Inverse Procedural Modeling: From Sketches to Buildings

Shuyuan Zhang

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

Procedural modeling have been widely used in the industry to generate a wide variety of 3D models both rapidly and in high quality, but creating the procedural models themselves have become a burden, requiring long-time effort of dedicated artists and programmers. Inverse procedural modeling aims to solve this problem leveraging the power of artificial intelligence to automatically infer the shape parameters of a procedural shape program. Many work had shown satisfactory results on the creation of buildings, chairs and other domains through computer graphics and computer vision techniques.

Inverse procedural modeling on buildings from sketches usually considers each part of the shape individually and obtains parameters for each component in a step-by-step fashion, and this work aims to consider the intended building shape as a whole, employing a series of decoder networks to obtain corresponding shape parameters for each part. A Directed-Acyclic-Graph (DAG) based procedural shape program, utilising Blender's Geometry Node functionalities, is constructed to create various shapes of buildings. A neural network is then trained on the data generated by this shape program and it predicts shape parameters from human sketches. As a final result, a Blender plugin is implemented as an user interface to facilitate intuitive interaction on the system, also for visualization purposes.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 800289
Date when approval was obtained: 2023-12-21

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Shuyuan Zhang*)

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Changjian Li, for the invaluable guidance along the way, and for guiding me into this field of research. I am lucky to have the chance to work on something I wholeheartedly enjoy and believe to be meaningful and helpful for the greater community and society as a whole.

I would also like to appreciate the University of Edinburgh, which provided a platform for me to enrich myself in this four years' study. I will never forget Appleton Tower, and the be-known AT level 9 study suits. I am deeply grateful for all of the professors, lecturers and staff's contributions to my education.

This appreciation extends to the city of Edinburgh, where I experienced some of the most enriching and treasured moments of my life. I will never forget the sunshine and morning fog of the Meadows, and I would also greatly recommend looking out of the window from AT level 9, enjoying the best of Edinburgh city views.

Special thanks to my family for their support and encouragement throughout this endeavor, Their belief in me has been a constant source of strength and motivation. I am also grateful to my friends whose companionship and positivity have helped me navigate through the challenges with resilience and determination.

I hope my work will benefit independent game developers in the future as an alternative way of creating 3D models.

Blender is a great piece of software.

Table of Contents

Chapter 1

Introduction

1.1 Background

The means of producing 3D models have been significantly expanded during the past few decades, and there exists in market various mature software to serve both professionals' and casual users' needs, such as AutoCAD and Blender. However, the learning curve for these tools are still steep and creating aesthetic and high quality mesh is still a field of profession requiring long-time efforts. Procedural modeling, as a solution, is widely employed as it can efficiently generate production-ready 3D models. With a properly-designed procedural model, cross-disciplinary users can obtain a variety of 3D models and are guaranteed about the quality of the final mesh.

There have been effort in procedurally generating shapes for terrain (Argudo et al. (2017), Ariyan and Mould (2015), Bradbury et al. (2014), Cordonnier et al. (2017)), vegetation in varying level of details (Longay et al. (2012), Pirk et al. (2012)), buildings with proper semantic meanings (Finkenzeller and Bender (2008)) or even an entire city in the form of land uses (Lechner et al. (2006)) or a road network (Chen et al. (2008)). More efforts can be found in the survey by (Galin et al. (2019)) in Terrain modelling and by (Smelik et al. (2014)) which covers many more aspects involved in creating a virtual world.

After proving the feasibility of this approach and seeing procedural modeling's wide use cases, it does not solve the entire problem: the creation of these procedural models have now become the new bottleneck to the efficiency of the 3D modeling workflow. It is similar to the phenomenon that modern software can solve numerous tasks but software engineering has become a new discipline. There is also the dilemma of diversity and complexity: in designing a procedural model, more parameters must be exposed to the user to provide a wider range of choice over the shape, but this yields a more complex procedural model and also a steeper learning curve for the end users when they try to understand all the options available and explore their individual ranges.

Inverse Procedural Modeling (IPM) aims to solve the above problems by leveraging advances in machine learning and artificial intelligence. Through employing Computer Graphics and Computer Vision techniques, IPM proposes to automatically find procedural models from user inputs. There are two aspects in terms of finding a procedural model: generating grammars and shape programs that can be constructed as a procedural model which can resemble user inputs solves the problem of creating procedural models from scratch, and predicting a set of parameters for an existing shape program solves the dilemma through removing the need for users to fill in the parameters.

1.2 Project Aims

This work aims to tackle an classic topic in sketch-based inverse procedural modeling, buildings, from the second aspect, predicting shape parameters. Sketches are chosen as the form of user input since it is one of the most natural ways of human expression, and they intrinsically carry direct visual features. This is a well-explored task and there exists various prior work with mature solutions. For example, Nishida et al. (2016) developed an iterative process in which users provide drawings of building components step-by-step. Leveraging Pearl et al. (2022)'s work, we can implement a multi-decoder system that accepts one full sketch of the shape and removes the need for multiple inputs on the building domain as well.

We summarize our work in three aspects: shape program and dataset generation, neural network, and an integrated user interface:

- a Directed-Acyclic-Graph (DAG) based shape program that functions as a procedural model for buildings, which we also used to generate a synthetic dataset; some illustrations on the shape outputs of this shape program is provided in Figure 1.1; our shape program can also be used along with a novel renderer pipeline that can create hand-drawing style sketches from a shape;
- an Encoder-decoder neural network with a series of multi-task decoders that accepts a sketch of a building and predicts corresponding shape parameters for the above shape program, and when the program is fed the predicted parameters, it produces a 3D model that resembles the sketch; this also proves the capabilities, also affected, of decoder networks to capture subtle differences and complex details;
- a User Interface that allows users to draw sketches and send to the neural network for shape parameter prediction, and then automatically loads the predicted parameter into the shape program, visualizing the resembled 3D model.

Blender is a suitable open-source software that can serve multiple purposes in our work: its Geometry Node system was used along with a third-party library, Geometry Scripts, to construct our shape program; its Freestyle module was also used to render 3D shapes as sketches, which was a crucial component for synthetic data generation; it also provides functionalities on customizing a user interface, which we used to build our own interface integration. Although this work is highly related to this software, we should note that the pipelines can generalize to any other implementations as well.

Our source code is available on GitHub.

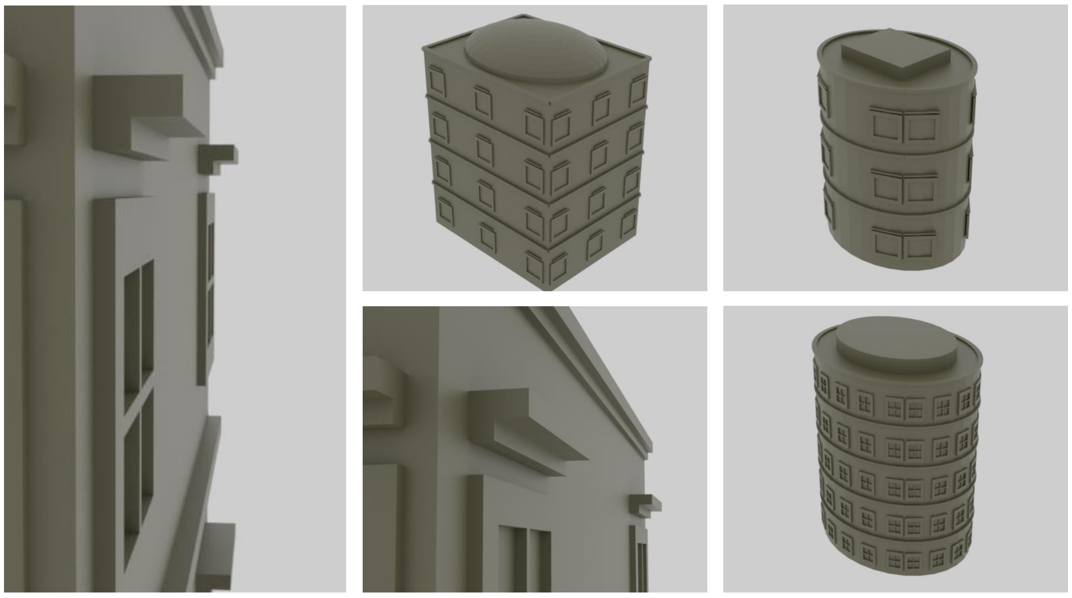


Figure 1.1: Illustrations on our shape program's outputs

1.3 Terminology

We use some abbreviations and terms through out this work, and to avoid any confusions, we summarize them here.

- **PM**: Procedural Modeling
- **IPM**: Inversed Procedural Modeling
- **DAG**: Directed Acyclic Graph
- **Shape Program**: a procedural program that accepts and interprets a set of input parameters describing the desired properties of a 3D model, and produces the corresponding 3D mesh.
- **Parameter(s)**: if unspecified, it refers to shape parameters, for example number of floors or if window panels are divided horizontally.
- **Ledges**: "window ledge" refers to the facade element that is usually part of a window shape, which is placed on top of a window for aesthetics or functionality like stopping raindrops; "floor ledge", taken out of our context usually refers to the shape element that connects the floor and a wall, however we use it to refer to the facade element that visually divide building mass and roof, or divide different floors on the exterior of a building - this is to distinguish from window ledges. If no prefix is given, "ledge" refers to the floor ledges.
- **Batch**: it usually refers to the batch size during neural network training, but as will be discussed in section 3.2.1, in the context of datasets, it refers to batches of images we generate as synthetic training data.
- **Synthetic / Distortion Dataset**: by synthetic we refer to the dataset of images

rendered by Blender's Freestyle module, and by distortion we mean the attempt to create distorted curves to mimic hand sketches during dataset creation and rendering. See section 3.2.2 for details.

In addition, we wrap each node, especially custom nodes, in our DAG-based shape program as math equations, such as $BuildingMass$. For shape parameters, they will be referred to in italics, like "*Window Panel Area*".

Chapter 2

Related Work

As mentioned previously, both Procedural Modeling and the newer Inverse Procedural Modeling fields are well-explored, thus there exists various work with different focusing points. Below sections will mention some representative ones and will focus on introducing two approaches which our work primarily builds upon.

2.1 Procedural Modeling

Procedural Modeling has been intensively used in different industries, of which the game industry is a major user of related techniques, and on various domains, from daily objects to larger systems like cities or planets. A recent example was No Man's Sky, a video game launched in 2016, which demonstrated how extensive use of procedural modeling on alien creatures, planets and stellar systems can lead to an unique and immersive experience (Tait and Nelson, 2022). Procedural modeling, however, did take a long term effort to become what it is nowadays, and both artificial objects and non-urban natural objects were points of interests for many research work. From as early as 2001, Parish and Müller (2001) developed a procedural shape program with L-systems to model cities, a complex urban system consisting of road networks and buildings. From a range of input images, such as population density, land-water boundary and height map, the system first uses an extended L-system to construct the road map in the form of a graph, which is then divided into building lots represented with polygons. With these building lot polygons as a base, another L-system generates building instances as a set of strings, which is then fed into a geometry parser to produce final building shapes. Müller et al. (2006) extended this path through introducing a more detailed CGA grammar on procedural building modeling. The grammar defines first an anchor point and a bounding box in space for the building shape, then deals with facade splits, occlusion, different shape primitives and snapping them altogether to form a final and optionally complex building model. To extend on the other aspect, Chen et al. (2008) proposed to connect the graph representing the road map with tensor fields, which allowed users to edit existing road network with a set of interactive tools. Above work is summarized in Figure 2.1, with visualizations from their respective paper.

Procedural modeling can also be used to generate non-artificial objects, such as terrains

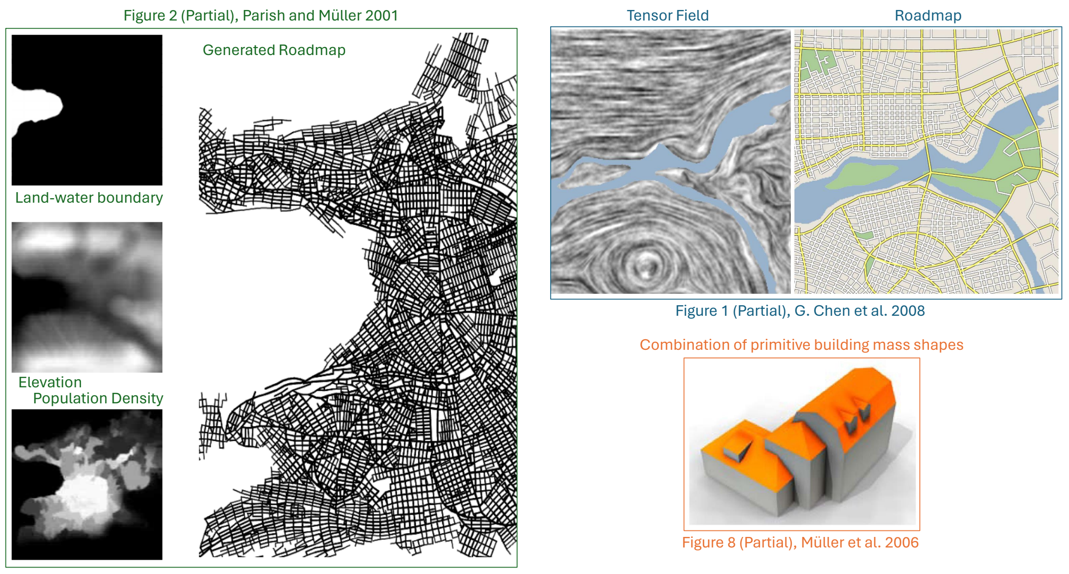


Figure 2 (Partial), Parish and Müller 2001
Generated Roadmap
Land-water boundary
Elevation
Population Density
Tensor Field
Roadmap
Figure 1 (Partial), G. Chen et al. 2008
Combination of primitive building mass shapes
Figure 8 (Partial), Müller et al. 2006

Figure 2.1: Left: cropped figure 2 from Parish and Müller (2001), demonstrating their inputs (land-water boundary, eleveation/height map and population density) and a possible roadmap generated by their L-system program; Upper Right: part of figure 1 from Chen et al. (2008), demonstrating a tensor field and corresponding roadmap generated after some iterations of user edits; Bottom Right: part of figure 8 from Müller et al. (2006), showing a building mass consisting of multiple shape primitives, built with CGA grammar.

and trees. Argudo et al. (2017) proposed an example-based approach to generate high-resolution terrain from a range of available inputs, including elevation map, vegetation distribution and drainage area. L-system is again useful, and as a result usually involved, in modeling natural objects like trees. Ijiri et al. (2006) designed a system which introduces more user control over L-system based tree modeling. More recently, Xu and Mould (2015) tackled the same task with a graph-based representation, in which trees are represented as a set of least-cost paths in a graph, and with guiding vectors assigned to each node in the graph to guide the "growth" of branches. Some of the results and visualizations are provided in Figure 2.2.

2.2 Inverse Procedural Modeling

As mentioned previously, inverse procedural modeling aims to solve two problems: the automatic creation of procedural models, and the automatic use (filling in parameters or fine-tuning parameters) of procedural models. On the first topic, L-system is again popular due to its convenient manipulability facilitated by its string representation. Št'ava et al. (2010) proposed a method which takes a 2D vector image, usually representing a tree's branches in 2D space, as input, outputs a parametric context-free L-system. It starts with recognizing similar elements in the input image, such as curves, poly-lines, and assigning them appropriate terminals to form an alphabet for the L-system; it then iteratively extracts L-system rules by comparing terminal symbols' positions and orientations in a 4D transformation space. On the other aspect of IPM, which is also more

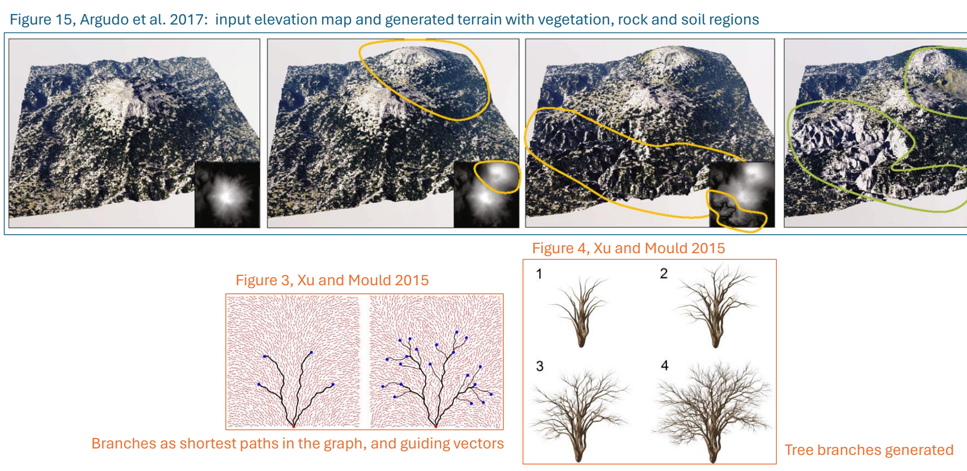


Figure 15, Argudo et al. 2017: input elevation map and generated terrain with vegetation, rock and soil regions
Figure 4, Xu and Mould 2015
Figure 3, Xu and Mould 2015
1
2
3
4
Branches as shortest paths in the graph, and guiding vectors
Tree branches generated

Figure 2.2: Upper: figure 15 from Argudo et al. (2017) showing corresponding input elevation map and generated terrain with high resolution of details; Lower: figure 3 and 4 from Xu and Mould (2015) visualizing the graph and tree branches as shortest paths, and the guiding vectors assigned to each node in the graph, as well as generated tree branches.

related to our context, Vanegas et al. (2012) proposed a framework which handles the basic interactions with an existing procedural model on cities for the user, introducing an alternative method for controlling high-level features instead. It automatically predicts suitable changes to the current procedural model through Markov chain Monte Carlo parameter searching to meet the desired indicator values. Some examples of feasible indicators include average sunlight exposure and average distance to the closest park, and an illustration with average shadowing value on the road/buildings as an indicator function is referenced in Figure 2.3.

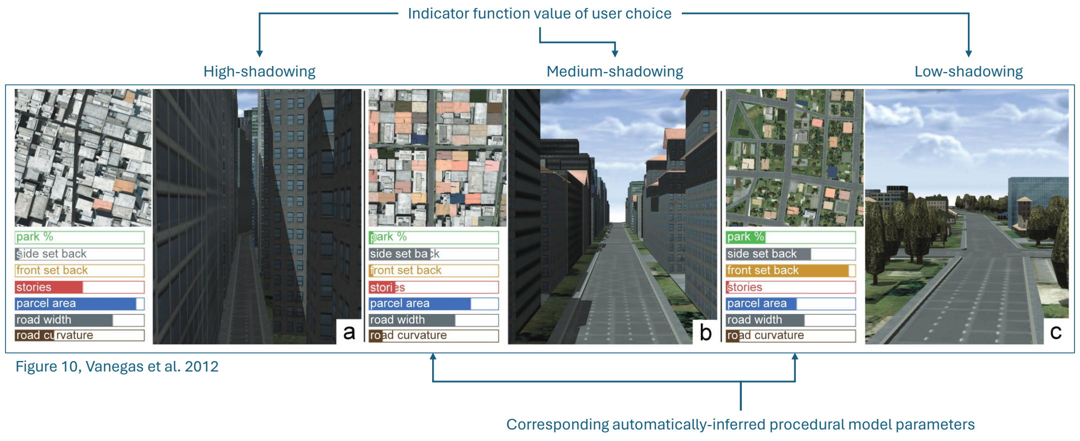

Indicator function value of user choice
High-shadowing
Medium-shadowing
Low-shadowing
park %
side set back
front set back
stories
parcel area
road width
road curvature
a
b
c
Figure 10, Vanegas et al. 2012
Corresponding automatically-inferred procedural model parameters

Figure 2.3: Figure 10 from Vanegas et al. (2012), showing how user-desired values are interpreted by the system and reflected on parameter changes in the base procedural model.

2.3 IPM on Buildings

There exists various work on inverse procedural modeling of buildings and urban objects, such as Mathias et al. (2011)'s grammar-based building reconstruction and Martin and Patow (2019)'s ruleset-rewriting approach on modifying existing procedural building models. Our work focus on sketch-based inverse procedural modeling of buildings and Nishida et al. (2016)'s interactive sketching method is the most relevant.

Nishida et al. (2016)'s work used inverse procedural modeling on the buildings domain. To build the shape program, a set of shape priors and primitives of different components of a building, for example different types of building masses and types of roofs, were pre-defined as grammar snippets, with adjustable parameters controlling the procedural shape of each, and these can be then combined into a full procedural model for buildings. A collection of their shape primitives are shown in Figure 2.4.

To create the synthetic dataset for training, different combinations of shape parameters were created through sampling the parameter space of each grammar snippet, and a geometric engine was employed to produce 3D shapes from these configurations. In terms of neural network, a convolution neural network was used first to classify input sketches to one of the grammar snippet that would be the most suitable for reconstructing the shape, and then another convolution neural network would carry out regression on the parameters of the chosen grammar snippet, aiming to minimize the loss between predicted shape parameters and ground truth values. On the user side, this process assumed that the user would start with drawing the building mass, and then provide sketches of subsequent components like roofs and windows in an iterative process. The full sketching process is also shown in Figure 2.4.

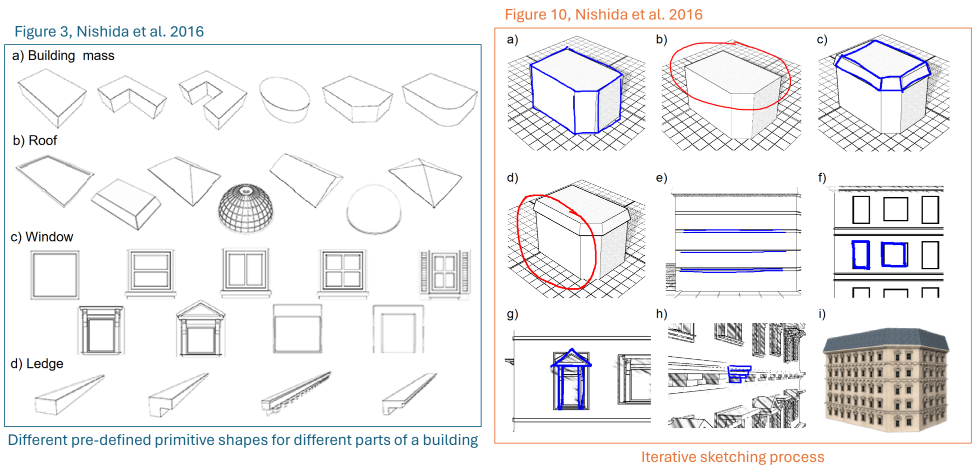


Figure 3, Nishida et al. 2016
a) Building mass
b) Roof
c) Window
d) Ledge
Different pre-defined primitive shapes for different parts of a building
Figure 10, Nishida et al. 2016
a)
b)
c)
d)
e)
f)
g)
h)
i)
Iterative sketching process

Figure 2.4: Left: figure 3 from Nishida et al. (2016) showing shape primitives defined as grammar snippets that can be combined together to form a building; Right: figure 10 from Nishida et al. (2016) again depicting an example sketching process - blue lines are user sketches, and red lines refer to a lasso tool for the user to specify the next basis plane for drawing.

Major advantage of their process is robustness, since no matter how distorted the given sketches are, the system will only fallback to one of its predefined types and shapes, which guarantees the quality of the output mesh. Another advantage is the simplicity of user-provided sketches due to their iterative process. The system expects sketches of one single component at each step, which lowers the requirements on both ends: the neural network is easier to train since its input and outputs are simpler than a detailed and complex sketch-parameters pair, and it is also easier for users to supply partial drawings instead of one full detailed sketch. It also expects fixed viewpoints for different component, for example three-quarter views for building mass and orthogonal projection from y-axis for windows and facades, which further lowers the burden on neural network training, however, puts forward a new challenge for users to provide drawings from a certain perspective.

The classification and regression CNNs would also be hard to scale to more diverse types of shapes, since the system needs one classification network per shape components (building mass, roof, etc.) and one regression network per component type (cubic roof, round roof, etc.).

As a follow-up on this work, two years later, Nishida et al. (2018) extended the above process by inferring the building shape from one single image shot by cameras instead, which contains information about camera angles, silhouette of the building mass, facade layout and texture. Their approach again consists of several convolution neural networks to predict shape parameters and optimization techniques to speed up the search through parameter space. Note that this work is a major extension on single image inference, but our objective is still based on sketches and the use of multiple decoder networks instead.

2.4 IPM with Encoder-Decoder Architecture

Proposed by Sutskever et al. (2014), the encoder-decoder architecture was first used for sequence-to-sequence modeling and on the machine translation task, and the idea was soon recognized as applicable to many other domains and widely adapted in various fields, such as SegNet (Badrinarayanan et al., 2017) and DeepLabv3+ (Chen et al., 2018) on Computer Vision and Graphics.

Pearl et al. (2022)'s work adapted this architecture on inverse procedural modeling, in the domain of daily objects including chairs, tables and vases, with multiple networks as the decoder. Their shape programs are in the form of Directed-Acyclic-Graphs, which implementation-wise is a Geometry Node Tree in Blender that provide a set of controllable parameters for the user to edit shape details on the produced mesh. By propagating user inputs through the nodes, the DAG program can ensure proper positioning of different shape elements. To facilitate understanding of this concept, an example DAG program is constructed with Blender Geometry Nodes and shown in Figure 2.5.

In terms of neural networks, in their work, an encoder network containing several convolutional neural networks accepts point clouds or sketch images as input and produces an embedding vector. Subsequently, a series of decoder networks consisting of dense layers will predict corresponding shape parameter values from the feature embedding. Intuitively, a forward pass of one decoder network is equivalent to one step

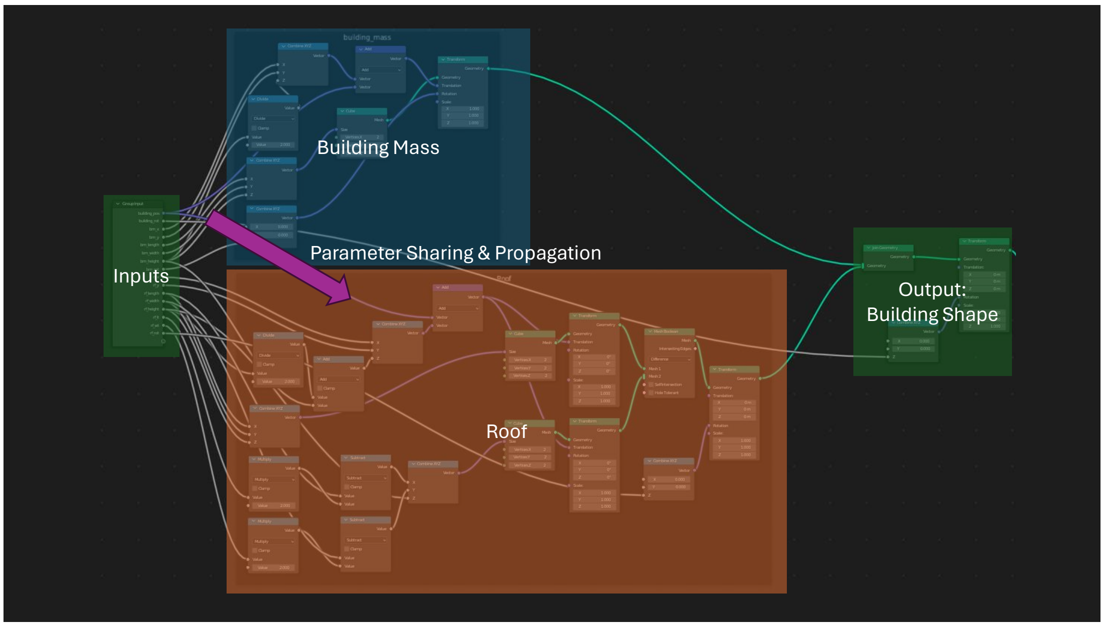


Figure 2.5: An exapmle DAG program. Note how parameters are propagated to where needed, for instance building mass's height is used by the roof component to determine its z-axis position.

in the iterative process in the aforementioned Nishida et al. (2016)'s work, as in, finding appropriate values for one or more parameters that the shape program accepts. Their pipeline and a partial segment of their DAG program in Blender is provided in Figure 2.6.

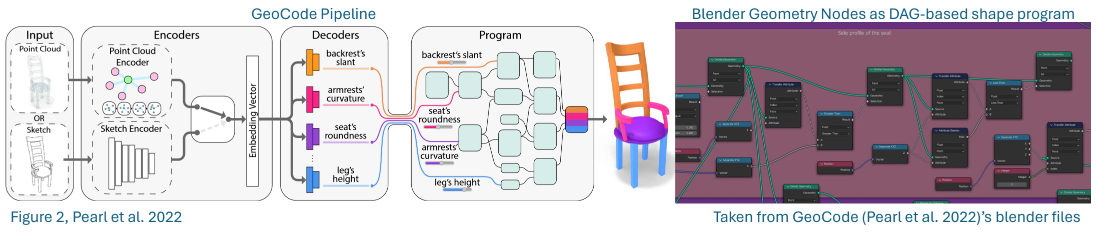


Figure 2.6: Left: figure 2 from Pearl et al. (2022) demonstrating their pipeline; right: part of Pearl et al. (2022)'s DAG program in Blender.

Pearl et al. (2022)'s work achieved single image inference from sketches through employing a multi-decoder architecture, however, their solution has limitations. Their process cannot be easily generalized to any shape programs, since the decoder configuration depends on the input parameters of the shape program in terms of quantity and data types. The quality of output shapes also depend heavily on the program itself, with little intervention from the neural network's capabilities. Finally, demonstrated domains - chairs, tables and vases - contain features that are clearly visible and can be captured effectively, such as the shape of the seat or the curvature of the legs. In terms of buildings, some features might be not as visible when one single sketch of the whole building shape is fed in, such as details of window frames or shape of ledges separating each floor. We then seek to experiment on the effectiveness of such an approach when we migrate the domain to buildings. We also seek to extend this architecture by employing a multi-task decoder architecture, which is responsible for predicting parameter values of one semantic component instead of one single parameter, and our approach will be discussed in section 3.3.2.

Chapter 3

Methods

In this chapter we discuss the major methods of our work, including the design and construction of our shape program on buildings, how we generated the synthetic dataset for training, our neural network architecture and the Blender User Interface.

As an overview, we first defined a shape program that accepts a set of input parameters and produces a 3D model for a building; we then created a synthetic dataset through sampling our parameter space and rendering shapes produced by the shape program as sketch images; our neural network consists of an encoder and a series of decoders, which will be trained to predict parameters for a given input sketch; our user interface allows users to draw sketches and send them to neural network for inference, and it automatically loads the output parameters into the shape program to visualize the procedural building predicted based on user sketch. This overall process is visualized in Figure 3.1.

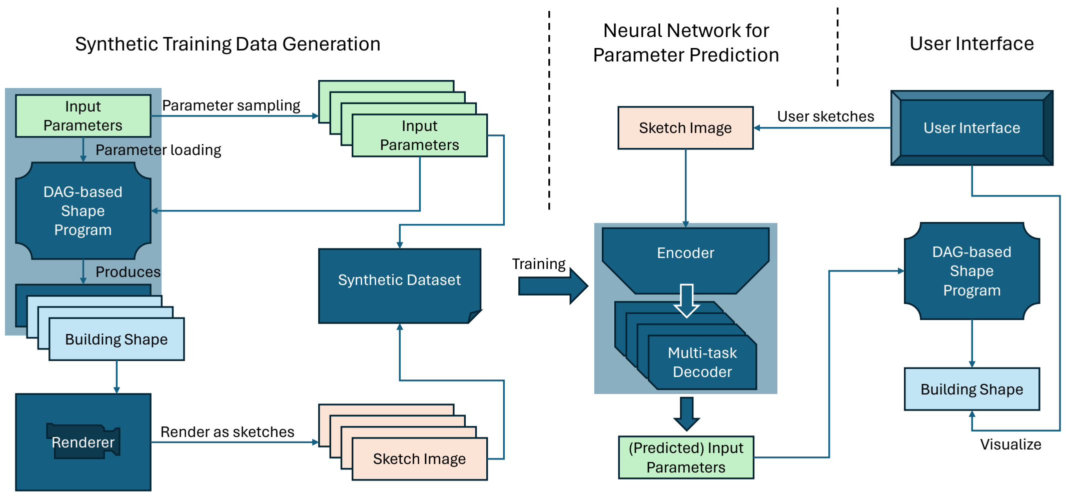


Figure 3.1: Overall process of our work, from how we generate training data to how the trained neural network is used.

3.1 Shape Program

We start with introducing our shape program on buildings. In short, it is a DAG-based procedural model written in Python and executed in Blender, and it produces raw mesh of a building.

3.1.1 Buildings

We consider a typical building to be consisting of the following components: building mass, roof, windows and ledges. This design inherits from Nishida et al. (2016)'s work, but consider a window shape as a combination of window frames, window panels and window ledge. In terms of ledges, it refers to the elements on a building facade that visually divide different floors. The building decomposition is visualized in Figure 3.2.

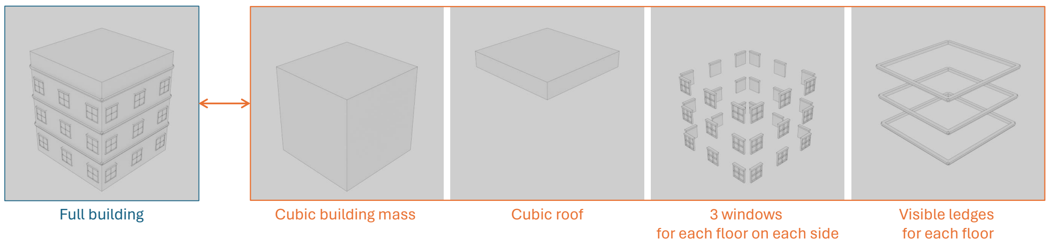


Figure 3.2: An example of a full building mesh with cubic building mass and roof, 3 floors and 3 windows on each side for each floor, with visible ledges dividing each floor.

The full design and individual components of a building in our context are shown in Figure 3.3. All other components other than building mass are connected the building mass, for example, roof is positioned on top of the building mass and facade elements including windows and ledges are attached on building mass surface. Each individual component (building mass, roof, window, ledge) has different variations in their shape, either discretely (different *type* of shapes like cubic or round) or continuously (*sizes*, *extrusion* into the shape for details, etc.). The height of building mass, along with number of floors, dynamically determines floor height; floor height and number of windows per floor on each side then determines maximum window sizes. These kinds of connection and parameter-dependencies are further discussed in section 3.1.3. This is a rather simple design comparing to what state-of-the-art procedural models on buildings can achieve (such as Schwarz and Müller (2015), and see section 2.1 or any application in game or visual effects industry), since implementing a more complex and realistic building shape program is not the major focus of our work.

We simplify shapes as follows: building mass can be cubic or cylindrical; roof can be cubic, cylindrical or a half-sphere; window panels can be divided horizontally, vertically or not divided; window ledges and floor ledges are a cubic shape with optional extrusion into the shape on x- and z-axis, resulting in an L-shape if extruded. These are visualized in Figure 3.4.

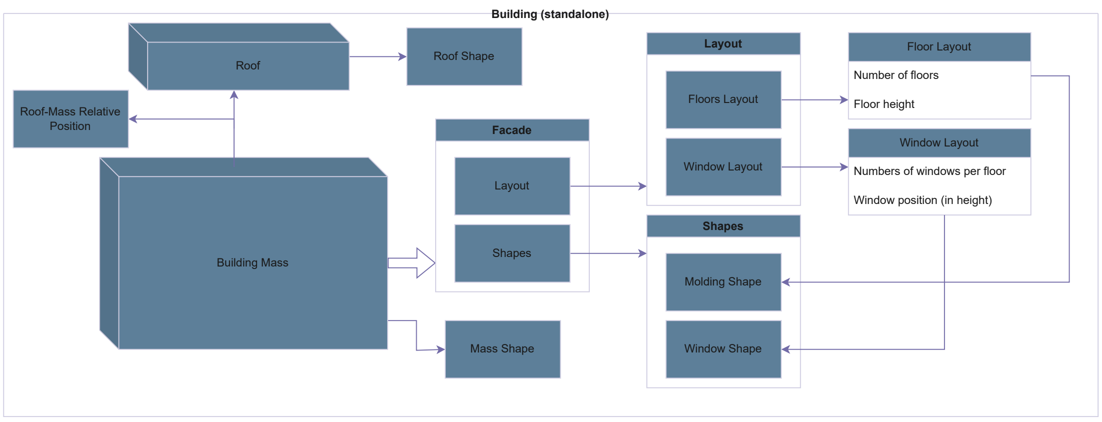


Figure 3.3: Building design diagram. Note the ubiquitous dependencies between components.

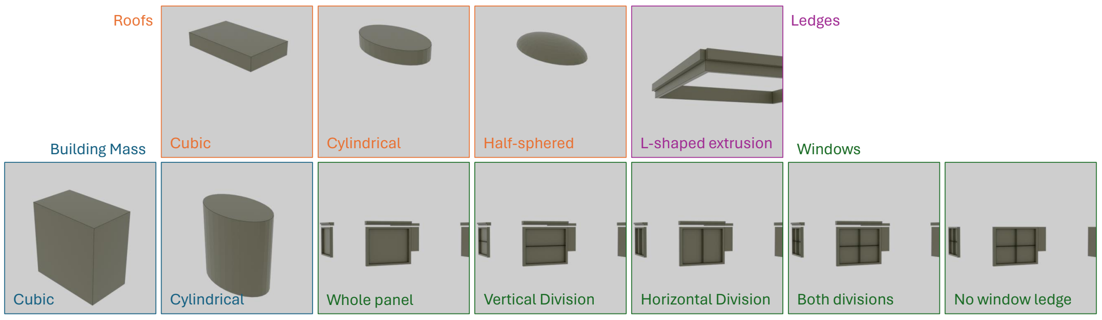


Figure 3.4: Different types of shape components supported by our DAG program for buildings. Note that continuous variations, such as size differences, are not visualized here.

3.1.2 Geometry Nodes and Geometry Scripts

Blender's Geometry Nodes is a node-based system that can handle many geometry operations in Blender, and it can be used to construct our DAG-based shape program. We refer readers to the official manual page for full introduction and to a book written by Lotter (2022) for more advanced usage guidance and examples. One drawback we noticed from Pearl et al. (2022)'s work on DAG programs is that such a node-based or graph-based system, although intuitive, cannot scale to larger systems that can express more complex shapes without sacrificing readability. The complexity of Pearl et al. (2022)'s shape program is visualized in Figure 3.5.

To solve this issue, a third-party library, Geometry Scripts, available on GitHub under the GPL-3.0 license, is used. It is a scripting API for Blender's Geometry Nodes system that address the aforementioned complexity issue with Geometry Nodes by allowing users to create node trees from Python codes. We again refer readers to its official documentation for more details. However, it is not the case that arbitrary Python codes can be used: methods from the API must be used all the time in order for the conversion to node trees to work. For example, a simple if-else statement needs to be expressed as a series of comparison operations yielding booleans and then a set of switch nodes to

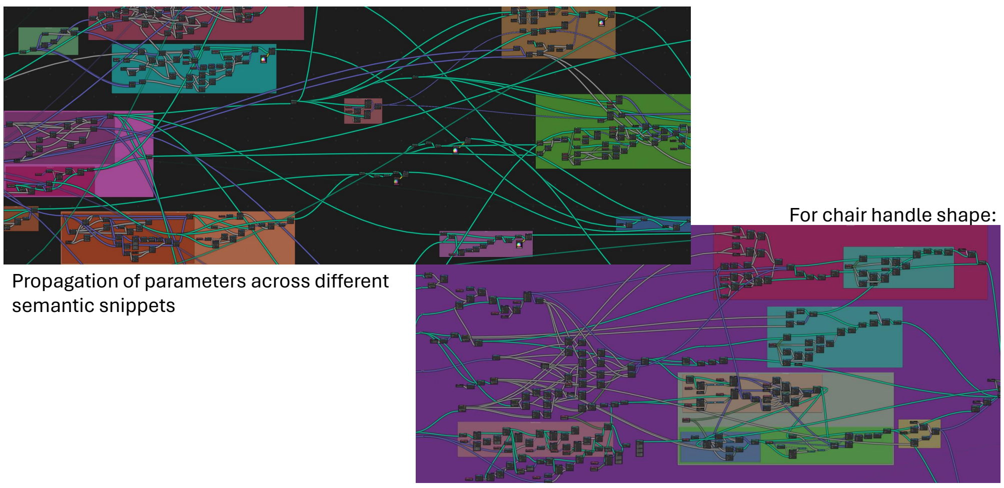
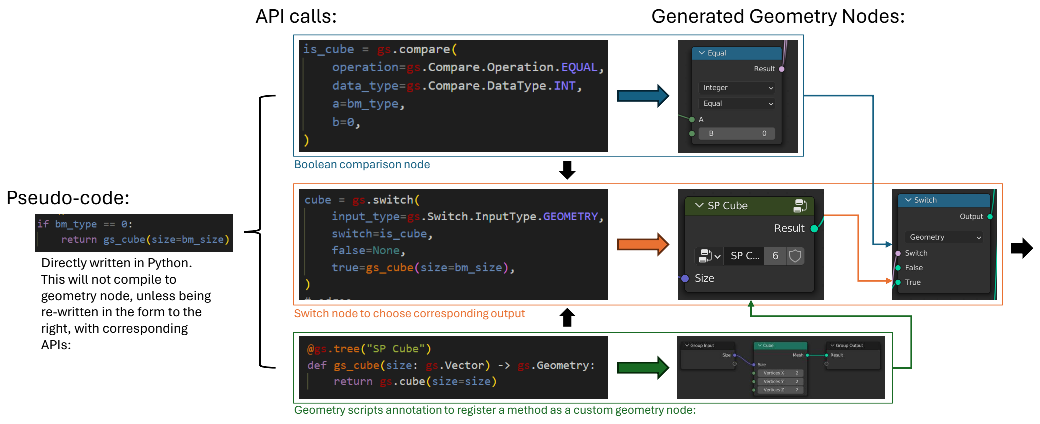

For chair handle shape:
Propagation of parameters across different semantic snippets

Figure 3.5: Blender Geometry Nodes built by Pearl et al. (2022) for a procedural chair. Note that these are partial screenshots since the system is larger than the maximum zoom-out level Blender's interface allows.

propagate corresponding values. This is illustrated in Figure 3.6. Another example is the use of for-loops. If we wish to iteratively instantiate a shape over a direction, instead of implementing the logic with for-loops, we need to define a curve that describes the direction and length, and to use a sampling node on the curve to create anchor points along the path, in the end instantiating mesh on all anchor points. This can be seen in section 3.1.3 and especially in Figure 3.8, on our DAG implementation.



```python
if bm_type == 0:
    return gs_cube(size=bm_size)
```

Directly written in Python. This will not compile to geometry node, unless being re-written in the form to the right, with corresponding APIs:

```python
is_cube = gs.compare(
    operation=gs.Compare.Operation.EQUAL,
    data_type=gs.Compare.DataType.INT,
    a=bm_type,
    b=0,
)
```

```python
cube = gs.switch(
    input_type=gs.Switch.InputType.GEOMETRY,
    switch=is_cube,
    false=None,
    true=gs_cube(size=bm_size),
)
```

```python
@gs.tree("SP Cube")
def gs_cube(size: gs.Vector) -> gs.Geometry:
    return gs.cube(size=size)
```

Figure 3.6: Usual logics cannot be written in Python directly, such as if-statements. They need to be expressed in the same way as if it is implemented in geometry nodes directly with built-in nodes. Green part also shows how to register a Python method as a custom node.

This also highlights a significant point, that the usage of this library does not lower the burden to build a complex DAG-based shape program, instead, it merely provides a more convenient way of organizing the shape program and enables features such as version control and convenient code commmenting.

3.1.3 Our DAG Program

We constructed our DAG-based shape program for buildings following the design described in section 3.1.1 and with the tools mentioned in section 3.1.2. The program starts by considering the overall building shape, and high-level parameters, once computed, will be propagated to subsequent components; after each individual component has determined on their respective detailed shapes based on these propagated values, their geometry output will be combined together as a final building mesh; one final step is in place to apply appropriate transformations on the final mesh, ensuring that the mesh is centered at world origin $(0,0,0)$ and will not exceed the viewport. This overall flow can be seen in Figure 3.7. We defined custom node trees for each individual components, and we used them in a final node tree that handles all high-level operations, such as the first and the last step mentioned above. These node trees are: $BasicBuilding$ for the overall building shape, $BuildingMass$, $Roof$, $BuildingWindowInstantiator$ with $WindowWithLedge$, and $BuildingFloorLedgesInstantiator$ with $CubedFloorLedgeForExtrusion$.

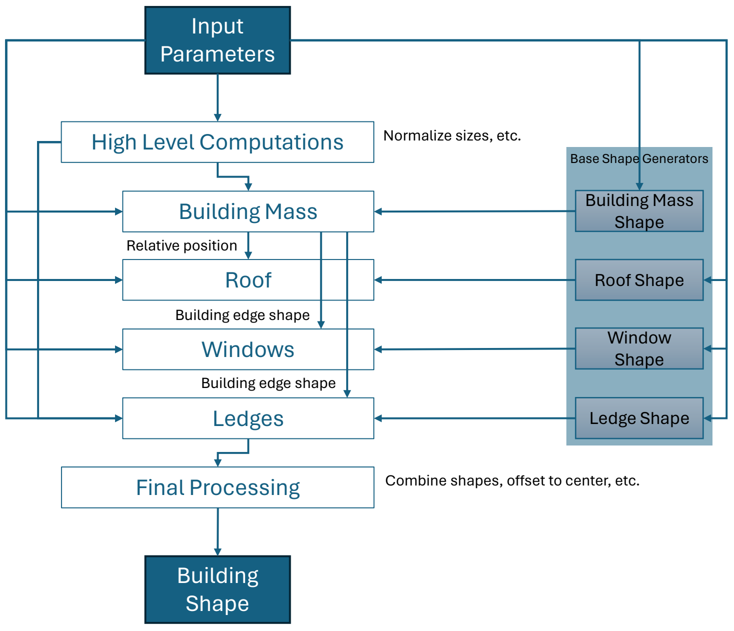


Figure 3.7: Overall flow for our DAG-based shape program.

The overall building shape is determined by user input on building mass size as a vector of 3 (size on x, y and z axis respectively), and we normalize the building size by making the diagonal length of building mass to 1. This calculation is shown in equation 3.1 and 3.2.

$$\mathbf{bm\_size}_{normalized} = \left[\frac{x}{||\mathbf{bm\_size}||}, \frac{y}{||\mathbf{bm\_size}||}, \frac{z}{||\mathbf{bm\_size}||} \right], \tag{3.1}$$

where

$$\mathbf{bm\_size} = [x,y,z], ||\mathbf{bm\_size}|| = \sqrt{x^2+y^2+z^2}. \tag{3.2}$$

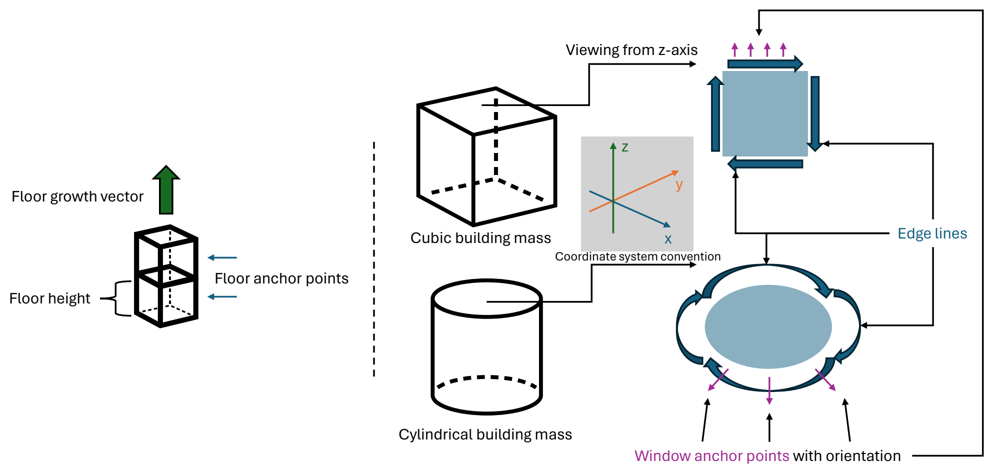


Figure 3.8: Left: building mass' floor growth vector, showing its usage of floor height (and number of floors), and how floor anchor points are centered on each floor; Right: visualizing edge lines for cubic and cylindrical building mass shape, and the window anchor points specifying window positions and orientations.

After obtaining the normalized building size, we divide it by number of floors to get a floor height. The floor height is an example of a computed high-level parameter, since it is inferred from other user inputs and is used in generating shapes for (all) other components: building mass, roof, windows and ledges.

BuildingMass node receives a vector containing pre-processed scale for one floor of building mass, and a discrete value (integer) indicating the type of the base shape, in this case 0 or 1, indicating a cubic or cylindrical building mass shape. It instantiates a corresponding geometry with input shape and size, and it also pre-computes a set of curves, depicting the four edges of the generated building mass. These curves are returned so that subsequent processes can generalize to any building mass shape without extra computation at each individual step, and this is visualized in Figure 3.8. Note that the *BuildingMass* node only returns a single floor of building mass, while the *BasicBuilding* node takes care of repeating the geometry "*num_of_floors*" times to form the full building mass shape, by computing another curve as the "grow" vector for floors and instantiating the single floor mesh on each point. This is also visualized in Figure 3.8.

BasicBuilding node then computes a bounding box for building mass, which it then use with roof height to obtain a z-axis coordinate as the center for the roof mesh. *Roof* node, just like *BuildingMass*, will then produce a mesh for roof of appropriate base shape and size according to input parameters, and the geometry will be lifted to the roof anchor point computed above. Similar methodology is used for windows and ledges, where a basic geometry will be generated and repeatedly instantiated on corresponding anchor points around the building mass. As its name suggests, *WindowWithLedge* and *CubedFloorLedgeForExtrusion* nodes produce the mesh and two instantiator nodes will handle the instantiation to proper places. For example, *BuildingWindowInstantiator* uses the aforementioned four curves depicting the building mass edges to determine the position and orientation to instantiate all windows on a floor, and it then uses the grow vector for floors to determine how to instantiate single-floor windows to all floors. Again, window anchor points with respective orientations on building mass edge lines are visualized in Figure 3.8. Note, that *CubedFloorLedgeForExtrusion* node produces a 2D grid mesh, so that *BuildingFloorLedgesInstantiator* can extrude the shape freely along the building mass edges according to the curve, without extra handling for the building mass being cubic or cylindrical. The extrusion method is visualized in Figure 3.9.

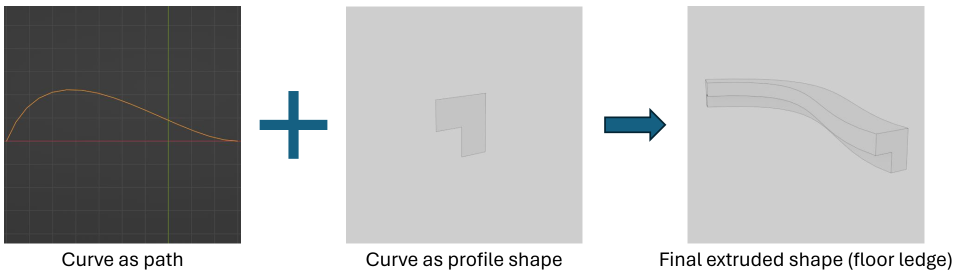


Figure 3.9: Extrude a profile curve/shape along a path described by another curve (curved or a straight line) and we end up with an L-shaped ledge for arbitrary building mass shapes.

The separation into shape generation nodes and instantiator nodes allows the shape program to treat partial shape programs on partial components as black box, that we can change the implementation of certain partial shapes or switch to use another node. It also allows us to, if we wish to, refer to some more complex but pre-defined base shapes for each component, and we can end up with a more complex and visually appealing procedural building model without much efforts.

A complete table of all parameters of our DAG program is provided in appendix A.1, and an example of a full parameter description is also provided in appendix A.2. We will discuss some examples here. Most of our parameters are named properly and are self-explanatory, such as "*Window Divided Horizontal*" and "*Window Divided Vertical*" controlling whether the window panels are divided, and "*Window Interpanel Offset Percentage Y/Z*" controlling the offset amount between window panels if divided. Their ranges are designed to be intuitive, for example "*Windows Left Right Offset*" has a range from -1 to 1, representing the leftmost and rightmost offset amount dynamically allowed by the building shape. Size vectors, such as "*Bm Size*" and "*Rf Size*", although will be normalized, have a sampling range from 0.5 to 1: lower values will result in extreme shapes which do not look natural as a building.

3.2 Dataset Generation

Due to the nature of this inverse procedural modeling task, we need to generate our own synthetic dataset with our shape program so that a neural network can be trained to regress parameters for it. Our *DAGDataset* is structured like follows:

```
DAGDataset<batch_num>_<images_per_batch>_<num_varying_params>/
├── images/
│   └── <batch_id>_<sample_id>.png
├── params/
│   └── <batch_id>_<sample_id>.yml
└── meta.yml
```

The file meta.yml contains supplementary information on the dataset, including camera angles used for rendering each batch (section 3.2.2), dataset name, a mapping of each parameter to each decoders (section 3.3.2), ranges (section 3.2.1)for each parameter for normalization and de-normalization purposes, and a mapping of switch labels (section 3.3.3).

3.2.1 Parameter Sampling

As the naming suggests, we generate data in batches, and this is due to how we implement our parameter sampling process. Instead of a fully random approach, in each batch, we randomly choose 5 out of all parameters as varying parameters and others as fixed. Each parameter is associated with a range: for discrete variables, it's a list of possible values it can take, and for continuous variables, it's pair of min/max values. We further divide this range to 10 intervals, and in a batch, we randomly decide on an interval for each fixed parameter so that its values are further constrained to a smaller range, while varying parameters do not have such limits, and they can be sampled from the full range. This process is formally stated in algorithm 1, it allows the model to be exposed to sketches where some features diverse significantly while exact same features do not exist.

3.2.2 Render

Having sampled values for each parameter, we now need to render the shape produced by our shape program if fed in these parameters. This can be divided into 2 steps: load the parameter into the procedural model to obtain a 3D shape, and render the shape as a sketch image.

3.2.2.1 Scene Setup and Parameter Loading

We define a placeholder cube in the 3D scene and apply the generated geometry node for buildings to it, and the geometry node modifier will overwrite the original geometry (cube) with generated building shape. We locate this object and its geometry node modifier, and then we iterate over the input parameters of the node tree, filling in corresponding values.

The camera is set up to always look at the building shape at (0, 0, 0), and its distance from the origin is jointly controlled by the radius and the height of a track it is placed on. By manipulating the track, we can change the viewing angle horizontally (or more easily, vertically, as in moving to the left or right) without affecting the distance from

Algorithm 1 Parameter Sampling for a Batch

```
1: Inputs:
     • $P$: list of parameters to be sampled, where each parameter $p_i$ is associated
       with a range
     • $v$: number of varying parameters
     • $n$: number of samples to be generated in this batch
2: Output: list of samples generated
3: Choose varying parameters $P_v$ and fixed parameters $P_f$ from $P$
4: Choose intervals for each $P_f$
5: batch_params = []
6: for $i \in n$ do
7:     cur_params = {}
8:     for $P_i \in P$ do
9:         if $P_i \in P_v$ then
10:            Sample from full range, add to cur_params
11:        end if
12:        if $P_i \in P_f$ then
13:            Sample from pre-determined interval, add to cur_params
14:        end if
15:    end for
16:    Add cur_params to batch_params
17: end for
18: return batch_params
```

the camera to the building shape by a series of simple trigonometry calculations. This is visualized in Figure 3.10.

3.2.2.2 Freestyle Rendering

We employ another Blender's built-in module, Freestyle, to render the building shape as sketches. In short, Freestyle handles a series of tasks such as edge selection and rendering in order to produce non-photorealistic renderings, including a hand-drawn style which is what we desire for sketch images. For comparison, see Figure 4.6 where cylinder's edge lines are on side surface is ignored by Freestyle rendering.

In our context, Freestyle renderer will render proper edges with black lines, and we leave only these lines to form our sketch drawings. We exclude plain building model from the image by setting them and the background to pure white, and thus we can use binarization on the rendered image to extract only sketch lines.

Since we expect input sketches to be hand-drawn, and they do not tend to follow strictly to perspectives and a certain viewing angle, we randomly disturb the rendering angles for each batch of images on both horizontal and vertical directions. For each batch, we choose a horizontal viewing angle from a range, which is a 5-degree walk-through from 30 degrees to 60 degrees, and when rendering each image, we apply an extra offset on both vertical and horizontal rotation within 20-degree range.

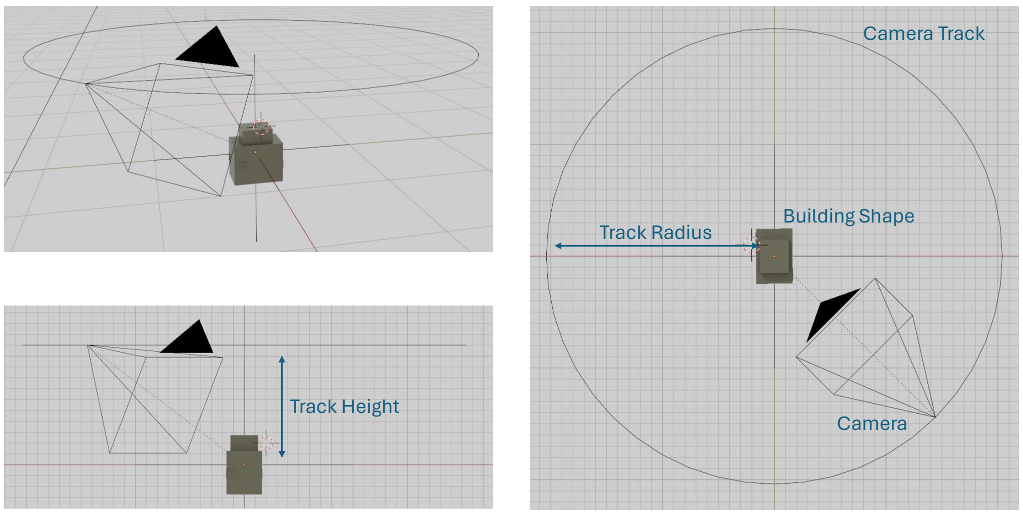


Figure 3.10: Scene set-up in Blender. We can make sure the length blue dashed line, which is the distance from camera to building shape, is not changing with viewing angle changes, by adjusting track's height and radius properly. Height is $2 \times \arcsin$ of horizontal angle and radius $2 \times \arccos$ of horizontal angle. Vertical angle changes can be done by rotating the track or offsetting the camera along the track path.

We give some examples from our synthetic dataset to show how Freestyle renderer can render mesh in a sketch style in Figure 3.11.

3.2.2.3 Our Own Curve Distortion Renderer

During early explorations for this project, we implemented our own curve distortion pipeline for rendering 3D shapes as sketches, which as its name suggests, includes one extra feature than the aforementioned Freestyle renderer: distortion of edge lines to curves to mimic hand-drawn lines.

The process can be summarized as a 4-step process: edge filtering, vertex disturbing, curve spawning and rendering. In edge filtering, we select only visible edges, and only sharp edges - non-sharp edges include repeated edges along a curved surface which human drawings do not tend to include but is present in discrete representation of meshes. This selection property is visualized in Figure 3.12. After obtaining desired edges, we record its midpoint coordinate and apply scaled random offsets to its two end vertices according to its length. With these new coordinate tuples (offset start, mid, offset end), we generate a bezier curve in place of the original edge, such that it will have a natural curvature instead of a straight line, which is more similar to human sketches. We then use Freestyle renderer again to render these curves. As a side note, we need to extrude these curves along a random axis slightly (1-e4 units), as a way to solidify these curves, such that the Freestyle renderer does not ignore them; to avoid repeated rendering of a curve, we employ the edge selection feature of Freestyle renderer by marking only one of the edges in the newly generated surface as the edge to be rendered,

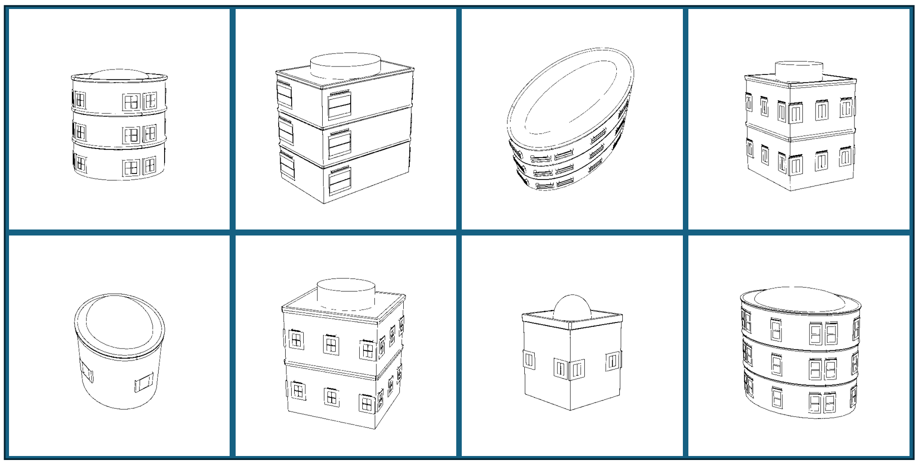


Figure 3.11: Some sketch images in our dataset. Note the variety of viewing angles.

and the marked edge is the same one as the generated bezier curve before extrusion.

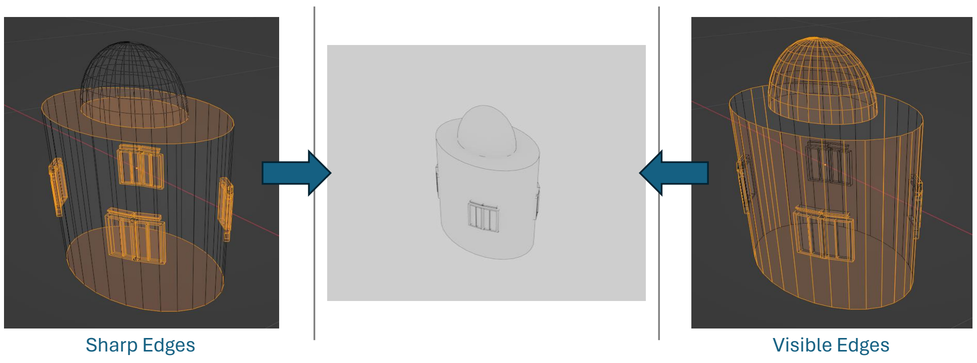


Figure 3.12: Sharp edges and visible edges on a cylindrical building shape, and the desired final render without any postprocessing that removes the grey building mesh.

As a side note, the generated building shape from the DAG node tree is not actual mesh until we convert them, and this is the Blender's default behavior; therefore, for the edge selection process to work on the actual shape instead of the placeholder cube, we create duplicates of our building shape and convert them during rendering, then remove them automatically after each render.

This process is optimized for three different types of objects: objects of straight lines, objects of cylindrical shapes and spheres, since they have different edge selection pipeline due to their geometric nature. Our building shape might contain multiple types of shapes, thus we implement a wrapper node around the *BasicBuilding* node, that can return full building shape, only building mass, only roof, only windows or only edges when given different flags as input. We then iterate through each component above, and apply the above process to spawn corresponding disturbed curves, in the end render all spawned curves. During rendering, we hide plain building model and all duplicated meshes, but we keep building mass and roof's duplicated mesh visible to block invisible curves, such as windows on the other side of the building shape. Similar to what we did with Freestyle-only rendering in previous section, by making the duplicate mesh as white as background, we can exclude the meshes during post-processing of binarizing and converting to grayscale.

We visualize some of the generated distortion renders in Figure 3.13, and we can observe some artifact: building mass and roof edges look like they are sketched multiple times due to the fact that we render their duplicate mesh along with generated curves, but this issue is also present on lots of human sketches where humans modify a line multiple times; the floor ledges on cylindrical building mass is messy, since it can only use striaght line renderer. Our algorithm did not optimise for partially occluded curves, therefore a work-around on roof and building mass is to block them with actual meshes during render; however, for ledges on cylindrical building mass, their shape is not strictly a cylinder since they are a straight line shape extruded along a circular path. Therefore we can only fallback to use straight line renderer for ledges until a better method on handling such shape is found. Due to these issues, we used synthetic datasets renderer with Freestyle in Blender as the major training data.

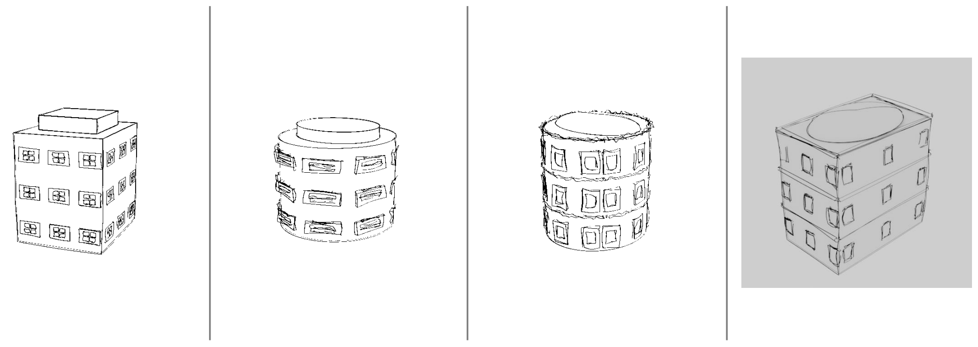


Figure 3.13: Post-processed and raw distortion renders

Furthermore, this algorithm is not perfectly stable and it will crash Blender with segmentation fault in extreme cases. The core algorithm was developed under assumption that it will handle individual and primitive shapes and was not optimised to handle full building shapes. The largest distortion dataset we created was at 496 images, which was not enough to properly train a neural network that can generalize to more diverse patterns. This will be discussed in section 4.1.3.

3.3 Neural Network

Having generated corresponding sketch images for different sets of parameters, we can proceed to train our neural networks. Our neural network architecture consists of an Encoder and a series of Decoder networks, and it takes as input 512 $\times$ 512 gray-scale images, predicts parameters for our shape program mentioned above. The overall architecture is shown in Figure 3.14.

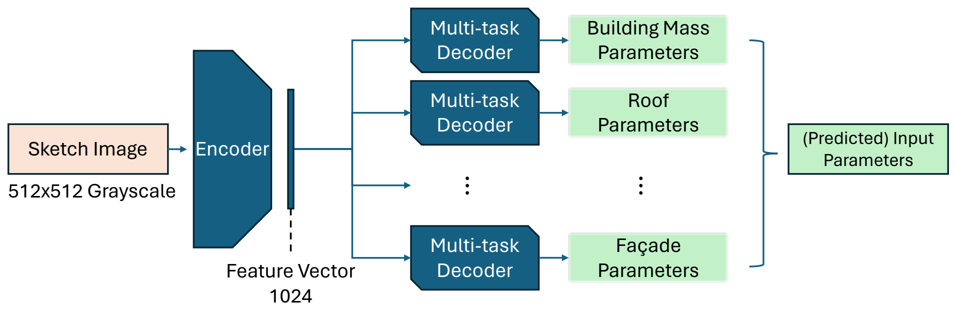


Figure 3.14: Overall architecture for our encoder-decoder neural network on parameter prediction.

3.3.1 Encoder

The encoder network accepts a 512x512 gray-scale single-channel image as input, and it embeds the image into a feature vector of length 1024. It contains 2 convolution layers with 3x3 kernel size and a padding of 1, of 32 and 64 output channels each, followed by ReLU activation layers and Max Pooling layers, and as a final projection, a linear layer producing the final feature embedding.

3.3.2 Param-aware Multi-task Decoder

In comparison to Pearl et al. (2022)'s work where they train decoders for each parameter, we organize decoders according to semantic components on a building. Our decoders thus include: *BuildingMassDecoder* on building mass base shape and size, *FacadeDecoder* on number of floors and windows on each side, *RoofDecoder* on roof base shape and size, *WindowMainDecoder* on horizontal/vertical offset for windows and window size, *WindowPanelDecoder* on window panel size, how they are divided and inter-panel offset if divided, *WindowLedgeDecoder* on whether there is a ledge for each window and if so, its size, its extrusion to form the L-shape and its height offset from the window, and finally *FloorLedgeDecoder* on whether floor ledges exist and their size and extrusion. This design aims to leverage the advantage of grouping semantically similar parameters under one network, such that some weights can be shared and the output layers become context-aware. We conducted experiments on this design choice and will discuss them in the section 4.1.2.

This set-up introduces a new problem, that some decoders need to predict both discrete and continuous values. We employ multi-task learning techniques and design a Multi-task decoder architecture, consisting of a classification branch for each discrete variable, and a regression branch for each continuous variable. The decoder will first process the feature vector with 2 shared dense layer, ending up with a processed feature vector of size 512, and this processed decoder-specific feature vector will be given to branches of different task. For each branch on a task, 2 linear layers are used to produce the final prediction of corresponding parameter type (probability logits for classification and raw values for regression). To facilitate proper parsing of parameters, we format the shape parameters as dictionaries.

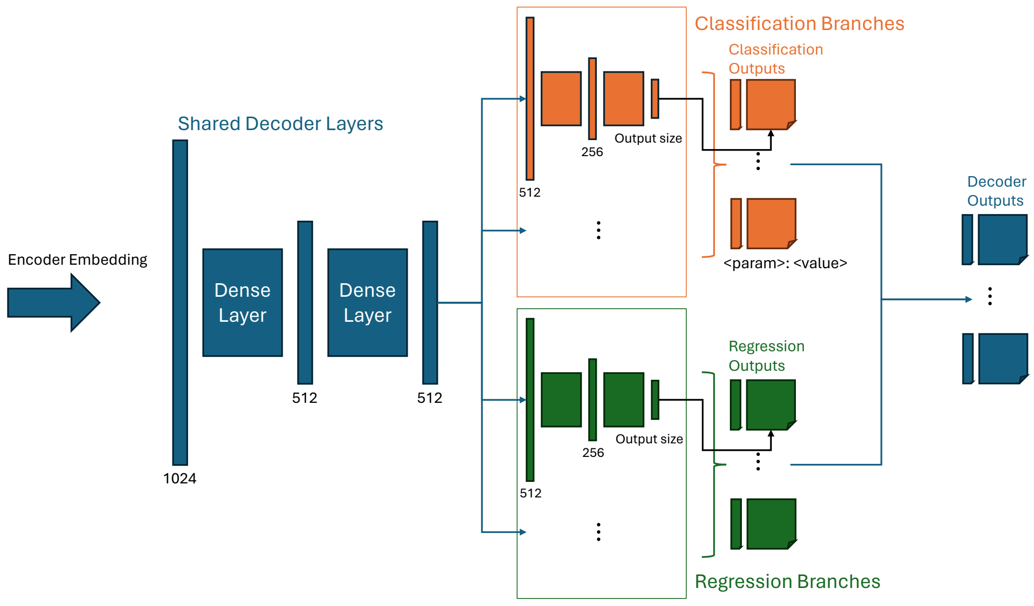


Figure 3.15: Param-aware (as dictionaries) Multi-task Decoder

To summarize, each decoder network accepts the feature embedding from encoder, and produces corresponding prediction results from their respective classification and regression branch according to how they are set-up with the parameter list above. These outputs, as dictionaries, are then combined back to form a complete set of shape parameters. After a further denormalization process on the data, these parameters can be loaded back into the shape program to reassemble the 3D building model from the sketch image. We visualize this architecture in Figure 3.15.

3.3.3 Custom Loss Criterion

Due to the use of multi-task decoder networks, we need to implement a custom loss criterion for the neural network training procedure to work. In short, we use cross entropy loss on classification tasks and L1 loss on regression tasks.

In a forward call to compute loss for a batch of neural network outputs, we first iterate through each decoder outputs and compute their combined loss by calculating a weighted average of classification and regression losses (in most of our training, the weight ratio is 1:1). Empirically, we found that our network is prune to overfit possibly due to the simple nature of our image inputs, thus counter-measures such as weight regularizors and dropout layers should be employed. This loss function thus accepts related inputs to control the weight regularization, such as which norm to use (L1, L2 or no regularization). Overfit issues will be further compared and discussed in the section 4.1.1.

Note that some classification tasks determine whether a shape component is present, and when the shape is not present, we wish to disable the gradient update on corresponding regression branches. We achieve this by utilising ground turth values (0 or 1) for these classifcation tasks as switch labels: we multiply the ground truth value to the loss value of corresponding regression branches, and if the shape component is not present on the sketch (gold label is 0), the loss for that regression task is set to 0, and otherwise unchanged (multiplication by 1).

"*Has Floor Ledge*" and "*Window Divided Horizontal*" are examples of above switch labels, and they control corresponding shape paramters like "*Window Interpanel Offset Percentage Y*". A table of all switch labels and parameters under their control is given in appendix A.3.

3.3.4 Training

Our training loop is similar to standard approaches. We first load the dataset and split data samples to train, validation and test set with ratio 0.8:0.1:0.1, and we create dataloaders for each set. We then assembly our full neural network by initializing the encoder model, loading metadata and initializing corresponding decoders. We then train the neural network with Adam optimizer and our custom loss function, logging validation set performance in terms of validation loss every 10 or 100 iterations (batches). In the end, we evaluate the model on test set with F1 score (F1) as the metric for classification tasks and Mean Absolute Error (MAE) for regression tasks, and we produce a visualization of the training and validation loss curves. For results, see section 4.1, and for final results, see section 4.1.4, specifically Figure 4.4.

3.4 User Interface

We leveraged Blender's detailed support on interface development to integrate our pipeline as a Blender plugin with a graphical user interface (GUI). Another reason is that we can use Blender's built-in *Annotate* tool to serve as a means for sketching directly. The interface is shown in Figure 3.16.

Note that our plugin also works on Pearl et al. (2022)'s work, GeoCode, and in fact, we attempted to integrate their pipeline as a plugin as one of the early exploration of this project.

3.4.1 Inference Pipeline

We first extract our inference pipeline from neural network training and testing procedure: load metadata and models, load one single image as input with proper transforms (to 512×512 grayscale), perform forward pass on the model, then parse and denormalize the outputs, and save it as a yml file.

We also implemented a batch inference process to facilitate more efficient testing and visualization, where the above inference process is applied to a set of images, usually the test set. The only difference is to perform forward pass on a batch of images instead.

The denormalization process utilises the model output value and parameter specifications from metadata, where it applies min/max denormalization on continuous variables like floats or uses *torch.argmax* to extract corresponding class indices on discrete variables.

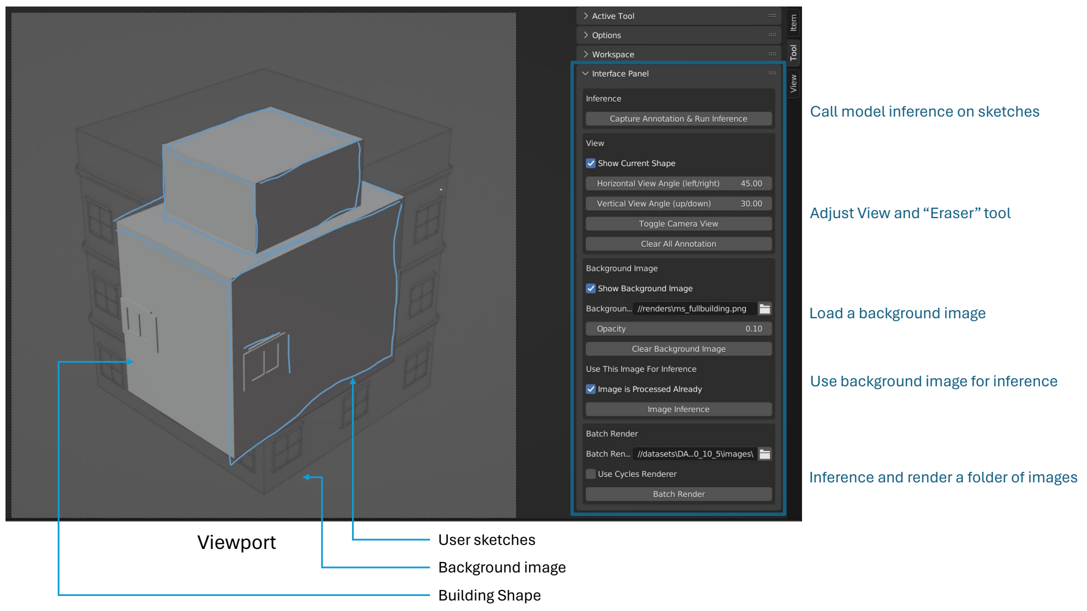

Active Tool
Options
Workspace
Interface Panel
Inference
Capture Annotation & Run Inference
View
Show Current Shape
Horizontal View Angle (left/right) 45.00
Vertical View Angle (up/down) 30.00
Toggle Camera View
Clear All Annotation
Background Image
Show Background Image
Backgroun... //renders\ms_fullbuilding.png
Opacity 0.10
Clear Background Image
Use This Image For Inference
Image is Processed Already
Image Inference
Batch Render
Batch Ren... //datasets\DA...0_10_5\images\
Use Cycles Renderer
Batch Render
Item
Tool
View
Call model inference on sketches
Adjust View and "Eraser" tool
Load a background image
Use background image for inference
Inference and render a folder of images
Viewport
User sketches
Background image
Building Shape

Figure 3.16: User Interface with elements and functionalities annotated. User sketches in blue lines are made with blender's built-in *Annotate* tool. Detailed explanation on different functionalities will be given in section 3.4.2.

3.4.2 Interface

We refer readers to Blender's official manual page on Add-on development for detailed introduction on how to implement a plugin with user interface. In short, it involves defining a set of operators to convey desired functionalities, and putting visual elements like buttons in place to trigger corresponding functionalities. As a side note, in order for Blender to use third-party libraries like PyTorch, a work around is to delete Blender's built-in Python directory and start Blender from a terminal with activated conda environment containing necessary dependencies, so that Blender will fallback to the current Python path, as long as the Python version is the same with what Blender was compiled on. For our work, we used Blender 3.2, which was bundled with Python 3.10. This is documented on official Tips and Tricks page.

As mentioned previously, we use Blender's built-in *Annotate* tool as the sketching tool, and after the user having completed the sketch, the core functionality of our user interface let the user to call model inference on the new sketch with one button click. The process will save the user sketches on a 512×512 grayscale image, and invoke the aforementioned inference pipeline; it then utilised a similar procedure on parameter loading mentioned in section 3.2.2.1 that loads predicted parameters into the building shape and visualize it in the scene.

There are also various auxiliary features:

- **Background Image**: it allows the user to load a background image as drawing reference, and the opacity can also be conveniently adjusted; this functionality also serve as a way to visualize differences between produced building shape and ground truth sketch.

- **Inference on Image**: the background image can also be used directly for inference, as long as it is a sketch-style image; this facilitates faster testing during development phase since it allows re-use of existing sketch images, and the user may also use this feature directly to verify model performance.
- **Batch Render**: the aforementioned batch inference functionality on a set of images. This creates a *pred* folder in the directory containing the images, and saves model outputs and renders of produced building shape in it.
- **Miscellaneous**: this includes small quality-of-life features like erasing all existing sketch/annotations, adjusting viewing angles vertically or horizontally, toggling camera view, etc.

Chapter 4

Results and Discussions

4.1 Results

We showcase our experiments conducted in this section and provide some insights into each of them. All experiments are conducted with the same dataset of 10k synthetic data samples, with a batch size of 32 and max epoch 5, logging validation loss every 100 iterations.

Table 4.1: Performance Metrics on 10k dataset; best in category is in bold, while best overall is in italics.

Architecture	Avg F1	Max F1	Avg MAE	Min MAE	Best Val Loss
no_layer_sharing	0.7956	0.9659	0.2086	0.0936	6.9078
shared_layer	**0.8160**	**0.9680**	**0.2015**	**0.0799**	**5.7685**
dropout	**0.8336**	0.9740	**0.1877**	**0.0726**	**4.9552**
dropout_l1	0.7403	0.9372	0.2110	0.0890	12.1698
dropout_l2	0.8095	0.9700	0.2023	0.0862	5.6561
dropout_l2_larger	0.8245	**0.9760**	0.2012	0.0787	6.8061
plain	0.8339	***0.9780***	0.1873	0.0730	4.6997
plain_larger	***0.8501***	0.9770	***0.1840***	***0.0679***	***4.5435***
plain_larger_dropout	0.8340	0.9730	0.1897	0.0721	5.0566

4.1.1 Experiments - overfit

Empirically, we found that the neural network is more prune to overfit issues comparing to usual models, and as mentioned previously in section 3.3.3, we suggest that this is due to the simplicity of our input sketches by their nature. We attempted to control the overfit by employing dropout layers and loss regularizors, and experimented with different batch sizes and dataset sizes, but the effectiveness is negligible; we further discover that the model without any overfit countermeasures yields the best performing model, with weights saved at best validation loss. Training curves of experiment on dropout and L2 regularizors is compared in Figure 4.1, and performances are summarized in table

4.1. Note that ”large” refers to the configuration of 2 shared layers instead of 1, 2 dense layers in each branch instead 1.

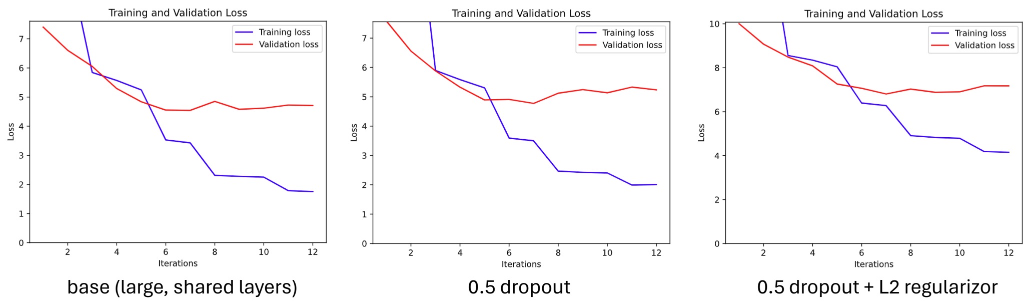


Figure 4.1: Experiment on controlling overfit. Dropout with L2 regularization is indeed the most effective in controlling overfit, but it yields worse overall performance.

One possible explanation is from the data’s perspective: our synthetic images have highly repetitive features since they are directly generated from perfect meshes, therefore certain amount of overfit, that is, the model memorizing some partial features exactly can lead to a slightly better performance. To address this issue, and also as will be mentioned in experiments with distorted sketches in section 4.1.3, we train our final model with best configuration but dropout layer activated.

4.1.2 Experiments - shared layers

We aim to verify the expected advantage of shared layers in decoder in this experiment. The decoder without layer sharing has 2 dense layers for each branch, while the layer-sharing decoder model has 1 common decoder dense layer, and then 1 dense layer for each branch. The training curves shown in Figure 4.2 indicate no significant difference in convergence rate but slightly better loss overall, and the performance comparison in table 4.1 suggests a minor overall advantage too. Taking into account that the layer-sharing model consists of less parameters, we end up preferring the layer-sharing approach in the decoder since it is more effective.

4.1.3 Test on distortion dataset

As mentioned in section 3.2.2.3, we have generated a dataset of 496 sketch images on building shapes with distorted curves. Although not enough for training, we experimented with it and showed that the quality was enough to fit a model, and we conducted testing with it using other models.

Training results are shown in Figure 4.3, and we can notice a better performance comparing to other models on other datasets. However, with such a small model and only 4 different sets of varying parameter patterns (4 batches during generation), it is likely that the model only learns how to distinguish between 4 patterns and it will perform well even on test set. Although invalid, it still shows that our distortion renders

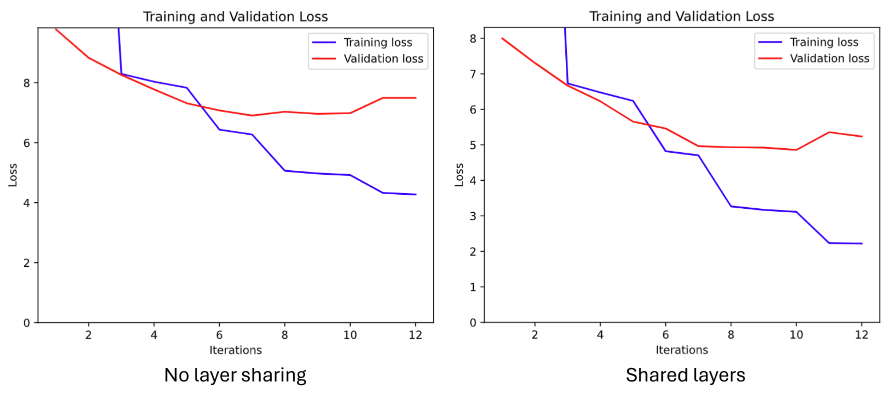


Figure 4.2: Layer sharing experiments. The axis are not perfectly matched since they are generated by our code automatically from individual loss records, but the trend and value differences can still prove the advantage of shared layers.

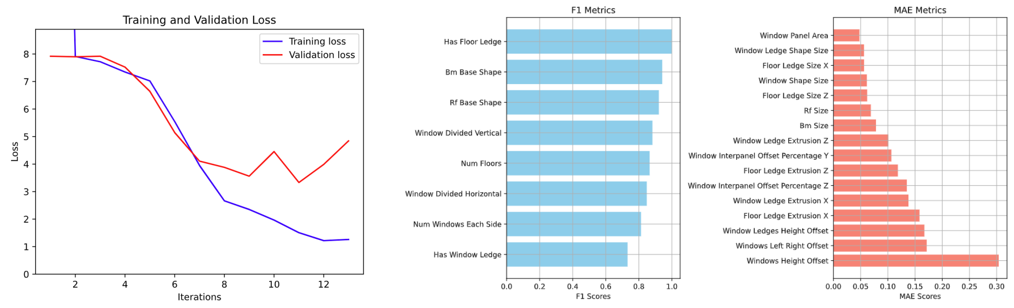


Figure 4.3: Training on Distortion dataset of 496 images

are good enough to ”train” and ”fit” a neural network, at the very least, they are not random curves.

Table 4.2: Cross-testing Performances; normal model refers to prior best: larger plain model with no weight regularizors

Architecture	**Avg F1**	**Max F1**	**Avg MAE**	**Min MAE**
distortion_model_on_normal	0.5232	0.7774	***0.2522***	0.1343
normal_model_on_distortion	0.6215	***0.9371***	0.2541	**0.1006**
normal_model_with_dropout_on_distortion	***0.6353***	0.8833	**0.2530**	0.1063

We thus use the distortion dataset as test set to verify other models and we summarize the cross-testing results in table 4.2. Although differences are minor, we found out that models with dropout perform slightly better on average, and thus we proceed to train our final model with dropout activated. This meets our expectation that dropout layers make models more flexible when presented with data of reasonably different distributions.

4.1.4 Final Model

Our best performing model is trained on 30k images (300 batches $\times$ 100 images per batch), with batch size 32, achieving a best validation loss of 5.0398 at iteration 26 (2600 batches past) and a final test loss of 4.9720. The loss number is not close to zero due to the way we calculate it (see section 3.3.3), but the model performance is satisfactory, with an average F1 score of 0.83 and an average MAE of 0.22. Detailed performance for each parameter is visualized in Figure 4.4.

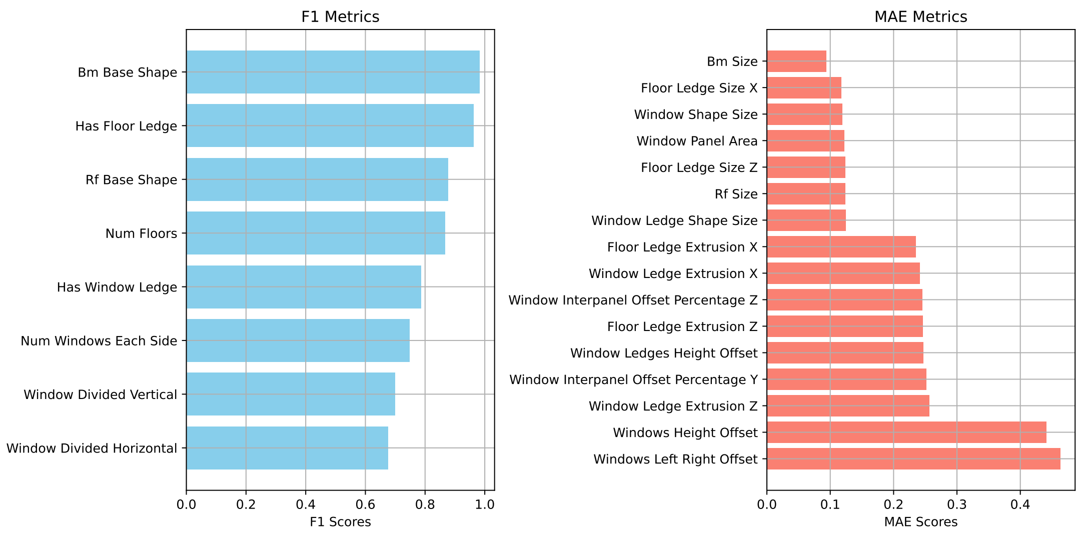


Figure 4.4: Performance on discrete and continuous variables, of F1 scores and Mean Absolute Errors, respectively. Sorted from better to worse performance. Bm is short for building mass, and rf refers to roof. We name window ledge and floor ledge explicitly to avoid ambiguity.

Some of the predicted shapes are visualized in Figure 4.5 and 4.6, which are building shape renders directly from Cycles render engine and line mode renders with corresponding ground truth as the background.

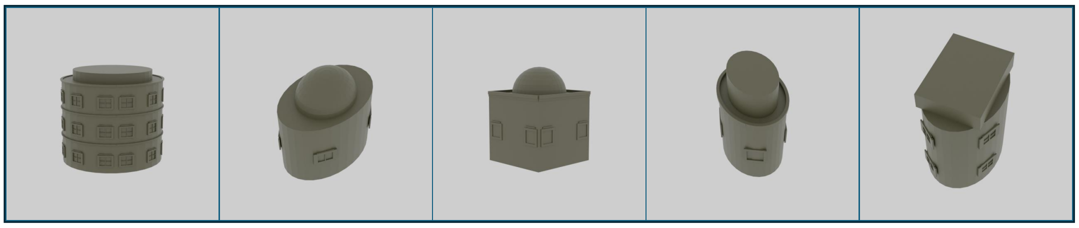


Figure 4.5: Cycles rendering of predicted building shapes.

4.2 Discussions

Figure 4.4 shows that model performance on visually smaller features, like windows and ledges, are worse comparing to more visible features like building mass sizes. This

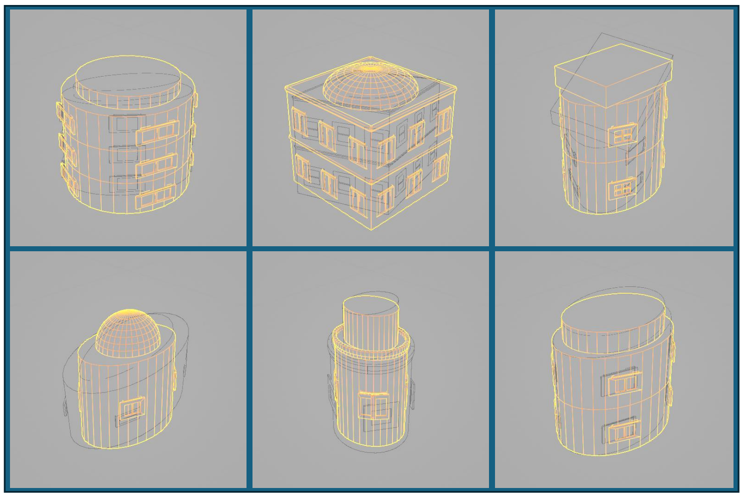


Figure 4.6: Yellow lines are predicted shape under line mode, while black lines are ground truth sketches. Some visual discrepancy comes from mis-aligned viewing angles.

meets our expectation since we are providing one single sketch of a full building shape with a limited resolution of 512 $\times$ 512 pixels, and it would reasonably be harder for the encoder to capture smaller details as good as larger features.

To address this problem, some solutions include enlarging input image size or extending the input domain further. For example, we could support the input of multiple images at the same time, and allowing the user to describe desired window shape in details in a separate image. The amount of drawings needed for this approach would then fall back to Nishida et al. (2016)'s work, but this can still be a one-step process instead of an iterative pipeline.

Chapter 5

Conclusions

In this project, we designed and implemented a DAG-based procedural shape program for buildings, and we generated a synthetic dataset with pairs of shape program parameters and corresponding sketch image of the produced shape to train a neural network, which includes an encoder and a series of multi-task decoder networks. We also attempted to build a novel renderer that mimic the hand-drawing style of human sketches. We utilised Blender to build a graphical user interface which allows user to provide sketches, run the neural network to predict shape parameters, and automatically loads the parameter back into the shape program to visualize the produced building shape. We showed that using multiple decoder networks can effectively extract shape parameters, while visually smaller features are harder to capture due to image resolution limitations. Although our implementation was highly integrated with Blender, the process can be generalized to use any other methods of implementing shape programs and user interfaces.

5.1 Future Work

In this section we discuss potential extensions to this project, and we wish to highlight a path that is inspired by our use of Geometry Scripts.

5.1.1 Extending Input Domain

As mentioned in section 4.2, we can attempt to solve the problem on capturing minor details through extending the input domain. Instead of inputting one single image, we allow for multiple images, for example, a compulsory one on overall building shape, and optional ones depicting shape details of certain components like windows. Optional images can be inputted by filling in pre-defined slots on the user interface, or they may come with semantic labels. This would require modifying the encoder network architecture to support such inputs of rich formats.

5.1.2 Distortion Dataset

One trivial improvement is to refine the distortion renderer mentioned in section 3.2.2.3, since it will naturally be more similar to human sketches. One potential and small adjustment is to train a model on synthetic dataset first, then fine-tune it on distortion dataset, if the algorithm performance is still not ideal.

5.1.3 Generating Code Snippets

Inspired by our use of Geometry Scripts library and how we can write Python code and end up with a DAG program, we can leverage recent advances in the Large Language Models (LLM) and train a model to output these Python codes or any other form of texts directly, which can be interpreted as a shape program. In fact, the Geometry Scripts library has such a feature in early testing that can convert existing Geometry Nodes back to Python codes, and thus we can leverage the large number of existing Blender Geometry Nodes programs to form a first training dataset of shape programs in their textual representations, such as code snippets, and corresponding (sketch) images that is produced by such shape programs. This can also generalize to any form of shape programs, not only procedural ones, since there also exists various CAD programs that produces shape, and can be converted to corresponding textual or code representations. This idea is visualized in Figure 5.1. If this can be achieved, we can then solve the first problem of Inverse Procedural Modeling mentioned in section 1.1, that is, creating procedural shape programs from scratch.

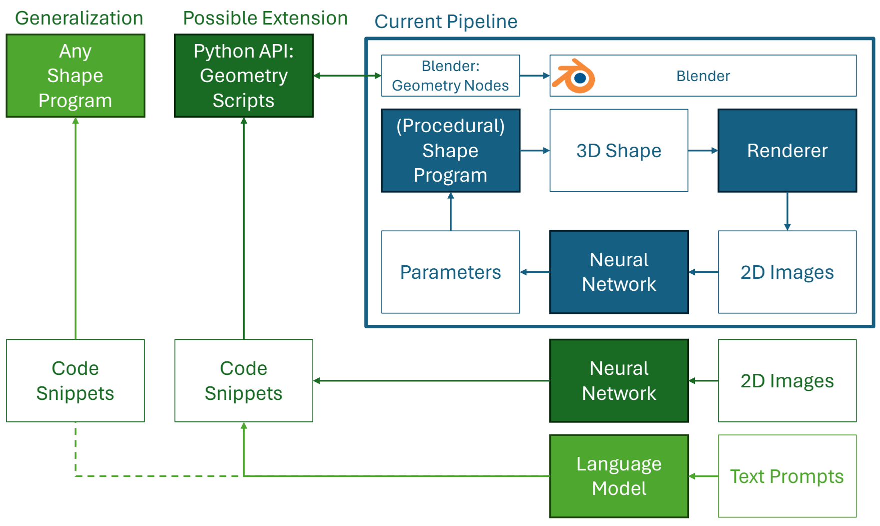


Figure 5.1: Extensions on a text-based approach, generating textual representations of shape programs instead.

Bibliography

Oscar Argudo, Carlos Andujar, Antonio Chica, Eric Guérin, Julie Digne, Adrien Peytavie, and Eric Galin. Coherent multi-layer landscape synthesis. *The Visual Computer*, 33(6-8):1005–1015, May 2017. doi: 10.1007/s00371-017-1393-6. URL `https://doi.org/10.1007/s00371-017-1393-6`.

Maryam Ariyan and David Mould. Terrain synthesis using curve networks. In *Proceedings of the 41st Graphics Interface Conference*, GI '15, page 9–16, CAN, 2015. Canadian Information Processing Society. ISBN 9780994786807.

Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017. doi: 10.1109/TPAMI.2016.2644615.

Gwyneth A Bradbury, Il Choi, Cristina Amati, Kenny Mitchell, and Tim Weyrich. Frequency-based creation and editing of virtual terrain. In *Proceedings of the 11th european conference on visual media production, London, UK. ACM, New York*, pages 13–14. Citeseer, 2014.

Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. In *ACM SIGGRAPH 2008 papers*, pages 1–10. 2008.

Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

Guillaume Cordonnier, Marie-Paule Cani, Bedrich Benes, Jean Braun, and Eric Galin. Sculpting mountains: Interactive terrain modeling based on subsurface geology. *IEEE Transactions on Visualization and Computer Graphics*, 24(5):1756–1769, 2017.

Dieter Finkenzeller and Jan Bender. Semantic representation of complex building structures. In *Computer Graphics and Visualization (CGV 2008)-IADIS Multi Conference on Computer Science and Information Systems, Amsterdam, The Netherlands*, 2008.

Eric Galin, Eric Guérin, Adrien Peytavie, Guillaume Cordonnier, Marie-Paule Cani, Bedrich Benes, and James Gain. A review of digital terrain modeling. *Computer Graphics Forum*, 38(2):553–577, 2019. doi: https://doi.org/10.1111/cgf.13657. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13657`.

Takashi Ijiri, Shigeru Owada, and Takeo Igarashi. The sketch l-system: Global control of tree modeling using free-form strokes. In *International symposium on smart graphics*, pages 138–146. Springer, 2006.

Thomas Lechner, Pin Ren, Ben Watson, Craig Brozefski, and Uri Wilenski. Procedural modeling of urban land use. In *ACM SIGGRAPH 2006 Research posters*, pages 135–es. 2006.

Steven Longay, Adam Runions, Frédéric Boudon, and Przemyslaw Prusinkiewicz. Treesketch: Interactive procedural modeling of trees on a tablet. In *SBIM@ Expressive*, pages 107–120. Citeseer, 2012.

Ruan Lotter. *Taking Blender to the Next Level: Implement advanced workflows such as geometry nodes, simulations, and motion tracking for Blender production pipelines*. Packt Publishing Ltd, 2022.

Ignacio Martin and Gustavo Patow. Ruleset-rewriting for procedural modeling of buildings. *Computers Graphics*, 84:93–102, 2019. ISSN 0097-8493. doi: https://doi.org/10.1016/j.cag.2019.08.003. URL `https://www.sciencedirect.com/science/article/pii/S0097849319301311`.

Markus Mathias, Andelo Martinovic, Julien Weissenberg, and Luc Van Gool. Procedural 3d building reconstruction using shape grammars and detectors. In *2011 International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission*, pages 304–311, 2011. doi: 10.1109/3DIMPVT.2011.45.

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, page 614–623, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933646. doi: 10.1145/1179352.1141931. URL `https://doi.org/10.1145/1179352.1141931`.

Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. Interactive sketching of urban procedural models. *ACM Trans. Graph.*, 35(4), jul 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925951. URL `https://doi.org/10.1145/2897824.2925951`.

Gen Nishida, Adrien Bousseau, and Daniel G. Aliaga. Procedural modeling of a building from a single image. *Computer Graphics Forum*, 37(2):415–429, 2018. doi: https://doi.org/10.1111/cgf.13372. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13372`.

Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 301–308, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 158113374X. doi: 10.1145/383259.383292. URL `https://doi.org/10.1145/383259.383292`.

Ofek Pearl, Itai Lang, Yuhua Hu, Raymond A. Yeh, and Rana Hanocka. Geocode: Interpretable shape programs, 2022.

Sören Pirk, Ondrej Stava, Julian Kratt, Michel Abdul Massih Said, Boris Neubert, Radomír Měch, Bedrich Benes, and Oliver Deussen. Plastic trees: interactive self-adapting botanical tree models. *ACM Transactions on Graphics (TOG)*, 31(4):1–10, 2012.

Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, 34(4 (Proceedings of SIGGRAPH 2015)):107:1–107:12, August 2015.

Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, 33(6):31–50, 2014. doi: https://doi.org/10.1111/cgf.12276. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12276`.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL `https://proceedings.neurips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf`.

Emma R Tait and Ingrid L Nelson. Nonscalability and generating digital outer space natures in no man's sky. *Environment and Planning E: Nature and Space*, 5(2): 694–718, 2022. doi: 10.1177/25148486211000746. URL `https://doi.org/10.1177/25148486211000746`.

Carlos A. Vanegas, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Paul Waddell. Inverse design of urban procedural models. *ACM Trans. Graph.*, 31 (6), nov 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366187. URL `https://doi.org/10.1145/2366145.2366187`.

Ling Xu and David Mould. Procedural tree modeling with guiding vectors. *Computer Graphics Forum*, 34(7):47–56, 2015. doi: https://doi.org/10.1111/cgf.12744. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12744`.

O. Šťava, B. Beneš, R. Měch, D. G. Aliaga, and P. Krištof. Inverse procedural modeling by automatic generation of l-systems. *Computer Graphics Forum*, 29(2):665–674, 2010. doi: https://doi.org/10.1111/j.1467-8659.2009.01636.x. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01636.x`.

Appendix A

Supplementary Materials

A.1 Parameter for Procedural Building Model

Table A.1: Parameter Ranges

Name	**Type**	**Min**	**Max**	**Values**	**Notes**
Bm Base Shape	states			0, 1	
Bm Size	vector	0.5	1.0		
Floor Ledge Extrusion X	float	0.0	1.0		
Floor Ledge Extrusion Z	float	0.0	1.0		
Floor Ledge Size X	float	0.5	1.0		
Floor Ledge Size Z	float	0.5	1.0		
Has Floor Ledge	bool			true, false	
Has Window Ledge	bool			true, false	
Num Floors	int	1	3		
Num Windows Each Side	int	1	3		
Rf Base Shape	states			0, 1, 2	
Rf Size	vector	0.5	1.0		
Window Divided Horizontal	bool			true, false	
Window Divided Vertical	bool			true, false	
Window Interpanel Offset Percentage Y	float	0.0	1.0		
Window Interpanel Offset Percentage Z	float	0.0	1.0		
Window Ledge Extrusion X	float	0.0	1.0		
Window Ledge Extrusion Z	float	0.0	1.0		
Window Ledge Shape Size	vector	0.5	1.0		
Window Ledges Height Offset	float	0.0	1.0		
Window Panel Area	vector	0.5	1.0		
Window Shape Size	vector	0.5	1.0		
Windows Height Offset	float	-1.0	1.0		
Windows Left Right Offset	float	-1.0	1.0		

A.2 Example Full Parameter File

```
Bm Base Shape: 1
Bm Size:
    - 0.5469961558823341
    - 0.8595860190380872
    - 0.8422490800187404
Floor Ledge Extrusion X: 0.9718873058425997
Floor Ledge Extrusion Z: 0.012460426613071686
Floor Ledge Size X: 0.8738743489277787
Floor Ledge Size Z: 0.6251809990810011
Has Floor Ledge: true
Has Window Ledge: true
Num Floors: 2
Num Windows Each Side: 1
Rf Base Shape: 0
Rf Size:
    - 0.9198597476390333
    - 0.8391562396905418
    - 0.6083579348444159
Window Divided Horizontal: false
Window Divided Vertical: false
Window Interpanel Offset Percentage Y: 0.46331911529950026
Window Interpanel Offset Percentage Z: 0.41987389848680495
Window Ledge Extrusion X: 0.333514149892149
Window Ledge Extrusion Z: 0.8602255944755194
Window Ledge Shape Size:
    - 0.5445528175572686
    - 0.8130952070728544
    - 0.5582955490053841
Window Ledges Height Offset: 0.12813910934131095
Window Panel Area:
```

- 0.5680374741365545
- 0.8360632399152894
- 0.9788619285849408

Window Shape Size:

- 0.576957710454006
- 0.6387979181666416
- 0.660421379722682

Windows Height Offset: 0.9178245070432143

Windows Left Right Offset: 0.18316941923838673

A.3 Switch Labels

Table A.2: Switches and Associated Parameters

Switch	**Associated Parameters**
Has Floor Ledge	Floor Ledge Size X
	Floor Ledge Size Z
	Floor Ledge Extrusion X
	Floor Ledge Extrusion Z
Has Window Ledge	Window Ledge Shape Size
	Window Ledge Extrusion X
	Window Ledge Extrusion Z
	Window Ledges Height Offset
Window Divided Horizontal	Window Interpanel Offset Percentage Y
Window Divided Vertical	Window Interpanel Offset Percentage Z