Interactive Graphical Shape Grammar

MICA LEWIS

This paper describes the design and implementation of an online app that allows users to define the rules of a shape grammar through a graphical interface. Its purpose is to demonstrate the concept of shape grammar using volumetric solids in an easily accessible online environment. Through a system of replacement rules, it allows the user to generate complex shapes from repeatedly applying a set of basic rules. A survey of users of varying familiarity with computer science topics, demonstrates it to be ineffective as a teaching tool on its own or as a commercial tool. It instead exists as a novel realization of its underlying concepts, shape grammars, and solid modelling, independent of a real world use case.

Additional Key Words and Phrases: Phrase Structure Grammar, Shape Grammar, Octrees, Volumetric Solids

ACM Reference Format:
Mica Lewis. 2019. Interactive Graphical Shape Grammar. (May 2019), 13 pages.

1 BACKGROUND

Shape grammars are a novel application of grammar that forgo the traditional use of tokens in 1-dimensional arrays of memory. They instead rely on abstract forms in 2 or 3 dimensional space. The abstract forms in question are often represented as points and lines in 2 dimensions, or as vertices, edges and faces in 3 dimensions. This paper attempts to outline a shape grammar relying on 3 dimensional volumetric solids. Solids are defined as the closed set of set of points under some boundary. Octrees can be used to approximate solids as they define regions of space in infinitely increasing granularity. A concession in the design specified in this paper is to rely solely on octrees as a representation of solids rather than use them as an approximation for more complex solids. This decreases the complexity of the design significantly, and makes interaction with the shape grammars more intuitive for the user.

The original definition of shape grammar was provided by Stiny and Gips in the 1971 paper "Shape Grammars and the Generative Specification of Painting and Sculpture"[4]. They arrived at this definition using phrase structure grammars as a template:

Definition. A *shape grammar*(SG) is a is a 4-tuple: $SG = (V_T, V_M, R, I)$ where

(1) V_T is a finite set of shapes
(2) V_M is a finite set of shapes such that ${V_T}^* \cap V_M = \emptyset$
(3) R is a finite set of ordered pairs (u, v) such that u is a shape consisting of an element of ${V_T}^*$ combined with an element of V_M and v is a shape consisting of (A) the element of V_T* contained in u or (B) the element of ${V_T}^*$ contained in u combined with an element of V_M or (C) the element of ${V_T}^*$ contained in u combined with an additional element of ${V_T}^*$ and an element of V_M.
(4) I is a shape consisting of elements of ${V_T}^*$ and V_M

This definition satisfies objective of the paper, a generative grammar for the purpose of generating paintings or sculpture. It can be used to generate interesting self similar shapes by applying shape rules by hand to a physical canvas. Possibly as a result of this method, certain limitations were put on the generation process and on the definition of shapes themselves. First, the shape rules can only add terminals. The right side of a rule may remove the initial non-terminal, add non-terminals, add terminals, but never remove any part of the larger

Author's address: Mica Lewis, mxl7287@rit.edu.

shape except for the single non-terminal matched from the left side of the rule. When working with shapes made of ink, identifying which subsections of a shape need to be removed and removing them can be difficult. Second, all intuitive transformations on a shape are allowed. When matching the left side of a rule to a sub-shape, it can be translated, rotated, scaled and mirrored. This is easy enough for a human to do with their eyes, but it would difficult for a machine to identify these lines if they have been moved, rotated, scaled and/or flipped without a consistent method of matching shapes. Third, there is no rigorous mathematical model underpinning the shapes and lines used in the paper. This is partially responsible for the first two limitations. Removing one shape from another can't be achieved without identifying what exactly is being removed. Identifying one shape within another is only intuitive to humans. Any automated approach would require an exact data structure or a computer vision algorithm. Fourth, when designing an example shape grammar, the authors set V_T to be a set containing one straight line, allowing ${V_T}^*$ to be any shape composed of straight lines. It follows that V_M can only contain shapes comprised of curved lines in order to be distinct from ${V_T}^*$. Although this is a helpful visual shorthand for which shapes are terminal and which are non-terminal, it restricts all completed shapes in the language defined by the grammar, to be linear.

Another definition was introduced by Stiny in the 1980 paper "Introduction to shape and shape grammars"[3] which remedied the limitations in the first definition. The paper begins by defining shape as "a limited arrangement of straight lines defined in a cartesian coordinate system with real axes and an associated euclidean metric". Critical to this is the concept of labeled shapes and maximal lines. Labeled shapes address the issue of ambiguity between shapes, particularly between shapes made of terminals and non-terminals, by assigning each shape a "label" that exists independent of the physical space the shape exists in. Once a label is attached to a shape, it can be differentiated from another shape with a different label even if they share geometry that would, under some transformation, be identical. When combining two shapes under this new definition, the set of lines within a shape must be reduced by combining all collinear lines. Once they are in reduced form, they are considered to be "maximal". By this method, each shape can be represented by a unique set of lines. There is no ambiguity to which arrangement of lines can form a shape. A shape under this definition could also be thought of as the set of all points on a set of lines, rather than the lines themselves. In these distinctions, arises a clear set of operations that can be performed on shapes. No longer are shapes simply forms to be draw on a canvas, they are mathematical structures on which operations can be performed.

Four useful operations that can be performed on a shape defined by maximal lines are shape union, shape intersection, shape difference and subshape. The shape union of two shapes ($s_1 \cup s_2$) is the set union of their constituent lines. When shapes are represented by their maximal lines, any two collinear lines put together will be represented as one singular line. The shape intersection of two shapes ($s_1 \cap s_2$) is the set intersection of points within their constituent lines. The intersection of two overlapping lines is the section of line shared by both lines. Shape difference ($s_1 - s_2$) follows from this notion of intersection. The difference of two shapes is the set difference of all line in the two shapes. The difference between two collinear lines is the section of one line not contained in the other. One shape is said to be a "subshape" of another ($s_1 \subseteq s_2$) if the lines of the first shape exist within the lines of second. This is identical to the intuitive method of identifying one shape within another from the 1971 definition, but now it can be formalized like the other operations. In these operations, we can begin to see shapes not as a set of lines, but as the set of all points on those lines.

Stiny goes on from defining boolean operations on shapes to defining a new method of applying shape rules. Rather than simply adding one shape on top of another, only altering the non-terminal, one shape can be subtracted from and another can be added to a larger shape. This allows for a simpler style of replacement where the left hand side of a rule is removed from a shape and in its place the right hand side of that rule is added. It is worth noting that under these new specifications shape grammars can simulate Turing machines[5], bringing them beyond the generative capabilities of phrase structure grammars. To account for the change in scale, rotation or position of a subshape, transformation is represented as $\tau(s)$. We can now reduce the three

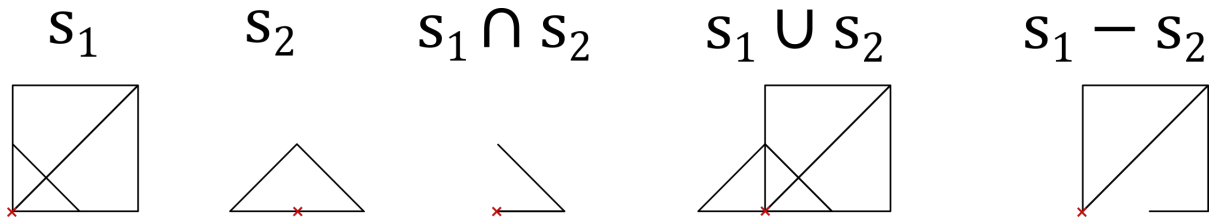

Fig. 1. Boolean operations on shapes. Assume that the shapes require no transformations before they are matched and the red x represents the origin. Note that overlapping lines do not affect each other unless they are collinear.

separate cases for the replacement rule (u, v) in a shape s (A, B and C from the first definition of shape grammars), to one case: if $u \subseteq s$ then $[s - \tau(u)] + \tau(v)$.

Describing the rules of a shape grammar in terms of boolean operations not only clarifies their application, but opens shape grammars up to other representations of shape that rely on the same operations. Another visual representation of shape that relies on this terminology is solid modelling. With the introduction of maximal lines, we can begin to see shapes not just as sets of objects but as closed sets of points. Just as maximal lines can be interpreted as the closed set of all points between two points, solids are the set of all points within a closed surface. It follows that solids are a good candidate to replace lines as the basis of shape in a shape grammar. Solids are often used solely because of their similarity to boolean operations on sets.

1.1 Related Work

The 2006 paper "Procedural Modeling of Buildings"[2] uses solids, or mass models as they are referred to in the paper, in a novel shape grammar implementation: CGA Shape. This approach deviates significantly from the original 4-tuple in its practical usage. It is a robust tool for generating architecture and less of a theoretical framework. Rather than use one type of rule application, CGA Shape opts for several classes of rules. The most basic of these rules adds 3D primitives as volumes, similar to the union of lines in the 1980 definition. Other rules involve subdividing the set, which is in strong opposition to the concept of maximal lines. Not only can one continuous solid be represented as several separate solids, but new "split rules" exist for the sole purpose of subdividing a solid. Another deviation is the omission of subtraction. Like the original 1971 paper, this grammar is additive and does not subtract one shape from another. However, the generative power of this grammar can not be understated. Solids provide a robust foundation for creating the 3D forms generated by CGA Shape.

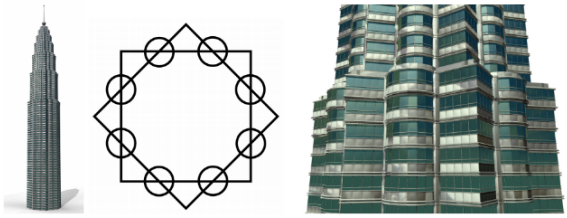

Fig. 2. CGA shape reconstruction of a Petronas Tower. The mass model of the tower (left) and the footprint (middle) reveal the elementary assembling of cubes and cylinders. Right: The same facl̀ğade rule has been applied onto the different type of solids. Image courtesy of Muller 2006[2] .

A novel approach to representing volumetric solids, that preserves boolean operations, is explored in the paper "Verified Adaptive Octree Representations of Constructive Solid Geometry Objects"[1]. Like "Procedural Modeling of buildings", this paper is about a tool for creating shapes defined in a text based language. It uses primitive shapes and boolean operations (including the union and subtraction critical to the 1980 definition of shape grammars) to describe complex 3D solids, or "Constructive solid geometry". What is noteworthy about this approach is that after the 3D forms are described, they are approximated as octrees, a tree based data structure

for storing 3D data. In an octree each node is represented as a cube, and has 8 child nodes, one for each of the 8 cubes that a cube can be subdivided into. They are often used for searching 3D space efficiently, being analogous to binary search trees, storing data in their leaf nodes. This data structure can also be used as a representation of volumetric solids. Rather than acting as a container for data like in the case of a search tree, an octree can define all points within the bounds of a node. This paper uses octrees to approximate all points within the solid defined by 3D primitives and boolean operators. The combinations of shapes provides an easy method of testing a point's membership within the solid. When approximating the solid this test can be used to create a shell of increasing granularity around the underlying geometry.

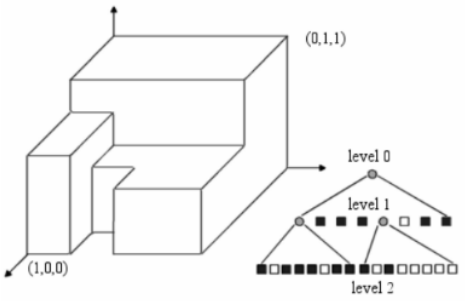

Fig. 3. A 3-dimensional of an octree and the corresponding tree data structure. Image curtosey of Eva Dyllong and Cornelius Grimm[1]

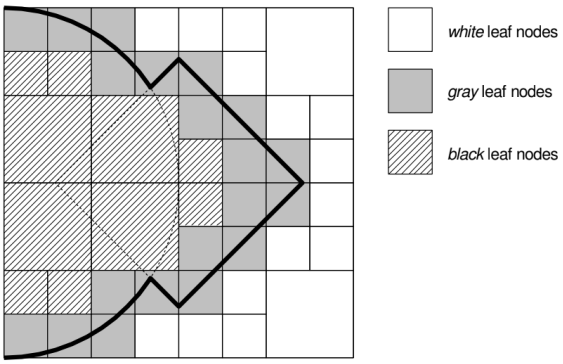

Fig. 4. A 2-dimensional solid and it's quad tree representation (analogous to its 3-dimensional counterpart). Image curtosey of Eva Dyllong and Cornelius Grimm[1]

2 DESIGN OVERVIEW

This paper attempts to outline an graphical, interactive implementation of shape grammar based purely on visual manipulation of shapes that preserves the 1980 definition of shape grammar but with Octree based solids. Some concessions will have to be made to that definition to take full advantage of the new data structure. To simplify the design as well as the interactive experience, solids are represented only as nodes on octrees and not

Index of child node	$x_a > x_m$	$y_a > y_m$	$z_a > z_m$
0	false	false	false
1	false	false	true
2	false	true	false
3	false	true	true
4	true	false	false
5	true	false	true
6	true	true	false
7	true	true	true

Table 1. A means of indexing the child nodes of an octree where conditions correspond to binary representation of indices.

as approximated shapes. The cubes that represent leaf nodes of the tree will be called voxels. They will define all points inside the closed boundary of the solid. Non-terminals and terminals will be represented as voxels. Like in the 1980 definition, voxels are added, removed, and reduced just as lines were. The operators performed on them are consistent with notion that they are a continuous closed set of 3-dimensional points. Addition is equivalent to set union; subtraction is equivalent to set subtraction. The rule of replacement (if $u \subseteq s$ then $[s - \tau(u)] + \tau(v)$) can be preserved and re-applied from lines in 2 dimensional space, to solids in 3 dimensional space.

2.1 Octree Data Scructure

The theoretically simplistic boolean operations can be difficult to represent using standard methods of 3-dimensional geometry. In order to be rendered interactively, these 3 dimensional solids must be reduced to vertices and triangles. Subtracting one continuously transformable shape from another and displaying geometry would inevitably require retopologizing. Octrees limit the shapes used to only cubes, and limit those cubes to positions on a grid with step size equal to the length of the cube. They provide elegant solutions to the problems of boolean operations without incurring significant cost to the storage of geometry. A fixed size 3-dimensional grid of voxels would provide even more simplistic solutions, but they don't fill shape grammar's critical need for a flexible scale. Large arrangements of fixed size voxels would incur extreme memory costs, given the n^3 nature of volume. Arrangements of voxels smaller than a fixed grid size can not exist. In a tree, all operations can be executed recursively and have the same effect regardless of scale.

Addition and subtraction by this method has many close corollaries. Adding to an octree is the same as adding to any ordered tree, but inputs are compared along three dimensions instead of one. Placing a new voxel involves traversing down the tree according to which child node the new voxel falls under. The location is compared to the midpoint of the current voxel in the recursion. In a octree designed for searching and storing data, the recursion would stop if some standard of uniqueness was met. In this case the level at which the recursion stops is specified at the beginning of the function by the size or level that the new voxel exists at. Voxels are geometrically identical to the regions defined by an octree's nodes (both are cubes), so level in the tree and scale of the voxel can be used interchangeably. Subtraction of a voxel from a shape works by the same general principal. The tree is traversed based on the location and level of subtraction. If a voxel at that location and level is found, it is removed. If the location falls within a voxel and exists at a lower level than that voxel, the larger voxel must be "cracked". In the process of cracking a voxel it is turned into a branch node and filled with leaf nodes (new voxels) at every index. The function then recurses on the appropriate child node in the same way addition would. The recursion continues cracking voxels until the given level is reached. Non-terminal voxels will not be cracked. The singularity of a non terminal is important for the process of matching and that would be lost if they could be subdivided. Subdividing a node into parts, picking a child node based on location, and recursing, requires

the same process in both addition and subtraction. Addition divides empty space to add filled space; subtraction divides filled space to add empty space.

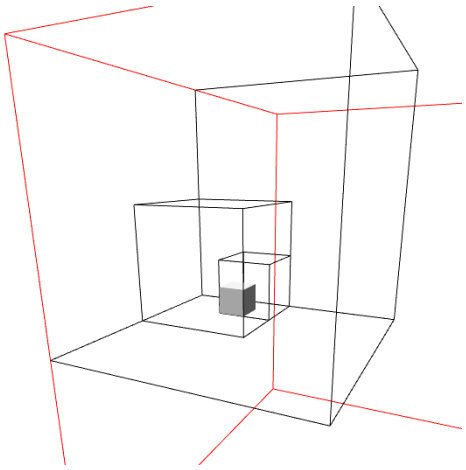

Fig. 5. A voxel added to empty space with each of parent nodes outlined in black and the root node outlined in red.

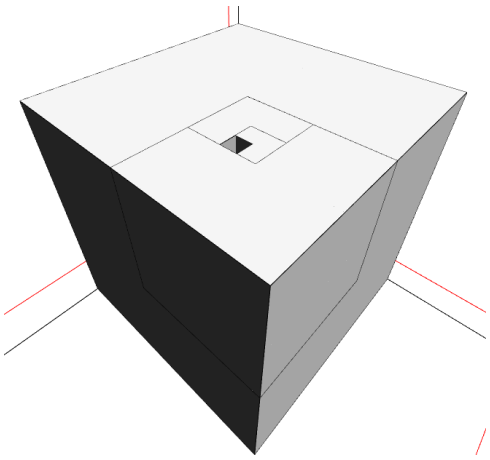

Fig. 6. A voxel subtracted from a larger voxel. Each branch node created by cracking is outlined in black.

Subdividing risks violating the principle of maximality that has guided the design of this grammar. In the case of cracking, the voxels remain maximal because any cracked voxel must remove a section, as cracking only occurs in that case. If a section is missing from a cube then the voxels inside of it uniquely describe the remaining space. However, the ambiguous case remains, of 8 voxels placed together under one node to cover a closed set identical to that of a single block. A method of merging after each addition is introduced to prevent this case. Similar to Stiny's reduction of ambiguous lines, if a voxel is added to a shape the parent branch must be checked to see if it has been completed by this new addition. This process must then recurse to the next parent and check

it for the same condition. If a large cube missing only one voxel several levels below it, then the addition of the missing voxel must merge each level in between until the whole cube is represented as one voxel. The act of subtraction must include a complementary operation. Non-terminals can be merged for the same reason that they can not be cracked, to preserve their singularity. In general any cluster of non terminals impairs their core function; to act as a single guiding point to match larger shapes. If there is ever an empty branch node, it must be cleared in the same way. Each parent node is removed until the recursion reaches the root node or a non-empty branch. The last function necessary for the replacement rule is the subset detection. When checking a shape for the existence of a voxel, the same method of recursion is used to traverse the tree. If a queried location is inside of a larger voxel, then the query returns with a positive result. In keeping with the notion that a voxel defines a closed set of points. Any set of closed points inside of a voxel is considered a subset.

With the functions of manipulating shapes based on individual voxels in place, we can apply these functions in using whole shapes under different transformations. Voxel data from one shape can be transformed by translation, scale and rotation with some limitations. Translation can only occur on the by steps the length of the largest voxel in a shape. If any steps smaller than the length of a voxel are made, then it would fall in between two branches in the tree. A voxel missaligned in this way would be impossible to represent using the methods specified up to this point. Scale can only happen by powers of two given that each edge of a voxel can only be divided by a power of two. Transforming the location and level of each voxel in one shape gives a set of inputs to be either added or subtracted to another shape.

To account for these limitations on transformation, certain restrictions must be applied to shapes contained on both the left and right sides of a shape rule. In designing these restrictions a compromise must be made between the simplicity of the design, and flexibility of shape grammars. On the left, a shape must be made that can be easily identified in a larger shape. To achieve this, left hand shapes must only have one non-terminal shape. The context around a non terminal can be checked for a match trivially, but checking an unrestricted shape with multiple non-terminals would introduced too much complexity to the system. Restricting the left hand side to exactly one non-terminal voxel, a guiding point of reference for the rest of the shape is created. This is useful for the case of a potential misalignment. If a non terminal were to be replaced by a larger voxel, one step in the grid of the non-terminal, would result in half a step of the larger voxel's grid, thus misaligning it. To avoid this while making the most of the aforementioned compromise, all shapes contained within rules contain no voxels larger than the left hand's non-terminal. All merging will be capped at that level. Larger voxels can still be produced by these rules, but only at the rule's point of application to a larger shape, through the merging function. Merging also applies to non-terminals as this is critical to making recursively upward scaling rules. With these restrictions in place, an initial shape of terminals and non-terminals can have these restricted rules applied to it on any set of transformations.

2.2 UI Design

To make this system of shape grammars as accessible as it is designed to be, a simplistic UI is required. It must reflect the underlying design decisions in a way that is intuitive. From the original 4-tuple, only the last two elements are relevant to the experience of creating the shape grammar. This requires multiple UI components to fully realize. To represent each rule within a shape grammar a scrollable list will suffice. Visualizing what is inside the rule can't be done inside a single list element and be easily interacted with, due to the space needed on a screen to full depict a 3D shape. Fully representing a shape to a user requires a large view port and typically some level of interactivity. The shapes on either side of a rule must not only be viewed by the user, but also edited. Displaying the shapes of only one rule at a time gives the user a clear picture of what the rule they are making does: replace the shape on the left side, with the shape on the right side. From the list of rules, one rule at a time can be selected to be displayed in two regions of the screen large enough to perform the basic functions of

shape manipulation. Also contained in the UI is the central shape that is being acted on by the rules. This would be the initial shape in the 4-tuple, distinguishing it as initial would require creating separate states of the UI for the initial state and the active state. This distinction does little to further the user's understanding of the system and is not included here. Each rule can be applied to some matching subset in the active shape (what would have initial shape under the original definition) at any point in the creation of rules or the active shape itself. Given the time complexity of comparing every voxel in one shape to every voxel in another shape, required to identify a match between contexts, the user must request a list of matches every time they wish to apply one. A button to trigger this matching process will generate a list of all instances of the top shape within the bottom shape. An active element of this list will be highlighted within the active shape. The user may then select an index within this list and apply it.

The displays containing the shapes only require the a small set of interactive functions, which can be achieved with little to no complicated overlay. Functions necessary to viewing a 3d object for the purpose of manipulation are: Orbiting, to view the shape from any angle while keeping the camera upright, Zooming, to move the camera closer to details, and Panning, to translate the camera. Adding and removing shapes can be achieved with a cursor in the 3D environment that highlights the region that will either become a voxel, or remove a voxel. In the octree this cursor needs to exist in a specific level, so two inputs, either buttons or keyboard inputs, must exist to move the cursor level up or down. The possible functions that can occur on a mouse click are adding terminals, adding non-terminals, and removing. Control of these can also occur in keyboard or UI button inputs. There is little to no context on elements of the UI but the hope is that if simplicity of the design is implemented, little to no context is needed to understand the tool.

3 IMPLEMENTATION

The implementation of this paper uses web based tools with Java style classes and structs. A web based approach was chosen to achieve the goal of accessibility. The first step of satisfying this criteria is ensuring anyone can access the product it with ease. JavaScript and HTML where the only languages used and they provide some other benefits on top of portability. UI can be quickly and easily constructed using HTML and the bootstrap library, allowing more time to be spent on development. The 3D UI elements are rendered with three.js, an open source JavaScript 3D library based in WebGL. JavaScript itself provides modest benefits the the creation of the source code. The recent introduction of Java style classes to JavaScript allows Java's rigidity in designing data structures to be merged with JavaScript's functional, UI driven aspects. Event handling would be a much bigger task in another system when in this case it should not be the focus of this project. The octree data structure and the necessary functions for handling shape grammars are best suited to an environment with type checking and manual control of pointers to avoid mistakes. JavaScript's class syntax maintains organization well enough that ambiguous types are not a severe problem.

The class structure of the program contains 4 major classes, Voxel, Shape, Display and Rule. Voxel represents a single node within the octree. It is used primarily as a struct with helper functions. Its functions include: finding the correct index of a child voxel relative to the midpoint, testing if a point is in its bounds, and finding the position of a child given an index. The Shape class handles the root voxel, as well as all functions essential to the octree such as adding a voxel, removing a voxel, cracking, merging, clearing, and expanding the root if a new voxel is placed out of bounds. The Display class handles all of the UI interaction with three.js. It contains a single shape class which the user manipulates through the UI functions of the display class. The Rule class also acts as a struct containing a left hand and right hand shape to be swapped in and out of the left and right displays.

The other elements of the UI that are not contained within a display, are those relevent to selecting rules and matches of rules. Each of these is represented as a button with associated event functions when those buttons are clicked. The button that finds matches is responsible for the code that calculates matches based on the location of

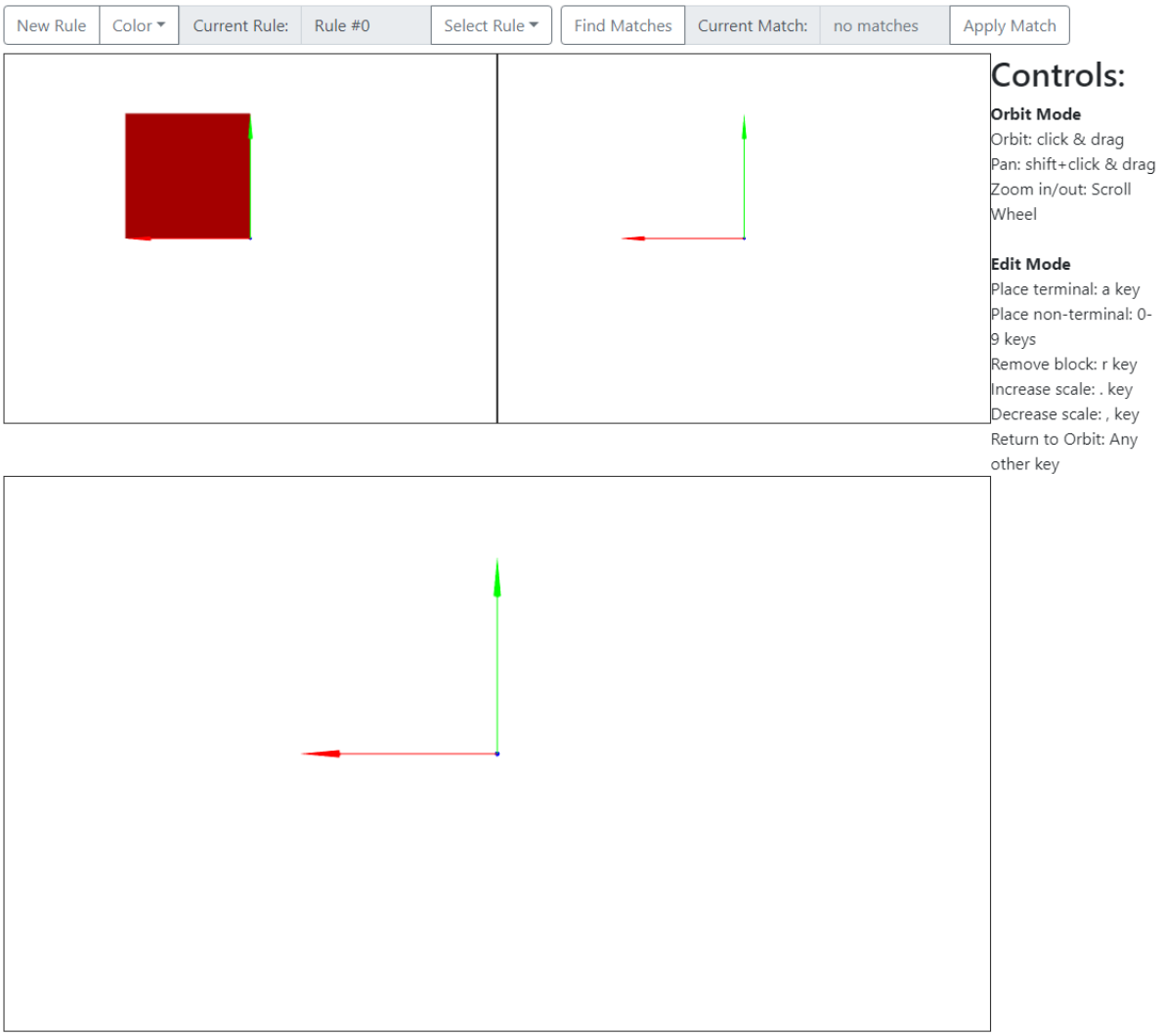

Fig. 7. A completed implementation of the product's Graphical User Interface

non-terminals, and saving a list of those matches as represented by matrix transformations of the matched shape to its subshape in the active shape. These transformations represent the change in scale and location between the left hand shape, and its corresponding match within the active shape. When a match is applied, the function associated with the apply match button applies the currently selected transformation matrix to the current left hand shape and removes it from the active shape. It then applies that same matrix right hand shape, and adds the result to the active shape, thus satisfying the $[s - \tau(u)] + \tau(v)$ equation from earlier definitions.

4 EVALUATION

To evaluate the end product for usability, I created and distributed a survey designed to test not only the usability of the tool, but it's ability convey the underlying concepts. The survey first asks what the user's experience with computer science is, either little to no experience with computer science, some experience with coding and/or computer science, or an education and/or career in computer science. The majority of the survey contains a pattern of tasks and then a 1-5 scale question asking if the user was unable to complete the task (a response of 1), or found the task very easy (a response of 5). This is followed by two instructions that involve adding and removing blocks to test the users ability to interact with the principles of solid modeling. The user is then instructed on how to match a single non-terminal to multiple non-terminals in the active shape and select a given match. The next section consists of two tasks based on the previous instruction. Both are in the form example graphics that depict a configuration of left and right hand shapes, along with an expected outcome of applying matches. Included after the tasks is an optional challenge that depicts a set of shapes and asks the user to devise a rule that can generate the language that contains these shapes. The final section of the test asks the user if they how the product represented the concept of shape grammar and volumetric modelling. Each concept was separated into questions about how well that concept was demonstrated, and how much the user learned. The responses on this question can then be compared with the first question to see how people of different levels of familiarity engaged with concepts. The final question asks the user how intuitive the controls were, and how much difficulty they had in using them.

In 6 responses, everyone was able to intact with the shape but the major slipping point was in the application of rules. 4 out of 6 participants said that creating the grey block was very easy, and two others gave it a 3 out of 5. Using the remove tool was similarly rated, so the concept of volumetric solids was clearly easy to grasp for most participants. The problems arose for many users in the instructions and tasks involving finding and replacing. Only 4 of 6 participants were able to find matches of a single non-terminal, and only 2 of those 4 found it easy. This number then shrunk to 3 out of 6 participants when they were asked to apply the matches they found. The first task proved to be the most difficult of the two. 2 out the 6 participants were able to recreate the shape or something like it shown as the expected result. The next task combined the solid modeling concept with the shape grammar concept to procedurally remove a section of a terminals. Surprisingly 4 out of 6 were able to complete this task. The only part that could be considered easier is the lack of a branching tree of non-terminals, and the rules reliance on location rather than scale. Only 4 out of 6 attempted the optional challenge and only 1 reported completing it. Interestingly the one who completed it marked themselves as having little to no experience in computer science.

In the conclusion of the survey, all participants responded as saying the app demonstrated at least some of the concepts of shape grammar and of volumetric modelling. For both questions asking how well the product communicated these concepts, the average was about 3. When asked how much they learned in the survey about grammars and about volumetric modelling, the responses were predictably higher for those inexperienced with computer science. Inexperienced users reported an average of 3.75 out of 5 while experienced users reported an average of 3 out of 5 when asked how much they learned about shape grammar. Inexperienced users reported an average of 3.33 while experienced users reported an average of 2 when asked how much they learned about volumetric modelling. After some feedback outside of the survey it became clear that some participants had a bias towards giving higher responses in order to reflect well on me personally. Taking that into account, the disparity in learning between experienced and inexperienced participants implies that some learning took place. Overall, these results do not disprove that the product could be used as a teaching tool, but they do strongly suggest that the product on its own can not be used as a teaching tool without a better description of the tool. The survey was clearly an inadequate description for most people, although a class in computer science theory, or one that covered the concept of grammars, might provide enough information for the tool to be useful.

5 CONCLUSION

When I set out to avoid making art I invariably make art. The act of divorcing myself from any artistic concepts, becomes a thesis unto itself. I tried to realize an abstract idea, but as I was making it I wasn't able to separate myself from the idea of making with artistic value.

The area of random generation interests me because of its implicit challenge to an artistic authorship. If you can make it, I can automate it. In one direction, I could have put hours of my time into CGA shape and cultivated a city that fit an aesthetic so well that it could be considered art. In doing that I have become an artist, not an engineer trying to challenge artists. I would have toiled away adjusting scales and comparing colors, just like an artist would. The end result would be an artistic experience, that someone could enjoy like they would a series of paintings or photographs. The success of it would hinge on the audience response. Could the landscape I generated land in a photography or conceptual art exhibit? Could it be enjoyed through the Internet, with a passive audience looking for something to catch their eye? These would be the criteria on which I would judge the product if I had gone that route.

In another direction I could have made a tool for an artist rather than challenge them. Say that whatever emerges from using that tool is art of their own license. Like the many Sigraph papers I had read, I wanted to make something that other artists could then use to make art with. The only artist who would actually use the tool I made was myself, and even that's yet to be seen. I didn't want to make a tool that an artist would actually use, I don't know the domain of CG design well enough to know what would actually be used or what hasn't been done before. The window between agreeing to an independent study, and submitting a topic is too narrow to fully investigate that market. I'd had the lingering idea of a high level random generation tool, that could be used to ease the tedium of designing those randomly generated scenes that would contend with art. Most of that interest was borne out of an interest in randomly generated city layouts and architecture. After some short research I realized that this had already been done, with astonishing similarity to my initial idea, in CGA shape.

I was left with two choices, either make a tool like CGA shape or use CGA shape. I didn't see the latter as an option, because then I would be an artist and not an engineer. Going into a career as a software engineer and graduating soon as a bachelor of science, being an artist wasn't acceptable. I felt compelled to be show myself as the engineer, but I couldn't resist the idea of what I made being recognized for artistic value. So I went chasing mathematical purity. I built it around the concept of shape grammar, the underlying basis for CGA shape. The system of octrees arose as a means of representing regions of space, instead of finicky edges and vertices, which differentiated my implementation from the early implementations of Stiny and Gips enough to call it my own. The theory fit well in my head, so I made the tool based around that, thinking it could then exist to communicate those concepts. What I had done was try to express my own interest in these ideas in an app that was compelling because of its simplicity, not its usefulness. Thus I fell into the trap of making art for someone, but in this case all of my artistic efforts went into the conceptual design of the thing, not the specifics of how to use it. I did what I could to imitate UI designers and make the experience of using the tool interesting, but that wasn't the core of it. The core of it was the theory.

The act of distributing the survey, of simply trying to explain it to people enough to get them to give me some data on which I could finish my report on the product, taught me more than the data itself. People wanted to know what they were supposed to engage with. To artists in the design field it's not a real tool, to the layman viewer it's not much to look at. It was made for me out of scientific curiosity. I had a learned about of a few general principles, like shape grammar and volumetric modeling, and I wanted to see them realized in a way that I could interact with. It is a standalone tool to make art, that only functions as an artistic expression unto itself. It exists to express a theory that I saw as having some implicit artistic value.

So here I am having spent so many hours on the thing, and only now is the purpose of what I made dawning on me. In trying to not make art I failed and I made art accidentally. I landed somewhere painfully in between

an artistic work, like Stiny and Gips's original generated painting, and a tool to create that artistic work, like CGA Shape. The conclusion I have drawn from this is that the pursuit of these theories of computer science and computer graphics, creating programs to express them, is in some respects just as much an artistic endeavor as what they are trying to automate. Even if it wasn't a teaching tool or an artistic tool, the end product expresses the concepts that it is based on.

REFERENCES

[1] Eva Dyllong and Cornelius Grimm. 2007. Verified Adaptive Octree Representations of Constructive Solid Geometry Objects. In *SimVis*.
[2] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural Modeling of Buildings. *ACM Trans. Graph.* 25, 3 (July 2006), 614–623. https://doi.org/10.1145/1141911.1141931
[3] G Stiny. 1980. Introduction to Shape and Shape Grammars. *Environment and Planning B: Planning and Design* 7, 3 (1980), 343–351. https://doi.org/10.1068/b070343 arXiv:https://doi.org/10.1068/b070343
[4] George Stiny, James Gips, George Stiny, and James Gips. 1971. Shape Grammars and the Generative Specification of Painting and Sculpture. In *Segmentation of Buildings for 3DGeneralisation. In: Proceedings of the Workshop on generalisation and multiple representation , Leicester*.
[5] Kui Yue and Ramesh Krishnamurti. 2013. Tractable Shape Grammars. *Environment and Planning B: Planning and Design* 40, 4 (2013), 576–594. https://doi.org/10.1068/b38227 arXiv:https://doi.org/10.1068/b38227

A STORING, VIEWING AND EDITING SHAPES

A.1 Estimated Schedule

The estimated completion date of these components was between week 5 and week 3, with 5-7 hours spent each week.

A.2 Actual Schedule

Viewing functions were complete around week two after approximately 5 hours of work. Another 6 hours were dedicated to finding and reading other papers during that period. The basics of adding and removing were complete around week 5 after approximately 15 hours of implementation work and 12 hours of finding and reading papers. After some issues with scaling between different matches, the first attempt at manipulating shapes was scrapped when the plan to use octrees was introduced. This new method was complete by week 10 after about 20 hours of work and 7 hours of reading papers.

B MATCHING AND REPLACEMENT

B.1 Estimated Schedule

Matching was assumed to be done by week 7 at the most and the full system of replacement with multiple rules was expected to be complete by week 12

B.2 Actual Schedule

The initial matching was complete by week 6, after approximately 5 hours of work, with another 6 hours of associated reading. This was also scrapped after octrees were introduced, but before any replacement code was added. The second attempt at matching was completed on week 13 after 20 hours of work. The final version of replacement and rule management was completed on week 14 after another 20 hours of work.

C UI FEATURES

C.1 Estimated Schedule

Some unspecified period of time was left after week 13 for any additional UI features.

C.2 Actual Schedule

5 days after the final version of the replacement code was finished, the UI was prepared for distribution in the survey. This took an additional 9 hours but added vital features that had been neglected in the creation of the replacement rules.

D PAPER

D.1 Estimated Schedule

The completion of the paper was assumed to be relatively trivial compared to coding tasks and was left within the same time frame as the UI updates.

D.2 Actual Schedule

The initial draft of the paper, which included related work and design, was complete by week 11 and took 10 hours of work and 6 hours of additional reading. The final draft was completed after the survey on the last day of work and took an additional 6 hours to complete a conclusion and go through a series of edits.

E USABILITY TESTING

E.1 Estimated Schedule

It was specified in the initial proposal that some set of examples and instructions should be made available to users. No time was allocated for this task.

E.2 Actual Schedule

At the beginning of week 15 these requirements became the survey designed to both instruct users on how to use the product, and get feedback on how usable it was. The creation and distribution of this survey, as well as a set of redundant instructions on the app itself, was completed at the beginning of week 15, and took approximately 4 hours.