A System for Generative Building Design

Michael H. Cooper

Master of Science
The University of Bath
March 2018

A System for Generative Building Design

submitted by

Michael H. Cooper

for the degree of Master of Science of the

University of Bath

March 2018

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this thesis has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signature of Author .....

Michael H. Cooper

Abstract

Generating entire building designs using parametric scripts is a powerful tool, potentially allowing architects and engineers to rapidly design and evaluate multiple building designs. Previous research has demonstrated the generation of visually realistic buildings & architectural plans using grammars, but has not demonstrated construction plan generation. The research presented shows that by using a functional programming language to write scripts, rather than a grammar, generation of construction detail plans is possible. To show this the author has built a prototype system in Python and Rhinoceros 3D, however the recommendation for a future system is to instead provide Revit with a Haskell interpreter.

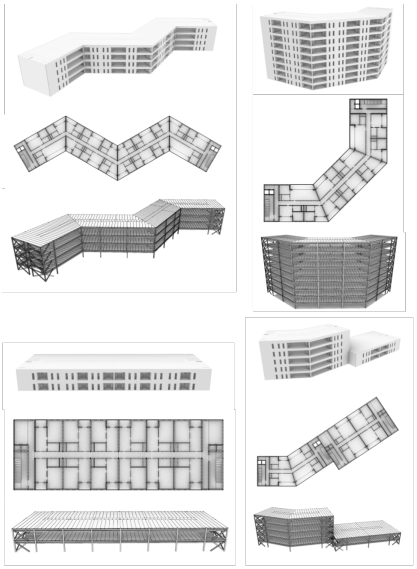


Figure 0-1: Demonstration of a number of buildings.

Acknowledgments

First and foremost I'd like to thank my supervisor Dr. Mac Yang for their help in the project.

Additionally I'd like to thank all of my friends from University, Sam for amazing emotional support, Toby for lovely hugs, Lucas for bringing in his Switch regularly and his humour, Hugo for kicking my a** at Smash Bro on that switch, George for some excellent gossip, Caroline for their amazing chat, and Tom for constantly backing me that the Queen's English is the only English. Of course there's more of you, but my Brain is fried as I'm writing this, so apologies if I haven't name checked you.

Finally I'd like to thank all of my friends and family from back home for keeping me sane, and entertaining me on my recuperation stays at home. Specifically I'd like to thank my twin sister Esme for always being there at the end of the phone whenever I was feeling low. And finally of course my dad, bother, mum and stepdad for looking after me and providing lot's of help to keep me going, I love you all.

Contents

Abstract **4**

Acknowledgments **5**

Contents **5**

List of Figures **8**

1 Introduction **12**

2 Literature Review **18**

2.1 Formal Grammars 18
2.2 Shape Grammars 20
2.3 CGA 23
2.4 Generation of Architectural Plans 27
2.5 Moving Towards Generation of Construction Plans 29

3 Requirements Analysis & System Specification **31**

3.1 Moving Away from Grammars 31
3.2 Applying rules 33
3.3 A more generalised shape 33
3.4 Expanding beyond shapes 35
3.5 Rule Recursion 36
3.6 Linked Objects 36
3.7 Upwards and Downwards Links 38
3.8 Defining object axis 38

3.9 Specification of the Generative Building Design System 39

4 System Design & Implementation 41
4.1 Implementing the System in Existing Frameworks and Software . . 41
4.2 Python Implementation. 43
4.3 Geometric Functions 46
4.4 Links are not Coherent 48
4.5 Scaling back the system 51
4.6 Adding return statements 52

5 System Re-Specification 56
5.1 Functional Programming Languages 56
5.2 Re-specifying the GBDS from a Functional Perspective 59

6 System Design & Implementation as a Functional Language 61
6.1 Rule Re-definition 61
6.2 Geometric Function Modification 63
6.3 Towards better scripts 64
6.4 Rotations are Still a Problem 65
6.5 Implementing Basis 68
6.6 Completed Prototype Details 69

7 Testing 77
7.1 Generating Buildings from Volumes 77
7.2 Generating Building Components 78
7.3 Generating Buildings with Detail 79
7.4 Practical Buildings. 81

8 Discussion 88
8.1 Is Functional Useful? 88
8.2 Fully Parametric Buildings 90
8.3 Writing Scripts in Practice 91
8.4 No Need to Linearise the Design Process 92
8.5 Poor Numerical Control in CAD 92

9 Future Work **94**
9.1 A Production System 94
9.2 Testing the Software on Users 96

10 Conclusions **97**

Bibliography **98**

Appendix **103**
Large Size Images 104
Core Code 109
Matrix Code 126
Building Script 132

List of Figures

0-1 Demonstration of a number of buildings. 4

1-1 Typical Flow for Building Design. 13
1-2 Formal grammar example, with production tree. 14
1-3 Shape grammar example, with production tree. 15
1-4 Comparing city generation. 16

2-1 A diagram illustrating a production tree for a phrase structure grammar (Chomsky, 1956). 19
2-2 Demonstration of road network growth using an L System (Parish and Müller, 2001). 24
2-3 Grammar by Wonka et al. (2003). 25
2-4 Comparison of a real plan with a generated plan, alongside a generated model (Merrell, Schkufza and Koltun, 2010). 29

3-1 Rule Pseudocode 39
3-2 Super-type pseudocode. 40

4-1 Python implementation of the Super-type without comments. Note that the colon means "of type", this is included as a comment for clarity. 44
4-2 Standard form of a rule within Python and terminating rule within Python. By not using an apply the generation sequence terminates. Again the colon means "of type". 45

4-3 Example of an apply function which applies one rule to the first object and the second rule to the rest. Again the colon means "of type". 46
4-4 General form of a geometric function. Again the colon means "of type". 46
4-5 Diaram demonstrating how grid splitting works using the system, note that in the actual result there are no cross links between split objects. 49
4-6 Diagram demonstrating how splitting can leave several potential linking options, consequently choosing which one to use is a problem. 50
4-7 Redefined object specification, removing links, parents, children and axis. 51
4-8 First script demonstrating a very basic parametric building, without any detail. 52
4-9 Generated wall and real wall. 53
4-10 Top are the redefined rule and base rule with return of objects. Bottom is the redefined applyFirst function as an example of how the apply functions work. Note that it collates the outputs of previous rules by adding the returned lists together. 54

5-1 A standard rule formalism within a functional framework using guards. Note that other rule control flows would be possible. This is intentionally syntactically similar to the widely known functional programming language Haskell HaskellWiki (2013). 60

6-1 A standard rule formalism within a functional framework for Python. If/Else statements have been used instead of guards. The syntax of python makes currying cumbersome, consequently it has not been included. 62
6-2 An example of an apply rule now with the predicate If/Else statements removed. 62
6-3 Next phase of script demonstrating a detailed building, however components are restricted to the orthogonal axis. Additionally some key practical features are missing such as doors and stairs. . 64

6-4 Global vs local coordinate systems. Notice that the rotated object has it's own orthogonally aligned axis, these are it's local coordinates. 65
6-5 An illustration of basis. 66
6-6 Object type definition. 69
6-7 In-built rules. 70
6-8 Apply functions. 71
6-9 Debugging functions. 72
6-10 Transformation functions. 73
6-11 Geometric manipulation functions. 74
6-12 Arrange and split functions. 74
6-13 Geometry generation functions. 76

7-1 Demonstrating a very basic parametric building without any detail from the first testing phase. 78
7-2 Demonstrating the parametric metal section wall function, insulation has been removed to see the construction detail. 79
7-3 Generated floor and real floor. 80
7-4 The third phase of testing. The building is detailed but lacks practical features. 81
7-5 Image of similar residential buildings near Canning Town in London. 3D mapping image from Google (2019) 82
7-6 A diagram of the building parameters. 83
7-7 A diagram of how the offset function works. 83
7-8 Greatly simplified recursive loops for generating optimised flat splits. 84
7-9 Building structure including cross bracing. 85
7-10 A diagram showing stair generation and two different height stairs. 86
7-11 Demonstration of a number of buildings. 87

Chapter 1

Introduction

The process of designing a building is a complex one that involves numerous different disciplines. A typical design process would be as follows: firstly the architects would create a set of concept designs which satisfy client requirements. Next the architects would liaise with the clients to develop a detailed plan for the building. These plans are passed on to a variety of designers, including for example civil engineers and interior designers, who finalise these plans into construction drawings ready for the contractors (Halpin, 1998, Designing Buildings Wiki, 2019). Presently most of this design work is done manually using computer based drawing tools, which is a time consuming task (Woudhuysen, 2004). In contrast to the construction industry the film industry is using computers to automatically generate entire cities of buildings for use in CGI effects.

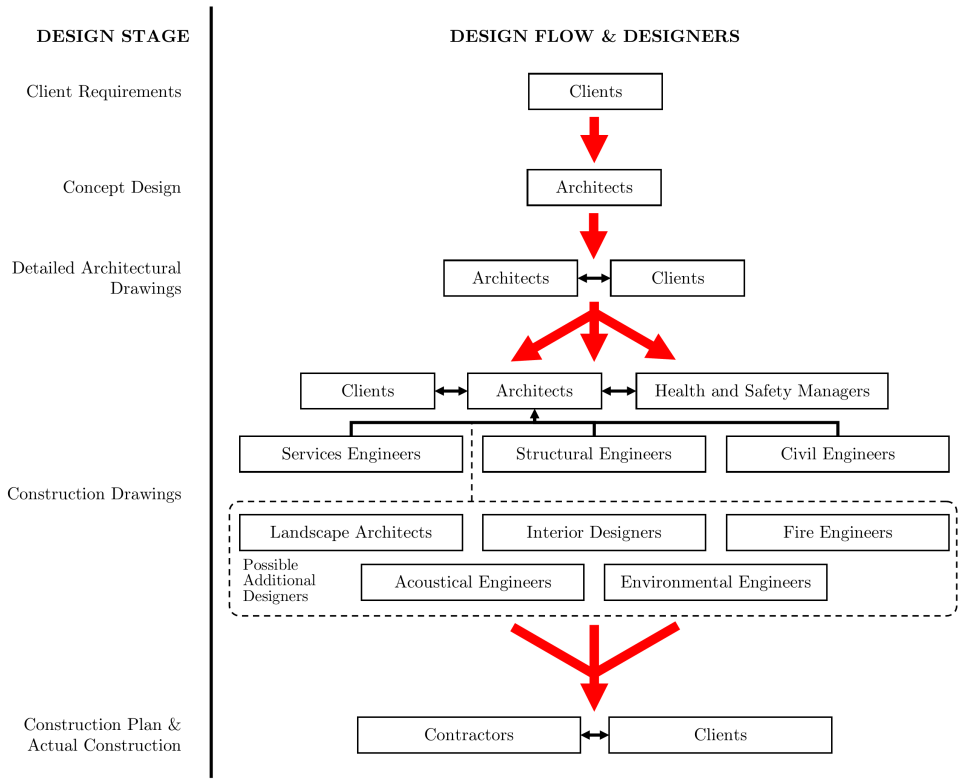


Figure 1-1: Typical Flow for Building Design.

Historically the film industry manually created all of their special effects. For example Blade Runner is known for it's incredibly detailed and realistic looking futuristic cityscapes. however to do this, the artists had to build physical scale models of every individual building by hand, a very labour intensive task (Savage, 2007). Even with the advent of CGI computer 3D artists were still left creating individual buildings by hand only now on a computer. These artists faced the same difficulties of manual design as present day architects and engineers. This problem was overcome by the use of a computational technique called a shape grammar.

A shape grammar is an extension of a formal grammar onto objects. A formal grammar is a set of rules which can be applied to generate a set of strings called a language (Chomsky, 1957). A grammar can be used to generate a string by

iteratively choosing a rule, then applying this on non-terminal symbols, to either generate more non-terminal symbols or terminal symbols. Because grammars are a set of defined rules, the languages they create can have precisely defined properties. A convenient way of visualising a generation process for a word is through a tree, where each node is a symbol and each vertex is an operation Ginsburg (1966).

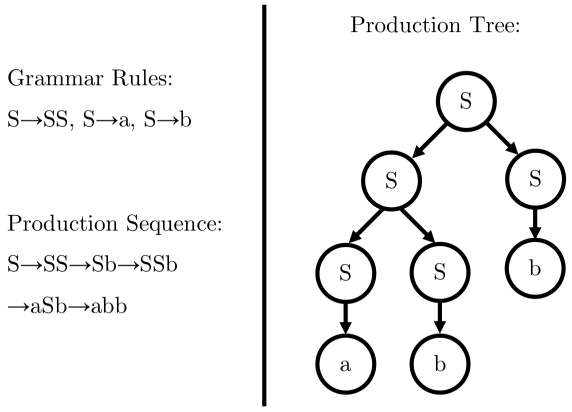


Figure 1-2: Formal grammar example, with production tree.

As the name suggests shape grammars extend formal grammars by operating on shapes instead of symbols. Consequently more complex shapes can generated from simpler shapes by applying a secession of operations, again this process can be visualised in a tree, see figure 1-3 (Hahn, Bose and Whitehead, 2006). Typically this process of generating shapes using a grammar is performed using a computer and 3D modeling software. The method has been demonstrated to create a wide range of different shapes including plants, and importantly for my research, buildings.

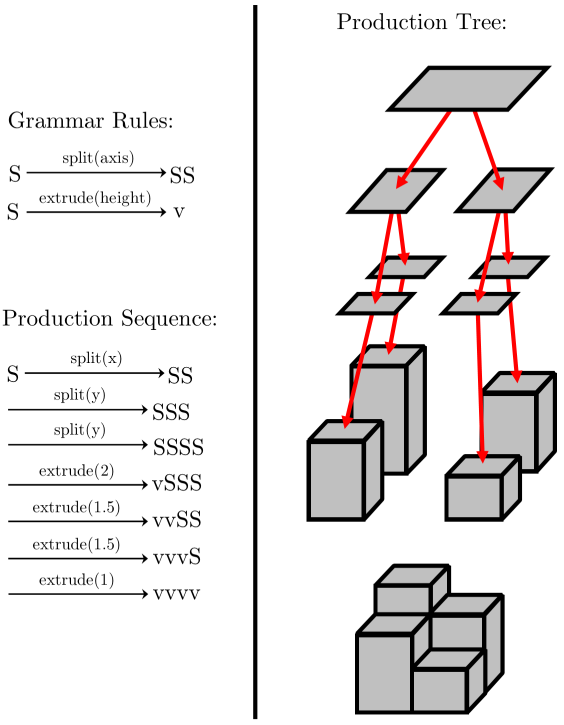


Figure 1-3: Shape grammar example, with production tree.

In 2001, Parish and Müller demonstrated that computers could be used to generate realistic looking cities using shape grammars. In 2006, they alongside other authors managed to push their software into a commercial package called ESRI City Engine (Müller et al., 2006). It is with this technique and often specifically using City Engine, that film 3D artists have managed to greatly reduce their workload, whilst creating ever more detailed visuals. Demonstrating the capabilities of City Engine, the sequel to the original Blade Runner, Blade Runner 2049, did not have it's cities made by hand but instead by a computer algorithm (Richardson, 2018).

(a) bla (1982) physical model built by hand.

(b) bla (2017) City scene generated by a computer.

Figure 1-4: Comparing city generation.

It would be useful if shape grammars could be used to generate buildings for architects and engineers. Most designers will follow a set of internalised rules to design a building then execute these rules to create plans, what a shape grammar would do is allow designers to offload the execution steps to a computer (Müller et al., 2006). Although this means the designers would still have to specify the set of operations to generate a design, once generated it could be rapidly modified by changing only a small number of operations. Contrast this to manual design, which could require a complete redrawing of a design. This leads to two key consequences, firstly productivity would likely increase as detailed design work is automated, secondly designing a large set of similar buildings becomes feasible, allowing for design optimisation.

Furthermore by generating buildings using a grammar numerical and system models could be simultaneously generated to allow for integrated simulation and checking. By doing this the structural and systems engineering could also be automated. This would lead to even greater productivity increases as the roles of the detailed designers could be automated. Additionally by automating the checking process, calculation errors could be eliminated ensuring a higher level of safety over manual designs (Woudhuysen, 2004).

As a building design evolves, the effort required to create the design increases. At the start concept designs drawings are very easy to make. Subsequently detailed plans are much harder to make, but still only require the involvement of the architects and clients. Then finally the construction plans are the hardest to create, involving a multitude of professional designers from different disciplines.

At all stages in this process a tool to generate buildings would be useful, however the utility increases with the detail that can be generated, as this represents a greater effort reduction.

Currently the grammars used in the film industry generate only visually realistic buildings, however for creating concept designs in architecture and urban design this is often fine and is regularly done (Janil, 2019). This represents the first level of utility to the construction industry. Merrell, Schkufza and Koltun (2010), Thaller et al. (2013) and Wu et al. (2018) have demonstrated that grammars can be created which are capable of generating detailed architectural design, representing greater utility. Moving to the next level of utility and construction detail designs is the research's purpose.

The research project then aims to design and then demonstrate a prototype system which would allow users to write scripts to generate construction detail building plans. This would represent the next rung up the ladder of utility for engineers and architects.

Chapter 2

Literature Review

A hierarchy of utility has been proposed classifying what tasks earlier work achieves, and how this is important to engineers and architects. At the bottom of this hierarchy are the grammar scripting tools by Müller et al. (2006), these tools only allow grammars to be made which can generate the hollow shells of buildings. Next up the rung of utility are tools for generating architectural designs without construction details as shown by Merrell, Schkufza and Koltun (2010), Thaller et al. (2013) and Wu et al. (2018). Initially how previous work fits into these two categories is explored in this section. Afterwards previous work and technologies are explored which wither move towards construction level detail or could help in designing such a system.

2.1 Formal Grammars

The concept of a formal grammar was first introduced by Chomsky in 1957 in his book Syntactic Structures. Chomsky's goal was to create a method to formally analyse the structure of languages in linguistic theory, at the time a field unrelated to computer science. To do this Chomsky theorised that grammatically correct sentences could be generated via a set of rules which operated on a vocabulary, calling the vocabulary an alphabet and the rules the grammar. Chomsky further theorised that the alphabet and the rules should be finite, despite the ability to produce sentences of infinite length. Chomsky used this precise definition of to help determine the grammar of a given language without knowing what it was.

The combination of the grammar and the alphabet Chomsky called a phrase structure grammar, today we would understand this as a context free grammar, or CFG.

Chomsky defined phrase structure grammars with a very precise algorithmic operation. Firstly an initial symbol is created, the starting symbol, which is non-terminal. A rule is applied which then replaces the initial symbol with any number of other symbols including the starting symbol, non-terminals and terminals. Rules are then recursively applied on non-terminals including starting symbols until the string has been entirely replaced with terminal symbols. Because the rules replace symbols with other symbols, phrase structure grammars of CFGs are also a subset of replacement grammars.

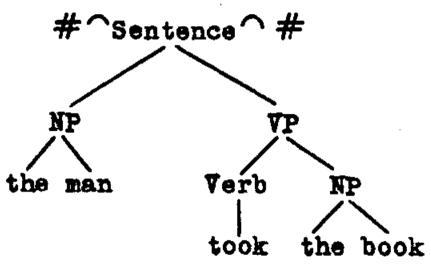


Figure 2-1: A diagram illustrating a production tree for a phrase structure grammar (Chomsky, 1956).

Early programming languages were low level and were in essence a list of operations for a computer to perform, where the computer executes the operations in the order of the program. Backus (1959) realised that the concept of a phrase structure grammar could be used to create a higher level language, where the order in which code is written is not necessarily the order of operation. By creating a programming language which was generated by a phrase structure grammar, an algorithm called a parser could be created to determine the production tree of the programming language and hence the order of operations for the computer to execute regardless of the order in which the code was written. Furthermore the parser at the same time as determining the production or parse tree, can also determine if there are grammatical errors in the code, also called syntax errors. This allows the program to be written in a much more human readable meta-language as opposed to machine instructions. This programming meta-language

was called: ALGOL (Ginsburg, 1966), whose structure influenced many modern languages. The realisation that a phrase structure grammar could be used to create programming languages lead to research from a mathematical basis, leading to the term context free grammar and the widely accepted rigorous mathematical definition of one (Ginsburg, 1966).

$$G = (V, \Sigma, R, S) \tag{2.1}$$

- G is the grammar.
- V is an alphabet of symbols called non-terminals. Rules cannot be applied to non-terminals.
- Σ is an alphabet of terminal symbols, called terminals. These are the symbols rules are applied to. $\Sigma \cap V = \emptyset$ as non-terminals cannot also be terminals and vice versa.
- R is the set of rules. Importantly rules are applied sequentially not simultaneously.
- S is the start symbol and is non-terminal. The first symbol must always be S, however S can be generated. $S \in \Sigma$.

This precise definition of a context free grammar is used by parsers to correctly generate parse trees to convert programming languages into machine instructions, and by Chomsky to determine the grammar of languages without an existing explicit grammar.

Although the symbols typically used in a CFG are single characters, this is not a requirement, for example Chomsky initially intended each non terminal symbol to be an entire word. This fact was used by Stiny and Gips (1971) to develop their concept of a shape grammar.

2.2 Shape Grammars

Stiny and Gips (1971) realised that art could be created using a generative replacement grammar much like a context free grammar, using shapes as symbols.

It is important to mention that they do not generate their images using computers but by following the rules manually. They describe a formal definition of a shape grammar almost identically to that of a context free grammar, their definition has been dited for clarity.

$$SG = (V_T, V_M, R, I) \tag{2.2}$$

- SG is the shape grammar.
- V_T is a finite set of shapes. This is the alphabet of shapes.
- V_M is a finite set of shapes such that ${V_T}^* \cap V_M = \emptyset$. This is the alphabet of markers or non-terminal shapes, and is distinct from the language of shapes.
- R is a finite set of ordered pairs (u, v) such that u is a shape consisting of an element of ${V_T}^*$ combined with an element of V_M and v is a a shape consisting of

  (A) the element of ${V_T}^*$ contained in u or

  (B) the element of ${V_T}^*$ contained in u combined with an element of V_M or

  (C) the element of ${V_T}^*$ contained in u combined with an additional element of ${V_T}^*$ and an element of V_M.

  The rules take a non terminal shape and a terminal shape and replace this with either a terminal shape, a terminal shape and a non-terminal shape or two terminal shapes and a non terminal-shape. Furthermore these rules are applied to any part of the shape which matches u including rotations and reflections.
- I is a shape consisting of elements ${V_T}^*$ and V_M. This is the initial starting shape, equivalent to S in a context free grammar.

In order to generate a shape, they apply the specified rules to all applicable parts of the starting shape simultaneously to create the next layer, then recursively apply the same rules to this layer creating another. This is in contrast to context free grammars which sequentially apply rules. Although the authors demonstrate generating a painting using 2D shapes, they note that the method could be used

with 3D shapes to create sculptures. Part of their motivation to creating such a rigorous formal description of a shape grammar is similar to Chomsky's, which is to be able to determine the grammar given a physical object.

Stiny and Mitchell (1978), realised that Stiny and Gips's definition of shape grammar could be used to generate architecture. Noting that Palladio (1570)'s book, I Quattro Libri dell'Architettura, translating to the The Four Books of Architecture, is essentially a rule book for designing in the Palladian style, they demonstrated that a grammar could implement these rules and generate building plans in the Palladian style. By analysing architecture with grammars, they propose that buildings can have a syntax, a set of rules by which they are designed; this was an important step in opening up the possibility of using grammars to generate buildings. Because they were limited by 2D manual drawing equipment they only generated 2D plans, however again they see no reason why grammars could not be extended to 3D building design.

Independently of Stiny and Gips's work, Lindenmayer (1968) defined a grammar based of a CFG to model algae growth, motivated by a desire to understand how biological entities grow. Lindenmayer's grammar first generates a string of characters using a CFG with given rules then creates a method to interpret or parse these strings into plant like branching structures. This presents a different way of generating geometry to Stiny and Gips. Lindenmayer's grammar essentially grows geometry from a single starting shape, in contrast Stiny and Gips's grammar adds progressive details from a large initial shape. Furthermore Lindenmayer's grammar is designed for computer use and does not require a human to interpret the rules. This form of grammar is called a Lindenmyer or L system

Stiny (1982) in a letter recognises that by implementing a shape grammar on a computer, geometry can be automatically created. to do this they suggest that for computer implementation of shape grammars to generatively produce models, it would be useful to relax the formal restrictions presented in earlier work. Specifically Stiny proposes that shapes are to be considered labelled objects in the sense of an object oriented paradigm and that the replacement rules are reversible, parametric and can generate entire sets of child objects as opposed to two. This recognises the utility of being able to design objects with grammars as opposed to analysing objects. This sweeping set of recommendations led to a widening of the field into design and the definition of a new class, set grammars,

so called because a single rule can generate a set of shapes.

2.3 CGA

Müller et al. (2006) were the first to demonstrate the use of a generative grammar to produce realistic looking buildings and cites. Their work builds upon two earlier papers that the authors developed largely independently of each other. The first by Parish and Müller (2001) demonstrates the generation of cites with mass model buildings, and the second paper by Wonka et al. (2003) demonstrates the generation of facades for buildings. By combining the generation of facades and the generation of mass models, Müller et al. (2006) generated realistic looking buildings.

Parish and Müller (2001) created a generative model for city design, such that they could automatically generate cities. To do this they specify a system of several distinct shape grammars which are used in succession to generate a higher level of detail each time. The first grammar is a predefined set of rules used to create a road network for a virtual city based on geographical maps. The second grammar then generates building lots based on the road network, again defined by the authors. They then use a grammar to generate building mass models, which are 3D untextured shells without any geometric detail. The final grammar then automatically generates textures for the building facades, however these are only proportioned to a single face and are not necessarily consistent over a whole building. The production rules of the four grammars are then interpreted by a program to generate a 3D virtual model of the city. In the paper they describe the program as a parser, this is distinct form a parser used for computer programs; the parser they create interprets the parse tree to create geometry, a parser used in programming interprets the input to create the parse tree. Parish and Müller (2001) were primarily motivated to use their technique for the entertainment industry to create CGI cities, however they also note that the technique could be useful for research and education for example in urban planning.

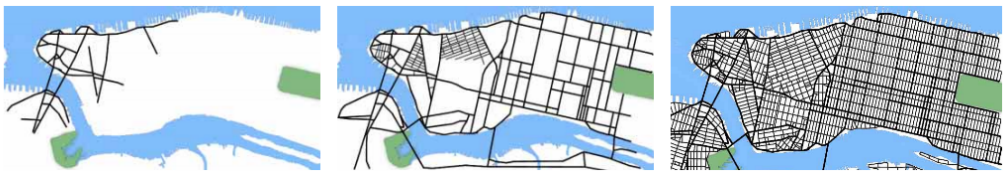
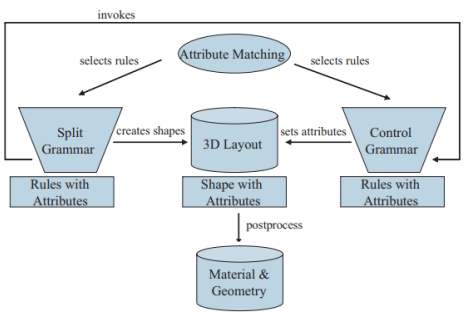


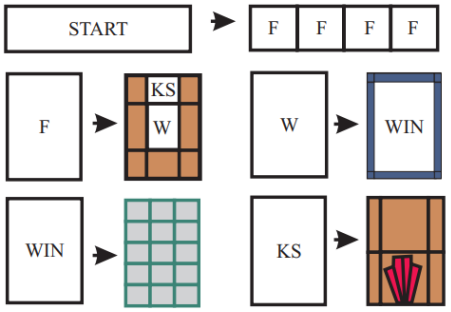
Figure 2-2: Demonstration of road network growth using an L System (Parish and Müller, 2001).

It is important to note that Parish and Müller (2001) use L Systems to generate geometry (Prusinkiewicz and Lindenmayer, 2012). Wonka et al. (2003) propose that an L System is inappropriate for generating most buildings because building design is typically done by subdividing a building mass into floors then rooms then components, becoming ever more detailed, which is in contrast to the growth process of an L System. The propose instead a Split Grammar, where larger shapes can be subdivided into smaller shapes; the much earlier Palladian Grammar by Stiny and Mitchell could be interpreted as one. The split grammar is also a set grammar in that it produces numerous objects from a split (Stiny, 1982)

Furthermore instead of creating a grammar with specific rules to create a limited set of objects they created a database of rules which could be combined with each other to make new grammars on the fly. Unlike earlier grammars which had hard coded rules, the database allows for new grammars to be generated by the computer itself. This enabled the authors to automatically generate facades by implementing a program which randomly chose rules to be added to the grammar.



(a) System architecture developed by Wonka et al. (2003). The spilt grammar produces geometry, and the control grammar decides in what order the split grammar's rules are applied.



(b) Demonstration of a split grammar (Wonka et al., 2003).

Figure 2-3: Grammar by Wonka et al. (2003).

Previously when generating objects through a shape grammar a human would decide in what order to implement the rules of the grammar, either through performing the derivation manually or by hard coding this into a program. Wonka et al. (2003) wanted to instead automatically generate facades and hence had to provide rules for their program to decide in what order to implement the rules of the split grammar, they realised this too could be done by a grammar. This lead to their two grammar system for generating facades. The first is a split grammar with a set of random rules, the second was a grammar that had predetermined rules to decide the order in which the rules of the first grammar would be applied.

CGA shape (Müller et al., 2006) combines elements from both of these earlier works to generate realistic looking buildings. It is one of the few programs to have been released as commercial software, hence it is the most well developed shape grammar for building design in existence, representing one of the most important developments in the field. The authors were specifically motivated by the large amount of time consumed by manually developing models of cities for the films and video games, and sought to provide designers a way in which to generate building geometry using a computer. To demonstrate their point they cite that the number of person-hours taken to produce the city manually in Superman

returns, at the time a contemporary film, totaled to 15 person-years.

It is important before discussing the software further to note that CGA is not itself a grammar, but a scripting language for creating grammars. The system has a number of pre-built functions that can be combined to make rules, which then create a grammar. Consequently the user defined scripts are the grammars not the system.

CGA implements all of the features for a grammar proposed by Stiny (1982). The first feature replicated is to create labelled objects. Shapes in CGA have several attributes, a symbol that is either terminal or non-terminal, a scope which is a bounding box of the geometry aligned to the x, y and z global axis, and an arbitrary geometry.

With CGA the authors enable very generalised grammars with few restrictions on the production rules available, specifically they allow for both generative L-System like rules and Split Grammar based rules. Consequently the scripts are capable of both generating building masses and decomposing them down into facades. Furthermore the production rules are also parametric in that they can produce varying outputs for the same input based on a parameter. They define the core rules as follows:

- Scope Rules: this is a set of rules which are similar to L-System generative rules and either transform existing shapes or generate new ones by altering the scope or bounding box of an object.

- Basic Split: this rule is directly inherited from the set grammar defined by Wonka et al. (2003), allowing for the splitting of shapes into smaller shapes along a given axis.

- Repeat: the repeat rule is similar to split but instead tiles a given number of shapes as many times a s possible along the horizontal axis, the purpose is primarily to create windows.

- Component Split: this creates lower dimensional objects when performing a split, this means that facades can be textured appropriately.

The authors also allow CGA to be extended with new productions rules which can either be entirely new or a combination of other production rules. By having

parametric rules which can create sets of objects they implement two more of Stiny (1982)'s recommendations. Finally instead of removing objects they merely hide them allowing for reversible rules, the final recommendation by Stiny (1982)

The final key difference between CGA's grammars and earlier shape grammars is that they are sequentially evaluated much like a context free grammar, meaning that rules are applied one after the other to generate geometry. This is in contrast to the original specification of a shape grammar by Stiny and Gips (1971), which has all rules applied simultaneously. Because the process is sequential the generation sequence is unambiguous and controllable, this is amenable to a computer as it is told explicitly in what order to perform operations. To control the sequence a priority is attached to the operations to express in what order they should take place, this can either be done automatically by a control grammar (Wonka et al., 2003), or manually.

CGA became the core feature of ESRI CityEngine, a commercial piece of software allowing designers to specify grammars to generate buildings and cities. However as discussed in the introduction CGA can only produce visually realistic shells of buildings, this is not the same as a set of architectural plans which require interior layouts. As a consequence CGA and CityEngine is of limited utility to the construction industry, and is primarily used for concept designs and urban planning, where not being able to produce interiors is less of a problem (Janil, 2019).

2.4 Generation of Architectural Plans

The key step to move from concept designs to architectural style plans is to be able to model interiors. Although City Engine would be capable of modeling interiors by creating a large number of convex primitive shapes, this would be a tedious process.

Numerous papers demonstrate the modelling of interiors. Merrell, Schkufza and Koltun (2010), Liu et al. (2013) and Wu et al. (2018) all develop parametric based modelling of interiors. The papers are primarily concerned with optimising the interior layout to a series of high level requirements or to replicating the environment of a building designed by a human, in order to automate the design process. The research is not focused on the automation of design but the enabling

of generative parametric design, consequently the focus is not on the optimisation efforts in these papers. All three produce usefully detailed plans or 3D models using a parametric computer algorithm. If a rule based parametric algorithm can accomplish the design of interiors then so too should a grammar like program. Indeed, both Thaller et al. (2013) and Becker et al., demonstrate this to be the case.

Much like Stiny and Gips (1971), Becker et al. are motivated by a need to analyse objects, specifically building interiors leading to them designing a shape grammar to generate floor plans. The generated floor plans are detailed but limited to being 2D drawings. Although 2D drawings are what are ultimately required for architectural plans, buildings are inherently 3D and it is unclear if the grammar could be used to design coherent plans over multiple floors.

Thaller et al. (2013) realised that by using a cuboid scope for it's shapes, CGA was limiting itself in the rules it could apply, instead they propose using convex polyhedra bounding boxes. A split operation in CGA does not split the geometry but the bounding box, consequently splits can only take place orthogonal to the scope or bounding box's axis (Müller et al., 2006). By using convex polyhedra, Thaller et al. (2013), could create more generalised split rules allowing for splits in any plane or with any other convex polyhedra. This enabled the authors to create functions to generate concave shapes consisting of a set of convex shapes, hence generating interiors to buildings.

Although more powerful than CGA, both still rely on approximating objects with bounding boxes instead of using the true geometry for their production rules' operations. For example it would be impossible in either of the systems to split a non-concave, non-compound shape, such as a torus.

Furthermore all of the grammars presented take a polygonal based approach to drawing, whereby geometry is represented by straight lines and flat surfaces. For example if one were to approximate a circle using an octagon, as with a polygonal approach, one would find the diameter of the circle varies depending on whether one measures from vertex to vertex or face to face; this fact is true regardless of how precisely one approximates a circle. As a consequence if applying shape operations such as unions or splits the intersections of shapes may not occur where expected, and successive operations such as what would happen with a shape grammar could exacerbate this. A computational method would be required

which can model any arbitrary curve, surface or solid exactly.

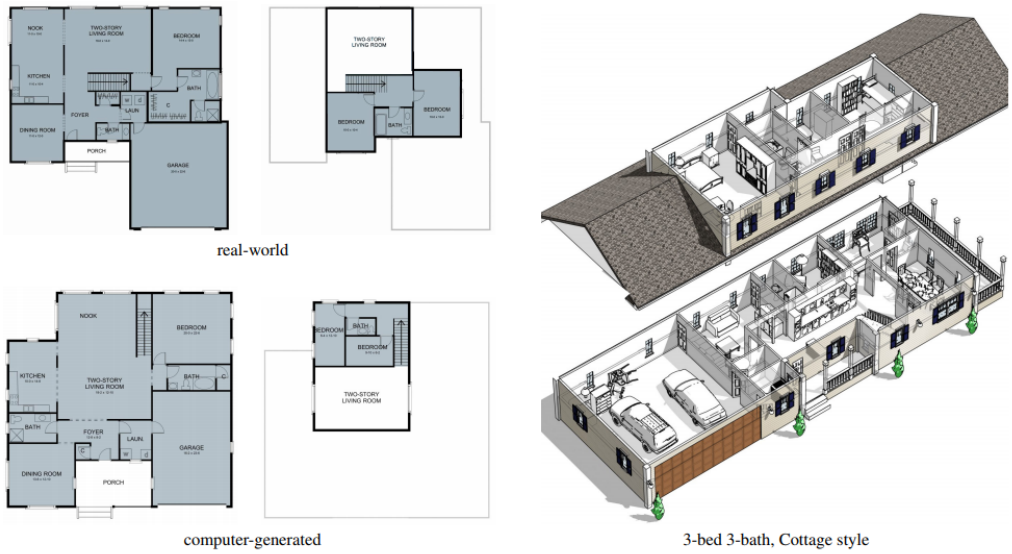


Figure 2-4: Comparison of a real plan with a generated plan, alongside a generated model (Merrell, Schkufza and Koltun, 2010).

Although the papers described in this section demonstrate the capability of designing architectural plans with detailed interiors, they are limited; none of them demonstrate the accurate generation of curves, all being polyhedral based, nor does Thaller et al. (2013) demonstrate true concave shape generation. Despite this they represent a much higher level of utility to CGA as architects could use the demonstrated methods to produce architectural design plans, even if they are restricted to being polyhedral.

2.5 Moving Towards Generation of Construction Plans

The purpose of the project is to move generative design to the next level of utility by creating a prototype system which allows users to write scripts to generate construction level detail. Some authors have so far demonstrated progress towards this goal, however no prior research has been discovered that comprehensively achieve it. In this section Kreo Design by Kreo Software Limited is discussed, as this is the closest to achieving this goal.

Kreo Design, although in a testing phase, is demonstrating the capability to automatically design structures to a high level of detail. In particular it will perform structural and some architectural design for steel frame, concrete beam/slab or precast buildings, which are restricted to being extrusions from a given floor plan. Furthermore it will also perform internal room layout. The software will generate a detailed BIM model for all above ground structural components, including rebar design for the concrete. Despite it's detailed output it is not fully comprehensive, specifically it does not go beyond basic architectural and structural design, for example it cannot do services design.

Furthermore Kreo Design is very restrictive in what users can produce, as it can only produce buildings with a linearly extruded floor plan made of blocks of structure without links between them. For example if creating a building with two lobes Kreo Design cannot put a corridor between the two lobes, in addition buildings which change their floor plan with increasing height are not possible. Finally Kreo Design is intended as an almost fully automatic design software, consequently users are not given full control over the process as would be in a grammar such as CGA.

Kreo Design suggests that it might be possible to develop a grammar beyond what has been demonstrated, however this is uncertain as the company do not in any way explain the underlying processes behind the software. In conversation with a representative of the company they indicated that they used a form of parametric modelling, a hard coded grammar, however there is no direct evidence of this.

Despite Kreo Design's apparent success, there is still clearly room to improve. Moving towards the proposed prototype system will require new ideas to be incorporated and realised.

Chapter 3

Requirements Analysis & System Specification

From the literature review it is clear that there is a gap in the work done. Previous work has demonstrated the ability to model visually realistic buildings and architectural interiors, but fails to demonstrate generating the detail required for construction plans. This gap in the work is perhaps not due to a poor research, but probably simply because no one has tried to create a system to write scripts to generate buildings. From this gap in the research the projects task of designing a system for scripting buildings was chosen.

In order to design a system specification an analysis of the various components it might need to incorporate has been included, alongside a discussion as to why these components are different from previous work.

The first topic discussed is perhaps the most key. Previous tools have focused solely on allowing users to script grammars, however it is wrong to simply assume that grammars are the right tool for the job.

3.1 Moving Away from Grammars

As discussed in section 2.3, CGA is not itself a grammar but a scripting tool to enable users to define grammars, however it would be inaccurate to define CGA's grammars as that. When compared to the precise definition of a shape grammar by Stiny and Gips (1971) there are several critical differences. Firstly A

shape grammar has an finite alphabet of shapes which may be generated. CGA can generate an arbitrary number of different shapes. Secondly the rules do not work like those in a shape or formal grammar. In the latter the rules work by pattern matching, using pairs to specify transitions from one shape in an alphabet to another. This is a non-deterministic process. GCA in contrast must be a practical program and hence generate objects deterministically. To do this pattern matching is used but priorities must be specified by users to ensure that generated objects have the correct rule applied.

Although CGA does not generate shape grammars in the traditional sense, it does not mean it is not a usefull program. Instead it is more appropriate to focus on why it provides a logical platform for parametric design. Using tree based derivations CGA creates a clear structure in which to generate a building, for example it makes sense to split a wall into floors, then split floors into windows and doors, creating a branching structure. By doing this CGA allows designers to follow a recursive order of generation which fits with what they might do manually.

By relinquishing the use of a grammar and instead focusing on the core functionality, much of the formal rigor of CGA can be removed, leaving a simpler program that can still do everything CGA does. Instead of using the prioritised pattern matching formality used by CGA, it would be much simpler for rules to directly call other rules and pass their outputs as arguments. CGA's grammar form is intended to allow one rule to be written for all cases, however this can be achieved in the direct passing form by allowing multiple rules to call a single other rule.

By moving towards direct calling of rules, scripts can be made more clear and computationally efficient. By writing scripts in a direct style, users know exactly what rules will be called all of the time, unlike the roundabout priority style used by CGA which could become confusing. Furthermore because the computer is not searching for unused objects from a list, and instead is being directed to the next rule, the process of generating has fewer computational steps and is hence more efficient.

3.2 Applying rules

Using rules in a passing manner is a logical step away from the rules of grammars however a new structure to use them needs to be devised. It would be impractical for every rule to be written from scratch, instead it would be more usefull to construct rules from already built code.

Unlike previous grammars a distinction now needs to be made between rules and geometric functions. Because rules now pass to other rules directly rules must have unique names even if sometimes they do the same thing. Consequently it would be usefull to package up very common geometric operations such as a split or a move into a function with parameters. This means that functions can be called to operate on objects before the new set is passed to another rule.

Rules will now be passed an object which is then operated on by a geometric function and then pass on it's new object to another rule directly. However this leaves a problem for geometric functions which produce multiple objects from a single input such as a split. To solve this a set of higher order functions could be created which apply a rule with all of the objects created by the geometric function. These functions would then behave much like "map" functions seen commonly in various programming languages.

It would be good to implement a series of these "apply functions" so that various different use cases are implemented. Map functions typically apply a function to all elements in a list, however in a system using this form it might be common for users to want to apply one function to the first element in a list and another to the rest. Because of this there should be a series of commonly used apply functions created.

3.3 A more generalised shape

The proposed system should be able to generate and handle arbitrary shapes, especially solids. Doing so would allow for the generation of complex details, objects which are almost always solid and often convex. To enable this to happen two key steps must be taken, the first is to remove the concept of bounding boxes. The next is to ensuring that objects are not polygonal, but instead specified exactly.

Earlier grammars typically work with polygonal mesh based objects (Müller et al., 2006, Thaller et al., 2013). The motivation for this is clear, as they are only intended for computer graphics, a hollow shell is adequate. Furthermore by working with hollow shells formed of polygonal surfaces, they make the program much more computationally efficient. However with the intent to create detailed construction models it would be usefull to be able to work with arbitrary shapes, including curves, surfaces and volumes that are defined in an exact manner. Doing so prevents errors from occurring when generating smoothly curved geometry.

Both Müller et al. (2006) and Thaller et al. (2013) use bounding boxes to define how geometric operations work. Thaller et al. realised that the simple cuboid bounding boxes by Müller et al. were limiting, however the entire concept of bounding boxes is limiting as discussed in section 2.4. It is proposed to simply remove bounding boxes. However this introduces a problem, the purpose of bounding boxes was to provide references on how geometric operations should work, for example splitting a volume is different to a surface, because the bounding boxes have been removed another method must be devised.

For geometric functions to work properly with different shapes it would be better to introduce shape types. Many of the programs developed earlier introduce different classes of shapes to do different problems with the examples of the different types of bounding boxes by Müller et al. (2006) and Thaller et al. (2013), however by trying to find one shape which does everything they inevitably approach road blocks preventing certain kinds of geometries being generated. Instead of attempting to create a single type which does everything, it would be better to introduce types descended from a single general super-type. This would mean that certain rules such as moves, rotations and scales can be applied to all objects, yet type specific functions can also be created. For example a split function works differently depending on whether curves, surfaces or volumes are passed to it.

Introducing types could cause confusion on behalf of users writing scripts, as it could be difficult to keep track of what object type is passed to and returned from a particular geometric function. For this reason it is proposed to implement type checking and type declaration into scripts much like the languages Java and Haskell (HaskellWiki, 2013, Gosling et al., 2000). Although typically these are done for compiled languages to locate error before compiling there is little reason

it cannot be implemented on a scripting system to help prevent and locate run time errors.

3.4 Expanding beyond shapes

By introducing types into the program a new opportunity is opened up. Current grammars are solely based on geometry so each objects has an predefined set of attributes. Again although not an issue for computer graphics, it would be helpful to be able to create types with non-geometric information attached. This would move the program from not just being able to generate a building's geometry but a detailed Building Information Model or BIM model. This would represent an even higher level of utility to designers than simply creating construction detail plans.

A BIM model not only represents the geometric characteristics of a building but also the functional characteristics alongside information necessary for construction (Azhar, 2011). Consequently objects in a BIM model have geometries, object related properties, purchasing costs and information about how it fits in a construction plan. BIM models then are used as a digital replica of a building from construction through to use and end of life.

Azhar (2011) show that Building Information Modelling is increasingly being used in the construction industry. However they are difficult to create as BIM models require a very high level of detail, and attaining this by hand is time consuming. If the system can also create BIM models and not just detailed geometric models then the system will allow for an even grater amount of work to be saved and hence represent a much more usefull program to designers; custom types should allow this to happen.

Non-geometric information included in types could for example include what the cost of a given object of a given size is; this is a common piece of extra information attached to objects in BIM. By doing this it could be possible for example to use the system to optimise the cost of a building by rapidly generating successions of slightly different buildings and then comparing them. Using a passing style this could be achieved automatically with a minor alteration to the rules by allowing rules to recursively call themselves.

3.5 Rule Recursion

Implementing loop like control is useful. CGA implements a rudimentary form of iteration yet this is cumbersome. CGA++ as demonstrated by Schwarz and Müller (2015) allows for enhanced explicit algorithmic control of the production tree by enabling the temporary halting of branches whilst other geometry is created, yet this is still grammar based, and as a result is an ad hoc solution. Allowing rules to call themselves is a much simpler method of enabling loops.

Rule recursion could be a very powerful tool, allowing functions to loop around until finding optimal solutions to problems such as cost or structural design. Unfortunately simply allowing functions to recursively call themselves is not adequate, as there is no way to terminate a recursive rule.

A method to implement a base case is to only pass an object onto another rule if a predicate on the object is satisfied. A predicate is a function which would be passed an object and then return true or false depending on chosen characteristics of the object. This enables users to set up situations whereby a rule is recursively called until a condition is not satisfied at which point the rule passes an object to a different function instead.

3.6 Linked Objects

Within grammar based design it is often useful to determine if two objects are attached to each other. CGA uses the concept of a scope or bounding boxes to allow for queries to be created to determine whether certain parts of the geometry intersect or not. Thaller et al. (2013) use convex polygonal hulls to allow for a wider range of geometry whilst still using the same query mechanism. Finally Krecklau and Kobbelt (2011) and Krecklau, Pavic and Kobbelt (2010) use a type of object called a tri-linear free-form deformable shape, which is a polygonal mesh that can be deformed into any shape, although potentially concave it cannot have holes. Whilst detecting intersections can be done for NURBS geometry, it is propose that instead of relying on implicitly discovering neighbouring geometry a better approach would be to explicitly define neighbouring geometry.

Linked lists are one of the most fundamental data structures within programming Parlante, they operate by directly pointing one element to the next element,

using this data elements can be explicitly connected to each other regardless of their location in memory. Linked lists can be expanded to become two way such that an elements connection to another is stored for the other element too, such that lists can be traversed in both directions. Furthermore linked lists can be expanded to higher dimensions such that a single element can link to multiple other elements. It is believe this concept could be used to explicitly link geometry together, for example, such that a beam can be directly linked to a bolt which is then linked to a column. This method of using linked lists would allow for computationally efficient way in which to discover neighbouring geometry. Krecklau and Kobbelt (2011) create a basic single dimension version of this with their tri-linear FFDs, however they do not allow links to multiple elements.

By allowing a generalised object with attributes linked lists can be implemented in the framework. Furthermore the explicit linking of objects creates a natural way of representing buildings as a whole. With computer graphics it matters little whether objects are actually connected, however for construction it does, it is important to know what is connected to what and how they are connected. For example pipe networks, cabling networks, and a buildings structure are all linked systems. Adopting this system would allow users to directly query endpoints of pipes in contrast to only being able to query neighbouring geometry.

By having the ability to automatically link geometry there exits a natural way to create linked systems, this allows for easy integration of existing design software for these linked systems. Present software lacks the ability to design the layout of linked systems but given a layout will decide what objects need to be used to create the system. For example Tekla Structural by Trimble will perform structural design and optimisation according to Eurocodes or American Standards if it is provided with an initial design. Further examples include Bentley Systems Raceway Design and Cable Management Software which design cable networks and Building Mechanical Design Software which designs M&E and HVAC systems. Consequently structural and systems design could likely be integrated with the system.

A linked system of objects could also make the generation of certain objects more logical too, for example the generation of walls. To illustrate, in the system, two cuboids which are adjacent to each other and which share a common face, would be linked via an object representing this surface; this would then be a chain

of three objects, cuboid, shared face, cuboid. From this it is trivial to insert a wall with thickness between the two volumes. Firstly the cuboids would be shortened by half the thickness of the wall, then the wall would be inserted with the same width and height as the surface. Finally a new surface would be inserted between the wall and the cuboids representing their now shared surfaces. Although a multistage process, it is analogous to inserting a new element in a linked list.

3.7 Upwards and Downwards Links

Keeping track of links could be a very usefull feature as it allows for efficient algorithms to be made to traverse along objects. Equally importantly it would be usefull to create functions which efficiently traverse up and down the generation tree. To enable this each object should keep lists linking each object to it's parent and all children.

These links would be implemented similarly to the horizontal links previously described. However unlike the horizontal links they directly link to their parent and all children. To enable this to feature to function properly objects cannot be deleted after they have been made, instead they should simply be hidden. To keep track of whether an object is hidden or not requires a hidden object marker, this ensures that the object is only displayed when appropriate.

3.8 Defining object axis

Scopes or bounding boxes were deemed to be problematic due to their restrictions on the types of geometry that can be generated, meaning that the should not be included in a system. This led however for the need of links to replace intersection algorithms. However the scopes were also used to determine an objects orientation for proper algorithm application. Object types are not a wholly adequate solution, whilst types can direct functions to use different algorithms, they cannot deal with rotated objects. A solution is presented in keeping track of objects axis.

An axis marker would consists of a rotation and translation vector, which would keep track of the objects current orientation. By doing this functions could rotate an object out of it's global axis and into a local axis, hence enabling functions to work properly.

3.9 Specification of the Generative Building Design System

From the analysis an initial outline for the system was devised, using the core components previously described. The system should be a scripting language that can generate arbitrary geometry in some kind of visualisation software. Furthermore all geometry should be represented exactly within the system

Next the primary feature is the structure of a rule in which they are first passed an object, then replace that object with one or more objects using a geometric function, and finally pass on the outputs to other rules using the apply functions.

```
ruleName : inputType -> outputType:
    geometricFunction( input )
    applyFunction( rule-0, predicate-0,
                   rule-1, predicate-1,
                   ...
                   rule-n, predicate-n )
```

Figure 3-1: Rule Pseudocode

Alongside the specification for the rule, an object super-type must also be specified. Summarising the features, the object should have several attributes: firstly the object should have a geometry; next it should have horizontal links which are a list of other objects; it should have a list containing it's parent, although most will only have one parent it should be able to have more than one; it should have a list of it's children; it should have an indicator of whether it's hidden or not; and finally it should have an axis to keep track of the objects position.

```
type Object:
    geometry : Geometry
    links : [Object]
    parent : [Object]
    children : [Object]
    hidden : Bool
    axis : Axis

type Axis:
    rotation : Vector - Order 3
    translation : Vector - Order 3
```

Figure 3-2: Super-type pseudocode.

These two components then create the foundations for the system. Although new types can be specified with extra attributes, the super-type represents the minimum required information for an object, leaving all sub-types inheriting from this.

Beyond the rule specification several inbuilt functions need to be included. The first is a series of geometric functions to handle geometry, such as splits ad moves. The next set are a series of apply functions which map rules and outputs to each other.

Chapter 4

System Design & Implementation

Implementing the Generative Building Design System, GBDS, proved to be much more difficult than simply following the recommendations set out in the requirements analysis and system specification. It was initially believed that links would be a very powerful tool, however it became clear throughout the initial development that they were untenable.

The first step taken in implementing the system was to decide on a framework to physically build tit within. Although custom visualisation software and a custom scripting language could have been built, this would have been difficult and overly time consuming.

4.1 Implementing the System in Existing Frameworks and Software

Instead of creating a system from scratch including all of the code, geometric operations, and the software to display the generated geometry, it was decided to use existing frameworks. To prototype the system a combination of a programming language and CAD software was used. By using existing software the only focusing on the features of the GBDS needed to be done, instead of features such as displaying geometry, which is what CAD software is designed to do.

Choosing a programming language to work with was an important task, two options could have been taken. The first would have been to implement the system via a custom built interpreter, which would have allowed for any variation of language to be built exactly as specified in section 3.9. The second option was to use an existing programming language with sufficiently similar features to the GBDS that it could be implemented as a module including the extra functionality to write scripts in the existing programming language

Although creating an interpreter would mean that the GBDS could be implemented exactly as described, it would limit the capability of GBDS scripts to that which is included only in the interpreter. This would have slowed down the development of creating a prototype of the GBDS, by creating roadblocks for even simple tasks such as arithmetic. Furthermore changes in how a script was created would have required significant changes to the interpreter. Finally when writing scripts users tend to want to work with familiar systems (Meyerovich and Rabkin, 2013), creating an entirely new scripting language for the GBDS would have been counter to this.

Instead it was decided to go with the second option, using an existing language and simply adding a module to it for the extra functionality. This allows for a much more rapid development process by allowing for GBDS scripts to be written in the same language, meaning all of the tools that come with a full programming language could also be used in scripts, allowing on occasion temporary hard coding to move on development.

The language needed to have several features such that it could be used to write scripts as set out in chapter 3. Firstly the language needed to not be wholly object-oriented such as Java (Gosling et al., 2000), instead it needed to be at a minimum multi-paradigm to allow for free functions that can be written in scripts. Next it needed to have type checking and type declaration. Finally the language needed to interface with an existing CAD program.

Computer graphics and geometric manipulation of objects is a formidable task and is an entire discipline of computer science on it's own, fortunately a class of programs is dedicated to doing just that. CAD software is specifically designed to enable designers to create and manipulate arbitrary objects and display them within a GUI. This satisfies the other part of the requirements for the GBDS which is generating, modifying then displaying geometries. Fortunately a number

of CAD programs interface with commonly used programming languages, allowing for the other part of the GBDS to be built.

Ultimately a combination of Rhinoceros 3D and Python was chosen for the framework. Python as a language satisfies most of the requirements. As python is a multi-paradigm general purpose language it meant that any form of GBDS scripts could be chosen allowing for flexibility in the design of GBDS. Furthermore it also allows an object oriented approach two, allowing the creation of types. Unfortunately python is does not have type declaration. From a users point of view this is unhelpful as it means that it is much easier to make errors, however because it is only unhelpful and not a critical feature it can be left out of the prototype. Furthermore because python itself is interpreted, debugging script errors is easier as debugging can take place line by line, checking each geometric action step by step

Rhinoceros 3D as a CAD program was chosen partly because it interfaces with python, but also partly because it satisfies the requirements for exactly representing geometry as set out in section 3.3. Rhinoceros 3D is a NURBS, non-uniform ration b spline, editor, as a result curved geometry is represented as exactly as a computer can by using a mathematical equation. This moves the system away from the polygonal meshes use in earlier work by Müller et al. (2006) and Thaller et al. (2013)

4.2 Python Implementation.

Following the framework set out in section 3.9 the specification for the object super type was then recreated in python using classes as types.

```
class Obj:
    def __init__(self, geom, parents = [], children = [],
                 links = [], axis = Axis(),
                 hidden = False):
        self.geom     = geom     # : Geometry
        self.parents  = parents  # : [Obj]
        self.children = children # : [Obj]
        self.links    = links    # : [Obj]
        self.axis     = axis     # : Axis
        self.hidden   = hidden   # : Bool

class Axis:
    def __init__(self, rotation = (0.0,0.0,0.0),
                 origin = (0.0,0.0,0.0)):
        self.rotation = rotation # : (float, float, float)
        self.origin   = origin   # : (float, float, float)
```

Figure 4-1: Python implementation of the Super-type without comments. Note that the colon means "of type", this is included as a comment for clarity.

Python allows for an almost exact replica of the super-type specification to be created, however as mentioned in section 4.1, there is no capacity for type checking attributes. Furthermore default values for certain attributes were included for convenience. It is important to note that for simplicity all objects were initially of the super-type. Consequently no sub-types were created. Using the class based definition of objects a general form for rules was devised.

```
def ruleName(obj): # : function(Obj)
    applyFuction(rule1, predicate1,
                 rule2, predicate2,
                 ...
                 ruleN, predicateN)(geometricFunction(obj))

def ruleName(obj): # : function(Obj)
    geometricFunction(obj)
```

Figure 4-2: Standard form of a rule within Python and terminating rule within Python. By not using an apply the generation sequence terminates. Again the colon means "of type".

The Python code is largely similar, with the recursive nature of rules remaining the same, specifically the use of predicate functions which evaluate to true or false based on an ouput object is identical. However, Python required several differences from the pseudocode: firstly rules are defined as individual functions; secondly python does not allow for type declarations, as a result this was not included; finally the apply functions work as higher order functions, this means that the geometric function outputs are directly used as the arguments for the apply functions.

Apply functions use function composition to map rules and predicates to the outputs of geometric functions, with rules only being applied to an object if the predicate evaluates to true using the object as an argument. In this way they are analogous to map functions in many programming languages.

```
def applyFirst(ruleA, predicateA,
               ruleB, predicateB):
    def f(objs):
        if predicateA(objs[0]):
            ruleA(objs[0])
        for obj in objs[1:]:
            if predicateB(obj):
                ruleB(obj)
    return f
```

Figure 4-3: Example of an apply function which applies one rule to the first object and the second rule to the rest. Again the colon means "of type".

The apply functions create a function who's argument is a list of objects output by a geometric function, and then applies rules to the list in a specific manner. The rules and predicates are supplied as the arguments from the apply function. Finally the output list can be of length one or more, but not zero, this prevents passing no objects onward, hence throwing an error.

4.3 Geometric Functions

The object super-type definition, rule and apply formalism represent the core features of the GBDS system, however using these nothing can actually be built, this requires geometric functions to create geometry.

```
def geomFuncName(obj): # : function(Obj)
    # Code to generate a list of new geometries of length
    # greater than or equal to zero.
    hide(obj)
    return objs # :[Obj]
```

Figure 4-4: General form of a geometric function. Again the colon means "of type".

A geometric function take in an object then returns a list of objects of length one or more. Before returning it hides the input object allowing the outputs to supersede the input without deleting it. Several key geometric functions were identified from Müller et al. (2006), Thaller et al. (2013) and Krecklau, Pavic and Kobbelt (2010) that appeared to be usefull to building generation namely:

- Split, this function takes geometry and splits it using planes equally spaced along a direction.
- Arrange, this function takes geometry and copies it along a direction using equal spaces.
- Move, this moves an object.
- Rotate, this rotates an object.
- Extrude, this takes a planar surface an extrudes it linearly in a given direction.
- Copy, this is used as an internal function allowing geometry to be copied to prevent modification of original objects.

It was decided to initially implement these functions then attempt to generate buildings in order to discover if these functions represent what is generally needed to design buildings or if more need to be implemented.

The move function was trivial to implement. The move function is supplied an additional argument of a 3D move vector, and then returns a single object within a list. All features are preserved aside from the geometry which was linearly translated and the axis origin vector which was added to the move vector. The extrude function was similarly trivial. Like the move function it took in a vector specifying the extrude direction and length and then returned a single object. The extrude function used Rhino's inbuilt extrude function to manipulate the geometry. All features are preserved including the axis origin vector.

The rotate function proved problematic. Rotating objects in 3D requires three distinct angles unlike 2D which only requires one. Several conventions exist for rotating objects in 3D as summarised by Diebel (2006), hence it is important to chose a convention that is easy to understand for users and is regularly used in

other CAD programs. Perhaps the easiest to understand and commonly used in engineering is the Tait-Bryan convention using roll, pitch and yaw, the same as for aircraft (Nasa, 2015). In this convention 3D objects are rotated first by rotating them counter-clockwise about the x-axis, roll, then CCW about the y-axis, pitch, and finally CCW about the z-axis, yaw. The roll, pitch, yaw convention was used because of it's easy to understand and hence user friendly nature. Rhinoceros 3D's inbuilt functions made the implementation of rotating the object trivial.

All of the attributes aside for the axis rotation vector are then preserved. However how to actually keep track of the objects rotation using the rotation vector was not clear, unlike linear translation. This is a result of rotations in 3D not being commutative, i.e. the order of successive rotations matters in where the objects end up (Diebel, 2006). Consequently keeping track of an objects rotation from it's initial position using an order three vector is not a trivial problem. This lead to the rotate function being unable to be implemented properly at this stage. Not only did the rotate function prove problematic but so too did the arrange and split functions.

The arrange function was the easier of the two. Similarly to the rotate function implementing the geometric part was not difficult, all that was needed was to copy then move objects within a loop. Furthermore initially this was only implemented in the orthogonal directions for simplicity. However how to link objects was not clear. Although at first the solution seems to be to link each successive object in a chain this is not helpful. For example of a user intends to create a grid of objects then each object can is only then linked in one direction as cross links cannot be made. Furthermore questions begin to arise about how to determine if objects are actually linked or not.

The split function suffers from the same problem with links as does the arrange function. Whilst implementing the geometric manipulation was relatively easy, understanding how to implement links became more difficult.

4.4 Links are not Coherent

It is worth revisiting the motivations for including links. Links were to represent geometric connections between objects, primarily shared surfaces. This firstly provides a natural method for generating geometry that exists at the boundaries

of two objects such as a wall between volumes and secondly allows for the creation of computationally efficient traversing functions. If implemented they could be useful, however it is unclear how to do this.

The primary problem with links is the grid issue. Creating the links for splits or arranges in 1D is trivial, each object is simply connected to the prior in a chain. However linking the results of splits or arranges in 2D and 3D is not trivial.

To perform splits and arranges in higher dimensions to generate grid like geometries each function is recursively applied on the results of the previous split or arrange. For example a grid could be created by performing a split in the x-axis, then another split in the y-axis on each of the objects resulting from the x-axis split. This is the crux of the issue, because generation happens in a tree like manner it would be very difficult to generate links across different dimensions, without using purpose built functions.

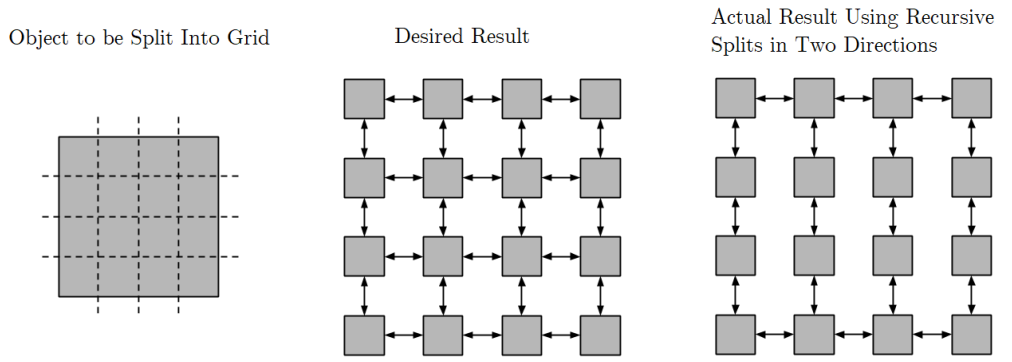


Figure 4-5: Diaram demonstrating how grid splitting works using the system, note that in the actual result there are no cross links between split objects.

It's not simply the problem with generating links across dimensions but also an issue with how to determine which objects need to be linked. Consider another example, a split in two is performed in the x-axis on a box, then two more splits in the y-axis. The first y-axis split operates on the first result and splits in two, the second operates on the second result and splits in three. Given that the objects no longer share surfaces how then would an algorithm decide what objects should link and what their shared surfaces are. This problem is illustrated in figure 4-6. The only way to do this generally is then to use algorithms which determine the

intersection of objects.

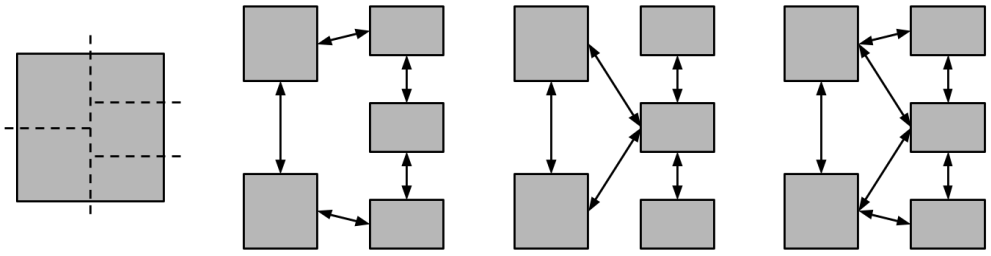


Figure 4-6: Diagram demonstrating how splitting can leave several potential linking options, consequently choosing which one to use is a problem.

Finding the intersection of two objects is computationally very expensive, negating the point of links in the first place which was to create faster functions and generation. Furthermore deciding what objects to intersect is not a deterministic process either, preventing a general algorithm being developed.

Although in a simple 1D split operation the links are clear, using the connecting faces of each split surface, in an arrange it is not clear what the links are. For example an arrange might be done so the objects might intersect or they might not. Again this requires the use of an algorithms to determine the intersection of objects.

Not only is it unclear as to how to implement links but it is also unclear how to use them. The idea of links comes from linked lists, performing tasks on them is computationally efficient because there exists directions straight to the next object in a list. They rely then on having directions to work properly, whether that's forwards or backwards, or even up and down in 2-way linked lists. However simply having a list of linked objects is largely pointless as deciding which is the next object is impossible for an algorithm to do as links have no direction, in fact it is impossible for an algorithm to even know from which of an objects links it came from. Links must therefore have direction.

It might make sense to have directionally linked lists such as a two way list but with an arbitrary number of dimensions. This means that deterministic traversal paths can be created across objects, however implementing this is even harder than simple links. This is because it is even less clear than just with a list of links where to put traversal paths.

Given the difficulties associated with implementing generalised links, in hindsight it is now obvious why Krecklau and Kobbelt (2011) only use 1D linked geometries. If links could sensibly be implemented then they could prove to be powerful, however it appears at this time to be an impractically difficult task to achieve.

4.5 Scaling back the system

Links are incoherent and complex to implement for general cases, and at this point it was unclear as to how to implement axis properly. These difficulties blocked progress for the development of the prototype. Whilst initially believed to be key to the success of the system, it was decided to drop these features in order to make progress elsewhere.

The first thing to be removed were links. Geometry generation using links, such as placing walls is not necessary as equally efficient methods can be devised. Specifically allowing splits to create zero length segments enables wall generation by splitting an input volume into another volume, then surface, then volume. Parent and child links were also removed as not only are they implicit from the generation tree but they no longer made sense without horizontal links. Finally in order to make progress the axis were removed whilst a better solution was devised.

Afterwards only the geometry and hidden marker was left in the object definition. The six geometric functions could then be written without much difficulty.

```
class Obj:
    def __init__(self, geom, hidden = False):
        self.geom      = geom      # : Geometry
        self.hidden    = hidden    # : Bool
```

Figure 4-7: Redefined object specification, removing links, parents, children and axis.

4.6 Adding return statements

Using the redefined language, creating the first scripts using the GBDS was not difficult, the first complex object generated was a simple parametric building with 3D volumes representing walls and floors, generated from an initial mass model. To move this forwards from 2D walls to buildable walls, a script was written to create metal section walls.

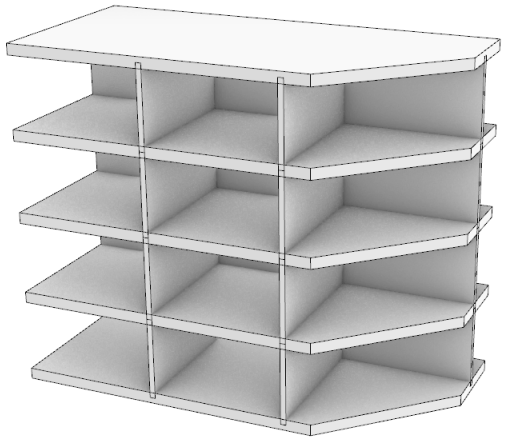
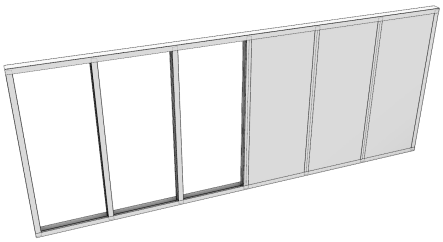


Figure 4-8: First script demonstrating a very basic parametric building, without any detail.

Metal section walls are a common method of wall construction (BSCA, SFL and SCI, 2019). For the purpose of the report they were chosen as the allow the testing of various different generation methods such as extruding objects and arranging parts. An extra function was created to enable the metal section wall script. Metal sections use a number of prebuilt components that are the same in every wall, as a result instead of generating these components an import function was created. This allows for geometry to be imported from a file. For example it would be easier to import a screw from an existing file then generate it using a script. The import rule for the metal section script was used to import the cross-sections for the c-shaped metal beams. These sections were then extruded

to the needed script using the extrude rule.

(a) Example of a real metal section wall. Image from SpeedClad (2019).



(b) Generated metal section wall, insulation has been partially removed to show post construction.

Figure 4-9: Generated wall and real wall.

At this point it was decided that sub-components, such as walls should be generated in separated scripts, which acted like functions. This simplified code by breaking it up into separate blocks for different parts. It also meant that separate name spaces exist, this allows for the definition of global parameters. Finally it enables code reuse, for example a metal section wall in one building is the same as another, consequently the code does not need to be rewritten each time but instead imported from file.

Furthermore a useful feature would be to have scripts return groups of objects, this would mean that objects generated by separate scripts would act like a single object in the main script. In order to do this the objects at the bottom of the generation would need to be retrieved. A simple way to do this is to add a return statement at the start of a rule, and modify the apply functions such that they return objects. This would mean that objects are then passed back up the generation tree from the very bottom.

```
def ruleName(obj): # : function(Obj)
    return applyFuction(rule1, predicate1,
                        rule2, predicate2,
                        ...
```

```
                        ruleN, predicateN)
                        (geometricFunction(obj))

def ruleName(obj): # : function(Obj)
    return [geometricFunction(obj)]
    # If the function returns a single object.

def ruleName(obj): # : function(Obj)
    return geometricFunction(obj)
    # if the function returns a list of objects.

def applyFirst(ruleA, predicateA,
               ruleB, predicateB):
    def f(objs):
        returnList = []
        if predicateA(objs[0]):
            returnList += ruleA(objs[0])
        for obj in objs[1:]:
            if predicateB(obj):
                returnList += ruleB(obj)
        return returnList
    return f
```

Figure 4-10: Top are the redefined rule and base rule with return of objects. Bottom is the redefined applyFirst function as an example of how the apply functions work. Note that it collates the outputs of previous rules by adding the returned lists together.

The redefined rules and apply functions work in unison to pass the base objects back to the top of the generation tree. The apply functions are constructed to only pass the output of the end objects and not any intermediate objects. The resulting output of each function is then a list of the base objects which it has generated. Consequently the top function in a sub script returns all of the objects generated. This solves the problem of accessing all of the objects to create a group

from them.

The return statement provides a natural manner to terminate the generation. Unlike a grammar which uses terminal symbols (Ginsburg, 1966), GBDS simply returns the last object without applying a rule to it.

By applying return statements to all of the rules, the rules no longer acted in a manner to that of a grammar, but instead very closely to a specific paradigm of programming language called functional. This lead to the realisation that what the GBDS needed to become was a functional programming language, as this provides a very precisely defined theoretical framework to follow.

Chapter 5

System Re-Specification

Using the relisation that the GBDS needed to become a functional programming language, the system could be re-specified within the context of a well established programming paradigm. Functional programming provides the framework for creating loops, maps, and flow control all within a recursive context that enables powerful tree generation.

5.1 Functional Programming Languages

Programming languages can be classified based on their operating paradigm O'Regan (2016). The kind most familiar to many programmers are imperative languages. These work by allowing programmers to define how to change a machines state or memory, for example loops in imperative languages are defined iteratively by changing the state of successive values in memory.

Imperative languages were borne out of the work done by Turing and their Turing Machine (O'Regan, 2016, Turing, 1937b). The Turing Machine was one of the oldest descriptions of computationally complete system and is a description of a practical computer. Fundamentally it uses state changes on a tape of symbols to perform calculations, as a result it is the foundation of modern imperative programming languages. In contrast at the same time Church (1936) developed their work on Lambda Calculus. Lambda Calculus describes a complete computation system based entirely off anonymous function composition.

Lambda Calculus is a way of programming logic that is evaluated, this is in

contrast to a Turing Machine which programs logic via state changes. O'Regan (2016) shows how this feature led to the development of functional programming languages. The defining feature of functional programming languages are that they evaluate the input rather than changing the state of a machine. This has several consequences:

- Because they evaluate programs without changing a machines state, they are free from side effects. The consequence of this is that outputs are always the same when running a program.
- Furthermore this feature means that programs can be reasoned about automatically, this allows for programs to be optimised.
- A key part of the reasoning is that a composed function is identical to it's constitute functions, this enables programs to be reduced into normal forms.
- Finally because they only evaluate functions, they cannot iterate and instead use recursion to evaluate lists or perform loops.

The features of functional programming languages are very relevant to GBDS. The fact that functions are guaranteed to return the same output with a given input is important as it would mean that regardless of where a rule is used, so long as the input is consistent it will always return the same result, consequently eliminating unexpected errors. However most importantly by forcing recursion, functional programming languages provide a consistent framework for creating scripts that include loops and trees.

Although it is important to note that anything that can be achieved using a functional framework can be achieved with an imperative imperative framework, both are Turing complete (O'Regan, 2016), the functional framework fits much better with the GBDS's recursive tree generation scheme. Practically scripts would be written in a functional programming language which has been extended to include interfaces to a CAD program to enable geometric operations. By doing so the power of a functional programming language is utilised whilst enabling users to write scripts that actually generate geometry.

Given that the intention is for users to script using a functional programming language as opposed to a grammar, it is worth finally making a comparison

between the two. From a computational theory standpoint it is likely that the grammars that can be made by Müller et al. (2006) in CGA and Schwarz and Müller (2015) in CGA++ are less powerful than a functional programming language; less powerful meaning that they could not compute and hence by extension generate everything a functional programming language could.

Returning to the computational theory discussed in chapter 2, all context free grammars can be recognised by a Pushdown Automaton, as demonstrated by Hopcroft (2008). This however is not as powerful as a Turing Machine; these can perform all computable operations including recognising CFGs, simply by simulating a Pushdown Automaton (Hopcroft, 2008). This means that a turing machine could generate languages beyond that of a CFG. As shown by Turing (1937a) themself, Turing Machines are equivalent in power to Lambda Calculus which as shown by O'Regan (2016) is equivalent to a properly constructed functional programming language with sufficient memory. From this it can be concluded that a functional programming language with sufficient memory can generate a greater set of languages than that of a CFG.

Whilst it is likely that a functional programming language is more powerful, in that it can generate more languages, than the grammars described by Müller et al. (2006) and Schwarz and Müller (2015), this has not been proven and would require rigorous analysis. The reason for the inability to be certain is that the grammars described are likely to be more powerful than a CFG because of the extra features, yet any practical functional language is limited by a computers finite memory, meaning that it is not equivalent in power to a Turing Machine. However the grammars described are limited largely to generation, they do not have complete parametric tools or access to readily available logic, unlike a functional language, leading to the conclusion that the latter is probably more powerful.

The functional programming language would appear as a scripting language attached to a CAD program, with extra functions for generating geometry. Implementing this practically would require that the geometric functions are isolated from the code using Monads, as these would need to be imperative HaskellWiki (2013). However in order for the prototype to be built quickly using existing code, ensuring the language was wholly functional was not made a priority and hence Monads were skipped. Python is a multi-paradigm language and can be used functionally, from this rules, objects and apply functions were then rede-

fined from a functional perspective, allowing for a better framework in which to implement certain ideas such as loops.

5.2 Re-specifying the GBDS from a Functional Perspective

Much of the formalism for making the GBDS functional already existed. The GBDS uses only recursion meaning that the result of the program does not modify earlier memory unexpectedly. Furthermore objects are passed on to new functions in a linear manner, meaning that the outputs of rules are the same given the same input. Finally apply functions are simply analogous to map functions. However the control flow on the apply functions is inappropriate leading to a necessary change.

They key difference is the control flow formalism. The apply functions use a predicate that works on the object to decide whether to apply a function or not. Consequently only a route between continuation and termination can be chosen, rather than a route between one function and another, this severely limits the capability. Furthermore forcing users to write the predicate each time is very cumbersome and time consuming, given that few loops need to be constructed. A better method is to remove the predicates and instead introduce optional guards at the start of rules. Using formal control flow operators such as guards enables multiple routes to be taken given the inputs.

Finally another move necessary is to remove the requirement that all rules must have an argument type of Obj. This means more sophisticated rules can be created that take in multiple arguments or arguments which are not of type Obj. Furthermore rules should also not need to return a list of objects, given that rules no longer need to be in the generation tree. For example it would be useful to have rules which can calculate the floor area of an object then return that to an if/else statement.

Although the syntax in Python makes currying inputs to functions impractical, it would make sense to specify that rules ought to curry arguments, as this is the appropriate formalism for a functional programming language (O'Regan, 2016). Currying makes the returned object implicit, consequently a return statement

would not be needed.

```
ruleName : inputType-0 -> ... -> inputType-n -> outputType:
ruleName inputArg-0 ... inputArg-n
    | cond True = applyFunction( rule-0 ... rule-n )
        ( GeometricFunctionA( inputArg ))
    | otherwise = applyFunction( rule-0 ... rule-n )
        ( GeometricFunction( inputArg ))
```

Figure 5-1: A standard rule formalism within a functional framework using guards. Note that other rule control flows would be possible. This is intentionally syntactically similar to the widely known functional programming language Haskell HaskellWiki (2013).

Although the psuedocode was written almost as if it were Haskell, a widely known functional programming language, Python does not allow code like this because of it's syntax. Consequently rules in the python code a constructed very differently from that of the pseudocode. However that does not mean that the code written in python does not provide the same end result.

Chapter 6

System Design & Implementation as a Functional Language

Following the re-specification of the GBDS to be a functional programming language with geometric functions, the python code was modified such that scripts could be written functionally. It is important to note that although much of the underlying code was left in it's imperative form, scripts appeared to behave as if they were functional. Although this might be problematic for a production system, it was deemed not to be a problem for the prototype, given the limited uses cases that were tested.

6.1 Rule Re-definition

Although Python is multi-paradigm and allows for functional programming, because it is not solely a functional language, it has very different syntax to traditional functional programming languages such as Haskell; in-particular it does not have type checking or type declaration. This means that certain things such as currying were too cumbersome to implement.

Rules were re-defined using python to follow the specification in section 5.2 as closely as possible. Because currying is difficult in Python, the inputs are supplied as arguments in the function definition. Next the guards were implemented

in Python as a series of if/elif/else statements. Finally the function calling follows the same format as the pseudocode, however Python requires explicit return statements.

```
def ruleName( arg0, ... argN ):
    if cond:
        return applyFuction(rule0, ... ruleN)
            (geometricFunction(obj))
    else:
        return applyFuction(rule0, ... ruleN)
            (geometricFunction(obj))
```

Figure 6-1: A standard rule formalism within a functional framework for Python. If/Else statements have been used instead of guards. The syntax of python makes currying cumbersome, consequently it has not been included.

Apply functions were also modified to remove the need for predicates. Although these still work using iteration rather than recursion, because this has no impact on how scripts superficially appear to work, it has not been changed given that the GBDS is only a prototype.

```
def applyFirst(ruleA, ruleB):
    def f(objs):
        returnList = []
        returnList += ruleA(objs[0])
        for obj in objs[1:]:
            returnList += ruleB(obj)
        return returnList
    return f
```

Figure 6-2: An example of an apply rule now with the predicate If/Else statements removed.

6.2 Geometric Function Modification

The current set-up for the geometric functions specify that each object that is generated is merely hidden when it is superseded by new objects, however this is unnecessary and computationally expensive. Because the system was moved towards functional, it is not necessary to keep all objects in existence as the program is guaranteed not to depend on earlier states of the program. Consequently all that is necessary is to work backwards up the generation tree to find parent objects. This means that input argument objects are deleted in the arrange and split geometric functions, and are modified in the other functions rather than being copied then hidden.

By reducing the number of objects that need to be copied or stored, the scripts themselves are more computationally efficient. Furthermore not storing past objects in memory ensures that the total size can be kept relatively small, with a maximum file size equal to the end result's file size.

Furthermore by ditching the method of making surfaces at the intersection of volumes a new more efficient method was devised, the zero length split. The split operation previously took volumes and split them into volumes using cutting planes placed at specific distances along a direction. If then a box was generated of length 10m, then splitting at relative distances of 0.3 and 0.7 would create a middle section of length 4m. If these two splits were moved together say 0.4 and 0.6 the middle section would now be 2m. These two cuts can be moved ever closer creating ever smaller middle sections, until at the limit it would be reasonable to conclude that the section would be a flat surface, representing the cross section of the volume. By doing this flat surfaces can be created from splits. For example a wall halfway through the box could be created by splitting at relative distances of 0.5 and 0.5, also producing a split volume either side.

This zero length split was extended to work at either end too, such that putting in splits at relative distances of 0.0 or 1.0 also creates surfaces. Using this method, series of walls can be generated in a succinct command. For example three enclosed volumes could be created by first splitting a box in the x-axis at relative distances of 0.0,0.33,0.33,0.66,0.66 and 1.0, then splitting each of the resulting volumes in the y-axis at relative distances of 0.0 and 1.0.

To produce an even more capable split function, it was modified so that it

worked on surfaces and curves too. For example splitting a surface can make more surfaces and curves, and splitting curves can make more curves and points. This provides a powerful yet natural manner in which to split up geometry.

6.3 Towards better scripts

The newly defined rules with better control flow, and the redefined split function allowed for scripts to be written in a much less cumbersome manner. using the new script format a cuboid building was generated that contained both constructable concrete plank floors, metal section walls with windows, and a steel structural frame.

Figure 6-3: Next phase of script demonstrating a detailed building, however components are restricted to the orthogonal axis. Additionally some key practical features are missing such as doors and stairs.

This was intended as a demonstration of the features of the program. Whilst a good demonstration of the GBDS, it was very difficult to do anything outside of the orthogonal axis. Consequently when trying to design a non-cubiod shaped building it became very difficult very quickly. This forced a revist to the problem of rotations.

6.4 Rotations are Still a Problem

The motivation for including an axis was to keep track of where an object has been transformed to such that it can be transformed back to it's original generation location. By enabling this an object can be operated on in it's local rather than global axis. For example splitting a rotated beam in it's local rather than global axis is very different, and often it is more helpful to do the former.

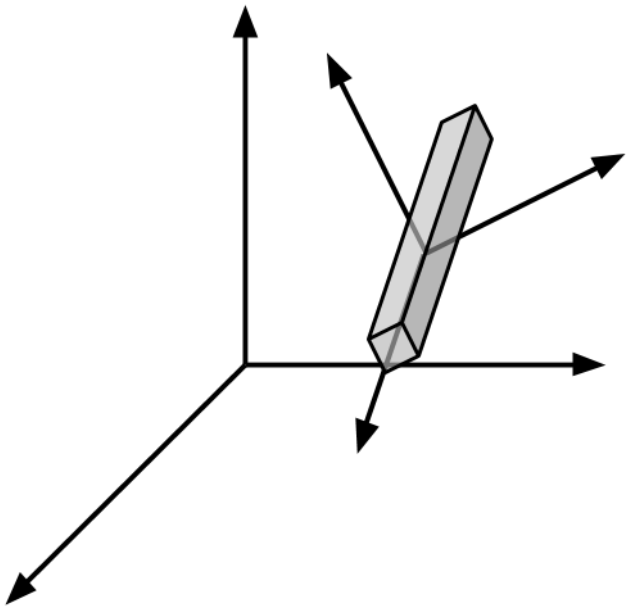


Figure 6-4: Global vs local coordinate systems. Notice that the rotated object has it's own orthogonally aligned axis, these are it's local coordinates.

The previous method of using a 3-vector to keep track of translation and another for rotation is too complicated because of the non-commutative properties of rotations in 3D. Instead a more comprehensive solution was required. Fortunately the need to transform objects between a local and global axis is very common in physics, engineering and computer science. The solution lies with basis and homogeneous coordinates as shown by Marsh (2006).

A basis is a representation of a vector in a particular coordinate system. For

example a vector might be noted as (1,1,1) in Cartesian coordinates, one particular basis, but in a basis that aligns with the vector it would instead be noted as (1.732,0,0). This scenario is illustrated in figure 6-5

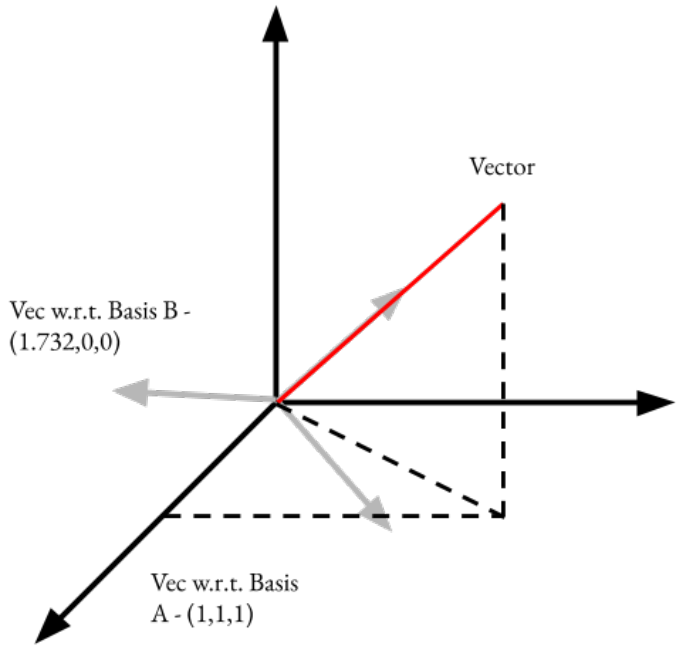


Figure 6-5: An illustration of basis.

It should be noted that local and global axis are just two basis of the same vector, and since in computer graphics objects are represented as vectors typically, then two basis of the same object. So the problem then is not how to change from local to global axis but from basis to basis. This is a well established problem. Basis transformation is solved by a simple matrix equation where X and Y are the same vector in two separate basis using transformation matrix A.

$$A^{-1}X = Y \quad , \quad X = AY \tag{6.1}$$

An equivalent expression can then be created for the local and global axis conversion. Let G be an object represented in global axis by some number of vectors and L be the object represented in local axis by some number of vectors.

$$A^{-1}G = L \quad , \quad G = AL \tag{6.2}$$

So it is clear that an object can be transformed from the local to global axis by multiplying the local vector representation by some matrix A and vice versa, by multiplying the global vector representation by the inverse of A. However determining what matrix A is, is impossible without extra information.

All transformations in 3D can be performed by multiplying some set of vectors by a 4 square matrix. The reason for using a 4 square matrix or homogeneous coordinates, over a 3 square matrix is that the former allows for affine transformations such as translations. If an object were to then be generated initially in global coordinates and transformed out of those coordinates it would be represented by the following equation, where G is the object represented in the initial coordinates, T is the transformation matrix and G' is the object represented in it's new coordinates.

$$G' = TG \tag{6.3}$$

But this equation is identical to the basis transformation equation, consequently it is clear that A is simply T the transformation matrix. What then about multiple successive transformations then using transformation matrices T and T'.

$$G' = TG \quad \Rightarrow \quad G'' = T'G' \quad \Rightarrow \quad G'' = (T'T)G \tag{6.4}$$

Again it is seen that A is simply equivalent to the series of successive transformation matrices multiplied by each other. Consequently to know A, all that needs to be done is to keep a matrix representing all transformations that an object has been through.

Practically realising this is trivial. When an object is generated for the first time, simultaneously an identity matrix of size 4 is also generated representing A for an object that hasn't been transformed. Then every time the object is transformed the matrix A is then multiplied by the transformation matrix. Consequently any time the object needs to be removed from it's global axis all that needs to be done is to transform it by then inverse of A.

6.5 Implementing Basis

Implementing basis required some modification of the code. The first step was to introduce 4 square matrix multiplication and 4 square matrix inversion. Unfortunately the version of python included with Rhinoceros 3D did not include any standard matrix manipulation libraries, such as numpy Oliphant (2006), meaning it was necessary to code this from scratch. To save development time raw python code was used courtesy of Ives (2018) for the matrix operations and code courtesy Vianello (2017) was used to implement a very efficient analytic 4 square inversion algorithm, with additional information from Miyazaki (1999).

Additional code was implemented to generate various transformation matrices, such as roll, pitch, yaw rotation matrices, scaling matrices and translation matrices. The code was then packaged into a dedicated 4 square matrix module in a separate file.

By introducing the capability to create all of the transformation matrices, the inbuilt translation and rotation commands in Rhinoceros 3D's API were superseded leaving just the matrix based transformation command in a single transformation geometric function. This created better structure for the rotate, translate and scale geometric functions as these now simply called the transformation geometric function with the appropriate matrices. Furthermore functions to transform between local and global coordinates were also created.

Sometimes it is useful to transform an object without moving it's basis, and sometimes it is useful to only transform an objects basis. To facilitate that, all of the transformation functions were modified to optionally transform either the basis or the object or both.

The split and arrange functions present a problem in how the sub objects should inherit their parent's basis. It was decided to simply inherit their parents basis but then translate each objects basis along the direction along which the objects are split or arranged, in the middle of the splitting planes or between arranging distances. Furthermore the split and arrange functions introduced another argument to optionally perform the function in either global or local coordinates automatically.

Finally the extrude function was modified such that it too could also be performed in the local or global coordinates. Furthermore extruded objects simply

inherited their parents basis without translation.

6.6 Completed Prototype Details

Following the work done in the two phases of specification and implementation a completed version of the prototype system was created.

Provided here are specific details of the functions available for script writing in the GBDS. Only the function definitions and return types have been included, the inner workings of most of the functions are unimportant for the purpose of the report. However the full code can be found in the annex **link**. the full details contain a number of additional geometric functions which were previously not mentioned, these are largely implemented as utility functions to enable faster script writing. In addition geometry creation functions were included, these are key for generating new geometry on the fly from provided parameters. These are used for example in generating insulation in a wall given the walls size.

```
class Obj:
    def __init__(self, geom, basis = t.idenMat()):
        self.geom = geom
        self.basis = basis
```

Figure 6-6: Object type definition.

This is the final version of the object type. Note that for the prototype other sub-types were not included. The object type includes geometry, which corresponds to a Rhinoceros 3D API object, and a basis, which is a 4 square matrix provided by the matrix module.

```
def deleteRule(obj):
    ...
    return []

def identityRule(obj):
    ...
    return [obj]
```

Figure 6-7: In-built rules.

These two functions are built in functions commonly used within scripts. The delete rule simply deletes objects and returns an empty list, this is useful for removing geometry after splits. The identity rule returns the object passed to it, it's used as a default rule argument for some of the apply functions when nothing should be done.

```
def applyAll(rule):
    def f(objs):
    ...
    return f

def applyFirst(ruleA, ruleB = identityRule):
    def f(objs):
    ...
    return f

def applyLast(ruleA, ruleB = identityRule):
    def f(objs):
    ...
    return f
```

```
def applyAlternate(ruleA, ruleB = identityRule):
    def f(objs):
    ...
    return f

def applySingle(rule):
    def f(obj):
    ...
    return f

def applyTwo(ruleA, ruleB):
    def f(objs):
    ...
    return f

def applyThree(ruleA, ruleB, ruleC):
    def f(objs):
    ...
    return f
```

Figure 6-8: Apply functions.

These are the apply functions and work much like map functions in many functional programming languages. The applyAll is identical to a map function, applying the given rule to all items in the list of objects passed to it. The apply-First, maps ruleA with the first object in the passed list and ruleB to the rest, in contrast applyLast maps ruleA with the last and ruleB with the rest. The applyAlternate maps objects in the passed list alternately with ruleA then ruleB. For example it would allow walls between volumes to have different functions. The applySingle is unique in that it only takes in a single object not in a list and runs the rule on it. The applyTwo and applyThree take in lists of objects of lengths 2 and 3 respectively then apply the given rules to the objects in the lists.

```
def showBasis(obj):

def select(obj):
    ...
    return obj
```

Figure 6-9: Debugging functions.

These two functions are for rudimentary debugging of scripts. The showBasis function generates a series of lines representing the unit vectors of the basis for the input object, this is useful in determining if objects actually have the correct basis. The select function selects the given object in the Rhinoceros 3D GUI, this allows users to verify that the correct object is being passed to a particular rule.

```
def transform(obj, matrix, moveBasis = True,
              moveObj = True):
    ...
    return obj

def move(obj, vec, moveBasis = True, moveObj = True):
    ...
    return obj

def rotate(obj, rotVec, pointVec = "cen",
           moveBasis = True, moveObj = True):
    ...
    return obj

def scale(obj, scaleVec, pointVec = "cen",
          moveBasis = False, moveObj = True):
    ...
```

```
    return obj

def golbalToLocal(obj):
    ...
    return basis, basisInverted
```

Figure 6-10: Transformation functions.

These functions perform transformations of objects, to improve efficiency they do not copy the input and delete it but merely modify it. The transform function is key as it is used by all of the others, it takes a transformation matrix and applies this to the object. The move function performs a translation given a vector of this. The rotate function performs a 3D rotation of a given roll pitch and yaw about the center point of an object, optionally a different rotation point can be provided. The scale function performs a linear scaling of an object about it's center point, optionally this point can be provided. Finally the globalToLocal function simply removes an object from it's global coordinates and returns the objects original basis, and the matrix inverse of this.

All of the primary transform functions optionally allow for either just the basis to be moved, just he object to be moved or both. Sometimes it is useful to do only one or the other, for example in correctly positioning the basis. The scaling function has the move basis option by default set to false, this is because scaling the basis means that when taken out of it's global axis the object returns to it's original size. Although in some instances it might be useful, it was decided from expirience this is more often not the case.

```
def extrude(obj, vector):
    ...
    return obj

def extrudeCurve(obj, vector):
    ...
```

```
    return obj

def extrudePoint(obj, vector):
    ...
    return obj

def subtractVolVol(obj, subtractor):
    ...
    return obj
```

Figure 6-11: Geometric manipulation functions.

The extrude functions simply take a lower dimensional object and extrude them along a linear vector path into higher dimensional objects. The extrude function extrudes surfaces to volumes, the extrudeCurve function extrudes curves to surfaces, and the extrudePoint function extrudes points to lines.

The cubtractVolVol function takes in an object and a subtractor, and subtracts the subtractor from the main object before deleting the subtractor. It is very useful for creating holes in volumes.

```
def arrange(obj, direction, spacing, number,
            local = True):
    ...
    return objs

def split(obj, direction, distances, objType = "vol",
          distType = "rel", local = True):
    ...
    return objs
```

Figure 6-12: Arrange and split functions.

The arrange function has five arguments. The first is the object to be copied. The direction is the orthogonal axis in which the copies are made. The spacing is the distance between the centers of objects. The number is the number of copies made.

The split function has six arguments. The first is the object to be split. Next is the orthogonal direction in which the object is split. The distances is a list of numbers representing where the cuts are to be made, note that these can make zero length splits. The next is a marker for the script writer to specify what kind of object is being passed to the function, whether a volume, surface or curve. The distType marker allows the users to specify whether the distances provided are relative to the objects length or an absolute value in global coordinates. Finally the split operation can optionally be done in either local or global coordinates.

```
def makePoly(points):
    ...
    return obj

def makeBox(x, y, z):
    ...
    return obj

def makeRec(x, y):
    ...
    return obj

def makeLine(x):
    ...
    return obj

def importObj(filepath):
    ...
    return obj
```

Figure 6-13: Geometry generation functions.

These functions allow for users to make geometry on the fly from scratch. These can be very useful for either starting generation trees or for replacing placeholder geometry. The makePoly function takes a set of points on a plane representing the vertices of a polygon, and makes a surface from them. The makeBox function generates a box of specified height, width and length about the orthogonal axis with it's center on the global origin, the box can then be transformed into position. Similarly the makeRec and makeLine functions generate lines and rectangles about the origin and orthognal to the global axis.

The importObj function is very useful as it allows for geometry to be imported from file, such as componets or Beam sections, that are pre-defined, have no parametric features and are tedious to generate using scripts. It has one argument specifying the file path of the Rhinoceros 3D geometry file to import and then imports the object in the same place as it exists in the file.

A number of other functions not seen here are shown in the annex, for reference these are not used within scripts, but instead enable the functioning of the geometric functions. For example several geometric functions share the same code parts, consequently these were abstracted out into one of these hidden functions.

Chapter 7

Testing

The goal of the Generative Building Design System was to allow users to generate fully parametric buildings using some form of a scripting language. Consequently as this is merely prototype software, the goal of the testing was to demonstrate the capabilities of the software and show that it can actually do what it was specified to do. In order to test the software a series of successively more complex buildings was generated, with each iteration of testing guiding the development of the prototype to enable the next iteration. Overall 4 iterations of testing were completed, the first demonstrated the ability to generate simple buildings consisting of walls and floors made of simple 3D volumes lacking any construction detail. This guided the prototype on towards enabling more complex constructions, allowing the second phase of testing enabling detailed building components to be made. From this the move towards functional was made, enabling the generation of the first building. However this building was necessarily cuboid shaped because of the lack of proper rotational control, consequently the system was edited to include basis. Finally the fourth and last stage of testing was performed in which a non cuboid parametric building was generated with much more realistic detailing than the previous building.

7.1 Generating Buildings from Volumes

The first key shift from previous grammars in terms of capability, beyond the logical restructuring of the scripts, was the ability to deal with solids over surfaces.

The split function being designed for use with volumes as opposed to surfaces allowed for very short but powerful scripts to be written, which enabled solid representation of building walls and floors, dividing up buildings internally.

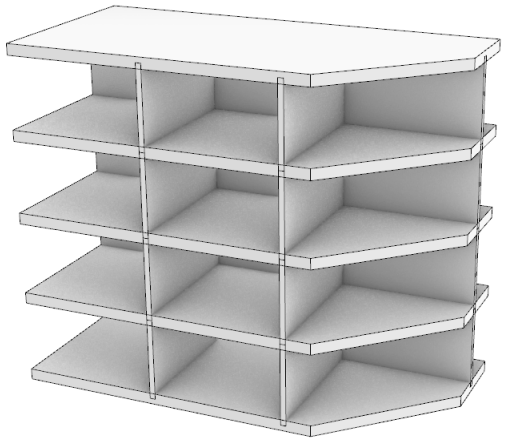


Figure 7-1: Demonstrating a very basic parametric building without any detail from the first testing phase.

Whilst demonstrating the capability to assign volumes representing walls and floors, it proved difficult to make functions that could generate the complex geometry for construction detail components. This prompted the introduction of the quality of life feature, the import function.

7.2 Generating Building Components

By introducing the import function, more complex geometries could be created rapidly. The import function meant that common components in construction do not need to be generated, but instead can be imported from pre-made libraries. To demonstrated this feature and move the previous building towards conductibility, a metal section wall script was created. Metal section walls are common in steel frame construction, and consist of a series of upright c-section posts held in place by c-section channels at the top and bottom of the wall. Insulation is then placed

between the posts, and plaster or cladding is screwed onto the posts.

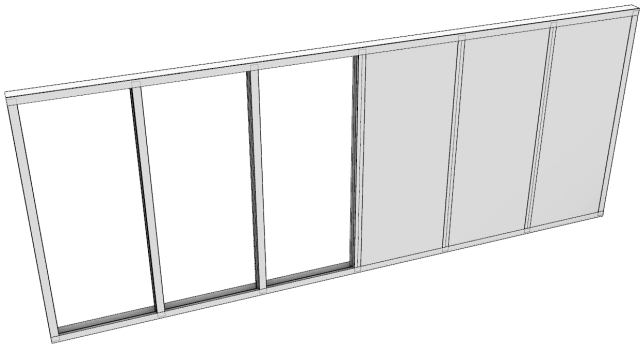


Figure 7-2: Demonstrating the parametric metal section wall function, insulation has been removed to see the construction detail.

The function created walls according to two parameters, the height and the length. Furthermore an optional parameter was introduced which specified the maximum spacing for the posts. The script demonstrates the import function, by importing the sections for the channels and posts from file, then extrudes them in another rule to the correct length. It also demonstrates the arrange feature which places the intermediate vertical posts, this is done parametrically to put them at an even spacing, no greater than the maximum, yet with the minimum number of posts. Furthermore it shows both the move and rotate functions, which shift the geometries of the posts into place such that they do not intersect. Finally the script demonstrates the use of the makePoly function, which generates the insulation and plasterboard paneling.

The resulting script demonstrated powerful sub-assembly generation. In order to integrate this into larger structures the capability for creating groups was added by moving the system towards functional.

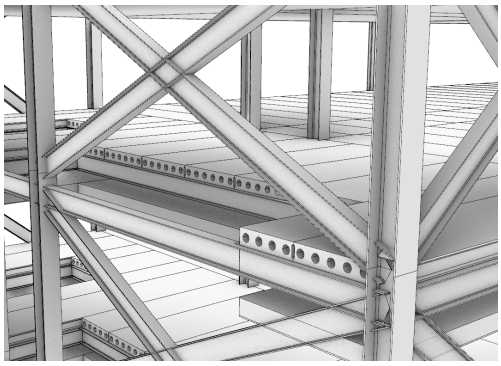
7.3 Generating Buildings with Detail

By adding the zero length split capability to the split function, the previous building could be reformulated to instead generate surfaces to represent floors and walls, which was less cumbersome to use. Furthermore this meant that the metal

section wall script could now be integrated as a separate function by simply calling up the length, height and center-point of the representative surfaces. Additionally because of the move to functional, the returned walls could be added as groups into the model.

To create floors another common construction method was chosen, concrete planks. These are pre-cast hollow concrete slabs that are secured to the top of beams. On top of these slabs is then laid a concrete scree, and then cosmetic flooring such as carpets or tiles. Below the beams typically hang plasterboards, leaving a void in between for services. These were generated much the same way that the metal section walls were, by importing the section of the planks and beams then extruding them to the right length. However unlike the walls, the concrete planks were arranged such that the final plank overlapped the edge of the floor and then was cut to size using the intersectVolVol function.

(a) Real world hollow core slab and steel beam construction. Image from IndiaPrecast (2017)



(b) Generated hollow core slab floors.

Figure 7-3: Generated floor and real floor.

Finally supporting the floors which support the walls, steel columns were generated. To generate the final version of the building, each of the generated surfaces were replaced respectively by the metal section walls, steel columns and concrete plank floors.

An extra detail was included to test the use of the splitting function on lower order objects. The surfaces representing the outside walls were split both vertically and then horizontally to create holes for windows. The metal section walls

were then generated on the wall subsections. Although having separated shorter sections framing the wall may seem wasteful as each part requires their own end posts, this is often done, as it simplifies construction in the same way it simplifies the script.

Figure 7-4: The third phase of testing. The building is detailed but lacks practical features.

Whilst the building represents a good step towards a high level of detail and constructability, fundamentally it is neither practical nor aesthetic. Firstly it lacks many practical details such as doors, stairs or cladding. Furthermore due to the limitations of the system at this point, it was impractical to generate anything not orthogonally aligned, limiting the type of buildings that could be generated. Moving away from orthogonal alignment required changes to the program, whereas the addition of practical features simply represented a requirement for a more developed script.

7.4 Practical Buildings.

The introduction of basis allowed for much more complex geometry to be developed with fewer lines of script, and enabled the practical generation of non-orthogonal geometry. Furthermore the removal of the predicates led to less cumbersome code, speeding up writing, helping the creation of more complex scripts.

As this was the final version of the prototype, a very detailed building model was designed, which represented not only something close to constructability, but also something practical that someone might want to have built. A parametric mid-rise residential building was designed, which followed a given series of lines, with a certain building width over a given number of stories.

Figure 7-5: Image of similar residential buildings near Canning Town in London. 3D mapping image from Google (2019)

A residential building was chosen as this represents a higher level of complexity than an office building, requiring a number of flats with minimum space requirements, with each flat having a number of rooms which must be aligned properly. The residential building represents a good combination of scripting difficulty coupled with utility to the construction market, as many buildings of this type are built each year.

The snaking construction type was chosen as this provides an opportunity to test using basis on non-orthogonal geometry, yet can be generated using simple rules. The input parameters for the building are a series of connected lines representing the snaking edge of the building, a building width representing how thick the snake is, the number of stories the building has and the floor to floor height. Optionally the ceiling void depth was also included.

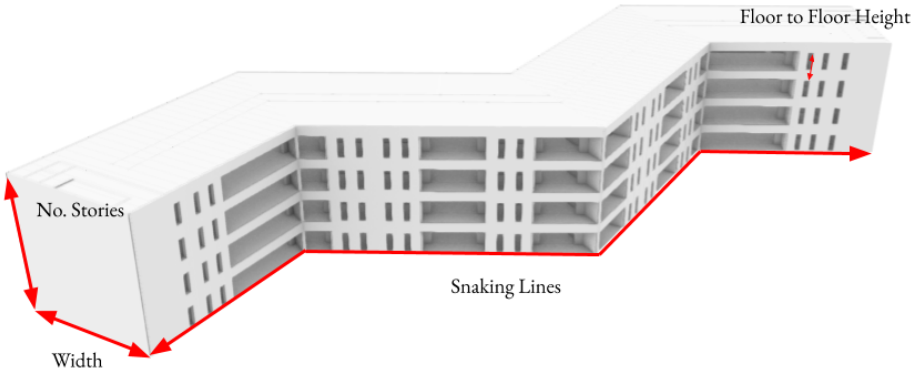


Figure 7-6: A diagram of the building parameters.

The first step in the generation was to create a script which laid out the positions of the rooms and walls within the building. In order to start the generation process, an ad-hoc offset function was created, which offset lines to create volumes and splitting lines. Ultimately this should be replaced by a script.

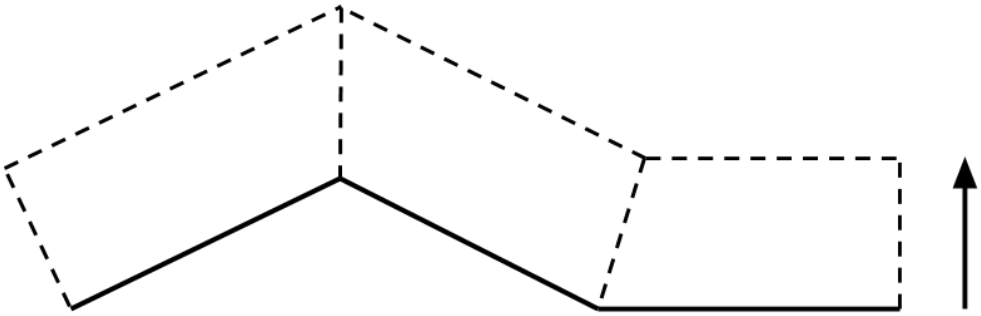


Figure 7-7: A diagram of how the offset function works.

Once the initial starting surfaces and lines had been generated, the output was fed to a starting rule. The end blocks were split between flats and stairwells with lifts. The mid blocks were just sent to become flats. Each of the blocks for the flats were then split to create corridors, then each of the resulting halves were then

oriented to become flats. One flat generation rule was created, however to do this some of the flats needed to have their basis rotated and mirrored, such that the various pieces of geometry would end up in the correct place. This demonstrates that not only are basis useful for creating rotated geometry, but also for handling mirror images of geometry using the same rule.

Once the flat orientation had taken place, a rule was created to determine the number of flats to generate in each segment such that there is a maximum number of flats whilst maintaining a minimum floor area. To do this a recursive loop was set up using guards, one of the core features of the program.

```
def splitFlats(obj, number=1):
    testObjs = split( obj, intSplit(number) )
    if testFlatSize(testObjs):
        return splitFlats(obj, number+1)
    elif number == 2:
        return applySingle(mirrorFlat)(obj)
    else:
        splitObjs = (split( copy(obj),intSplit(number-1) ))
        return applyAlternate(mirrorFlat,
                              FlatPartyWallCopy)(splitObjs[1:])

def testFlatSize(objs):
    if rs.SurfaceArea(objs[0].geom)[0] < globalFlatArea:
        return False
    elif len(objs) == 1:
        return True
    else:
        return testFlatSize(objs[1:])
```

Figure 7-8: Greatly simplified recursive loops for generating optimised flat splits.

As the surfaces representing the rooms were created, lines representing the

walls were also generated. These lines were then extruded to represent the walls as surfaces, and these surfaces were then split again to make windows and doors. The surface representations of the walls were then replaced by an enhanced version of the metal section wall script. The script was updated to make use of the basis and to also allow for various different types of panelling to be specified on either side of the wall.

The copy command was used to copy over the high level geometry so that the flooring could be generated. Again an updated version of the concrete plank floor script was used. The geometry was copied a second time such that the columns could be placed correctly within the structure, additionally cross bracing was implemented. Cross bracing is a feature that is necessary in most steel framed buildings to brace against wind pressure, this feature was lacking in the previous iteration but was included here.

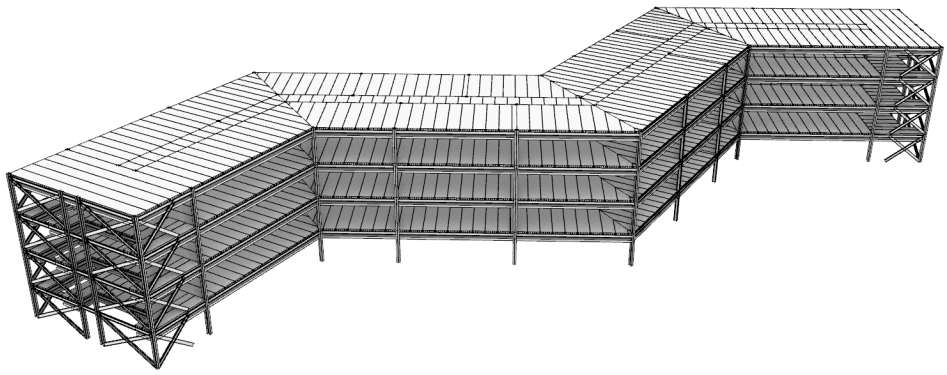


Figure 7-9: Building structure including cross bracing.

Next a script was created to generate fully parametric stairs for the stairwells. This script used advanced features of the system, specifically intersections and basis. Firstly a rectangle is extruded along an angled path, creating a parallelogram prizm, this is in contrast to previous extrusions which occurred orthogonally. Next cutting boxes are created along the path of the stair using the arrange function, these boxes are then intersected with the stair to cut steps into it. Finally the stair is positioned into place. This all happens parametrically, such that the stair

may cover any necessary height.

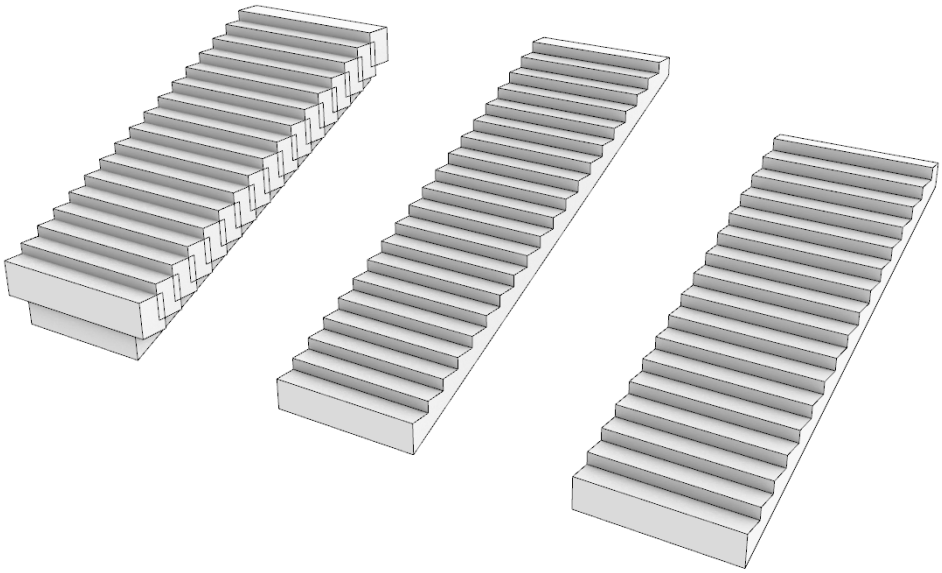


Figure 7-10: A diagram showing stair generation and two different height stairs.

Finally the start of the code was amended to arrange each starting object over a given number of stories, in addition the top floor was fed into separate flow so as not to include a hole for the stairs in the roof. , This cumulated in a parametric building encompassing a detailed structural and architectural design. Building services were not included, although risers for services were. Furthermore the building was not checked in accordance with building regulations, such as the structural design, see the discussion for details. This script demonstrated the successful generation of a detailed structure using the system, leading to the ending of the testing cycle.

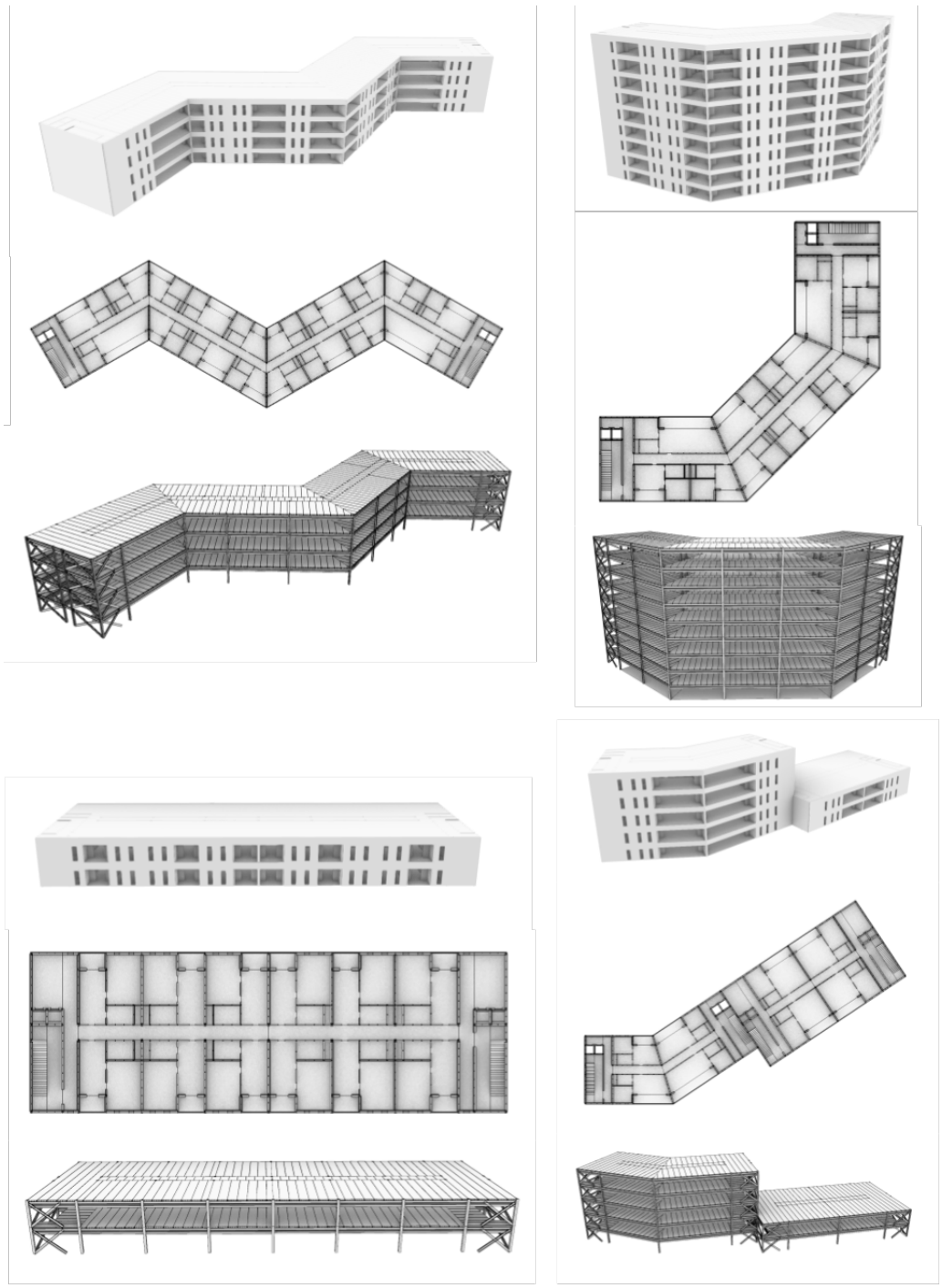


Figure 7-11: Demonstration of a number of buildings.

Chapter 8

Discussion

The Generative Building Design System, was intended to be both a piece of software allowing users to write scripts to generate parametric buildings, and a series of formalisms on how to linearise the building design process. However it was realised throughout the process of the research that there is no need to change the process of designing a building to fit the system, and that the system is unnecessary instead a better solution is to marry a widely know functional programming language with BIM software, along with guidance and tutorials on how to develop useful scripts.

8.1 Is Functional Useful?

Prior systems rely on allowing users to construct grammars to generate geometry, however this is an inappropriate tool for what they try to achieve. Grammars are a theoretical concept intended to allow for the a-posterior analysis of languages. In linguistics, languages are analysed to determine their grammar and in programming languages are analysed to determine their parse structure. Although grammars are a list of rules to generate sentences in languages, they were not created to generate bu to analyse

Pursuing grammars to be used to analyse 3D shapes as proposed by Stiny and Gips (1971) is a sensible use of them. As an example it would make sense to use grammars as a framework to determine a the rules an architect uses to design a building, or perhaps the rules plants have evolved to grow. However using a

grammar to generate geometry is inappropriate, because as a system they are not only overly rigorous, but also non-deterministic as used by Stiny and Gips (1971).

The non-determinism becomes apparent when two rules are written which can replace the same object. In the analysis form of grammars this is not an issue as the final object is there, so determining which rule has been used is trivial, but determining what rule to be used in the generation form is impossible without some kind of rule priority. This is what Müller et al. (2006) did to allow users to specify in what order rules should be applied, however as this research has shown there is a much less cumbersome way of doing this.

The usefulness of the grammar scripting system developed by Müller et al. (2006) is not because they are a grammar, but rather because the generation of geometry as a tree is logical and natural. Design is often a process from little detail to lots of detail, starting off with only large scale features, then adding increasing amounts of detail until a final product. Tree-like generation suits this from of design. At the top of the tree are the overarching initial objects such as a building mass, then at each descending layer of the tree is added more detail, first the floors are made, then rooms, then walls, then windows in walls, then connectors holding walls to windows, and so forth until the bottom.

Instead of assigning a priority to rules, a less cumbersome method is to simply have each rule directly call the next. Using a functional programming language then made sense, this naturally provides a framework for tree like generation. Functional programming languages are recursive by nature, with successive functions calling each other. Because of this recursive calling of functions it is trivial to generate and manipulate trees. All that needs to be done is for functions to call more than one function at a time. Facilitating this was trivial by simply implementing user friendly variants of the common map function in the form of the apply functions.

Using a functional programming language for writing scripts instead of a grammar is not only less cumbersome and more logical, but also more powerful. By simply directing each rule to the next, the need for priorities are eliminated making it much more apparent what the generation tree is. Furthermore by passing rules, there is no need for any kind of pattern matching eliminating the need for object names as used by Müller et al. (2006). Furthermore as demonstrated in section 5.1, a functional programming language with sufficient memory is more

powerful than a grammar from a theoretical standpoint.

Furthermore several generation features now have a framework to follow. Loops can now be rigorously constructed using recursion and guards; grouping objects is simply done by returning the base objects up the generation tree; parameters can be manipulated properly. In Schwarz and Müller's program, these are implemented in an ad-hoc fashion using pauses and priorities. Writing scripts using a functional programming language with extra functions for generating geometry proved to be powerful, enabling the generation of much more sophisticated objects than have previously been generated by others.

8.2 Fully Parametric Buildings

Using a functional programming language allowed for a parametric building script to be developed in a matter of weeks, which generates detailed buildings that can be readily regenerated with different parameters. The resulting buildings represent a significant development on what had been achieved before. Whilst many programs had demonstrated certain important features that would be needed for a system, none were general enough to show the whole lot. The developed script demonstrates splitting floors, arranging items, manipulating geometry on the fly with the stair generation and optimising floor area using a loop. From a theoretical standpoint a functional programming language can do anything previously done, however practically the efforts of earlier work could be replicated without much difficulty.

One of the biggest differences between what has been demonstrated lies in the level of detail generated. Previous systems especially Müller et al.'s and Schwarz and Müller's, are only useful for generating building shells, whilst useful for computer graphics, it does not represent a high level of utility in construction. However the buildings shown incorporate most of the details that would be required for a buildings construction, including the architectural and structural details, representing a much higher level of utility.

The other big difference is the generality of scripts. Müller et al. (2006), Schwarz and Müller (2015) and Thaller et al. (2013) are all restricted to a specific set of objects, in contrast the system developed was designed to be capable of generating any kind of object. For example the stairs in the building would be

very difficult to generate as a single object in other systems, yet in this one proved trivial to create. The generality of the scripts though increases the challenge of generating objects, especially when there are multiple paths to arrive at the same output.

Developing a detailed model requires complex scripts, with many sub functions for various building components. Because of this it difficult and time consuming to write scripts to a construction level of detail. For example, although the architectural and structural features were included, services were not. This is not because they cannot be done, but instead because of a lack of expertise on behalf of the author.

8.3 Writing Scripts in Practice

Although writing fully parametric scripts is potentially difficult and time consuming, the fact that they are fully parametric could ultimately save time. Creating these scripts will allow designers to explore a very large number of options in a small amount of time, allowing for various cost and benefit analysis to be set up, this process could even be automated. Furthermore modifications to a detailed design are easy, it's just a matter of changing the desired parameters. Additionally once a single building script has been written, it is likely that code could be reused amongst many other designs, this was done in this research project throughout the iterations of the buildings with regards to the metal section walls and hollow core floors. Demonstrating this, the multiple buildings shown are all from the same script, just with different initial parameters.

Code sharing would help negate the problems with the effort required to create scripts. For example standard libraries or modules of functions could be created, such that creating certain kinds of walls is as simple as calling up a single command. To an extent this has already been done, all of the sub component scripts were placed in a single separate file, however it is likely that to make useful software larger libraries will need to be made.

Fortunately there is potential incentive for construction component manufacturers to create scripts for their products. If this were to be used widely, manufactures would need to provide scripts for their products or face loosing out to competitors who do provide scripts. However for this to happen the product

would need to be widely used which requires that scripts are easy to make, requiring libraries. Consequently early on libraries would need to be made without help from the manufacturers. Alternatively it may be seen that specific designers may also create their own libraries to become competitive.

The building did not contain the service design, however this is simply a lack of expertise from the author, this is likely to be similar in industry. Typically there are separate designers for each part of the process each with their own expertise, yet this does not impact on the software with each designer providing their own scripts to contribute to each part of the building. Although this seems counter to the initial desire to linearise the design process, the design process does not actually need be linearised.

8.4 No Need to Linearise the Design Process

When beginning the research project it was assumed that the design process needed to be linearised in order to create sensible scripts, however this is not the case. When generating buildings it was found that the easiest way to incorporate different aspects such as architectural and structural design was to separate them in different generation trees early on. For example the separation of the structural and architectural design in the building happens just after the layout of the flats. This leads to almost concurrent design, although from the computers perspective it processes one branch before the other, from the users perspective each step is at the same depth on the tree.

8.5 Poor Numerical Control in CAD

Although not discussed in the methods, one of the most difficult problems overcome was managing Rhinoceros's numerical errors. When creating geometry it is common to construct geometry from other geometry, such as creating line tangents or calculating intersections. Often when performing these tasks it is impossible to represent dimensions exactly in a number system. For example when calculating the circumference of a circle the constant pi must be used, which is an irrational number and hence can't be represented exactly in any base system. The problem is compounded further when dealing with geometry in computer systems. Com-

puters have finite memory and hence cannot store numbers with infinite precision, for example representing the fraction 1/3 as a decimal floating point number exactly is impossible, as the last digit must be rounded. As a result all computer based geometry systems have inherent numerical error.

Most CAD and BIM software use 64bit floating point numbers. 64 bit floating point numbers are useful in that they can represent both very large and very small numbers to a high level of precision, however they are problematic in that they don't exactly convert to decimal numbers, leading to further errors as designers typically put measurements at whole decimal numbers. Consequently most CAD and BIM programs have two potential sources of error, firstly precision limits relating to memory size, and secondly floating point representation errors.

Although it is impossible to know if the leading CAD and BIM software use numerical control features to prevent errors, the author has discovered that both Rhinoceros 3D and AutoCAD (Autodesk, 2019a), present errors likely to be because of bad numerical control. It is unlikely that numerical errors are likely to be seen in every day use as only certain use behaviours make them appear. Numerical errors become noticeably large when generating geometry from other geometry, as this causes errors to compound and magnify each other. Unfortunately this process of generating geometry from geometry is exactly what the system does.

Because of this feature, errors in Rhinoceros 3D became rapidly evident when writing several of the geometric functions. The biggest problem was writing the split function. When creating zero length splits the function intersects a plane with the geometry to be split, when done in the middle of the object this is robust, however at the objects end, the planes would consistently fail to intersect because despite the plane being created at the end it actually misses it due to numerical errors. To make it robust each plane was placed ever so slightly inwards, which is not an ideal solution.

It is unknown if numerical errors occur elsewhere, however it is entirely possible they may crop up unexpectedly causing the system to fail to work properly. What is needed is better numerical control from the geometry program linked to the system.

Chapter 9

Future Work

Given the limitations of the system: The errors caused by Rhinoceros 3D; Python's syntax preventing type declaration; and the fact Python is not wholly functional; it would be good to suggest a better system. The proposal of what a GBDS system would look like is also the key aim of the report.

9.1 A Production System

Three key requirements have been established for a production system: BIM software to handle the geometry, display geometry and interface with other industry software; a functional scripting language for the software; and a library of existing scripts to generate common features.

Previously it had been assumed that the scripts themselves can wholly encode the necessary BIM information, allowing a CAD program such as Rhinoceros 3D to become a BIM program. However this is no longer believed to be the best option. Users prefer familiar software (Galitz, 2007), and forcing them to essentially code BIM into every script would be pointless, instead using existing BIM software would make more sense.

It might again seem that new BIM software should be used, given that generating geometry recursively causes repeated operations which can quickly magnify numerical errors. However again using the argument by Galitz (2007), that users prefer familiar software, it would be better to instead use the most popular BIM program Revit (Autodesk, 2019b). Despite the clear failures of existing CAD and

BIM software usability is key to making a practical system. This would also likely mean a higher up-take of software, as the system would be released simply as a module withing Revit, meaning that users can try out the program in a familiar environment.

Although python was used to create the prototype, it is not wholly functional nor does it allow type declaration. It could be possible to make a custom scripting language to interface with the BIM software. Although this would allow the creation of a simple scripting language, it would be better to instead use an existing and widely used functional programming language, again for the same reasoning that Revit should be used as the BIM interface.

Haskell suits this purpose well. Not only is it one of the most widely used functional programming languages, it is strongly typed and is wholly functional (HaskellWiki, 2013, O'Regan, 2016). Furthermore can be interpreted rather than compiled, this lends itself to the rapid development of scripts, as users would not need to compile them each time. This also facilitates debugging as interpreted languages can be run line by line.

To integrate with BIM software a Haskell interpreter should be created, which also includes additional functions for geometric manipulation. Unfortunately generating geometry is not a purely functional task and requires some level of imperative design, this is not likely to be an issue as Haskell already implements IO features. It would be likely that the geometric manipulation would be implemented similarly to the not purely functional IO features of Haskell.

Functional programming in CAD has been explored by others. Nandi et al. (2018) demonstrate a CAD program that can only be used by creating scripts in a custom language, however it is argued that the importance of familiarity and well supported systems is much more important. Furthermore because their system can only be used via scripting it does not allow flexibility in partial scripts, which could ease users into learning the program.

Finally in order to make the system usefull a library of functions composed of scripts for common components should be made. For example this might contain scripts to generate walls or beams or pipes. This could make the system much more accessible to new users and allow for much faster script creation.

9.2 Testing the Software on Users

The purpose of the research was to simply determine whether it was possible to create a GBDS, and if so what this might look like, consequently no user studies were performed. However it would be usefull to determine whether the system would actually be useful and whether designers would use it. This would gauge a need for the software, evaluate it's current features, and compare it critically against other parametric design systems such as Grashopper or just manual design.

Many CAD and BIM programs allow for scripts in a variety of languages to be written yet all of these use imperative languages. The author could only find a single other paper describing a functional scripting language for CAD, lambda CAD, by Nandi et al. (2018), it is important to note that this is not a commercial system. Because of this it would be important to compare how users might view scripting in an entirely different paradigm.

Although conclusions can be made at this time on what the prototype system is capable of, because no user studies have been performed it is vital to do this. User studies would allow for definite conclusions about the practicality of the system when used by everyday engineers and architects.

Chapter 10

Conclusions

This research has been a journey to try and create a GBDS system. Initially it was assumed that the system needed to be grammar-like and contain links between objects. This then morphed into a functional programming language as a prototype, guiding what a release system would look like. This release system would likely consist of a Haskell interpreter for Revit, with a library of existing scripts for common construction objects.

An initial system specification was devised based upon earlier work, and included features that might be able to move beyond grammars for computer graphics. However this specification included unworkable features such as links, and poorly implemented features such as axis tracking. Dropping many of the early features gave momentum to the research allowing the creation of the first scripts, this in turn led to a better understanding of the requirements.

From the first scripts it was realised that adding return statements to rules could allow for groups to be made. Yet these return statements also gave way to the realisation that scripting in a functional programming language could provide the logical framework necessary for the system. Specifically by scripting in a functional language, a natural framework exits for generating geometry in a tree like manner, and using recursion for loops.

Writing scripts following a functional framework meant that much of the cumbersome syntax of the earlier versions could be dropped. However a key issue remained; Dealing with non orthogonally aligned objects was difficult as there was no way to keep track of rotation in 3D. This cumulated in the introduction of

a transformation formalism known as basis. Implementing this via homogenous coordinates, where every transformation is a four square matrix, overcame earlier problems with transfomring objects using vectors.

By resolving the rotation tracking problem using basis, the software had progressed into a finished prototype. The prototype was then demonstrated by creating a script for a detailed parametric building. This script demonstrated detail and generalism that had not previously been achieved by earlier research. This satisfies the requirements set out in the introduction which was to move towards detailed construction level designs. Furthermore by using a functional programming language over a grammar, the system is much more capable, not only practically but from a theoretical standpoint.

However the system has not been evaluated in a user study to determine whether it is actually any better than other scripting languages in CAD and BIM programs, or if it is more useful than designing without a script. This is work that needs to be done. It would be good if beforehand a production system could be built.

From the work done a clear image has emerged of what a production system should look like. Using well known languages and software is likely to be key in getting designers to adopt it. Consequently instead of recommending a custom BIM package with a custom scripting language, a combination of Revit with a Haskell interpreter is recommended. The Haskell interpreter would likely manifest as a Revit plugin, with additional custom functions to perform geometric manipulation. These geometric functions would be implemented much like the stock IO functions already in Haskell; this is to keep the core code purely functional. Finally to increase the ease of script writing a collection of pre-made functions, such as wall or beam generation, would be created and collated into a library.

Bibliography

Autodesk, 2019a. Autocad for mac and windows: Cad software. Available from: `https://www.autodesk.co.uk/products/autocad/overview`.

Autodesk, 2019b. Revit: Bim software. Available from: `https://www.autodesk.com/products/revit/overview`.

Azhar, S., 2011. Building information modeling (bim): Trends, benefits, risks, and challenges for the aec industry. *Leadership and management in engineering*, 11(3), pp.241–252.

Backus, J.W., 1959. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proceedings of the International Comference on Information Processing, 1959*.

Becker, S., Peter, M., Fritsch, D., Philipp, D., Baier, P. and Dibak, C. Combined grammar for the modeling of building interiors.

Blade Runner, 1982. Warner Bros.

Blade Runner 2049, 2017. Warner Bros.

BSCA, SFL and SCI, 2019. Infill walling. Available from: `https://www.steelconstruction.info/Infill_walling`.

Chomsky, N., 1956. Three models for the description of language. *IRE Transactions on information theory*, 2(3), pp.113–124.

Chomsky, N., 1957. *Syntactic structures*. Mouton.

Church, A., 1936. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2), pp.345–363.

Designing Buildings Wiki, 2019. Building design process. Available from: `https://www.designingbuildings.co.uk/wiki/Building_design_process`.

Diebel, J., 2006. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16), pp.1–35.

Galitz, W.O., 2007. *The essential guide to user interface design: an introduction to gui design principles and techniques*. John Wiley & Sons.

Ginsburg, S., 1966. *The mathematical theory of context free languages.[mit fig.]*. McGraw-Hill Book Company.

Google, 2019. Google maps. Available from: `https://www.google.com/maps`.

Gosling, J., Joy, B., Steele, G. and Bracha, G., 2000. *The java language specification*. Addison-Wesley Professional.

Hahn, E., Bose, P. and Whitehead, A., 2006. Persistent realtime building interior generation. pp.179–186. Available from: `http://doi.org/10.1145/1183316.1183342`.

Halpin, D.W., 1998. *Construction management*. 2nd ed. New York: Wiley.

HaskellWiki, 2013. Haskell — haskellwiki,. [Online; accessed 10-September-2019]. Available from: `https://wiki.haskell.org/index.php?title=Haskell&oldid=56799`.

Hopcroft, J.E., 2008. *Introduction to automata theory, languages, and computation*. Pearson Education India.

IndiaPrecast, 2017. Menu. Available from: `http://www.indiaprecast.com/hollow-core-slab/warehousing-by-hollowcore-slab.html`.

Ives, T., 2018. Simple matrix inversion in pure python without numpy or scipy. Available from: `https://integratedmlai.com/matrixinverse/`.

Janil, R., 2019. Marseille urban planning project. Available from: `https://www.esri.com/en-us/arcgis/products/esri-cityengine/success-stories/marseille-urban-planning`.

Krecklau, L. and Kobbelt, L., 2011. Procedural modeling of interconnected structures. *Computer graphics forum*. Wiley Online Library, vol. 30, pp.335–344.

Krecklau, L., Pavic, D. and Kobbelt, L., 2010. Generalized use of non-terminal symbols for procedural modeling. *Computer graphics forum*. Wiley Online Library, vol. 29, pp.2291–2303.

Lindenmayer, A., 1968. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3), pp.280–299.

Liu, H., Yang, Y.L., AlHalawani, S. and Mitra, N.J., 2013. Constraint-aware interior layout exploration for pre-cast concrete-based buildings. *The Visual Computer*, 29(6-8), pp.663–673.

Marsh, D., 2006. *Applied geometry for computer graphics and cad*. Springer Science & Business Media.

Merrell, P., Schkufza, E. and Koltun, V., 2010. Computer-generated residential building layouts. *Acm transactions on graphics (tog)*. ACM, vol. 29, p.181.

Meyerovich, L.A. and Rabkin, A.S., 2013. Empirical analysis of programming language adoption. *Acm sigplan notices*. ACM, vol. 48, pp.1–18.

Miyazaki, D., 1999. Inverse matrix of 2-by-2 matrix, 3-by-3 matrix, 4-by-4 matrix. Available from: `http://cg.info.hiroshima-cu.ac.jp/~miyazaki/knowledge/teche0023.html`.

Müller, P., Wonka, P., Haegler, S., Ulmer, A. and Van Gool, L., 2006. Procedural modeling of buildings. *Acm Transactions On Graphics (Tog)*, 25(3), pp.614–623.

Nandi, C., Wilcox, J.R., Panchekha, P., Blau, T., Grossman, D. and Tatlock, Z., 2018. Functional programming for compiling and decompiling computer-aided design. *Proceedings of the ACM on Programming Languages*, 2(ICFP), p.99.

Nasa, 2015. Aircraft rotations. Available from: https://www.grc.nasa.gov/www/k-12/airplane/rotations.html.

Oliphant, T.E., 2006. *A guide to numpy*, vol. 1. Trelgol Publishing USA.

O'Regan, G., 2016. *Introduction to the history of computing : A computing history primer*, Undergraduate Topics in Computer Science. Cham: Springer International Publishing.

Palladio, A., 1570. *I quattro libri dell'architettura*. Dominico de' Francheschi.

Parish, Y.I. and Müller, P., 2001. Procedural modeling of cities. *Proceedings of the 28th annual conference on computer graphics and interactive techniques*. ACM, pp.301–308.

Parlante, N. Linked list basics.

Prusinkiewicz, P. and Lindenmayer, A., 2012. *The algorithmic beauty of plants*. Springer Science & Business Media.

Richardson, K., 2018. Esri cityengine used to create oscar-nominated effects in blade runner 2049. Available from: https://www.esri.com/esri-news/releases/18-1qtr/{\\}esri-cityengine-used-to-create-oscarnominated-effects-in-blade-runner-2049.

Savage, A., 2007. Blade runner at 25: Why the sci-fi f/x are still unsurpassed. *Popular Mechanics*. Available from: https://www.popularmechanics.com/science/a1775/4218376/.

Schwarz, M. and Müller, P., 2015. Advanced procedural modeling of architecture. *ACM Transactions on Graphics (TOG)*, 34(4), p.107.

SpeedClad, 2019. Steel framing systems (sfs). Available from: http://www.speedclad.co.uk/facade-products/steel-framing-systems-sfs/.

Stiny, G., 1982. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1), pp.113–114. https://doi.org/10.1068/b090113, Available from: http://doi.org/10.1068/b090113.

Stiny, G. and Gips, J., 1971. Shape grammars and the generative specification of painting and sculpture. *Ifip congress (2)*. vol. 2.

Stiny, G. and Mitchell, W., 1978. The palladian grammar. *Environment and Planning B*, 5, pp.5–18.

Thaller, W., Krispel, U., Zmugg, R., Havemann, S. and Fellner, D.W., 2013. Shape grammars on convex polyhedra. *Computers & graphics*, 37(6), pp.707–717.

Turing, A.M., 1937a. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4), pp.153–163.

Turing, A.M., 1937b. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1), pp.230–265.

Vianello, A., 2017. Matrix inversion without numpy (python). Available from: `https://codedump.io/share/GCNSbWGphkFZ/1/matrix-inversion-without-numpy`.

Wonka, P., Wimmer, M., Sillion, F. and Ribarsky, W., 2003. *Instant architecture*, vol. 22. ACM.

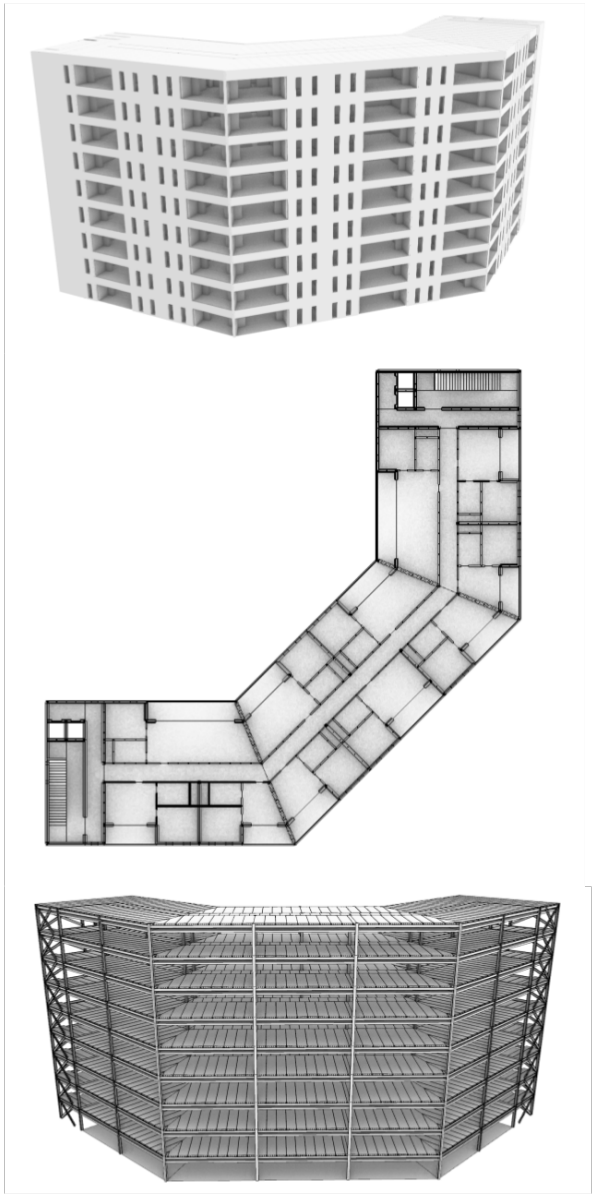
Woudhuysen, J., 2004. *Why is construction so backward?* Chichester: Wiley-Academy.

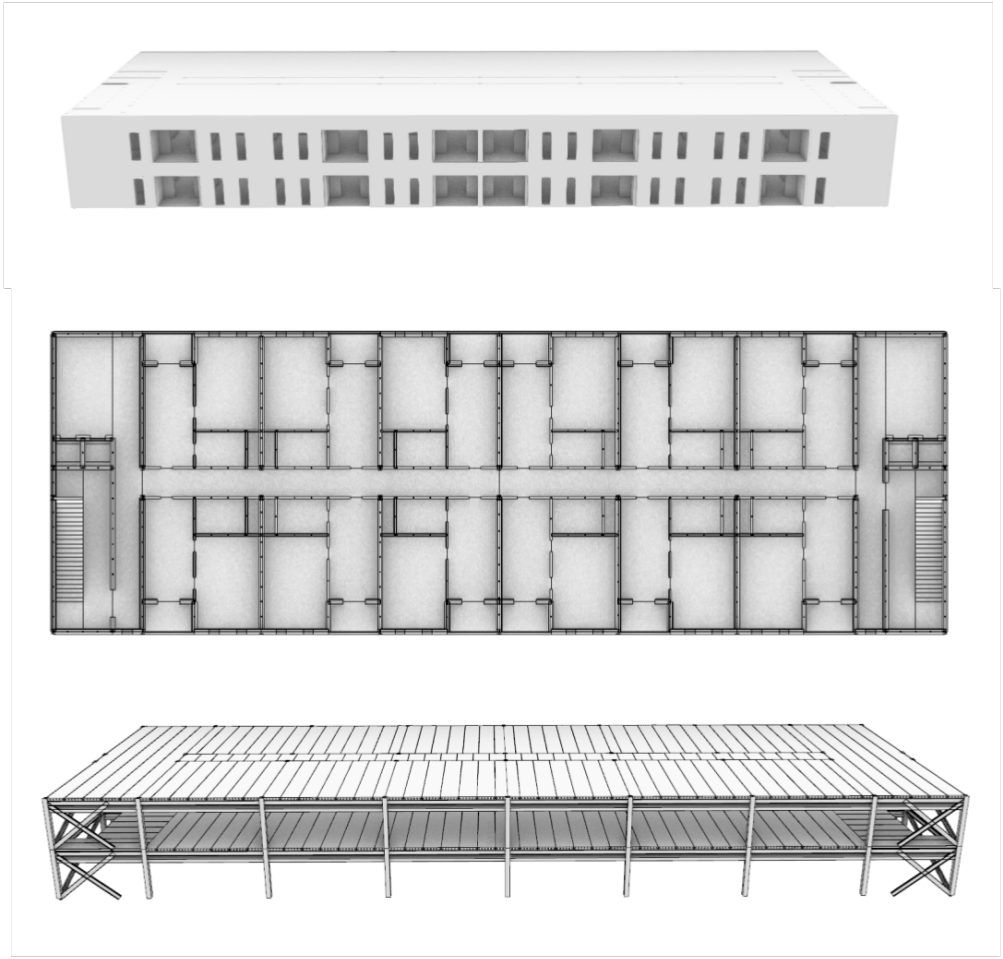
Wu, W., Fan, L., Liu, L. and Wonka, P., 2018. Miqp-based layout design for building interiors. *Computer graphics forum*. Wiley Online Library, vol. 37, pp.511–521.

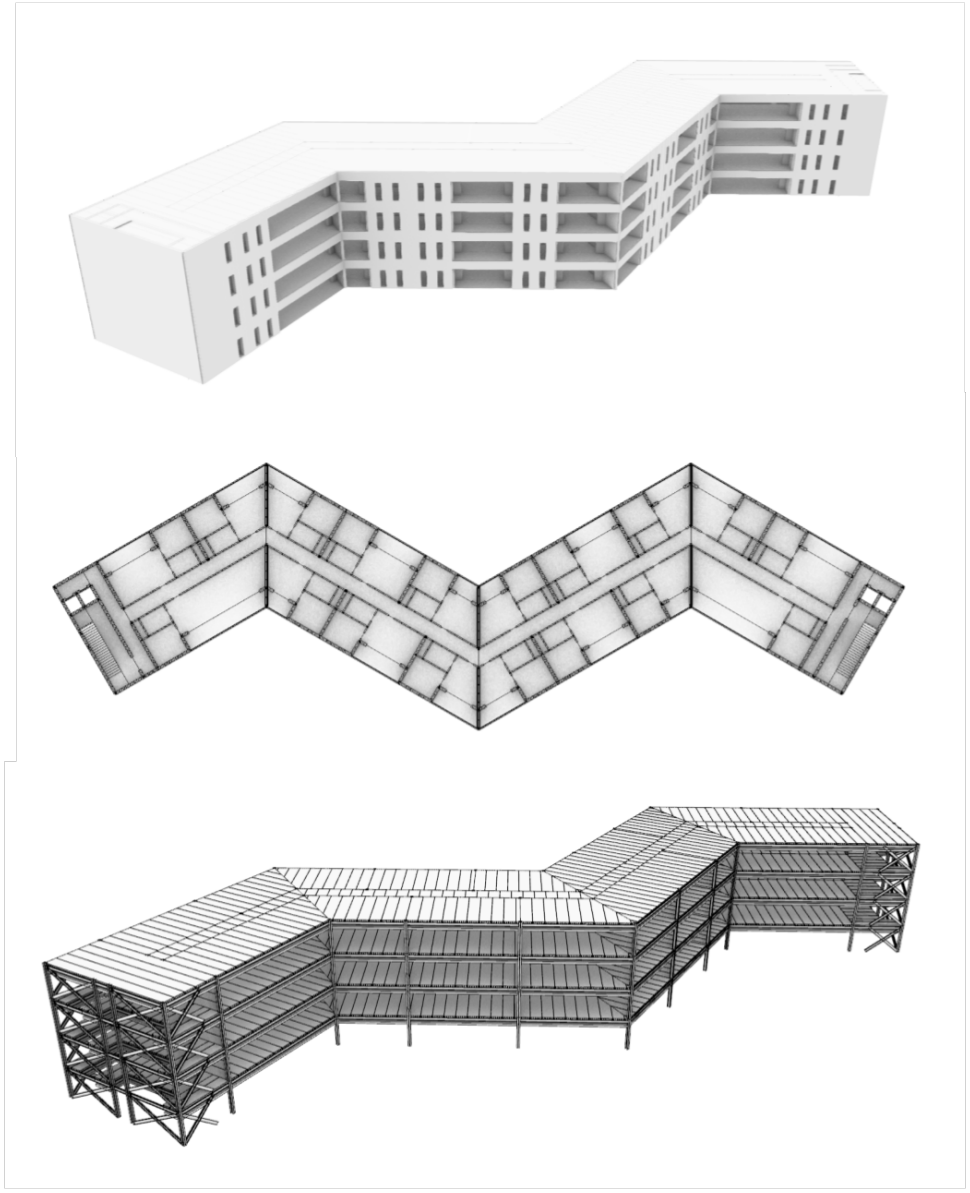
Appendix

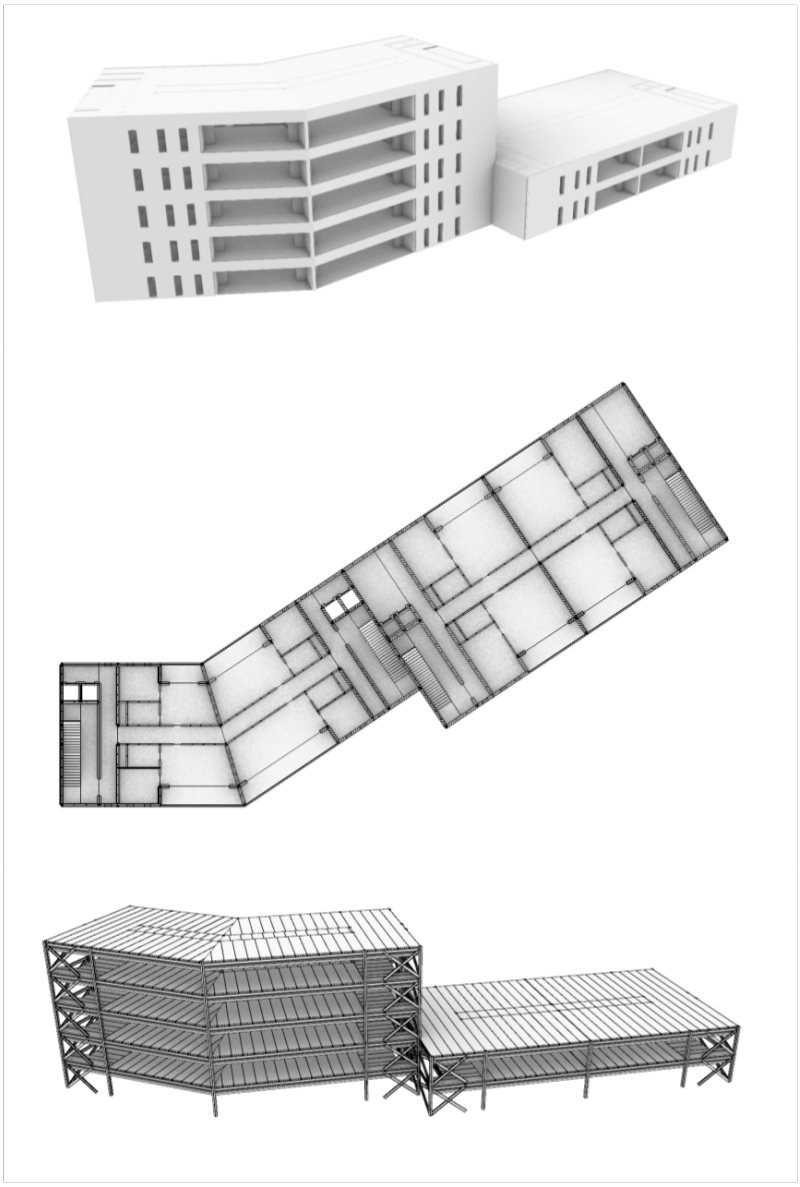
Large Size Images

Included here are large size versions of the demonstration images from the testing chapter. This has been done for convenience for paper versions of the dissertation.









Core Code

This is the core code for the system, it contains all of the geometric functions, apply functions, utility functions and the object class definition. The core code is what enables the script writing. This essentially is the GBDS system.

```
"""
Final version of BIMcurry!
"""

#########################################################################################################################
############################################################ Imports ####################################################
#########################################################################################################################


import rhinoscriptsyntax as rs
import os
import math as m
import transMat as t


#########################################################################################################################
########################################################## Object Classes ###############################################
#########################################################################################################################


class Obj:
    """
    An Obj or Object represents a discrete entity within the BIMcurry plugin.
    """
    def __init__(self, geom, basis = t.idenMat()):
        """
        This is the class constructor for the object.  It contains thwo components the geometry and the basis, or for
        use of more familiar language the local axis.
        """
        self.geom = geom
        # Geometry of the object.
        self.basis = basis
                # Basis to allow for transformations between global and local axis.


#########################################################################################################################
```

```
37 ########################################## Standard Rules & Apply Functions ##########################################
38 ######################################################################################################################
39
40 def deleteRule(obj):
41     """
42     A rule which deletes an object.
43     """
44     delete(obj)
45     return []
46
47
48 def identityRule(obj):
49     """
50     Simply returns the object passed to it unmodified.
51     """
52     return [obj]
53
54
55 def applyAll(rule):
56     """
57     Applies a function to all objects in a given list.
58     """
59     def f(objs):
60         returnList = []
61         for obj in objs:
62             returnList += rule(obj)
63         return returnList
64     return f
65
66
67 def applyFirst(ruleA, ruleB = identityRule):
68     """
69     Applies ruleA to the FIRST element in the list, and ruleB to the rest
70     """
71     def f(objs):
72         returnList = []
73         returnList += ruleA(objs[0])
74         for obj in objs[1:]:
75             returnList += ruleB(obj)
76         return returnList
77     return f
78
79
80 def applyLast(ruleA, ruleB = identityRule):
81     """
```

```
    Applies ruleA to the LAST element in the list, and ruleB to the rest
    """
    def f(objs):
        returnList = []
        for obj in objs[:-1]:
            returnList += ruleB(obj)
        returnList += ruleA(objs[-1])
        return returnList
    return f


def applyAlternate(ruleA, ruleB = identityRule):
    """
    Applies ruleA and ruleB alternately.
    """
    def f(objs):
        returnList = []
        for i in range(len(objs)):
            if i % 2 == 0:
                returnList += ruleB(objs[i])
            else:
                returnList += ruleA(objs[i])
        return returnList
    return f


def applySingle(rule):
    """
    Applies the rule to a single object if the predicate is true.
    """
    def f(obj):
        return rule(obj)
    return f


def applyTwo(ruleA, ruleB):
    """
    Applies a two rules to a list of two objects.
    """
    def f(objs):
        returnList = []
        returnList += ruleA(objs[0])
        returnList += ruleB(objs[1])
        return returnList
    return f
```

```
127
128
129 def applyThree(ruleA, ruleB, ruleC):
130     """
131     Applies a three rules to a list of three objects.
132     """
133     def f(objs):
134         returnList = []
135         returnList += ruleA(objs[0])
136         returnList += ruleB(objs[1])
137         returnList += ruleC(objs[2])
138         return returnList
139     return f
140
141
142 ##########################################################################################################################
143 ############################################## Geometric Functions ########################################################
144 ##########################################################################################################################
145
146
147 def showBasis(obj):
148     """
149     A quick function which shows the basis for any object using simple 1,1,1 lines.
150     """
151     x = rs.AddLine((0,0,0),(1,0,0))
152     y = rs.AddLine((0,0,0),(0,1,0))
153     z = rs.AddLine((0,0,0),(0,0,1))
154     rs.TransformObject(x, obj.basis)
155     rs.TransformObject(y, obj.basis)
156     rs.TransformObject(z, obj.basis)
157
158
159 def select(obj):
160     """
161     Selects an object to show which one it is.
162     """
163     rs.SelectObject(obj.geom)
164     return obj
165
166
167 def getBounds(geom):
168     """
169     Returns the bounds and midpoints for a geometry.
170     """
171     bounds = rs.BoundingBox(geom)
```

```
172     # Calculates the bounding box of the object.
173     xMin = bounds[0][0]
174     xMax = bounds[-2][0]
175     yMin = bounds[0][1]
176     # Bounding box minimums.
177     yMax = bounds[-2][1]
178     zMin = bounds[0][2]
179     zMax = bounds[-2][2]
180     # Bounding box maximums.
181     xCen = (bounds[0][0]+bounds[-2][0])/2.0
182     yCen = (bounds[0][1]+bounds[-2][1])/2.0
183     zCen = (bounds[0][2]+bounds[-2][2])/2.0
184     # Bounding box centers.
185     return [(xMin,yMin,zMin),(xMax,yMax,zMax),(xCen,yCen,zCen)]
186
187
188 def getLocalBounds(obj):
189     """
190     Returns the bounds and midpoints for an object in it's local coordinates.
191     """
192     basis, inverseBasis = golbalToLocal(obj)
193     # Transforms the object into it's local axis.
194     bounds = getBounds(obj.geom)
195     # Gets the bounds.
196     obj = transform(obj, basis)
197     # Puts it back into the global axis
198     return bounds
199
200
201 def copy(obj):
202     """
203     Copies an object including it's geometry and basis.
204     """
205     copiedGeom = rs.CopyObject(obj.geom)
206     copiedBasis = t.copy(obj.basis)
207     # Does the copying.
208     return Obj(copiedGeom, copiedBasis)
209
210
211 def delete(obj):
212     """
213     Deletes an object.
214     """
215     rs.DeleteObject(obj.geom)
216     return []
```

```
217
218
219 def makePoly(points):
220     """
221     Makes a polygon from a list of corners.
222     """
223     points = points + [points[0]]
224     # Closes the polygon.
225     curve = rs.AddPolyline(points)
226     curve = rs.CloseCurve(curve)
227     # Creates the planar polygon from the points and closes it.
228     surface = rs.AddPlanarSrf(curve)
229     # Creates a surface from the polygon.
230     rs.DeleteObject(curve)
231     # Deletes the curve
232     return Obj(surface)
233
234
235 def makeBox(x, y, z):
236     """
237     Makes a box of given length, width, height and constructs it with it's center on the origin at 0,0,0.
238     """
239     box = rs.AddBox([ (0,0,0),(0,y,0),(x,y,0),(x,0,0),
240                       (0,0,z),(0,y,z),(x,y,z),(x,0,z) ])
241     # Makes the box.
242     rs.MoveObject( box, (-x/2, -y/2, -z/2) )
243     # Puts the box center at the origin.
244     return Obj(box)
245
246
247 def makeRec(x, y):
248     """
249     Makes a rectangle much like makeBox.
250     """
251     rec = makePoly([ (0,0,0),(0,y,0),(x,y,0),(x,0,0) ])
252     # Makes the rectangle.
253     rs.MoveObject( rec.geom, (-x/2, -y/2, 0) )
254     # Puts the rectangle center at the origin.
255     return rec
256
257
258 def makeLine(x):
259     """
260     Makes a line.
261     """
```

```
262     line = rs.AddLine( (-x/2,0,0),(x/2,0,0) )
263     return Obj(line)
264
265
266 def transform(obj, matrix, moveBasis = True, moveObj = True):
267     """
268     Performs a matrix transformation on an object given the object and the transformation matrix.  Has the option to not
269     move the basis.  Also has the option to only move the basis.
270     """
271     if moveObj:
272         rs.TransformObject(obj.geom, matrix)
273     # This transforms the objects geometry.
274     if moveBasis:
275         obj.basis = t.multiply(matrix, obj.basis)
276     # This transforms the objects basis by the same matrix.
277     return obj
278
279
280 def golbalToLocal(obj):
281     """
282     Takes an object in it's global axis an transforms it into local axis.  This returns the object, original basis and
283     original inverse basis.
284     """
285     basis = obj.basis
286     basisInverted = t.invertMat(obj.basis)
287     # Inverts the basis matrix.
288     transform(obj, basisInverted)
289     # Takes the object out of local axis using the inverted matrix.
290     return basis, basisInverted
291
292
293 def move(obj, vec, moveBasis = True, moveObj = True):
294     """
295     Moves an object and it's ascociated basis.  Uses a transformation matrix to do this.
296     """
297     matrix = t.translateMat(vec)
298     # Creates a translation matrix.
299     obj = transform(obj, matrix, moveBasis, moveObj)
300     # Does the translation.
301     return obj
302
303
304 def rotate(obj, rotVec, pointVec = "cen", moveBasis = True, moveObj = True):
305     """
306     Rotates an object about a given point using the roll pitch yaw convention.  If the point is not civen then the
```

```
307     default is to use the objects own center.
308     """
309     if pointVec == "cen":
310         bounds = getBounds(obj.geom)
311         pointVec = bounds[2]
312     # If the pointVec is the default then it sets it to the objects center.
313     rotMatrix = t.rotateMat(rotVec)
314     # Creates rotation matrix, this by default rotates about 0,0,0.
315     traMatrix = t.translateMat(pointVec)
316     traMatInv = t.translateMat( (-pointVec[0],-pointVec[1],-pointVec[2]) )
317     # Creates the translation matrix and it's inverse.
318     rotMatrix = t.multiply(traMatrix, t.multiply(rotMatrix, traMatInv))
319     # Creates the rotation matrix about pointVec rather than 0,0,0.
320     obj = transform(obj, rotMatrix, moveBasis, moveObj)
321     # Does the translation.
322     return obj
323
324
325 def scale(obj, scaleVec, pointVec = "cen", moveBasis = False, moveObj = True):
326     """
327     Performs a scaling of the object about a point, note by default this does not move the basis.
328     """
329     if pointVec == "cen":
330         bounds = getBounds(obj.geom)
331         pointVec = bounds[2]
332     # If the pointVec is the default then it sets it to the objects center.
333     scaMatrix = t.scaleMat(scaleVec)
334     # Creates the scaleing matrix, this by default is about point 0,0,0.
335     traMatrix = t.translateMat(pointVec)
336     traMatInv = t.translateMat( (-pointVec[0],-pointVec[1],-pointVec[2]) )
337     # Creates the translation matrix and it's inverse.
338     scaMatrix = t.multiply(traMatrix, t.multiply(scaMatrix, traMatInv))
339     # Creates the scaling matrix about pointVec rather than 0,0,0.
340     obj = transform(obj, scaMatrix, moveBasis, moveObj)
341     # Does the translation.
342     return obj
343
344
345 def extrude(obj, vector):
346     """
347     Extrudes a surface along the direction and distance of the given vector.
348     """
349     extrudeLine = rs.AddLine((0.0,0.0,0.0),vector)
350     # Makes the line to extrude along.
351     extrudedObject = rs.ExtrudeSurface(obj.geom, extrudeLine)
```

```
352     # Performs the extrusion
353     rs.DeleteObject(extrudeLine)
354     rs.DeleteObject(obj.geom)
355     # Deletes the extruding line and surface.
356     obj.geom = extrudedObject
357     # Reassigns the new geometry to the object.
358     return obj
359
360
361 def extrudeCurve(obj, vector):
362     """
363     Extrudes a curve along the direction and distance of the given vector.
364     """
365     extrudeLine = rs.AddLine((0.0,0.0,0.0),vector)
366     # Makes the line to extrude along.
367     extrudedObject = rs.ExtrudeCurve(obj.geom, extrudeLine)
368     # Performs the extrusion
369     rs.DeleteObject(extrudeLine)
370     rs.DeleteObject(obj.geom)
371     # Deletes the extruding line and surface.
372     obj.geom = extrudedObject
373     # Reassigns the new geometry to the object.
374     return obj
375
376
377 def extrudePoint(obj, vector):
378     # ACTUALLY GO AND FUCKING DO!
379     return obj
380
381
382 def importObj(filepath):
383     """
384     Imports geometry from file and returns an object. NOTE currently uses the filepath to the h drive directory. Places
385     the geometry at a position of 0,0,0 with no rotation.
386     """
387     rs.Command(os.path.normpath("-_Insert_H:/Dissertation/RhinoPythonScripts/"+filepath)+"_Objects_Enter_0,0,0_1_0")
388     # Constructs the filepath.
389     importedGeom = rs.LastCreatedObjects()
390     # Returns the object that was created by the command.
391     return Obj(importedGeom)
392
393
394 def arrange(obj, direction, spacing, number, local = True):
395     """
396     Copies an object into a line over a given direction with given spacing for a given number of times. If local is
```

```
397     true then it puts the object into local coordinates before operating.
398     """
399     if local:
400         basis, inverseBasis = golbalToLocal(obj)
401     # If local then puts object in local coordinates.
402     objs = []
403     # Sets up the output.
404     for i in range(number):
405         if direction == "x":
406             moveVec = (i*spacing,0,0)
407         if direction == "y":
408             moveVec = (0,i*spacing,0)
409         if direction == "z":
410             moveVec = (0,0,i*spacing)
411         # Sets up the vector to move by.
412         copiedObj = copy(obj)
413         # Creates a copy of the object.
414         copiedObj = move(copiedObj, moveVec)
415         # Moves the object and basis
416         if local:
417             transform(copiedObj, basis)
418         # If local then move the copied geometries back.
419         objs.append(copiedObj)
420     delete(obj)
421     # Hides the input.
422     return objs
423
424
425 def split(obj, direction, distances, objType = "vol" , distType = "rel", local = True):
426     """
427     Splits an object which is a volume along relative distances in either the x, y or z directions.
428     """
429     if local:
430         basis, inverseBasis = golbalToLocal(obj)
431     # If local then puts object in local coordinates.
432     objs = []
433     # Output list of objects
434     if distType == "abs":
435         distances = relFromAbs(obj, direction, distances)
436     # Make distances relative if they are put in as absolute.
437     distances = [0.0] + distances + [1.0]
438     # Adds initial values at 0 and 1 to simplyify the algorithm.
439     for i in range(len(distances)-1):
440         if (distances[i]-distances[i+1] == 0.0):
441             # Splits into a flat plane.
```

```
442             sectPlane = makeSectPlane(obj, direction, distances[i])
443             # Creates the plane to be intersected with
444             if objType == "vol":
445                 if ((distances[0] == 0.0) or (distances[0] == 1.0)):
446                     newObj = intersectSrfVolResistant(sectPlane, obj)
447                 else:
448                     newObj = intersectSrfVol(sectPlane, obj)
449             if objType == "srf":
450                 newObj = intersectSrfSrf(sectPlane, obj)
451             if objType == "crv":
452                 newObj = intersectSrfCrv(sectPlane, obj)
453             delete(sectPlane)
454             # Performs the intersection.
455         else:
456             # Splits into a volume
457             sectBox = makeSectBox(obj, direction, distances[i], distances[i+1])
458             # Creates the box to be intersected with.
459             if objType == "vol":
460                 newObj = intersectVolVol(sectBox, obj)
461             if objType == "srf":
462                 newObj = intersectSrfVol(obj, sectBox)
463             if objType == "crv":
464                 newObj = intersectCrvVol(obj, sectBox)
465             delete(sectBox)
466             # Performs the intersection
467         bounds = getBounds(newObj.geom)
468         newObj.basis = t.translateMat(bounds[2])
469         # Puts a new basis at the center of the objet created.
470         if local:
471             transform(newObj, basis)
472         # If local then move the copied geometries back.
473         objs += [newObj]
474         # Adds the new object to the output list.
475     delete(obj)
476     # Hides the initial object this must be done after, as some of the commands rely on selecyting the object.
477     return objs
478
479
480 def intersectVolVol(objA, objB):
481     """
482     Intersects two volumes with each other to create another volume.
483     VOLUME , VOLUME -> VOLUME
484     """
485     geomA = rs.CopyObject(objA.geom)
486     geomB = rs.CopyObject(objB.geom)
```

```
487     # Copies the objects so that the old ones are kept.
488     intersection = rs.BooleanIntersection(geomA, geomB)
489     # Performs the intersection.
490     if intersection is None:
491         rs.DeleteObject(geomA)
492         rs.DeleteObject(geomB)
493     # Failsafe if the intersection doesn't actually intersect
494     return Obj(intersection)
495
496
497 def intersectSrfVolResistant(objSrf, objVol):
498     """
499     Intersects a surface with a volume to create a surface.
500     robust than trimBrep.
501     SURFACE , VOLUME -> SURFACE
502     This uses specifically commands to prevent floating point errors.
503     """
504     rs.Command("SelID "+str(objSrf.geom[0]))
505     if type(objVol.geom) == list:
506         rs.Command("SelID "+str(objVol.geom[0]))
507     else:
508         rs.Command("SelID "+str(objVol.geom))
509     # Somtimes the guid is a list and sometimes not...
510     # Issues commands to select the two objects
511     # This behaviour is properly weird check it out.
512     rs.Command("intersect")
513     # Issues the command to intersect the objects.
514     curve = rs.LastCreatedObjects()
515     # Selects the object created by the command.
516     intersection = rs.AddPlanarSrf(curve)
517     # Creates a surface from the curve
518     rs.DeleteObject(curve)
519     return Obj(intersection)
520
521
522 def intersectSrfVol(objSrf, objVol):
523     """
524     Intersects a surface with a volume to create a surface.
525     SURFACE , VOLUME -> SURFACE
526     """
527     geomSrf = rs.CopyObject(objSrf.geom)
528     # Copies the surface geometry.
529     intersection = rs.TrimBrep(geomSrf, objVol.geom)
530     if len(intersection) == 0:
531         intersection = rs.CopyObject(objSrf.geom)
```

```
532             rs.DeleteObject(geomSrf)
533         # Trims the surface by the volume to create the intersection of the two.  CAUSES FP ERROR + OUTPUTS LIST.
534         return Obj(intersection)
535
536
537 def intersectSrfSrf(objA, objB):
538         """
539         Intersects a surface with a surface to make a curve.
540         SURFACE, SURFACE -> CURVE
541         """
542         intersection = rs.IntersectBreps(objA.geom, objB.geom)[0]
543         # Performs the intersection.
544         return Obj(intersection)
545
546
547 def intersectCrvVol(objCrv, objVol):
548         """
549         Intersects a curve with a volume to make a curve.
550         CURVE, VOLUME -> CURVE
551         """
552         points = rs.CurveBrepIntersect(objCrv.geom, objVol.geom)
553         # Performs intersection.  SEEMS TO OUTPUT A LIST OF TWO ELEMENTS WHERE ONLY THE LAST CONTAINS POINTS WTF?
554         intersection = rs.AddLine( points[1][0], points[1][1] )
555         # Stitches the two resulting points together.
556         rs.DeleteObject(points[1][0])
557         rs.DeleteObject(points[1][1])
558         # Deletes the points
559         return Obj(intersection)
560
561
562 def intersectSrfCrv(objSrf, objCrv):
563         """
564         Intersects a curve with a volume to make a curve.
565         CURVE, VOLUME -> CURVE
566         """
567         intersection = rs.CurveBrepIntersect(objCrv.geom, objSrf.geom)
568         # Performs intersection. For some reason returns a list!? I guess a curve can have multiple intersections.
569         return Obj(intersection[1][0])
570
571
572 def makeSectBox(obj, direction, distance1, distance2):
573         """
574         Creates a box to be intersected with in the split operation.
575         """
576         bounds = rs.BoundingBox(obj.geom)
```

```
    # Calculates the bounding box of the object.
    xMin = bounds[0][0]  -1
    xMax = bounds[-2][0] +1
    yMin = bounds[0][1]  -1
    yMax = bounds[-2][1] +1
    zMin = bounds[0][2]  -1
    zMax = bounds[-2][2] +1
    # Explicitly extracts the various bounds.
    if (direction == "x"):
        xMin = (distance1 * (bounds[-2][0]-bounds[0][0])) + bounds[0][0]
        xMax = (distance2 * (bounds[-2][0]-bounds[0][0])) + bounds[0][0]
    if (direction == "y"):
        yMin = (distance1 * (bounds[-2][1]-bounds[0][1])) + bounds[0][1]
        yMax = (distance2 * (bounds[-2][1]-bounds[0][1])) + bounds[0][1]
    if (direction == "z"):
        zMin = (distance1 * (bounds[-2][2]-bounds[0][2])) + bounds[0][2]
        zMax = (distance2 * (bounds[-2][2]-bounds[0][2])) + bounds[0][2]
    # Determines which bounds should be replaced given the conditions and converts from relative to absolute distance.
    cutBox = rs.AddBox([ (xMin,yMin,zMin),(xMax,yMin,zMin),(xMax,yMax,zMin),(xMin,yMax,zMin),
                         (xMin,yMin,zMax),(xMax,yMin,zMax),(xMax,yMax,zMax),(xMin,yMax,zMax) ])
    # Creates the cutting box.
    return Obj(cutBox)


def makeSectPlane(obj, direction, distance):
    """
    Creates a plane to be intersected with.
    """
    bounds = rs.BoundingBox(obj.geom)
    # Calculates the bounding box of the object.
    xMin = bounds[0][0]  -1
    xMax = bounds[-2][0] +1
    yMin = bounds[0][1]  -1
    yMax = bounds[-2][1] +1
    zMin = bounds[0][2]  -1
    zMax = bounds[-2][2] +1
    # Explicitly extracts the various bounds.
    if (direction == "x"):
        distance = (distance * (bounds[-2][0]-bounds[0][0])) + bounds[0][0]
        cutPlane = makePoly([ (distance,yMin,zMin),(distance,yMax,zMin),(distance,yMax,zMax),(distance,yMin,zMax) ])
    if (direction == "y"):
        distance = (distance * (bounds[-2][1]-bounds[0][1])) + bounds[0][1]
        cutPlane = makePoly([ (xMin,distance,zMin),(xMin,distance,zMax),(xMax,distance,zMax),(xMax,distance,zMin) ])
    if (direction == "z"):
        distance = (distance * (bounds[-2][2]-bounds[0][2])) + bounds[0][2]
```

```
622             cutPlane = makePoly([ (xMin,yMin,distance),(xMax,yMin,distance),(xMax,yMax,distance),(xMin,yMax,distance) ])
623         # Determines which bounds should be replaced given the conditions and converts from relative to absolute distance.
624         return cutPlane
625
626
627 def relFromAbs(obj, direction, distances):
628         """
629         Converts into relative distances from absolute distances.
630         """
631         bounds = rs.BoundingBox(obj.geom)
632         # Calculates the bounding box of the object.
633         if direction == "x":
634             factor = (bounds[-2][0]-bounds[0][0])
635         if direction == "y":
636             factor = (bounds[-2][1]-bounds[0][1])
637         if direction == "z":
638             factor = (bounds[-2][2]-bounds[0][2])
639         # Explicitly extracts the various bounds.
640         for i in range(len(distances)):
641             distances[i] = distances[i]/factor
642         # Divides the distances by the length of the object in the particular direction.
643         return distances
644
645
646 def offsetSingle(obj, distance, angleA, angleB, lineLeft = False, lineRight = False):
647         """
648         Offsets a single line into a surface.
649         """
650         objs = []
651         # makes the output.
652         basis = obj.basis
653         basisInverted = t.invertMat(obj.basis)
654         transform(obj, basisInverted)
655         # Puts object in local coordinates.
656         bounds = getBounds(obj.geom)
657         pointA = (bounds[0][0],0,0)
658         pointB = (bounds[1][0],0,0)
659         pointC = (bounds[1][0] + distance*m.tan(m.radians(angleB)),distance,0)
660         pointD = (bounds[0][0] - distance*m.tan(m.radians(angleA)),distance,0)
661         # Calculates the corners of the polygon.
662         if lineLeft:
663             line = Obj(rs.AddLine(pointA,pointD), t.rotateMat((0,0,angleA)))
664             line = move(line, getBounds(line.geom)[2], moveObj = False)
665             objs += [line]
666         # Make the lefthand line.
```

```
667     objs += [makePoly([pointA,pointB,pointC,pointD])]
668     # Makes the polygon.
669     if lineRight:
670         line = Obj(rs.AddLine(pointB,pointC), t.rotateMat((0,0,angleB)))
671         line = move(line, getBounds(line.geom)[2], moveObj = False)
672         objs += [line]
673     # Makes the righthand line.
674     delete(obj)
675     # Deletes the original line.
676     for i in objs:
677         transform(i, basis)
678     return objs
679
680
681 def lineAngle(pointA,pointB):
682     """
683     Calculates the angle of a line given two points.
684     """
685     opposite = (pointB[1]-pointA[1])
686     adjacent = (pointB[0]-pointA[0])
687     # Calculates the opposite and adjacent.
688     return m.degrees(m.atan2(opposite, adjacent))
689
690
691 def offsetLine(points, distance):
692     """
693     Creates a series of lines from points then creates offset surfaces from them.
694     """
695     objs = []
696     # outputs.
697     for i in range(len(points)-1):
698         center = (((points[i][0]+points[i+1][0])/2), ((points[i][1]+points[i+1][1])/2), 0)
699         # Calculates the center point of the line.
700         lineLen = ((points[i+1][0]-points[i][0])**2 + (points[i+1][1]-points[i][1])**2)**0.5
701         # Calculate the length of the line.
702         currAngle = lineAngle(points[i],points[i+1])
703         # Calculates the angle of the line
704         line = makeLine(lineLen)
705         line = move(line, center)
706         line = rotate(line, (0,0,currAngle))
707         # Creates the line and moves it into position
708         angleA = 0.0
709         angleB = 0.0
710         # Default options for the ends simply set the angles equal to zero
711         if i != 0:
```

```
712             anglePrev = lineAngle(points[i-1],points[i])
713             angleA = (anglePrev-currAngle)/2
714         if i != len(points)-2:
715             angleNext = lineAngle(points[i+1],points[i+2])
716             angleB = (currAngle-angleNext)/2
717         # Calculates angle A and B which are the averages of the previous and next.
718         if i != len(points)-2:
719             objs += offsetSingle(line, distance, angleA, angleB, lineLeft = True)
720         else:
721             objs += offsetSingle(line, distance, angleA, angleB, lineLeft = True, lineRight = True)
722         # Creates the offset surfaces.
723     return objs
724
725
726 def subtractVolVol(obj, subtractor):
727     """
728     performs a boolean subtraction on two solids.
729     """
730     origObj = copy(obj)
731     geomList = rs.BooleanDifference(obj.geom, subtractor.geom)
732     if len(geomList) != 0:
733         obj.geom = geomList[0]
734         delete(origObj)
735         return obj
736     else:
737         delete(obj)
738         return origObj
739
740
741 ##########################################################################################################################
742 ##########################################################################################################################
743 ##########################################################################################################################
```

Matrix Code

This is the code for handling the matrices for the basis, this is key to enabling proper transformations and rotations.

```
"""
This is to handle the complex mathematics regarding transformation matrix manipulation and local axis.
"""


################################################################################################################
############################################################ Imports ###########################################
################################################################################################################


import math as m
import rhinoscriptsyntax as rs


################################################################################################################
################################################### Geometric Functions ########################################
################################################################################################################


def printMat(A):
    """
    Prints a matrix nicely.
    Thanks to https://integratedmlai.com/matrixinverse/
    """
    for row in A:
        print([round(x,3)+0 for x in row])


def idenMat():
    """
    Creates an identity matrix of size 4x4.
    """
    return [[1.0, 0.0, 0.0, 0.0],
            [0.0, 1.0, 0.0, 0.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 1.0]]
```

```
37
38
39  def zeroMat():
40      """
41      Creates a zeros matrix of size 4x4.
42      """
43      return [[0.0, 0.0, 0.0, 0.0],
44              [0.0, 0.0, 0.0, 0.0],
45              [0.0, 0.0, 0.0, 0.0],
46              [0.0, 0.0, 0.0, 0.0]]
47
48
49  def multiply(A, B):
50      """
51      Multiplies two 4x4 matricies together.
52      Thanks to https://integratedmlai.com/matrixinverse/
53      """
54      C = zeroMat()
55      for i in range(4):
56          for j in range(4):
57              total = 0
58              for k in range(4):
59                  total += A[i][k] * B[k][j]
60              C[i][j] = total
61      return C
62
63
64  def copy(A):
65      """
66      Copies a 4x4 matrix.
67      Thanks to https://integratedmlai.com/matrixinverse/
68      """
69      B = zeroMat()
70      for i in range(4):
71          for j in range(4):
72              B[i][j] = A[i][j]
73      return B
74
75
76  def invertMatInexact(A):
77      """
78      Returns the inverse of the passed in matrix.
79      Thanks to https://integratedmlai.com/matrixinverse/
80      """
81      n = 4
```

```
    AM = copy(A)
    I = idenMat()
    IM = copy(I)
    # Section 3: Perform row operations
    indices = list(range(n)) # to allow flexible row referencing
    for fd in range(n): # fd stands for focus diagonal
        fdScaler = 1.0 / AM[fd][fd]
        # FIRST: scale fd row with fd inverse.
        for j in range(n): # Use j to indicate column looping.
            AM[fd][j] *= fdScaler
            IM[fd][j] *= fdScaler
        # SECOND: operate on all rows except fd row as follows:
        for i in indices[0:fd] + indices[fd+1:]: # skip fd row
            crScaler = AM[i][fd] # cr stands for "current row".
            for j in range(n): # cr - crScaler * fdRow
                AM[i][j] = AM[i][j] - crScaler * AM[fd][j]
                IM[i][j] = IM[i][j] - crScaler * IM[fd][j]
    return IM


def invertMat(A):
    """
    Brute force analytical matrix inversion - this is exact so long as the determinate != 0.
    thanks to https://codedump.io/share/GCNSbWGphkFZ/1/matrix-inversion-without-numpy
    and http://cg.info.hiroshima-cu.ac.jp/~miyazaki/knowledge/teche23.html
    """

    b00 = (A[1][1]*A[2][2]*A[3][3] + A[1][2]*A[2][3]*A[3][1] + A[1][3]*A[2][1]*A[3][2] - A[1][1]*A[2][3]*A[3][2]
        - A[1][2]*A[2][1]*A[3][3] - A[1][3]*A[2][2]*A[3][1])
    b01 = (A[0][1]*A[2][3]*A[3][2] + A[0][2]*A[2][1]*A[3][3] + A[0][3]*A[2][2]*A[3][1] - A[0][1]*A[2][2]*A[3][3]
        - A[0][2]*A[2][3]*A[3][1] - A[0][3]*A[2][1]*A[3][2])
    b02 = (A[0][1]*A[1][2]*A[3][3] + A[0][2]*A[1][3]*A[3][1] + A[0][3]*A[1][1]*A[3][2] - A[0][1]*A[1][3]*A[3][2]
        - A[0][2]*A[1][1]*A[3][3] - A[0][3]*A[1][2]*A[3][1])
    b03 = (A[0][1]*A[1][3]*A[2][2] + A[0][2]*A[1][1]*A[2][3] + A[0][3]*A[1][2]*A[2][1] - A[0][1]*A[1][2]*A[2][3]
        - A[0][2]*A[1][3]*A[2][1] - A[0][3]*A[1][1]*A[2][2])

    b10 = (A[1][0]*A[2][3]*A[3][2] + A[1][2]*A[2][0]*A[3][3] + A[1][3]*A[2][2]*A[3][0] - A[1][0]*A[2][2]*A[3][3]
        - A[1][2]*A[2][3]*A[3][0] - A[1][3]*A[2][0]*A[3][2])
    b11 = (A[0][0]*A[2][2]*A[3][3] + A[0][2]*A[2][3]*A[3][0] + A[0][3]*A[2][0]*A[3][2] - A[0][0]*A[2][3]*A[3][2]
        - A[0][2]*A[2][0]*A[3][3] - A[0][3]*A[2][2]*A[3][0])
    b12 = (A[0][0]*A[1][3]*A[3][2] + A[0][2]*A[1][0]*A[3][3] + A[0][3]*A[1][2]*A[3][0] - A[0][0]*A[1][2]*A[3][3]
        - A[0][2]*A[1][3]*A[3][0] - A[0][3]*A[1][0]*A[3][2])
    b13 = (A[0][0]*A[1][2]*A[2][3] + A[0][2]*A[1][3]*A[2][0] + A[0][3]*A[1][0]*A[2][2] - A[0][0]*A[1][3]*A[2][2]
        - A[0][2]*A[1][0]*A[2][3] - A[0][3]*A[1][2]*A[2][0])

```

```
    b20 = (A[1][0]*A[2][1]*A[3][3] + A[1][1]*A[2][3]*A[3][0] + A[1][3]*A[2][0]*A[3][1] - A[1][0]*A[2][3]*A[3][1]
        - A[1][1]*A[2][0]*A[3][3] - A[1][3]*A[2][1]*A[3][0])
    b21 = (A[0][0]*A[2][3]*A[3][1] + A[0][1]*A[2][0]*A[3][3] + A[0][3]*A[2][1]*A[3][0] - A[0][0]*A[2][1]*A[3][3]
        - A[0][1]*A[2][3]*A[3][0] - A[0][3]*A[2][0]*A[3][1])
    b22 = (A[0][0]*A[1][1]*A[3][3] + A[0][1]*A[1][3]*A[3][0] + A[0][3]*A[1][0]*A[3][1] - A[0][0]*A[1][3]*A[3][1]
        - A[0][1]*A[1][0]*A[3][3] - A[0][3]*A[1][1]*A[3][0])
    b23 = (A[0][0]*A[1][3]*A[2][1] + A[0][1]*A[1][0]*A[2][3] + A[0][3]*A[1][1]*A[2][0] - A[0][0]*A[1][1]*A[2][3]
        - A[0][1]*A[1][3]*A[2][0] - A[0][3]*A[1][0]*A[2][1])

    b30 = (A[1][0]*A[2][2]*A[3][1] + A[1][1]*A[2][0]*A[3][2] + A[1][2]*A[2][1]*A[3][0] - A[1][0]*A[2][1]*A[3][2]
        - A[1][1]*A[2][2]*A[3][0] - A[1][2]*A[2][0]*A[3][1])
    b31 = (A[0][0]*A[2][1]*A[3][2] + A[0][1]*A[2][2]*A[3][0] + A[0][2]*A[2][0]*A[3][1] - A[0][0]*A[2][2]*A[3][1]
        - A[0][1]*A[2][0]*A[3][2] - A[0][2]*A[2][1]*A[3][0])
    b32 = (A[0][0]*A[1][2]*A[3][1] + A[0][1]*A[1][0]*A[3][2] + A[0][2]*A[1][1]*A[3][0] - A[0][0]*A[1][1]*A[3][2]
        - A[0][1]*A[1][2]*A[3][0] - A[0][2]*A[1][0]*A[3][1])
    b33 = (A[0][0]*A[1][1]*A[2][2] + A[0][1]*A[1][2]*A[2][0] + A[0][2]*A[1][0]*A[2][1] - A[0][0]*A[1][2]*A[2][1]
        - A[0][1]*A[1][0]*A[2][2] - A[0][2]*A[1][1]*A[2][0])

    detA = A[0][0]*b00 + A[0][1]*b10 + A[0][2]*b20 + A[0][3]*b30

    Ainv = [[b00/detA, b01/detA, b02/detA, b03/detA], [b10/detA, b11/detA, b12/detA, b13/detA],
            [b20/detA, b21/detA, b22/detA, b23/detA], [b30/detA, b31/detA, b32/detA, b33/detA]]

    return Ainv


def translateMat(vec):
    """
    Creates a translation matrix.
    """
    return [[1.0, 0.0, 0.0, vec[0]],
            [0.0, 1.0, 0.0, vec[1]],
            [0.0, 0.0, 1.0, vec[2]],
            [0.0, 0.0, 0.0,    1.0]]


def scaleMat(vec):
    """
    Creates a translation matrix.
    """
    return [[vec[0],     0.0,     0.0, 0.0],
            [   0.0, vec[1],     0.0, 0.0],
            [   0.0,     0.0, vec[2], 0.0],
            [   0.0,     0.0,     0.0, 1.0]]

```

```
172
173 def rollMat(roll):
174     """
175     Creates a roll matrix.
176     """
177     roll = m.radians(roll)
178     return [[1.0,          0.0,          0.0, 0.0],
179             [0.0, m.cos(roll), -m.sin(roll), 0.0],
180             [0.0, m.sin(roll),  m.cos(roll), 0.0],
181             [0.0,          0.0,          0.0, 1.0]]
182
183
184 def pitchMat(pitch):
185     """
186     Creates a pitch matrix.
187     """
188     pitch = m.radians(pitch)
189     return [[ m.cos(pitch), 0.0, m.sin(pitch), 0.0],
190             [          0.0, 1.0,          0.0, 0.0],
191             [-m.sin(pitch), 0.0, m.cos(pitch), 0.0],
192             [          0.0, 0.0,          0.0, 1.0]]
193
194
195 def yawMat(yaw):
196     """
197     Creates a yaw matrix.
198     """
199     yaw = m.radians(yaw)
200     return [[m.cos(yaw), -m.sin(yaw), 0.0, 0.0],
201             [m.sin(yaw),  m.cos(yaw), 0.0, 0.0],
202             [       0.0,         0.0, 1.0, 0.0],
203             [       0.0,         0.0, 0.0, 1.0]]
204
205
206 def rotateMat(vec):
207     """
208     Creates a full tate-bryon rotation matrix.
209     """
210     Rx = rollMat(vec[0]) # roll
211     Ry = pitchMat(vec[1]) # pitch
212     Rz = yawMat(vec[2]) # yaw
213     # Creates the individual matricies.
214     return multiply(Rz,multiply(Ry,Rx))
215
216
```

```
217 ##########################################################################################################################
218 ##########################################################################################################################
219 ##########################################################################################################################
```

Building Script

This is the script which generates the buildings in the results chapters. This particular one generates a specific building, with the rest generated by modifying the global parameters and floorplan. The coding style here is different to that in the other code as it is intended to represent how a practical script may be written. The code for the library functions is included in the same script for convininces.

```
"""
Final building for the dissertation.
"""

import rhinoscriptsyntax as rs
import os
import math as m
import transMat as t
from BIMcurry import *

##########################################################################################################
################################################## Library Functions #####################################
##########################################################################################################

def intSplit(number):
    """
    Creates a list tthat provides the locations for an integer number of splits.
    """
    output = []
    for i in range(number-1):
        output.append((i+1)/number)
    return output

def intSplitWalls(number):
    """
    Creates a list tthat provides the locations for an integer number of splits.
    """
    output = []
    for i in range(number-1):
        output.append((i+1)/number)
        output.append((i+1)/number)
    return output
```

```
33
34  def metSecWall(obj, spacing = 1.0, panelA = "none", panelB = "none"):
35      """
36      Rule to make a parametric infill wall, with a variety of different options.
37      """
38
39      basis = obj.basis
40      basisInverted = t.invertMat(obj.basis)
41      transform(obj, basisInverted)
42      # Removes from the original basis.
43
44      bounds = getBounds(obj.geom)
45      length = bounds[1][0] - bounds[0][0]
46      height = bounds[1][2] - bounds[0][2]
47      number = int(length/spacing)-1
48      if number <= 0:
49          number = 0
50      spacing = length/(number+1)
51      # Calculates global variables.
52
53      def importRail():
54          return  applySingle(shiftRail)  (importObj( "Sections/U-124-70-12.3dm" ))
55      def shiftRail(obj):
56          return  applySingle(arrangeRail)  (move( obj, (0,0,0.035) ))
57      def arrangeRail(obj):
58          return  applyFirst(rotateBotRail, rotateTopRail)  (arrange(obj, "z", height-0.07, 2))
59      def rotateTopRail(obj):
60          return  applySingle(extrudeRail)  (rotate( obj, (90, 180, 90) ))
61      def rotateBotRail(obj):
62          return  applySingle(extrudeRail)  (rotate( obj, (-90, 180, 90) ))
63      def extrudeRail(obj):
64          return  applySingle(centObjs) (extrude( obj, (length,0,0) ))
65      # Creates the top and bottom rails
66
67      def importSide():
68          return  applySingle(shiftSide)  (importObj( "Sections/C-120-75-12.3dm" ))
69      def shiftSide(obj):
70          return  applySingle(arrangeSide)  (move( obj, (0.0375,0,0.02) ))
71      def arrangeSide(obj):
72          return  applyFirst(rotateRightSide, rotateLeftSide)  (arrange(obj, "x", length-0.075, 2))
73      def rotateLeftSide(obj):
74          return  applySingle(extrudeSide)  (rotate( obj, (0, 0, 90) ))
75      def rotateRightSide(obj):
76          return  applySingle(extrudeSide)  (rotate( obj, (0, 0, -90) ))
77      def extrudeSide(obj):
```

```
78          return  applySingle(centObjs) (extrude( obj, (0,0,height-0.04) ))
79      # Creates the side rails.
80
81      def importMids():
82          return  applySingle(shiftMids)  (importObj( "Sections/C-120-75-12.3dm" ))
83      def shiftMids(obj):
84          return  applySingle(arrangeMids)  (move( obj, (spacing,0,0.02) ))
85      def arrangeMids(obj):
86          return  applyAll(rotateMids)  (arrange(obj, "x", spacing, number))
87      def rotateMids(obj):
88          return  applySingle(extrudeMids)  (rotate( obj, (0, 0, 90) ))
89      def extrudeMids(obj):
90          return  applySingle(centObjs) (extrude( obj, (0,0,height-0.04) ))
91      # Creates the mid posts.
92
93      def genIns():
94          panel = makePoly([(0,0,0),(spacing,0,0),(spacing,0,height-0.04),(0,0,height-0.04)])
95          return  applySingle(shiftIns)  (extrude( panel, (0,0.1176,0) ))
96      def shiftIns(obj):
97          return  applySingle(arrangeIns)  (move( obj, (0,-0.0585,0.02) ))
98      def arrangeIns(obj):
99          return  applyAll(centObjs) (arrange( obj, "x", spacing, number+1 ))
100     # Creates the insulation
101
102     # Optional panels over the drywall
103     def genDryWallA():
104         panel = makePoly([(0,0,0),(length,0,0),(length,0,height),(0,0,height)])
105         return  applySingle(shiftDryWallA)  (extrude( panel, (0,0.015,0) ))
106     def shiftDryWallA(obj):
107         return  applySingle(centObjs)  (move( obj, (0,0.062,0) ))
108     def genDryWallB():
109         panel = makePoly([(0,0,0),(length,0,0),(length,0,height),(0,0,height)])
110         return  applySingle(shiftDryWallB)  (extrude( panel, (0,0.015,0) ))
111     def shiftDryWallB(obj):
112         return  applySingle(centObjs)  (move( obj, (0,-0.062-0.015,0) ))
113     # Used for putting just drywall panels on.
114     def genInsulationWallA():
115         panel = makePoly([(0,0,0),(length,0,0),(length,0,height),(0,0,height)])
116         return  applySingle(shiftInsulationWallA)  (extrude( panel, (0,0.1,0) ))
117     def shiftInsulationWallA(obj):
118         return  applySingle(centObjs)  (move( obj, (0,0.062,0) ))
119     def genInsulationWallB():
120         panel = makePoly([(0,0,0),(length,0,0),(length,0,height),(0,0,height)])
121         return  applySingle(shiftInsulationWallB)  (extrude( panel, (0,0.1,0) ))
122     def shiftInsulationWallB(obj):
```

```
123             return  applySingle(centObjs)  (move( obj, (0,-0.062-0.1,0) ))
124         # Used for putting insulation panels on.
125         def genDryWallAShift():
126             panel = makePoly([(0,0,0),(length,0,0),(length,0,height),(0,0,height)])
127             return  applySingle(shiftDryWallAShift)  (extrude( panel, (0,0.015,0) ))
128         def shiftDryWallAShift(obj):
129             return  applySingle(centObjs)  (move( obj, (0,0.062+0.1,0) ))
130         def genDryWallBShift():
131             panel = makePoly([(0,0,0),(length,0,0),(length,0,height),(0,0,height)])
132             return  applySingle(shiftDryWallBShift)  (extrude( panel, (0,0.015,0) ))
133         def shiftDryWallBShift(obj):
134             return  applySingle(centObjs)  (move( obj, (0,-0.062-0.015-0.1,0) ))
135         # Used for putting drywll over insulation panels.
136
137
138         def centObjs(obj):
139             return applySingle(returnToBasis) (move( obj, (bounds[2][0]-length/2, 0, bounds[2][2]-height/2) ))
140         # Shifts the walls objects to a center point.
141         def returnToBasis(obj):
142             return [transform(obj, basis)]
143
144         objs = importRail()+importSide()+importMids()+genIns()
145
146         if panelA == "dry":
147             objs += genDryWallA()
148         if panelB == "dry":
149             objs += genDryWallB()
150         if panelA == "party":
151             objs += genInsulationWallA() + genDryWallAShift()
152         if panelB == "party":
153             objs += genInsulationWallB() + genDryWallBShift()
154
155         delete(obj)
156         # Deletes the orginal object.
157         return objs
158
159 def makeStair(height):
160         """
161         Makes a staircase of a given height.
162         """
163
164         width = height*2
165         number = int(width/0.3) + 1
166
167         def makeSection():
```

```
168         return applySingle(rotateSection) (makeRec( 2,0.5 ))
169     def rotateSection(obj):
170         return applySingle(extrudeSection) (rotate( obj, (90,0,0), moveBasis = False ))
171     def extrudeSection(obj):
172         return [extrude( obj, (0,width,height) )]
173 
174     def makeSteps():
175         return applySingle(rotateSteps) (makeRec( 2.5, 0.5 ))
176     def rotateSteps(obj):
177         return applySingle(extrudeSteps) (rotate( obj, (90,0,0), moveBasis = False ))
178     def extrudeSteps(obj):
179         return applySingle(shiftSteps) (extrude( obj, (0,0.4,0) ))
180     def shiftSteps(obj):
181         return applySingle(rotateIntoAxis) (move( obj, (0,-0.1,0.4) ))
182     def rotateIntoAxis(obj):
183         return applySingle(arrangeSteps) (rotate( obj, (-26.56505118,0,0), (0,0,0) ))
184     def arrangeSteps(obj):
185         return applyAll(rotateBackIntoAxis) (arrange( obj, 'y', (5**0.5)*(3/20), number, local = False ))
186     def rotateBackIntoAxis(obj):
187         return [rotate( obj, (26.56505118,0,0), (0,0,0) )]
188 
189     def cutSteps(objs, obj):
190         obj = subtractVolVol(obj, (objs[0]))
191         if len(objs) == 1:
192             return obj
193         else:
194             return cutSteps(objs[1:], obj)
195 
196     steps = makeSteps()
197     stair = makeSection()[0]
198     cutSteps(steps, stair)
199 
200     return stair
201 
202 def makeBeam(obj):
203     """
204     Makes a beam on a horizontal line in the x direction.
205     """
206     basis = obj.basis
207     inverseBasis = t.invertMat(basis)
208 
209     obj = transform(obj, inverseBasis)
210     bounds = getBounds(obj.geom)
211     length  = bounds[1][0]-bounds[0][0]
212 
```

```
213     def importSection():
214         return applySingle(rotateSection) (importObj( "Sections/UKB-305-165-40.3dm" ))
215     def rotateSection(obj):
216         return applySingle(moveSection) (rotate( obj, (90, 0, 90), moveBasis = False ))
217     def moveSection(obj):
218         return applySingle(extrudeSection) (move( obj, (-length/2, 0, 0), moveBasis = False ))
219     def extrudeSection(obj):
220         return applySingle(returnToBasis) (extrude( obj, (length,0,0) ))
221     def returnToBasis(obj):
222         return [transform(obj, basis)]
223 
224     delete(obj)
225     return importSection()
226 
227 def hollowCoreFloor(obj):
228     # Creates a hollow core floor.
229     basis = obj.basis
230     inverseBasis = t.invertMat(basis)
231 
232     obj = transform(obj, inverseBasis)
233     bounds = getBounds(obj.geom)
234     length = bounds[1][0] - bounds[0][0]
235     width = bounds[1][1] - bounds[0][1]
236     if width % 1.2 == 0.0:
237         number = int(width/1.2)
238     else:
239         number = int(width/1.2) + 1
240 
241     cuttingObject = extrude( obj, (0,0,1) )
242 
243     def importPlanks():
244         return applySingle(rotatePlanks) (importObj( "Sections/HC-1200-200.3dm" ))
245     def rotatePlanks(obj):
246         return applySingle(arrangePlanks) (rotate( obj, (90,0,90), moveBasis = False))
247     def arrangePlanks(obj):
248         return applyAll(shiftPlanks) (arrange( obj, "y", 1.2, number ))
249     def shiftPlanks(obj):
250         return applySingle(extrudePlanks) (move( obj, (-length/2.0, -width/2.0+0.6, 0.1) ))
251     def extrudePlanks(obj):
252         return applySingle(cutPlanks) (extrude( obj, (length,0,0) ))
253     def cutPlanks(obj):
254         cut = intersectVolVol( obj, cuttingObject ).geom
255         delete(obj)
256         if cut is None:
257             return []
```

```
        else:
            return applySingle(returnToBasis) (Obj(cut))
    def returnToBasis(obj):
        return [transform( obj, basis )]

    objs = importPlanks()
    delete(cuttingObject)
    return objs

def makeColumn(obj, height):
    """
    Makes a column from a point.
    """
    basis = obj.basis
    inverseBasis = t.invertMat(basis)

    obj = transform(obj, inverseBasis)
    bounds = getBounds(obj.geom)

    def importSection():
        return applySingle(extrudeSection) (importObj( "Sections/UKC-305-305-97.3dm" ))
    def extrudeSection(obj):
        return applySingle(returnToBasis) (extrude( obj, (0,0,height) ))
    def returnToBasis(obj):
        return [transform(obj, basis)]

    delete(obj)
    return importSection()

def makeBracing(obj):
    """
    Makes bracing given a wall.
    """
    basis = obj.basis
    inverseBasis = t.invertMat(basis)
    obj = transform(obj, inverseBasis)

    bounds = getBounds(obj.geom)
    length = bounds[1][0] - bounds[0][0]
    height = bounds[1][2] - bounds[0][2]
    strutLength = (length**2+height**2)**0.5
    angle = m.degrees(m.atan2(height, length))

    def makeBrace():
        return applySingle(shiftBrace) (makeBeam(makeLine(strutLength))[0])
```

```
303     def shiftBrace(obj):
304         return applySingle(copyBrace) (move( obj, bounds[2] ))
305     def copyBrace(obj):
306         return applyFirst(rotateBraceA, rotateBraceB) ([obj, copy(obj)])
307     def rotateBraceA(obj):
308         return applySingle(returnToBasis) (rotate( obj, (0, angle, 0) ))
309     def rotateBraceB(obj):
310         return applySingle(returnToBasis) (rotate( obj, (0, -angle, 0) ))
311     def returnToBasis(obj):
312         return [transform(obj, basis)]
313 
314     delete(obj)
315     return makeBrace()
316 
317 ##########################################################################################################################
318 ################################################## Geometry Set Up ########################################################
319 ##########################################################################################################################
320 
321 ################################################ Sets up Code Geometry ####################################################
322 def makeStairwellA(obj):
323     bounds = getLocalBounds(obj)
324     length = bounds[1][0] - bounds[0][0]
325     relativeDistance = 6/length
326     return applyLast(splitBlock, StairwellCopy) (split( obj, 'x', [relativeDistance], 'srf' ))
327 # Makes a 6m wide stairwell on the first block.
328 def makeStairwellB(obj):
329     bounds = getLocalBounds(obj)
330     length = bounds[1][0] - bounds[0][0]
331     relativeDistance = 1-(6/length)
332     basis, inverseBasis = golbalToLocal(obj)
333     scale(obj, (-1,1,1), moveBasis = True, moveObj = False)
334     transform(obj, basis)
335     return applyLast(splitBlock, StairwellCopy) (split( obj, 'x', [1-relativeDistance], 'srf' ))
336 # Makes a 6m wide stairwell on the last block.
337 def splitBlock(obj):
338     return applyThree(BeamCopy, makeCorridor, BeamCopy) (split( obj, 'y', [0.0,1.0], 'srf' ))
339 def makeCorridor(obj):
340     bounds = getLocalBounds(obj)
341     width = bounds[1][1] - bounds[0][1]
342     relativeDistance = 1/width
343     return applyThree(splitFlats, splitCorridor, mirrorBottom) (split( obj, 'y',
344         [0.5-relativeDistance, 0.5+relativeDistance], 'srf' ))
345 # Splits the building blocks to make the corridors.
346 def splitCorridor(obj):
347     return applyThree(BeamCopy, CorridorCopy, BeamCopy) (split( obj, 'y', [0.0,1.0], 'srf' ))
```

```
348 def mirrorBottom(obj):
349     basis, inverseBasis = golbalToLocal(obj)
350     scale(obj, (1,-1,1), moveBasis = True, moveObj = False)
351     transform(obj, basis)
352     return applySingle(splitFlats)(obj)
353 # Mirrors vertically the bottom flats.
354 def splitFlats(obj, number=1):
355     testObjs = split( copy(obj), 'x', intSplit(number), 'srf' )
356     if testFlatSize(testObjs):
357         applyAll(delete)(testObjs)
358         return splitFlats(obj, number+1)
359     elif number == 2:
360         applyAll(delete)(testObjs)
361         return applySingle(mirrorFlat)(obj)
362     else:
363         applyAll(delete)(testObjs)
364         splitObjs = (split( copy(obj), 'x', intSplitWalls(number-1), 'srf' ))
365         return FlatCopy(splitObjs[0]) + applyAlternate(mirrorFlat, FlatPartyWallCopy)(splitObjs[1:])
366 # Perfomrs a recursive loop to split the flat blocks into the largest number of flats, whilst still exceeding the global
367 # flat area.
368 def testFlatSize(objs):
369     if rs.SurfaceArea(objs[0].geom)[0] < globalFlatArea:
370         return False
371     elif len(objs) == 1:
372         return True
373     else:
374         return testFlatSize(objs[1:])
375 # Recursively loops through objects to find if they are smaller than the global flat floor area.
376 def mirrorFlat(obj):
377     basis, inverseBasis = golbalToLocal(obj)
378     scale(obj, (-1,1,1), moveBasis = True, moveObj = False)
379     transform(obj, basis)
380     return applySingle(FlatCopy)(obj)
381 # Mirrors the end flats so they generate properly.
382
383 ################################################## Copies Geometry ##################################################
384
385
386 def StairwellCopy(obj):
387     return applyLast(StairwellRoof, StairwellStory)(arrange( obj, "z", floorHeight, numberFloors ))
388 def StairwellStory(obj):
389     return applySingle(splitStairwellX)(copy(obj)) + applySingle(splitEndsY)(copy(obj)) \
390         + applySingle(splitEndBlocksColsX)(obj)
391     #return applySingle(splitEndsY)(copy(obj)) + applySingle(splitEndBlocksColsX)(obj)
392 def StairwellRoof(obj):
```

```
393     return applySingle(splitStairwellX)(copy(obj)) + applySingle(roofY)(copy(obj)) \
394         + applySingle(splitEndBlocksColsX)(obj)
395     #return applySingle(roofY)(copy(obj)) + applySingle(splitEndBlocksColsX)(obj)
396
397 def CorridorCopy(obj):
398     return applyAll(CorridorStory)(arrange( obj, "z", floorHeight, numberFloors ))
399 def CorridorStory(obj):
400     return applySingle(makeFloorTopper)(copy(obj)) + applySingle(rotateFloorPlanks)(obj)
401     #return applySingle(rotateFloorPlanks)(obj)
402
403 def FlatCopy(obj):
404     return applyAll(FlatStory)(arrange( obj, "z", floorHeight, numberFloors ))
405 def FlatStory(obj):
406     return applySingle(makeFlat)(copy(obj)) + applySingle(rotateFloorPlanks)(obj)
407     #return applySingle(rotateFloorPlanks)(obj)
408
409 def FlatPartyWallCopy(obj):
410     return applyAll(FlatPartyWallStory)(arrange( obj, "z", floorHeight, numberFloors ))
411 def FlatPartyWallStory(obj):
412     return applySingle(rotatePartyWall)(copy(obj)) + applySingle(rotateFloorBeam)(copy(obj)) \
413         + applySingle(makeBlockColumn)(obj)
414     #return applySingle(rotateFloorBeam)(copy(obj)) + applySingle(makeBlockColumn)(obj)
415
416 def BeamCopy(obj):
417     return applyAll(BeamStory)(arrange( obj, "z", floorHeight, numberFloors ))
418 def BeamStory(obj):
419     return applySingle(shiftFloorBeam)(obj)
420     #return applySingle(shiftFloorBeam)(obj)
421
422 def EndWallCopy(obj):
423     return applyAll(EndWallStory)(arrange( obj, "z", floorHeight, numberFloors ))
424 def EndWallStory(obj):
425     return applySingle(rotateEndWalls)(copy(obj)) + applySingle(rotateFloorBeam)(copy(obj)) \
426         + applySingle(splitEndWall)(obj)
427     #return applySingle(rotateFloorBeam)(copy(obj)) + applySingle(splitEndWall)(obj)
428
429 def PartyWallCopy(obj):
430     return applyAll(PartyWallStory)(arrange( obj, "z", floorHeight, numberFloors ))
431 def PartyWallStory(obj):
432     return applySingle(makeBlockPartyWalls)(copy(obj)) + applySingle(rotateFloorBeam)(copy(obj)) \
433         +applySingle(makeBlockPartyCols)(obj)
434     #return applySingle(rotateFloorBeam)(copy(obj)) +applySingle(makeBlockPartyCols)(obj)
435 # These functions are the staging posts for after the main geometry has been laid out.
436
437 ##########################################################################################################################
```

```
438 ######################################################## Interior ########################################################
439 ##########################################################################################################################
440
441 ################################################ Makes the Flats Rooms ###################################################
442 def makeFlat(obj):
443     bounds = getLocalBounds(obj)
444     length = bounds[1][0]-bounds[0][0]
445     relativeDistance = 1-(4.5/length)
446     return applyLast(extWallsRooms, extWallsBalcony) (split( obj, 'x', [relativeDistance], 'srf' ))
447 # This splits the flat into the living room and the other rooms.
448 def extWallsBalcony(obj):
449     return applyThree(exteriorWallBalcony, makeBalcony, corridorWallDoor) (split( obj, 'y', [0.0,1.0], 'srf' ))
450 # This splits the living room into the corridor wall and the exterior wall.
451 def extWallsRooms(obj):
452     return applyThree(exteriorWallWindow, makeRooms, corridorWall) (split( obj, 'y', [0.0,1.0], 'srf' ))
453 # This splits the other rooms into the corridor wall and the exterior wall.
454 def makeRooms(obj):
455     bounds = getLocalBounds(obj)
456     length = bounds[1][1]-bounds[0][1]
457     relativeDistance = 1-(2.5/length)
458     return applyThree(makeFloorTopper, interiorWall, makeRiser) (split( obj, 'y',
459         [relativeDistance, relativeDistance], 'srf' ))
460 # This splits the other rooms into the bedroom and bathroom.
461 def makeRiser(obj):
462     bounds = getLocalBounds(obj)
463     length = bounds[1][0]-bounds[0][0]
464     relativeDistance = 1-(1/length)
465     return applyThree(makeFloorTopper, rotateIntWall, deleteRule) (split( obj, 'x',
466         [relativeDistance, relativeDistance], 'srf' ))
467 # This splits the bathroom to make a riser.
468 def makeBalcony(obj):
469     bounds = getLocalBounds(obj)
470     length = bounds[1][1]-bounds[0][1]
471     relativeDistance = (2/length)
472     return applyThree(intWallsBalcony, balconyWall, intWallsRoom) (split( obj, 'y',
473         [relativeDistance,relativeDistance], 'srf' ))
474 # This splits the living room to make the balcony.
475 def intWallsRoom(obj):
476     return applyLast(interiorWallDoor, makeFloorTopper) (split( obj, 'x', [1.0], 'srf' ))
477 # This makes the interior wall on the living room.
478 def intWallsBalcony(obj):
479     return applyLast(exteriorWallNook, makeFloorTopper) (split( obj, 'x', [1.0], 'srf' ))
480 # This makes that little nook wall for the balcony.
481
482 ################################################ Designates the walls. ###################################################
```

```
def corridorWall(obj):
    return applySingle(replaceDryWalls) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid )))
# Makes the internal corridor wall that has no door.
def corridorWallDoor(obj):
    return applySingle(makeCorridorDoorX) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid) ))
def makeCorridorDoorX(obj):
    bounds = getLocalBounds(obj)
    length = bounds[1][0] - bounds[0][0]
    relativeDistanceA = 1-(1.5/length)
    relativeDistanceB = 1-(2.262/length)
    return applyThree(replaceDryWalls, makeCorridorDoorZ, replaceDryWalls) (split( obj, 'x',
        [relativeDistanceB,relativeDistanceA], 'srf' ))
def makeCorridorDoorZ(obj):
    return applyFirst(deleteRule, replaceDryWalls) (split( obj, 'z', [2.04], 'srf', 'abs' ))
# Makes the internal corridor wall that has A door.
def exteriorWallWindow(obj):
    return applySingle(makeWindowX) (extrudeCurve( obj, (0,0,floorHeight) ))
def makeWindowX(obj):
    return applyAlternate(makeWindowZ, replaceExtWalls) (split( obj, 'x', [1,1.67,2.83,3.5], 'srf', 'abs' ))
def makeWindowZ(obj):
    return applyThree(replaceExtWalls, deleteRule, replaceExtWalls) (split( obj, 'z', [0.19,2.31], 'srf', 'abs' ))
# Makes the windows for the bedroom.
def exteriorWallBalcony(obj):
    return applySingle(makeBalconyRail) (extrudeCurve( obj, (0,0,floorHeight) ))
def makeBalconyRail(obj):
    return applyFirst(deleteRule, replaceExtWalls) (split( obj, 'z', [2.5], 'srf', 'abs' ))
# Makes the wall fronting the balcony floor and designates the rail.
def exteriorWallNook(obj):
    obj = rotate( obj, (0,0,90), moveObj = False  )
    return applySingle(replaceExtWalls) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid) ))
# Makes the peice of wall between the bedroom and balcony.
def rotateIntWall(obj):
    return applySingle(interiorWall) (rotate( obj, (0,0,90), moveObj = False ))
def interiorWall(obj):
    return applySingle(replaceDryWalls) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid) ))
# Standard bit of interior wall
def interiorWallDoor(obj):
    obj = rotate( obj, (0,0,90), moveObj = False  )
    return applySingle(makeIntWallDoorX) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid) ))
def makeIntWallDoorX(obj):
    return applyAlternate(makeIntWallDoorZ, replaceDryWalls) (split( obj, 'x',
        [0.869,1.631,3.369,4.131], 'srf', 'abs' ))
def makeIntWallDoorZ(obj):
    return applyFirst(deleteRule, replaceDryWalls) (split( obj, 'z', [2.04], 'srf', 'abs' ))
# Makes the doors connecting the rooms to the living room.
```

```
528 def balconyWall(obj):
529     return applySingle(balconyWallEnds) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid) ))
530 def balconyWallEnds(obj):
531     bounds = getLocalBounds(obj)
532     length = bounds[1][0] - bounds[0][0]
533     relativeDistanceA = (1/length)
534     relativeDistanceB = 1-(1/length)
535     return applyThree(replaceExtWalls, deleteRule, replaceExtWalls) (split( obj, 'x',
536         [relativeDistanceA,relativeDistanceB], 'srf' ))
537 def makeBlockPartyWalls(obj):
538     bounds = getLocalBounds(obj)
539     relativeDistance = 1/globalWidth
540     return applyThree(rotatePartyWall, deleteRule, rotatePartyWall) (split( obj, 'y',
541         [0.5-relativeDistance, 0.5+relativeDistance], 'crv' ))
542 # Makes the party walls between the blocks.
543 def rotatePartyWall(obj):
544     return applySingle(partyWall) (rotate( obj, (0,0,90), moveObj = False ))
545 def partyWall(obj):
546     return applySingle(replacePartyWalls) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid) ))
547 # Makes the partition walls between flats.
548 def splitStairwellX(obj):
549     return applyFirst(makeLiftsExtWalls, makeLandingExtWalls) (split( obj, 'x', [4], 'srf', 'abs' ))
550 # Splits between corridor and the liftshafts plus stairwells.
551 def makeLandingExtWalls(obj):
552     return applyThree(rotateLandingWall, makeLandingIntWalls, landingWallExtrude) (split( obj, 'y', [0.0,1.0], 'srf'))
553 # This splits the landing to make the exterior walls.
554 def makeLandingIntWalls(obj):
555     return applyLast(rotateLandingWallInt, makeFloorTopper) (split( obj, 'x', [1.0], 'srf'))
556 # This again splits the landing to make the party walls.
557
558 ############################################ Makes the rooms in the landing. ############################################
559 def makeLiftsExtWalls(obj):
560     return applyThree(rotateLiftExtWall, makeLifts, liftExtWallExtrude) (split( obj, 'y', [0.0,1.0], 'srf'))
561 # Splits the lift and stair section to make exterior walls.
562 def makeLifts(obj):
563     return applyFirst(makeStairsIntWallX, makeLiftShaftsY) (split( obj, 'y', [11], 'srf', 'abs' ))
564 # Splits between lifts and stairwells.
565 def makeStairsIntWallX(obj):
566     return applyFirst(makeStairsIntWallY, rotateStairIntWall) (split( obj, 'x', [1.0], 'srf'))
567 # Makes a wall bounding the stairs to the landing
568 def makeStairsIntWallY(obj):
569     return applyFirst(splitStairsX, extrudeStairIntWallB) (split( obj, 'y', [1.0], 'srf'))
570 # Makes a wall seperating the lifts and the stairs.
571 def splitStairsX(obj):
572     return applyFirst(splitStairsY, makeFloorTopper) (split( obj, 'x', [2], 'srf', 'abs' ))
```

```
573 # Makes the landing for the stairs.
574 def splitStairsY(obj):
575     return applyThree(makeFloorTopper, makeStairHoleX, makeFloorTopper) (split( obj, 'y',
576         [2,floorHeight*2+2], 'srf', 'abs' ))
577 # Designtates the area for the actual stairs.
578 def makeStairHoleX(obj):
579     return applyFirst(makeStairHoleY, rotateStairHoleWall) (split( obj, 'x', [1.0], 'srf'))
580 def makeStairHoleY(obj):
581     return applyThree(makeStairHoleWall, replaceStairs, makeStairHoleWall) (split( obj, 'y', [0.0,1.0], 'srf'))
582 # Designates the walls which surround the stair hole in the floor.
583 def makeLiftShaftsY(obj):
584     return applyThree(makeLiftShaftsX, makeLiftFront, makeFloorTopper) (split( obj, 'y', [2.0,2.0], 'srf', 'abs' ))
585 def makeLiftShaftsX(obj):
586     return applyAlternate(rotateLiftSide, deleteRule) (split( obj, 'x', [0.5,0.5,1.0], 'srf'))
587 # Makes the lift shafts.
588
589 ########################################## Designates the walls for the landing. ##########################################
590 def rotateStairHoleWall(obj):
591     return applySingle(makeStairHoleWall) (rotate( obj, (0,0,90), moveObj = False ))
592 def makeStairHoleWall(obj):
593     return applySingle(moveStairHoleWall) (extrudeCurve( obj, (0,0,ceilingVoid) ))
594 def moveStairHoleWall(obj):
595     return applySingle(replacePartyWalls) (move( obj, (0,0,floorHeight-ceilingVoid) ))
596 # Makes the walls for the stair hole
597 def rotateLandingWall(obj):
598     return applySingle(landingWallExtrude) (rotate( obj, (0,0,180), moveObj = False ))
599 def landingWallExtrude(obj):
600     return applySingle(landingWallWindowX) (extrudeCurve( obj, (0,0,floorHeight) ))
601 def landingWallWindowX(obj):
602     bounds = getLocalBounds(obj)
603     length = bounds[1][0]-bounds[0][0]
604     relativeDistance = (0.355/length)
605     return applyThree(replaceExtWalls, makeWindowZ, replaceExtWalls) (split( obj, 'x',
606         [0.5-relativeDistance, 0.5+relativeDistance], 'srf'))
607 # Makes the exterior walls for the landing.
608 def rotateLandingWallInt(obj):
609     return applySingle(landingWallIntExtrude) (rotate( obj, (0,0,90), moveObj = False ))
610 def landingWallIntExtrude(obj):
611     return applySingle(splitLandingWallInt) (extrudeCurve( obj, (0,0,floorHeight-ceilingVoid) ))
612 def splitLandingWallInt(obj):
613     bounds = getLocalBounds(obj)
614     length = bounds[1][0]-bounds[0][0]
615     relativeDistance = (1/length)
616     return applyThree(replacePartyWalls, deleteRule, replacePartyWalls) (split( obj, 'x',
617         [0.5-relativeDistance, 0.5+relativeDistance], 'srf'))
```

```
# Makes the end party walls.
def rotateLiftExtWall(obj):
    return applySingle(liftExtWallExtrude) (rotate( obj, (0,0,180), moveObj = False ))
def liftExtWallExtrude(obj):
    return applySingle(replaceExtWalls) (extrudeCurve( obj, (0,0,floorHeight) ))
# Makes the external walls fo the lift and stairs area.
def rotateStairIntWall(obj):
    return applySingle(extrudeStairIntWall) (rotate( obj, (0,0,90), moveObj = False ))
def extrudeStairIntWall(obj):
    return applySingle(splitStairIntWallX) (extrudeCurve( obj, (0,0,floorHeight) ))
def splitStairIntWallX(obj):
    return applyThree(replacePartyWalls, splitStairIntWallZ, replacePartyWalls) (split( obj, 'x',
        [1, 2.8], 'srf', 'abs'))
def splitStairIntWallZ(obj):
    return applyFirst(deleteRule, replacePartyWalls) (split( obj, 'z', [2.04], 'srf', 'abs' ))
# Makes the wall sperating the landing and stairs.
def extrudeStairIntWallB(obj):
    return applySingle(replacePartyWalls) (extrudeCurve( obj, (0,0,floorHeight) ))
# Makes the wall between the stairwell and lifts.
def makeLiftFront(obj):
    return applySingle(splitLiftFrontX) (extrudeCurve( obj, (0,0,floorHeight) ))
def splitLiftFrontX(obj):
    return applyAlternate(splitLiftFrontY, replacePartyWalls) (split( obj, 'x', [0.5,1.5,2.5,3.5], 'srf', 'abs' ))
def splitLiftFrontY(obj):
    return applyLast(replacePartyWalls, deleteRule) (split( obj, 'z', [2.1], 'srf', 'abs' ))
# Makes the front walls of the lifts including doors.
def rotateLiftSide(obj):
    return applySingle(makeLiftSide) (rotate( obj, (0,0,90), moveObj = False ))
def makeLiftSide(obj):
    return applySingle(replacePartyWalls) (extrudeCurve( obj, (0,0,floorHeight) ))
# Makes the sides of the lift.
def rotateEndWalls(obj):
    return applySingle(extrudeEndWalls) (rotate( obj, (0,0,90), moveObj = False ))
def extrudeEndWalls(obj):
    return applySingle(replaceExtWalls) (extrudeCurve( obj, (0,0,floorHeight) ))
# Makes the walls at the end of the structure.

########################################## Replaces the 2D walls with 3D ones ##########################################
def replaceDryWalls(obj):
    #return [obj]
    return metSecWall(obj, spacing = 1.0, panelA = "dry", panelB = "dry")
# Puts dry walls over all of the 2d surfaces marked for internal walls
def replacePartyWalls(obj):
    #return [obj]
    return metSecWall(obj, spacing = 1.0, panelA = "party", panelB = "party")
```

```
# Puts in party walls.
def replaceExtWalls(obj):
    #return [obj]
    return metSecWall(obj, spacing = 1.0, panelA = "party", panelB = "party")
# Puts in external walls.
def makeFloorTopper(obj):
    #return [obj]
    return [extrude( obj, (0,0,0.03) )]
# Makes a scree topper for the floors.
def replaceStairs(obj):
    #return [obj]
    basis = obj.basis
    delete(obj)
    stair = makeStair(floorHeight)
    stair = move(stair, (0,-floorHeight,0), moveBasis = False)
    return [transform(stair, basis)]
# Makes the stairs.

##########################################################################################################################
########################################################## Floors ########################################################
##########################################################################################################################

def splitEndsY(obj):
    return applyThree(shiftFloorBeam, splitEndsX, shiftFloorBeam) (split( obj, 'y', [0.0,1.0], 'srf' ))
# Makes the external beams for the stairwells.
def splitEndsX(obj):
    return applyLast(rotateFloorBeam, splitLift) (split( obj, 'x', [1.0], 'srf' ))
# Makes the internal beam at the stairwells.
def splitLift(obj):
    return applyThree(splitStair, liftTBeamY, shiftFloorPlanks) (split( obj, 'y',
        [globalWidth-4.0,globalWidth-2.0], 'srf', 'abs' ))
def liftTBeamY(obj):
    return applyThree(shiftFloorBeam, liftTBeamX, shiftFloorBeam) (split( obj, 'y', [0.0,1.0], 'srf' ))
def liftTBeamX(obj):
    return applyThree(deleteRule, rotateFloorBeam, shiftFloorPlanks) (split( obj, 'x', [4.0,4.0], 'srf', 'abs' ))
# Makes the transfer beams for the lifts.
def splitStair(obj):
    return applyThree(shiftFloorPlanks, stairTBeamY, shiftFloorPlanks) (split( obj, 'y',
        [2,floorHeight*2+2], 'srf', 'abs' ))
def stairTBeamY(obj):
    return applyThree(shiftFloorBeam, stairTBeamX, shiftFloorBeam) (split( obj, 'y', [0.0,1.0], 'srf' ))
def stairTBeamX(obj):
    return applyThree(deleteRule, rotateFloorBeam, shiftFloorPlanks) (split( obj, 'x', [2.0,2.0], 'srf', 'abs' ))
# Makes the transfer beams forr the stairs.
def rotateFloorBeam(obj):
```

```
    return applySingle(shiftFloorBeam) (rotate(obj, (0,0,90), moveObj = False))
def shiftFloorBeam(obj):
    return applySingle(makeFloorBeam) (move( obj, (0,0,-0.3525+floorHeight) ))
def makeFloorBeam(obj):
    #return [obj]
    return makeBeam(obj)
# Replaces the lines with actual beams
def rotateFloorPlanks(obj):
    basis = obj.basis
    inverseBasis = t.invertMat(basis)
    matrix = t.multiply(basis, t.multiply(t.rotateMat((0,0,90)), inverseBasis))
    return applySingle(shiftFloorPlanks) (transform(obj, matrix, moveObj = False))
def shiftFloorPlanks(obj):
    return applySingle(copyFloors) (move( obj, (0,0,-0.2+floorHeight) ))
def copyFloors(obj):
    return applyFirst(makeFloorPlanks, shiftCeiling) ([obj, copy(obj)])
def makeFloorPlanks(obj):
    #return [obj]
    return hollowCoreFloor(obj)
# Makes the floorPlanks.
def shiftCeiling(obj):
    return applySingle(makeCeiling) (move( obj, (0,0,0.2-ceilingVoid) ))
def makeCeiling(obj):
    return applyFirst(makePlaster, shiftInsulation) ([obj, copy(obj)])
def makePlaster(obj):
    return [extrude( obj, (0,0,0.015) )]
def shiftInsulation(obj):
    return applySingle(makeInsulation) (move( obj, (0,0,0.015) ))
def makeInsulation(obj):
    return [extrude( obj, (0,0,0.2) )]
# Makes the ceiling

def roofY(obj):
    return applyThree(shiftFloorBeam, roofX, shiftFloorBeam) (split( obj, 'y', [0.0,1.0], 'srf' ))
# Makes the external beams for the stairwells.
def roofX(obj):
    return applyFirst(rotateFloorPlanks, rotateFloorBeam) (split( obj, 'x', [1.0], 'srf' ))
# Makes the internal beam at the stairwells.

##########################################################################################################################
########################################################## Structure #####################################################
##########################################################################################################################

def makeBlockPartyCols(obj):
    bounds = getLocalBounds(obj)
```

```
753     relativeDistance = 1/globalWidth
754     return applyThree(makeBlockColumn, deleteRule, makeBlockColumn) (split( obj, 'y',
755         [0.5-relativeDistance, 0.5+relativeDistance], 'crv' ))
756 # Makes the party walls between the blocks.
757 def makeBlockColumn(obj):
758     return applyThree(shiftColumn, deleteRule, shiftColumn) (split( obj, 'y', [0.0,1.0], 'crv' ))
759 # makes the colums for each of the blocks.
760 def splitEndBlocksColsX(obj):
761     bounds = getLocalBounds(obj)
762     length = bounds[1][0]-bounds[0][0]
763     relativeDistance = 4.0/length
764     return applyThree(splitEndBlocksColsY, deleteRule, makeBlockPartyCols) (split( obj, 'x',
765         [relativeDistance, 1.0], 'srf'))
766 # Makes the internal wall for the bracing.
767 def splitEndBlocksColsY(obj):
768     return applyThree(extrudeBracing, deleteRule, extrudeBracing) (split( obj, 'y', [0.0, 1.0], 'srf'))
769 def splitEndWall(obj):
770     bounds = getLocalBounds(obj)
771     relativeDistance = 1/globalWidth
772     return applyThree(splitEndSeg, deleteRule, splitEndSeg) (split( obj, 'y',
773         [0.5-relativeDistance, 0.5+relativeDistance], 'crv' ))
774 def splitEndSeg(obj):
775     return applyThree(shiftColumn, rotateBracing, shiftColumn) (split( obj, 'y', [0.0,1.0], 'crv' ))
776 # makes the colums for each of the blocks.
777 def rotateBracing(obj):
778     return applySingle(extrudeBracing) (rotate( obj, (0,0,90), moveObj = False ))
779 def extrudeBracing(obj):
780     return applySingle(shiftBracing) (extrudeCurve( obj, (0,0,floorHeight-0.3525) ))
781 def shiftColumn(obj):
782     return applySingle(replaceColumn) (move(obj, (0,0,0) ))
783 def replaceColumn(obj):
784     #return [obj]
785     return [makeColumn(obj, floorHeight)]
786 # Replaces with actual columns
787 def shiftBracing(obj):
788     return applySingle(replaceBracing) (move(obj, (0,0,0) ))
789 def replaceBracing(obj):
790     #return [obj]
791     return [makeBracing(obj)]
792
793 ##########################################################################################################################
794 ############################################################ Setup ########################################################
795 ##########################################################################################################################
796
797 floorHeight = 3.5
```

```
# Global floorHeight variable.  Note that floor to ceiling height is 2.5m
ceilingVoid = 1
globalWidth = 15
globalFlatArea = 50.0
numberFloors = 9

##########################################################################################################################
########################################################## Startup ########################################################
##########################################################################################################################



def generate(objs):
    return EndWallCopy(objs[0]) + makeStairwellA(objs[1]) + applyAlternate(splitBlock, PartyWallCopy)(objs[2:-2]) \
        + makeStairwellB(objs[-2]) + EndWallCopy(objs[-1])

floorPlan = offsetLine( [(0.0,0.0,0.0),(15.0*(3.0**0.5),15.0,0.0),(30.0*(3.0**0.5),0.0,0.0),
(45.0*(3.0**0.5),15.0,0.0),(60.0*(3**0.5),0.0,0.0)], -globalWidth )

generate(floorPlan)
```