Infinite Photorealistic Worlds using Procedural Generation: Supplementary Materials

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

Contents

1. Figures Extended

First, we provide a large sample of random, non-cherry-picked RGB images from our dataset generator (Figs. 1, 2, 3, 4). 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. 5 6), 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.

2. Experiments

To validate the usefulness of the generated data, we produce 30K image pairs for training rectified stereo matching. We train RAFT-Stereo [16] 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 [16].

†work done while a student at Princeton University

Figure 1. 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

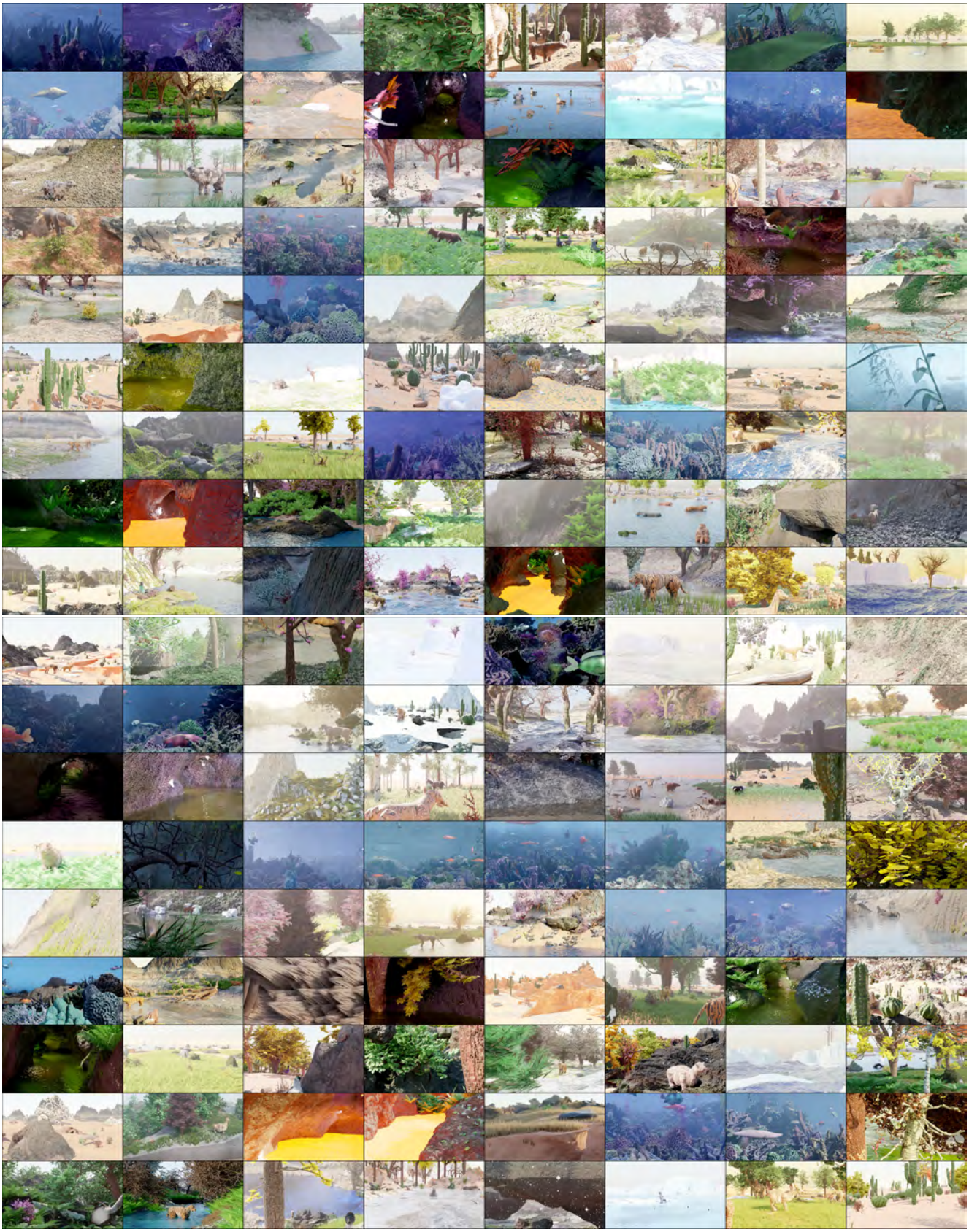


Figure 2. 576 randomly generated, non-cherry-picked images produced by our system (Part 2 of 4). Images are compressed due to space constraints - please see infinigen.org

Figure 3. 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 4. 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

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 [1] and visualize the predicted disparity maps from RAFT-Stereo [16] in Fig. 7. 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 [25], 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. 2, 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 8, 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 [2]	25.282	2K
FallingThings [26]	12.199	62K
Sintel-Stereo [5]	10.253	2K
HR-VS [28]	9.296	780
Li et al. [15]	9.271	177K
SceneFlow [18]	7.837	35K
TartanAir [27]	6.504	296K
Ours (Infinigen 30K)	**5.527**	31K

Table 1. 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. 7. In Tab. 1, 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. 1 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.

3. Dataset Generation

3.1. 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.

3.2. Ground Truth Extraction

We develop custom code for extracting ground-truth directly from the geometry. Prior datasets [5, 9–11, 15] 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. 9. In Fig. 11, 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

Training Dataset	Adirondack	**Jadeplant**	Motorcycle	Piano	Pipes	Playroom	Playtable	Recycle	Shelves	Vintage	Avg
FallingThings [26]	8.3	43.3	**12.3**	18.2	25.3	29.7	50.0	10.4	43.3	45.6	28.6
Sintel-Stereo [5]	35.7	62.9	31.1	24.1	31.9	41.7	60.1	30.8	55.8	76.1	45.0
HR-VS [28]	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. [15]	23.9	80.2	40.7	32.0	40.3	49.1	67.5	36.6	51.7	42.3	46.4
SceneFlow [18]	**7.4**	41.3	14.9	16.2	33.3	**18.8**	**38.6**	**10.2**	**39.1**	29.9	**25.0**
TartanAir [27]	15.5	45.1	18.1	**12.9**	28.4	25.6	51.0	20.9	49.1	**28.2**	29.5
InStereo2K [2]	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 2. Bad 3.0 (%) ↓ error on the Middlebury [25] 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.

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. 9. 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. 9. 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.

3.3. 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. 12. 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.

4. 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. 3–10 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 3–10, placed at the end of this document as they fill several pages.

5. 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. 13. 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. 13. When annotations of this format are detected, the transpiler automatically inserts calls to appropriate random number generators into the resulting code.

6. 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.

6.1. 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.

6.2. Dynamic Resolution

In Fig. 10, 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.

6.2.1 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.

6.2.2 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.

6.2.3 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.

7. Asset Implementation Details

7.1. 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.

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.1.4 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.

7.2. Terrain

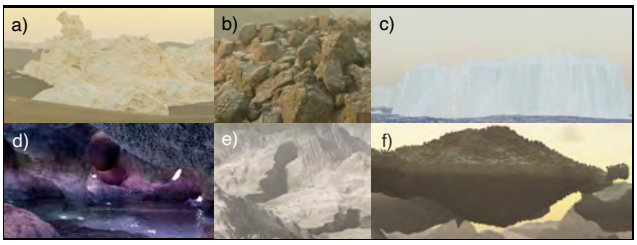


Figure 14. Terrain elements, including a) wind-eroded rocks, b) Voronoi rocks, c) Tiled landscapes, d/e) Caves, and f) Floating Islands.

7.2.1 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 [23] from FastNoise Lite [22] with domain warping adapted from an article [7]. See Fig. 14(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. 14(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 [6], 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 [19], snowfall by diffusion algorithm in Landlab [13] [3] [12]. 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. 14(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 [17]. 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. 14(d) for an interior view of a cave and Fig. 14(e) for how it cut from mountains.

Floating islands Besides natural scenes, we also have fantastical terrain elements, e.g., floating islands (Fig. 14(f)) by gluing mountains and upside down mountains together.

7.2.2 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. 7.1 for details. Boulders are placed on the terrain mesh as placeholders.

7.2.3 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. 7.1.

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 [4]. 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.

7.2.4 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.

7.2.5 Lighting

The majority of scenes are lit only by the sun and sky. We simulate these using the built in *Nishita* [20] 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.

7.3. Plants

7.3.1 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. 21 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.

7.3.2 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 [24] 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. 15.

7.3.3 Cactus

Globular Cactus is modeled after cactus from genus *Ferocactus*, as shown in Fig. 16a). 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. 16b). 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. 16c). 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.

7.3.4 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.17.

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

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

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.

7.3.5 Mushroom

Mushrooms are modeled after real mushrooms from genus *Agaricus* and *Phallus*, as shown in Fig. 21 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 7.3.6), 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.

7.3.6 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. 18. We elaborate these templates for the main coral bodies as follows:

Leather Coral is modeled after corals from genus *Sinularia*, as shown in Fig. 18a). It features curvy surfaces that folds on it self. We implement Leather Coral using a iterative mesh generation technique named Differential Growth from [21]. 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. 18b). 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. 18c). It features wart-like growth on its surface. We implement Cauliflower Coral after [14]'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. 18d). 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:

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

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. 18e). It features honeycomb-shaped holes on the surface of the coral. We use the same Gray-Scott reaction-diffusion model [8] 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. 18f). 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. 18g). 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. 18h). 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.

7.3.7 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. 19.

Kelps Kelps are large brown algae that lives in shallow waters, as shown in Fig. 19a). 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. 19b). We create seaweed assets using the Differential Growth model as in the Leather Coral assets (See Section Appendix 7.3.6). 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.

7.4. 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 7.3.6. 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. 22. 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.

7.5. Marine Invertebrates

7.5.1 Mollusk

Mollusk is the collection of animals that includes most snails and shells, as shown in Fig. 23.

Snails is modeled after animals mostly from the class *Gastropoda*. It features snails of following shape: Conch (Fig. 23 a), from family *Strombidae*, Auger (Fig. 23 b), from family *Terebridae*), Volute (Fig. 23 c),from family *Volutidae*) and Nautilus (Fig. 23 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. 23 e), from family *Pectinidae*), Clam (Fig. 23 f), from family *Veneridae* and Mussel (Fig. 23 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.

7.5.2 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.

7.6. Creatures

7.6.1 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. 7.6.4). 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 $360^\circ * v$. 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.

7.6.2 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.

7.6.3 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 24. 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.

7.6.4 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 NM 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.

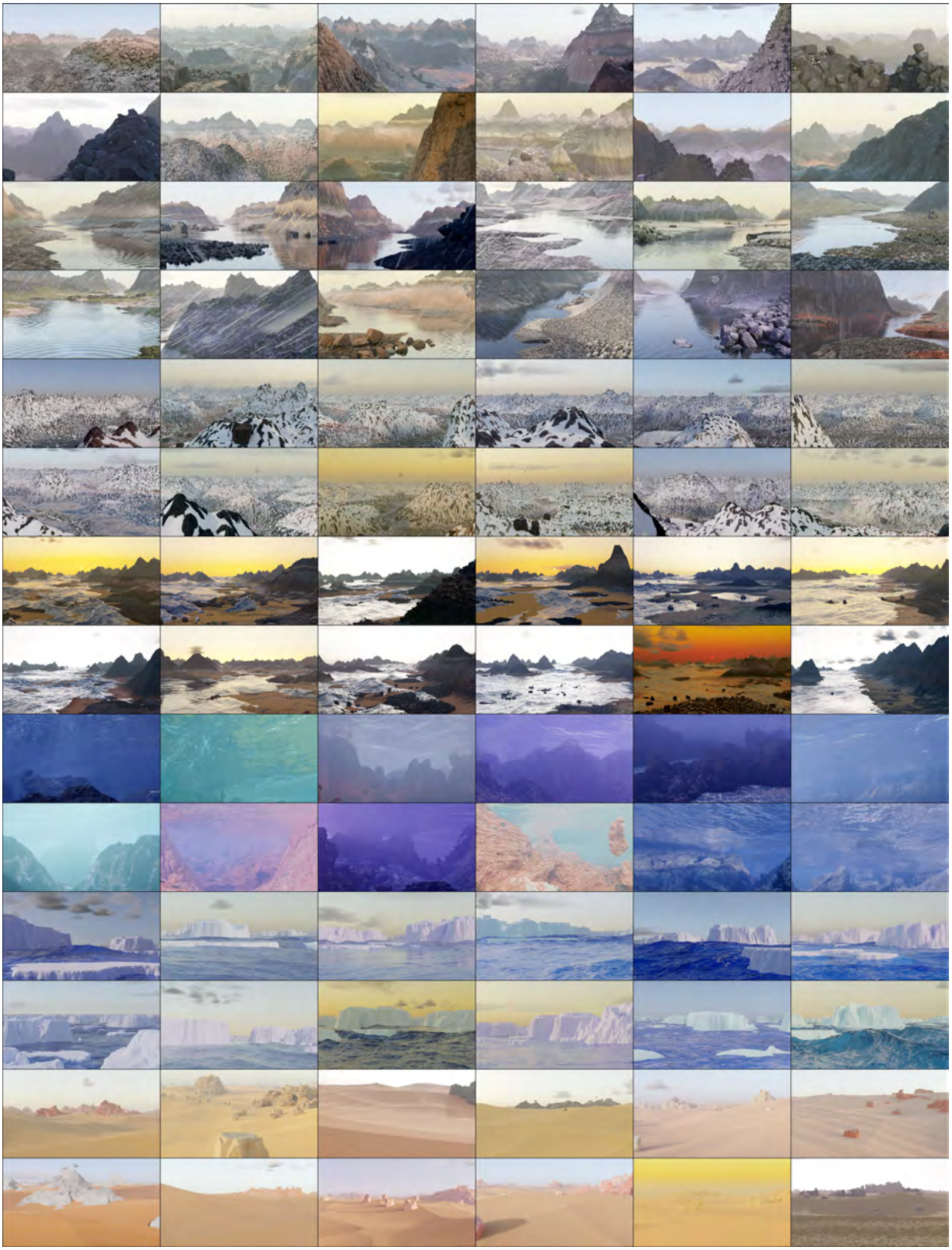


Figure 5. 144 randomly generated, non-cherry-picked images of terrain produced by our system (Part 1 of 2). Images are compressed due to space constraints - please see infinigen.org

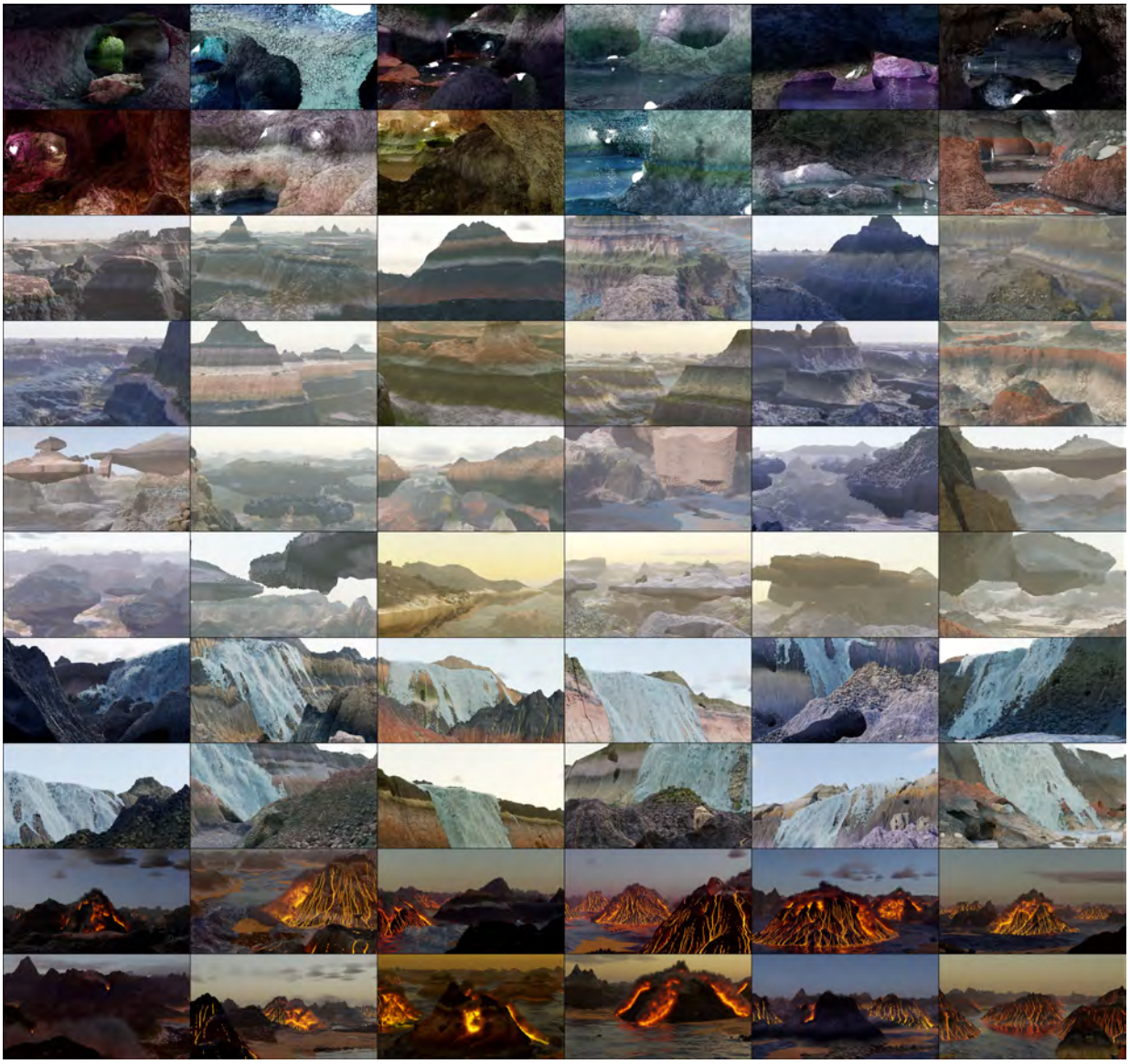


Figure 6. 144 randomly generated, non-cherry-picked images of terrain produced by our system (Part 2 of 2). Images are compressed due to space constraints - please see infinigen.org

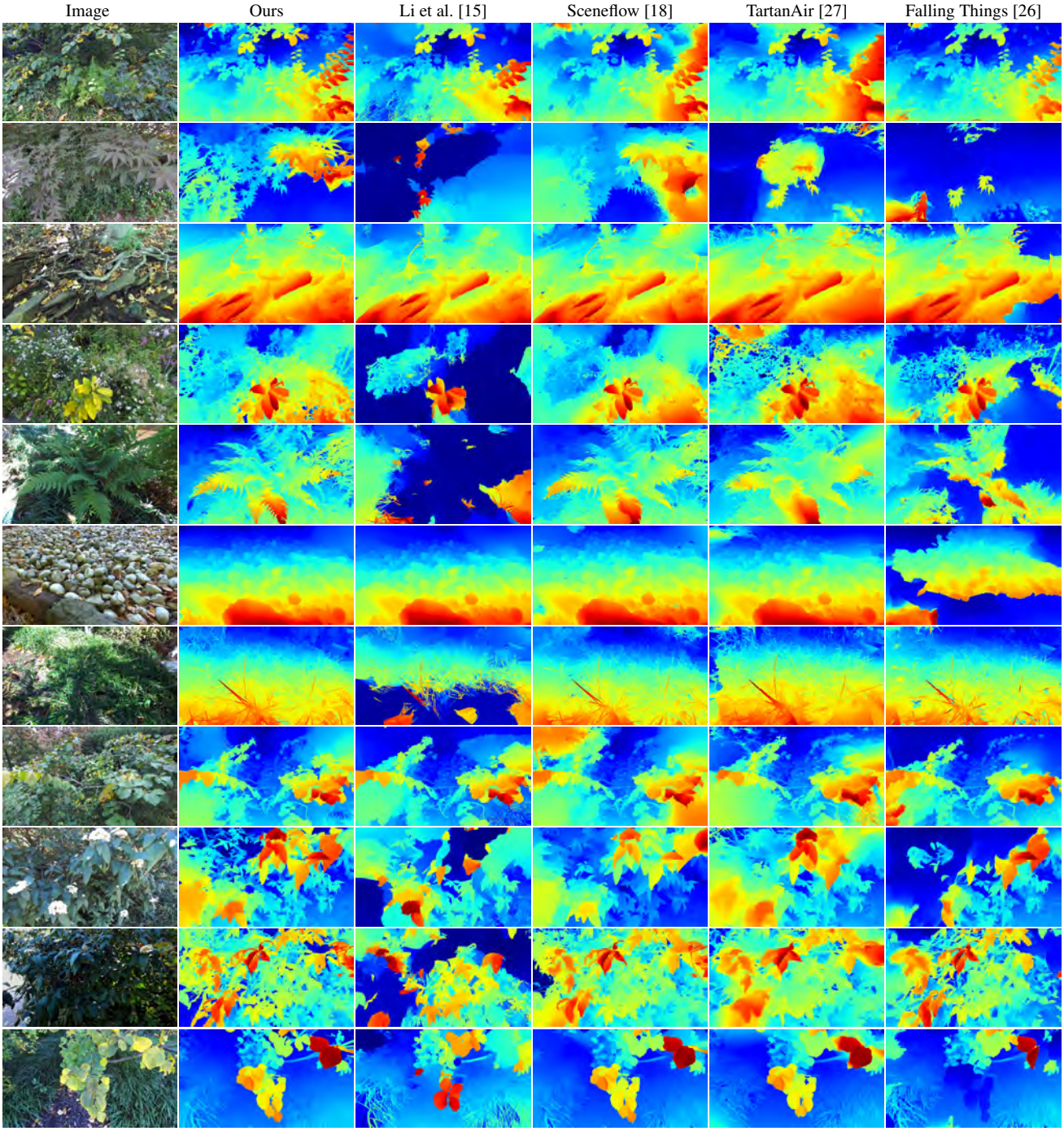


Figure 7. Qualitative results on natural stereo photographs. Rectified input images are captured at 2208×2484 resolution using a calibrated ZED 2 stereo camera [1]. Our data generator helps RAFT-Stereo generalize well to real images of natural objects.

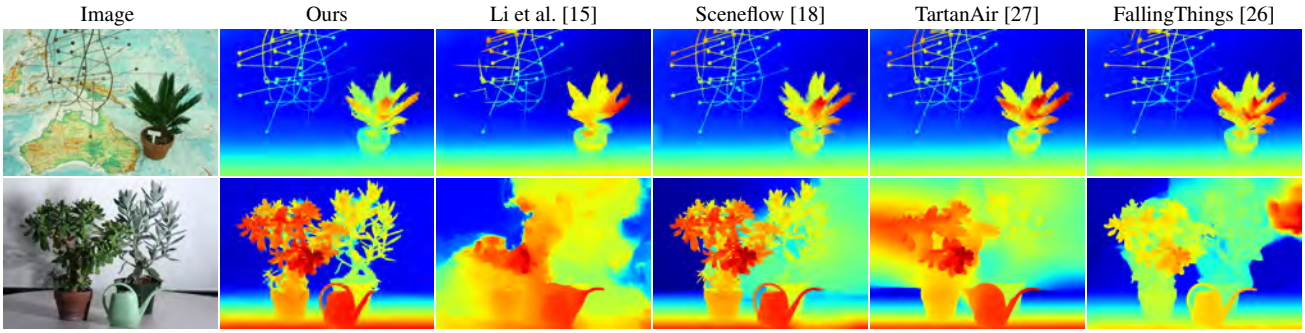


Figure 8. Qualitative results on the *Plant* and *Australia* Middlebury [25] test images. RAFT-Stereo trained using Infinigen generalizes well to images with natural objects.

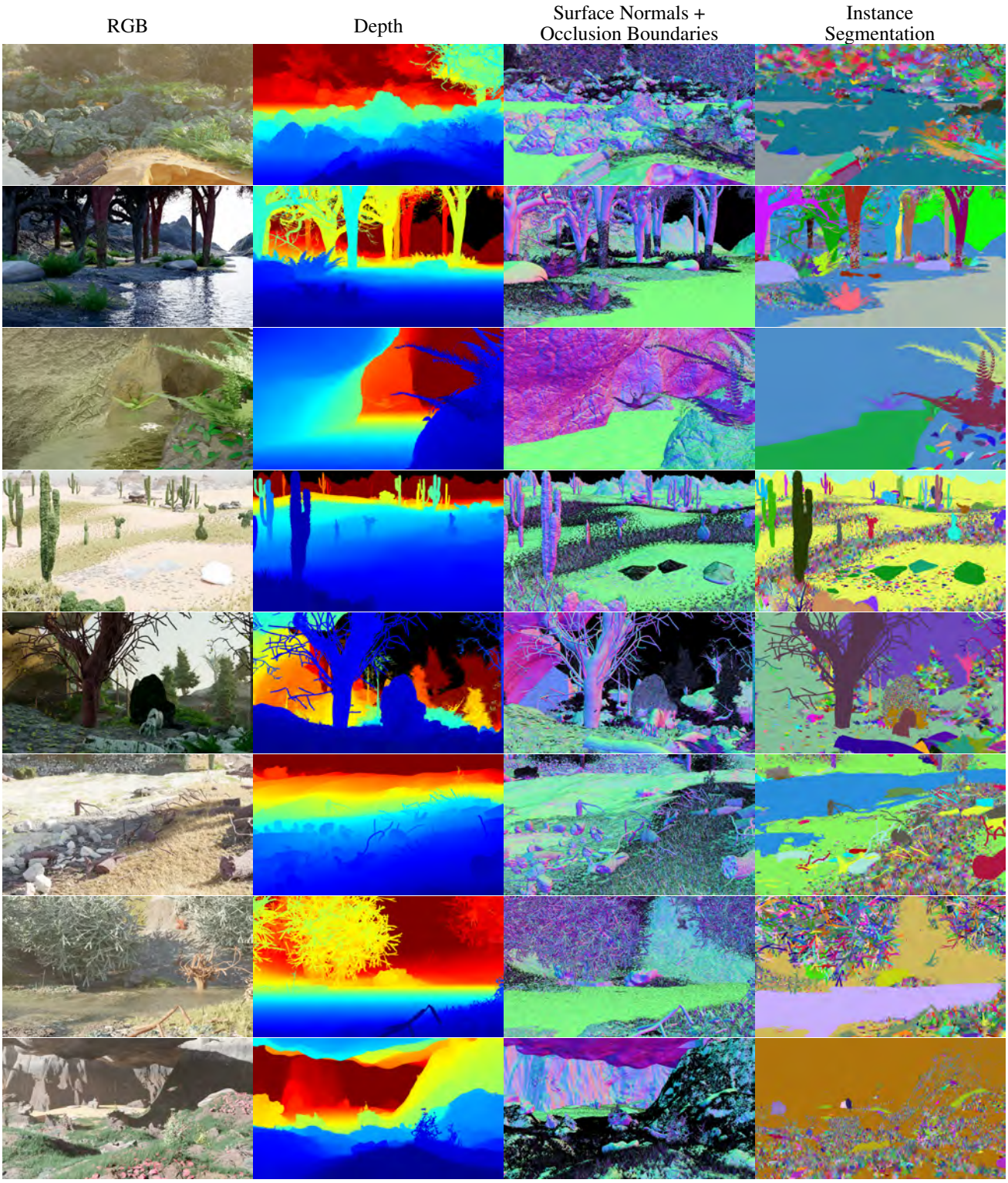
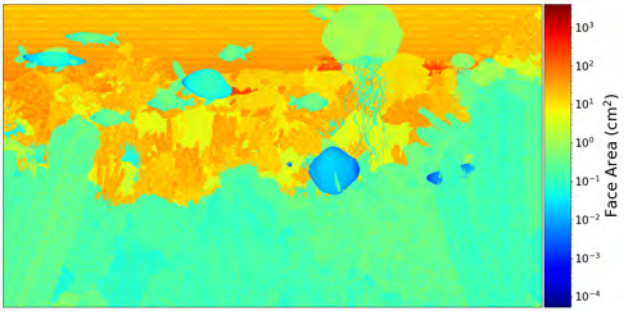
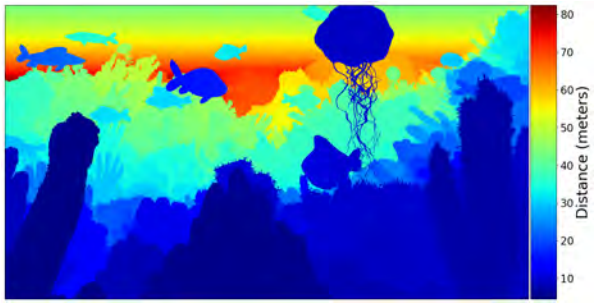


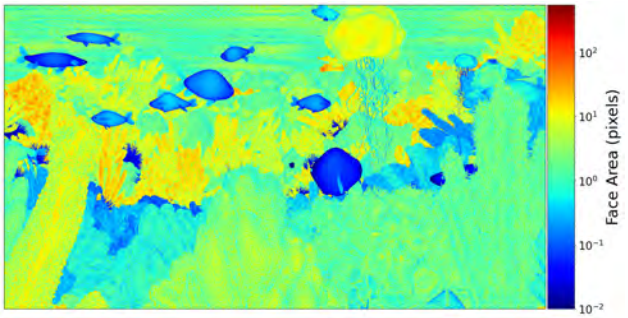
Figure 9. High-Resolution Ground Truth Samples. We show select ground truth maps for 8 example Infinigen images. For space reasons, we show only Depth, Surface Normals / Occlusion and Instance Segmentation. Our instance segmentation is highly granular, but classes can be grouped arbitrarily using object metadata. See Sec.3.2 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.



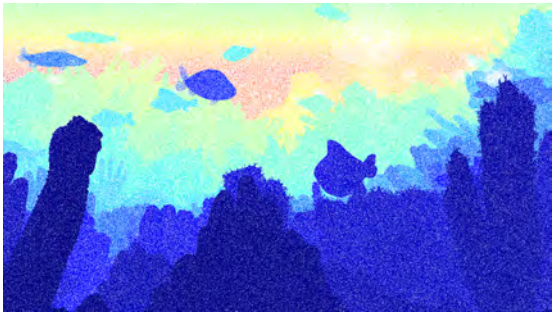
(b) Distance of faces from the camera (i.e. depth). Distance is proportional to the area of faces.



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

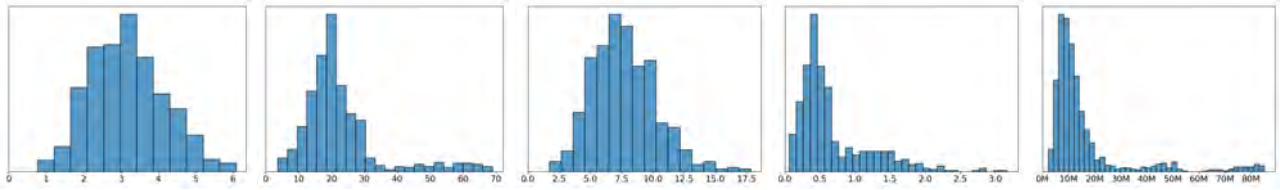
Figure 10. Dynamic Resolution Scaling. Faces further from the camera are made smaller (a) such that they appear to be the same size from the camera's perspective (c). We show the depth map for reference (b).

(a) Input Image for reference.



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

Figure 11. Our ground truth is computed directly from the underlying geometry and is always exact. Prior methods [5, 9–11, 15] 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 12. 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.

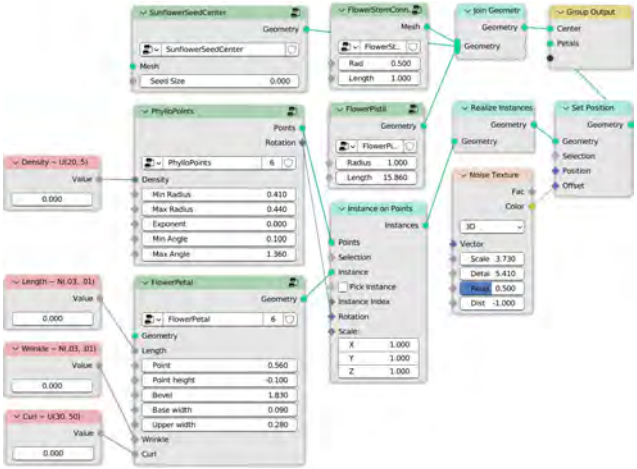


Figure 13. An example node-graph, as shown as input to the transpiler in Fig. 3 of the main paper. Dark green nodes are *node groups*, containing user-defined node-graphs as their implementations. Red nodes show tuned constants, with annotations for their distribution.

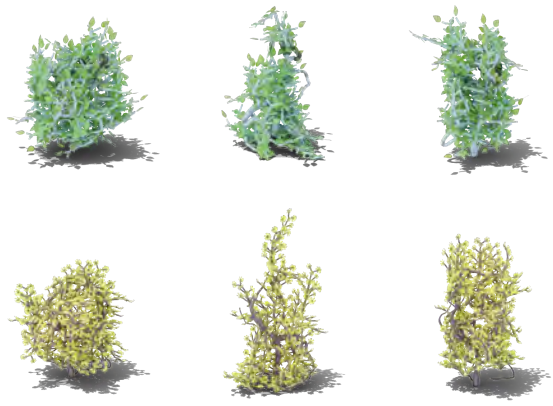


Figure 15. We control the shape of bushes by by specifying the distributions of the attraction points. Each row are the same bush species with different shapes (left to right: ball, cone, cube).

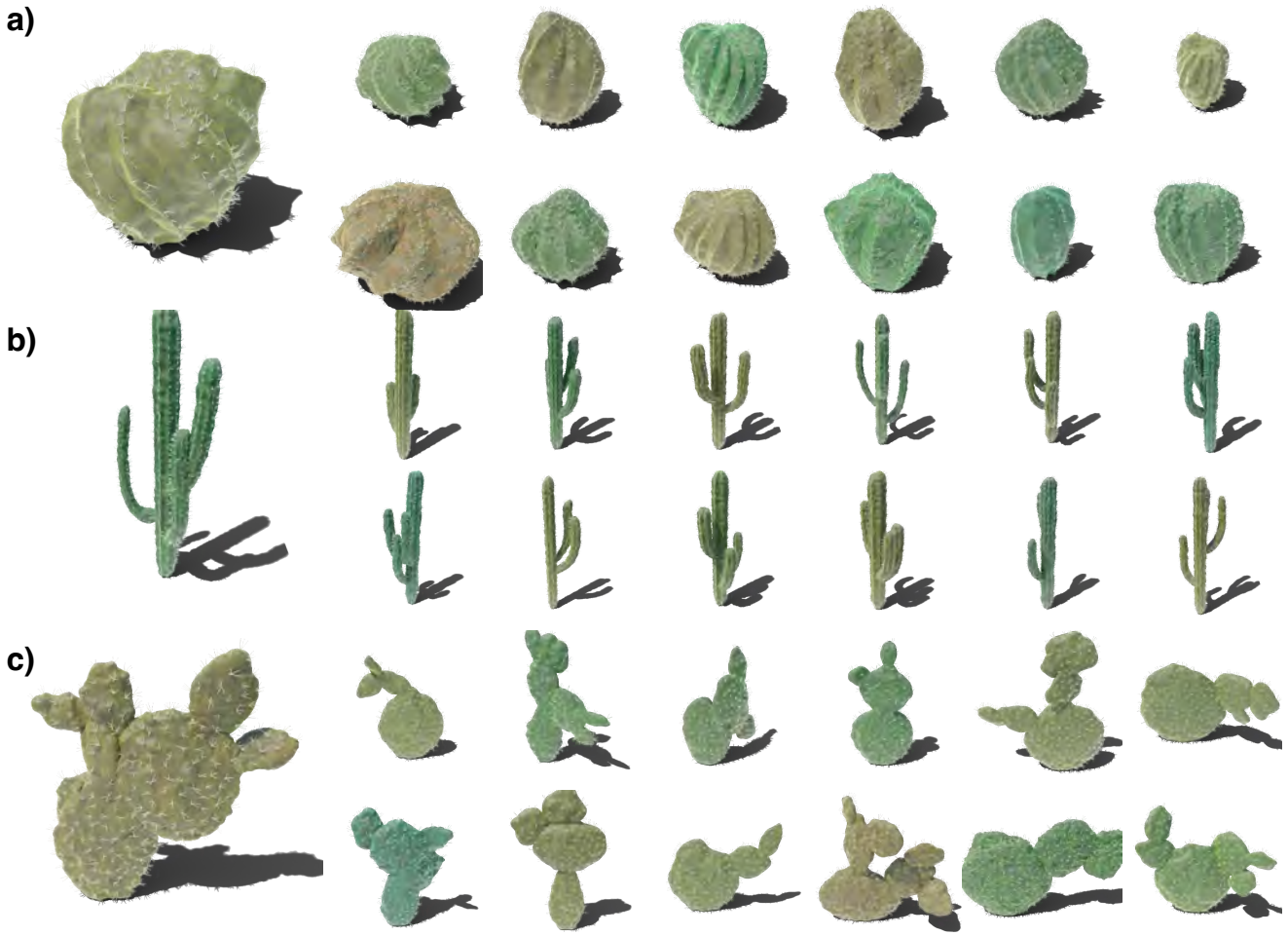


Figure 16. All classes of cacti included in Infinigen. Each row contains one class of cactus: **a)** Globular Cactus; **b)** Columnar Cactus; **c)** Pricky pear Cactus.

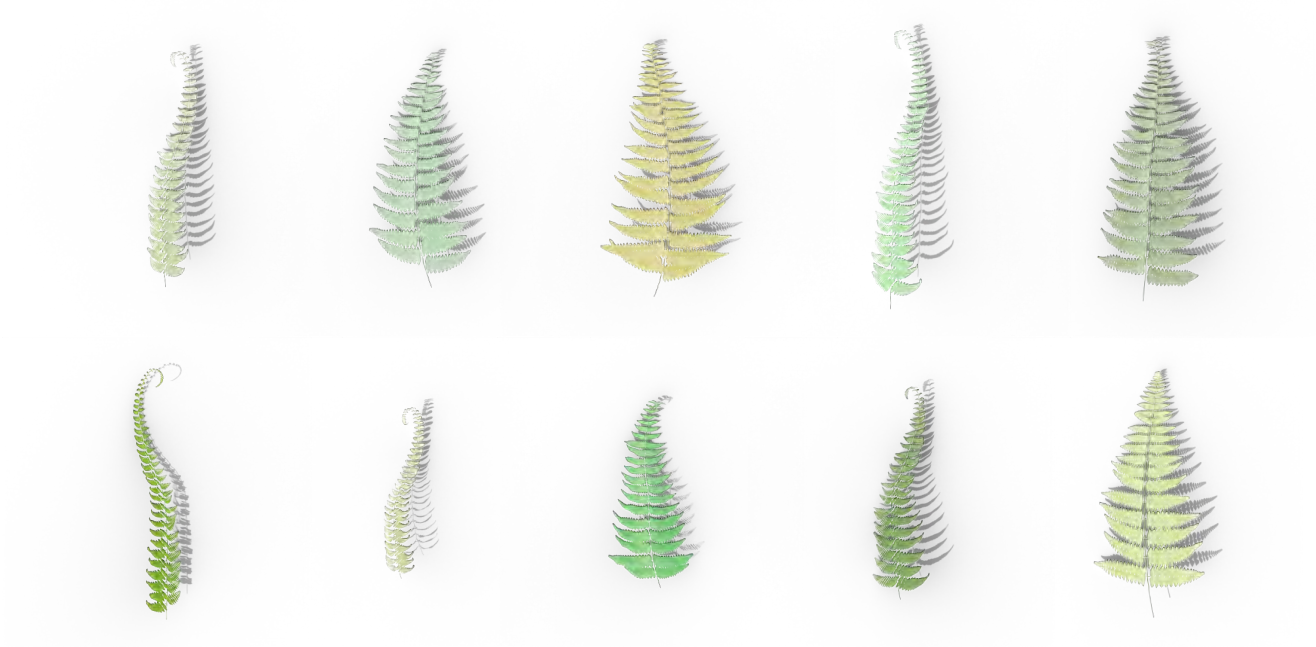


Figure 17. Assets of fern pinnae included in Infinigen. A fern consists of a random number of pinnae in the same color with random orientations.

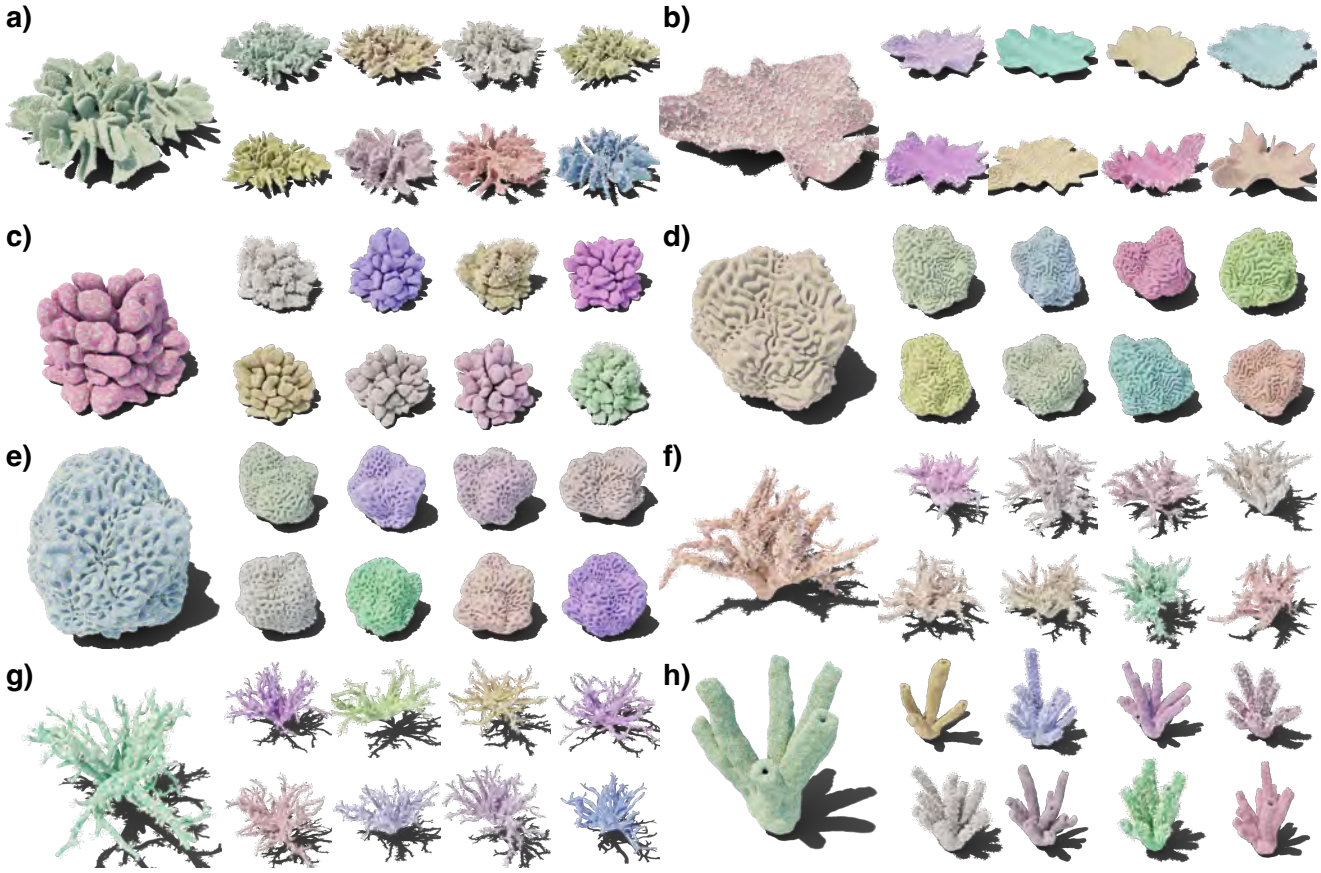


Figure 18. All classes of corals included in Infinigen. Each block contains one class of coral: **a)** Leather Coral; **b)** Table Coral; **c)** Cauliflower Coral; **d)** Brain Coral; **e)** Honeycomb Coral; **f)** Bush Coral; **g)** Twig Coral; **h)** Tube Coral.

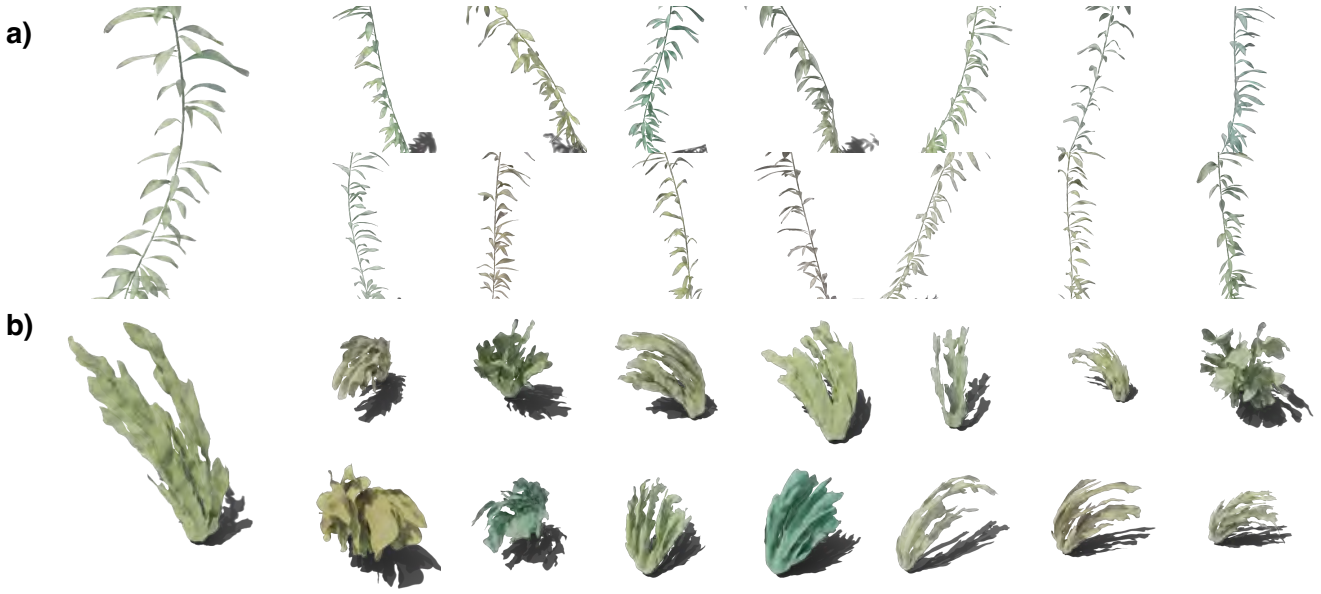


Figure 19. Kelps **a)** and seaweeds **b)** examples, each occupying one row.

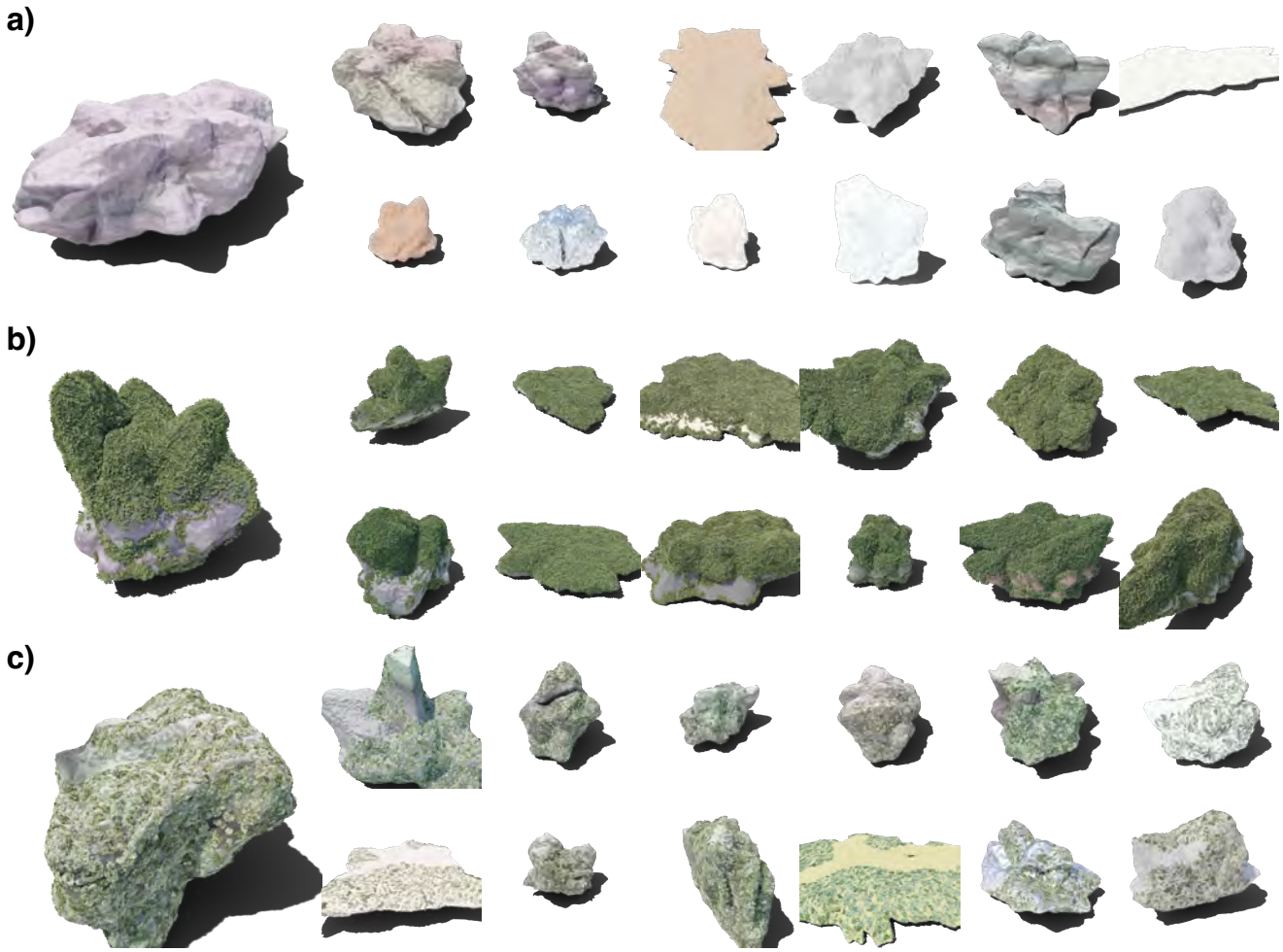


Figure 20. Boulder assets with different rock cover surfaces. In particular, each row of boulders are under **a)** no surface; **b)** moss surface; **c)** lichen surface.

Figure 21. Mushrooms **a)** and pinecones **b)**.

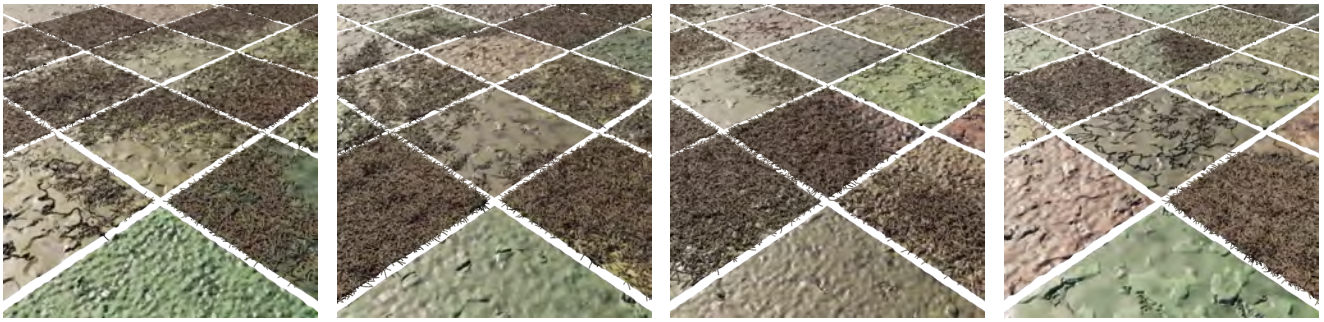


Figure 22. Pine needles scattered onto the ground with varying density.

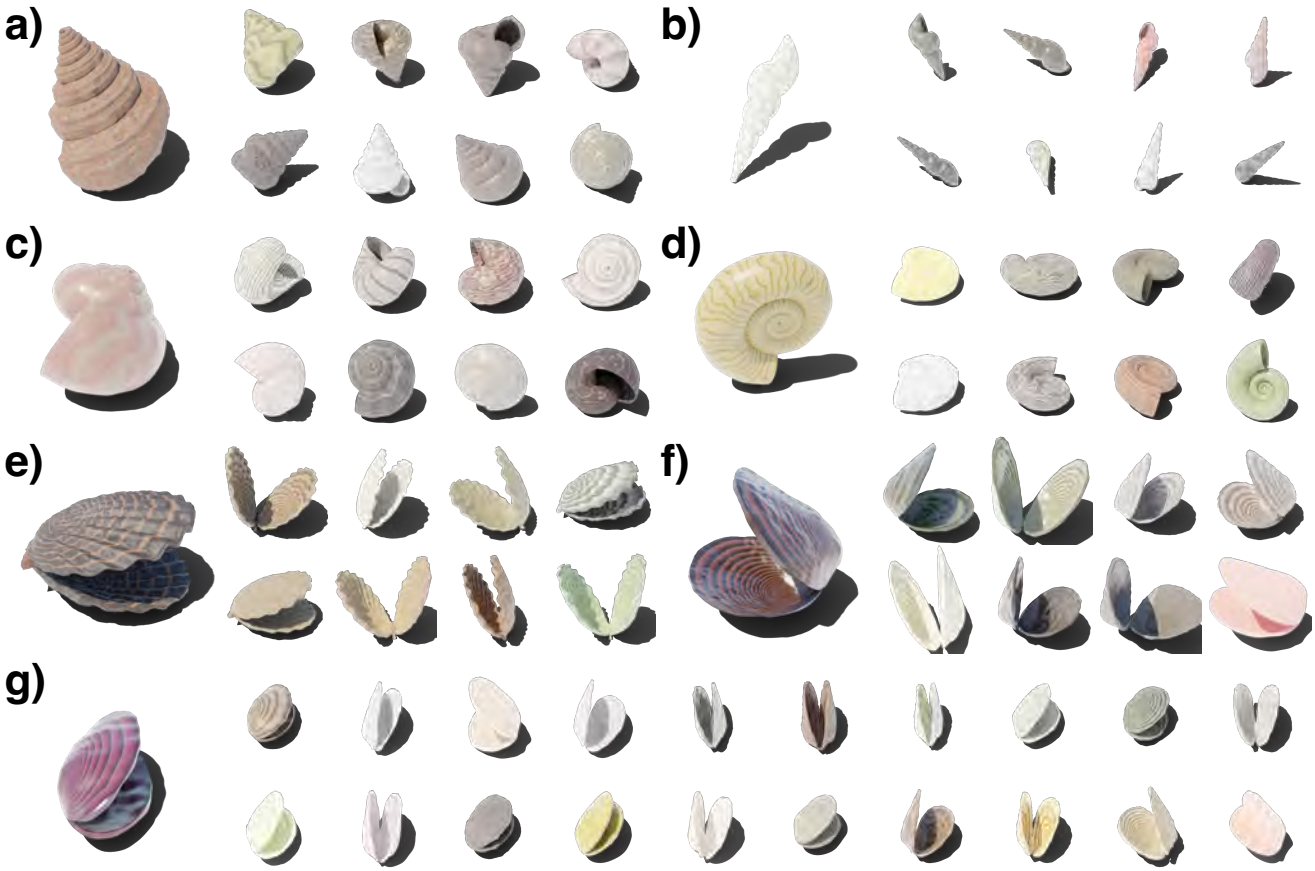


Figure 23. Different classes of mollusks, with each block representing a class of mollusk: **a)** Conch, **b)** Auger, **c)** Volute, **d)** Nautilus, **e)** Scallop, **f)** Clam, **g)** Mussel.

Figure 24. 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 3. 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 4. 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 5. 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 6. 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 7. 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 8. 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 9. 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 10. 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. 4, and specifies a realistic distribution of other assets to create a fully realistic natural environment.

References

[1] Zed 2. https://www.stereolabs.com/zed-2/.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] Peter Gray and Stephen K. Scott. Chemical oscillations and instabilities: Non-linear chemical kinetics. 1990.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] Nick McDonald. Soilmachine. https://github.com/weigert/SoilMachine, 2022.

[20] 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.

[21] Boris Okunskiy. Introducing differential growth addon for blender. https://boris.okunskiy.name/posts/blender-differential-growth.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.