UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Mathias Plans

Shape Grammar Editor

Master's Thesis (30 ECTS)

Supervisor: Jaanus Jaggo, MSc

Tartu 2023

Shape Grammar Editor

Abstract:

This thesis presents a new way to generate procedural meshes in Godot games. Two new tools were developed. First, the shape grammar editor that can be used to define parameters for a shape grammar. The grammar rules are defined on a visual representation of the shapes, which makes the designing process of the grammar intuitive. Then, that shape grammar is used in Godot plugin to create meshes in real-time applications. Both tools are built on newly devised algorithms and data structures that handle the topology and geometry of 3D shapes. It is shown that the tools given in the grammar editor are enough to create variable shaped objects.

Keywords:

Procedural generation, shape grammar, formal grammar, generative grammar, CAD, Godot

CERCS: P170, computer science, numerical analysis, systems, control

Kujugrammatika Redigeerija

Lühikokkuvõte:

Lõputöö kirjeldab uut meetodit, mida saab kasutada protseduurilise generatsiooni tarbeks Godot mängudes. Arendati kaks uut tööriista. Esimene tööriist on kujugrammatika redigeerija, mida saab kasutada kujugrammatika parameeter defineerimiseks. Grammatika reeglid defineeritakse visuaalse kasutajaliidese abil, mis teeb selle protsessi intuitiivseks. Teine tööriist on Godot lisaprogramm, mis võimaldab kasutaja defineeritud grammatikat kasutada reaalajalistes rakendustes. Mõlemad tööriistad põhinevad uutel algoritmidel ja andmestruktuuridel, mis käsitlevad kolmemõõtmeliste kujude topoloogiat ja geomeetriat. Lõputöö näitab, et antud tööriistad on piisavad, et luua varieeruva kujuga objekte.

Võtmesõnad:

Protseduuriline generatsioon, kujugrammatika formaalne grammatika, generatiivne grammatika, raalprojekteerimine, Godot

CERCS: P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaat-juhtimisteooria)

Table of Contents

1 Introduction 4
2 Procedural Generation Methods 6
2.1 Formal Grammar 6
2.2 Shape Grammar 7
2.3 Split Grammar 12
3 Design and Implementation 14
3.1 Editor 16
3.1.1 Boundary Representation 17
3.1.2 Winged-edge Data Structure 18
3.1.3 Shape and Symbol 20
3.1.4 Rule 21
3.1.5 Anchor 25
3.1.6 Cut 30
3.1.7 Cutting Methods 32
3.1.8 Persistence 35
3.1.9 Compiler 36
3.1.10 Tester 38
3.2 Plugin 39
3.2.1 Interpreter 39
3.2.2 GrammarState 40
3.2.3 GrammarMesh 40
3.2.4 GrammarInstance3D 41
4 Results 42
4.1 Tasks 42
4.2 System Usability Scale 44
4.3 Demonstration 45
4.4 Future Improvements 46
5 Conclusion 48
References 49
Appendix 50
I. Plugin User Manual 50
II. Accompanying files 52
III. License 53

1 Introduction

3D video games require many models to populate their virtual worlds. For example, a city-building game must have hundreds of building models to look alive. Having only one model of a building would not make it possible for the player to express their full creativity. For a city-building game to be more interesting, multiple models of buildings are provided. Adding more models, however, would increase the game's development budget. Creating art assets is a time-consuming and expensive process[1]. Therefore, developing a city-building video game that allows players to build complex cities is not cheap.

A popular city-building game, Cities: Skylines, has offloaded the responsibility of creating 3D models to the community by providing modding support for the game. This approach has worked well for them; it is the most-played city-building game on Steam. The game has thousands of player-made assets, providing a lot of variety.

On the other hand, the high number of assets poses a memory problem. Using over 800 assets requires the user to have 16GB of RAM for optimal performance[2], although if performance is not a concern, a lot more assets can be loaded[3]. Storing these buildings on disk takes a lot of space as well. Either way, memory is a bottleneck when using many assets.

Developers have used procedural generation to reduce the number of assets while keeping the diversity. The simplest way is to take one asset and change some property of it. A 3D model can be coloured, effectively providing the player with the same number of assets as there are colours. However, since the underlying assets are the same, the method generates assets that are very similar to one another. This becomes especially apparent when these assets are placed next to each other, which often happens in city-building games. More advanced procedural generation techniques must be used to generate a wider variety of buildings.

A popular solution for the generation of architecture is to use shape grammar. The rule-based replacement of shapes lends itself to regular structures such as buildings. One type of shape grammar in particular, split grammar, is very intuitive and easy to implement. In split grammar, the users are restricted in what kind of rules they can define. It is like sculpting;

[1] https://www.zippia.com/3d-artist-jobs/salary
[2] https://steamcommunity.com/sharedfiles/filedetails/?id=1411897315
[3] https://steamcommunity.com/app/255710/discussions/0/4731597528368146680/

the material is removed until an artwork remains. Any shape can be generated if the initial shape is large enough.

While some tools can use shape grammar to generate buildings, they can only be used to generate pre-built assets. None offer the possibility to generate the shapes during the run-time of a game. The goal of this thesis is to develop tools that allow intuitive creation of 3D assets using shape grammar in Godot games.

This thesis develops two new tools. The first tool is an editor for shape grammar. This editor is used to create and modify rules of split grammar. The editor must (a) be intuitive to solve the problem of reducing the development costs of 3D assets, and (b) offer a variety of tools that allow users to create as many possible architectures as possible. The second tool is a plugin which makes it possible to use the created rules to generate meshes in Godot 4 games.

This thesis:

- gives an overview of how shape grammar works and how it is used to procedurally generate architecture;
- describes the design and implementation of the tools, including what algorithms were used, and what is the software architecture of the tools;
- discusses how well the tools fulfil the goals. Some practical use cases are presented, and the intuitiveness of the editor is tested.

In chapter 2 of this thesis, definitions for formal grammar, shape grammar, and split grammar are detailed. In chapter 3, the existing work is briefly discussed, and the editor's and plugin's design and implementation are presented. In chapter 4, the results of the thesis are analysed and discussed. It is followed by the conclusion in chapter 5. Finally, the appendix contains a tutorial for the plugin and information about the accompanying files.

An AI tool Grammarly was used during the writing of this thesis. It was used to fix grammatical errors and wording in this thesis. No ideas or paragraphs were generated with AI tools.

2 Procedural Generation Methods

Procedural content generation is the algorithmic creation of game content with limited or indirect user input [1]. This thesis uses shape grammar as the procedural generation method. Shape grammar is a type of formal grammar. This chapter defines formal grammar, describes the concept of shape grammar, and introduces its derivative, split grammar.

2.1 Formal Grammar

A few preliminary definitions are necessary before formal grammar can be defined. Grammar is a concept in linguistics. We call formal language a set of words or sequences of symbols. Traditionally, the words are strings, but they can also be graphs or shapes. Every word in a formal language is made of letters. The set of all the letters is called the alphabet. For example, the English language as a formal language requires the alphabet to have all the English letters. Another famous alphabet is the set consisting of zero and one. The language of this alphabet is a subset of all binary numbers. With the definitions of formal language and the alphabet, formal grammar can be defined.

Formal grammar is a set of rules which determine how to construct valid words of a formal language from its alphabet. These rules are called *production rules* or *replacement rules*. A production rule takes a sequence of symbols and replaces them with another sequence of symbols. *Context-free grammar* only allows rules that replace one symbol at a time. The rules are written as A→B, where A (the *left-hand side* or *LHS*) is the sequence of symbols replaced with the sequence of symbols B (the *right-hand side* or *RHS*). Symbols that do not have any replacement rules are called *terminal symbols*. Grammar is *deterministic* if all non-terminal symbols have only one replacement rule. For non-deterministic grammar, a rule can be selected randomly (optionally with weights).

Formal grammar also has a *starting symbol*. This symbol is from the set of nonterminal symbols of the formal language. During the *interpretation* of the grammar, the starting symbol is the first symbol written down. Then the symbols are replaced until only terminal symbols remain or a certain number of replacements have been done.

The order of replacements can determine the outcome of the grammar. With *sequential* replacement order, symbols are replaced one at a time. The interpreter iterates through the written symbols, and when a non-terminal symbol is detected, it is replaced according to the appropriate grammar rule. After that, the interpreter can continue where it stopped or restart

from the beginning. With *parallel* replacement, all the non-terminal symbols are replaced at the same time. In a sense, the replacement is done one generation at a time. Interpreters that use *parallel* replacement order can limit the procedural generation to a fixed number of generations. Sequential and parallel replacement orders may produce different results; some use cases prefer one.

Shaker [2] presents different use cases for procedural generation with formal grammar in games. One of the most popular methods is the Lindenmayer system (L-system). It can be used to generate vegetation or other fractal-like objects. Lindenmayer systems are formal grammars that use parallel replacement order. L-systems also have an additional mechanism to convert words (that are in text form) into geometric structures. Due to its fractal nature, L-systems are most often used to generate vegetation [2]. An example tree is given in Figure 1. The grammar system implemented in this thesis has some similarities with L-systems. More powerful grammars for architecture generation are built on shapes.

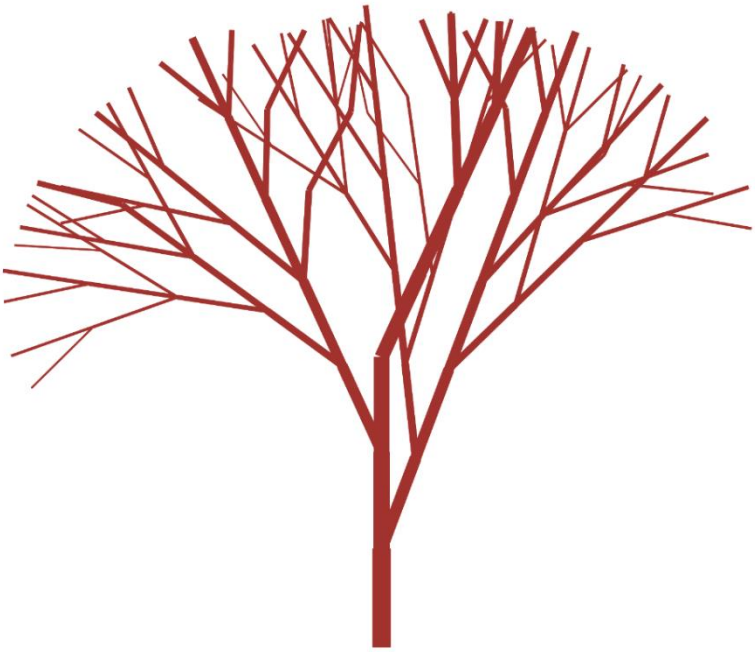

Figure 1. A tree generated with an L-system.

2.2 Shape Grammar

Shape grammar was first formalized by Stiny [3]. He defined a *shape* to be a finite collection of non-collinear lines. Intuitively, lines are *collinear* if they can be merged to form a single line. This is illustrated in Figure 2. In the figure, there are three pairs of lines. The pairs marked with red crosses cannot be merged into one line, so they are not collinear. The pair

marked with the green checkmark can be merged into a single line, which is drawn after the arrow. Therefore, that pair of lines are collinear.

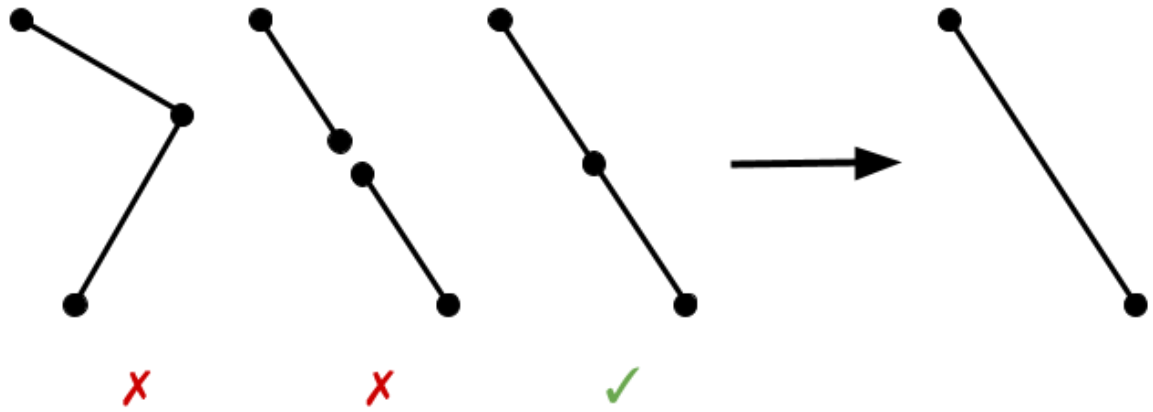

Figure 2. Line collinearity.

The shape only has non-collinear lines. Thus, no two lines in the shape can be merged. This makes the shapes unique; each geometrical shape can be represented by only one possible set of lines. Stiny [3] also demonstrated that Boolean operations could be defined on shapes. One can take a union, intersection, or difference between two shapes. With a union, some lines in the set might merge to form longer lines. A shape is a sub-shape of another shape if all the lines in the first shape are sub-lines of lines in the other. This is demonstrated in Figure 3, where the shape in the left is a sub-shape of the shape in the right. All geometrical shapes that contain only straight lines can be represented with these definitions [3].

Figure 3. Sub-shape example.

Shapes can have labels associated with them. One shape may have multiple labels in the form of a set of labeled points. Each point can be associated with a line, making that line distinguishable from other lines. Stiny also refers to these points as symbols [3]. The labels are demonstrated in Figure 4. In the figure, the shape has been assigned two different labeled points: circles and squares. A line can have more than one labeled point on it. Labeling the

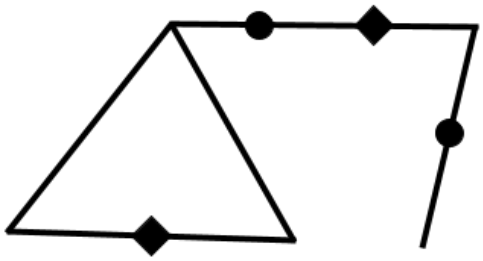

Figure 4. Labels on lines.

shape in a certain way makes it chiral; it cannot be mapped to its own rotation or translation. This is useful when defining the replacement rules for shape grammar.

The coordinates of line endpoints can be variable. This makes the shape *parametric*. Parameters are assigned at the start of the interpretation and then *propagated* by the grammar rules. The label coordinates can also be parametric [3].

Transformations can be applied to the shapes. Two shapes are like each other when one is a transformation of another. The possible transformations are translation, scale, rotation, reflection, or any composition of them [3]. The transformations are shown in Figure 5.

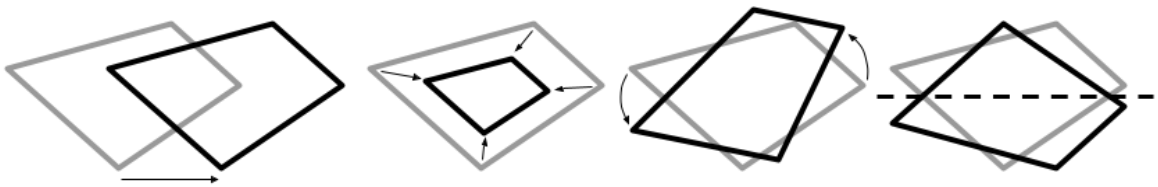

Figure 5. Translation, scale, rotation, and reflection.

The rules of shape grammar are like the rules in formal grammar. On the left, a shape to be replaced is specified. On the right, a replacement shape is given. A simple shape grammar rule is presented in Figure 6, where a cross shape is replaced by a diagonal line. The replacement shape can also be an empty shape. If the shapes are parametric, the parameter propagation is also specified.

Figure 6. Simple grammar rule.

During the interpretation of the grammar, an *initial shape* is first drawn. A rule can be used on the drawn shape if some transformation of the left-hand side (LHS) of the rule is a sub-shape of the drawn shape. When the rule is applied, the transformed LHS of the rule is removed from the drawn shape, and the right-hand side (RHS) of the rule with the same transformation is drawn [3].

The interpretation of the grammar is shown in Figure 7. In the figure, there are four shapes. These shapes form a sequence of generations of the grammar when using the rule in Figure 6. The first shape is the initial shape. The blue lines show how the LHS of the rule is a sub-shape of the first shape. That section of the shape is then replaced by the diagonal line, which crosses the other two lines, as seen in the second shape. Now there are two places where the rule can be used, highlighted with blue lines in the second and third shapes. These places are also replaced by the diagonal lines. Therefore, the resulting shape is two diagonal lines, which is the fourth shape in the figure.

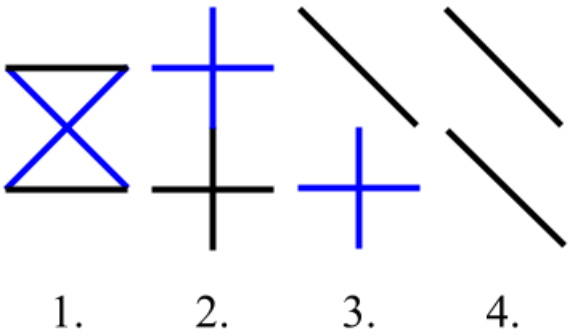

Figure 7. Interpretation of a grammar.

One shape can be a sub-shape of the drawn shape in many ways, a few of which could intersect with each other. This means that shape grammar rules must be used in sequential order. As an example, in Figure 8, the rule of Figure 6 can be applied in two different places. By using the rule in one of the possible places, the other one becomes unusable. Thus, the parallel rule order cannot be used.

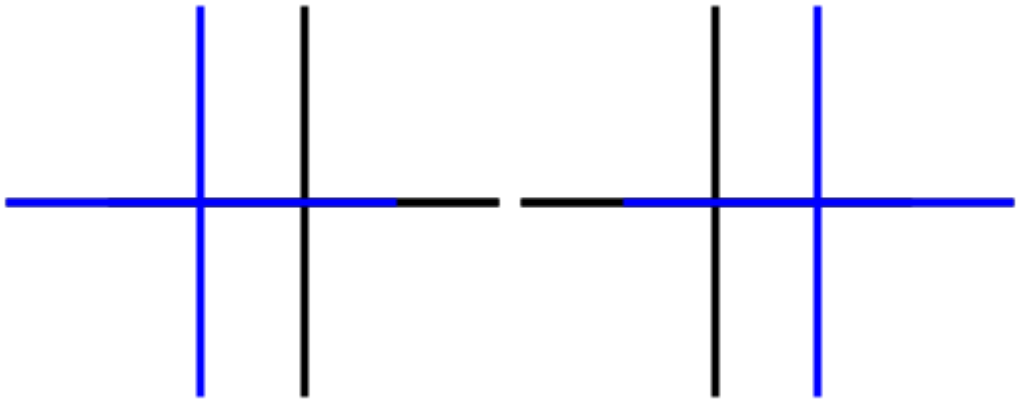
Figure 8. The rule can be used at two places.

A more complex example of shape grammar is given in Figure 9. On the top of the figure, the rule is defined. The LHS of the rule is a triangle with one of the lines marked with a symbol. The RHS of the rule are two triangles, one in another. The symbol is put on one of the inner triangle edges. Below the rule is a line of possible productions. Shapes 1 and 2 are the initial shape and the first generation of the interpretation. The next generation has two possible shapes. Shape 3.1 is the result when the middle triangle of shape 2 was replaced. Shape 3.2 is the result when the bottom-left triangle of shape 2 was replaced. Both the middle and the bottom-left triangles of the shape 2 are candidates for replacement because they are a transformed version of the LHS of the rule.

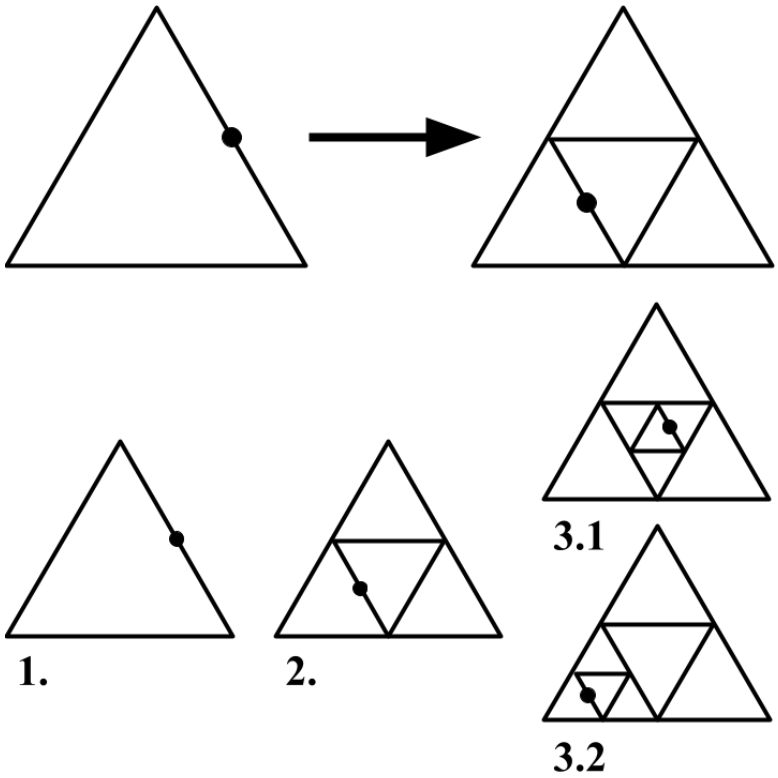

Figure 9. Shape grammar rule and its generations.

Embedding, the job of detecting transformed sub-shapes of the drawn shape, is non-trivial. Creating a real shape-grammar interpreter is difficult. Only a few of them exist. Hong and Economou [4] show that only few implementations are not restricted to certain types of shapes, transformations, or complexity of geometry [4]. Due to the difficulty of embedding, the shape-grammar interpreters are not performant and therefore do not meet the criteria for real-time applications. Restricting the grammar rules and possible shapes can make the embedding process easier. This thesis uses split grammar, one of the restricted derivations of shape grammar.

2.3 Split Grammar

Wonka and Wimmer [5] introduced split grammar, a restricted form of shape grammar. The kinds of shapes used in split grammar are limited to surfaces in the case of 2D and solids in the case of 3D. Unlike in Stiny's shape grammar, the labels are not free-standing points. Instead, the shapes themselves have symbols. This is shown in Figure 10, where two example shapes are given. The cube shape has been assigned the symbol "A".

The rule application is made according to the symbol instead of the topology of the shape. Thus, the embedding is trivialized by searching for symbols from a database. The possible designs of split grammar are more limited compared to Stiny's shape grammar. This is because lines are not primitives. However, split grammar shapes are sufficient for the procedural generation of architecture [5].

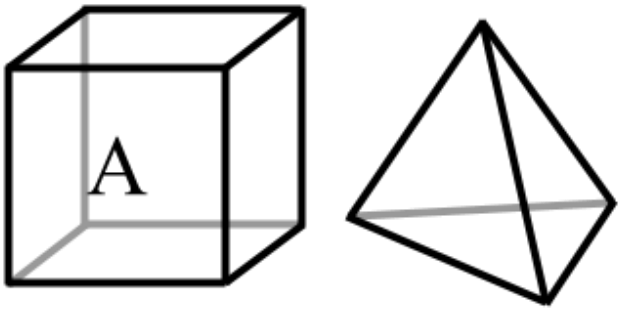

Figure 10. Legal shapes in split grammar.

Two kinds of rules can be defined for split grammar: split rule, which splits a shape into multiple non-intersecting sub-shapes by introducing one or more cuts, and replacement rule, which replaces a shape with another. The replacement rule must be defined so that the replacement shape fits into the replaced shape. Both kinds of rules generate shapes that are bounded by the original shape [5]. The benefit of this restriction on the rules is that it is impossible to create a grammar that "grows shapes into itself." The phenomenon is illustrated in Figure 11, where the growth into itself is highlighted with the red circle.

Split grammar allows the rule application order to be either sequential or parallel. Since replacements do not interfere with surrounding shapes, parallel rule order can be used. Using the parallel rule order makes split grammars similar to L-systems. As will be seen in Design and Implementation chapter, the cuts can be represented with tree structure, and L-systems are good at generating trees. Therefore, the intuitiveness of L-systems transfers to split grammars as well.

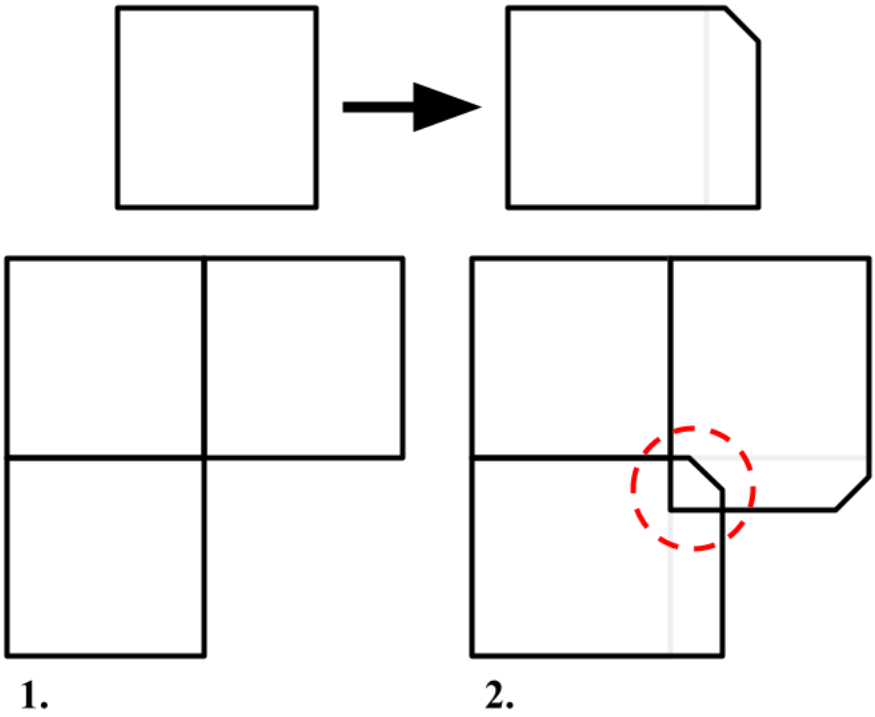

Figure 11. Overlapping growth.

Split grammar is by default parametric. Wonka and Wimmer [5] also introduced two systems for parameter propagation and attribute matching. Their parameter propagation is carried out with control grammar. This is a regular formal grammar where the terminal symbols are assignments of attributes to the shapes. Attribute matching is a system that gives the user a better control over the rule selection process [5]. However, these systems do not fall into the scope of this thesis. Fortunately, split grammar by itself is powerful enough to generate a wide variety of houses [5]. With the definitions for formal grammar, shape grammar, and split grammar, the details of the tool implementation can be described.

3 Design and Implementation

Defining rules for a shape grammar is not a trivial activity. Previous implementation of this topic by the author [6] lets users implement the rules as functions in C# code. A rule function takes a shape as one of the arguments and returns the replacement shapes. This approach is the easiest to implement. However, the handling of the topology is left to the user. For every rule, the connections between vertices of output shapes must be established manually [6]. Human minds are capable of spatial thinking, but complex shape grammar rules require complex spatial thought. As a result, the learning curve of such code-based systems is high. It also takes more time to create visual products by writing code which leads to low productivity.

Like formal grammar on text symbols, the shape grammar rules can be represented with text in the form of production rules. Müller et al. [7] introduced a system where the rules are specified on the symbols, not shapes. These rules are translated to operations on shapes during the compilation of the grammar. This method also offers users spatial transformation functions, such as split, copy, translation, and rotation. A small example grammar is given for building layout in Figure 12. This is the most popular way to define shape grammar rules [7]. Esri CityEngine[4], a commercial tool for creating virtual cities, uses this approach.

```
1:  lot ⇝ S(1r,building_height,1r)
      Subdiv("Z",Scope.sz*rand(0.3,0.5),1r){ facades | sidewings }
2:  sidewings ⇝
      Subdiv("X",Scope.sx*rand(0.2,0.6),1r){ sidewing | ε }
      Subdiv("X",1r,Scope.sx*rand(0.2,0.6)){ ε | sidewing }
3:  sidewing
      ⇝ S(1r,1r,Scope.sz*rand(0.4,1.0)) facades : 0.5
      ⇝ S(1r,Scope.sy*rand(0.2,0.9),Scope.sz*rand(0.4,1.0))
        facades : 0.3
      ⇝ ε : 0.2
4:  facades ⇝ Comp("sidefaces"){ facade }
```

Figure 12. Gramma in text form (Müller, 2016).

The implementations by Plans [6] and Müller [7] are both text-based, which may hinder users who are not familiar with writing software. A more intuitive way to define shape grammar rules is to use a 3D representation in the style of Computer-Aided Design (CAD)

[4] https://www.esri.com

software. Figure 13 shows how cutting by plane is done in Fusion 360[5] and how the same operation is done in the Esri CityEngine. While the code is useful when the exact numbers are important, the 3D representation is much more intuitive. Displaying a 3D render of the rule on a screen lets artists focus on the shape and design while the software handles topology transformations. Therefore, one of the software components of the Shape Grammar Editor is a 3D editor. Users can edit 3D representations of split grammar rules by utilizing different cutting methods.

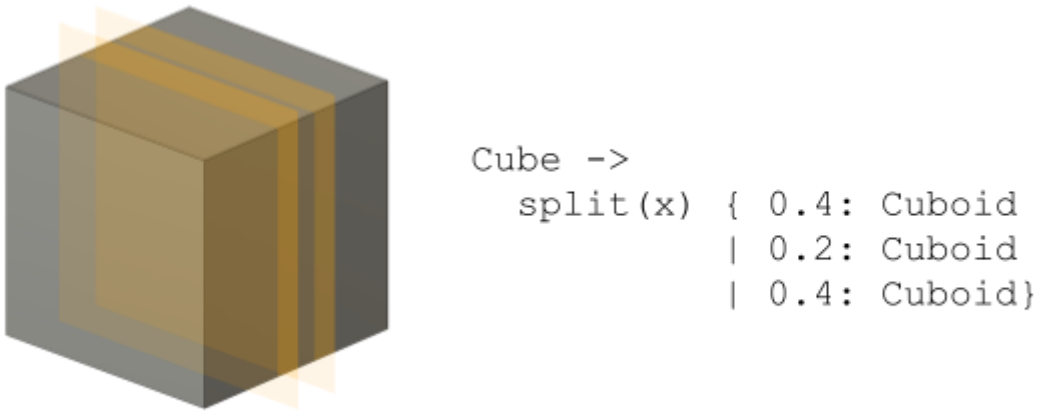

Figure 13. 3D modelling verses code.

The second component of the Shape Grammar Editor is a game engine plugin. This plugin takes the defined grammar as input and generates grammar words as the output. Although the intended use case of this plugin is to generate buildings in games, it can be potentially ported to any platform that might require it. For example, a Blender plugin would be useful for generating shapes and then manually sculpting on top of them. For this thesis, the plugin was implemented for the Godot 4 game engine.

Godot 4 was chosen as the implementation platform for the components. The choice was made for multiple reasons. Firstly, Godot is an open-source game engine that has been proven useful for implementing non-game applications. Secondly, it supports many platforms, including Windows, Linux, and the Web. Thirdly, the game engine has a custom Python-like scripting language, which makes it simple to prototype and implement the necessary algorithms. Fourthly, Godot provides multiple built-in geometry functions, mostly related to ray and polygon intersections. And finally, integrating multiple software

[5] https://www.autodesk.com/products/fusion-360

components can be simply done by creating addons and uploading them to Godot Asset Library. The asset library made it simple to integrate the plugin into the editor.

3.1 Editor

The editor is the only component with a graphical user interface (UI). The editors interface layout is presented in Figure 14. The edit view is in the centre of the screen, indicated by number 1 in the figure. In this section, the user can see the visualizations of symbols and rules and use cutting tools to modify the shape. Each shape in this view is assigned a colour to make them distinguishable from each other.

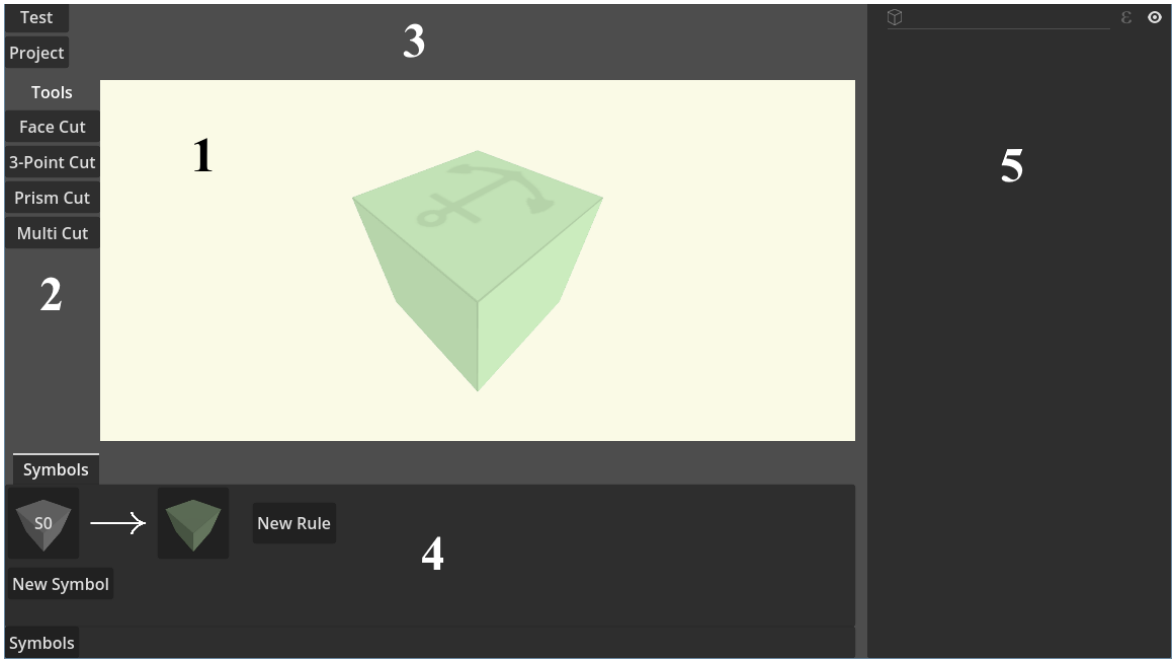

Figure 14. Editor UI layout.

The cutting tools are available from the *toolbar* in area 2. The toolbar is not always visible, it will only appear when the subject shape is in the edit view. Some tools are configurable by users. This is done via the *tool info* section in area 3. This UI component's contents change depending on the selected tool. The tools and their settings are detailed in chapter 3.1.7 Cutting Methods.

The user can see all the grammar symbols and rules in a *symbol explorer*, located at the bottom of the window in area 4. The symbol explorer is used to also create new symbols and new rules. The symbol explorer is explained in chapters 3.1.3 Shape and Symbol and 3.1.4 Rule.

During the definition of rules, the cuts on the shape form a hierarchical structure. This structure is illustrated in a *tree view* in area 5. Tree view is also used to assign symbols to sub-shapes and control the visibility in the edit view. The tree view is detailed in chapter 3.1.4 Rule. Finally, the grammar can be tested and saved with buttons in the top left corner, above the toolbar. The compilation process is described in chapter 3.1.9 Compiler and saving in chapter 3.1.8 Persistence.

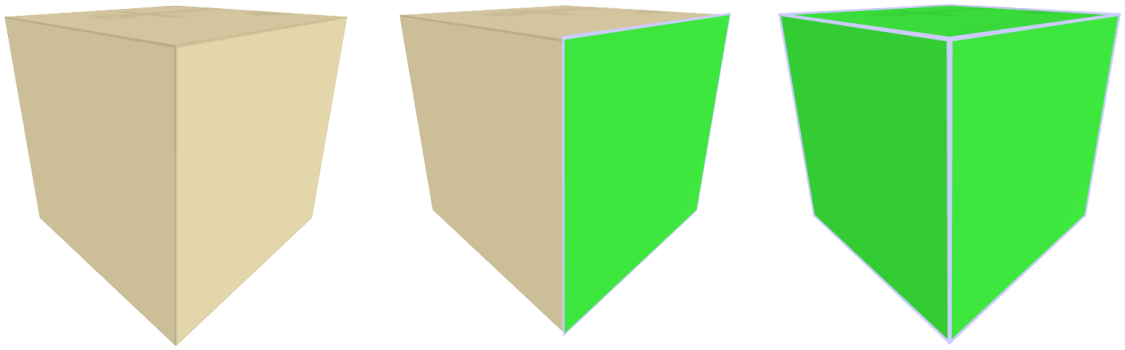

Figure 15. Selection modes.

Most of the user interaction with the editor will be carried out in the edit view. This is where the sculpting part of the grammar definition happens. In the center of the edit view, a symbol or a rule is visualized. This visualization can be rotated by dragging the mouse while holding down the left mouse button (LMB). Some operations require the user to specify the object they want to affect. Faces and shapes are selectable by clicking on them with LMB in the edit view. The selections of face and shape are cycled. By clicking on an unselected face, the face is selected. This is illustrated in Figure 15 with the middle shape. Clicking on that face again selects the shape that the face is part of. In the figure this is the third shape. Clicking that face a third time or pressing “Esc” key unselects the shape. The unselected shape is the first shape in Figure 15. While the visualization in the edit view is a 3D mesh, operations are done on a different shape representation.

3.1.1 Boundary Representation

Boundary representation is a method for representing volumes by partitioning the inside and outside of the volume by a boundary. The boundary can be thought of as the skin of the volume. The skin can be constructed with flat surfaces such as triangles and quads, but also with curves by using splines as the edges. The triangle mesh is an example of boundary representation. The skin of the volume is constructed by stitching together multiple

triangles. Triangle mesh is a widely used boundary representation; most popular graphics libraries are built on triangles.

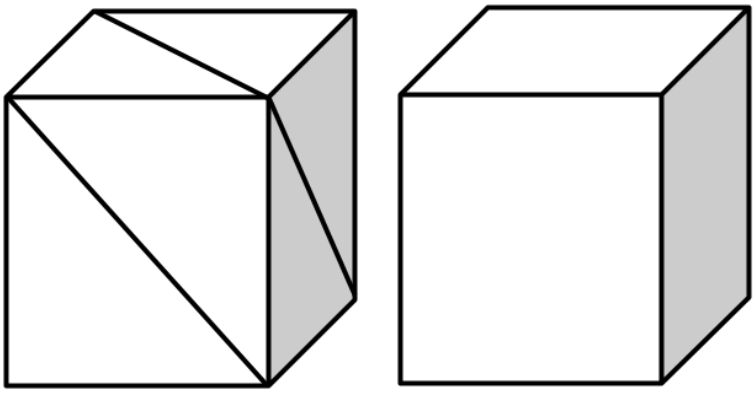

Figure 16. Triangle mesh and quad mesh.

Polygon meshes are like triangle meshes, but there are no restrictions on which polygons can be used. Compared with triangle meshes, fewer polygons might be required to represent the same volume. This makes the use of algorithms that are sensitive to the polygon count more efficient. As seen on Figure 16, to represent a cube, triangle mesh needs twice as many polygons as quad mesh. On the other hand, the data structure for polygon meshes is more complex. For triangle meshes, lists of vertices and connections between them are usually sufficient. For polygon meshes, the winged-edge data structure is used.

3.1.2 Winged-edge Data Structure

Polygon mesh contains information about vertices, edges between the vertices, and loops of edges that form faces. The winged-edge data structure is a way to connect all these components together in a way that accessing adjacent structures can be accomplished in constant time. The data structure is visualized on Figure 17. All the components are connected to edges; an edge has a pointer to the origin and destination vertices, pointers to left and right faces, pointers to clockwise (CW) and counterclockwise (CCW) edges from origin and destination vertices. Vertices are described by their location in 3D space and hold a reference to one of

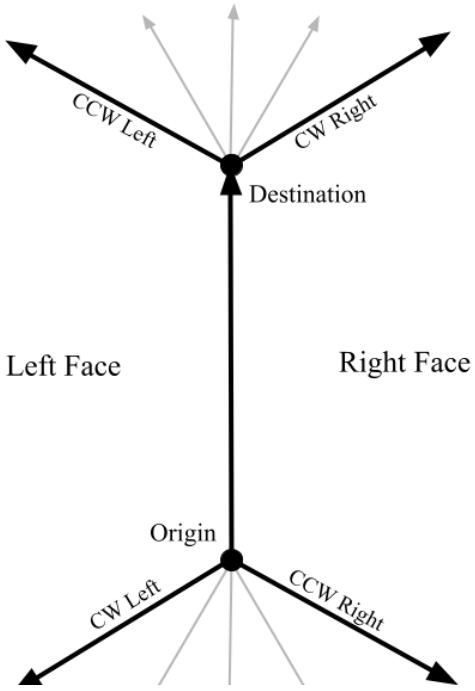

Figure 17. Winged-edge data structure.

the edges that are connected to it. All the other edges can be accessed by traversing the edge pointers.

As an example, to traverse all the edges cycling a face in clockwise order, the algorithm should enter clockwise right or clockwise left edges, depending on the direction of the edge. If the iterated face is on the right, clockwise right is chosen. If the iterated face is on the left, clockwise left is chosen.

To traverse over the edges connected to a vertex in clockwise order, an edge is first selected that has the vertex as the origin or the destination. Then, when the vertex is the destination, the algorithm should go to the counterclockwise left edge. When the vertex is the origin, the algorithm should go to the counterclockwise right edge.

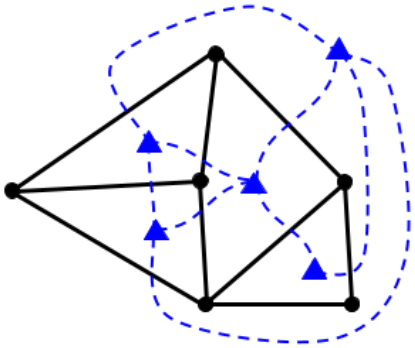

Figure 18. Dual graphs.

All the shapes in the editor are polyhedrons. Polyhedrons are a type of planar graph. Planar graphs have a dual graph, which is illustrated with the blue graph in Figure 18. In the dual graph, the nodes represent the faces, and edges represent the face adjacencies. An example dual graph of a polyhedron is given in Figure 19. In the figure, the octahedron is the dual graph of the cube. Dual graph of a dual graph is the graph itself, so the cube is the dual graph of the octahedron.

In the same way, there exists a dual graph for the winged-edge data structure. The dual of the winged-edge structure is called *dual shape*. The faces of the winged-edge structure become vertices, and an edge is established between vertices if two faces are left and right faces of some edge. The locations of the vertices in the dual shape can be defined as averages of the face vertex locations. In this thesis, dual shapes are only used for topological operations, therefore the vertex locations can be left undefined.

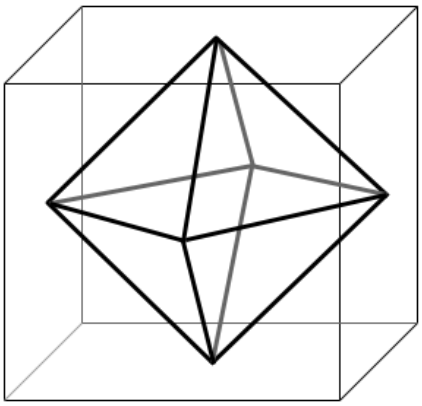

Figure 19. Dual polyhedrons.

It should be noted that the dual shape preserves the order of the edges of a face. Iterating over the edges of a shape's face in clockwise order is equivalent to iterating over the edges connected to a common vertex in the dual shape in clockwise order.

The implemented winged-edge data structure for the editor differs a little from the standard definition. Firstly, each geometrical edge has two topological edges. This way, every face is surrounded by winged-edges that go clockwise. This is illustrated on Figure 20, where the topological connections of a tetrahedron are presented on a planar graph. Edges with matching colours belong to the same face. Notice that all the edges go clockwise around the face. This is useful for triangulation and makes the traversal algorithms simpler to implement, at the cost of using more memory. Secondly, the data is held in arrays, not separate edge objects. Although this changes the data structure significantly, the algorithms do not change at all, since every connection to the edge is still accessible in constant time. This data structure allows the use of many different graph-based algorithms on the shapes.

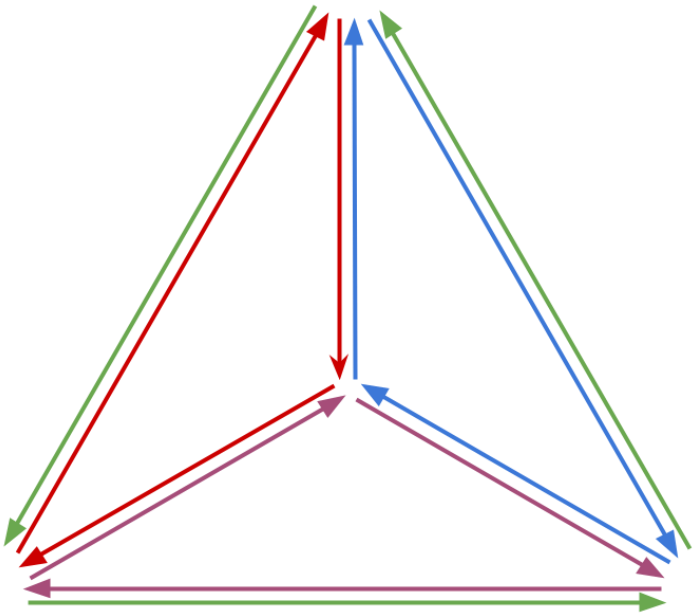

Figure 20. Each face is cycled by a directed edge in the clockwise direction.

3.1.3 Shape and Symbol

A winged-edge structure that encompasses a solid represents some shape that a user wishes to use in the shape grammar. The structure can be processed to produce a triangle mesh which can be used for rendering or exporting. Each shape has an associated symbol attached to it. A symbol is an abstract version of a shape; each shape is an instance of a symbol.

All the symbols are listed in symbol explorer. This is illustrated in Figure 21. In the figure, there are two symbols “S0” and “S1”. A new symbol can be created by pressing the “New Symbol” button. Symbols created in this way have cube topology. A way to create symbols with different topologies is described in chapter 3.1.4 Rule.

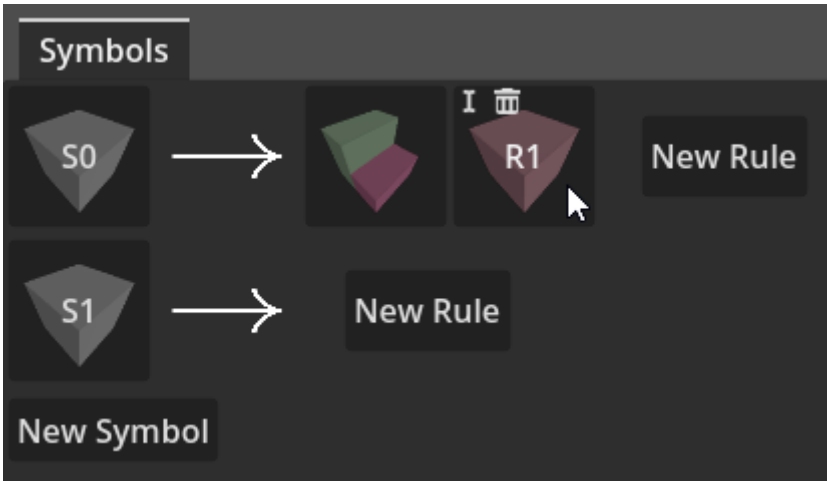

Figure 21. Symbol explorer and its components.

The UI component for symbols is illustrated in Figure 22. When the component is clicked, the symbol appears in the edit view. While hovering over the component, two buttons appear in the top-left corner of the component. The first button is for changing the name of the symbol. When it is clicked, the name in the centre becomes editable. The second button is for deleting the symbol. The editor does not have undo/redo functionality, so when a mistake is made, it can be deleted and created again. As seen in Figure 21, all the symbols are followed by an arrow. After the arrow, symbol rules are listed.

Figure 22. Symbol.

3.1.4 Rule

Users can create new rules by clicking the “New Rule” button next to the symbol they want to replace. As an example, the “New Rule” button in Figure 23 creates a new rule for the symbol “A”. After the creation of a new rule, an instance of the symbol is displayed in the edit view and an UI component is added to the symbol explorer. This component is identical to the symbol component seen in Figure 22, except the name is hidden by default. The name of the rule is not actively used during the definition of grammar; therefore, it is not necessary to display it continuously. In Figure

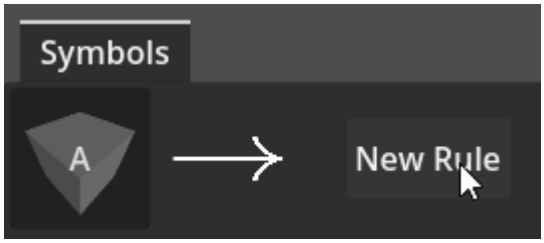

Figure 23. The “New Rule” button.

21, the symbol “S0” has two rules defined for it. The “New Rule” button is always available because one symbol can have multiple rules defined for it.

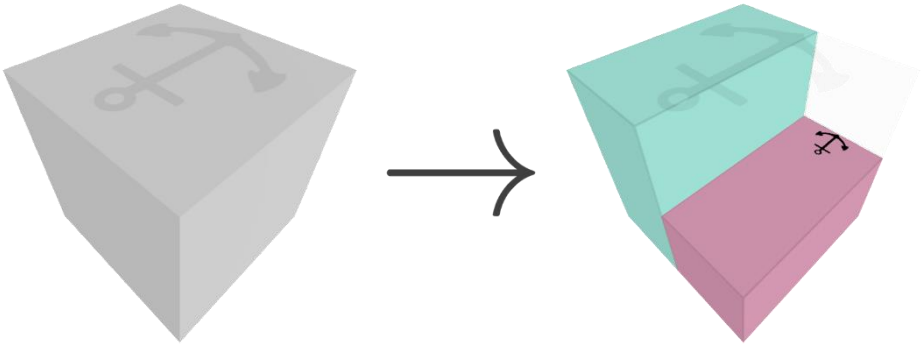

Figure 24. Rule definition.

The user can use different cutting methods to split the rule shape into multiple sub-shapes. This process is used to define the rule. This is illustrated in Figure 24, where a cuboid symbol “A” is split into two sub-shapes. The sub-shapes can also be individually split, which makes it possible to define complex rules. The complexity of the rule is more apparent by inspecting the tree view of the rule, shown in Figure 25. The tree view visualizes the *rule tree*, which is a data structure used to store the mutable form of the rule. The tree’s nodes represent two entities. Non-leaf nodes represent cuts on shapes. There may be many layers of cut nodes, which indicates that many iterations of sub-shapes were cut. Non-leaf nodes also store references to winged-edge structures that they formerly represented. Leaf nodes represent the sub-shapes. These are the resulting shapes that are generated when the rule is used on the original symbol. In Figure 25, the cube was cut twice, as seen by the rows with the scissor symbol. The rows marked with 1 and 2 in the tree view correspond with the shapes 1 and 2 in the edit view. The tree view can be used to manipulate the rule with different functionalities.

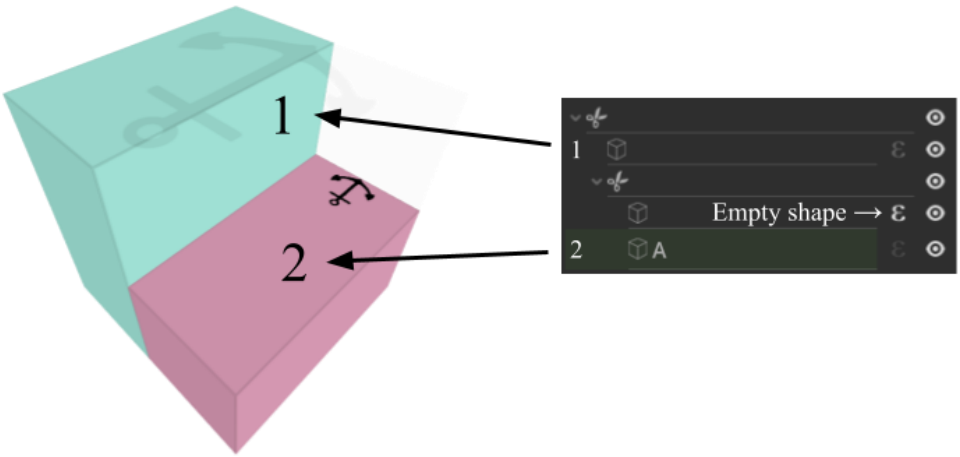

Figure 25. Components of the tree view.

Some sub-shapes may be obstructed by other sub-shapes, making them inaccessible from the editor view. To apply a cut on a sub-shape, it needs to be visible to the user. Therefore, the user must have control over the sub-shape's visibility. The visibility of a sub-shape can be changed in the tree view by clicking the eye icon, as seen on the right side of every row in Figure 25. Sub-shape visibility has two modes. The first mode is *visible*. In this mode, the sub-shape is fully opaque, and it can be selected. The second mode is *X-ray*. In this mode, the sub-shape is rendered with transparency and is not selectable. The sub-shapes behind that sub-shape are selectable instead. The visibility modes are illustrated in Figure 26. On the left side, all the sub-shapes are visible. On the right side, one of the shapes is in X-ray mode. That sub-shape is faintly visible, and the face behind that sub-shape is selected. Changing the visibility of the sub-shapes is helpful for creating complex rules with many layers of cuts. The visibility, however, does not affect the definition of the rule.

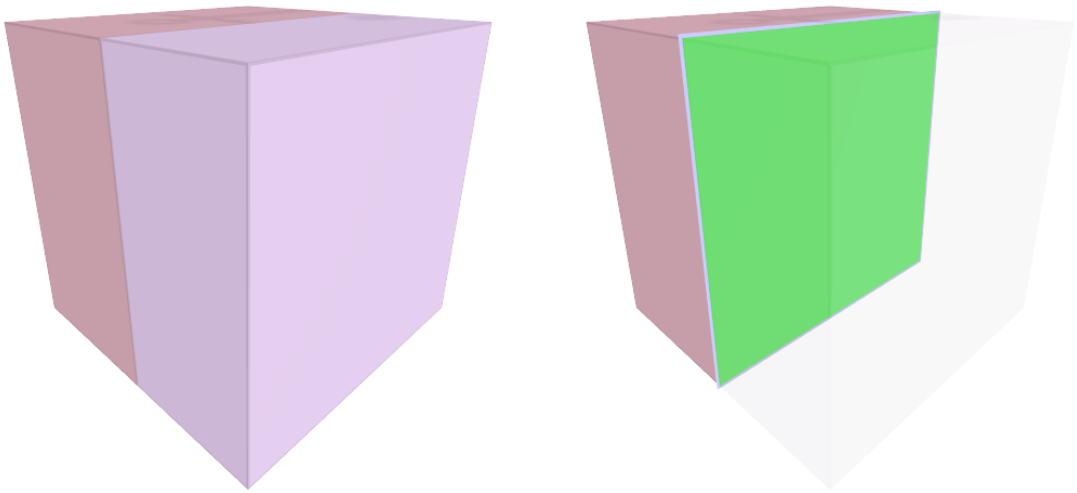

Figure 26. Visibility modes.

While a sub-shape can be set to invisible in the editor, the rule will still generate that sub-shape during the interpretation process. To remove volume from a symbol, a sub-shape can be specified to be *epsilon*. In formal grammar, epsilon represents emptiness. In the split grammar, it, therefore, represents the empty shape. The epsilon shapes are not visible in the editor view but are accessible in the tree view. In Figure 25, one of the sub-shapes has been specified as an empty shape. This is reflected in the edit view as well, where the shape has some volume removed compared to the symbol shape. Unlike invisible shapes, epsilon shapes are not generated when the rule is used on the symbol.

Non-empty sub-shapes can be terminal or non-terminal. Users can determine what symbols the sub-shapes will have by entering the name of a symbol in the tree view. This can be seen in Figure 25, where the shape on line 2 has been assigned symbol “A”. When the symbol field is left empty, it is automatically assigned an anonymous terminal symbol. Entering the name of a symbol that does not exist allows the user to create a new symbol. This is demonstrated in Figure 27, where the shape 1 was assigned symbol “B”, which does not exist. By clicking the button that appears at the end of the line, a new symbol is created that has the topology and geometry of the sub-shape it was created from. In the figure, the sub-shape is colored cyan, and the new symbol is represented by the gray shape on the right.

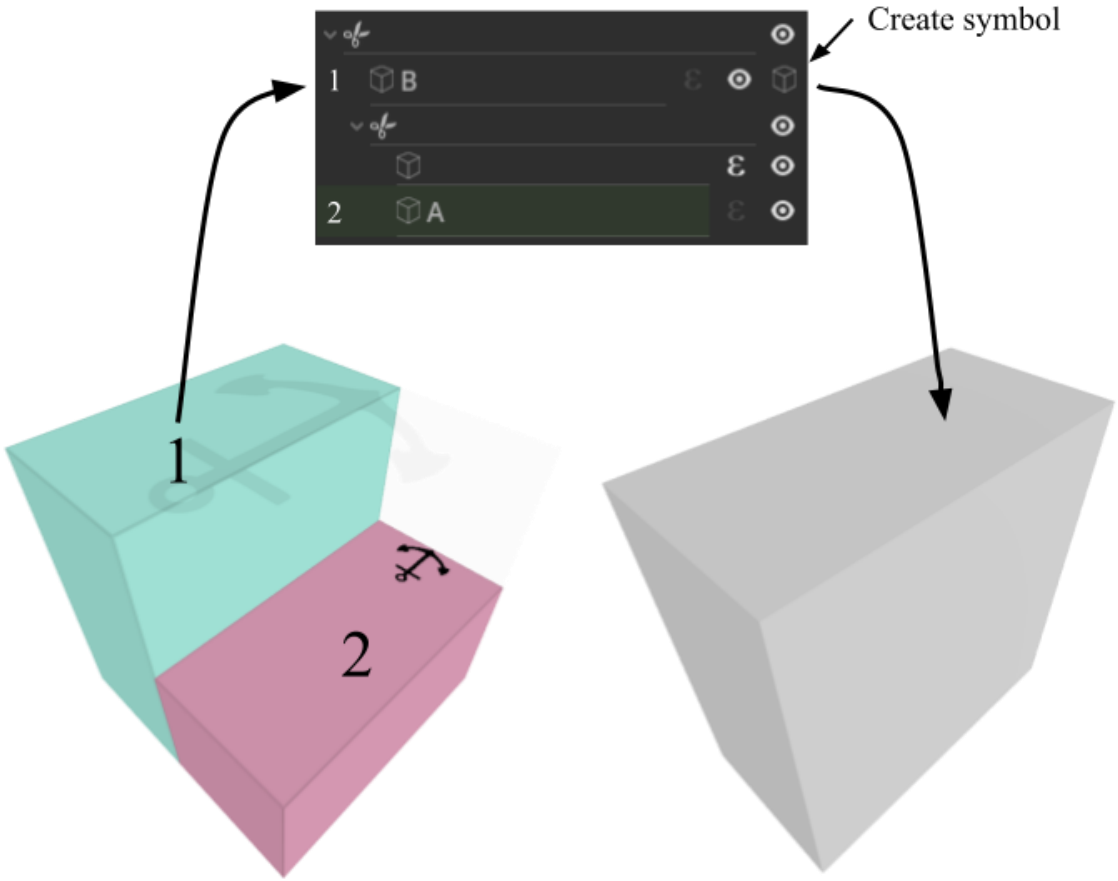

Figure 27. Symbol creation via tree view.

Shapes are instances of symbols. This means that the topology of the symbol and the sub-shape must match. Thus, entering the name of a symbol that does exist has two possible results. When the topologies are unequal (e.g., symbol has cube topology, but the shape is a tetrahedron), the user is notified by turning the background colour of the symbol box red. The sub-shape is still terminal and without a symbol. In the opposite case, when the topologies are equal, the sub-shape is treated as an instance of the symbol. This is signalled

to the user by the green background, as seen in Figure 27. Therefore, that sub-shape is non-terminal and can be replaced during the grammar interpretation. The topologies of the symbol and the sub-shape might match ambiguously. If the shape is a mirror image of itself, the user must specify the intended direction of the symbol. This is done using anchors.

3.1.5 Anchor

The ambiguity of the topology equivalence lies in vertex ordering. This is illustrated in Figure 28. On top of the figure, a rule is defined on a square symbol, with corner vertices ordered in a clockwise way. That square is replaced by a triangle, so that the edges of the triangle are between vertices 1, 4, and 3. Below the rule, four different vertex orders are given to the four squares and the results when these squares are replaced with the rule.

The top-right square has the same vertex ordering as the rule, so the replacement looks like one expects. The indices of the top-left square are mirrored compared to the top-right shape. This means that the replacement triangle is also mirrored. The bottom-left square has been

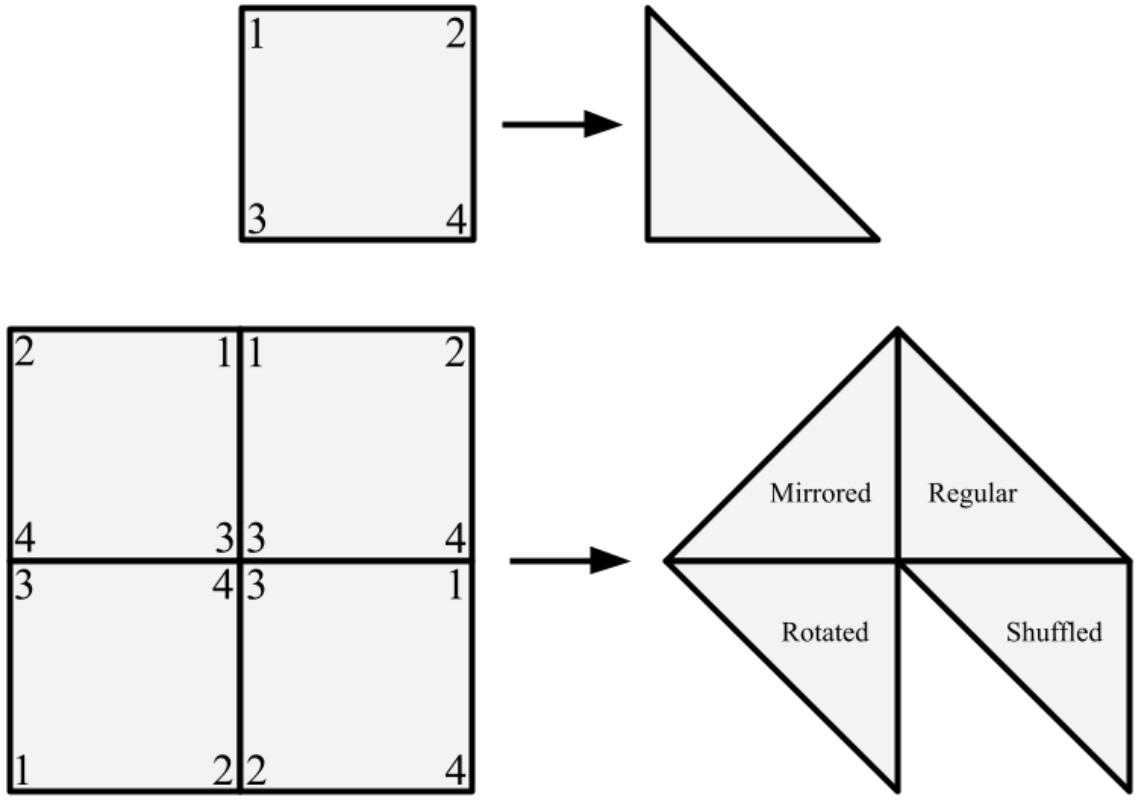

Figure 28. Different vertex orders.

rotated 180°, therefore the replacement is also rotated. Lastly, the bottle-right square's vertices are shuffled. This changes the topology of the shape. Notably, the vertices 1 and 4 are next to each other, which they are not in the rule definition. Because new edges are

between vertices 1, 3, and 4, which are in three different corners of the bottom-right square, the result is still a triangle. The result with shuffling does not make sense all the time. Multiple sub-shapes of the rule might start intersecting with each other. This kind of vertex order is undesirable.

Therefore, it is important that the users have an ability to influence what order the vertices are in. The user usually intends the rule to be used in one specific direction. A pillar should not go from wall to wall. Instead, it must rise from floor to ceiling. On the other hand, the topology should be preserved when changing the direction of the shape. In the editor, the vertex order is determined by the placement of an *anchor*.

The shapes are represented with the winged-edge data structure. The vertices in that structure are nodes of the graph; they do not form an ordered set. Thus, the vertices must be ordered with some algorithm before a rule can be applied on it. This order must be unique for each topology and anchor pair. This way, the user can change the vertex order by changing the location of the anchor. The vertex order must be unique because the rules are defined for one specific vertex order. If there were two different orders for one topology and anchor pair, rules with different vertex orders could be used on the shape, defeating the purpose of the anchor.

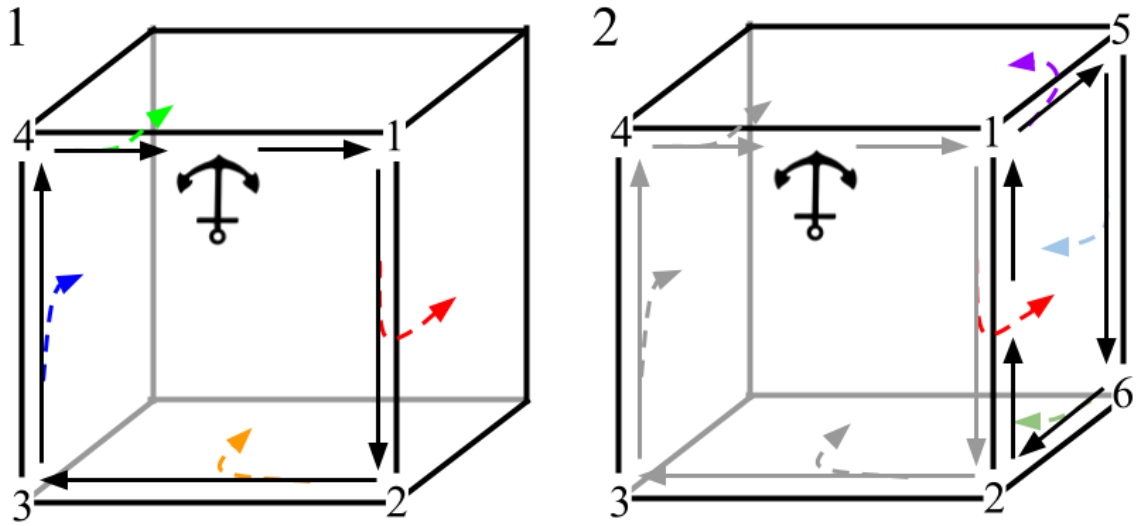

Figure 29. Face traversal in clockwise direction.

It is possible to order the vertices of a shape by traversing its faces. This can be done by doing a breadth-first traversal of the dual shape, because the nodes of the dual shapes are the faces of the shape. The ordering algorithm is illustrated in Figure 29. For each face, its

vertices are iterated in clockwise order. The vertices are ordered by the time they were first encountered during the traversal.

The vertices of a face form a cycle. Thus, to start the iteration, the first vertex must be specified. This is done by placing down the anchor. The anchor also specifies the first face of the dual shape traversal. This can be seen in Figure 29, where the first cube's face with the anchor is iterated in clockwise direction. The first vertex in the clockwise direction of the anchor is the first vertex and is marked with number one. Then the subsequent vertices are numbered as well, until the first vertex is encountered.

For other faces, the first vertex is determined by the edge that face was traversed into. In Figure 29, the colored arrows indicate the breadth-first traversal candidate paths. The first candidate is marked with the red color. The second cube shows how the second face is iterated and new vertices are ordered. As can be seen, already ordered vertices are not reordered.

The traversal path is also uniquely determined by the anchor position. The traversal candidates are entered into the traversal queue in clockwise order. In Figure 29, the queue is inserted with red path, yellow path, blue path, and green path, in that order. Then, on the second face, the queue is filled with purple path, sky-blue path, and lime green path, in that order. At every step, the insertion to the queue is unique. This means that the items coming out of the queue are not in random order. Thus, the traversal path is unique to the topology and anchor pair. Therefore, the vertex order is unique as well.

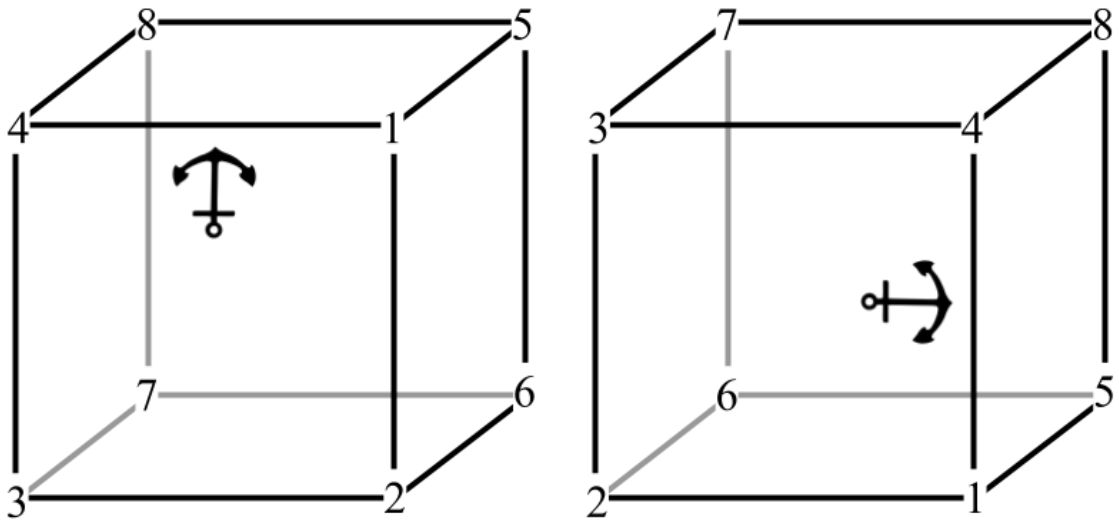

Figure 30. Final vertex order.

The algorithm ends when all the vertices are ordered. The final ordering of the cube can be seen in Figure 30 on the left. To the right of it, another ordered cube is given, but with a different anchor position. The anchor was placed to another edge, as if it was rotated by 90°. As can be seen, the vertex order for the new anchor rotated with it, thus demonstrating how the anchor can be used to determine the direction of the shape.

The anchor appears on the shape right after the user assigns a symbol to the sub-shape. The direction of the sub-shape can be changed by selecting the anchor. The user knows that the anchor is selected when its colour turns green. Pressing the keyboard's "," and "." keys move the anchor counterclockwise and clockwise on the face. To move the anchor onto another face, the user must press the "M" key on the keyboard and then select the face where the anchor should be placed.

Each symbol also has a *reference anchor*. This anchor is faintly visible on the symbol or rule in the edit view, as seen in Figure 31. This anchor gives the user the direction of the symbol. When the user wants to replace a sub-shape with that symbol in a rule, they must place the anchor according to the location of the reference anchor.

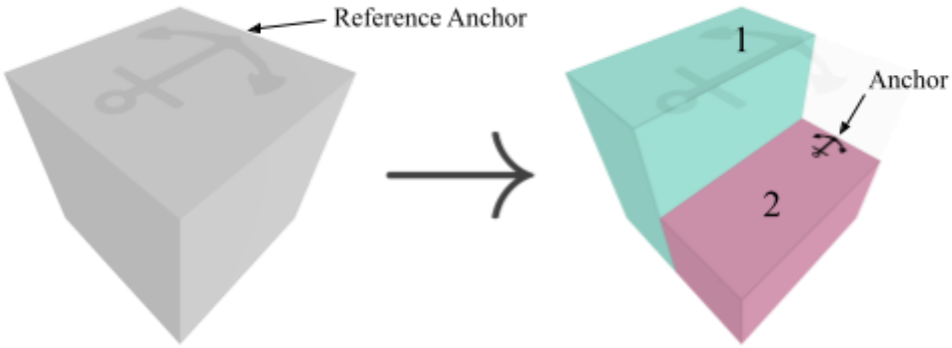

Figure 31. Reference anchor and anchor.

For example, the reference anchor for the symbol "A" in Figure 31 is on the top of the shape. On the RHS of the rule, the left arm of the reference anchor is pointing towards the side where the shape 2 is. The shape 2 was assigned the symbol "A", so the anchor appeared on the shape. That anchor's left arm points in the same direction as the reference anchor. Therefore, during the interpretation, the shape 2 is replaced by the symbol "A", which is replaced by the rule, all while the anchor keeps pointing left. Four generations of applying this rule on a cube are given in Figure 32.

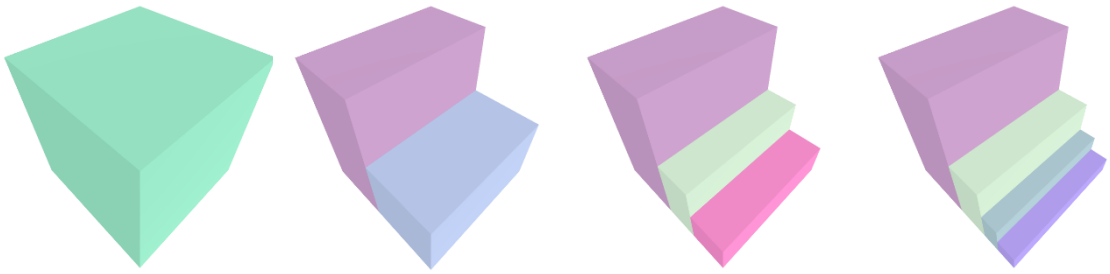

Figure 32. Four generation of applying the rule in Figure 31.

Changing the location of the anchor will change the direction in which the symbol “A” is placed, and that direction will propagate to all the replacements that are done on that symbol and its future potential replacements. In Figure 34, an alternative rule is given, where the anchor was rotated 90° compared to the rule in Figure 31. In Figure 34, four generations are given for that rule. As can be seen, the generations create spiralling steps, rotating the symbol “A” by 90° each time it is placed.

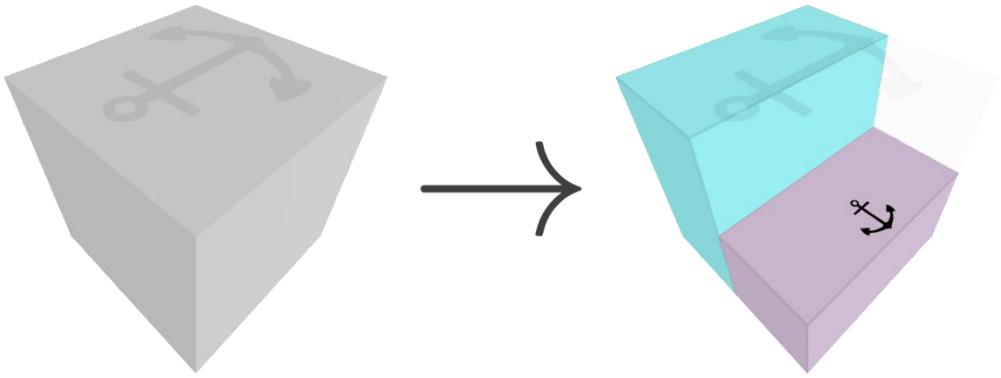

Figure 33. A rule with different symbol direction.

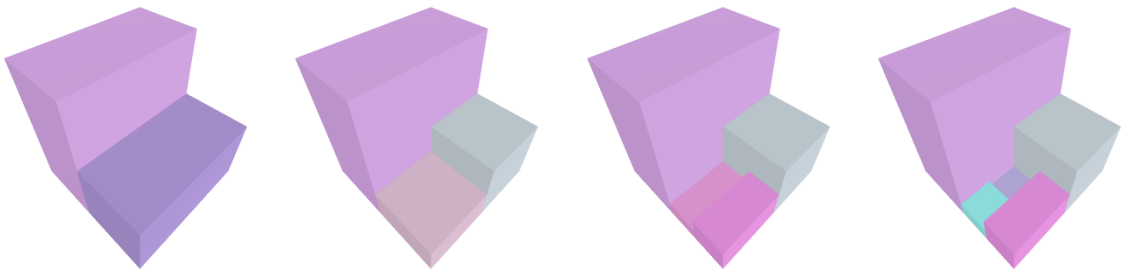

Figure 34. Four generations of applying the alternative rule.

Sub-shapes with an anchor cannot be cut. To cut these sub-shapes further, the symbol must be unassigned. Cuttable shapes can be cut by different cutting methods, all of them based on an intersection of a shape and a plane.

3.1.6 Cut

To make a cut, a shape and a cutting plane is needed. The shape is in the winged-edge form, and the plane is characterized by a 3D point and a normal vector. At the end of the cut, there should be two sub-shapes, one on both sides of the cutting plane. The overview of the cutting algorithm is given in Figure 35. In the figure, black edges represent edges of the shape that is going to be cut. Yellow vertices and edges represent the construction stage of the first sub-shape and the purple vertices and edges represent the construction stage of the second sub-shape. Blue vertices are cut vertices and are in both sub-shapes.

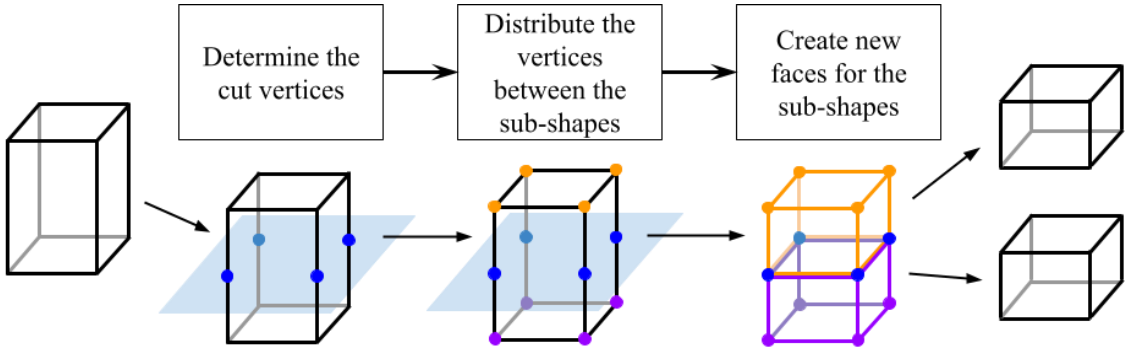

Figure 35. Cutting algorithm overview.

Some cuts require that new vertices are generated. Two different cuts are presented in Figure 36. With the first cut, all the existing vertices can be reused when creating the sub-shapes. For the second cut, four new vertices must be created. So, the first step of the cut algorithm is to identify reusable vertices and create new ones when necessary.

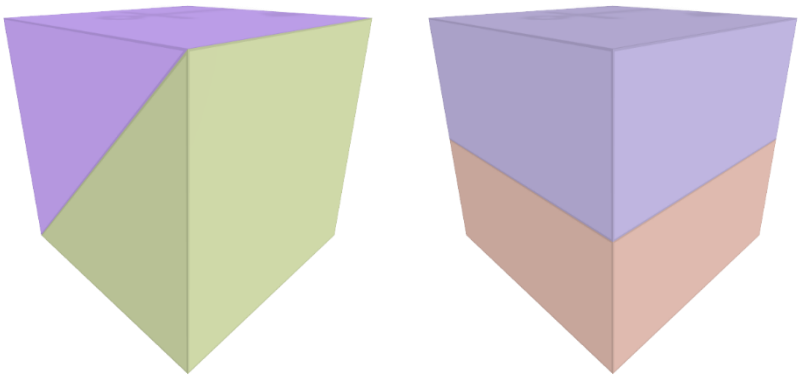

Figure 36. Two cuts on a cube.

The positions of these vertices are located by doing a line-plane intersection for each edge of the shape. In addition to finding the location of these vertices, it is important to store the endpoints of the edges and how to construct the cut location from them. This information is used to compile grammar rules later.

The next step of the cutting algorithm is to determine which vertices belong to what sub-shape. This is done by traversing the vertices of the shape and adding them to the sub-shapes depending on which side of the cutting plane they are located. All the vertices found in the first step are also added to both sub-shapes.

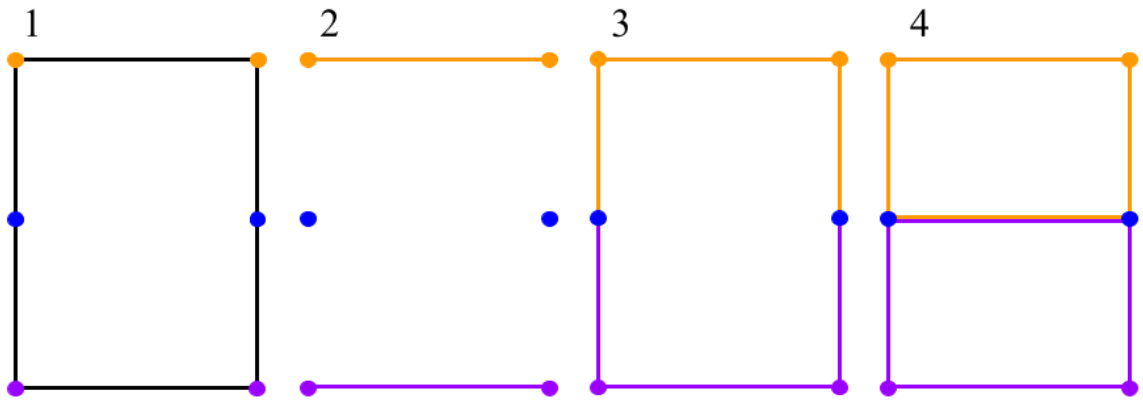

Figure 37. Stages of face construction.

After determining the vertices of the sub-shapes, faces must be constructed. A face crosses the cutting plane if some of its vertices are on both sides of it. If a face does not cross the cutting-plane, it is copied to the appropriate sub-shape. If a face does cross the cutting-plane, an algorithm is used to split the face into two. This algorithm is illustrated in Figure 37. At stage 1, only vertices have been partitioned between sub-shapes. The original edges are drawn with black lines. At stage 2, the edges between vertices are added to appropriate sub-shapes except ones that cross the cutting-plane. At stage 3, these connections are replaced with connections to the vertices found in the first step of the algorithm. Lastly, at stage 4, a connection is created between the cut vertices of the new shape. Both sub-shapes also must create a new face alongside the cutting plane, shown with an arrow in Figure 38.

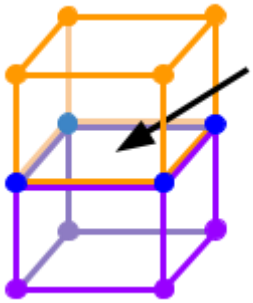

Figure 38. Missing faces.

It should be noted that cutting convex faces preserves their convexity. It is known that the intersection of two convex polygons is also convex. Cutting the face with a straight line can be thought of as finding two intersections of the polygon with bigger convex quads, as seen on Figure 39. Therefore, if all the grammar shapes are derived from a shape with convex faces, all the faces in the grammar are convex. This is the case in the editor, because the default shape is a cube. This property is useful when converting a polygon mesh into a triangle mesh.

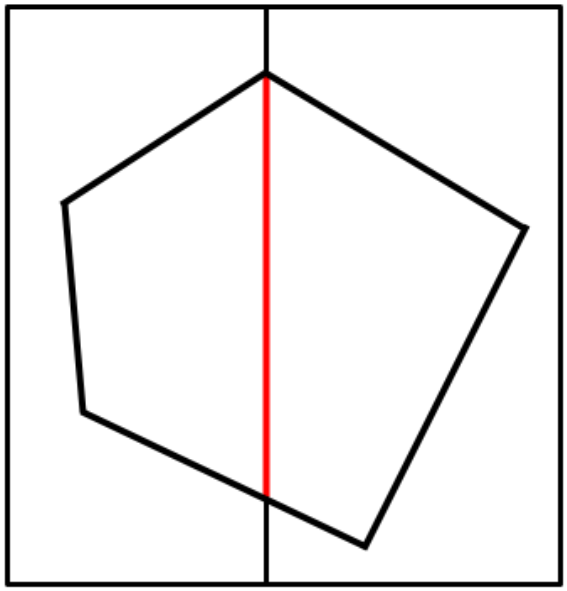

Figure 39. Cutting a shape is equivalent of
taking intersections of the shape and big quads.

3.1.7 Cutting Methods

Users of the grammar editor are given multiple tools to cut shapes. The cutting methods are accessible in the toolbar on the left side of the screen. The toolbar only appears when a rule is selected. The users are provided with four methods for cutting the shapes.

3.1.7.1 Face Cut

The Face Cut tool is the most basic cutting method. The use of this tool is illustrated in Figure 40. The user must select one of the faces of the shape in the middle of the screen. After selecting the face, a cutting plane can be created by pressing the “C” key on the keyboard. This cutting plane follows the cursor alongside its normal vector. This way, the user can move the plane where they want to cut. When the plane is in a suitable place, the cut is completed by right-clicking. A more sophisticated version of this tool is the Prism Cut.

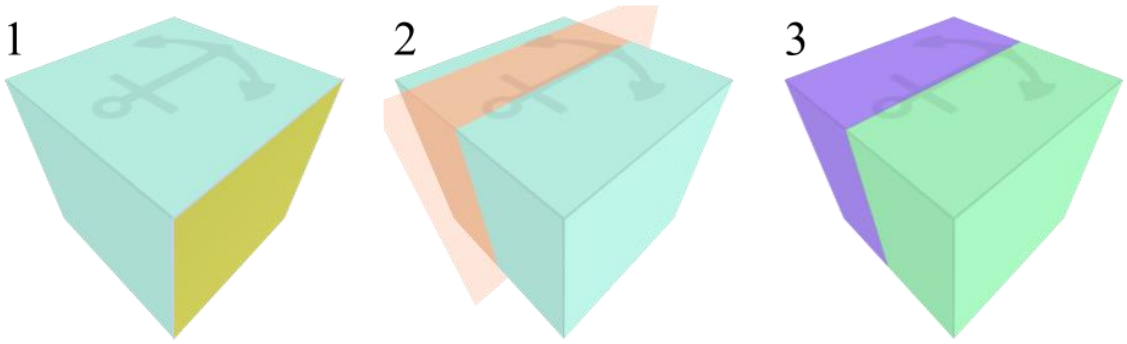

Figure 40. Steps to make a face cut: select a face, press “C” and drag, right-click to complete.

3.1.7.2 Prism Cut

The use of this tool is illustrated in Figure 41. While with the Face Cut tool, the user could select any face, the Prism Cut tool requires the face to be a base of a prism. The cutting plane is created by pressing the “C” key. A prism has two bases, so instead of moving along the normal vector, the cutting plane moves between the faces. The user can also snap the cutting plane to a grid by holding down the “Ctrl” key. However, the grid might not allow the cut to split the shape into equal sub-shapes. Therefore, a more precise tool is necessary.

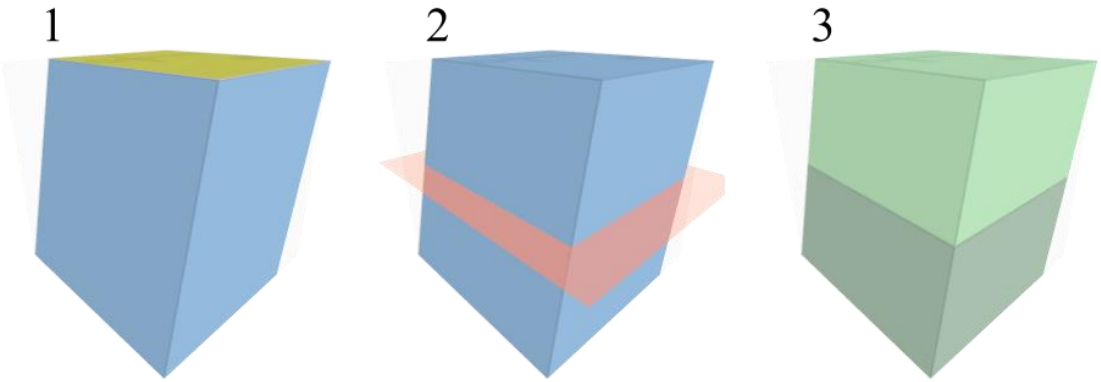

Figure 41. Steps to make a prism cut: select a face, press “C” and drag, right-click to complete.

3.1.7.3 Multi-Cut

The Multi-Cut tool is an abstraction of the Prism Cut tool. The use of this tool is illustrated in Figure 42. First, the user must specify the face and the number of cuts. The number of cuts can be modified in the tool info section of the user interface when the multi-cut tool is active. After pressing the “C” key, the editor automatically calculates the suitable locations for the cutting planes and completes the cut according to them. As a result, the prism was cut into multiple sub-prisms. The number of sub-prisms is one more than the number of cuts, which the user had to define at the start.

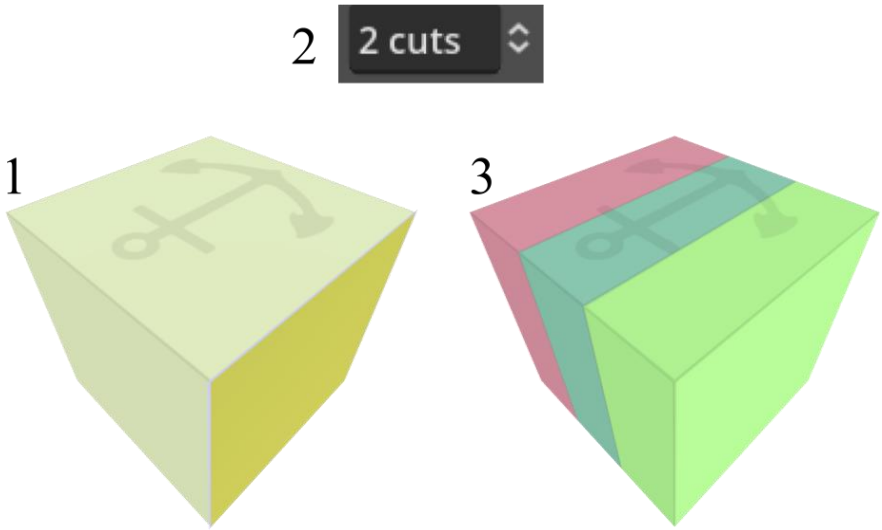

Figure 42. Steps to make a multi-cut: specify the number of cuts, select a face, press “C”.

Using only the Face Cut, Prism Cut, and Multi-Cut makes it difficult to sculpt the shape into some new topology. This is because the cutting planes are defined by one of the faces, making it possible to “shorten” the shapes. For example, starting with a cuboid, using these tools makes it impossible to create a sub-shape that is not a cuboid. A cutting plane is always parallel to one of the faces of a sub-shape. Thus, a tool that can define any cutting plane is needed.

3.1.7.4 3-Point Cut

This tool is also called “Tri-Point Cut.” The user must specify the locations of three points on the editable shape. From the three points, a cutting plane can be defined. This method can be used to cut across sub-shape boundaries. If the triangle that is formed from the three points intersects multiple sub-shapes, all of them are cut along the plane.

The use of the tool is illustrated in Figure 43. After activating the tool, a red circle appears on one of the edges or vertices of the shape. This circle represents one of the three points that the user must define the location for. The circle follows the cursor. To move it onto vertices, the user must hold down the “Shift” key. The placement on the edges can be snapped to a grid by holding down the “Ctrl” key. The location of the point can be locked down by doing a right-click. After locking down the placement of the point, the user must repeat the process for the remaining points. When all three points’ locations are specified, a

cutting plane is defined, and the cut is done on the shapes that the triangle between the points intersects with.

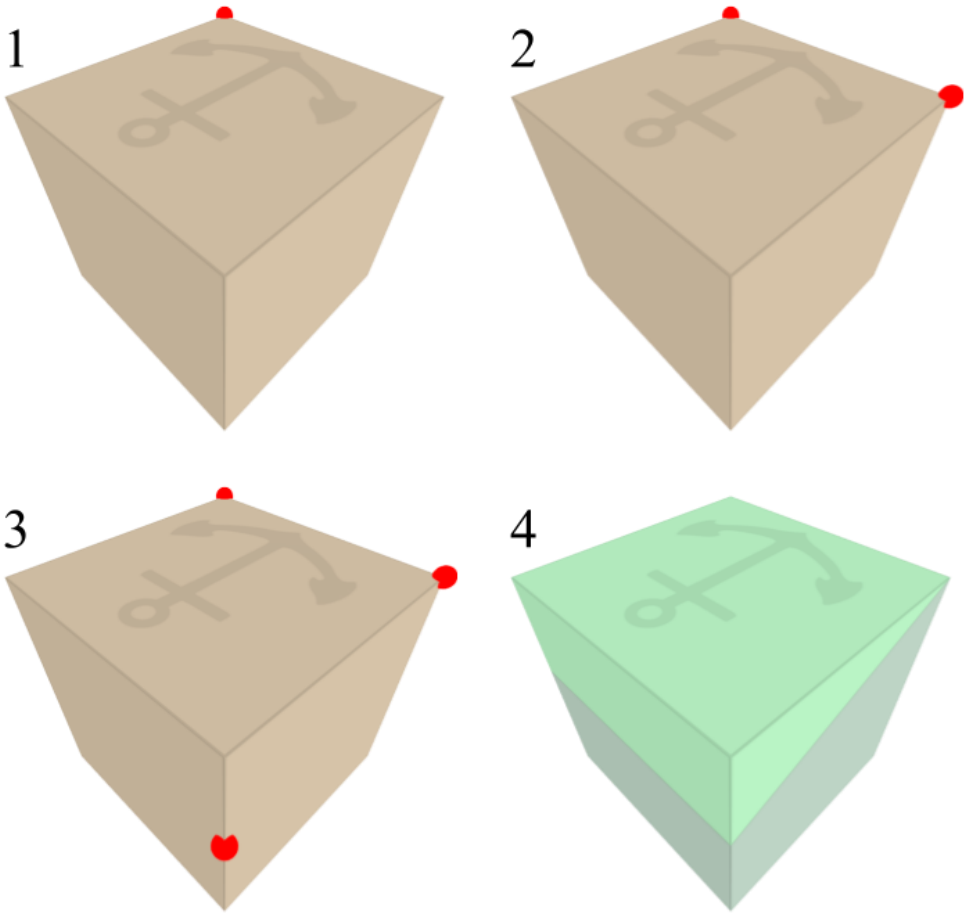

Figure 43. Steps to make a 3-point cut: place three cutting points.

The 3-Point Cut tool is very versatile. It can generate sub-shapes with a completely new topology. As is seen in Figure 43, triangular and pentagonal prisms were created from a cube. Alongside Face Cut, Prism Cut, and Multi-Cut, the 3-Point Cut offers users a toolkit that can be used to sculpt any kind of shape. Next to cutting shapes, the editor has a few more features. One of them is the ability to save the project into a file.

3.1.8 Persistence

The users can save grammars to a file. There are two methods to save the project. First method, called “Save”, saves the project to a previously specified file. The user only must pick the file path on the first “Save”. After, that path is reused. To change the destination of the save file, the users can use the second method, “Save As”. This method asks the user for

the destination path on every evocation. Both methods are available as buttons in the “Project” pop-up window, which can be accessed by clicking the “Project” button. The menu is illustrated in Figure 44, where two first buttons are for saving the project, and the third button is for loading.

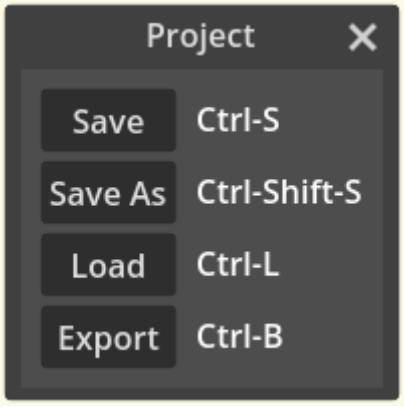

Figure 44. Project menu.

The save file contents are binary encoded Godot primitives and arrays. The encoder provided by Godot does not support custom objects, therefore all of them had to be converted into arrays. This also means that pointers between objects could not be saved; after decoding, the location of a saved object might be different. After the user loads the save file, the editor state must be equivalent to the state when it was saved.

That problem was solved by algorithmically retracing the user's steps in the editor. This creates connections between objects automatically. Where the connections between objects were not necessary, full data replacement was used instead.

At the beginning of the save file, an array of symbols and rules from the symbol explorer are saved. This data is later used to retrace the user’s steps. Next, information about the symbol creation is saved. This information is important so that the user cannot create conflicting symbols after loading the project. For example, a symbol with an ID that already exists. Finally, information about each individual rule is saved. A rule is represented by a rule tree, which is serialized into an array of tree items. During loading, the serialized tree is parsed back into a tree, which replaces the one that was created after loading symbol data.

It should be noted that the project save file is not intended for importing the grammar to the plugin. The contents of the save file are for only recreating an editor state. The files that are used in plugins are much more optimized. The creation of the importable files is done by the compiler.

3.1.9 Compiler

Compiling is a process of converting the boundary representation of shapes, symbols, and rules into lean data structures that only hold information necessary for grammar interpretation. These data structures are `GrammarShape`, `GrammarSymbol`, and `GrammarRule`. All of them are collected in the structure `Grammar`, which can be exported

from the editor and imported into the plugin. The structure of these types is presented in this sub-chapter. How these data structures are used during the interpretation process is discussed in Plugin chapter.

The shape's geometry and topology are defined with `GrammarShape` and `GrammarSymbol`, respectively. `GrammarShape` is an array of vertices as 3D vectors and a pointer to `GrammarSymbol`. `GrammarSymbol` has information about the number of vertices, and how these vertices are connected to form faces. This information is encoded as a list of face loops, where each loop is a list of vertex indices. The data structure has a field for ID and name, to make it accessible to users. Lastly, `GrammarSymbol` has a list of pointers to `GrammarRule` objects.

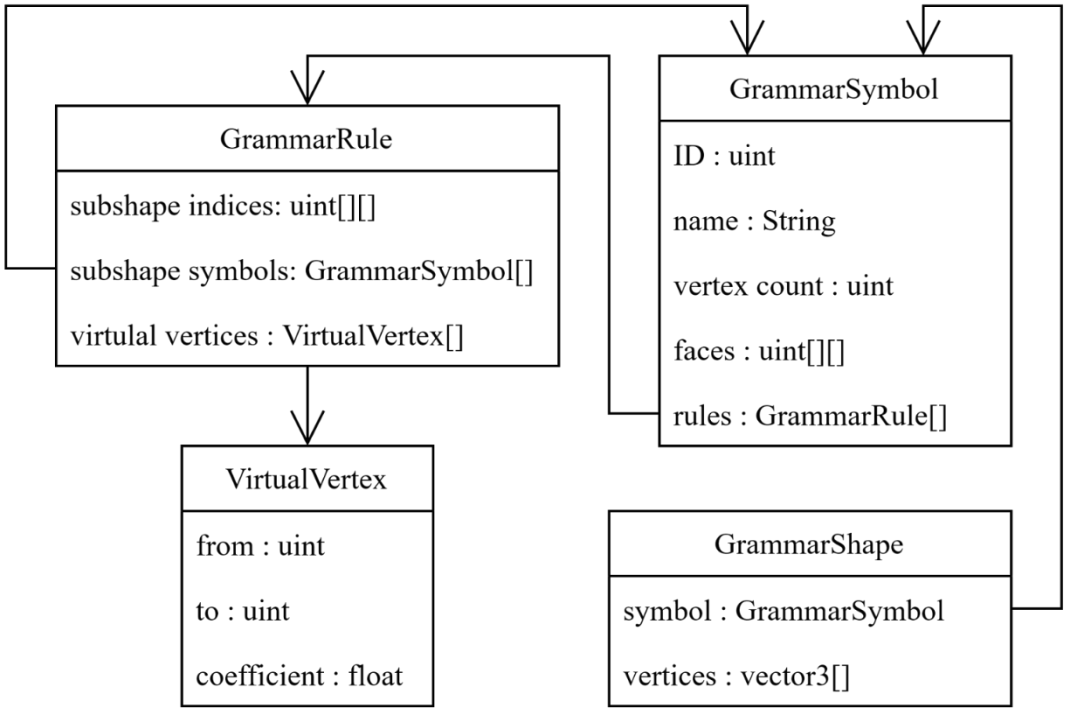

Figure 45. Class diagram for lean types.

`GrammarRule` represents a rule that is defined on a symbol. The data structure consists of three arrays. The first array holds virtual vertices. Virtual vertices are vertices that do not exist at the start of the usage of rules. The array holds instructions on how to construct these vertices. The instructions are encoded as two vertex indices and a linear interpolation coefficient. The other arrays hold information about the sub-shapes of the rule. The second array specifies which symbol to assign to the sub-shape when it is created and the third array vertex indices that are used to construct the sub-shape. All the classes and the connections between them are presented in Figure 45.

The shape's geometry and topology are present in the winged-edge data structure. The construction of the `GrammarShape` and `GrammarRule` is done by traversing the winged-edge graph and collecting the locations of vertices, vertex indices that are defined by the anchor, and faces.

The construction of `GrammarRule` is more complex. Rules are represented as trees. During interpretation, it would be inefficient to traverse a tree each time a rule is used. As discussed in chapter 3.1.4 Rule, non-leaf nodes hold information about the cuts and leaves hold information about the sub-shapes. Therefore, the virtual vertices can be gathered from non-leaf nodes and the sub-shape data from leaves. When a leaf node is an empty shape, it is ignored.

The list of `GrammarShape`, `GrammarSymbol`, and `GrammarRule` objects are collected into a `Grammar` object. The `GrammarShape` objects are symbol prototypes, which are used as starting shapes. `GrammarSymbol` and `GrammarRule` objects are the components of the shape grammar. The `Grammar` object can be serialized, which converts all the objects into arrays. The serialization contains the information about pointers between the objects, so they can be reconnected during the deserialization. The grammar is exported by saving the serialized `Grammar` object into a file. The grammar resource file has an appendix ".grammar".

The `Grammar` object inherits from Resource type, which means that it can take advantage of the resource system of the Godot. One of the advantages is that the resources are cached; a resource is loaded from the disk only once. This resource can be used with the plugin to use the user-defined grammar in Godot games. The editor uses the plugin as well, so that users can test their grammars in the editor.

3.1.10 Tester

Users can test their grammar by activating the testing mode. This is done by pressing the "Test" button in the project panel. When the testing mode is entered, all the UI elements are hidden, and the edit view is replaced with the test view. In the middle of the test view, the initial shape appears. The initial shape is determined by which symbol/rule was selected before entering the testing mode.

In the top part of the windows, the "Next generation" button appears. Pressing that button triggers a rule interpretation step. During the step, all the non-terminal shapes are gathered,

and grammar rules are applied on them. Then, the non-terminal shapes are replaced by the results of the rule applications.

The shape in the testing view can be exported into an STL mesh file by pressing the “Export Mesh” button next to the “Next generation” button. This allows the user to create 3D assets with the editor. The testing mode can be exited by pressing the “Test” button again. The edit view reappears, with the previously selected symbol/rule in the center.

The Editor is the main component of this thesis, therefore the largest. Used data structures and algorithms were presented, as well as main components of the shape grammar and how the users can define them. Lastly, the export formats and testing were detailed. The editor has many functions that allow users to create interesting shape grammars. The next section describes how to use these shape grammars to generate interesting shapes in Godot Game Engine applications.

3.2 Plugin

The shape grammar plugin for Godot is the second software component of this thesis. It was created as a proof of concept that grammars defined with the grammar editor can be used to generate meshes in real time. As such, the plugin offers the users with only one node type, `GrammarInstance3D`. That node is a mesh instance, where the mesh is acquired from the `GrammarState` resource. The `GrammarState` resource holds a state of a grammar, which is specified with the `Grammar` resource. The `GrammarState` is mutated with the grammar interpreter.

3.2.1 Interpreter

The interpreter takes a list of `GrammarShape` objects as the input and returns a list of replacements. The replacement shapes are determined by the `GrammarRule` objects, which were selected from `GrammarSymbol` objects linked to the shapes. The rule selection is uniformly random for shapes which’ symbol has more than one rule defined for them.

After the rule is selected, it is used on the shape. First, an array of vertices is created. All the vertices of the shape are added to it. Next, the instructions for virtual vertices are followed to generate more vertices, which are added to the array as well. Finally, sub-shapes are constructed by taking vertices from the array according to sub-shape vertex indices and then assigning appropriate symbols to them.

3.2.2 GrammarState

The `GrammarState` object is a resource that holds a state of a grammar. The user variables are available in the inspector, which is shown in Figure 46. The "Grammar" field is for specifying what `Grammar` resource to use. "Seed" is a 48-bit number represented in hexadecimal form. This seed is for the random number generator, which is used when randomly selecting the rules. The "Generations" field dictates how many times the interpreter should be used. The interpreter is used automatically if "Auto Generate" is checked. The "Initial Symbol" is a dropdown menu where the user can determine which symbol to use as the starting symbol. This field only appears when the "Grammar" field is filled, and the contents of the drop-down menu are automatically gathered from the `Grammar` resource.

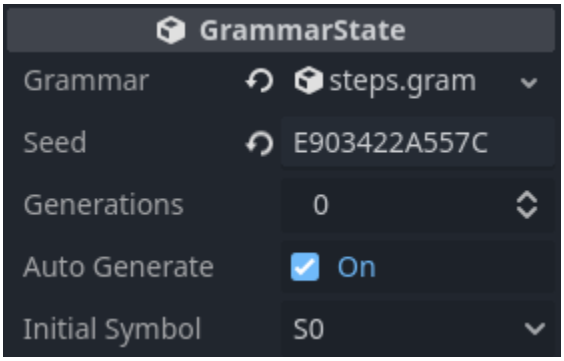

Figure 46. `GrammarState` inspector.

Alternatively, the generation can be done manually in code. In that case, the "Auto Generate" option must be disabled, and the `generate_all` function should be called. Using the `next_generation` function, the user can use the interpreter one generation at a time, even exceeding the generation count specified with the "Generations" field. The state can be reset by calling the `start` function. All the inspector fields in Figure 46 are accessible in code as well.

The state is stored as two arrays of `GrammarShape` objects. One array is for terminal shapes, and the other for non-terminal shapes. The latter is used as the input for the interpreter. After the interpretation, the output is split and appended to appropriate arrays. When all the necessary generations have been done, the arrays of shapes must be converted into triangle meshes so they can be rendered. This is done with the `GrammarMesh` object.

3.2.3 GrammarMesh

The `GrammarMesh` resource is a new `Mesh` type that uses the arrays in `GrammarState` to generate a mesh. The faces of the shapes are triangulated with a fan triangulation algorithm, which can be used because they are convex polygons. The normals are calculated from the face vertices using a cross product. Finally, the vertex colour is specified. By default, the colour is white.

The GrammarMesh resource must be linked with a GrammarState. This can be done by assignment in inspector or code. During the assignment, the GrammarMesh connects a callback to GrammarState's changed signal. This callback generates a new mesh when the GrammarState changes. This means that the user must do nothing to keep the GrammarMesh up to date with GrammarState. The GrammarMesh always represents the shapes in the GrammarState correctly.

3.2.4 GrammarInstance3D

GrammarMesh is a resource. Thus, it cannot be used as a node in a scene tree. The GrammarInstance3D node is used to instance the GrammarMesh. The user can either specify the GrammarState object or a GrammarMesh directly, as seen in Figure 47. If the "Grammar State" field is assigned, the grammar instance queries the state for the mesh automatically. In case the GrammarMesh exists as a resource on a disk, it can be assigned directly to the "Grammar Mesh" field.

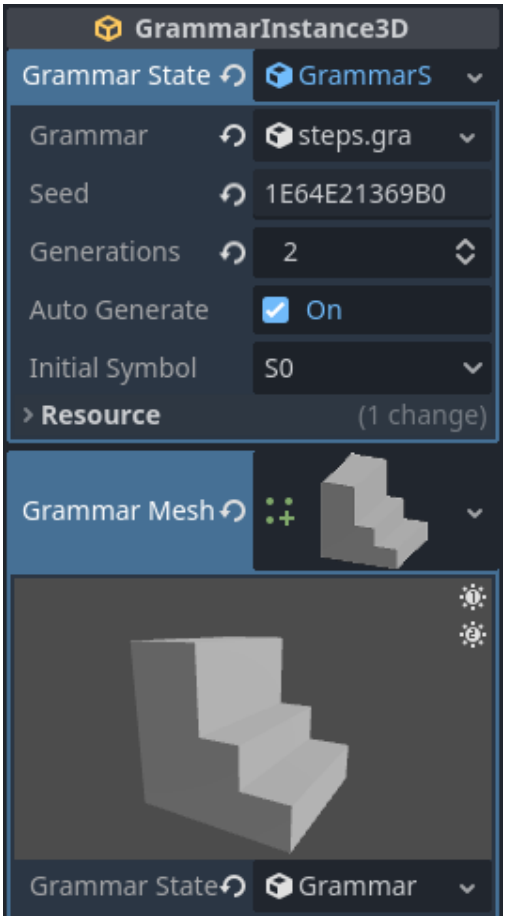

Figure 47. GrammarInstace3D inspector.

In the figure, the fields have been maximized, showcasing that the GrammarState and GrammarMesh can be modified via the GrammarInstace3D node. A tutorial on how to use this node is in Appendix I.

If the "Auto Generate" field is checked for the GrammarState, the generated mesh can be seen in the editor as well. As with any other mesh instance in Godot, it can be scaled, rotated, and translated. A material can be assigned to it, although UV coordinates have not been calculated.

4 Results

The goal of this thesis was accomplished. There now exists two tools that allow users to design custom split grammar and use it in Godot game during run-time. This chapter gives an overview of the quality of these tools. The intuitiveness was tested on five experts. The analysis of this test is discussed in the first and second sub-chapters. A demonstration grammar was created and presented in the third sub-chapter. Finally, possible future improvements are presented in the fourth sub-chapter.

For user testing, a questionnaire was constructed. First, some background information about the subject was gathered. The subjects of the user testing have experience in 3D modeling, are confident in their programming skills, and are familiar with computer graphics in general. Four of the subjects have had experience with formal grammar, and three of them have heard about shape grammar. After that, the users were asked to complete a set of tasks and rate the difficulty of each. After the tasks, the users were given the opportunity to write about their experience. This was used to gather details about what the users found difficult.

4.1 Tasks

The users were given nine tasks to complete, which together acted as a tutorial for the editor. The tasks are included in the accompanying files, which are described in the Appendix II. The topics of the tasks were:

Task 1. Rule Creation
Task 2. “Face Cut” tool
Task 3. Testing
Task 4. Rules with multiple cuts, volume removal
Task 5. Recursive rules, anchor placement
Task 6. “3-Point Cut” tool
Task 7. “Multi-Cut” tool
Task 8. “Global Cut” option of “Face Cut” tool
Task 9. Using multiple rules together, independent task

In each task, the user was given instructions on how to use the editor’s UI components or key-board keys to complete the tasks. An overview of the tasks is given in the table. The version of the editor that the users were tested on is different to the final version. The testing version does not support saving or exporting and has a separate button for manual

compilation. The later feature was added to the final version as the result of the feedback gathered from the users.

By the end of all the tasks, the users created all the rules for a generation of a Mayan pyramid, which is shown in Figure 48. A Mayan pyramid. Task 9 was left without detailed instructions to test if the users could learn enough about the tools from previous tasks.

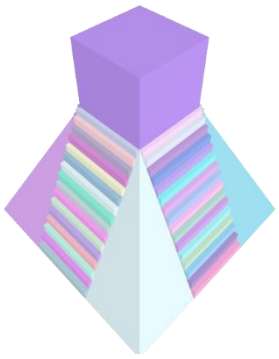

Figure 48. A Mayan pyramid.

The users were asked to rate the difficulty of each task on a 1 to 10 scale. The average scores with standard deviations are given in Figure 49. The error bars visualize the standard deviation of the score. As can be seen, tasks 1 to 4, 7, and 8 were simple for the subject. The highest score of four was given to task two by one of the users. Rest of the users rated these tasks with ones, twos, and threes. Task 6 was also rated similarly except by one user who found the controls for the "3-Point Cut" tool annoying because they conflicted with a screenshot software installed on their computer.

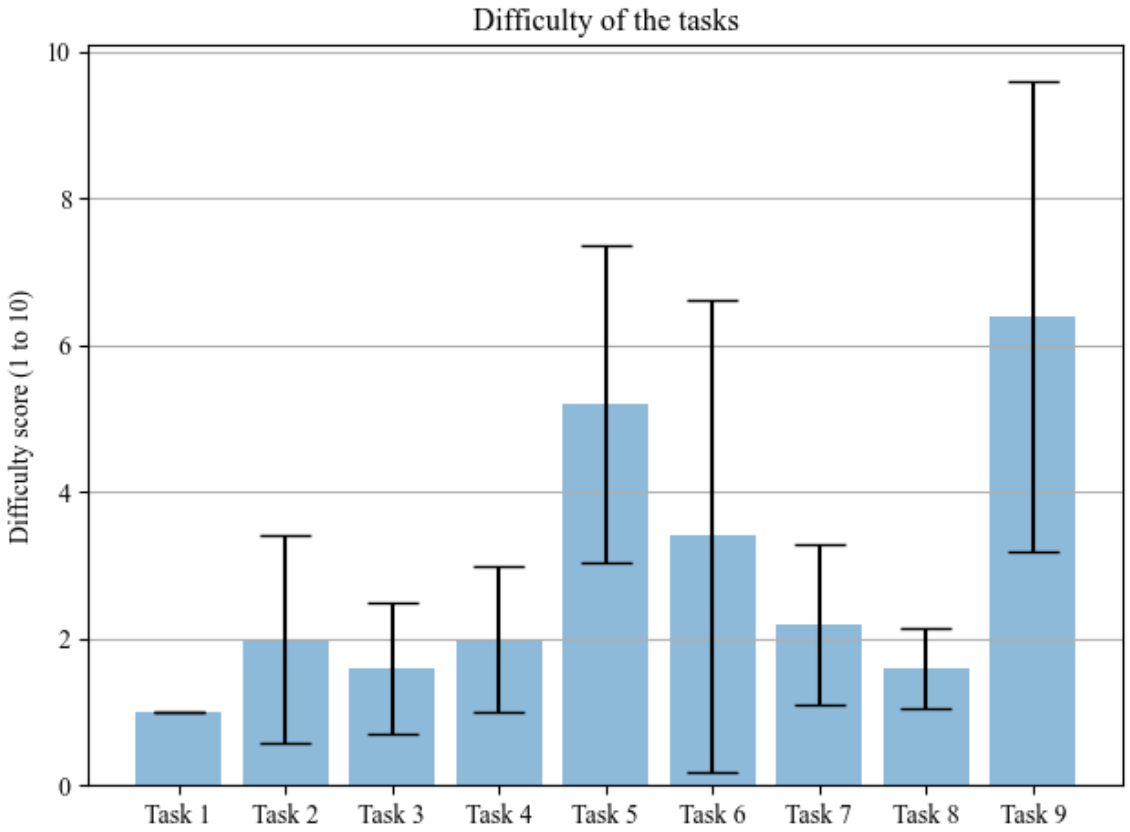

Figure 49. Average rated difficulty of the tasks.

All the subjects reported the difficulty of task 5 to be high. Three people rated it with four, one with five, and one with nine. The instructions for the task were not clear enough. One person noted that understanding the anchor placement took more effort than other tasks. Moreover, the task failed in teaching the logic of anchors to two of the users. This is the reason why task 9 was rated as difficult.

Task 9 was the most difficult task. One person rated it with two, three with five or more, and one with ten which signifies that they did not finish the task. Two of the users rated it high due to not understanding the anchors. The task required the users to move eight anchors in total. One of them relied on trial and error to finish the task. The other did not manage to move the anchors from one face to another in the correct manner. One user was frustrated due to the clunkiness of moving the anchors; rotating and hiding the shapes eight times in total takes too much time. On the other hand, the person who rated the task with two found it simple and even fun.

While the cutting tools were picked up quickly by the users, some aspects of them are not intuitive. The subjects reported that there were too many hot-keys, and canceling cuts needs to be made clearer. The user experience would be boosted by adding the undo button. The users wrote that they had to redo some tasks multiple times, in which case they had to delete the rule before retracing their steps. The undo button would have solved the problem with one click. Finally, the anchor placement is not intuitive to use. The rotation of the shape must be visualized better to make the logic of the anchors intuitive. The movement procedures of the anchor should be redesigned completely.

4.2 System Usability Scale

The second part of the questionnaire asked the users to score ten sentences with one of five responses, ranging from “Strongly Agree” to “Strongly Disagree”. The scores can be compiled to give a single number on the System Usability Scale (SUS). This is a method to assess the usability of systems developed by Brooke [8]. Usability is subjective. However, SUS has still been found useful when trying to determine how easily the tool can be used [8].

The editor has a SUS score of 39.5 out of 100. It should be noted that this number is not a percentage. This number can only be used to compare the usability of the editor with the usability of another tool. Bangor et al. [9] determined what individual scores on average intuitively mean. According to their scale, the score 39.5 is above “Poor” but well below

"OK". For it to be "Good", the SUS score needs to be above 70. Therefore, from the expert opinion, the editor is not intuitive to use. However, the editor provides enough tools to create varied architecture.

4.3 Demonstration

A simple grammar was created for demonstration purposes. As seen in Figure 50, the grammar can create one- and two-story buildings. The houses are hollow, and the ground floor always has a door, so they are enterable. Walls have windows with four windowpanes. The roof can be either gable or pyramid style. In the figure, two ground floor types are also demonstrated. One type has a wide door and the wall with the door does not have windows. The other has a smaller door but has windows next to it.

This grammar has 13 non-terminal symbols and 16 rules defined on them. 10 of the non-terminal symbols only have one rule defined for them. The other three make the grammar non-deterministic, hence there is variety in the output. The grammar is used in a demo Godot game, where three `GrammarInstance3D` nodes have been placed next to each other. The user can move around with WASD controls and rotate the camera with a mouse. Figure 50 and Figure 51 are from that application. Pressing "Enter" gives the grammar states new random seeds, which regenerates the houses. To exit the demo, "Esc" must be pressed. The grammar and the application are included in the accompanying files of this thesis.

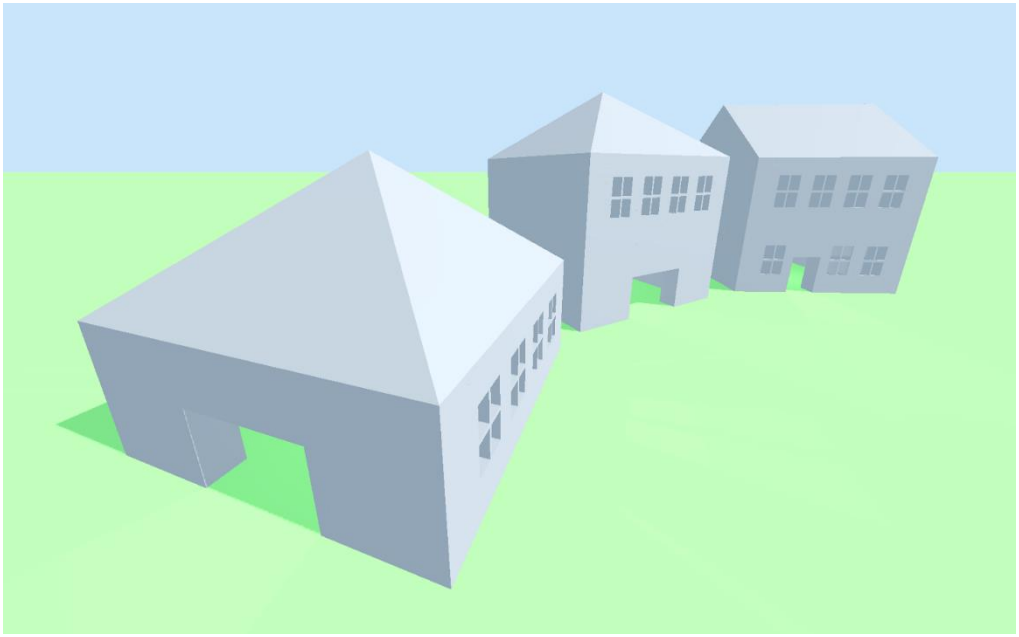

Figure 50. Three different generations of the same grammar.

The grammar was created in one hour, demonstrating that it is possible to create a variety of simple houses in a short period of time. As more rules are added, the variety increases. However, the current implementation is not optimized well. When the generation of houses is triggered, there is a noticeable lag spike. The lag spike is more apparent when the camera is moving during the generation. The performance is one of the topics that must be handled in future work.

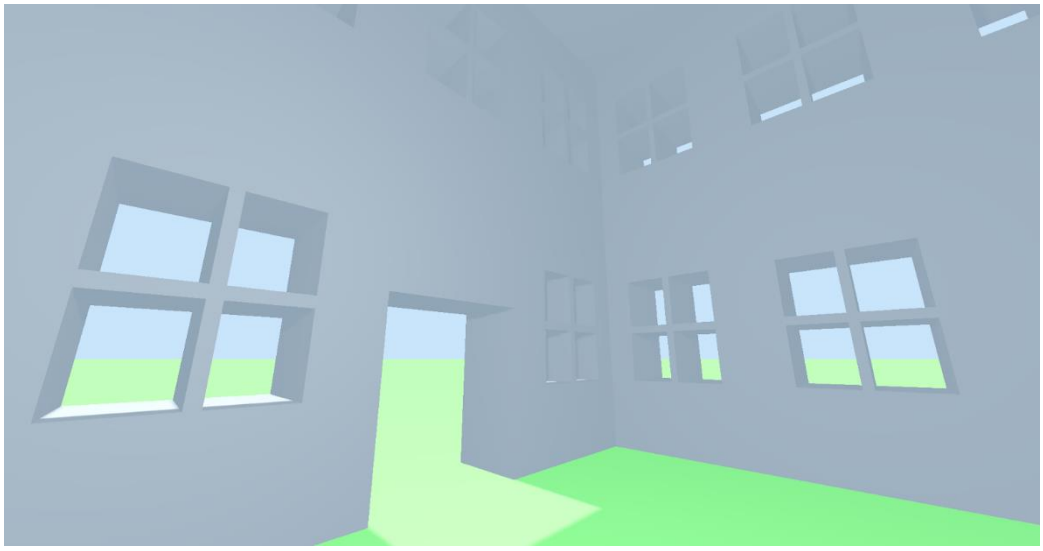

Figure 51. Inside one of the generated buildings.

4.4 Future Improvements

Godot scripting language is not performant enough to withstand the generation requests without creating some lag. Possible workarounds are to do the generation on a separate thread or spread it between frames. To improve the performance in any scenario, the interpreter should be rewritten in C++. This would also make the interpreter portable to other game engines. Other expensive operations, such as triangulation and mesh generation should also be in C++ for better performance.

The main thing to improve in the editor is the intuitiveness. The current anchor placement system is overly abstract. It is cumbersome to use and has too many different possible operations. Therefore, the system should be redesigned entirely. One possible avenue is to visualize the orientation of the sub-shape with geometry. The rules defined on the assigned symbol could be used on the sub-shape to give the user a review of what the sub-shape would turn into during interpretation. During the user testing, a lot of feedback was gathered about other systems as well, which could be improved.

The editor would also benefit from more cutting tools. During the definition of the grammar for the demonstration, it became apparent that some lacking functions would have made the definition process much easier. One possible new tool could be a variation of the “Multi Cut” tool, which sub-divides a shape into three parts, where the two sub-shapes around the middle one would be equal. This tool would be useful when adding margin regions to sub-shapes. For example, the demonstration grammar produces windows with equal margins on both sides.

An important part of meshes is coloring them with textures. This is currently impossible, because UV coordinates are not calculated for the vertices of the shapes. In fact, it would be difficult to use a single texture to color the whole building with one texture, because the building is different every time the seed changes. Instead, a material system should be implemented, where the user specifies a tiled material for each face. Then, when using the plugin, the materials can be specified in GrammarMesh or GrammarInstance3D.

Finally, global-space coordinates should be considered during the interpretation. For example, this would make it possible to specify a minimum and maximum distance between a face and the cutting plane. Walls could be created with constant width, regardless of the initial shape’s scale. For this, the control grammar and attribute matching from Wonka’s (2003) work should be implemented.

Overall, the results are good. While the main software components were implemented successfully and offer a variety of tools for grammar design, there are still limitations. Some users were comfortable with the editor, while others struggled to complete the task. This suggests that there is still much room for improvement. The grammar created for demonstration purposes shows that user defined shape grammar can be used in real time. However, tools were lacking which would have made the design process simpler. The future of this software can go in many directions, some of which have been discussed. At the current stage, the tools provided by this thesis are only suitable for experts who have experience with shape grammar. To make it more accessible, more work needs to be done.

5 Conclusion

Formal grammar was defined, which is the basis for the shape grammar. This thesis was basing its methods on split grammar, which is a limited version of shape grammar. The limitations make the split grammar more intuitive for users and much easier to implement in the software.

The goal of this thesis was to develop tools that allow intuitive creation of 3D assets with split grammar in Godot. Two software components were developed. The main software component, the grammar editor, allows users to use 3D tools to define split grammar rules. This approach differs from the norm in that most common shape grammar tools define grammar rules in text form. Main aspects of the editor were described, such as symbols, rules, starting shapes, and anchors. Lastly, used algorithms and data structures for vertex ordering, cutting, and compilation were detailed.

The secondary software component, the Plugin, lets users use the grammar that they defined with the editor in Godot 4 games. Procedural generation software, such as Esri CityEngine, only generates structures inside the software. With this software, it is impossible to generate shapes during the run-time of a game. The plugin is the solution to this problem, because it allows run-time generation of shapes outside of the grammar editor. This provides unique value to the author's knowledge.

The editor was tested on users. From the gathered feedback, the editor was proven to be a good prototype, and the areas which need more improvement in the future were specified. From the user testing it is apparent that the editor in the current state is only meant for people with previous experience with shape grammar. The tools must be made more intuitive.

References

[1] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios Yannakakis, "What is Procedural Content Generation? Mario on the borderline," in *Proceedings of the 2nd Workshop on Procedural Generation in Games*, 2011.

[2] Noor Shaker, Julian Togelius, and Mark J. Nelson, *Procedural Content Generation in Games*. Switzerland: Springer International Publishing Switzerland, 2016.

[3] George Stiny, "Introduction to shape and shape grammars," *Environment and Planning B*, vol. 7, pp. 343-351, 1980.

[4] Tzu-Chieh Kurt Hong and Anthanassios Economou, "Five Criteria for Shape Grammar Interpreters," in *Design Computing and Cognition'20*, 2022, pp. 191-207.

[5] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky, "Instant Architecture," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 669-677, 2003.

[6] Mathias Plans, "Procedural Generation of Unique Buildings," University of Tartu, Tartu, BSc Thesis 2021.

[7] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool, "Procedural Modeling of Buildings," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 614-623, July 2006.

[8] John Brooke, "SUS: A quick and dirty usability scale.," *Usability Eval. Ind.*, vol. 189, 1995.

[9] Aaron Bangor, Philip Kortum, and James Miller, "Determinign What Individual SUS Scores Mean: Adding an Adjective Rating Scale," *Journal of User Experience*, vol. 4, no. 3, pp. 114-123, May 2009.

Appendix

I. Plugin User Manual

First, the plugin has to be installed. This is done by either getting it from the Asset Library, Git repository, or accompanying files. It has to be made sure that the directory with plugin scripts is in `res://addons` folder.

Next, the plugin needs to be activated by going to Project > Project Settings… > Plugins and checking the "Enable" box for "Shape Grammar Interpreter".

Next, a scene must be created. In this scene, a child node can be added to the root node, as seen in Figure 52

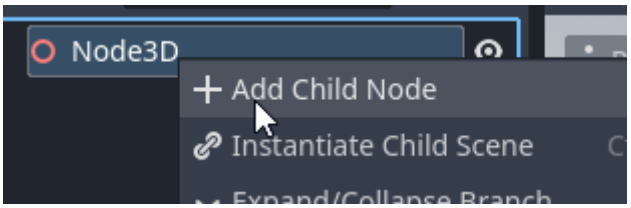

Figure 52 Create a child node.

This child node has to be with `GrammarInstance3D` type, as seen in Figure 53.

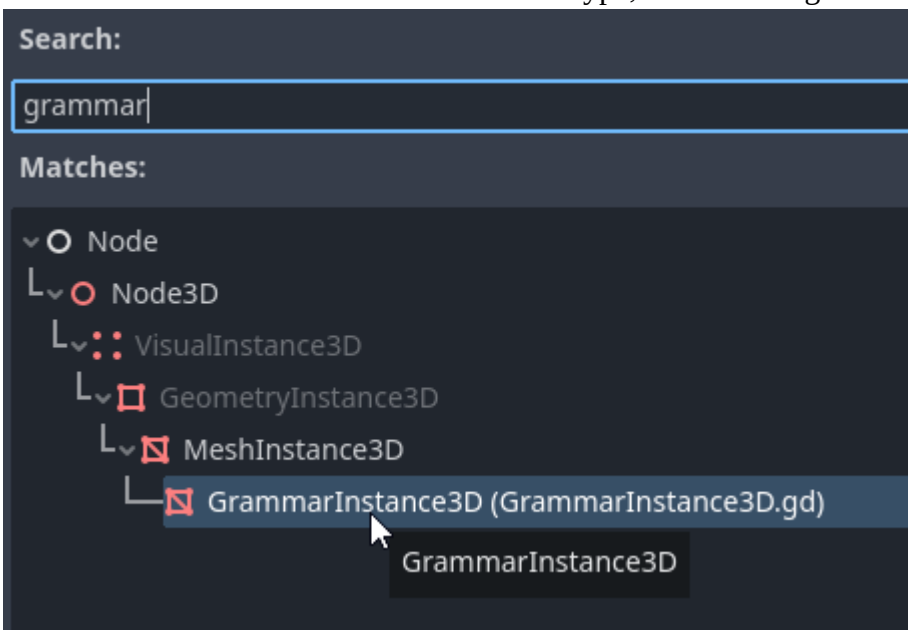

Figure 53. `GrammarInstance3D` node from the plugin.

In the inspector of the grammar instance, a new grammar state resource has to be created. This can be done by clicking on the "Grammar State" field and then "New GrammarState". Then, after expanding the resource, a grammar file can be uploaded, as seen in Figure 56.

After clicking on the "Grammar" field, there is an opportunity to load this resource from the disk. In Figure 55, "house.grammar" was chosen as the grammar.

The final step is to change the number of generations and the starting symbol. With the configuration in Figure 56, a house was generated. The “house.grammar” file is included in the acompanying files.

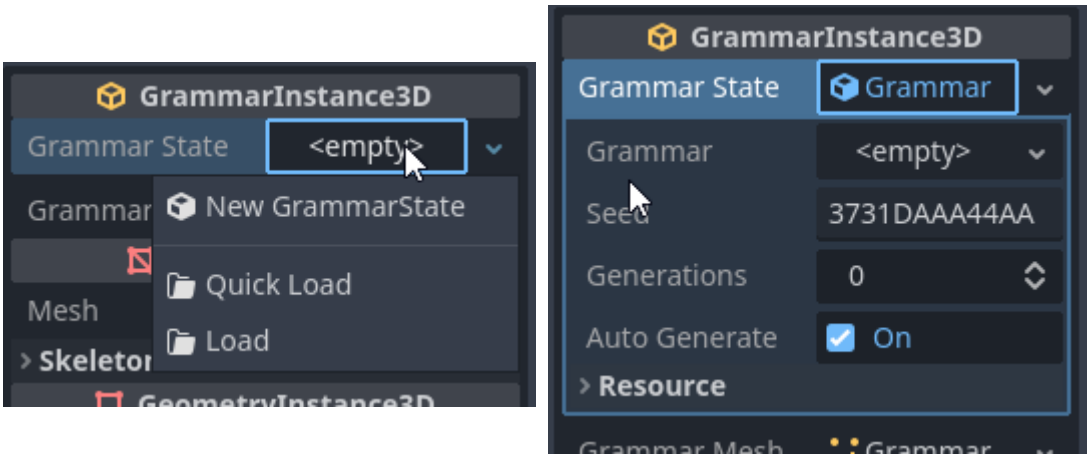

Figure 54. Grammar State creation

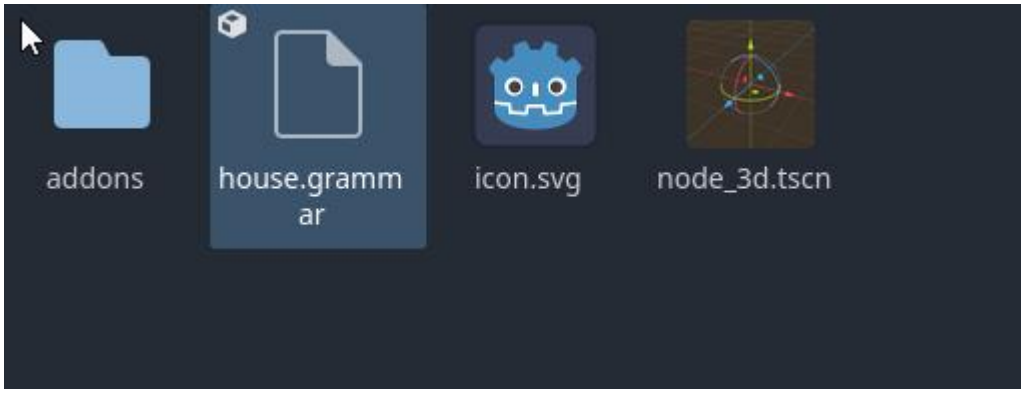

Figure 55. Select a grammar file.

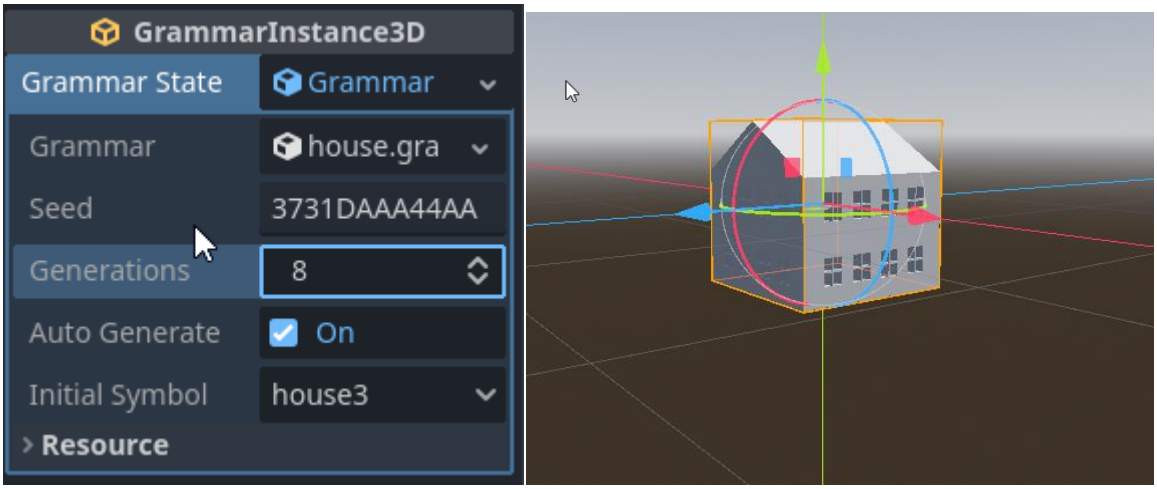

Figure 56. Starting with “house3” symbol
and automatically generating 8
generations produces a house.

II. Accompanying files

The accompanying files are all in a directory “sge-files”. In “sge-files/demo”, the demonstration project’s windows build is located. By activating the executable, the demonstration program should activate. The Godot 4 project is also in this directory, in “demo-project.zip” archive.

In “sge-files/editor-final”, the latest build for the editor is located. The Godot 4 project is also in this directory, as a git repository. The editor is also accessible from GitHub with the following link: https://github.com/mathiasplans/grammar-editor.

In “sge-files/editor-test”, the web build that was used to test users is in an archive. The web build is also accessible from the following link: https://mathiasplans.itch.io/shape-grammar-editor.

In “sge-files/house-grammar”, the demonstration grammar’s grammar file for the plugin and the save file for the editor is located.

In “sge-files/plugin-repository”, the plugin code is located. The plugin is also on GitHub: https://github.com/mathiasplans/shape-grammar-plugin and Godot Asset Library: https://godotengine.org/asset-library/asset/1871.

The “sge-files” also has two files. A PDF file “questionnaire.pdf” has the form that the testers used when giving the feedback. This PDF is constructed from a Google Form document. Another file is “user_data.csv”. This holds all the responses the users gave. No sensitive data was gathered during the testing process.

III. License

Non-exclusive licence to reproduce thesis and make thesis public.

I, **Mathias Plans**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

    1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

    1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Shape Grammar Editor,

supervised by Jaanus Jaggo,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **09.05.2023**