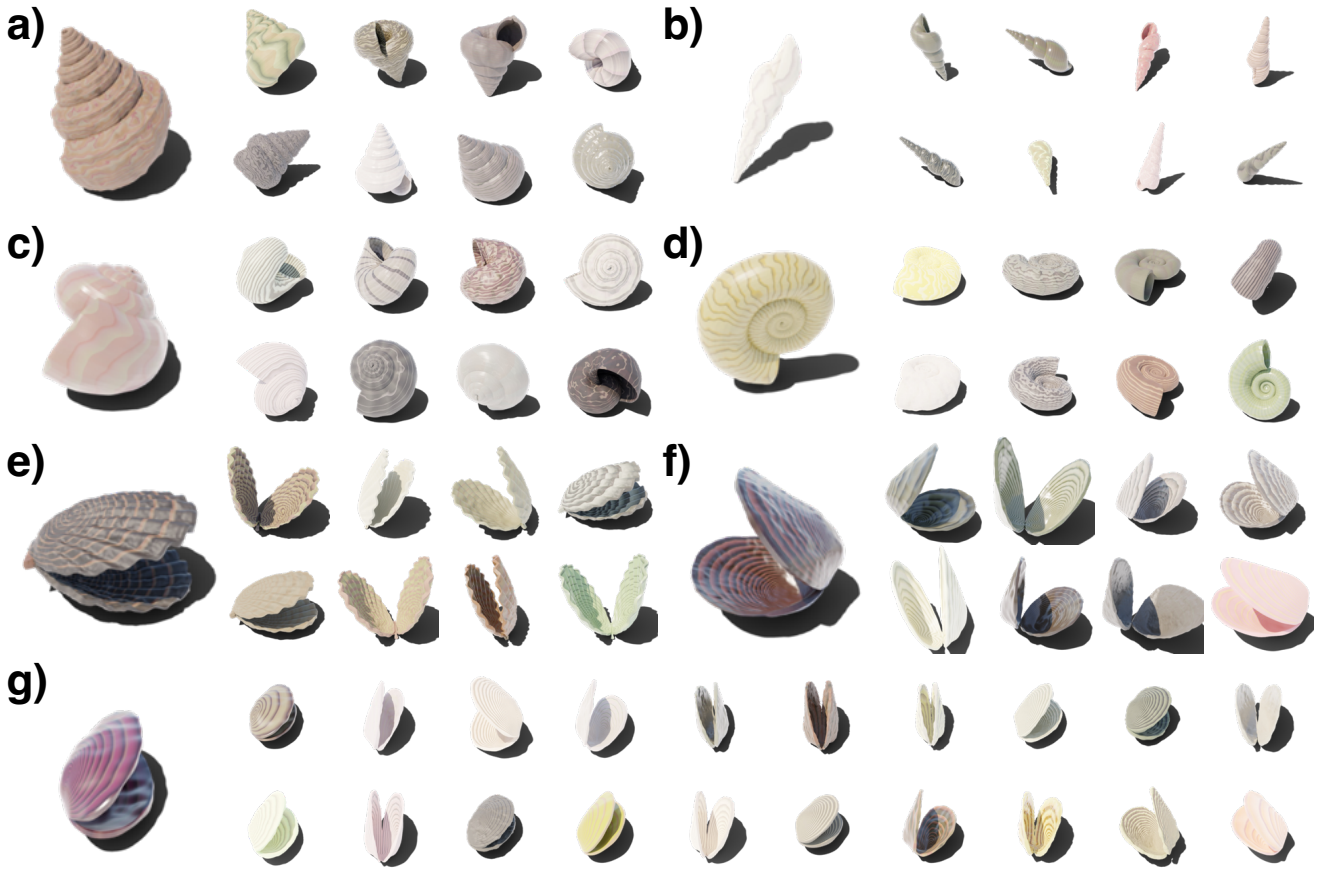


Figure W. Different classes of mollusks, with each block representing a class of mollusk: **a)** Conch, **b)** Auger, **c)** Volute, **d)** Nautilus, **e)** Scallop, **f)** Clam, **g)** Mussel.

Figure X. We provide procedural rules to combine all available creature parts, resulting in diverse fantastical combinations. Here we show a random, non-cherry-picked sample of 80 creatures. Despite diverse limb and body plans, all creatures are functionally plausible and possess realistic fur and materials.

Material Generators	Interpretable DOF	Named Parameters
Mountain	2	Noise Scale, Cracks Scale
Sand	5	Color Brightness, Wave Scale, Wave Distortion, Noise Scale, Noise Detail
Cobblestone	13	Stone Scale, Uniformity, Depth, Crack Width, Stone Colors (5), Mapping Positions (2), Roughness
Dirt	9	Low Freq. Bump Size, Low Freq Bump Height, Crack Density, Crack Scale, Crack Width, Color1, Color2, Noise Detail, Noise Dimension
Chunky Rock	4	Chunk Scale, Chunk Detail, Color1, Color2
Glowing	1	Color1
Granite	6	Speckle Scale, Color1, Color2, Speckle Color1, Speckle Color2, Speckle Color 3
Ice	7	Color, Roughness, Distortion, Detail, Uneven Percent, Transmission, IOR
Mud	12	Wetness, Large Bump Scale, Small Bump Scale, Puddle Depth, Percent water, Puddle Noise Distortion, Puddle Noise Detail, Color1, Color2, Color3, WaterColor1, WaterColor2
Rock	0	
Sandstone	18	Ridge Polynomial (2), Ridge Density, Ridge High Freq., Ridge Noise Mag., Ridge Noise Scale, Ridge Disp:Offset Magnitude, Roughness, Crack Magnitude (2), Crack Scale, Color1, Color2, Dark Patch Percentages (3), Micro Bump Scale, Micro Bump Magnitude
Snow	3	Average Roughness, Grain Scale, Subsurface Scattering
Soil	10	Pebble Sizes (2), Pebble Noise Magnitudes, Pebble Roundness, Pebble Amounts, Voronoi Scale, Voronoi Mag., Base Colors (2), Darkness Ratio
Stone	10	Rock Scale, Rock Deepness (2), Noise Detail, Noise Roughness, Crack Scale, Crack Width, Color1, Color2, Roughness
Aluminium	2	Bump Offset, XY Ratio
Fire	2	Blackbody Intensity, Smoke Density
Smoke	2	Color, Density
Ocean	5	Wave Scale, Choppiness, Foam, Main Color, Cloudiness
Lava	10	Color, Rock Roughness, Amount of Rock, Lava Emission, Min Lava Temp., Max Lava Temp., Voronoi Noise, Turbulence, Wave Scale, Perlin Noise
Surface water	5	Color, Scale, Detail, Lacunarity, Height
Water	6	Ripple Scale, Detail, Ripple Height, Noise Dimension, Lacunarity, Color
Waterfall	3	Color, Foam Color, Foam Density
Bark	9	Displacement Scale, Z Noise Scale, Z Noise Amount, Z Multiplier, Primary Voronoi Scale, Primary Voronoi Randomness, Secondary Voronoi Mix Weight, Secondary Voronoi Scale, Color
Bark Birch	5	Noise Scale (2), Noise Detail (2), Displacement Scale,
Greenery	13	Color Noise (3), Roughness Noise (3), Roughness Min/Max (2), Translucence Noise (3), Translucence Min/Max (2)
Wood	3	Scale, XY Ratio, Offset
Grass	6	Wave Scale, Wave Distortion, Musgrave Scale, Musgrave Distortion, Roughness Min/Max (2), Translucence
Leaf	2	Base Color, Vein Color
Flower	5	Diffuse Color, Translucent Color, Translucence, Center Colors (2), Center Color Coeff.
Coral Shader	5	Bright Color, Dark Color, Light Color, Fresnel Color, Musgrave Scale
Slime Mold	7	Edge Weight, Spline Parameter Cutoff, Seedlings Count, Min Distance, Bright Color, Dark Color, Musgrave Scale
Lichen, Moss	6	Bright Color, Dark Color, Musgrave Scale, Density, Min Distance, Instance Scale
Bird	7	Bird Type, Head Ratio, Stripe Width, Stripe Noise, Neck Ratio, Color1, Color2.
Bone	3	Bump Scale, Bump Frequency, Bump Offset.
Chitin	3	Boundary Width, Noise Weight, Thorax Size.
Horn	8	Noise Scales (2), Noise Details (2), Mapping Control Points (4)
Reptile Brown	3	Circle Scale, Circle Boundary, Noise
Fish Body	7	Scale Size, Scale Noise, Fish Type, Scale Offset, Color1 Ratio, Color2 Ratio, Noise
Fish Fin	7	Offset Z, Offset Y, Shape, Bump Noise, Fin Type, Bump Weight, Transparency
Giraffe	4	Scale, Noise, Circle Width, Belly
Reptile	3	Scale, Offset, Noise
Reptile Gray	2	Noise1, Noise2
Reptile 2-Color	2	Color1, Color2
Scale	3	Scale Size, Scale Noise, Scale Rotation
Slimy	2	Scale, Offset
Spot Sparse	3	Spot Scale, Color1, Color2
3-Color Spots	2	Spot1 Ratio, Spot2 Ratio
Tiger	4	Belly, Stripe Distortion, Stripe Frequency, Stripe Shape
2-Color Spots	4	Offset, Spot Scale, Ratio, Noise
Mollusk	8	UV Pattern Ratio, Scale, Distortion, Pattern Type, Hue Range, Saturation Range, Value Range, Colors Per Pattern
Num. Generators: 50	Total: 271	

Table C. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Material Generators*.

Terrain Generators	Interpretable DOF	Named Parameters
3D Noise, Wind-Eroded Rocks	0	
Caves	3	Cavern Size, Tunnel Frequency, Fork Frequency
Voronoi Rocks, Grains	2	Rock Frequency, Warping Frequency
Sand Dunes	2	Dune Frequency, Warping Frequency
Mountains, Floating Islands	2	Mountain Frequency, Num. Scales
Coast line	2	Coast curve frequency, Height mapping function
Ground Slope	0	
Still Water, Ocean	0	
Atmosphere	0	
Tiled Landscape	0	
Scene Types (Arctic, Canyon, Cave, Cliff, Waterfall, Coast, Desert, Mountain, Plain, River, Underwater, Volcano)	6	Tile Types, Tile Heights, Tile Frequency, Element Probabilities, Water Level, Snow
Num. Generators: 26	DOF: 17	

Table D. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Terrain Generators*. Terrain is heavily simulation and noise-based, so has few interpretable DOF but uncountable internal complexity.

Lighting, Weather & Fluid Generators	Interpretable DOF	Named Parameters
Dust, Rain, Snow, Windy Leaves	6	Density, Mass, Lifetime, Size, Damping, Drag,
Cumulus, Cumulonimbus, Stratocumulus, Altocumulus	13	Density, Anisotropy, Noise Scale, Noise Detail, Voronoi Scale, Mix Factor, Increased Emission, Angular Density, Mapping Curve (6)
Atmospheric Fog, Dust	5	Density Min, Density Max, Color, Noise Scale, Anisotropy
Lava/Water	6	Viscocity, Viscocity Exponent, Surface Tension, Velocity Coord, Spray Particle Scale, Flip Ratio
Fire/Smoke	12	Max Temp, Gas Heat, Bouyancy, Burn Rate, Flame Vorticity, Smoke Vorticity, Dissolve Speed, Noise Scale, Noise Strength, Surface Emission, Turbulence Scale, Turbulence Strength
Sky Light	8	Overall Intensity, Sun Size, Sun Intensity, Sun Elevation, Altitude, Air Density, Dust Density, Ozone Density
Caustics	5	Scale, Sharpness, Coordinate Warping, Power, Spotlight Blending
Glowing Rocks	3	Wattage, Colors, Shape Distortion
Camera Lighting (Flashlight, Area Light)	3	Wattage, Light Size, Blending
Num. Generators: 19	DOF: 61	

Table E. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Lighting, Weather and Fluid Generators*

.

Rock Generators	Interpretable DOF	Named Parameters
Rocks	3	Aspect Ratio, Deform, Roughness
Stalagmite / Stalactite	3	Num. Extrusions, Length, Z Offset Variance
Boulder	6	Initial Vertices Count, Is Slab, Large Extrusion Probability, Small Extrusion Probability, Large Extrusion Distance, Small Extrusion Distance
Num. Generators: 4	DOF: 12	

Table F. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Rock Generators*.

Plant & Underwater Generators	Interpretable DOF	Named Parameters
Flower	8	Center Radius, Petal Dimensions (2), Seed Size, Petal Angle Range (2), Wrinkle, Curl
Maple	7	Stem Curve Control Points, Stem Rot. Angle, Polar Mult. X, X Wave Control Points, Y Wave Control Points, Warp End Rad., Warp Angle
Pine	4	Midpoint (2), Length, X Angle Mean
Broadleaf	14	Midrib Length, Midrib Width, Stem Length, Vein Asymmetry, Vein Angle, Vein Density, Subvein Scale, Jigsaw Scale, Jigsaw Depth, Midrib Control Points, Shape Control Points, Vein Control Points, Wave X, Wave Y
Ginko	11	Stem Control Points, Shape Curve Control Points, Vein Length, Blade Angle, Polar Multiplier, Vein Scale, Wave, Scale, Margin Scale, X Wave Control Points, Y Wave Control Points
Pinecone	13	Bud Float Curve Angles, Bud Float Curve Scales, Bud Float Curve Z Displacements, Bud Instance Rotation Perturbation, Bud Instance Probability, Bud Instance Count, Profile Curve Radius, Max Bud Rotation, Rotation Frequency, Stem Height, Bright Color, Dark Color , Musgrave Scale
Urchin	12	Subdivision, Z Scale, Bevel Percentage, Spike Probability, Girdle Height, Extrude Height, Spike Scale, Base Color, Girdle Color, Spike Color, Transmission, Subsurface Ratio
Seaweed	7	Ocean Current, Deform Angle, Translation Scale, Expansion Scale, Bright Color, Dark Color, Musgrave Scale
Jellyfish	16	Cap Height, Cap Scale, Cap Perturbation Scale, Long Opaque Tentacles Count, Short Transparent Tentacles Count, Arm Screw Angle, Arm Screw Offset, Arm Taper Factor, Arm Displacement Strength, Arm Min Distance, Arm Placement Angle Threshold, Bright Color, Dark Color, Transparent Color, Fresnel Color, Musgrave Scale
Kelp	14	Ocean Current, Axis Shift, Axis Length, Axis Noise Stddev, Leaf Scale, Leaf Float Curve Length, Leaf Float Curve Width, Leaf Float Curve Z Displacement, Leaf Rotation Perturb, Leaf Tilt, Leaf Instance Probability, Leaf Instance Rotation Stride, Leaf Instance Rotation Interpolation Factor, Leaf Instance Count
Shells (Scallop, Clam, Mussel)	9	Top Control Point, Shell Interpolation Ratio, Shell Float Curve Angles, Shell Float Curve Scales, Radial Groove Scale, Radial Groove Frequency, Hinge Length, Hinge Width, Angle Between Shells
Snail (Volute, Nautlius, Conch)	8	Cross Section Affine Ratio, Cross Section Spiky Perturbation, Cross Section Concavity, Lateral Movement, Longitudinal Movement, Rotation Frequency, Scaling Ratio, Loop Count
Reaction Diffusion Coral	10	Intialization Bump Count, Initialization Bump Stride, Timesteps, Step size, Diffusion Rate A, Diffusion Rate B, Feed Rate, Kill Rate, Perturbation Scale, Smooth Scale
Tube Coral	6	Face Perturbation, Short Extrude Length Range, Long Extrude Length Range, Extrusion Direction Perturbation, Drag Direction, Extrusion Probability
Laplacian Coral	8	Timesteps, Kill Rate, Step size, Tau, Eps, Alpha, Gamma, Equilibrium Temperature
Tree Coral	9	Branch Count, Secondary Branch Count, Tertiary Branch Count, Horizontal Span, Length, Secondary Length, Tertiary Length, Base Radius, Radius Decay Ratio
Diff. Growth Coral	8	Colony Count, Max Polygons, Noise Factor, Step Size, Growth Scale, Drag Vector, Replusion Radius, Inhibit Shell Factor
Coral Tentacles	7	Min Distance, Z Angle Threshold, Radius Threshold, Density, Branch Count, Branch Length, Color
Grass Tuft	10	Num. Blades, Length Std., Curl Mean, Curl Std., Curl Power, Blade Width Variance, Taper Mean, Taper Variance, Base Spread, Base Angle Variance
Fern	15	Pinna Rotation (2), Pinnae Rotation (2), Pinnae Gravity, Age, Age Variety, Num Pinna, Pinnae Contour, Num Pinnae Varieties, Num Leaves, Leaf Width Randomness, Num Pinnae, Pinnae Rotation Randomness (2),
Mushroom	17	Cross Section Float Curve Angles, Cross Section Float Curve Scales, Cross Section Center Offset, Cross Section Z Rotation, Stem Length, Stem Radius, Cap Groove Ratio, Cap Scale Ratio, Cap Radius Float Curve Height, Cap Radius Float Curve Radius, Has Web, Umbrella Radius, Umbrella Height, Bright Color, Dark Color , Light Color, Musgrave Scale
Flower Stem	15	Branch Leaf Rotation, Branch Leaf Instance, Branch Stem Radius, Branch Rotation Coeff, Branch Leaf Density, Stem Branch Density, Stem Branch Scale, Stem Branch Range, Stem Leaf Instance, Stem Leaf Rotation, Stem Flower Instance, Stem Flower Scale, Stem Rotation Coeff. STem Radius, Num Versions, Rotation Z
Cactus Spikes	6	Branches Count, Secondary Branches Count, Min Distance, Top Cap Percentage, Density, Color
Globular Cactus	5	Groove Scale, Groove Count, Rotation Frequency, Profile Curve Height, Profile Curve Radius
Columnar Cactus	9	Radius Decay Branch, Radius Decay Root, Radius Smoothness Leaf, Branch Count, Nodes Per Branch, Nodes Per Second Level Branch, Base Radius, Perturbation Scale, Groove Scale
Pricky Pear Cactus	5	Leaf Profile Curve Width, Leaf Profile Curve Height, Leaf Instance Scale Ratio, Leaf Instance Placement Angles, Leaf Instance Count
Num. Generators: 30	DOF: 258	

Table G. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Plant and Underwater Invertebrate Generators*.

Creature Generators	Interpretable DOF	Named Parameters
Geonodes Quadruped Body	12	Start Rad., End Rad., Ribcage Proportions (3), Flank Proportions (3), Spine Coeffs (3), Aspect Ratio
Geonodes Bird Body	3	Start Rad., End Rad. Aspect Ratio, Fullness
Geonodes Carnivore Head	15	Start Rad., End Rad., Snout Proportions (3), Aspect Ratio, Lip Muscle Coeff. (3) Jaw Muscle Coeff. (3), Forehead Muscle Coeff (3)
Geonodes Neck	5	Start Rad. End Rad., Neck Muscle Coeffs. (3)
NURBS Bodies/Heads (Carnivore, Herbivore, Fish, Beetle and Bird)	8	Start/End Dims (4), Proportions, Angle Offsets, Profile Offset, Bump Offset,
Jaw	9	Rad1, Rad2, Width Shaping, Canine Size, Incisor Size, Tooth Density, Tooth Crookedness, Tongue Shaping, Tongue X Scale
QuadrupedBackLeg	15	Start Rad., End Rad., Aspect Ratio, Thigh Coeffs (6), Calf Coeffs (6),
QuadrupedFrontLeg	21	Start Rad., End Rad., Aspect Ratio, Shoulder Coeffs. (6), Forearm Coeffs (6), Elbow Coeffs (6)
Bird Leg	9	Start Rad., End Rad., Aspect Ratio, Thigh Coeffs (3), Shin Coeffs (3)
Insect Leg	9	Start Rad., End Rad., Carapace Rad. Spike Length, Spike Start Rad., Spike End Rad., Spike Range (2), Spike Density
Ridged Fin	10	Width, Roundness, Ridge Frequency, Offset Weight (2), Ridge Rot., Affine (2), Noise Ratio (2)
Feather Tail	9	Feather Dims (3), Max Rotation (3), Rotation Randomness (3)
Feather Wing	7	Start Rad., End Rad., Feather Density, Feather Form Sculpting, Wing Extendedness, Feather Rot. Randomness (2)
Beak	17	Curve Y, Curve Z, Hook Coeff. (2), Hook scale (2), Hook Pos. (2), Hook Thickness (2), Crown Scale, Crown Coeff. (2), Bump Scale, Bump L, Bump R, Sharpness
MammalEye	9	Radius, Eyelid Thickness, Eyelid Fullness, Tear Duct Placement (3), Eye Corner Placement (3)
Ear	5	Start Rad., End Rad., Depth, Thickness, Curl Angle
Insect Mandible	4	Start Rad, End Rad, Curl, Aspect Ratio
Nose	3	Radius, Nostril Size, Smoothness
Hoof	8	Claw Y Scale, Claw Z Scale, Claw Sag, Angle Length, Angle Rad. Start, Ankle Rad. End, Upper Shape, Lower Shape
Horn	6	Rad. Start, Rad. End, Ridge Thickness, Ridge Density, Ridge Depth, Height
Foot	12	Start Rad., End Rad., Toe Density, Toe Dimensions (3), Toe Splay, Footpad Radius, Claw Curl, Claw Dimensions (3)
Tail	4	Start Rad., End Rad., Curl, Aspect Ratio
Cotton, Skin, Rubber Simulation	8	Max Bending Stiffness, Max Compression Stiffness, Goal Spring Force, Pin Stiffness, Shear Stiffness (2), Tension Stiffness, Pressure
Running Animation	6	Steps Per Second, Stride Length, Gait spread, Stride Height, Upturn, Downstroke
Short Hair, Fluffy Hair, Feathers	18	Clump Num., Avoid Eyes Dist., Avg. Length, Avg. Puff, Length Noise (2), Puff Noise (2), Combing, Strand Noise (3), Tuft Spread, Tuft Clumping, Hair Radius, Intra-clump Noise, Length Falloff, Roughness
Carnivore Genome	15	Head Ratio, Head Attachment, Jaw Ratio, Jaw Attachment, Eye Attachment (3), Nose Attachment, Ear Attachment (3), Shoulder Dist., Shoulder Splay, Leg Ratio
Herbivore Genome	22	Neck Start T., Hoof Angle, Foot Angle, Head Interp Temp., Jaw Ratio, Jaw Attachment, Eye Attachment (3), Nose Attachment, Ear Attachment (3), Shoulder Dist., Shoulder Splay, Leg Ratio, Include Nose, Include Horns, Horn Attachment (3), Body Interp Temp
Bird Genome	17	Head Ratio, Head Attachment, Tail Attachment (2), Leg Length Ratio, Foot Size Ratio, Leg Attachment (3), Wing Length Ratio, Wing Attachment (3), Head Ratio, Eye Attacment (3)
Insect Genome	8	Leg Density, Leg Splay, Leg Length Ratio, Include Mandibles, Mandible Attachment (3), Has Hair
Fish Genome	11	Dorsal Fin Ratio, Pelvic Fin Ratio, Pectoral Fin Ratio, Hind Fin Ratio, Fin Attachment (3), Eye Attachment (3) Body Interp Temp.
Random Genome	10	Has Wings, Locomotion Type, Hair Type, Interp Temperature, Head Type, Has Eyes, Nose Type, Has Jaw, Has Ears, Has Horns
Num. Generators: 39	DOF: 315	

Table H. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Creature Generators*.

Tree Generators	Interpretable DOF	Named Parameters
Random Tree, Pine Tree, Bush	26	Growth Height, Trunk Warp, Num. Trunks, Branching Start, Branching Angle, Branching Density, Branch Length, Branch Warp, Pull Dir. Vertical, Pull Dir. Horizontal, Outgrowth, Branch Thickness, Twig Density, Twig Scale, Twig Pts, Twig Branching Start, Twig Rot. Randomness, Twig Branching Density, Twig Init Z,Twig Z Randomness, Twig Subtwig Size, Twig Subtwig Momentum, Twig Subtwig Std., Twig Size Decay, Twig Pull Factor, Space Colonization Shape
Num. Generators: 3	DOF: 26	

Table I. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Tree Generators*.

Scene Composition Generators	Interpretable DOF	Named Parameters
Scene Generators (Arctic, Canyon, Cave, Cliff, Coast, Desert, Forest, Mountain, Plain, River, Under Water)	110	Asset/Scatter inclusion probabilties (39), Num. Creature/Plant Subspecies (4), Noise Mask Scales (30), Mask Tapering Coeff. (12), Normal Mask Thresholds (14), Placement Densities (5), Placement Habitats (6)
Num. Generators: 11	DOF: 110	

Table J. Our full system contains 182 procedural asset generators and 1070 interpretable DOF. Here we show parameters for just our *Scene Composition Configs*. Each config references a terrain composition generator from Fig. D, and specifies a realistic distribution of other assets to create a fully realistic natural environment.