Voxel-Space Shape Grammars

Zacharia Crumley
University of Cape Town
South Africa
zacharia.crumley@gmail.com

Patrick Marais
University of Cape Town
South Africa
patrick@cs.uct.ac.za

James Gain
University of Cape Town
South Africa
jgain@cs.uct.ac.za

ABSTRACT

We present a novel extension to shape grammars, in which the generated shapes are voxelized. This allows easy Boolean geometry operations on the shapes, and detailing of generated models at a sub-shape level, both of which are extremely difficult to do in conventional shape grammar implementations. We outline a four step algorithm for using these extensions, discuss a number of optional enhancements and optimizations, and test our extension's performance and range of output. The results show that our unoptimized algorithm is slower than conventional shape grammar implementations, with a running time that is $O(N^3)$ for a N^3 voxel grid, but is able to produce a broad range of detailed outputs.

Keywords:

procedural generation, shape grammars, voxels

1 INTRODUCTION

For video games, virtual environments, and cinema special effects, cost-effective content creation is an increasing concern. The amount of models, animations, textures, and sounds needed for these applications has been steadily growing with the increase in computational power and the quality of graphics. It is now at a point where hundreds of modellers, animators, and artists will work for months or years to create the content necessary for a single mainstream video game or blockbuster film. The large size of these teams means the costs involved are significant, and in spite of the number of people working on the project, long development times are still the norm. For this reason, content creators have begun turning to *procedural generation*, as a way of decreasing costs and shortening development times.

Procedural generation refers to methods designed to algorithmically generate content, instead of having it hand-crafted. Minimal human interaction is required – generally limited to setting the initial parameters of the algorithm, or providing example inputs.

Today procedural generation is increasingly used to generate large amounts of high quality content, particularly plants, landscapes, and textures. This is evidenced by the growth of commercial procedural generation software, such as SpeedTree[1], Terragen[2], and CityEngine[3]. There are many procedural generation algorithms [14] but our research focuses specifically on *shape grammars* [18]. These are a type of formal grammar, consisting of an axiom (the initial item to begin with) and a set of production rules which modify, add, or replace items.

Shape grammars are distinguished from conventional grammars, in that their rules operate directly on geometric shapes instead of symbols from an alphabet. Their production rules include geometric operations on shapes (such as rotation, scaling, etc.), in addition to shape replacement (as grammars do with symbols). An example of a basic shape grammar is shown in figure 1.

Shape grammars were originally developed in architecture as a tool for formalizing architectural design. Although still used for that purpose, they are also finding use in computer science as a method for procedurally generating models of buildings and other structures. This is the application our research focuses on.

Conventional shape grammars operate on mesh representations of shapes. These are moved, rotated, subdivided and otherwise operated on until a final collection of shape meshes is produced: the output of the shape grammar.

However, there are two major problems with conventional mesh-based shape grammars, as used in procedural generation and both are difficult to solve.

Firstly, it is difficult to robustly apply constructive solid geometry (CSG) or Boolean geometric operations on

[1] http://www.speedtree.com/
[2] http://www.planetside.co.uk/
[3] http://www.procedural.com/

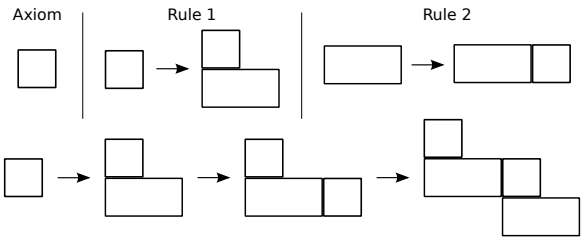

Figure 1: A simple shape grammar that produces an infinite series of 2D stairs, and its first three iterations.

meshes: overlapping edges and vertices must be identified and trimmed or removed, whichever is appropriate. This can lead to complications around numeric stability, slivers, degenerate triangles, and other issues. Alternatively, this redundant geometry can be left hidden, but this is inefficient. It may also be necessary to create new geometry. This process is complex, and there are edge cases that remain problematic. Even with such an algorithm, the overlap between different shapes can lead to texture seams. This is visually unappealing and difficult to overcome.

The second limiting problem of mesh shape grammars is that texturing is done at a per-shape level. The faces of each shape, or pre-made piece of geometry, have fixed texture coordinates and associated 2D textures that are projected onto the faces. This means it is difficult to have texture details that span multiple shapes, or control the textures at a sub-shape level.

For example, it would be difficult to create racing strips running along a car model produced by a shape grammar, since the different sections of the car are made up of different polygons created by different rules.

Modifying shape grammars to operate in a voxel-space solves these limitations. CSG operations on voxels are trivial, solving the first deficiency. For the second, textures can be assigned on a per-voxel basis, which allows details to more easily span shapes and removes the texture's association with a specific shape.

Using voxels presents some challenges, such as the need for large amounts of memory and storage, and the discrete nature of the underlying grid, which can introduce aliasing artifacts and other sampling issues. However, these problems can be dealt with, or worked around, using tree data structures for efficiently managing space, which also allow us to use high resolution voxel grids, reducing the impact of aliasing artifacts.

This paper presents our preliminary research into interpreting shape grammars in a voxel space, in contrast to the traditional mesh geometry approach. Our goal is to extend the expressive range of shape grammars with the ability to easily and robustly apply Boolean geometry operations.

Our major, and novel, contribution is an algorithm for this process. The algorithm is made up of four main stages, and produces a mesh model, suitable for use in real-time, or offline, 3D graphics. We also discuss optimizations and optional steps for the algorithm, as well as testing its performance and range of output.

2 RELATED WORK

Stiny and Gips [18] first developed shape grammars in an attempt to formalize architectural design. Although still used in architecture, they have also been adapted for use in the procedural generation of structures for other applications [19, 12].

Early shape grammars were simplistic with a limited range of output, but over time several ideas from procedural generation, most notably from L-systems, were incorporated into shape grammar implementations. Most notably, environmental sensitivity and stochastic rules [3]. With environmental sensitivity, rules can query the current set of shapes and adjust their output based on the information they get. Stochastic rules introduce randomness, by randomly choosing different outputs and parameters to introduce variation in the shapes generated.

Shape grammars were later extended to building generation using *split grammars* [19], which focus on subdivision of shapes. For example, a building's wall is divided first into floors, then the floors into different rooms, and finally the walls of the rooms into different windows. Split grammars are particularly well suited to façade generation [19] due to the regular, grid-like layout of building windows. They work by recursive subdivision of a shape, guided by following productions from a rule set. The final set of shapes that arise from the repeated subdivision form a model of the desired building wall. However, split grammars are less successful at creating internal structure. Early approaches to this problem tended to be simplistic, such as using n-sided prisms as the split grammar axioms [4]. Subsequently, better methods were developed, such as starting from the building's footprint obtained from aerial imagery [7], and creating the building's structure with shape grammar rules [12].

Along with the other extensions mentioned, the features of split grammars have since been incorporated into current grammar implementations, unifying all features under one grammar, for greater ease-of-use [12].

The range of output possible from modern shape grammar implementations is extremely broad [12] and as a result, shape grammars are considered the industry standard for procedurally generating architecture. They are able to create whole cities with realistic buildings (using additional techniques [15] to create the road networks and block layout). The best example of this in

practice is the CGA grammar of CityEngine[4], which procedurally creates cities and buildings.

Example-based shape grammars can be used to generate different models in the same style as the given-example. This can be done from an image of a building's façade [13], or from existing models [2].

The problem of easily, and visually, editing shape grammars has also been addressed by Lipp et al. [9] in their work on interactively editing shape grammars for architecture. Their approach is primarily concerned with operating on the grid-like façades of buildings, but does feature methods for assisting in the overall building structure creation.

Recent work [1, 5] has also developed methods for automatically creating a structural skeleton for models generated by shape grammars. These skeletons can then be used to create animations or run structural simulations on the generated models using a physics engine.

However, shape grammars do have limitations. In split grammars, shapes that span multiple subdivisions, and shape intersections, are difficult to handle gracefully. In addition, particularly unusual building designs with complex elements, such as tunnels and interior hollows, are very hard to generate.

Voxels have seen previous use in procedural generation, predominantly in games [5] and terrain representation. However, 3D texture synthesis methods have been extended to create 3D models. Merell's algorithm [11] works by assigning a cuboid section of geometry to each voxel type, and then keeping a record of how these cubes of geometry can be placed adjacent to each other while keeping the resulting model consistent. Texture synthesis methods are employed to create a, potentially infinite, voxel grid that corresponds to a consistent model. One downside is that this requires the manually created cuboid sections of geometry.

Another modelling approach that allows Boolean geometry operations while avoiding the issues around mesh-based CSG was proposed by Leblanc et al. [8]. Their approach allows modeling of shapes using Boolean geometry operations, but is substantially more complex than a voxel-based approach, and does not solve the issue of creating surface detail at a sub-shape, or trans-shape, level.

3 FRAMEWORK

The process of interpreting a shape grammar to produce a model in our framework consists of five stages (one of which is optional), and requires two inputs from the user. The final output is a textured mesh, suitable for use in modern graphics applications. The two user inputs are:

[4] http://www.procedural.com/

[5] http://www.minecraft.net/

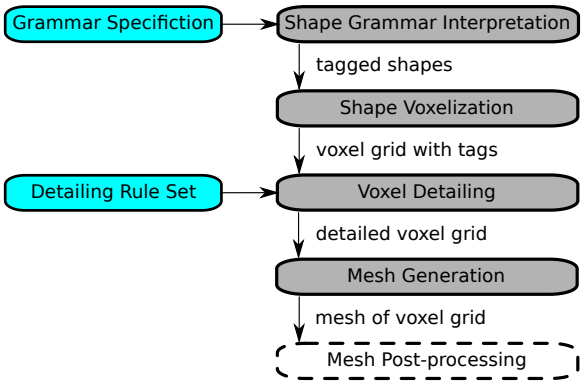

Figure 2: A flow chart of our algorithm. The cyan boxes are user inputs; the grey boxes are the four stages of the algorithm. Arrow labels show the output of each stage. Mesh post-processing is optional.

Shape Grammar Specification: This is the set of productions and associated parameters for the shape grammar itself.

Detailing Rule Set: The collection of rules that are used to assign visual detailing information to the voxel grid. This information is used when displaying the created model.

The five stages of the algorithm are:

1. **Shape Grammar Interpretation:** The input grammar is run to produce a collection of shapes.
2. **Shape Voxelization:** The shapes from the previous step are voxelized into a voxel grid.
3. **Voxel Detailing:** Surface voxels in the grid are assigned visual detailing information.
4. **Mesh Generation:** A mesh representation of the voxel shape is produced, using the marching cubes algorithm.
5. **Mesh Post-processing** (optional) The generated mesh is smoothed and refined.

Below, we cover each of the five stages in detail, explain their operation, what inputs they use, and what outputs they generate. Figure 2 shows an overview of the process.

3.1 Shape Grammar Interpretation

The first stage of the algorithm requires that we specify a shape grammar and then iterate it to produce the output set of shapes. This step is very similar to applying a conventional shape grammar, with some minor differences. A set of shape grammar rules, and an axiom shape, are provided by the user. This rule set is then run on the axiom, producing new shapes and modifying existing shapes on each iteration of the rules. This continually-updated set of shapes (the current shape set) converges to the final output of the shape grammar. An

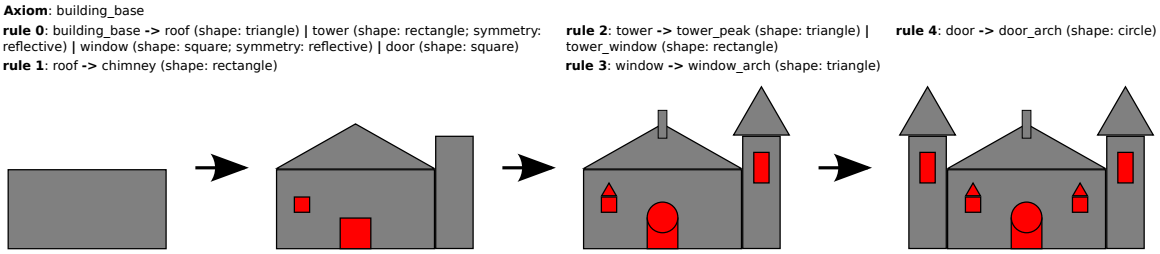

Figure 3: A simple shape grammar being interpreted in parallel to form a basic building. Grey shapes are additive, and red ones are subtractive. Position and size information in the example grammar are not shown, for the sake of simplicity. Note that the final step involves the symmetric copies being created, as the grammar only requires two iterations to complete.

Algorithm 1 Shape Grammar Interpretation

```
currShapeSet ← {Axiom}
iterations ← 0
while iterations < MaxIterations and currShapeSet.hasNonTerminals() do
  if parallelExecution = TRUE then
    for all i ∈ currShapeSet do
      i ← doRuleDerivation(i)
    end for
  else
    a ← getFirstNonTerminal(currShapeSet)
    a ← doRuleDerivation(a)
  end if
  iterations ← iterations + 1
end while
for all i ∈ currShapeSet do
  if i.hasSymmetry() then
    s ← i.createSymmetricCopies()
    currShapeSet.insert(s, i − 1)
  end if
end for
```

example of this stage is shown in figure 3. Pseudocode for this process is shown in algorithm 1.

This process will terminate under one of two conditions: Either after a user-defined maximum number of iterations, or once all shapes are terminal and none of the rules can be performed on the set of shapes.

Rules can be interpreted in parallel (as done in L-systems [16]), or in series (as done in traditional formal grammars [12]). These are appropriate in different situations, depending on the type of model being generated by the shape grammar. Serial rule derivation is suitable for most situations, except for models that have fractal qualities, where parallel rule derivation is advised.

Any of the many enhancements and extensions to grammar generation methods can be used here: environmental sensitivity, stochastic rules, a derivation tree for querying earlier shape set states, and more.

Our implementation includes these three extensions, as well as split grammar operators [12]. For more information on these extensions, we refer readers to the literature [16, 12, 3]. We also have a collection of standard utility functions for common operations, such as scaling, translation, rotating, and hollowing shapes. All that is required from this stage of the generation process is a specification of the final set of shapes.

There are two grammar extensions that we found very useful for generating models in our experiments.

First is the tagging of shapes with metadata. One of the operations that the shape grammar rules can perform is to add metadata tags to shapes in the current shape set. We implemented these as arbitrary strings. These serve to preserve additional information about the shapes that can be used in later stages of the generation algorithm. For example, in generating a castle, a cylinder (and its children if recursive tagging is used) could be tagged with “type:tower” and “material:stone”. These tags indicate additional properties of the shape that enhance later detailing and texturing.

Secondly, we support the specification of symmetry in the grammar rules. Our grammar implementation has special operators for indicating that a shape, or group of shapes, (and any child shapes that derive from them) should be cloned to create symmetrical versions.

We support two types of symmetry: rotational and reflective. In rotational symmetry, three arguments are provided to the operator, from which positioning information for the symmetrical copies can be derived: P_{rot} - the center point around which the symmetric branches are rotated; V_{rot} - a vector normal to the plane of rotation, and N_{rot} - the number of rotational copies to create.

For reflective symmetry, only two arguments are required to fully construct the mirror copies of the shape, or group of shapes, to be reflectively copied: P_{ref} - a point on the plane of reflection, and V_{ref} - a normal to the plane of reflection.

Symmetry information is specified when the grammar is run, but symmetric copies are only added once the grammar rules terminate. This is done as a post-process because further shapes could be added to the set of shapes undergoing symmetry, in iterations after the

symmetry is specified. Rather than tracking the symmetric copies and updating each of them for every change in shape, we simply flag the set of shapes for symmetry and wait until the rule derivation completes, before creating the symmetric copies.

Once the shape grammar has finished, a full specification of the final output set of shapes is passed to the next stage. This includes positions, dimensions, orientations, tags, and any other relevant information.

3.2 Shape Voxelization

In this phase, the shapes output from the shape grammar are voxelized into a voxel grid. This is analogous to rasterizing vector graphics into a pixel format. An example of this process is shown in figure 4, and pseudocode in algorithm 2.

Algorithm 2 Shape Voxelization

```
shapes ← getShapeGrammarOutput()
shapes ← sortByPriority(shapes)
shapes_bbox ← getBoundingBox(shapes)
gridResolution ← getVoxelGridResolution()
voxelGrid ← initializeEmptyGrid(gridResolution)
for all i ∈ shapes do
    i ← scaleShape(i, gridResolution, shapes_bbox)
end for
for all i ∈ shapes do
    voxelGrid.voxelizeShape(i)
end for
return voxelGrid
```

Due to the large memory requirements of storing voxel grids naïvely, it is infeasible to store the grid as a 3D array. Our implementation uses an *octree*, to efficiently manage space [17].

It is possible to use other tree data structures for storing the voxel grid, such as point region octrees (PR-octrees), kd-trees, or R-trees. In the general case, where no assumptions can be made about the data sets to be stored, and no special look-ups are required, the best option is an octree [17]. This is because the other tree types all require re-balancing (an expensive operation).

Tags associated with the shapes to be voxelized are assigned to the relevant voxels. In the case of overlapping shapes, it is possible that a voxel may inherit tags from multiple shapes. This is not problematic at this stage, but may cause ambiguities during detailing, which could have unintended consequences. Users should bear this in mind when designing grammars.

The order in which shapes are added is also important, because shapes may be additive or subtractive (additive for creating solid structures, or subtractive for carving empty spaces out of solids). Adding and subtracting geometry in this manner is not commutative. Hence a grammar may generate unintended results, depending on the order in which the shapes are voxelized.

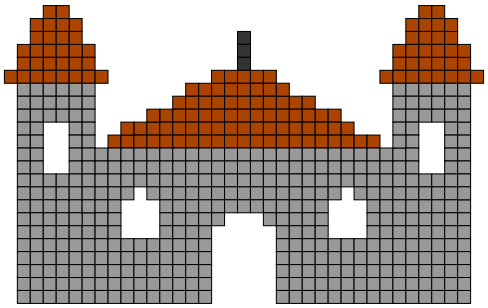

Figure 4: The output of the simple building shape grammar from figure 3 after being voxelized. The colours of the voxels correspond to the tags they inherited from the shapes. Grey indicates 'material:wall', brown indicates 'material:roof', and the dark grey 'material:chimney'.

To resolve this ambiguity, the shape grammar can assign a *priority* to the shapes. This is an integer that determines when the shape will be voxelized. Before voxelization, the shapes are sorted by priority, and added in sorted order. This allows a user to control when shapes are added, and resolve order-dependency issues.

Finally, we can manually edit the voxel grid once the shapes have been voxelized. This could be done to allow hand-crafted modifications to the output of a grammar, or because the user is dissatisfied with some aspect of the output that is difficult to correct in the grammar.

Manual editing is important for artists and modellers, and our shape grammar extensions do not restrict it at all, although it requires voxel editing software.

The final output of this stage is a 3D voxel grid, where each voxel is either solid or empty, and may have metadata tags associated with it.

3.3 Voxel Detailing

In this stage of the algorithm, voxels are assigned an appearance in the final model. This can include, but is not limited to, texturing information, bumps maps, displacement maps, lighting information, and materials. This is done on a per-voxel basis, by a user-created *rule set* which operates on each voxel individually. These rules may iterate over the voxels multiple times, allowing the creation of complex multi-pass detail. For example, cellular automata patterns could be created, since they map very well onto the discrete, gridded nature of voxels. The scope of these rules is extremely broad, and features such as context-sensitivity and randomness can easily be included. Everything from assigning a simple texture based on position, to randomized complex multi-pass procedural methods are possible. A simple example of a voxel grid after undergoing detailing is shown in figure 5, and pseudocode of the detailing process is found in algorithm 3.

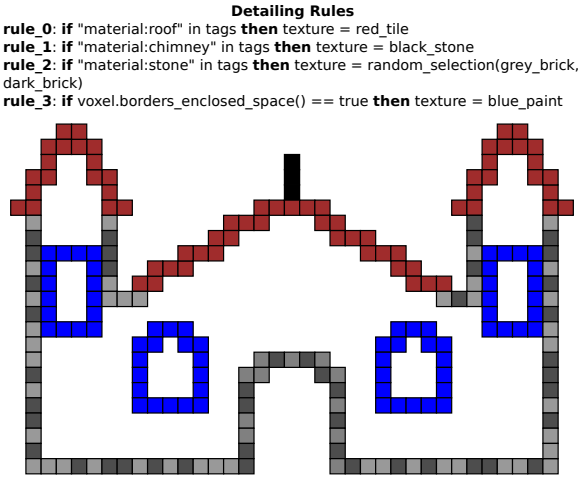

Figure 5: The voxel grid from figure 4 after detailing. Each voxel has been assigned a texture in accordance with the detailing rule set supplied. Non-surface voxels are ignored, and are not displayed in the diagram.

Algorithm 3 Voxel Detailing

```
detailingRuleSet ← getDetailingRuleSet()
maxIterations ← getMaxDetailingIteration()
surfaceVoxels ← voxelGrid.getSurfaceVoxels()
for i = 1 to maxIterations do
    for all j ∈ surfaceVoxels do
        n = voxelGrid.getNeighbouringVoxels(j)
        j.detailTags ← detailingRules.runRules(j, n)
    end for
end for
```

Detailing is done on a per-voxel level as opposed to the per-shape level of conventional shape grammars because this allows more complex procedural detailing of generated models, and it is much easier for detail features to span shapes and work on sub-shape scales. This also circumvents the problem of texture seams between adjacent shapes prevalent in conventional grammars. The disadvantage to this freedom is more complexity for the user. This complexity could be reduced in two ways. Firstly, by creating a visual rule editor to use, as opposed to text-based programming. Secondly, by designing an interface that allowed rapid prototyping of rules on small examples, to quickly detect problems. However, we did not implement these, and leave them to future work.

Relevant details about the voxel are passed to the rule set. In our implementation these details are: tags associated with the voxel; normal of the voxel; the maximum resolution of the octree; the coordinates of the current voxel; the count of the current iteration of the rule set, and the above details for all neighbouring voxels, within a user-specified radius.

It should also be noted that this stage is independent of previous steps. A detailing rule set can be applied to any voxel grid, and does not need to concern itself with how that data set was produced.

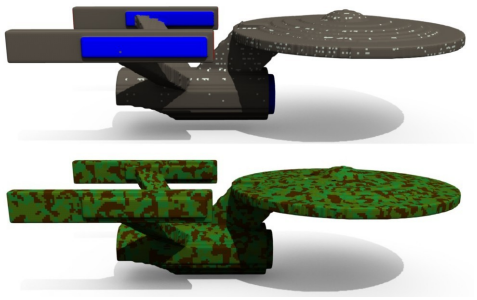

Figure 6: Our Enterprise model with two different detailing rule sets applied to it. Above, with its original detailing; below, with a camouflage pattern. This shows how detailing rule sets that are not reliant on shape tags, such as camouflage, can be applied to any voxel grid.

It is possible to have a detailing rule set that is completely independent of metadata tags. For example, detailing that creates a consistent pattern across the entire model without using the context information from the tags. These detailing rules will work on any model provided to them, regardless of tags. An example of such a detailing rule set being applied to a model intended for a different rule set is shown in figure 6.

However, in most practical situations, we expect that rules from the detailing rule set will be dependent on metadata tags in the voxel data set. For example, detailing a house's walls with brick textures and the doors with wooden ones requires that the two parts of the model be distinguished. Hence the user should ensure that the shapes in the shape grammar are properly tagged for the detailing stage.

Running snippets of code for each voxel in a grid can be extremely slow, especially so if the grid is large, or the rule's code is complex. For this reason, we only run the rule set on surface voxels in the grid.

We define a surface voxel to be any solid voxel in the grid which is 26-connected to at least one empty voxel. Because the marching cubes algorithm does not generate triangles for completely empty or solid space, only voxels on the border between solid and empty space will affect the final model. All others can be ignored.

Using the hierarchical structure of the octree, surface voxels can be quickly identified. If a node of the octree does not have any children, then only the voxels around the edge of that node need be checked further. For all reasonable models, this dramatically reduces the number of surface voxel candidates that need to be checked, improving speed by an order of magnitude or more.

The final output of this step is a voxel grid where all surface voxels have been assigned detailing information. This information must unambiguously provide all

information required for rendering, either as is, or when converted to a mesh.

3.4 Mesh Generation and Post-processing

There are many methods for rendering voxel grids. These are often based on ray tracing, or point rendering. In some situations it may be suitable to render the output of the shape grammar with these methods, but most graphics applications today work with triangle meshes, not voxel data sets. For this reason, we need to convert our voxel data set into a mesh that can be used in conventional raster graphics applications, such as modern 3D game engines.

The marching cubes algorithm [10] is a well-established solution to the problem of extracting a mesh representation of a voxel grid or isosurface. We make use of it to produce a mesh version of our generated model.

The algorithm outputs a list of triangles, each of which can be associated with a voxel in the input grid. Each triangle is then assigned textures, materials, and other detailing information from the surface voxel.

Hence we end up with a fully textured and detailed mesh representation of the voxel grid that the original shape grammar produced.

The mesh produced by the marching cubes algorithm is suitable for direct use in graphics applications, but its visual quality could be improved by post-processing.

One of the problems with the marching cubes algorithm is that the output mesh has visual artifacts caused by the discrete nature of the voxels. Curves in particular, are not fully captured during the meshing process, and will instead appear 'bumpy', although increasing the resolution of the voxel grid can reduce this.

The severity of this problem can be reduced by an appropriate mesh smoothing algorithm, which will significantly decrease the impact of such artifacts [6].

It should be noted though, that naïve smoothing algorithms can lose details that are not artifacts, and should be retained, such as sharp corners. For this reason, we recommend the use of one of the more advanced smoothing algorithms, which will retain these features. There are a number of such algorithms, but in general these advanced methods of smoothing come at the cost of more complexity and a longer running time.

3.5 Optimizations

There are several optimizations to our algorithm that we did not implement due to time constraints. These have the potential to dramatically reduce running times, and hence we discuss them here.

Voxelization of shapes can be performed extremely efficiently by exploiting the hierarchical nature of the octree. Beginning at the root of the tree, query the intersection between each of the eight children of the octree node, and the shape to be added. If the area covered by a child node is entirely within the shape, then that voxel is set with the shape's information, if there is a partial intersection between the child node and the shape, then the algorithm is recursively called on that node.

While faster, this is more complex to implement, and requires an exact collision detection algorithm. We suggest that future implementations make use of this method to greatly reduce run times.

The implementation of marching cubes can also be substantially accelerated by only marching over the surface voxels of the voxel grid. Since solid or empty regions will not produce triangles, the voxels of those regions need not be processed. The list of surface voxels from the detailing step can be re-used here.

4 TESTING AND EXPERIMENTATION

In order to evaluate our voxel-space extensions to shape grammars we undertook three experimental tasks: performance testing, where we analyzed the time and memory required; variation testing, where we produce multiple similar models from a single grammar; and output range testing, where we examine the range of outputs our algorithm can produce, and its ability to generate models of well-known structures.

4.1 Performance

We analyzed our algorithm's performance across a variety of voxel grid sizes and user inputs. The two main results of interest are the time taken to generate a model, and the peak memory usage of the process.

We decomposed timing into the four stages of the algorithm to get an idea of their relative durations (post-processing was excluded as it is an optional step).

Testing was performed on a PC with an Intel Core 2 Duo clocked at 2.4Ghz and 3 gigabytes of RAM.

Performance testing was conducted with a selection of 36 shape grammar and detailing rule-set combinations, at 4 different voxel grid resolutions. The selection of grammars and rule sets was specifically chosen to encompass a wide range of complexity. figure 7 shows the timing results across all of the 36 models.

Before analyzing the results it should be noted that our implementation was strictly intended as a single-threaded proof-of-concept. Hence, performance was not a priority, and there is large scope for improvements in this area (as mentioned in section 3.5.) Nonetheless, we include our results as we believe they provide a baseline for comparison to future implementations of our work.

The first thing to note is that the shape grammar interpretation is orders of magnitude faster than the other

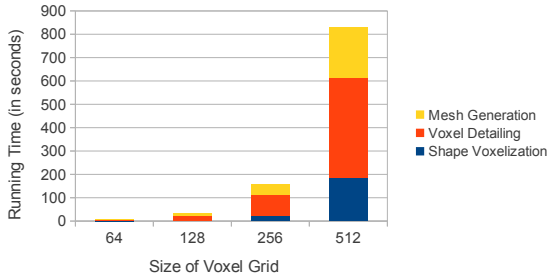

Figure 7: A cumulative graph of the average times taken for our algorithm to run on 36 different inputs, covering a range of complexities. Shape grammar interpretation is not shown as it was negligible compared to the other three stages.

stages, due to its independence from the voxel grid resolution. The average interpretation time was 50 milliseconds. Due to the minuscule relative time, grammar interpretation is not shown in figure 7.

As expected the running times of the other stages of the algorithm is approximately cubic in the size of the voxel grid. This is expected, as their running time is directly proportional to the number of voxels to operate on, which scales cubically with the size of the grid.

The biggest cause of variation in running times is the number of iterations in the surface detailing. Because each voxel must be processed for every iteration, the number of iterations makes a large difference in the amount of processing to be done, especially for higher voxel grid resolutions.

Peak memory usage followed the same pattern of being cubic in the resolution of the voxel grid. The minimum and maximum amount of memory used, across all testing inputs, were approximately 150 and 1400 megabytes, respectively.

These running times are significant for larger resolution grids. However, in practice, users can prototype their grammars and rule sets on lower resolution models and, once satisfied with them, then do off-line generation of a high resolution model for actual use. This means the long running times for large models will not significantly disrupt work-flow.

4.2 Variation and Range Testing

Variation testing involved randomizing the parameters in several of our shape grammars, and producing multiple models from them. The objective is to ensure that our algorithm is capable of producing many different models that share a similar style, from a single shape grammar. A selection of the models produced in this manner are shown in figure 8.

As can be seen from the images, our algorithm is capable of producing a variety of models, sharing a common theme and style, from a single shape grammar and detailing rule set, by randomizing the parameters of the shape grammar and detailing rules.

To test the power of our shape grammar extensions, we created shape grammars and detailing rule sets representative of a broad variety of models, including imitations of well-known existing structures. A selection of these generated models are shown in figure 9.

5 LIMITATIONS

There are two limitations to our voxel-space shape grammar algorithm that could restrict its potential uses.

Firstly, in order to obtain a high quality model from a voxel data set, the set must be at a high resolution, so as to remove “blocky” visual artifacts caused by the discrete nature of a voxel grid. Mesh smoothing as a post-process helps, but it is not sufficient on its own.

However, the higher the resolution, the slower the voxel detailing process is. This is because each voxel in the model must be detailed, and the number of voxels is cubic in the dimensions of the voxel grid.

Secondly, texturing at a per-voxel level may be insufficient in certain cases, such as for curved surface details, where the discrete nature of the underlying voxel data can cause visual artifacts. For example, an elaborate spiral design with fine curved detail on the side of a spaceship would almost certainly run into sampling issues if created with a detailing rule set.

A possible solution to this problem would be allowing the addition of decal textures to the final version of the mesh. These decals would replace the existing details in certain locations and display detail that could not be created within the detailing rule set framework.

It must be noted though, that neither of these limitations are critical, and none of them should be problematic in the majority of cases.

6 CONCLUSION

We have presented a novel extension to conventional shape grammars, where the shape output of the grammar is voxelized, allowing more robust Boolean geometry operations and a new per-voxel approach to detailing the surface of generated models.

These extensions address two shortcomings in current shape grammar implementations: the support of complex shapes through CSG, and sub-, or trans-, shape detailing at per-voxel level, for more elaborate and controlled texturing.

Our algorithm is slower and more memory intensive than conventional shape grammar implementations, but not outside acceptable limits. Additionally, our extensions allow the generation of a wide range of models, including variations from a single shape grammar.

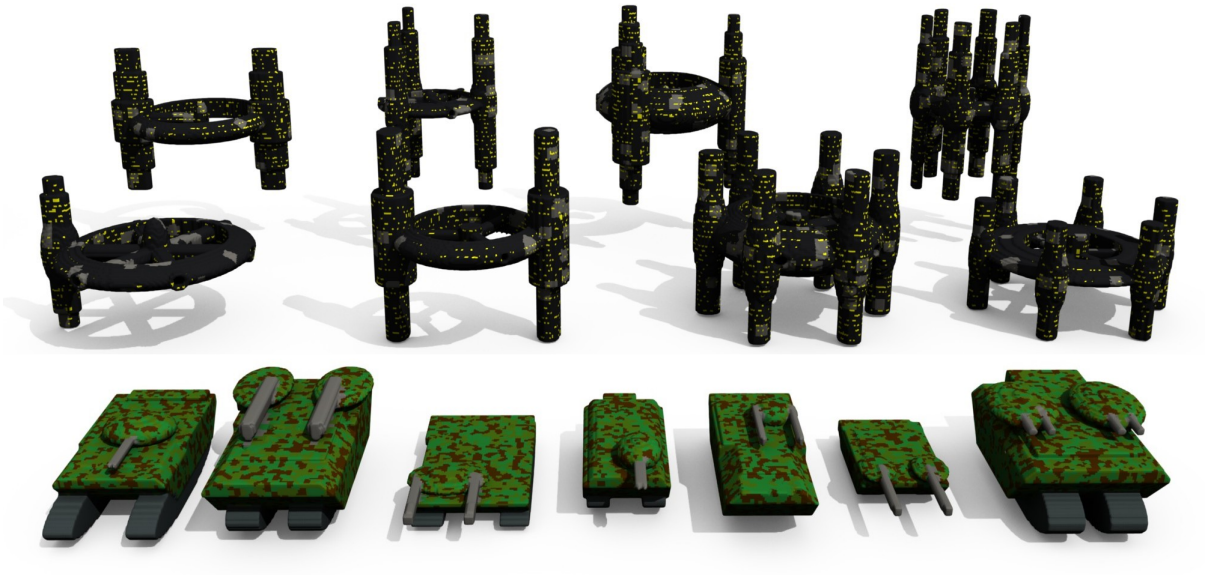

Figure 8: A selection of tanks and space stations produced by two of our shape grammars, using randomized parameters in their rules. This shows how a single grammar can produce multiple models in the same style. These models were all generated from cubic voxel grids of resolution 256.

Figure 9: A broad selection of the models produced by our algorithm, using a range of detailing rule sets and shape grammar extensions, including cellular automata patterns, symmetry and multiple-pass textures. All of these models were generated using a cubic voxel grid of size 256.

Our extensions add new functionality to shape grammars, without losing existing capabilities, and significantly increase the range of achievable content.

6.1 Future Work

The per-voxel detailing stage could be expanded to address the interior of generated models. In our work, we have only performed detailing on the surface voxels of the model, but the method could be extended to detail the interior voxels too. This would allow the creation of details such as rooms inside generated buildings.

Post-processing could also be done on the voxel grid before it is detailed. This could be used to add, remove, and tag voxels to create detail corresponding to damage, wear and tear over time, growth of mold, and more.

The voxel detailing rules could be extended to operate at multiple resolutions of the voxel grid. Octree nodes could easily be coalesced to form a lower-resolution version of the model, to which the rule set could then be applied. This would allow the creation of large scale detailing initially, working down to finer details as the rules are run at higher resolutions.

Finally, a solution to the constraint of texturing being limited to a per-voxel level is the use of decal textures on the generated mesh. Detailing could be extended to

allow arbitrary textures to be projected onto the generated mesh, complimenting the textures assigned in the detailing step. This would allow texturing beyond the per-voxel level our system is currently limited to.

ACKNOWLEDGEMENTS

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the authors and are not necessarily to be attributed to the NRF.

Funding assistance for this research was also provided by the University of Cape Town.

7 REFERENCES

[1] Richard Baxter, Zacharia Crumley, Rudolph Neeser, and James Gain. Automatic addition of physics components to procedural content. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '10, pages 101–110, New York, NY, USA, 2010. ACM.

[2] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. A connection between partial symmetry and inverse procedural modeling. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 104:1–104:10, New York, NY, USA, 2010. ACM.

[3] Peter Eichhorst and Walter J. Savitch. Growth functions of stochastic lindenmayer systems. *Information and Control*, 45(3):217–228, June 1980.

[4] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, pages 87–ff, New York, NY, USA, 2003. ACM.

[5] Martin Ilčík, Stefan Fiedler, Werner Purgathofer, and Michael Wimmer. Procedural skeletons: kinematic extensions to cga-shape grammars. In *Proceedings of the 26th Spring Conference on Computer Graphics*, SCCG '10, pages 157–164, New York, NY, USA, 2010. ACM.

[6] Thouis R. Jones, Frédo Durand, and Mathieu Desbrun. Non-iterative, feature-preserving mesh smoothing. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 943–949, New York, NY, USA, 2003. ACM.

[7] R. G. Laycock and A. M. Day. Automatically generating large urban environments based on the footprint data of buildings. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03, pages 346–351, New York, NY, USA, 2003. ACM.

[8] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Modeling with blocks. *The Visual Computer (Proc. Computer Graphics International 2011)*, 27(6-8):555–563, June 2011.

[9] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–10, New York, NY, USA, 2008. ACM.

[10] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, August 1987.

[11] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 105–112, New York, NY, USA, 2007. ACM.

[12] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 614–623, New York, NY, USA, 2006. ACM.

[13] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 85, New York, NY, USA, 2007. ACM.

[14] F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and modeling: a procedural approach*. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[15] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[16] P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[17] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[18] G. Stiny and J. Gips. Shape grammars and the generative specification of painting and sculpture. *Information processing*, 71:1460–1465, 1972.

[19] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 669–677, New York, NY, USA, 2003. ACM.