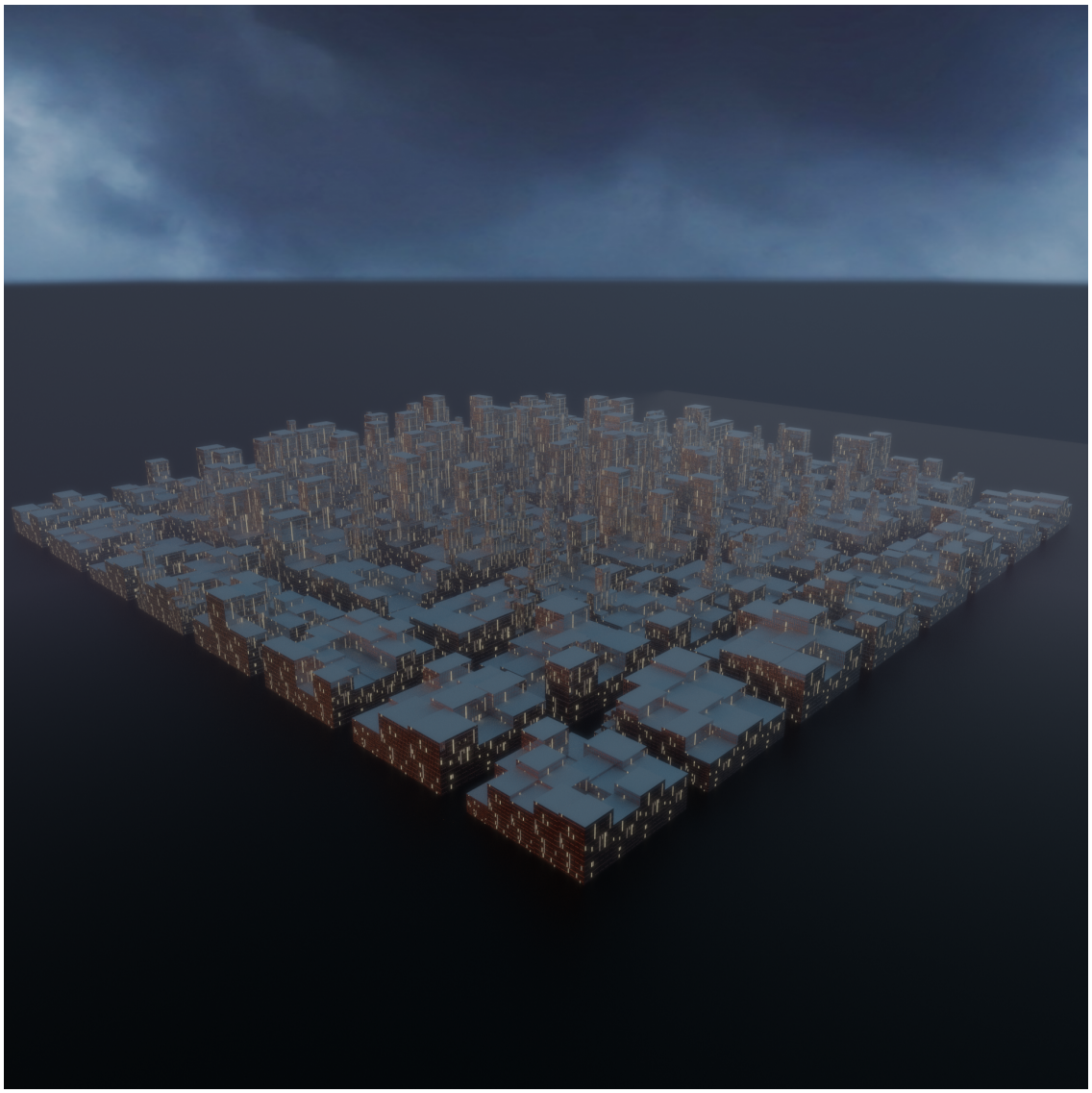
Procedural Generation of Buildings

Candidate Number: 1026702
Final Honour School of Computer Science - Part B

Trinity 2020



Contents

Chapter 1

Introduction

1.1 Motivation

Creating a virtual city for a film, TV show or video game is time-consuming. Artists often have to design many individual buildings from scratch. Rather than precisely modelling each building in a city, it may be easier to describe the style of the buildings required and then have a computer generate many buildings that all match that style. Previous research has shown that there are effective ways of describing to a computer how to construct a building but these methods still require a certain level of expertise. They often require a computer scientist or other specialist to work alongside an artist.

As well as the modelling of specific buildings, extensive research has been done regarding the layout of cities. These methods can generate road layouts, parks and plot locations according to some given parameters [6] [1]. We will not focus on the challenge of city layout in this project because if we can create a set of buildings, then methods already exist that could appropriately combine these buildings together to form a city.

We will also not address the challenge of adding textures to the buildings we generate. Instead we will focus on the geometry of the buildings. Methods already exist for the procedural generation of façades and textures. There are even inverse modelling strategies that can be used to automatically create a façade-generating grammar from a given example [7]. Any textures we have used in examples in this report are not of our own design and are simply there to demonstrate how our methods can be combined with textures to create realistic buildings.

1.2 Aims

For this project, we have two main aims:

1. Define a grammar that allows users to describe a building using a set of construction rules. We will also create a parser that can generate 3D models from a given grammar.
2. Define a method of reverse-engineering a grammar from a set of example buildings. The inferred grammar should be able to create many unique buildings that are similar to the examples.

1.3 Contributions to the Field

We provide an analysis of existing strategies used to procedurally generate buildings. Based on this analysis, we define a new, descriptive building grammar that has the advantage of being easy to reverse-engineer from a set of example buildings.

We provide a parser that can be used to allow anyone to create their own buildings from a hand-written grammar. The parser is designed to easily allow others to customise the definition of a grammar and add new operations.

We define an intermediate representation of buildings that is somewhere between grammars and 3D geometry called *operation graphs*. These operation graphs provide a novel formalisation of building structure that we believe could be useful in future research.

We describe a new method of generating a grammar from a set of example buildings that attempts to capture the style of the examples. We outline where this new method excels as well as its limitations.

Part I

The Forward Problem

Chapter 2

Existing Research and Potential Approaches

Extensive research has been carried out regarding the procedural generation of buildings. Many of these methods involve a rule-based grammar that describes how to construct the buildings. These grammars often allow some form of non-determinism so that a single grammar is able to generate various different structures. A popular grammar that has been built upon by others is *CGA Shape* [4]. The language of this grammar includes various operations that act on an oriented bounding box in 3D space. *CGA Shape* has been used to successfully create realistic buildings using a reasonable number of manually-written rules.

Figure 2.1: A procedurally generated model of Pompeii created by the authors of [4] that uses a *CGA Shape* grammar with 190 manually-written rules.

The main limitation of *CGA Shape* is that it takes skill to describe buildings using the grammar. To address this problem, some building grammars try to provide more intuitive ways of editing grammar rules [2] and others suggest more complex and descriptive operations. In Part 2, we will target the problem by automatically inferring grammar rules from example buildings.

We could create our own more-descriptive grammar that allows precise modelling of intricate buildings but this would make it more challenging to reverse-engineer a grammar (the second of our aims for this project). Instead, we will base our grammar on *CGA Shape* with a few modifications and simplifications that make the backwards challenge more achievable. In Part 1 of this report, we will describe the grammar we have created and how we can generate 3D buildings from a given grammar.

Chapter 3

Technology Used

We would like our grammar parser to be easily accessible to the user. It should be as simple as providing a text file containing the grammar, pressing a button, and getting back a 3D model of the generated building(s). At first, we explored the option of integrating our building generator into Blender, a popular 3D modelling application. Blender provides a Python API allowing developers to create custom add-ons. This seemed like a good option as we thought it would also save us having to write our own geometry library for dealing with 3D objects, since Blender could provide this functionality. After experimenting with Blender integration, it became clear that instead of simplifying the process, it added various complications. Blender had unimplemented functions in its API as it was still in development and it was hard to ensure high cohesion within our code as we were constantly working around Blender's own code. Figures 3.1 and 3.2 show our experimentations with Blender.

Instead of integrating with Blender, we decided the output of running a grammar would be a .obj file. This file could then be easily imported into the user's 3D modelling software of choice. Avoiding Blender meant we needed to create our own library for processing 3D geometry. The main data structure that we needed to be able to represent and manipulate was an oriented bounding box in 3D space. We experimented with a few of Python's geometry libraries such as *sympy.vector* and its *CoordSys3D* class. Although a useful tool, it became clear later in the project that many of the functions we were using to manipulate a coordinate system were slow and not designed to be applied to a large number of objects. Instead, we settled on implementing our own classes, making use of lower-level libraries such as *NumPy* to carry out matrix transformations on large numbers of vertices.

To assist in the writing of the grammar parser, we used the Python library SLY (Sly Lex Yacc) after exploring various other options. SLY provides intuitive Lexer and Parser classes that are used to define the symbols and syntax of your language. Prior knowledge of Lex/Yacc made implementing our parser a simple and enjoyable process. Other parsing libraries considered, such as *Lark*, provided sophisticated features and utilities that were not necessary for our simple parser. The speed of the parser was not a concern since we were not planning on parsing large programs with complicated constructs. In the next chapter we will describe the grammar we have created.

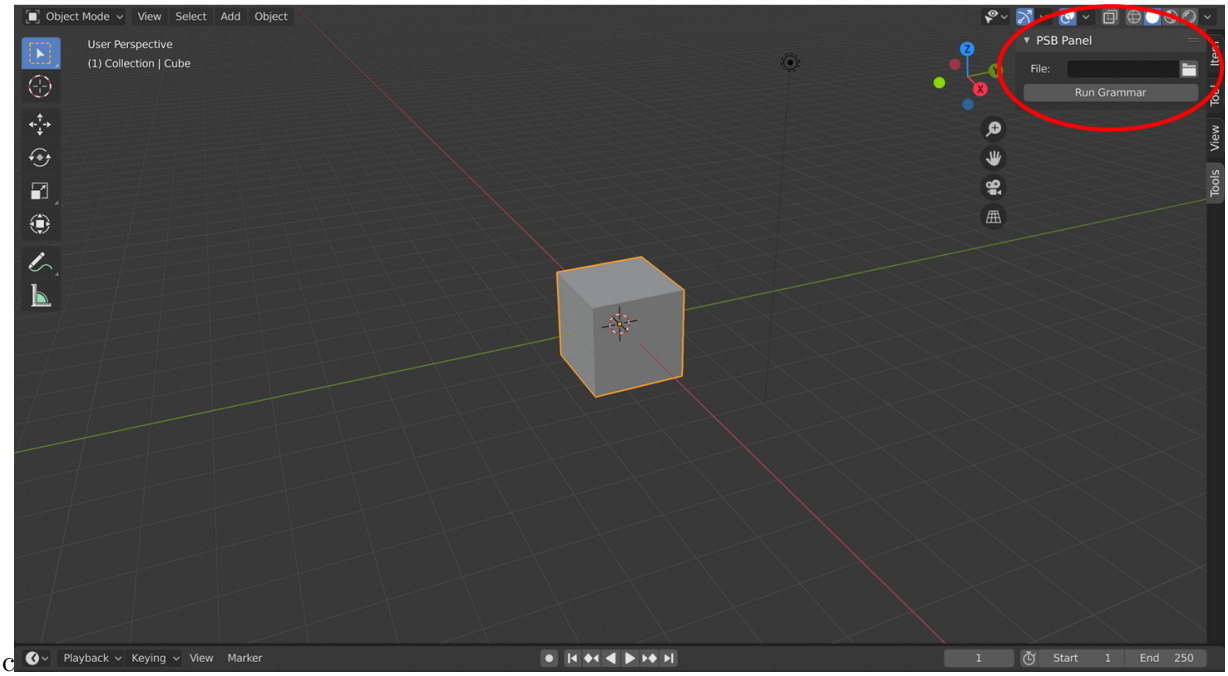


Figure 3.1: Our legacy Blender add-on was called PSB (Procedural Structures for Blender). The idea was that a user could select an object in their Blender scene and run a grammar on that object to create a building. The building would fill the volume given by the selected object.

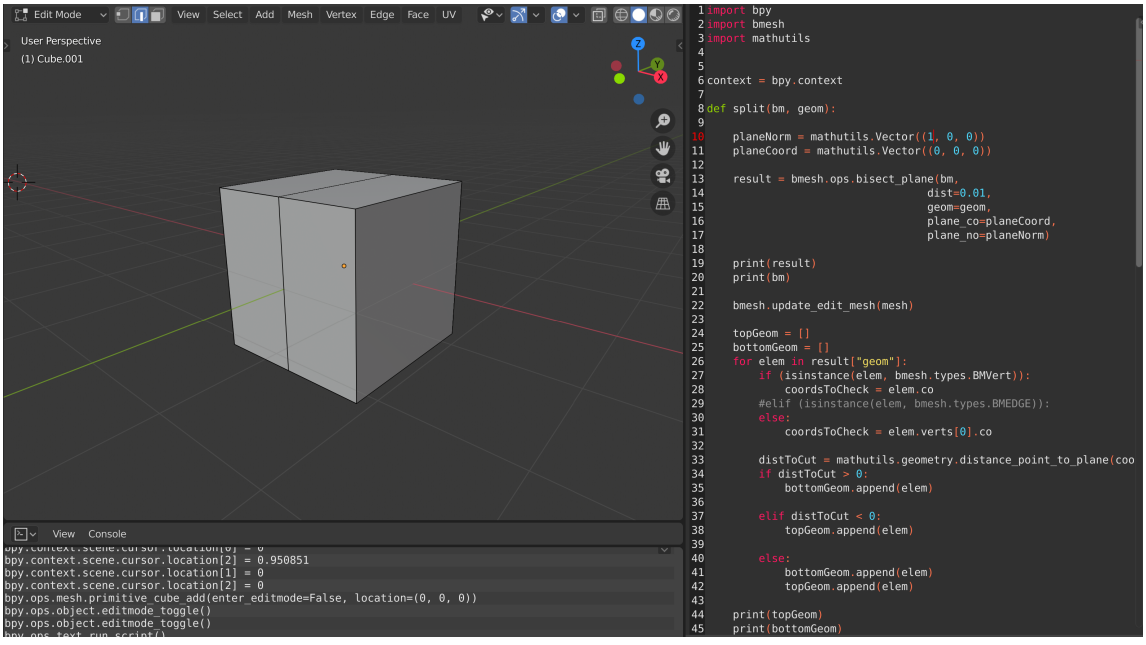


Figure 3.2: We successfully implemented a split operation using Blender's `bisect_plane` function (the box in the image is cut along the x-axis). However, we wanted it to be easy to carry out further operations on each half of the split object. Unimplemented functionality within Blender's API made this difficult.

Chapter 4

Representing Buildings as Grammars

4.1 Grammar Definition

The grammar we use to represent buildings is inspired by *CGA Shape*. Our **building grammar** is similar to a context-free grammar but allows for rules to have parameters, conditions and priorities. Before we describe our building grammar, a few definitions:

Definition 1. *A **scope** is an oriented bounding box in 3D space, defined by its size, position and rotation on each axis. Intuitively it can be thought of as a plot in which a building will be constructed. S denotes the set of all scopes.*

Definition 2. *For a set A, we define A^* to be the set of sequences such that each element of a sequence in A^* is an element of A. I.e. $A^* = \{(a_1, \ldots, a_n) \mid n \in \mathbb{N},\ a_i \in A\}$*

Definition 3. *We use G to denote the infinite set of all geometric objects in 3D space. We can think of elements of G as sets of vertices, edges and faces.*

Formally, a **building grammar** H is a 5-tuple (L, l_0, P, O, R) where:

- L is a finite set of rule **labels**
- $l_0 \in L$ is the label of the **start rule**
- $P \subset G$ is a finite set of **primitive objects**. The primitive objects of a simple house might be a window, door, cuboid and roof. We can then use **operations** to manipulate and combine these primitive objects to create a building (see Figure 4.1).
- $O \subseteq (S \to S^*) \times (G^* \to G)$ is a finite set of **operations**. Each operation consists of two functions - a scope-changing function of type $S \to S^*$ and a combining function of type $G^* \to G$.
- $R \subseteq L \times O \times (L \cup P)^* \times \mathbb{R} \times \Sigma \times E$ is a finite set of **rules**. Here, Σ is just the set of all strings and E is the set of all boolean expressions. A **rule** is a 6-tuple $(l, op, C, p, args, cond)$ where:
  - $l \in L$ is the rule's **label**
  - $op \in O$ is the rule's **operation**

– $C \in (L \cup P)^*$ is an ordered sequence of **labels** and **primitive objects** representing the rule's **children**

– $p \in \mathbb{R}$ is the relative **priority** of this rule compared to other rules with the same label. It is important to note that this is a priority rather than a probability. The priorities of rules with the same label may add up to some value other than 1. The priorities are proportional to the probability that the rule is chosen.

– $args \in \Sigma$ is a list of parameters that are passed to the rule.

– $cond \in E$ is a condition that must evaluate to true for a rule to be considered for execution. It is a boolean expression that may contain the rule's parameter names.

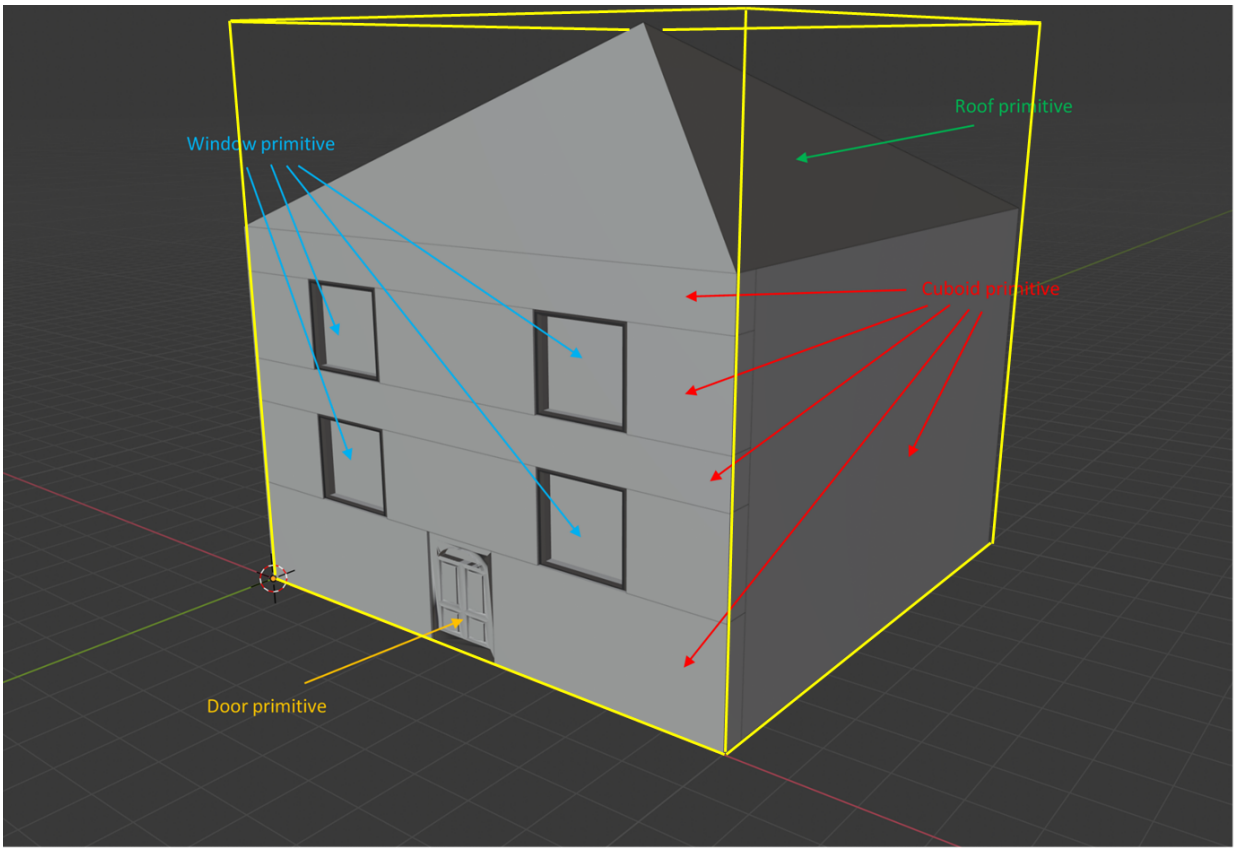


Figure 4.1: An example building with labelled primitive objects. The yellow box indicates the **scope** in which the building was generated.

We now recursively define how to **run** a rule label l in a given scope $s \in S$:

1. Let $R_l = \{(l', op, C, p, ps, cond) \in R \mid l' = l \wedge cond \text{ evaulates to } true \text{ given } ps\}$ be the set of all rules with label l.
2. Let p_{sum} be the sum of the priorities of all rules in R_1. Choose a rule $r = (l, op, C, p, ps, cond)$ from R_1 such that the probability of choosing r is $\frac{p}{p_{sum}}$.
3. Let $(f, g) = op$ so that f is the scope-changing function of op and g is the combining function.
4. Let $T = f(s)$ be the sequence of scopes generated by running the scope-changing function f

on the given scope s. These are the child scopes of *op*.

5. Construct a sequence of geometric objects $C' \in G^*$ by running each child operation in C on its corresponding scope in T. For primitive children, we simply fill the corresponding scope with the primitive shape:

$$C'_i = \begin{cases} C_i \text{ run in scope } T_i & \text{if } C_i \in L \\ C_i & \text{if } C_i \in P \end{cases}$$

6. The final geometry given by running l in scope s is calculated by combining the results of running the children in their scopes: $g(C')$.

Given a grammar $H = (L, l_0, P, O, R)$, we can **run** H in a scope s by running l_0 on s.

Although this formal definition is useful, it is not the most intuitive way to describe a grammar. In the next section we will describe a syntax that can be used to write the rules of a grammar.

4.2 Grammar Syntax

To make it easier to write a grammar, we will now outline the syntax and built-in operations that can be compiled by our parser.

A rule in the grammar has the following format:

$$label(params) : cond \longrightarrow operation : priority$$

Where:

- *label* is the rule label. As in context-free grammars, multiple rules may share a label.
- *params* is a set of optional parameters to the rule. When a rule is passed parameters, the parameters act like local variables that can be appear in *cond*, *operation* and *priority*.
- *cond* is an optional condition for the rule. It must evaluate to true for the rule to be considered for execution.
- *priority* is an optional priority that this rule is run relative to other rules with the same label.

As mentioned in our formal definition, a rule runs on an oriented bounding box in 3D space, called a scope. Given an initial scope and starting rule we can run a grammar. Intuitively, running a grammar works similarly to context-free grammars - wherever a label L appears on the right hand side of a rule, we replace L with the right hand side of a rule labelled with L.

Next, we will describe the different operations that can appear on the right hand side of a rule.

4.3 Operations

4.3.1 Primitive

The primitive operation $I(primitive\_name)$ can be thought of as the terminal symbols of other types of grammar.

$$I(prim\_name) \quad \text{represents the insertion of the object } prim\_name$$

When run on a scope S, this rule will place the primitive shape named *prim_name* inside the bounding box defined by S.

4.3.2 Split

The *split* operation has the following syntax:

$$split(axis)\{size_1 : op_1 \mid size_2 : op_2 \mid \ \ldots \ \mid size_n : op_n\}$$

Note, we will interpret the z-axis as upwards. Given a scope S, the *split* operation splits the scope along $axis$ into n new scopes where the size of $scope_i$ along the axis given by $axis$ is $size_i$. Each op_i is then run on its corresponding $scope_i$. A *split* operation using absolute sizes in this way has the following requirement:

$$\sum_{i=1}^{n} size_i = S.size(axis)$$

Alternatively, we can make sizes relative using the prefix $\sim$. The following example splits the given scope in half in the x-direction (regardless of the given scope) and fills each half with the primitive shape called *rect* (a cuboid):

$$split(x)\{\sim 1 : I(rect) \mid \ \sim 1 : I(rect)\}$$

We can also mix relative and absolute sizes. Given a *split* operation running on scope S, along axis $axis$, with absolute sizes s_1 to s_n and relative sizes r_1 to r_m, we can calculate the actual split sizes a_1 to a_m corresponding to each relative size as follows:

$$a_i = \frac{r_i}{\sum_{i=1}^{m} r_m} \left(S.size(axis) - \sum_{i=1}^{n} s_n \right)$$

4.3.3 Labels as Operations

A rule label may also take the place of an operation. When the grammar is run, a rule label in place of an operation is replaced with the right hand side of a rule matching that label.

$$\begin{aligned} mySplitRule &\longrightarrow split(x)\{\sim 1 : leftSide \mid \ \sim 1 : rightSide\} \\ leftSide &\longrightarrow I(rect) \\ rightSide &\longrightarrow I(triangle) \end{aligned} \tag{4.1}$$

The grammar above splits the given scope in half in the x-direction, filling the left half of the split with a cuboid and the right side with a triangle. We can adapt this grammar to instead recursively split the right side of the sub-division:

$$\begin{aligned} mySplitRule &\longrightarrow split(x)\{\sim 1 : I(rect) \mid \ \sim 1 : maybeSplit\} \\ maybeSplit &\longrightarrow I(rect) : 0.5 \\ maybeSplit &\longrightarrow mySplitRule : 0.5 \end{aligned} \tag{4.2}$$

Just as in grammar 4.1, the left side of the initial split is filled with a cuboid but now the right side is recursively split. The right half is split again with probability 0.5, otherwise the recursion terminates and the right side is filled with a cuboid. This non-deterministic grammar can create different objects on each run (as shown in Figure 4.2).

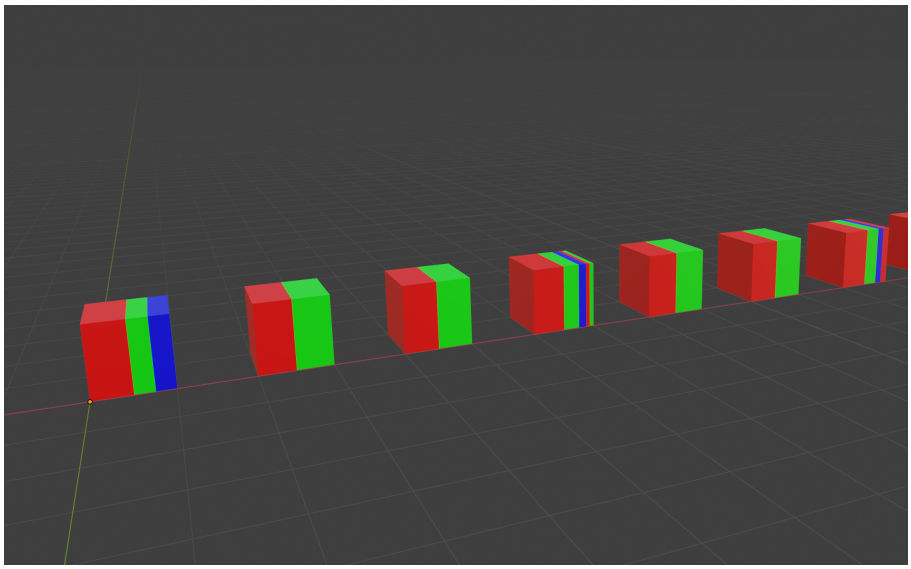


Figure 4.2: Examples of objects created by grammar 4.2. The colours have been manually added to make the split sections clearer. The objects vary due to the non-deterministic nature of the grammar.

4.3.4 Repeat

We have included two forms of repeat operation. Both split the given scope along the specified axis into equally-sized child scopes and then run an operation on each of these scopes.

$$repeat(axis, size)\{op\} \tag{4.3}$$

$$repeatN(axis, N)\{op\} \tag{4.4}$$

The first version makes each child scope have the given *size* along the given *axis* and repeats as many times as necessary to fill the scope. This is the version of repeat used in *CGA Shape*. The second version (which from experience seems to be more useful) splits the scope into N equally-sized child scopes. Note that in operation 4.3, the actual scope size will be adjusted so that the provided scope is completely filled. When run on a scope S, we have the following equivalence between the repeat operations:

$$repeat(axis, size)\{op\} = repeatN\left(axis, \left\lceil \frac{S.size(axis)}{size} \right\rceil\right)\{op\}$$

4.3.5 Nil

The nil operation essentially does nothing. When run on a scope, it leaves that box of space empty. The example below generates a house with a garage on the side (as shown in Figure 4.3). The *nil* operation is used to represent the empty space above the garage. Variations of this house-with-garage example will be used throughout our report.

$$\begin{aligned} start &\longrightarrow split(x)\{\sim 2 : house \mid \sim 1 : garageSide\} \\ house &\longrightarrow split(z)\{\sim 2 : I(rect) \mid \sim 1 : I(roof)\} \\ garageSide &\longrightarrow split(z)\{\sim 1 : I(rect) \mid \sim 2 : nil\} \end{aligned} \tag{4.5}$$

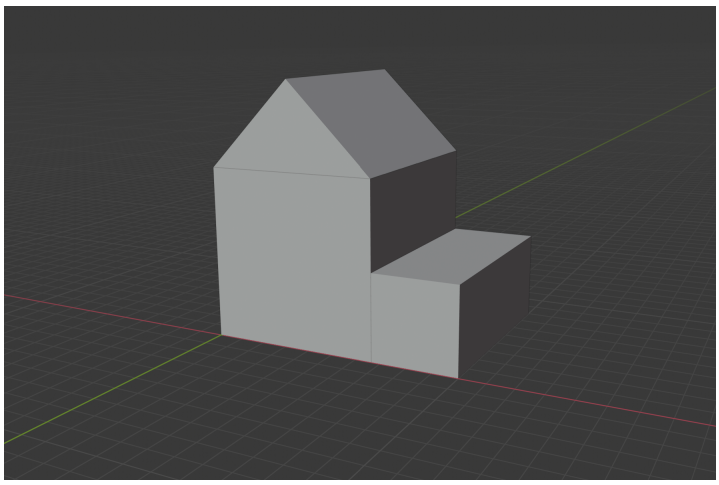


Figure 4.3: The result of running grammar 4.5

4.3.6 Scope-changers

We define the following scope-changing operations:

$T(\delta x, \delta y, \delta z)\{op\}$	represents a translation along each axis
$S(s_x, s_y, s_z)\{op\}$	represents a resizing of each axis of the current scope
$R(axis, \theta)\{op\}$	represents a rotation around $axis$ by θ radians

These operations are useful but have the problem that they can introduce overlapping geometry. Grammar 4.6 shows an example of this. Given an initial scope of a 4x4x4 cube, the grammar splits the cube in the x-direction into two equally-sized scopes. The first of these scopes is then translated in the x-direction by 1 unit before being filled with a cuboid. The second scope is just filled with a cuboid. This results in the first cuboid overlapping the second (see Figure 4.4).

$$plot \longrightarrow split(x)\{2 : T(1,0,0)\{I(rect)\} \mid 2 : I(rect)\} \tag{4.6}$$

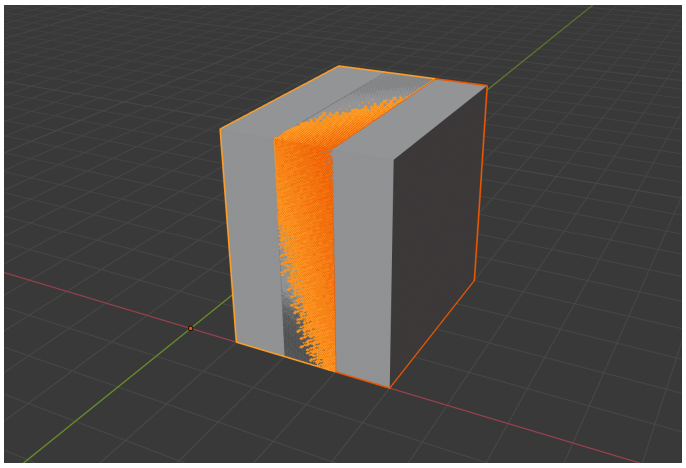


Figure 4.4: The result of running grammar 4.6. The two cuboids overlap. The initial scope that the grammar was run on was a 4x4x4 cube with its corner at the origin. The artefacts you see are due to the overlapping objects.

There are two ways to address the problem of overlapping geometry. We can leave it up to the grammar-writer to ensure the grammar can not produce overlaps. Or we can introduce the ability to check for overlapping geometry in the grammar. This second method is what *CGA Shape* does using occlusion checks. For this project we will not implement these checks and will instead trust the grammar-writer has not introduced overlaps. This makes the reverse-engineering of a grammar less challenging.

4.3.7 Parameterisation, Conditions and Constants

Rules may be parameterised and can contain conditions. In a grammar file, constants may be defined before the rule definitions begin. In the grammar below we pass an *axis* parameter to the wall operation that determines the axis along which to repeat a sequence of windows.

$$
\begin{aligned}
wallSize &= 1 \\
numWindows &= 3 \\
walledHouse &\longrightarrow split(z)\{\sim 2 : walls \mid \sim 1 : I(roof)\} \\
walls &\longrightarrow split(x)\{wallSize : wall(y,0) \mid \sim 1 : walls2 \mid wallSize : wall(y,1)\} \\
walls2 &\longrightarrow split(y)\{wallSize : wall(x,0) \mid \sim 1 : nil \mid wallSize : wall(x,1)\} \\
wall(axis, wStyle) &\longrightarrow repeatN(axis, numWindows)\{window(wStyle)\} \\
window(s) : s == 1 &\longrightarrow I(window1) \\
window(s) : s == 2 &\longrightarrow I(window2)
\end{aligned}
$$

Paramaterised rules and conditions also allow for rules to act like a loop by using recursion. In the following example, $repN(axis, N)$ is equivalent to the operation $repeatN(axis, N)\{I(rect)\}$.

$$
\begin{aligned}
repN(axis, N) : N > 0 &\longrightarrow split(axis)\{\sim 1 : I(rect) \mid \sim (N-1) : repN(axis, N-1)\} \\
repN(axis, N) : N \leq 0 &\longrightarrow nil
\end{aligned}
$$

4.3.8 Randomness

So far, the only way to introduce randomness in a grammar is to have multiple rules with the same label and to associate a priority to each one. In our house-with-garage example (grammar 4.5), we might want the garage to be located on either the left or right according to priorities. We might also want to add variation in the height of the garage. To allow for this second kind of randomness, we introduce the function $rand(a, b)$ into the grammar syntax. Whenever $rand(a, b)$ is evaluated while running a grammar, it will return a random number uniform in $[a, b]$. Below is the grammar for a house with a random-height garage on either the left or right side.

$$\begin{aligned} start &\longrightarrow split(x)\{\sim 2 : house \mid \sim 1 : garageSide\} : 0.5 \\ start &\longrightarrow split(x)\{\sim 1 : garageSide \mid \sim 2 : house\} : 0.5 \\ house &\longrightarrow split(z)\{\sim 2 : I(rect) \mid \sim 1 : I(roof)\} \\ garageSide &\longrightarrow split(z)\{\sim (rand(0.75, 1.25) : I(rect) \mid \sim 2 : nil\} \end{aligned} \tag{4.7}$$

In addition to $rand(a, b)$, the function $randint(a, b)$ returns a random integer in the range a to b (inclusive of both a and b).

This concludes our description of the syntax of the grammar. There are a couple of unimportant operations that we have not introduced here and it is worth noting that our grammar is very adaptable so the syntax for a new operation could easily be added in the future. Next we will discuss how a grammar is represented internally once it has been parsed.

Chapter 5

From Grammars to Operation Graphs

5.1 Operation Graphs

Now that we have defined the syntax of a grammar, we need an internal representation. A grammar can be thought of as a graph where nodes represent operations. An edge from operation A to operation B exists when B is a child operation of A. The leaves of an operation graph are primitive shapes (I operations). Edges in a graph may be labelled with parameters - like the sizes corresponding to each child operation of a split. Figure 5.1 shows an operation graph which represents the simple house-with-garage example. The grammar corresponding to this operation graph is repeated below. Since there is no recursion in this particular example, the operation graph is directed and acyclic (a DAG).

$$\begin{aligned} start &\longrightarrow split(x)\{\sim 2 : house \mid \sim 1 : garageSide\} \\ house &\longrightarrow split(z)\{\sim 2 : I(rect) \mid \sim 1 : I(roof)\} \\ garageSide &\longrightarrow split(z)\{\sim 1 : I(rect) \mid \sim 2 : nil\} \end{aligned}$$

If each label of our grammar corresponded to just one operation then we could form an operation graph for the grammar consisting only of the operations we have described in the syntax. However, a single label L may correspond to many rules, each of which have a different operation. In this case, we introduce a new *choose* operation that represents L. The child operations of the new *choose* operation are the operations appearing on the right hand side of the rules labelled with L. When a *choose* operation is run, it chooses one of its child operations according to their relative priorities and conditions. Figure 5.2 represents grammar 4.2 (repeated below) and demonstrates this choose operation.

$$\begin{aligned} mySplitRule &\longrightarrow split(x)\{\sim 1 : I(rect) \mid \sim 1 : maybeSplit\} \\ maybeSplit &\longrightarrow I(rect) : 0.5 \\ maybeSplit &\longrightarrow mySplitRule : 0.5 \end{aligned}$$

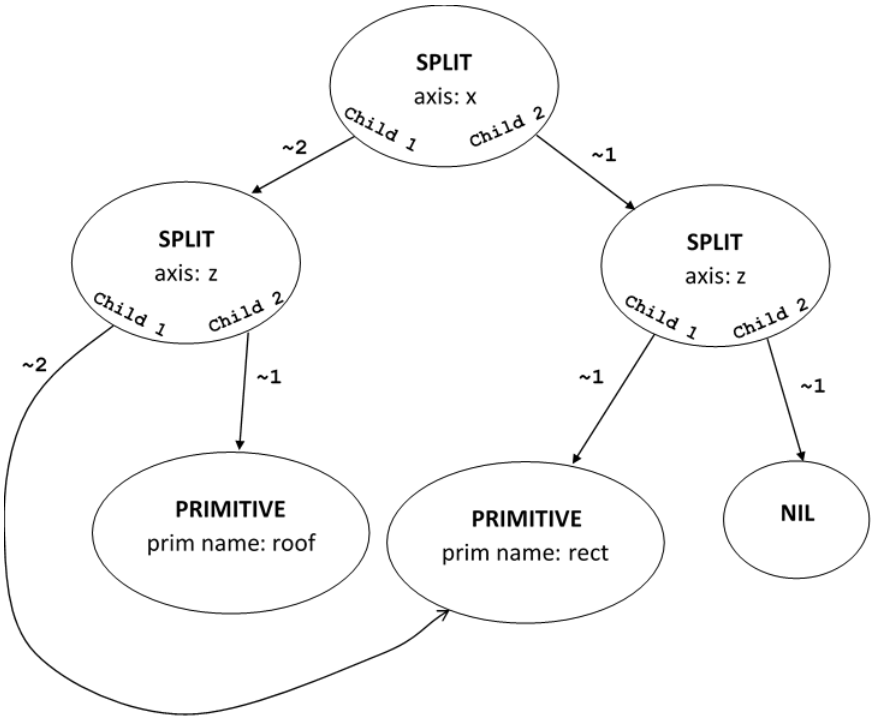


Figure 5.1: The operation graph for the house-with-garage example. In our code, the nodes an operation graph are instances of operation classes. The edges in the graph represent pointers from parent operation objects to their children. The parameters of nodes (such as the axis or primitive object name) are the attributes of the class.

We will now discuss how these operation graphs are represented in our code.

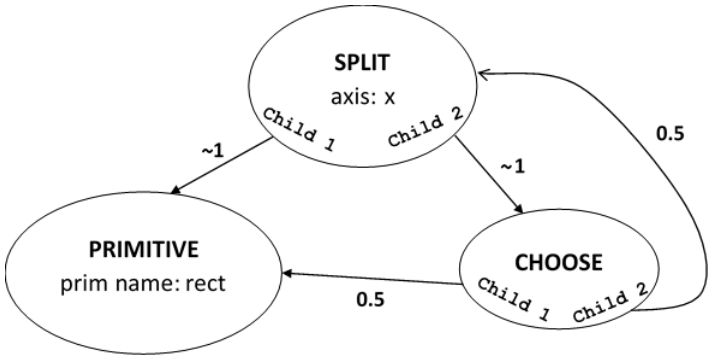


Figure 5.2: The operation graph for the recursive split example. Notice how the *choose* operation is used to choose between operations with the same label.

5.2 Representing an Operation

In our code, each operation has its own class and all operation classes inherit some base attributes and methods from the `Op` class. The most important of these attributes are an operation's own arguments, `args`, its child operations, `childOps`, and the arguments corresponding to each child, `perChildArgs`. As an example, for the *split* operation, `args` contains the axis to split along, `childOps` contains a list of the operations to run on each section of the split, and `perChildArgs` contains a list of sizes corresponding to each split section.

The most important method that must be implemented by any operation class extending `Op` is the `run` function. This defines what happens when the operation is run on a given scope. The following pseudo-code gives an outline of what an operation's `run` function might do:

```
CLASS SomeOp EXTENDS Op{
  ...
  FUNCTION run(context, scope, environment){
    possibly modify context by adding new geometry
    possibly modify the environment by assigning new values to variables
    possibly run child operations, perhaps passing a new scope/environment
  }
  ...
}
```

The `run` function takes three parameters:

- `context` contains the mutable geometry of the building we are generating. An operation may add new geometry to this context. The only operation we have discussed that does this is the primitive operation represented by $I(prim\_name)$ in the syntax.
- `scope` contains the immutable scope which this operation should run in. As a reminder, a scope is an oriented bounding box in 3D space. The operation can't modify the scope but can pass new scopes to its child operations.
- `environment` contains an assignment of variable names to values. If an operation is parameterised, the values corresponding to its parameters will be in `environment`.

5.2.1 Split Operation

The run function for `OpSplit` is a good example. Below is its Python implementation:

```
1 class OpSplit(Op):
2     ...
3     def run(self, context, scope, env):
4         axis, sizes = self.args.evaluate(env)
5         childScopes = scope.split(axis, sizes)
6         for childOp, childScope in zip(self.childOps, childScopes):
7             childOp.run(context, childScope, env)
8     ...
```

The split operation first evaluates its arguments in the current variable environment (line 4). Then the scope for each child operation is calculated (line 5). Finally, each of the child operations is run on its corresponding scope (lines 6-7).

5.2.2 Choose Operation

The run function for the choose operation first checks any conditions associated with its child operations and then chooses one based on their priorities.

```
class OpChooseRuleWithPriority(Op):
    ...
    def chooseChild(self, env):
        priorities, conds = self.args.evaluate(env)[0]
        # remove children whose conditions fail
        # remember index of the valid children in the original list
        validPriorities = [(i,p) for i,p in
                           enumerate(priorities) if conds[i]]
        # Pick a random number less than the max cumulative priority
        rand = random() * sum(x[1] for x in validPriorities)
        i = 0
        cumulativePriority = validPriorities[0][1]
        while cumulativePriority < rand:
            i += 1
            cumulativePriority += validPriorities[i][1]
        return self.childOps[validPriorities[i][0]]

    def run(self, context, scope, env):
        # Choose child using priorities and conditions then run it
        self.chooseChild(env).run(context, scope, env)
    ...
```

5.2.3 SetParams Operation

We introduce a new *SetParams* operation which is used whenever an operation is passed parameters. A *SetParams* operation has a single child - the operation being passed the parameters. Its run function simply evaluates the values being passed in the current variable environment and then updates the environment with the parameters' values.

```
class OpSetParams(Op):
    ...
    def run(self, context, scope, env):
        paramVals = self.args.evaluate(env)
        child = self.childOps[0]
        envNew = env.copy()
        envNew.update(zip(child.paramNames, paramVals))
        child.run(context, scope, envNew)
```

5.2.4 Primitive Operation

The primitive I operation adds a primitive shape to the context.

```
class OpPrimitive(Op):
    ...
    def run(self, context, scope, env):
        prim = self.args[0]
        context.addPrim(prim, scope)
    ...
```

It is up to *context* to decide exactly what adding a primitive shape does. It could simply print an output saying 'Adding primitive *prim_name* within bounding box given by *scope*'.This implementation might be handy for testing but a more useful implementation of *context* puts the generated geometry of an operation graph into a 3D object file. We will discuss this implementation in the next chapter.

Chapter 6

From Operation Graphs to Geometry

We want the final output of the forward model to be 3D objects that could be used by an artist. After experimenting with a few 3D file formats, we settled on .obj. This file type is commonly used and can be imported into most popular 3D modelling software (Blender, MeshLab, Maya etc.). The final stage of the forward model is to generate a .obj file from an operation graph.

As we saw in the previous chapter, when an operation's `run` function is called it acts on a given *scope* (a bounding box in 3D space). A scope is an instantiation of the *Scope* class whose main attributes are size, position and orientation. When an operation such as *OpSplit* is run, it calls a corresponding function in the *Scope* class that returns the appropriate child scopes:

```
# Split this scope along axis into sections with sizes given by
# splitSizes. Returns a scope for each split section.
def split(self, axis, splitSizes):
        newScopes = []
        pos = self.pos
        rot = self.rotMat
        axisVector = self.axisVector[axis]
        unchangedSizes = self.size * (1-axisVector)
        offset = np.array([0,0,0])
        # Calculate the scale factor for any relative sizes
        scale = self.calcScale(axis, splitSizes)

        for s in splitSizes:
                # Scale if it's a relative size
                size = s.size * scale if s.isRelative else s.size
                actualSizeVec = size * axisVector
                sizeVec = actualSizeVec + unchangedSizes
                tVec = rot.dot(offset)
                # Create a new scope for this split section
                newScopes.append(Scope(pos+tVec, rot, sizeVec))
                offset = offset + actualSizeVec
        return newScopes
```

When a primitive operation is run, instead of manipulating the scope, we call the *addPrim* method of the *context*:

```
class OpPrimitive(Op):
    ...
    def run(self, context, scope, env):
        prim = self.args[0]
        context.addPrim(prim, scope)
    ...
```

The *addPrim* function of *context* transforms the vertices of the primitive shape to the given scope (line 13 below) and then writes the geometry data to the object file that represents our final model.

```
1  class ContextOBJ:
2
3      # Create a new context with prims being the known primitives
4      def __init__(self, prims=basicPrims):
5          # Start with an empty file
6          self.objFileText = ""
7          self.prims = prims
8          self.vertCount = 0
9
10     def addPrim(self, primName, scope):
11         prim = self.prims[primName]
12         # Transform the vertices into the given scope
13         newVerts = scope.putVertsInScope(prim.verts, prim.box)
14         # Add the new data to the object file we are building
15         self.objFileText += f"o {primName}\nv "
16         self.objFileText += self.vertsToObjText(newVerts)
17         self.objFileText += f"\n{prim.otherData}"
18         # Add the prim's face data but make sure the faces reference
19         # the correct vertices by offsetting the vertex indices
20         self.objFileText += self.offsetVertIndices(prim.faceData)
21         self.vertCount += prim.numVerts
```

On line 432 above we made use of the *putVertsInScope* method of the *Scope* class. This uses a scope's position, size and rotation to transform a primitive object from its current bounding box to the scope:

```
    # Return the transformation matrix represented by this scope
    def toMat(self):
        scaleReCentreTranslate =
          np.array([
            [self.size[0], 0, 0, self.pos[0] + self.size[0]/2],
            [0, self.size[1], 0, self.pos[1] + self.size[1]/2],
            [0, 0, self.size[2], self.pos[2] + self.size[2]/2],
            [0, 0, 0, 1]
          ])
```

```
        r = self.rotMat
        rotate = np.array([[r[0][0], r[0][1], r[0][2], 0],
                           [r[1][0], r[1][1], r[1][2], 0],
                           [r[2][0], r[2][1], r[2][2], 0],
                           [0, 0, 0, 1]])
        return np.matmul(scaleReCentreTranslate, rotate)

    # Move vertices from their current bounding box to this scope
    def putVertsInScope(self, verts, box):
        transMat = np.matmul(self.toMat(), box.matToUnitSquare())
        return np.matmul(verts, transMat.swapaxes(0,1))
```

Creating an object file from an operation graph is now as simple as running the first operation in the graph (the operation corresponding to the start rule of the grammar). This operation will perhaps manipulate the initial scope and run child operations with new scopes. Whenever we run a primitive operation we add to the object file we are constructing. Once our run function terminates we can return the object file to which all our primitive objects have been added. The entire process of generating a 3D object from a grammar file is described by the three functions below:

```
# Given a grammar, write the result of running that grammar
# to an output file
def grammarToObj(self, grammarName, startRule, startScope, outFile):
    opGraph = self.grammarFileToOpGraph(grammarName, startRule)
    self.opGraphToObj(opGraph, startScope, outFile)

# Given a grammar file and start rule,
# return the operation graph corresponding to that grammar
def grammarFileToOpGraph(self, grammarName, startRule):
        print("Creating operation graph from grammar...")
        with open(self.grammarDir + grammarName) as f:
                grammarText = f.read()
        parser = GrammarParser()
        return parser.parse(grammarText)[startRule]

# Given an op graph, run it on the given scope
# and write the output to a file
def opGraphToObj(self, opGraph, startScope, outFile, context):
        context.reset()
        print("Running op graph to create obj")
        opGraph.run(context, startScope, {})
        print("Writing obj to file")
        context.writeToFile(outFile)
```

We will now discuss some examples of buildings that can be generated using the forward model we have described.

Chapter 7

Examples and Extensions

7.1 House with a Garage

We have discussed the simple example of a house with a garage throughout this forward modelling section. As a reminder, the grammar for a simple building with a garage (of varying height) on either the left or right was given by:

$$\begin{aligned} start &\longrightarrow split(x)\{\sim 2 : house \mid \ \sim 1 : garageSide\} : 0.5 \\ start &\longrightarrow split(x)\{\sim 1 : garageSide \mid \ \sim 2 : house\} : 0.5 \\ house &\longrightarrow split(z)\{\sim 2 : I(rect) \mid \ \sim 1 : I(roof)\} \\ garageSide &\longrightarrow split(z)\{\sim (rand/2) + 0.75 : I(rect) \mid \ \sim 2 : nil\} \end{aligned} \tag{7.1}$$

Using our parser, we can use this grammar to generate the 10 buildings shown in Figure 7.1.

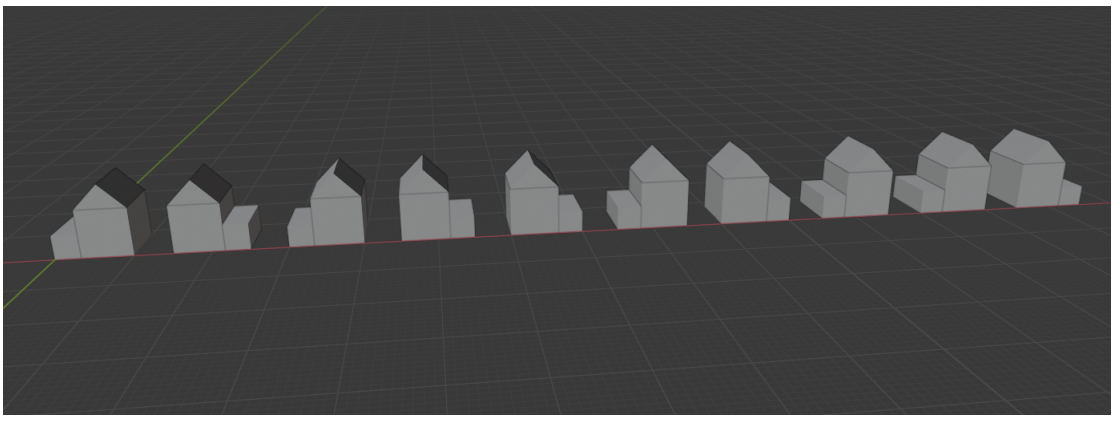


Figure 7.1: 10 buildings generated by grammar 7.1

In grammar 7.1 we make use of the *roof* primitive object to create a gable roof. We could easily have used some other roof object as the primitive shape to create a different style of roof. We could even do a similar thing for the garage and use a garage primitive rather that just a plain box. Figure 7.2 shows objects generated by a grammar that uses a garage primitive.

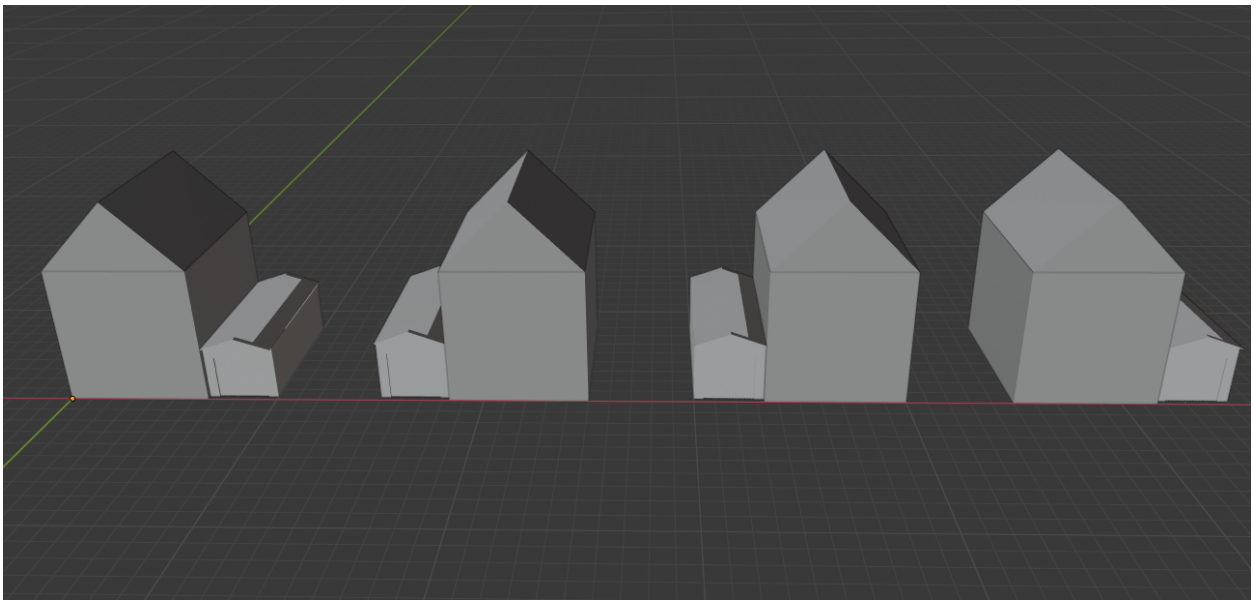


Figure 7.2: Buildings with primitive garage object.

7.2 Better Roofs

In our next example, we will create a set of family homes with more interesting roofs. Using just four primitive shapes (Figure 7.3), we can create the houses with complex roofs shown in Figure 7.4. The start of the grammar for these buildings is shown below. Lines 2-5 split the initial scope along the x-axis into three sections. The centre section is set back on the y-axis and then split in the z-direction to form the main body of the building and the roof (lines 6-10).

```
1  setbackDistance = 1
2  plot --> split(x){
3               ~(rand(0.8,1.2)) : leftSide |
4               ~2 : mid |
5               ~(rand(0.8, 1.2)) : rightSide}
6  mid --> split(y){
7              ~setbackDistance : nil |
8              ~1 : split(z){
9                  ~2 : I(rect) |
10                 ~1 : midRoof}}
11 midRoof --> split(y){
12                 ~1 : R(z,3*pi/2){I(right_angle_triangle)} |
13                 ~1 : R(z,pi/2){I(right_angle_triangle)}}
14 ...
```

One of two things may happen on each side of the house. With probability 0.5 there will be an extrusion from the main house, otherwise it will just be an end to the main section of the house. The options are indicated by the two `leftSide` rules on lines 16 and 21 (each with priority 0.5).

```
15 ...
16 leftSide --> split(y){
17                  ~setbackDistance : nil |
18                  ~1 : split(z){
19                      ~2 : I(rect) |
20                      ~1 : endLeft}} : 0.5
21 leftSide --> split(y){
22                  ~setbackDistance : extrusion |
23                  ~1 : split(z){
24                      ~2 : I(rect) |
25                      ~1 : roofCorner}} : 0.5
26 rightSide --> split(y){
27                   ~setbackDistance : nil |
28                   ~1 : split(z){
29                       ~2 : I(rect) |
30                       ~1 : endRight}} : 0.5
31 rightSide --> split(y){
32                   ~setbackDistance : extrusion |
33                   ~1 : split(z){
34                       ~2 : I(rect) |
35                       ~1 : R(z,3*pi/2){roofCorner}}} : 0.5
36 ...
```

The front-facing end of an extrusion may have two different forms (as indicated by the two different definitions of `endLeft` on lines 50 and 51). These two options can be seen in the buildings in Figure 7.4. The `roofCorner` rule on line 56 combines the three slanted primitives to create a corner (as shown in Figure 7.5). This roof corner can then be rotated and used wherever it's needed (e.g. lines 25 and 35 of the grammar).

```
37 ...
38 extrusion --> split(y){
39                   ~(rand(1,2)) : nil |
40                   ~4 : split(z){
41                       ~2 : I(rect) |
42                       ~1 : split(y){
43                           ~1 : sideFront |
44                           ~3 : sideMid}}}
45 sideMid --> split(x){
46                 ~1 : R(z,pi){I(right_angle_triangle)} |
47                 ~1 : I(right_angle_triangle)}
48 sideFront --> sideMid
49 sideFront --> split(x){~1 : I(corner) | ~1 : R(z,pi/2){I(corner)}}
50 endLeft --> midRoof : 0.5
51 endLeft --> split(y){~1 : I(corner) | ~1 : R(z,3*pi/2){I(corner)}}
52 endRight --> midRoof : 0.5
53 endRight --> split(y){
54                  ~1 : R(z,pi/2){I(corner)} |
55                  ~1 : R(z,pi){I(corner)}}
56 roofCorner --> split(x){
57                    ~1 : split(y){
58                        ~1 : R(z,pi){I(right_angle_triangle)} |
59                        ~1 : R(z,3*pi/2){I(corner)}} |
60                    ~1 : split(y){
61                        ~1 : R(z,pi/2){I(trough)} |
62                        ~1 : R(z,pi/2){I(right_angle_triangle)}}}
```

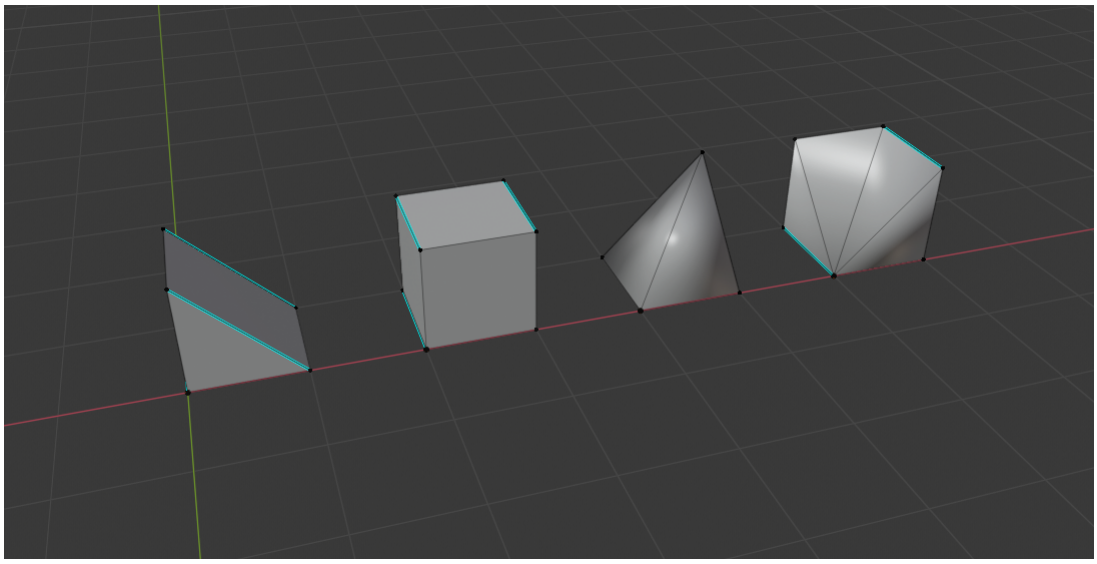


Figure 7.3: The four primitives used to construct the buildings in 7.4.

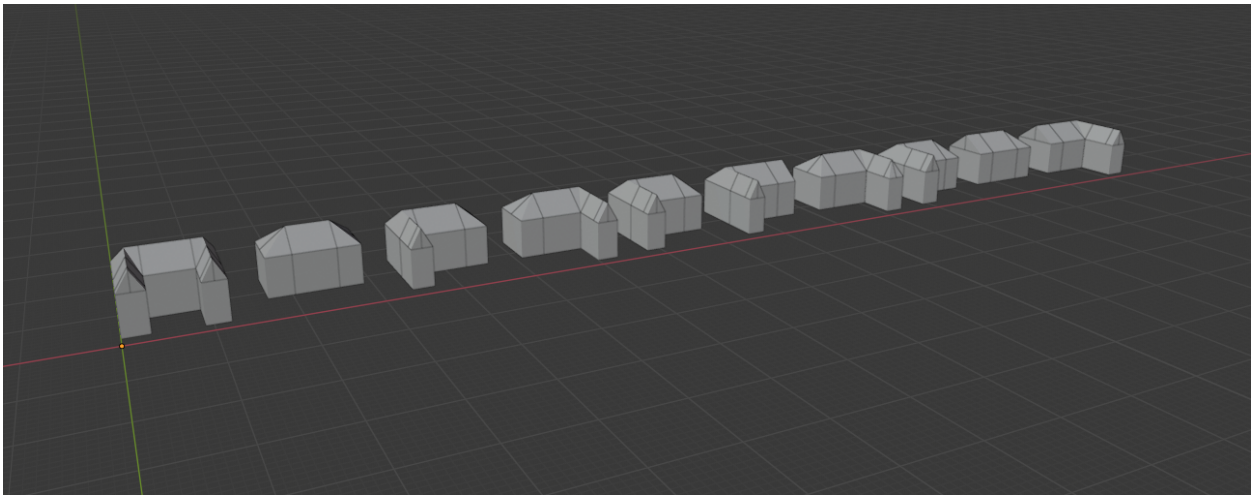


Figure 7.4: Buildings with better roofs.

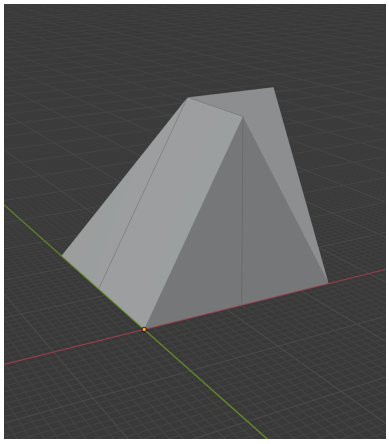


Figure 7.5: The corner of a roof.

7.3 Modern Houses

For this example, we took inspiration from the modern building shown in Figure 7.6 but added some extra floors. This example showcases the ability to create complex buildings from simple primitive shapes. Other than the windows and doors, the only primitive shape used is a cuboid. Even the railings of the balconies are constructed using grammar rules rather than being primitive objects. The buildings generated by our modern house grammar can be seen in Figure 7.7. We will now give an outline of how the grammar is designed.

After defining some constants, the first step of the grammar splits the given plot into a random number of floors (line 12). For all levels other than the bottom level there are two options. Either the level will just be a plain cuboid (line 25) or it will have a balcony (line 26). We have not gone to the trouble of adding windows to the top levels but we could easily have done so if desired.

```
1  wallDepth = 0.7
2  windowDepth = 0.2
3  windowDividerWidith = 0.3
4  doorWidth = 2
5  doorHeight = 4
6  doorDepth = 0.8
7  maxFloors = 5
8  minFloors = 2
9  railingThickness = 0.2
10 railingSegmentLength = 1
11 pillarWidth = 0.6
12 plot --> floors(randint(minFloors,maxFloors))
13 floors(numFloors) --> split(z){
14                          ~1 : base |
15                          ~(10*numFloors) : split(y){
16                              ~1 : split(x){
17                                  ~1 : nil |
18                                  ~20 : house(numFloors)} |
19                              ~1 : nil} |
20                          ~(10*(maxFloors-numFloors)) : nil}
21 house(n) --> split(z){
22                  ~1 : bottomLevel |
23                  ~(n-1) : repeatN(z,n-1){level}}
24 base --> I(rect)
25 level --> I(rect) : 0.5
26 level --> split(x){
27               ~1 : split(y){
28                   ~1 : balcony |
29                   ~(rand(2,3)) : I(rect)} |
30               ~(rand(5,6)) : I(rect)} : 0.5
31 balcony --> split(z){
32                 ~1 : I(rect) |
33                 ~3 : split(x){
34                     railingThickness : railings(y) |
35                     ~1 : split(y){
```

```
36                          railingThickness : railings(x) |
37                          ~1 : nil}} |
38                  ~6 : nil}
39  ...
```

The railings of the balcony are modelled using the *repeat* operation. The `railings` rule on line 41 takes an axis as a parameter so can be used to generate railings in both the x-direction (line 36 above) and y-direction (line 34 above).

```
40  ...
41  railings(ax) --> repeat(ax){railingSegmentLength : railing(ax)}
42  railing(ax) --> split(ax){
43                      ~1 : I(rect) |
44                      ~4 : split(z){
45                          ~7 : nil |
46                          ~1 : I(rect)}}
47  ...
```

For the ground floor of the building we may or may not have a pillar in the corner (lines 55 and 59). In previous grammar examples, and on the upper floors of this building, we just use a cuboid to represent the body of a building so the walls have no thickness. For the ground floor however, we use a different cuboid for each wall to give them thickness so that we can split them up to add windows and a door. The wall rules on lines 75 and 76 take two parameters indicating the orientation of the wall. The first parameter indicates the direction of the length of the wall and the second indicates the direction of the depth (it is assumed that the height is in the z-direction).

```
48  ...
49  bottomLevel --> split(x){
50                      ~10 : split(y){
51                          ~(rand(0,2)) : nil |
52                          ~10 : groundFloor} |
53                      ~(rand(0,2)) : nil}
54  block --> I(rect)
55  groundFloor --> split(x){
56                      ~1 : leftWithPillar |
57                      ~(rand(3,4)) : frontAndBack |
58                      wallDepth : wall(y,x)} : 0.5
59  groundFloor --> split(x){
60                      wallDepth : wall(y,x) |
61                      ~1 : frontAndBack |
62                      wallDepth : wall(y,x)} : 0.5
63  leftWithPillar --> split(y){
64                         ~1 : split(x){
65                             pillarWidth : split(y){
66                                 pillarWidth : I(rect) |
67                                 ~1 : nil} |
68                             ~1 : nil |
69                             wallDepth : doorBit} |
70                         ~(rand(1.5,2.5)) : I(rect)}
```

```
71  frontAndBack --> split(y){
72                       wallDepth : frontWall |
73                       ~1 : nil |
74                       wallDepth : wall(x,y)}
75  wall(ax1, ax2) --> I(rect) : 0.2
76  wall(ax1, ax2) --> split(ax1){
77                         ~(rand(1,2)) : I(rect) |
78                         ~1.5 : split(z){
79                             ~1 : I(rect) |
80                             ~2 : windows(ax1, ax2) |
81                             ~1 : I(rect)} |
82                         ~(rand(1,2)) : I(rect)} : 0.8
83  ...
```

For the windows we make use of a primitive object (line 92) and repeat it a random number of times (line 85) with a divider between each one (line 87). The door of the building is also represented by a primitive object (line 100).

```
84  ...
85  windows(ax1, ax2) --> repeatN(ax1, randint(1,4)){window(ax1, ax2)}
86  window(ax1, ax2) --> split(ax1){
87                           windowDividerWidith : I(rect) |
88                           ~1 : split(ax2){
89                               ~1 : nil |
90                               windowDepth : wind(ax1) |
91                               ~1 : nil}}
92  wind(ax) : ax == y --> I(window3)
93  wind(ax) : ax != y --> R(z,pi/2){I(window3)}
94  frontWall --> I(rect)
95  doorBit --> split(y){
96                  ~1 : I(rect) |
97                  doorWidth : split(z){
98                      doorHeight : split(x){
99                          ~1 : nil |
100                         doorDepth : R(z,pi/2){I(door)} | ~1 : nil} |
101                 ~1 : I(rect)} |
102                 ~1 : I(rect)}
```

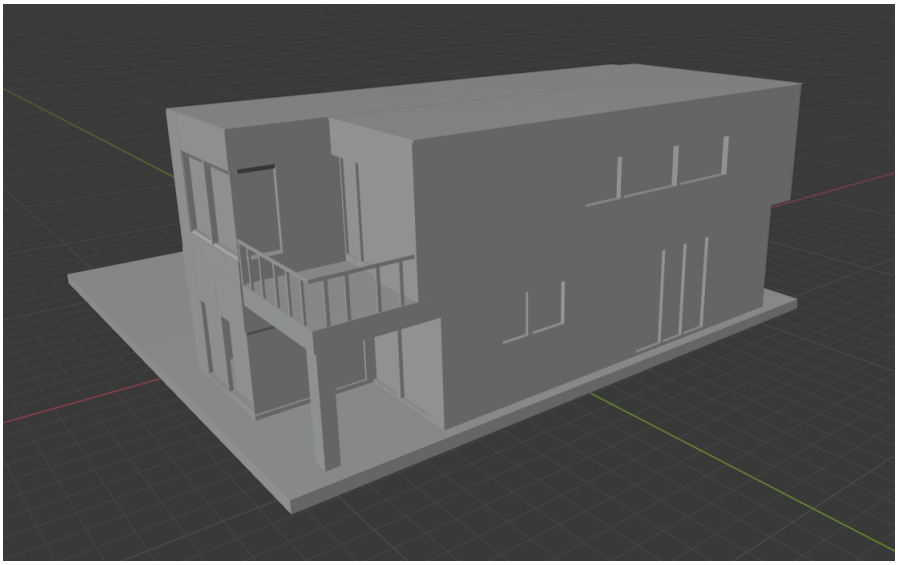


Figure 7.6: A modern building design which our grammar was inspired by. The 3D model for this building is available for free on TurboSquid[1].

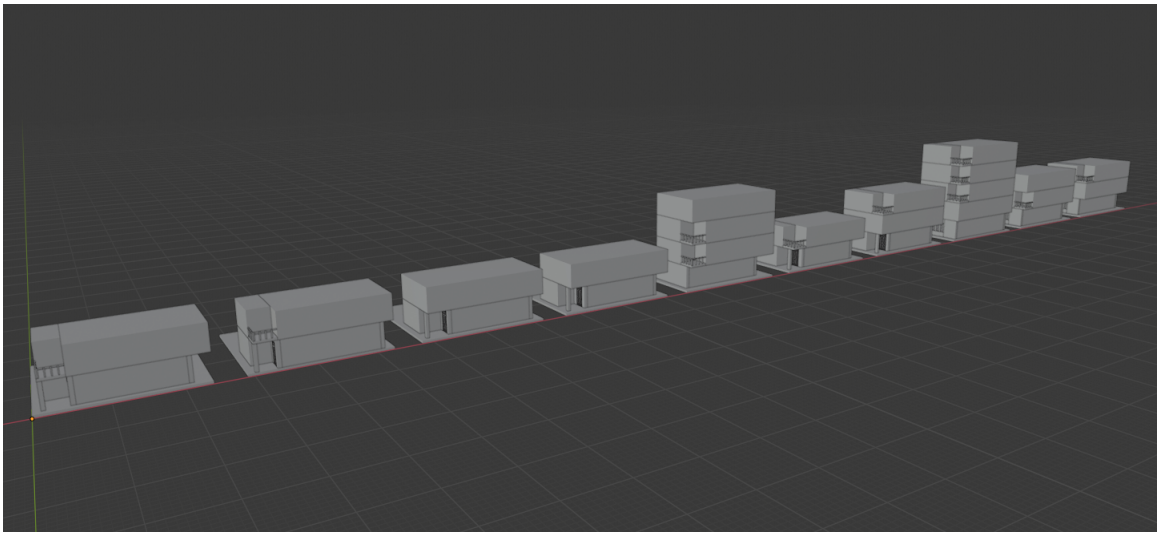


Figure 7.7: A set of generated modern buildings.

[1] https://www.turbosquid.com/3d-models/3d-model-house-modern-1204626

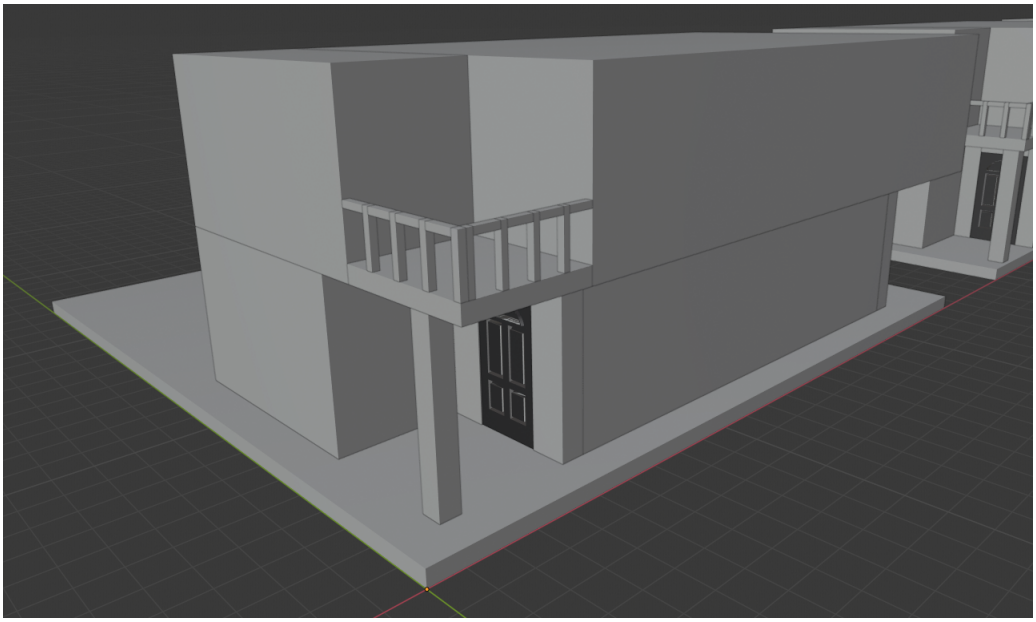


Figure 7.8: The balcony here is represented explicitly in our grammar using a repeat rule rather than being an imported primitive shape. The door beneath the balcony is a primitive object.

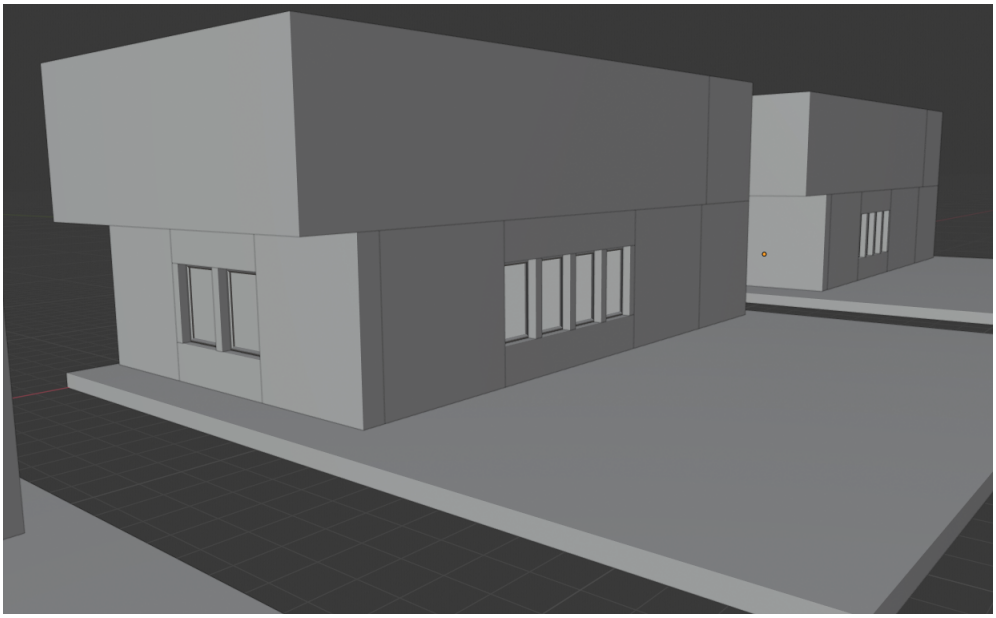


Figure 7.9: Different buildings have different numbers of windows and the windows vary in size across buildings. The windows are primitive objects.

7.4 Skyscrapers

Although our building grammar is mainly designed to model the structure of specific buildings, in this example we will also model the layout of the buildings. Instead of filling a plot with a single building, we will instead fill it with a whole city. Figure 7.10 shows the output of the grammar we will now describe. In Figure 7.11 we have shown that once textures have been added to the generated buildings, it can create quite a realistic city. We wrote the grammar for the city in just a couple of hours and recorded a time-lapse[2] of the process to demonstrate how the design evolved.

The first steps of the grammar split the given plot into blocks that are aligned in rows and columns. We could have achieved this by using the repeat operation along each axis but instead we use recursion (lines 5-13). By doing this, we can pass each block two parameters indicating which row and column it is in. The buildings in a block can then vary depending on the location of that block within the city. In our example, we make buildings closer to the centre have a higher probability of being taller. Each block consists of a random number of buildings (lines 15-17).

```
1  N = randint(8,10)
2  M = randint(8,10)
3  F = (N*N+M*M)/4
4  plot --> columns(1)
5  columns(c) : c <= M --> split(x){
6                              ~4 : rows(c,1) |
7                              ~1 : nil |
8                              ~(5*(M-c)) : columns(c+1)}
9  columns(c) : c > M --> nil
10 rows(c,r) : r <= N --> split(y){
11                            ~4 : block(c,r) |
12                            ~1 : nil |
13                            ~(5*(N-r)) : rows(c,r+1)}
14 rows(c,r) : r > N --> nil
15 block(c,r) --> repeatN(x, randint(3,5)){
16                    buildings((c*(M-c) + r*(N-r))/F)}
17 buildings(height) --> repeatN(y, randint(3,5)){building(height)}
18 ...
```

[2]The time-lapse video can be found here. The video quality can be improved by downloading.

Each building can either be a skyscraper or a small building. Buildings with a larger `height` parameter (calculated previously on line 16 based on distance from the centre of the city) have a higher chance of being a skyscraper. A skyscraper can be one of three different kinds (lines 26-28) whereas a small building is just a small cuboid (line 22).

```
19 ...
20 building(height) --> skyscraperPlot(rand(0.7,1),height) : height
21 building(height) --> smallBuilding(height) : 2+(1-height)
22 smallBuilding(t) --> split(z){~rand(1,4) : I(rect) | ~10 : nil}
23 skyscraperPlot(scalar, h) --> split(z){
24                                  ~(scalar*h) : skyscraper |
25                                  ~(1-(scalar*h)) : nil}
26 skyscraper --> tiered(randint(3,7)) : 0.2
27 skyscraper --> I(rect) : 0.5
28 skyscraper --> blocky : 0.3
29 tiered(n) : n > 1 --> split(z){
30                         ~(rand(0.5,1.5)) : I(rect) |
31                         ~(n-1) : nextTier(n-1)}
32 tiered(n) : n == 1 --> I(rect)
33 nextTier(numTiers) --> split(x){
34                           ~1 : nil |
35                           ~(rand(5,10)) : nextTier1(numTiers) |
36                           ~1 : nil}
37 nextTier1(numTiers) --> split(y){
38                            ~1 : nil |
39                            ~(rand(5,10)) : tiered(numTiers) |
40                            ~1 : nil}
41 blocky --> split(x){
42                ~1 : towers(0.7) |
43                ~1 : towers(1) |
44                ~1 : towers(0.7)}
45 towers(h) --> split(y){
46                   ~1 : tower(rand(0.3, 0.7),h) |
47                   ~1 : tower(rand(0.8, 1),h) |
48                   ~1 : tower(rand(0.3, 0.7),h)}
49 tower(h,max) --> split(z){~(h*max) : I(rect) | ~(1-(h*max)) : nil}
```

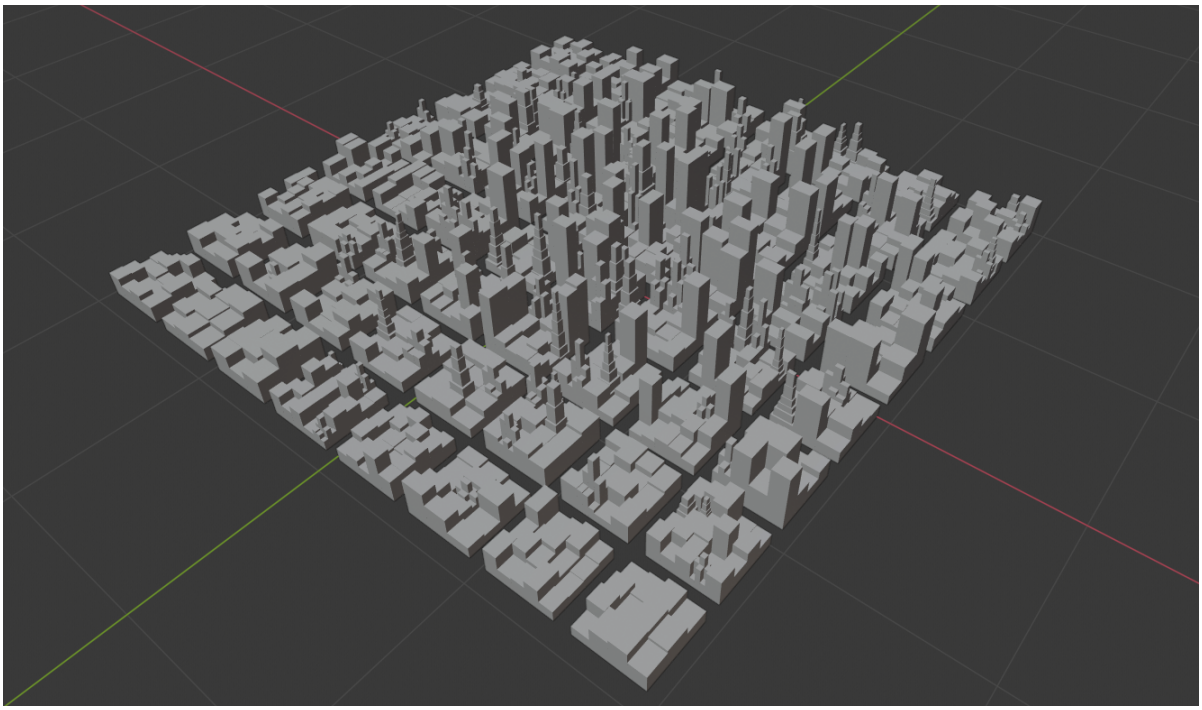


Figure 7.10: A city generated using our building grammar.

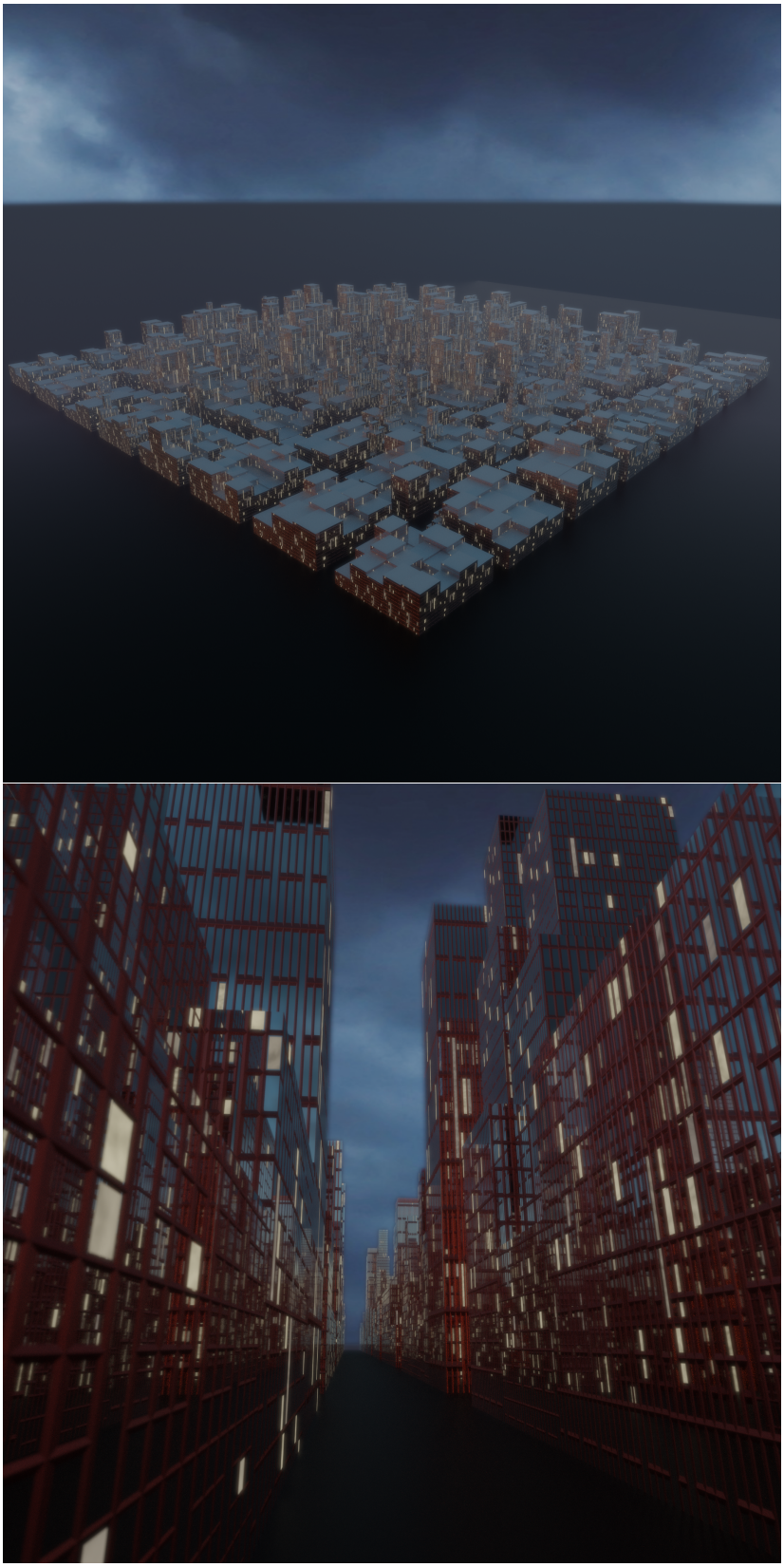


Figure 7.11: Rendered images of the city with textures added.

7.5 Limitations

One advantage that some other building grammars have over ours is that they are better at dealing with non-rectangular building footprints. In CGA Shape, for example, the initial scope for a building might be a triangular plot on the ground. The first rule of the grammar might then extrude this plane to create the volume which the house will fill. Creating non-rectangular shapes is possible in our grammar using non-rectangular primitives and by using rotation but it is less intuitive since the initial scope must always be a cuboid. This was one of the sacrifices we made when designing our grammar that makes the reverse challenge easier.

The other main sacrifice we made was to omit context checks within our grammar. In CGA Shape, a grammar can use contextual information to influence rules. For example, a grammar might add a door to a wall only if it is facing a street. Or perhaps a garage might be added to the side of a house but only if doing so would not cause an overlap with any other house. By omitting the ability for these kinds of checks we again make our grammar simpler and easier to reverse-engineer but perhaps a little less realistic or descriptive.

7.6 Extensions

One great extension to our work would be to integrate our modelling solution with other existing technologies. As previously mentioned, methods already exist to model the textures and façades of buildings as well as the layout of cities. Combining these methods with our procedural modelling of building structures would complete the process of procedurally modelling a whole city.

The process of adding a new built-in operation (like *split* or *repeat*) to our grammar is simple. Developers using our grammar might discover that a new operation, such as one that splits the current scope into a grid, would be useful and implementing this extension would be easy.

A key feature of our grammar is the ability to choose different operations to perform according to priorities. It allows some buildings to be more likely than others. The other way in which randomness is introduced into our grammar is through random values. Currently, whenever a random value is generated it is a taken from a uniform distribution between a minimum and maximum value - this is what the function $rand(min, max)$ does. A nice extension would be to allow for other distributions, such as a Gaussian distribution, to be used to evaluate random variables. This would likely allow for more realistic buildings. In our house-with-garage example, the garage's height was randomly chosen from a uniform distribution whereas in reality the heights of garages from different buildings is more likely to follow a Gaussian (or other) distribution. Although modelling distributions in this way would increase the quality of the forward model, it is worth noting that doing so would make the backward challenge more difficult.

Although our forward solution has its limitations, we hope you will agree that they are worthwhile sacrifices to make when it comes to solving the backwards problem.

Part II

The Backwards Problem

Chapter 8

Existing Research and Potential Approaches

For part 2 of this report we will focus on our second aim:

2. Define a method of reverse-engineering a grammar from a set of example buildings. The inferred grammar should be able to create many unique buildings similar to the examples.

Defining a forward model is useful but, as previously discussed, it still requires expertise to be able to write a grammar for a set of buildings. We will now investigate how we can generate new buildings from examples so the artist does not have to learn a new skill.

Attempts have already been made to provide more intuitive ways of describing the style of buildings. Some methods [2] try to provide a user-friendly interface and useful edit rules that makes the process of grammar-writing easier. This still requires an artist to learn how to use the software and adjust their current work flow. Ideally artists should be able to draw the first buildings of a city and then get a computer to generate the rest.

Merrel's [3] method of model synthesis gets closer towards this goal. Their method generates new buildings from a single example building with labelled 'model pieces' (equivalent to our primitive objects). The pieces are rearranged and repeated according to a set of rules that result in similar-looking buildings that still 'make sense'. Although parameters may be adjusted to alter the generated buildings, it would be useful to have a grammar that represents these final structures. If we can generate a grammar then a user could manually adjust it to modify the outputs. Merrel's method also takes just a single example building whereas it is more likely that an artist would want to provide a few example buildings that capture the whole style. By using multiple examples we should be able to detect what aspects of a building are unchanged and which parts vary, leading to more accurate new buildings being generated.

An alternative approach [5] attempts to generate a grammar for a building from a single photograph. A user simply has to draw the outline of the building in the photo and the program creates a 3D model. The method uses machine learning techniques to predict the structure of non-visible faces. In addition to modelling the given building, the system can adjust parameters and add variation to create buildings similar to those in the photo. This is a great technique for modelling buildings in the real world and is certainly an interesting read but for films or video games we often want to create imaginary, stylised designs rather than real buildings.

In our solution, we will use a set of example buildings to create a non-deterministic operation graph (our internal representation of buildings) that can create similar buildings. If a user wants to be able to modify the reverse-engineered grammar then we can output an editable building grammar representing the combination of the example buildings. Similarly to Merrel's use of labelled model pieces, we will leave it up to a human (or other software) to label the primitive objects that make up each building. Figure 8.1 shows a diagram of our backward model.

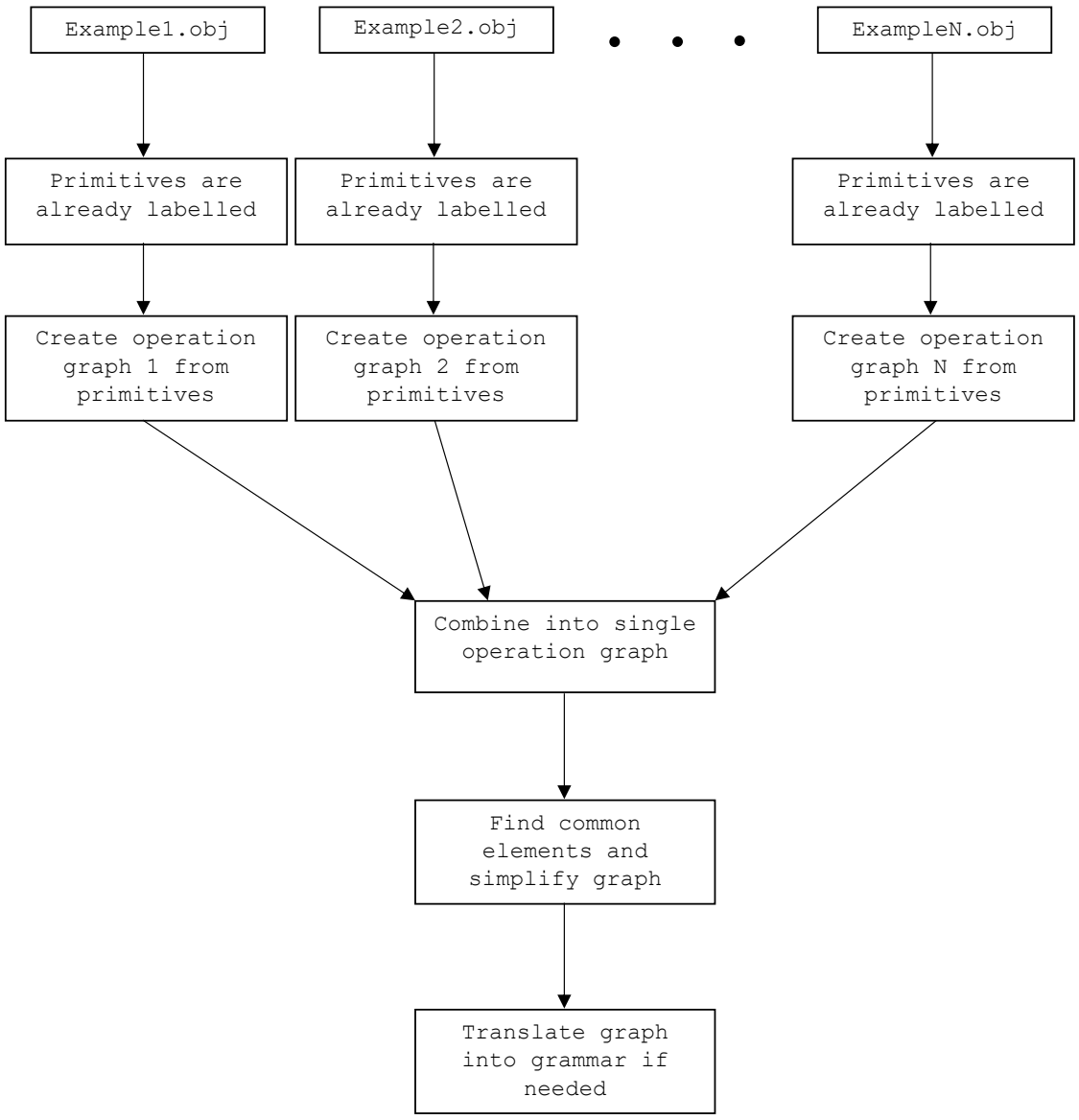


Figure 8.1: The backward model

Chapter 9

From Geometry to Operation Graphs

The goal of the backward model is to be able to infer a grammar from example buildings. The first step towards this is to create an operation graph from a single example. We can only do this if it is possible to represent the example building as an operation graph. We will only be considering buildings that have no overlapping primitives. Although we *can* represent overlapping geometry using the S, T, and R operations, inferring grammars of this kind is harder.

The first challenge of creating an operation graph from a building is to determine which parts of the building are primitive objects. Since research has already been done in this area, we aren't going to focus on automatically solving this problem. Example buildings will be provided as .obj files and primitive shapes must be manually labelled. In a .obj file, sets of vertices and faces can be grouped together into an object with a label. These groups will represent the primitive shapes. Any two objects with the same label are assumed to represent the same primitive. The .obj file excerpt below shows two different primitives (*rect* and *triangle*) where the *rect* primitive is used twice.

```
o rect
v 0.00000 0.00000 0.00000
v 3.33333 0.00000 0.00000
...
o rect
v 3.33333 0.00000 0.00000
v 10.00000 0.00000 0.00000
...
o triangle
v 3.33333 0.00000 6.66667
v 10.00000 0.00000 6.66667
...
```

Now that we have a set of labelled primitive shapes we can calculate a bounding box for each primitive. We will use these boxes to create an operation graph containing just split and primitive operations. Later we can try to simplify the graph and introduce operations like *choose* and *repeat*. We will now discuss the challenge of creating split operations from the bounding boxes.

9.1 The Box-Splitting Problem

We have n boxes, each one bounding a primitive shape. We will call the box that bounds all of these primitive boxes a **container**. We need to find an axis-aligned split plane which partitions the whole **container** into two parts without cutting any boxes in half. This means the split plane should go from one side of the container to the other without intersecting any boxes. We then need to recursively repeat this splitting process for each container in the partition until every container exactly contains either a primitive shape or empty space. This process is illustrated in 2D in Figure 9.1. Once we've done this, we can use the container splits to create an operation graph using *OpSplit*, *OpPrimitive* and *OpNil*. This operation graph will be deterministic (it will only generate the exact building that we input) and hence will be a DAG.

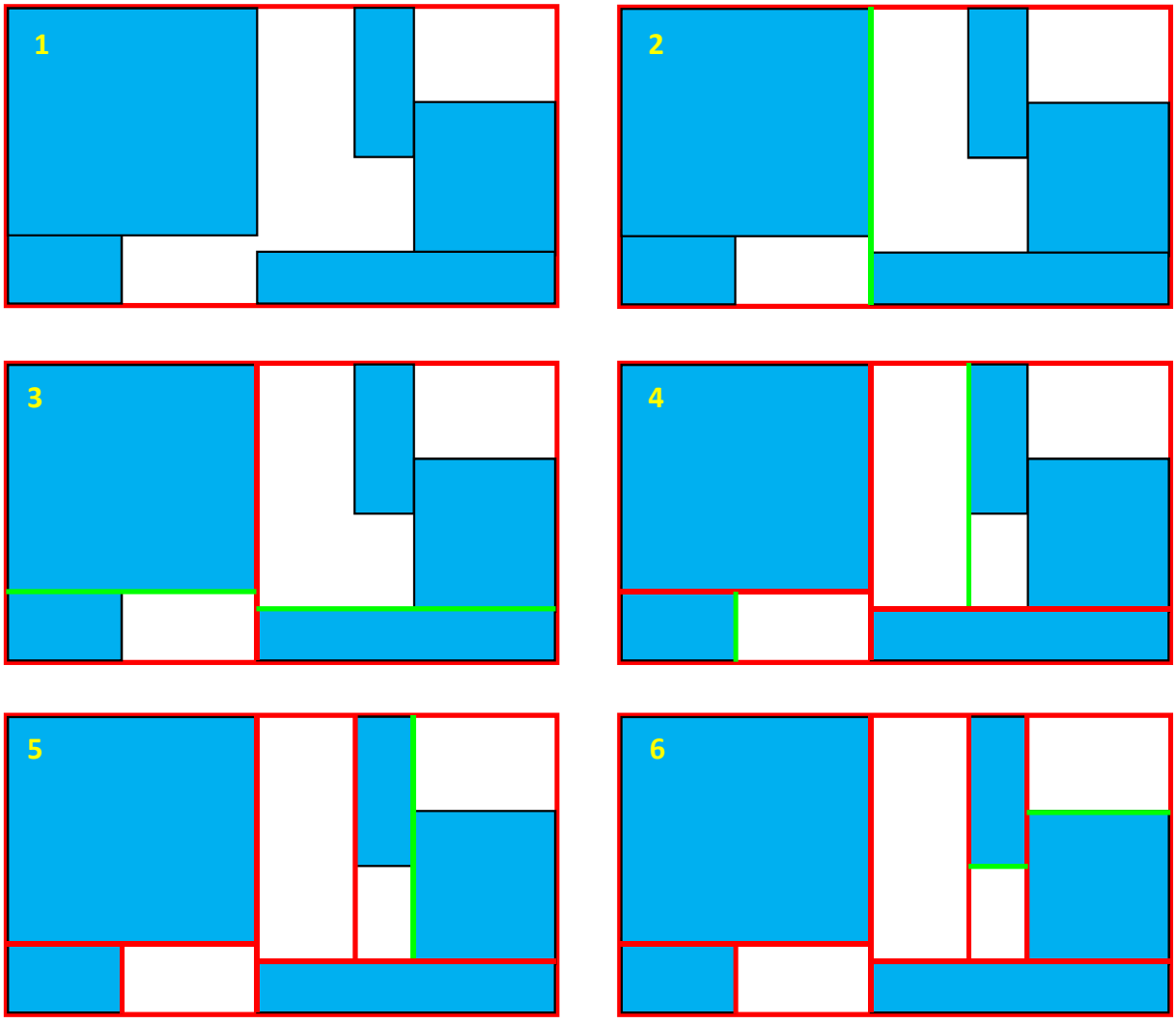


Figure 9.1: A 2D illustration of the box-splitting problem. The blue rectangles represent the bounding boxes of primitive objects. Green lines represent the new splits introduced on that iteration. The red lines outline the **containers**.

9.1.1 The Naive Approach

The first thing to note is that all split lines will be along an edge of at least one primitive shape. In 3D, the splits will be planes and will correspond to one of the 6 half-planes that form each primitive box. A naive approach to the box-splitting problem in 3D is outlined below in pseudo-code. Additional processing is needed to calculate the actual split operations with appropriate sizes but this does not affect the running time.

```
FUNCTION splitBoxes(boxes){
  FOREACH box IN boxes{
    FOREACH plane THAT bounds box{
      IF plane does not intersect any box in boxes{
        LET before = boxes before the plane
        LET after = boxes after the plane
        beforeSplit = splitBoxes(before)
        afterSplit = splitBoxes(after)
        RETURN split(plane, beforeSplit, afterSplit)
      }

    }
  }
}
```

This naive approach takes $O(n^2m)$ time where n is the number of primitive bounding-boxes and m is the length of the longest path in the resulting operation graph. We can be sure that such a path exists because (as previously mentioned) our operation graph will always be a DAG. We may have to iterate through all n boxes and for each we may have to check 6 bounding planes. For each of these planes we need to check if it intersects with all other boxes, taking $O(n)$ time. Once we've found a splitting plane, which takes $O(n^2)$ time, we need to recursively call *splitBoxes*. In total, *splitBoxes* will be called $O(m)$ times since on each recursion we increase the maximum path length in the operation graph by 1. This gives us the total running time of $O(n^2m)$.

9.1.2 Sorting Boxes

We can improve the time taken to find a valid split by ordering our boxes. The easiest way to imagine this is to look at the one-dimensional case. Instead of 3D boxes, we look at intervals between two numbers (a, b). Given a set of intervals we can find a split point (assuming one exists) in $O(n \log n)$ time by ordering the bounds of the intervals. We can in fact find all valid splits in $O(n \log n)$ time. The algorithm below returns a list of lists where each list contains intervals belonging to the same split. The idea is that we keep track of how many unclosed intervals we have opened. Whenever we close an interval, we check whether there are none left open. If so, then we can split at that point.

```
FUNCTION splitIntervals(intervals){
  each interval consists of two bounds
  each bound has a pointer to its interval
  LET bounds = a list containing both bounds of all intervals
  LET sorted = bounds sorted. Max bounds precede min bounds if two are equal
  LET unmatched = 0
  LET thisSplit = []
  LET splits = []
  FOREACH bound IN sorted{
    IF bound is a minimum{
      unmatched += 1
    }
    ELSE{
      unmatched -= 1
      thisSplit.append(the interval this bound is part of)
      IF unmatched == 0{
        We have no unmatched bounds so can split here
        splits.append(thisSplit)
        thisSplit = []
      }
    }
  }
}
```

iteration	bound	interval	thisSplit	splits	unmatched
0			[ ]	[ ]	0
1	0	(0,4)	[ ]	[ ]	1
2	1	(1,5)	[ ]	[ ]	2
3	2	(2,5)	[ ]	[ ]	3
4	4	(0,4)	[(0,4)]	[ ]	2
5	5	(1,5)	[(0,4), (1,5)]	[ ]	1
6	5	(2,5)	[ ]	[[(0,4), (1,5), (2,5)]]	0
7	5	(5,9)	[ ]	[[(0,4), (1,5), (2,5)]]	1
8	8	(8,9)	[ ]	[[(0,4), (1,5), (2,5)]]	2
9	9	(5,9)	[(5,9)]	[[(0,4), (1,5), (2,5)]]	1
10	9	(8,9)	[ ]	[[(0,4), (1,5), (2,5)], [(5,9), (8,9)]]	0
11	11	(11,14)	[ ]	[[(0,4), (1,5), (2,5)], [(5,9), (8,9)]]	1
12	14	(11,14)	[ ]	[[(0,4), (1,5), (2,5)], [(5,9), (8,9)], [(11,14)]]	0

Above is an example execution with the following intervals. Each row in the table indicates the variable values at the end of that iteration. Figure 9.2 illustrates how we can split the intervals below.

```
intervals = [(0,4),(1,5),(2,5),(5,9),(8,9),(11,14)]
allBounds = [0,4,1,5,2,5,5,9,8,9,11,14]
sortedBounds = [0,1,2,4,5,5,5,6,8,9,9,11,14]
```

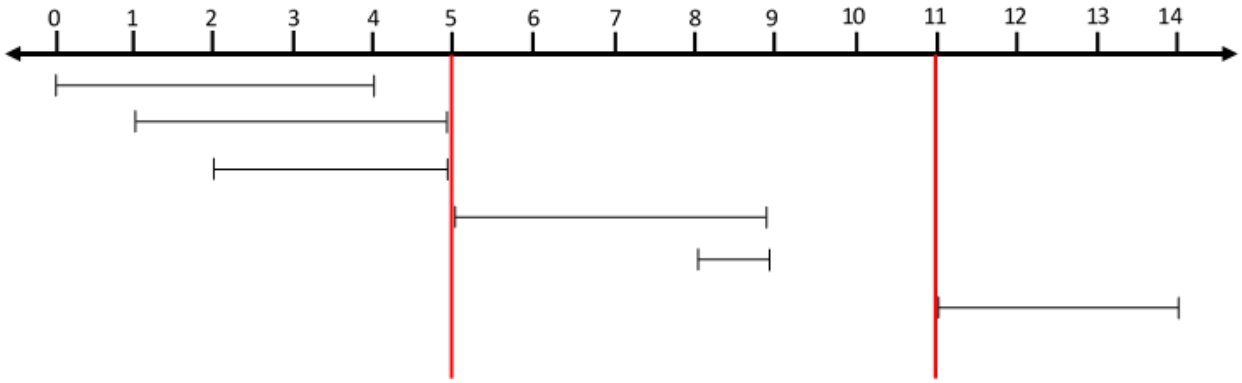


Figure 9.2: An illustration of how we can split intervals in one-dimension. Our goal is to find the red split lines that partition the intervals.

To extend this to our problem in 3D, we just need to look for splits on all three axes using the same interval method. By doing this we can improve the running time of the naive algorithm to $O(mn \log n)$. However, this method still wastes computation time by re-sorting the bounds on each recursive call of the function.

9.1.3 Ordered-Box Algorithm

We now propose an alternative **ordered-box** algorithm with worst-case running time $O(n \log n + nm)$. This method only sorts the boxes once for each axis. The algorithm is easily generalised to higher dimensions and for a dimension d, it has running time $O(d(n \log n + nm))$. In this algorithm, each box stores a reference to its smallest container. A container is a **primitive container** if it exactly contains a primitive box (i.e. it contains just one primitive box and no empty space). The initial container is a bounding box for all the boxes and all boxes reference this container at the start. Just as in the previous algorithm, we sweep along an axis and process the ordered bounds. Instead of just keeping one `unmatched` counter, we keep a counter for each container. This means that when checking for overlaps and possible splits we only compare primitives within the same container. We will now outline the Python implementation of this algorithm.

There are 3 classes involved in our box-splitting implementation.

1. The `Box` class represents a bounding box for a primitive object. Its attributes include bound objects for each axis as well as a pointer to the smallest `Container` that contains it.
2. The `Bound` class stores these bounds. Each `Bound` object stores its value, a pointer to the Box it is part of and whether it is a min or max bound.
3. The `Container` represents the containers we have already discussed. Each container object has its own `unmatched` counter that is used to determine where it can be split.

As we sweep through the boxes looking for splits, each `Container` keeps track of unmatched boxes within that container and creates a `child` container to contain the current split section. Whenever the end of a box is found, the `addToChild` method of the box's container is called:

```
1  # Add box to the current child
2  # If child can be split off and a new child started then do so
3  def addToChild(self, box):
4      self.unmatched -= 1
5      if self.child == None:
6          self.resetChild()
7      # Increase the size of child to also fit box
8      self.child.expandToFit(box)
9      # self.child is now the smallest container containing box
10     box.container = self.child
11     # If we no longer have unmatched boxes then we can split
12     # and start a new child
13     if self.unmatched == 0:
14         # If self.child exactly fits box, we can just replace
15         # with a PrimContainer
16         if self.child.numPrims == 1
17           and self.child.perfectlyContains(box):
18             bounds = self.child.bounds
19             box.container = PrimContainer(box.prim, bounds)
20             self.children.append(box.container)
21         else:
22             self.children.append(self.child)
23         self.resetChild()
```

When `addToChild` is called, we first decrement the container's unmatched counter. The box is then added to the current child container. If the current child container can now be successfully split off (because the unmatched counter has reached zero) then we do so.

Once all containers are primitive containers, we know that no more splits need to be done. We can then convert the initial container (a container bounding all the boxes) into an operation. This works by creating a new split operation (line 35 below) whose child operations are calculated by recursively converting a container's child containers into operations (line 19 below):

```
1  # Convert this container into a split operation
2  def toOp(self):
3      sizes = []
4      childOps = []
5      # There may be empty space at beginning of this container
6      # that needs to represented with a nil operation
7      prev = self.bounds[self.axis][0]
8      for i in range(len(self.children)):
9          child = self.children[i]
10
11         # Empty space between this and previous child?
12         # If so, fill it with a nil operation
13         nilSize = child.bounds[self.axis][0] - prev
14         if nilSize > 0:
15             childOps.append(OpNil())
16             sizes.append(Size(nilSize, True))
17
18         # Convert this child to an operation
19         childOps.append(child.toOp())
20         ax = self.axis
21         s = child.bounds[ax][1] - child.bounds[ax][0]
22         sizes.append(Size(s, True))
23         prev = child.bounds[self.axis][1]
24
25     # We also need to check for empty space after the last child
26     nilSize = self.bounds[self.axis][1] - prev
27     if nilSize > 0:
28         childOps.append(OpNil())
29         sizes.append(Size(nilSize, True))
30
31     # If there is only one child just return that instead
32     if len(childOps) == 1:
33         return childOps[0]
34     else:
35         return OpSplit(axisName[self.axis],
36                        perChildArgs=tuple(sizes),
37                        childOps=childOps)
```

The function for converting a given set of primitive boxes into an operation is shown below. The loop on line 18 keeps sweeping along each axis trying to split containers until no container contains

more than one box. When the max bound of a box is processed, we call the `addToChild` function as described earlier (line 33). Once all boxes are contained within a primitive container, we simply convert the initial container to an operation (line 35).

```
1  def boxesToOp(boxes, initialContainer):
2      initialContainer.numPrims = len(boxes)
3      for box in boxes:
4          box.container = initialContainer
5
6      bounds = [sorted(
7                  (bound for b in
8                    boxes for bound in
9                      (b.bounds[ax][0], b.bounds[ax][1])
10                 ),
11                 key=lambda b: (b.val, b.low)
12                ) for ax in (0,1,2)]
13
14     axis = cycle((0,1,2))
15     allPrimitiveContainers = False
16
17     # Done when all boxes' containers are primitive containers
18     while not allPrimitiveContainers:
19         allPrimitiveContainers = True
20         ax = next(axis)
21         for bound in bounds[ax]:
22             box = bound.box
23             container = box.container
24             if not container.isPrim:
25                 allPrimitiveContainers = False
26                 if bound.low:
27                     # Inc unmatched counter for this box's container
28                     container.incUnmatched()
29                 else:
30                     # Add this box to its containers current child
31                     # Also updates the box's container pointer
32                     # Will do split if container's counter is zero
33                     container.addToChild(box)
34
35     return initialContainer.toOp()
```

For simplicity, we began by implementing the naive algorithm to solve the box-splitting problem. The running times were noticeably slow for buildings with many primitives so we created a testing framework that automatically generates 'buildings' (in fact just a jumble of boxes) with specified parameters (e.g. number of primitives, max depth of operation graph and max branching factor). The graph in Figure 9.3 shows a comparison of the naive algorithm and the more efficient ordered-box algorithm. Interestingly, despite the theoretical running times of $O(n^2m)$ and $O(n \log n + nm)$, the actual running times seemed to increase linearly for large values of n for both methods. We suspect that this might have something to do with the structure of the randomly generated test buildings. Compiler optimisations may also have had an impact.

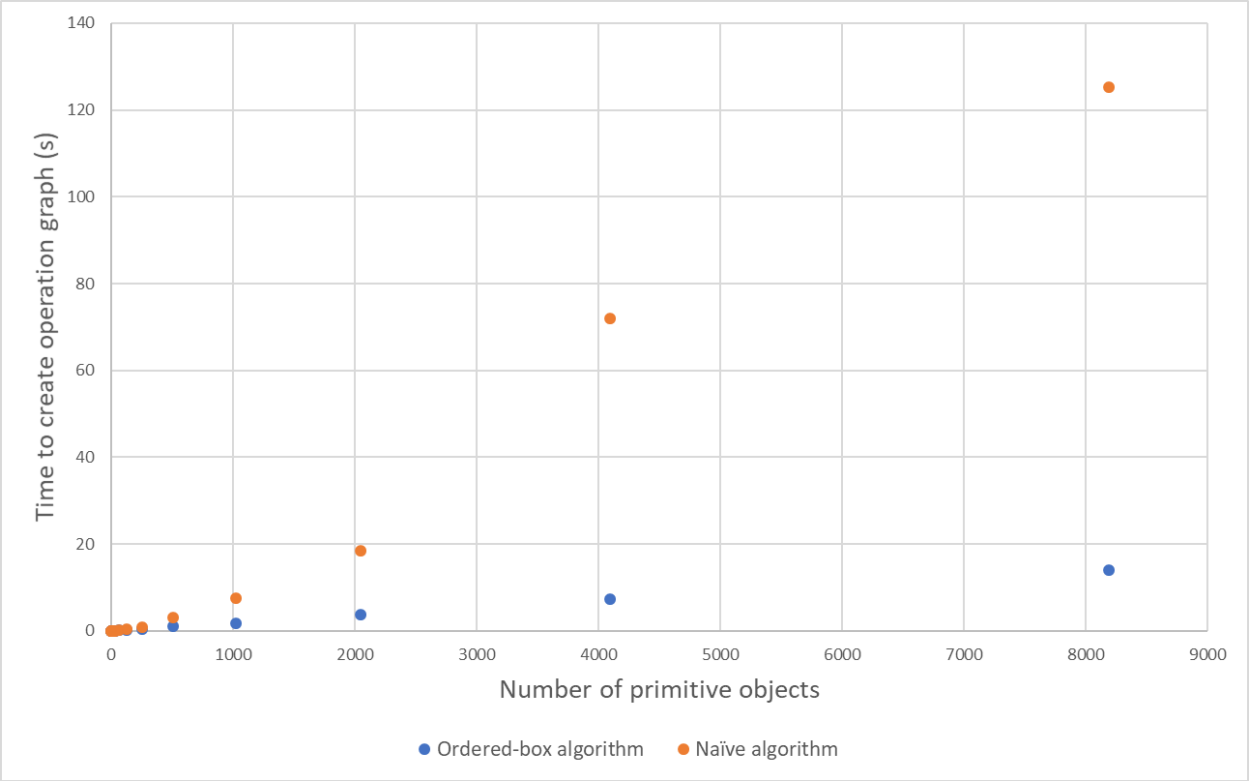


Figure 9.3: A comparison of running times for the two box-splitting algorithms. The ordered-box algorithm managed to create an operation graph from an object file containing over 8000 primitive shapes in less than 14 seconds whereas the naive algorithm took more than 2 minutes to process the same number of primitives. In this specific test, all operation graphs have a max depth of 6 and a max branching factor of 5 but the naive approach was outperformed in all test cases.

Chapter 10

Combining Operation Graphs

10.1 Combining with the Choose Operation

Given a number of example buildings, we have seen how to create an operation graph for each building. We now aim to combine these operation graphs to create a single non-deterministic operation graph that generates more buildings similar to the examples. Formally, given buildings B_1 to B_n we've seen how to get Op_1 to Op_n such that Op_i is a deterministic operation graph representing B_i. We now aim to combine Op_1 to Op_n into a single non-deterministic operation graph Op^* such that Op^* creates buildings similar to the given B_is.

The most simple way to combine a set of operation graphs is to create a new *OpChoose* operation which chooses one of the operation graphs and runs it. If our example buildings were a house, a church and a skyscraper we could create a new operation that chooses between these three options. Although this new operation does represent the three buildings, it is not very interesting. The new operation could be used to create a city of houses, churches and skyscrapers... but they would all look the same. The problem that this solution does not address is detecting similar components in the buildings and using these to create new and varying buildings.

10.2 Identifying Equal Operations

Identifying identical components of different buildings is the first step towards identifying similar components. We can generate a hash for an operation and use this to unify identical operations in an operation graph. Taking this a step further, we can take a set of operation graphs, combine them with the choose operation and then simplify the resulting operation graph. By doing this we will identify and unify identical components across different buildings. Figure 10.1 uses the house-with-garage example to show the result of combining and simplifying two operation graphs.

10.3 Simplifying a Split Operation

In the house-with-garage (Figure 10.1), we were able to unify the two split operations that represented the garage side of the building because all of their parameters were equal. But what if the

garage varied in height as was the case in a grammar discussed earlier (repeated below)?

$$\begin{aligned} start &\longrightarrow split(x)\{\sim 2 : house \mid \ \sim 1 : garageSide\} : 0.5 \\ start &\longrightarrow split(x)\{\sim 1 : garageSide \mid \ \sim 2 : house\} : 0.5 \\ house &\longrightarrow split(z)\{\sim 2 : I(rect) \mid \ \sim 1 : I(roof)\} \\ garageSide &\longrightarrow split(z)\{\sim (rand(0.75, 1.25) : I(rect) \mid \ \sim 2 : nil\} \end{aligned} \tag{10.1}$$

We want to be able to recognise that the two garages have the same structure even if their parameters vary. Instead of just searching for identical operations in an operation graph, we can find similar operations and try to combine their parameters to create a new operation that represents them both. In the garage example, we can create a new operation that generates a garage with a height lying in the range determined by the two example garages. In general, given two splits both with the same axis and child operations and with sizes S_1 to S_n and T_1 to T_n respectively, we can create a new *split* operation where each section size is randomly taken from the range determined by S_i and T_i:

Operation 1:
$OpSplit(axis, sizes = [s_1, \ldots, s_n], childOps)$
Operation 2:
$OpSplit(axis, sizes = [t_1, \ldots, t_n], childOps)$

Combination of operations 1 and 2:
$OpSplit(axis, sizes = [rand(s_1, t_1), \ldots, rand(s_n, t_n)], childOps)$

We can apply the same parameter-combining logic to other operations in a similar way. We can further simplify the *split* operation by collapsing child splits. For a *split* along a given axis, if it has a child that is also a *split* and is along the same axis then we can combine child and parent to create a single *split* operation.

$$\begin{aligned} child_i &= OpSplit(axis = x, sizes = [s_1, \ldots, s_n], childOps) \\ mySplit &= OpSplit(axis = x, sizes = [t_1, \ldots, t_n], [child_1, \ldots, child_n]) \end{aligned}$$

$$\begin{aligned} newSizes &= [t_1, ..., t_{i-1}] +\!\!+ [s_1, ..., s_n] +\!\!+ [t_{i+1}, ..., t_n] \\ newChildOps &= [child_1, ..., child_{i-1}] +\!\!+ childOps +\!\!+ [child_{i+1}, ..., child_n] \\ mySplit &== OpSplit(axis = x, newSizes, newChildOps) \end{aligned}$$

The final way in which we can simplify a *split* is by using the *repeat* operation to replace any *split* operation with equal child operations and equal sizes.

10.4 Simplifying a Choose Operation

We can simplify the *choose* operation by combining identical child operations. For now, we will ignore any conditions associated with each child operation. E.g.

$$\begin{aligned} &OpChoose(priorities = [1,3,5], childOps = [op_a, op_b, op_a] \\ ==&OpChoose(priorities = [6,3], childOps = [op_a, op_b]) \end{aligned}$$

Again, similar child-flattening rules exist for other operations.

Just using these basic simplifying rules, we can detect identical or similar elements in an operation graph and combine them into a single operation. All of these simplifying and combining rules are contained with an operation's *simplify* function. So given Op_1 to Op_n, representing a set of example buildings, we simply construct Op^* using the *choose* operation and then run *simplify*.

$$Op^* = OpChoose(priorities = [1, ..., 1], childOps = [Op_1, ..., Op_n]).simplify()$$

In Chapter 12 we will discuss some examples where we have combined a set of example buildings.

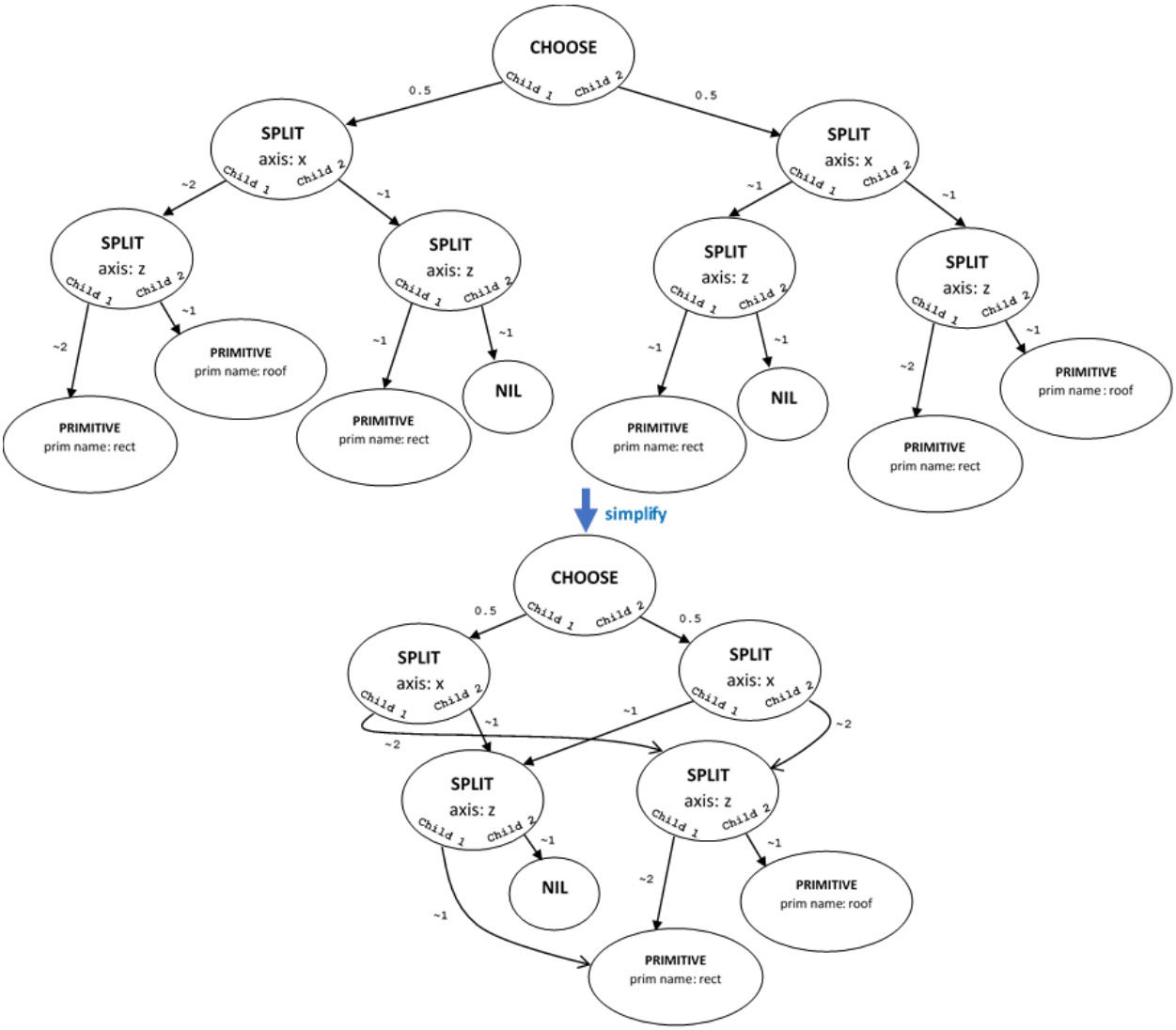


Figure 10.1: Given two example operation graphs (one with a garage on the right and one with a garage on the left) we can first combine with the choose operation and then simplify by unifying equal operations.

Chapter 11

From Operation Graphs to Grammars

So far we have described how to create an operation graph from a set of example buildings. This is all that is required to be able to run the operation graph to create new, similar buildings. However, it may also be useful to provide a human-readable grammar for the set of example buildings. Given this grammar, a user could then make some modifications to the buildings created and adjust various parameters.

Going from an operation graph to a grammar is simple. Most operations correspond directly to operations that can be used in the grammar. E.g.

$$\begin{gathered} OpSplit(axis, sizes = s, childOps = c) \\ \downarrow \\ split(axis)\{s_1 : c_1 \mid ... \mid s_n : c_n\} \end{gathered}$$

Where we have a *choose* operation we simply need to create a rule for each child operation and give them all the same label. For a *choose* operation with child operations op_1 to op_n and corresponding priorities p_1 to p_n, we can construct a rule for each child operation:

$$\begin{aligned} chooseRule &\longrightarrow op_1 : p_1 \\ chooseRule &\longrightarrow op_2 : p_2 \\ &... \\ chooseRule &\longrightarrow op_n : p_n \end{aligned}$$

In the forward model, it was possible to write grammars with nested operations such as:

$$start \longrightarrow split(x)\{1 : split(y)\{1 : I(rect) \mid 1 : I(rect)\} \mid 1 : I(rect)\}$$

Nesting operations in this way is a matter of preference and readability. When reverse-engineering a grammar we will never nest operations (except the primitive I operation) and will instead create an extra rule:

$$start \longrightarrow split(x)\{1 : start1 \mid 1 : I(rect)\}$$
$$start1 \longrightarrow split(y)\{1 : I(rect) \mid 1 : I(rect)\}$$

The readability of grammars is greatly improved by good rule labels. Unfortunately our reversed-engineered grammars are unable to create meaningful rule labels so it is up to a user to then edit labels for readability.

Chapter 12

Examples and Extensions

For most of our tests (and for the examples we will show here) we used a grammar to generate example buildings rather than drawing each one by hand. This has the following advantages:

1. It is often much faster (particularly for a computer scientist) to describe a building with a grammar rather than using 3D modelling software to draw one.
2. By using a non-overlapping grammar, we know it is possible to reverse-engineer the grammar from the examples.
3. We can easily compare the reverse-engineered grammar to the actual grammar that generated the examples and use the similarity as a measure of success.

We will re-use the examples discussed in Chapter 7. This time we will be generating a number of example buildings using the grammar and then trying to create buildings similar to the examples and reverse-engineer the grammar.

12.1 House with a Garage

For our first example we will try to reverse-engineer the simple house-with-garage grammar. The garage can appear on either side of the house and its height can vary within some range. We will generate 10 example buildings using the following grammar (written in a machine-readable form):

```
plot --> split(x){~2 : house | ~1 : garage} : 0.5
plot --> split(x){~1 : garage | ~2 : house} : 0.5
house --> split(z){~2 : I(rect) | ~1 : I(triangle)}
garage --> split(z){~(rand(0.75,1.25)) : I(rect) | ~2 : nil}
```

The 10 example buildings are shown in Figure 12.1. The first step of the backwards model is to create an operation graph for each of the examples and then create a single operation graph representing them all. In this case, since there is no recursion, this will be an operation tree. The operation tree inferred from the 10 examples is shown below in a text format. The indentation indicates the parent/child relationships between operations. Although some identical operations are repeated, internally these are represented by the same operation object.

```
OpChooseRuleWithPriority(((0.7, 0.3), (True, True)))
  OpSplit(0, (RelSize(6.67), RelSize(3.33)))
    OpSplit(2, (RelSize(6.67), RelSize(3.33)))
      OpPrimitive('rect')
      OpPrimitive('triangle')
    OpSplit(2, (RelSize(rand(3.07,3.73)), RelSize(rand(6.27,6.93))))
      OpPrimitive('rect')
      OpNil()
  OpSplit(0, (RelSize(3.33), RelSize(6.67)))
    OpSplit(2, (RelSize(rand(3.07,3.73)), RelSize(rand(6.27,6.93))))
      OpPrimitive('rect')
      OpNil()
    OpSplit(2, (RelSize(6.67), RelSize(3.33)))
      OpPrimitive('rect')
      OpPrimitive('triangle')
```

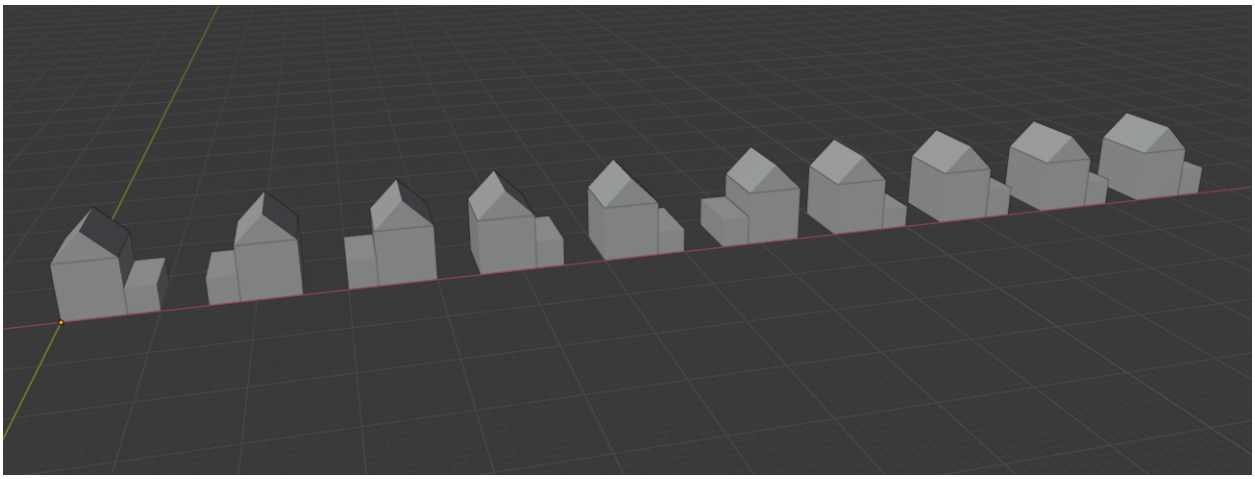


Figure 12.1: 10 example buildings generated by the house-with-garage grammar.

Once we have obtained this operation graph we can generate the new buildings shown in Figure 12.2.

If we would like to be able to modify the inferred building generator, we can create a human-readable grammar:

```
rule0 --> rule1 : 0.7
rule0 --> rule4 : 0.3
rule1 --> split(x){~6.67 : rule2 | ~3.33 : rule3}
rule2 --> split(z){~10 : I(rect) | ~5 : I(triangle)}
rule3 --> split(z){~(rand(3.07, 3.73)) : I(rect) |
                   ~(rand(6.27, 6.93)) : nil}
rule4 --> split(x){~(3.33) : rule3 | ~(6.67) : rule2}
```

There are clear similarities between this inferred grammar and the grammar that generated the examples. The initial choice between `rule1` and `rule4` represents the options for which side of the

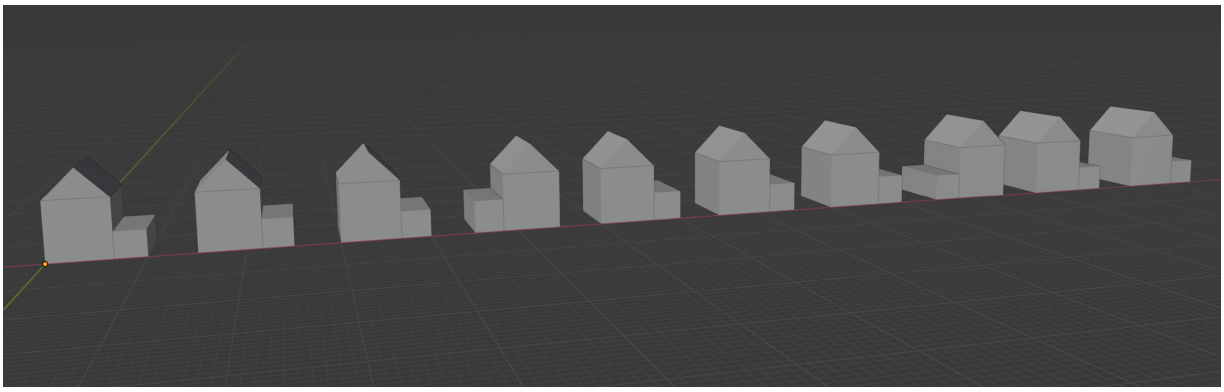


Figure 12.2: 10 buildings inferred by the example buildings in figure 12.1

house the garage is on. In our 10 examples in Figure 12.1, 7 out of the 10 buildings have the garage on the right side, hence our inferred grammar has a 0.7 chance of placing the garage on the right. In `rule3` of the inferred grammar we can see how the various heights of the example garages are used to create a range in which the new garages will be generated. The minimum and maximum heights of the garages in the examples are used to infer the minimum and maximum heights of garages that can be generated by the reverse-engineered grammar. A drawback of this approach is that we fail to capture the true distribution of garage heights. For example, even if our 10 examples consisted of 9 small garages with height between 1 and 2 and 1 large garage with height 10 then our inferred grammar would generate buildings where the garage height is uniformly distributed in the range from 1 to 10. A nice extension to our reverse modelling solution would be to try and model the distribution of parameters in the examples and then generate new buildings according to these distributions.

12.2 Modern Houses

Figure 12.3 shows 10 example buildings with the modern design we described in Part 1. When we attempt to infer some new designs from the examples we get the buildings in Figure 12.4. In these generated buildings we see that although they have varying numbers of floors, all of the buildings are the same height. This is because our reverse model assumes that all example buildings were generated from the same initial 3D plot and therefore have the same height. It does not attempt to detect any empty space that might be part of the initial plot. To get better results we can restrict our example buildings to two floors. After this restriction, we can generate the inferred buildings in Figure 12.5. These are better proportioned and look appropriately similar to the examples.

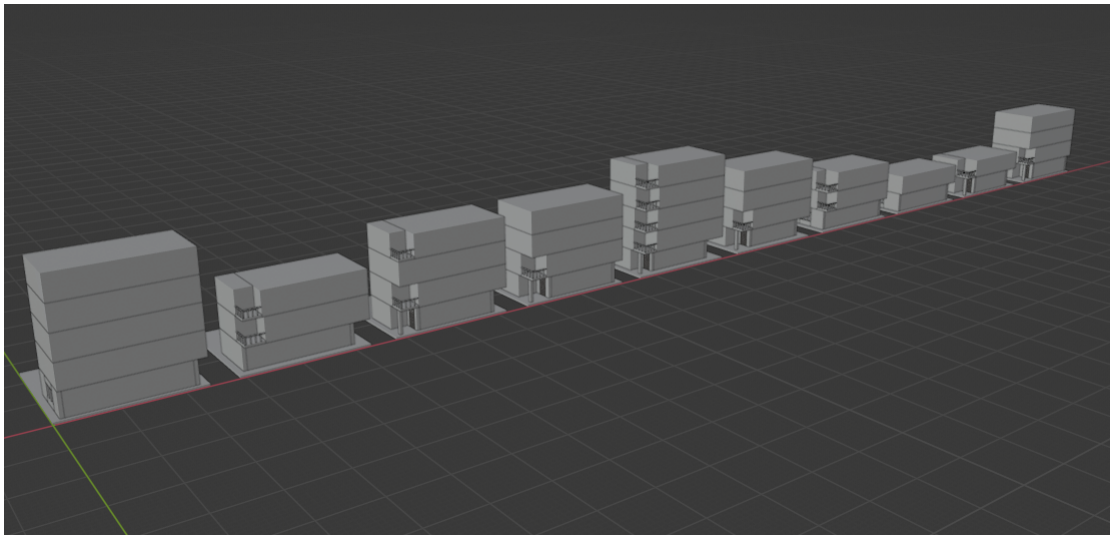


Figure 12.3: 10 example modern buildings

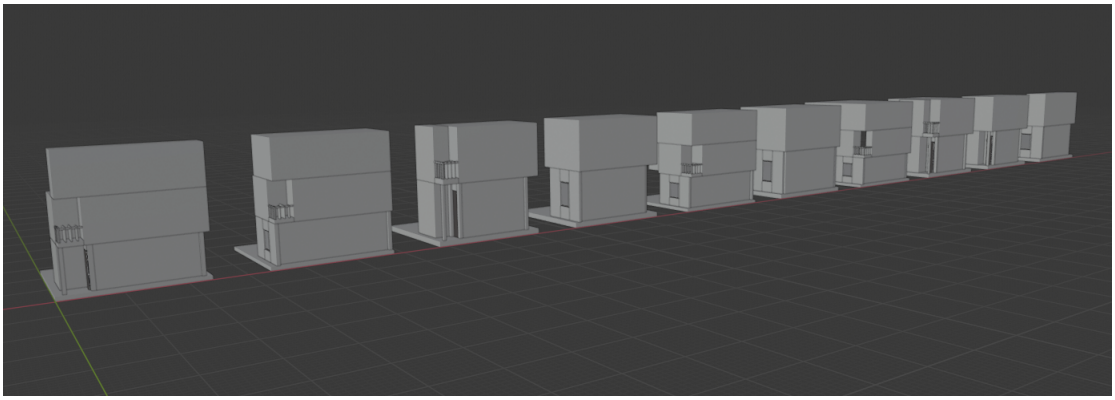


Figure 12.4: Buildings inferred by the example buildings in figure 12.3. The height of the buildings is the same despite the different numbers of floors.

Figure 12.6 shows a close-up view of a balcony on one of the buildings that reveals an error in the backward model. The cuboids that make up the balcony are strangely elongated in some places. In many places throughout the modern house model we use a split operation where one section of the split is filled with a cuboid and the rest is nil (empty space). We believe what has happened here is that our backward model has combined the operation which creates the balcony bars with an

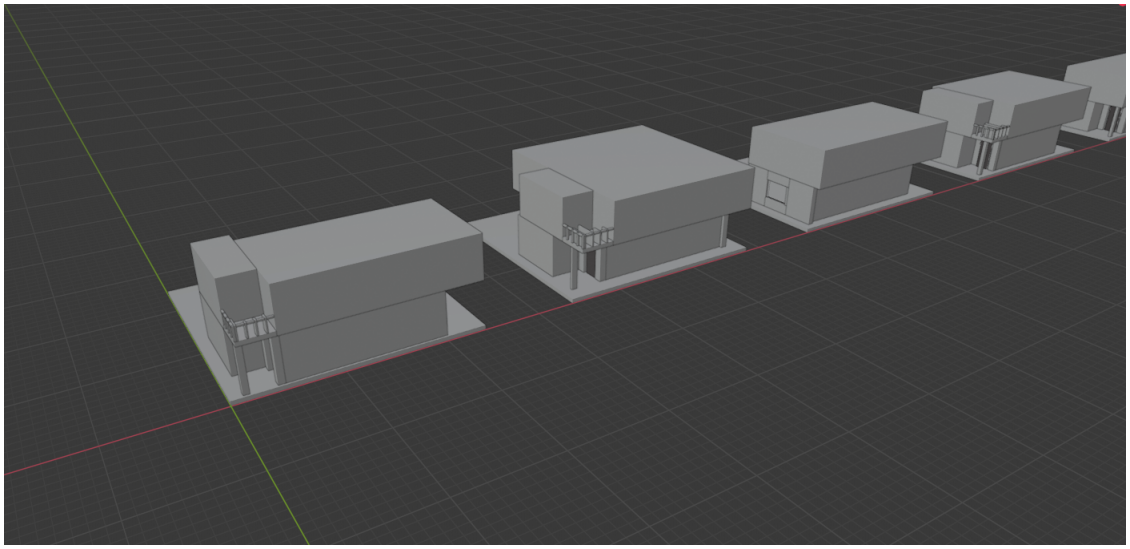


Figure 12.5: Buildings inferred by example modern buildings with just two floors.

operation that creates a wall of the building in order to try and simplify the operation graph. One way of preventing this error would be to label the components of the balcony as different primitive objects.

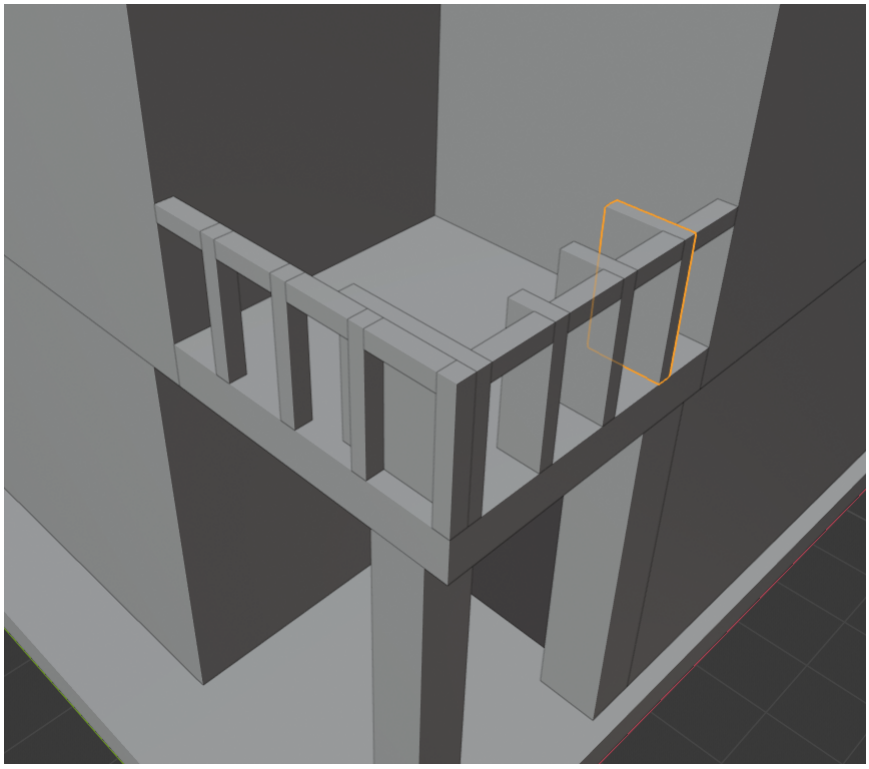


Figure 12.6: The balcony of one of our generated buildings. Note the strangely shaped balcony bars (one is highlighted).

12.3 Skyscrapers

The most complex grammar we described in the forward model was the grammar for a city of skyscrapers. The grammar makes use of parameterised rules, something that we don't attempt to reverse engineer, so we were interested to see if our reverse model could still produce good results. Figure 12.7 shows some example cities that we provided to our reverse model, and Figures 12.8 and 12.9 show the inferred cities. As you can see, the inferred models have managed to capture the height variation of buildings throughout the city, with buildings nearer the centre being taller.

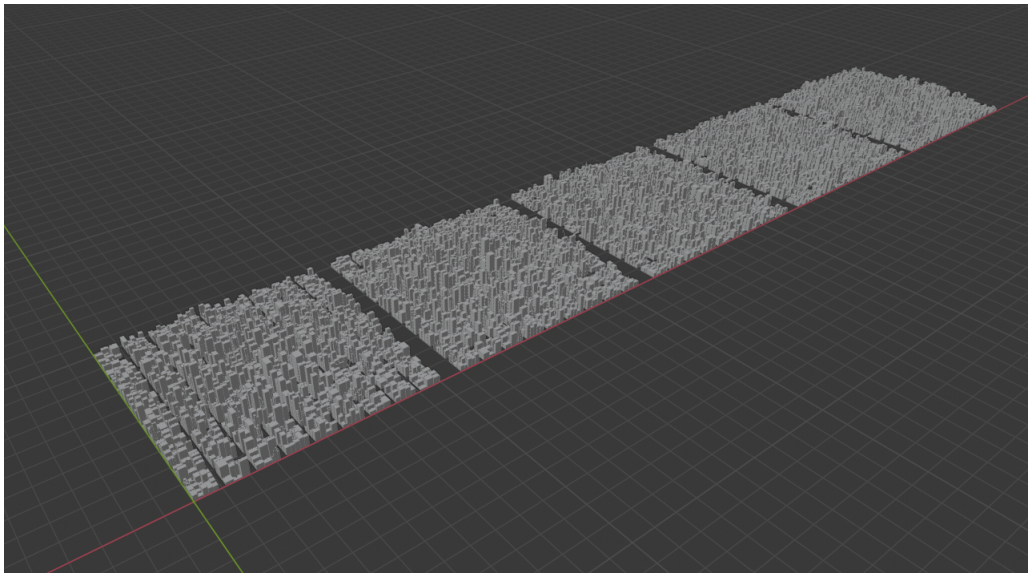


Figure 12.7: Example cities.

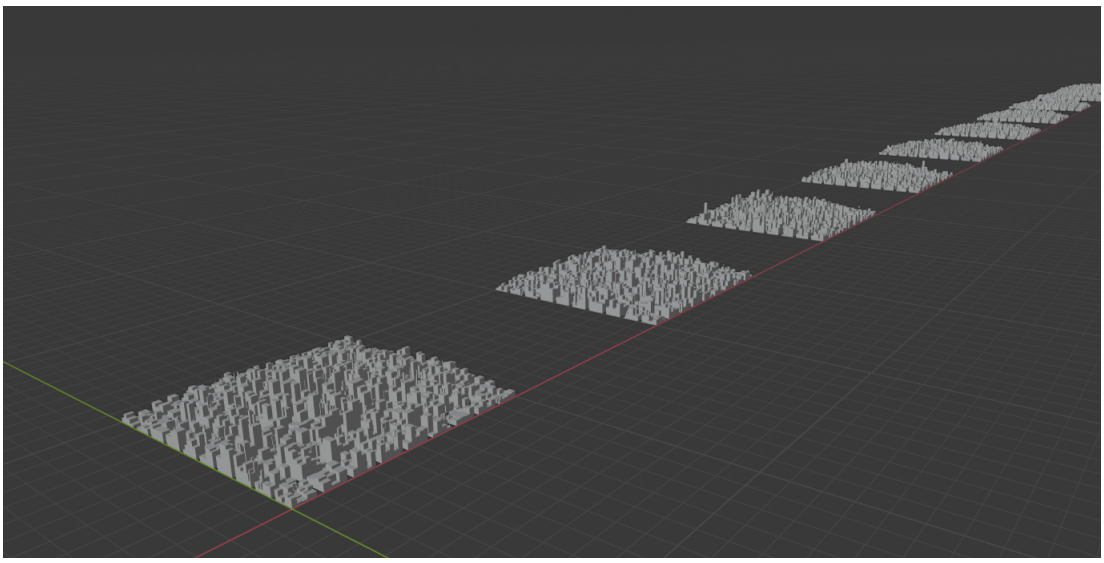


Figure 12.8: New cities generated by the reverse model.

The three different skyscrapers are all present but the tiered version has been badly inferred (Figure 12.10). We are still investigating this problem but, similarly to the balcony, we suspect it could be solved by introducing primitives that are used only for that kind of building.

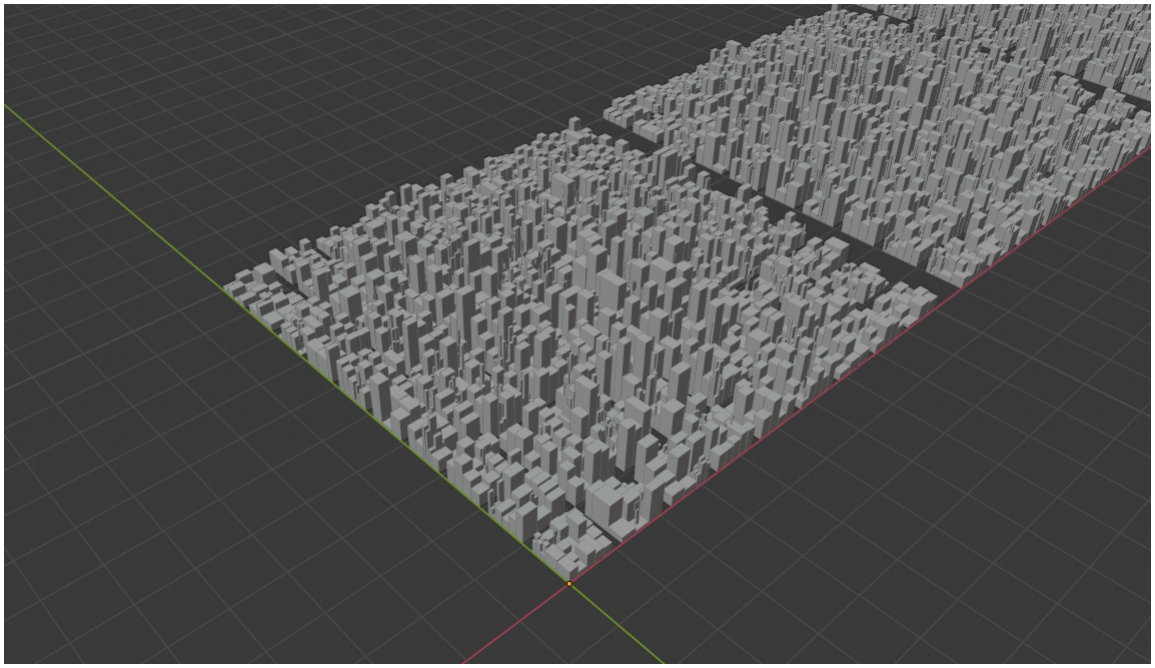


Figure 12.9: Close up of an inferred city.

Since we have not attempted to reverse-engineer parameterised rules, the grammar that our reverse model manages to generate contains hundreds of rules. A great extension to our model would be to attempt to infer these parameterised rules and combine similar rules with slightly different arguments into a single rule that accepts a parameter. For example, we could combine two operations which split along different axes but have the same child operations into a single operation which takes an axis parameter. We think that by doing this we could greatly reduce the number of rules in the inferred city grammar.

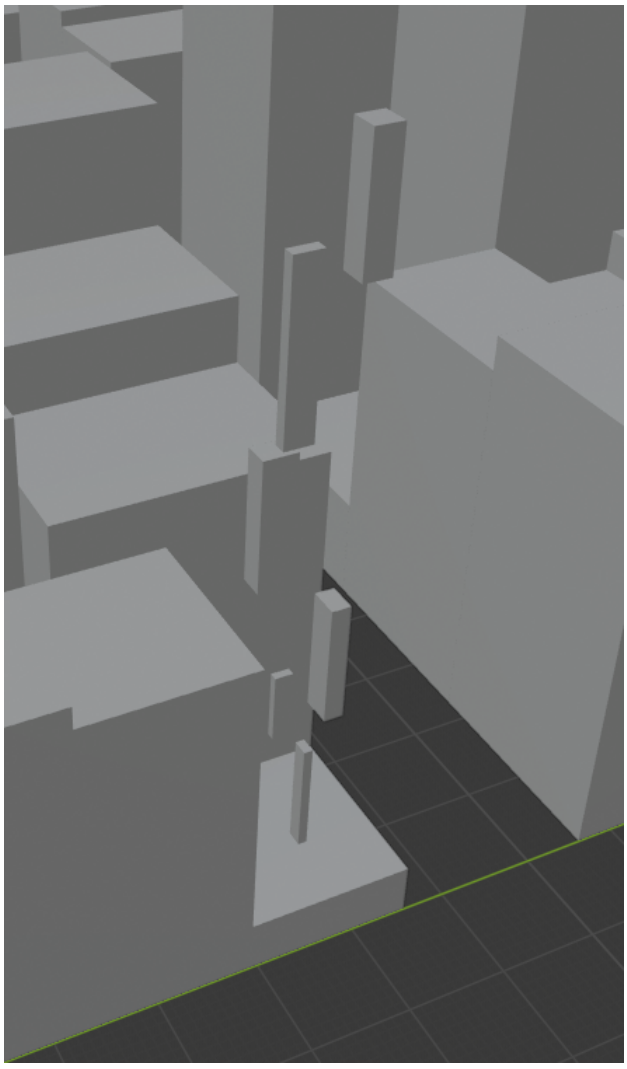


Figure 12.10: The floating cuboids in this building should be arranged in a tiered skyscraper. This was not quite the futuristic look we were going for.

12.4 Extensions

We have already discussed one extension in the form of parameterised rule inference. In the house-with-garage example we also mentioned how it would be interesting to try and model the precise distribution of parameters rather than just using minimum and maximum values to create a uniform distribution within the range.

Although we have covered some good examples, it would be interesting to investigate cases where the example buildings have greater variation. The main reason we have not attempted this is the time it takes to design the example buildings. If we improved our own 3D modelling skills, we could design example buildings manually rather than using a grammar.

As well as parameterised rules, there are other features of our building grammar that we discussed in our forward model but have not attempted to infer in the backwards challenge. Our *Better Roofs* example made use of the rotation operation to reuse roof shapes with different orientations. Our backwards model is not designed to detect rotated primitives so each orientation of a primitive would have to be manually labelled as its own primitive object. The backward model would not be able to detect the similarity between two objects that are identical except orientation. Adding rotation detection would be a fairly simple and worthwhile extension.

Another extension is to attempt to increase the variation in generated buildings by applying some architectural logic. For example, even if we were given example buildings where the garage always appears on the right, then we might guess that the garage could also appear on the left because this 'makes sense' architecturally. Perhaps a model could be created that learns this architectural logic from many different grammars for many different buildings. Given some example buildings, the model could use this logic in addition to the examples to create a variety of buildings. Perhaps the generated buildings could then be manually reviewed and the model could continue to learn and improve.

The possible extensions to this project are exciting and seemingly endless and we are looking forward to returning to them in the future.

Chapter 13

Conclusion

Recall our aims for this project:

1. Define a grammar that allows users to describe a building using a set of construction rules. We will also create a parser that can generate 3D models from a given grammar.
2. Define a method of reverse-engineering a grammar from a set of example buildings. The inferred grammar should be able to create many unique buildings that are similar to the examples.

We achieved the first of these aims in Part 1. We described our building grammar, introduced some of the built-in operations and used these operations to showcase some example grammars and their corresponding buildings.

We also discussed the parser which we have implemented for the grammar. The parser's implementation is designed to make modifications and extensions a simple process. Given a grammar file, our parser can compile the grammar to create an operation graph, our internal representation of a building. We described the structure of these graphs and explained how they are represented with operation classes in our code. An operation graph can then be run on a given scope to generate a .obj file.

In Part 2 we focussed on the second aim. We described the box-splitting problem and outlined potential solutions. We compared the solutions in terms of both theoretical and practical running time. Once we had given a method of creating a single operation graph from an example, we showed how we can combine multiple operation graphs and detect similarities. The resulting combined operation graph could then be used to create buildings similar to the examples.

Finally, we outlined how an operation graph could be translated into a grammar. We explored some examples and evaluated the results by comparing the reverse-engineered grammar to the actual example-generating grammar. The limitations and possible extensions of our method were also discussed.

I have thoroughly enjoyed working on this project. It has been very rewarding having visual outputs from my code and I have had fun experimenting with the different buildings that you can create with the grammar. The final code for the project is over 2000 lines with thousands more of alternative ideas and tests. I have written more than 50 grammars experimented with many more

in the process. All code (including legacy code) and grammar examples can be found on GitHub[1]. I look forward to seeing whether others can create some interesting buildings and use the reverse modelling to speed up their own artistic pipeline. I expect I will return to this project in the future as there is clear potential to take it even further.

[1] `https://github.com/JUSTOM/procedural-buildings`

Bibliography

[1] Saskia Groenewegen. *Procedural City Layout Generation Based on Urban Land Use Models*. PhD thesis, 03 2009.

[2] Hua Liu, Hongxin Zhang, and Hujun Bao. Semiautomatic rule assist architecture modeling. In Lizhuang Ma, Matthias Rauterberg, and Ryohei Nakatsu, editors, *Entertainment Computing – ICEC 2007*, pages 315–323, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, page 105–112, New York, NY, USA, 2007. Association for Computing Machinery.

[4] 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.

[5] 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.

[6] 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.

[7] Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. Inverse procedural modeling of facade layouts. *ACM Transactions on Graphics*, 33, 08 2014.