Structural Synthesis using a Context Free Design Grammar Approach

Mikael Hvidtfeldt Christensen, MSc.
Aarhus, Denmark.
structuresynth.sf.net
e-mail: mikael@hvidtfeldts.net

Abstract

This paper introduces Structure Synth, a 3D structure generator based on design grammar specifications.
Noam Chomsky pioneered the use of formal grammars to describe the structure and syntax of language. These formal grammars were classified according to their expressive power. Of special importance here is the class of Context Free Grammars, originally believed to be powerful enough to model natural languages. While Chomsky's formal grammars describe structure in one-dimensional strings (symbolic sequences), Chris Coyne created the Context Free Design Grammar, an extension of the formal grammars modeling two-dimensional structures using a simple set of primitives (e.g. squares and circles).
Structure Synth is the natural extension of these ideas into three dimensions. The user specifies a grammar, and the program generates one of the many possible structures adhering to the syntax of the grammar. Compared to general-purpose programming, the restrictions of context-free systems encourage the user to discover and explore the systems. And even though the syntax limits the complexity of the rules, the resulting structures are often highly complex and nearly always unpredictable and surprising.

Introduction

Structure Synth is a software system for creating and exploring structures defined by a set of transformation rules. This paper describes the ideas behind Structure Synth together with an introduction to the methods it is inspired by.
The paper is organized as follows: First, formal grammars are introduced and it is discussed how they can be used for generating content. This is followed by a description of how Context Free Design Grammars can be used to generate two-dimensional graphical content.
The second part describes how Structure Synth extends these ideas into three dimensions. The purpose is not to give a complete reference to all aspects of the application, but to illustrate the syntax and provide examples of the different types of structure that can be generated.
The paper concludes with a discussion of why context-free grammars are interesting in relation to generative art, and presents some possible future directions for Structure Synth.

Formal Grammars

A formal language is a set of strings, where each string is a finite sequence of symbols. Formal languages can be specified in different ways: for languages with a finite number of strings, it would be possible to list all strings, but a more convenient way is to describe a formal language by a set of rules, which may generate the strings in the vocabulary. It is important to note that formal languages have no direct connection to the natural languages. Formal languages are mathematical, formalized concepts. They may be used when trying to describe or model the structure of natural languages, but bear in mind that the entities in a formal language (the strings and the symbols) may represent structure at many different levels. For instance, the strings in a formal language could represent words, sentences, or paragraphs of text in a natural language depending on the level of structure being modeled.

Noam Chomsky studied the structure of formal languages and created a hierarchy that classified the languages according to the generative power of their formal grammar [1]. A formal grammar is one way to describe or generate a formal language. A formal grammar is a set of rules (sometimes called production rules) which operates on two kinds of symbols: the terminal symbols, which are the symbols that the strings in the formal language are composed of, and the non-terminals, which are intermediate symbols used during the derivation. By applying the production rules to a start symbol in the formal grammar, all the possible strings in the formal language are created. For instance, consider the following toy-example for describing a small subset of the English language:

S → NP VP
VP → V NP
NP → DET N
DET → the | a | an | ...
N → dog | boy | cat | ....

Here the start symbol is a sentence (S), which is composed of a noun phrase (NP) and a verb phrase (VP). The verb phrase again consists of a verb (V) and a noun phrase, and the noun phrase consists of a determiner (DET) and a noun (N). These are the non-terminals of this toy grammar. Finally, there are a few production rules for substituting the noun and determiner with terminals, which here are actual English words. An example of a sentence analyzed (or constructed) using this grammar is shown in Figure 11.

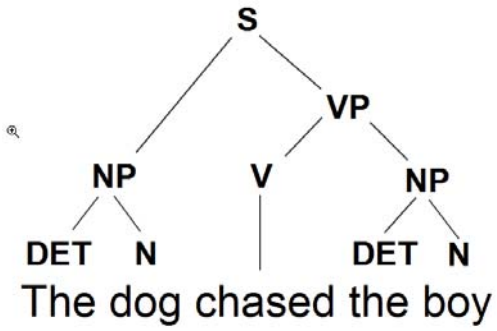

Figure 11: An example of using a formal grammar to describe the structure in an English sentence. Here the strings in the grammar correspond to complete sentences, and the terminal symbols are English words. The structure is shown in a hierarchical tree, corresponding to the production rules listed above.

The production rules above all share a specific simple form: they consist of a single symbol, which transforms into one or more new symbols. Thus, when deriving strings using the grammar, it is not necessary to take the context (the surrounding symbols) into account, which is referred to as a context-free transformation. It is also possible to construct production rules which are context-sensitive. For instance, a transformation rule of the form: αAβ → αXβ would imply that A could be substituted by X, but only if the A was surrounded by α and β. This is an example of a context-sensitive transformation.

The hierarchy Chomsky created for the formal grammars contains the following classes:

Type 0	All formal grammars. Here there are no restrictions on the production rules.
Type 1	Context-sensitive grammars. These are grammars that can be expressed using context-sensitive production rules, such as the one mentioned above: αAβ → αXβ where α and β are arbitrary symbols, and A is a non-terminal and X a non-empty symbol.
Type 2	Context-free grammars. Here the left side of the production rules consists of single non-terminal. This class of grammars has been used to study the structure of several natural languages [19]. In addition, the syntax of many computer languages (such as Java and C#) belongs to this class[2].
Type 3	Regular expressions. This class puts additional constraints on the right side of the production rule. It will not be discussed here, since it is not relevant for following discussion.

[2] The Backus-Naur notation [19], which is often used to describe the format of computer languages, is a notation for context-free grammars.

Formal grammars are often used to analyze and identify structure in linguistics and computer science. But instead of starting with a string in a formal language and tracing it back to the start symbol in the grammar, the reverse process is also possible. That is, given the production rules for a formal language, we can generate arbitrary strings in the vocabulary of that language. This is easy since we can just apply randomly chosen production rules to the start symbol until only terminal symbols are left. For instance, given a grammar as shown in Figure 11, we could generate syntactically correct English sentences.

SCIgen [2] is an example of a generator, which builds random computer science papers using a context-free grammar. Several examples exist where papers created by SCIgen have been accepted by editors that were not aware that the content was computer generated. The most notable example is a paper by SCIgen, which was accepted in the Elsevier journal 'Applied Mathematics and Computation' in 2007 (a placeholder page for the now removed article can be found at [3]).

Here is an example of its output:

> ....In our research, we use pseudorandom methodologies to show that the transistor and the transistor are regularly incompatible. On a similar note, we emphasize that PloySerfism manages compilers. Indeed, DHTs and voice-over-IP have a long history of interfering in this manner. However, model checking might not be the panacea that theorists expected.
>
> In this position paper we explore the following contributions in detail. For starters, we prove that gigabit switches can be made unstable, cooperative, and adaptive. We demonstrate not only that the infamous stable algorithm for the construction of randomized algorithms by Kobayashi et al. is in Co-NP, but that the same is true for massive multiplayer online role-playing games.
>
> We proceed as follows. We motivate the need for extreme programming. Second, we place our work in context with the prior work in this area. Ultimately, we conclude.....

And here is a fragment of the hand-made context-free grammar SCIgen use:

> EVAL_ANALYZE_ONE → note the heavy tail on the CDF in EXP_FIG, exhibiting DIFFERENT EVAL_MEASUREMENT
>
> EVAL_ANALYZE_ONE → the many discontinuities in the graphs point to DIFFERENT EVAL_MEASUREMENT introduced with our hardware upgrades
>
> EVAL_ANALYZE_ONE → bugs in our system caused the unstable behavior throughout the experiments
>
> EVAL_ANALYZE_ONE → Gaussian electromagnetic disturbances in our EXP_WHERE caused unstable experimental results
>
> EVAL_ANALYZE_ONE → operator error alone cannot account for these results

Here the upper cased words are the non-terminals, which are to be substituted. This is only a small subset - the complete grammar contained approximately 3000 production rules. As is evident the grammar contains large contiguous fragments of text. Still the resulting output shows a great deal of variation.

Context Free Design Grammar

Formal grammars produce strings, that is, one-dimensional sequences of symbols. But for graphical content, we usually need two or three dimensions. One way to achieve this would be by interpreting the one-dimensional sequence as an encoding that produces a two- or three-dimensional result. For instance, each symbol in the sequence could be interpreted as an action such as 'move forward one unit', 'turn left 90 degrees', etc. This is the approach Lindenmayer systems [4] use to produce graphical output. However, Lindenmayer systems require two decoupled steps: the generation of a 1-dimensional sequence, and the subsequent transformation into a 2D or 3D illustration.

A more direct approach was suggested by Chris Coyne in his Context Free Design Grammar (CFDG) [5][3]. Similar to formal grammars a Context Free Design Grammar has production rules and non-

[3] Chris Coyne had previously experimented with using grammars for producing text. The SCIGen project mentioned in the

terminal and terminal symbols. The terminal symbols are now geometrical primitives - in Chris Coyne's original implementation circles and squares were used as the basic primitives.
The Context Free Design Grammar extends the syntax of the formal grammars by including transformation operators[4]. These transformation operators modify the current rendering state. Possible transformations include the rotation and scaling of the current coordinate system and modifications of the hue or saturation of the current drawing color. Notice that while rendering states have been introduced in the CFDG, the actual expansion of the non-terminal symbols is still context-free - it does not depend on the history or rendering state of the system. As is the case for formal grammars, one non-terminal symbol may have several different possible substitutions. The CFDG makes it possible to assign different weights to the different production rules for a given symbol.
The following is an example of a Context Free Design Grammar:

```
startshape SEED

rule SEED {
  SQUARE {}
  SEED { y 1.2 size 0.99 rotate 2.5 brightness 0.0015 }
}

rule SEED 0.04 {
  SQUARE {}
  SEED { y 1.2 s 0.9 r 1.5 flip 90 }
  SEED { y 1.2 x 1.2 s 0.8 r -60 }
  SEED { y 1.2 x -1.2 s 0.6 r 60  flip 90 }
}
```

Now by applying the transformation rules to the start rule, outputs such as the ones in Figure 12 can be created. These images were created using Context Free Art [6], an implementation of a Context Free Design Grammar created by Mark Lentczner and John Horigan.

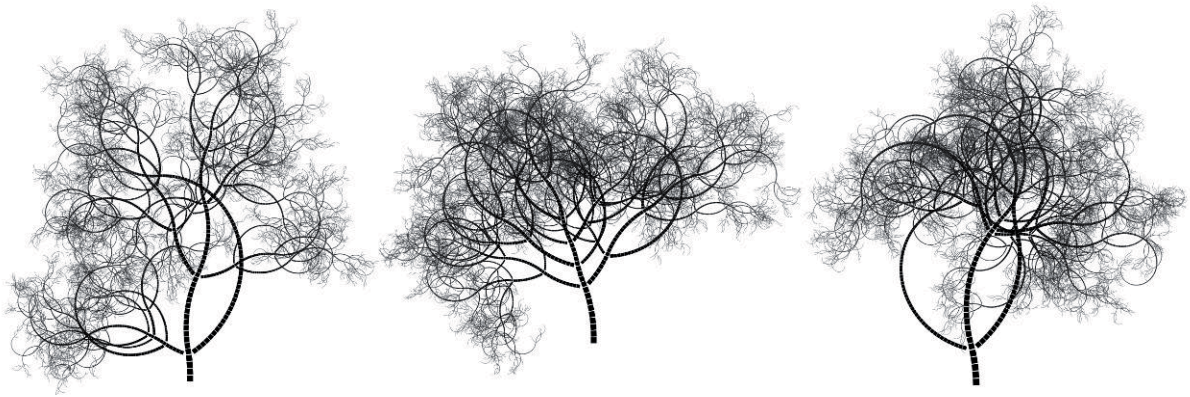

Figure 12: An example of a Context Free Art system. The three structures are instances of the same system, but with different random seeds.

Chomsky's formal languages consist of *finite* strings. In contrast, systems specified by a context-free design grammar often emit an infinite number of terminals. In practice, this is overcome by applying some kind of termination rule, such as stopping the production if the primitives become too small to be visible. Also, whereas the natural representation of a string in a formal language is a sequence of symbols, Context Free Design Grammars produce rule expansions which are naturally represented in abstract, hierarchical trees (a rule may spawn one or more rule calls, corresponding to new branches in these hierarchical trees).

previous section was inspired by a high school paper generator created by Chris Coyne.

[4] The Context Free Art developers use a different terminology - their *adjustment rules* correspond to the *transformation operators* in this paper.

Similar to Context Free Design Grammars, most Lindenmayer systems also grow potentially unlimited strings. Lindenmayer also introduced a notation for representing hierarchical trees in his Lindenmayer systems [4]. By interpreting brackets in the output as creating new branches, it becomes possible to create tree-like hierarchical structures. An output such as A[B][C] would be interpreted as A being the root of the tree with two branches, B and C[5]. Similar to Context Free Design Grammars, Lindenmayer also described the use of multiple production rules with different weights - something he referred to as Stochastic Lindenmayer systems. This means the Context Free Design Grammars are very close in expressive power to what Lindenmayer would have classified as a Stochastic Context-Free Bracketed L-system. CFDG systems offer a couple of advantages, though. The CFDG unites the substitution rules and the geometrical operators. This makes the representation slightly more intuitive and it makes it possible to implement more flexible termination rules. For instance, the rule expansion can be terminated, whenever the geometrical primitives become too small to be visible. A similar termination rule would be difficult to implement in a Lindenmayer system, since the rule expansion is separated from the geometric representation.

Structure Synth

Structure Synth extends Context Free Art into three dimensions. Its syntax is derived from the original Context Free Design Grammar but with a few key differences.

Termination criteria: In Context Free Art the recursion automatically terminates when the objects produced are too small to be visible. This is a very elegant solution, but it is not possible to extend to a dynamic 3D world, where the user can move and zoom with the camera. Instead, several options exist in Structure Synth for terminating the rendering, such as specifying a maximum recursion level, or a maximum number of objects, or setting a fixed minimum size.

Transformations and primitives: Since Structure Synth operates in three-dimensional space, a new set of transformations and primitives was necessary. The transformations include translations, flipping and rotations about the three Cartesian axes, and the new set of primitives include volumetric objects such as spheres, boxes and lower dimensional objects such as triangles, lines and dots.

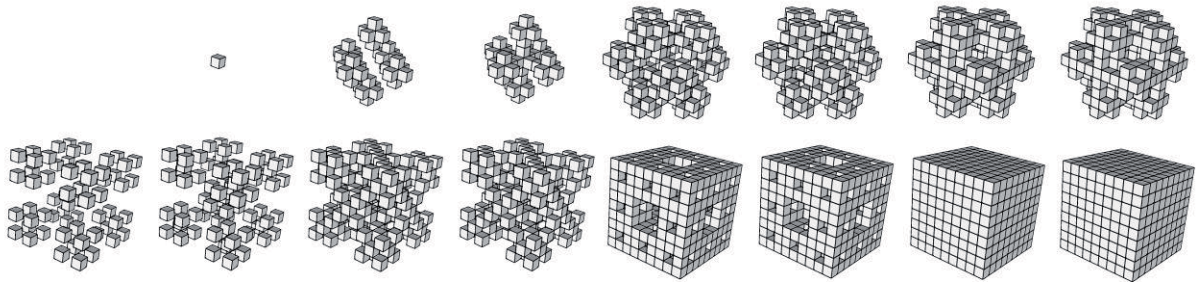

Figure 13: Rule retirement and substitution. An extension allows rules to be changed after a number of iterations. In this case, a rule makes spatial subdivisions until a specified recursion depth is reached - this makes it possible to create Menger fractal variations such as these [7].

Language extensions: A few new features were added to Structure Synth. A rule retirement system makes it possible to substitute one rule for another after a specified maximum recursion level. Even though this is a slight violation of the context-freeness, this was included in order to make it possible to create objects such as those in Figure 13. A new coloring system was also introduced making it possible to use random colors from different color pools, including sampling colors from a bitmap file. Random Seed Synchronization (See Figure 19) makes it possible to synchronize the random number streams whenever a rule branches (calls two or more new rules).

Language reductions: The start shape is no longer explicitly declared. Instead, all commands at top-level scope are implicitly converted into an anonymous start rule. In addition, Context Free Art defines two different forms of modifiers, which are placed after the rule designator: square brackets and curly brackets, where the modifier order is not significant for the square brackets. Structure Synth, on the

[5] Implementation-wise, the left bracket would push the current state, and the right bracket would pop the current system state.

other hand, only uses curly brackets placed before the rule designator, and the transformation order is always significant.
The following Structure Synth system creates the output shown in Figure 14:

```
set background white

{ h 30 sat 0.7 } seed
{ ry 180 h 30 sat 0.7 } seed

rule seed weight 100 {
  box
  { y 0.4 rx 1 s 0.995 b 0.995 } seed
}

rule seed weight 100 {
  box
  { y 0.4 rx 1 ry 1 s 0.995 b 0.995 } seed
}

rule seed weight 100 {
  box
  { y 0.4 rx 1 rz -1 s 0.995 b 0.995 } seed
}

rule seed weight 6 {
  { rx 15 }  seed
  { ry 180 h 3  }  seed
}
```

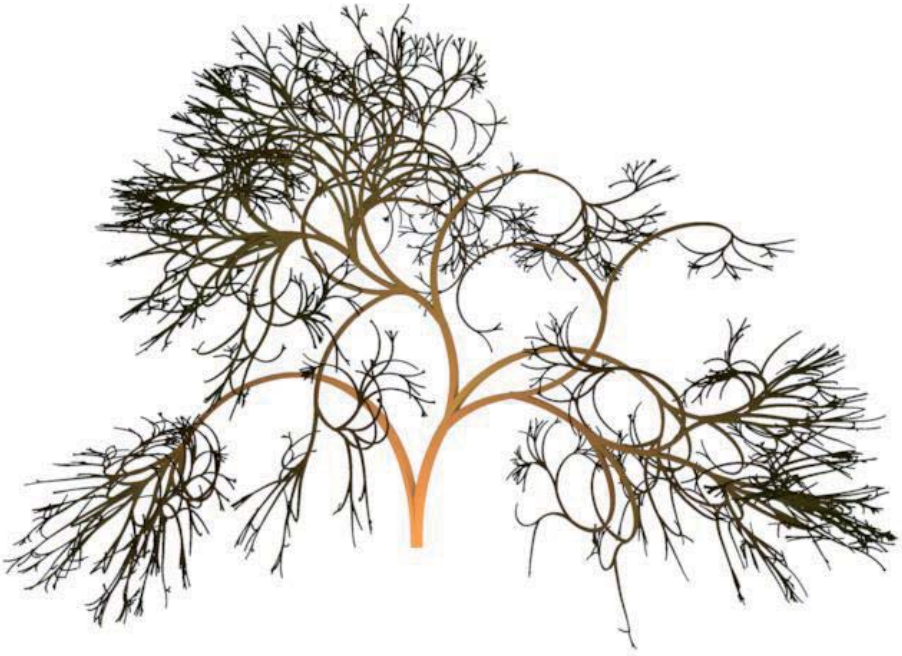

Figure 14: Three-dimensional version of Figure 12 created in Structure Synth.

Differences to procedural programming

A Structure Synth grammar like the one above may look similar to a normal computer program – the syntax is quite close to the syntax of procedural programming languages like C, Java, Pascal, or Basic. And instead of thinking of the system as a grammar and its output as strings in the language specified by this grammar, it is perhaps easier to think of the grammar as a restricted subset of an ordinary computer language, just without parameter passing and conditional logic.
The similarities may be a bit deceptive though, since there are two major differences: functions (which are the rules in the CFDG terminology) may have multiple definitions each with an arbitrary weight. Moreover, recursion is handled *breadth first*.
The last point requires further explanation: Whenever a procedural programming language executes a function or procedure, it does so in sequential order – the individual statements in the function are executed in the order of appearance[6]. If one of the statements is a procedure call, this procedure is executed and must complete before the next statement is executed. The state of the currently executing function (the return address pointer, local variables, etc.) is typically stored in stack frames on a call stack, in order to be able to return after executing a function. Put differently, this means the function call tree for the program is traversed *depth-first*. Recursion in Structure Synth is handled differently. Instead of a call stack, there is generational queue system: whenever a rule is encountered, all sub rule calls and primitives in the rule definition are pushed onto a new queue that will be evaluated at the next generation. This means the rules are traversed *breadth first* – all calls at the same recursive depth are processed at the same time. Consider the following example:

```
Procedure recurse() {
  recurse();
  drawBox();
}
```

```
Rule recurse {
  recurse
  box
}
```

Example of recursion in a traditional computer language to the left and in Structure Synth to the right.

A traditional programming language would never reach the ‘drawBox()’ function call. It would recurse until the call stack overflowed. In contrast, in Structure Synth the first generation would process both the ‘recurse’ and 'box' statement. (The ‘recurse’ statement would be expanded into new 'recurse' and 'box' statements and scheduled for execution on the next generation queue).

Technical implementation notes

Structure Synth provides a graphical environment with a multiple tab editor, syntax highlighting, and OpenGL preview. Besides the integrated OpenGL view, it is possible to export structures to third-party software (such as Sunflow [8] and POV-Ray [9]) using an extensible template based export system.
Structure Synth is written in C++ using the Nokia Qt framework [10]. It uses the OpenGL API [11] for visualization and the Mersenne Twister RNG [12] for random numbers (the C standard library random number generator is insufficient, since two independent random number streams are used: one for geometry and one for colors). It uses a hand-written recursive descent parser to parse the grammar, from which a binary representation of the transformation rules is created. All geometrical transformations (translation, rotation, and scaling) are stored in 4x4 (homogeneous) matrices.
Structure Synth is open source (dual licensed under the GPL and LGPL [13]) and cross-platform (including Windows XP and Vista, Mac OS X, Linux, and FreeBSD). The source and binary files are hosted at SourceForge and can be downloaded from [14].

[6] The compiler may have some liberty to reorder the instruction order between the defined sequence points in the language, but this is not relevant for our discussion.

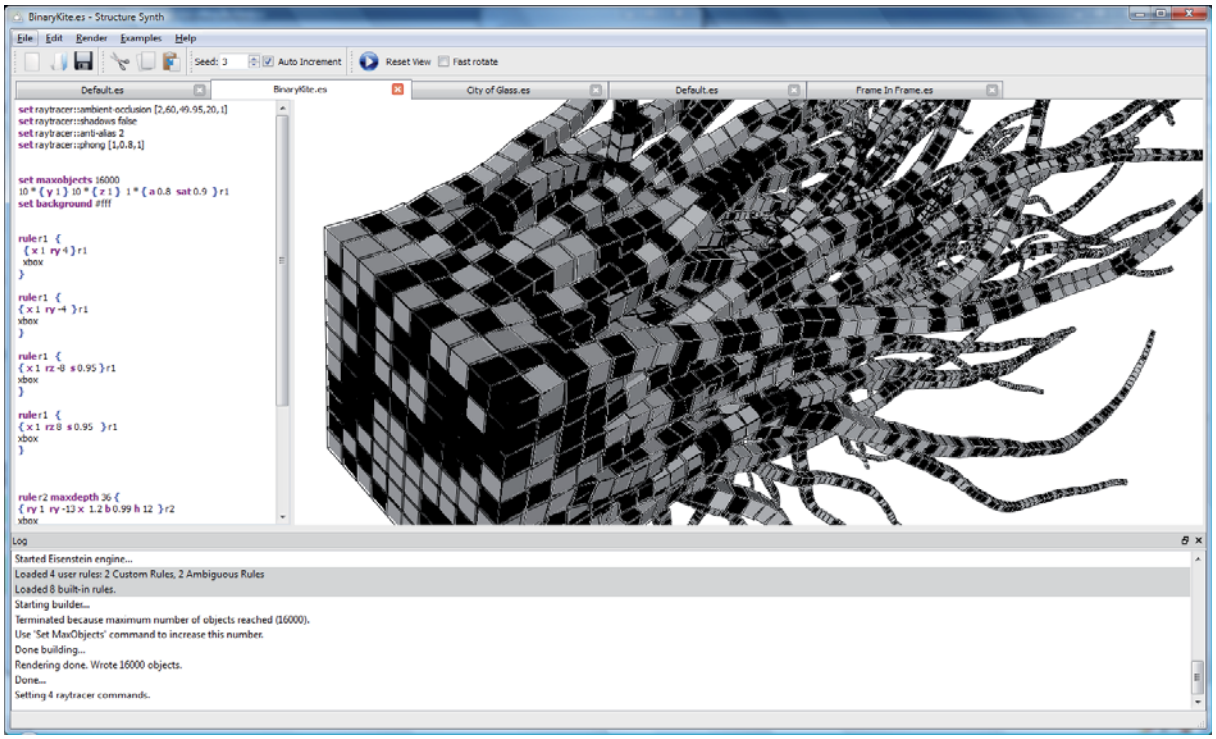

Figure 15: Structure Synth graphical user interface showing a tabbed interface, with a syntax-highlighting editor and an integrated OpenGL preview.

Examples of systems

The purpose of this section is to illustrate some typical aspects of Structure Synth.
Structure Synth makes it possible to formulate systems, which are deterministic (reproducible) each time the system is instantiated, but also makes it possible to create stochastic systems with inherent random behavior (see Figure 17). Yet, the recursive nature of both types of systems often results in very complex images. Many of the stochastic Structure Synth systems also display a lot of diversity. Some examples of different instances with the same system are shown in Figure 16. Stochastic systems with near-continuous transformations (meaning the state is changed slowly) often look organic or biological. (See Figure 18)
While some deterministic systems (such as the Menger fractal in Figure 13) may exhibit self-similarity, it is also possible to create stochastic systems which are self-similar in Structure Synth. This may be done by using the random seed synchronization, which makes it possible to spawn branches that will be governed by identical random number sequences[7].

[7] Normally ambiguous rule substitutions are resolved using a random number generator. This means that two different branches, each starting with identical symbols, may end up with different expansions. The random seed synchronization is a special command for synchronizing two different branches - ensuring their expansion will be identical.

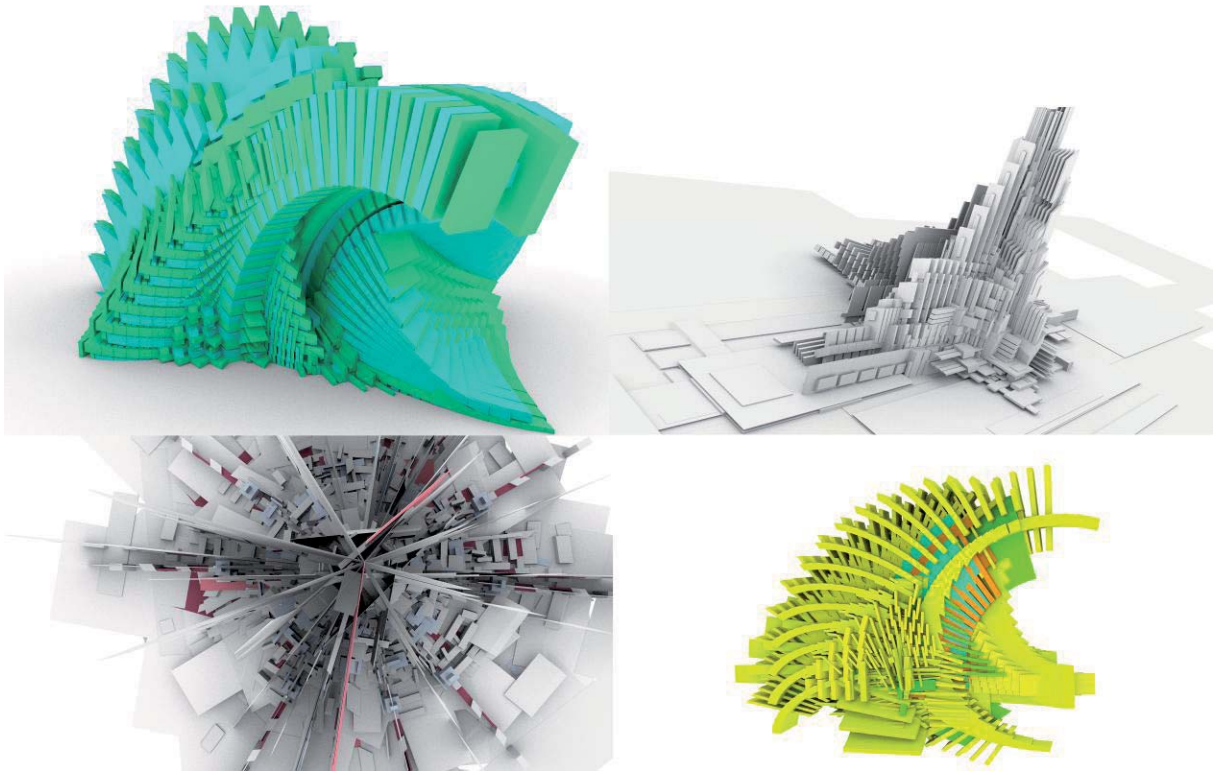

Figure 16: Diversity. All of the images above are instances of the same grammar but with different random seed. Images: The Nabla System [15], with seeds 29, 338, 7 (radial), 201.

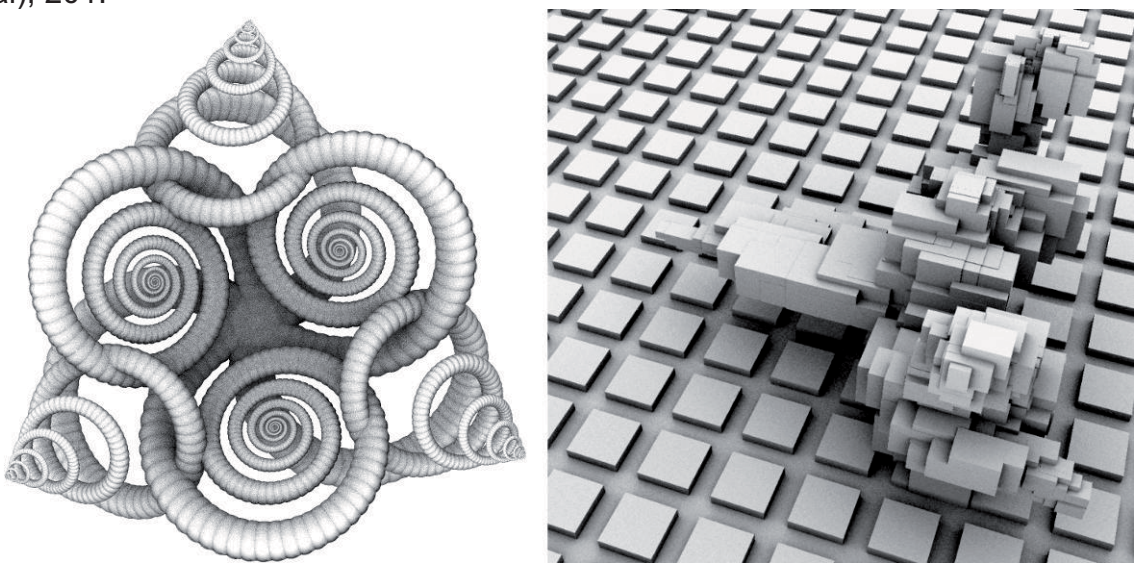

Figure 17: Deterministic versus stochastic systems. The picture on the left has no ambiguous rules. In contrast, the structure on the right will be different every time the system is instantiated. [16]

Figure 18: Organic. These two images are variants of the Nouveau system [17] - a system based on random continuous transformations. Such images often have an organic appearance.

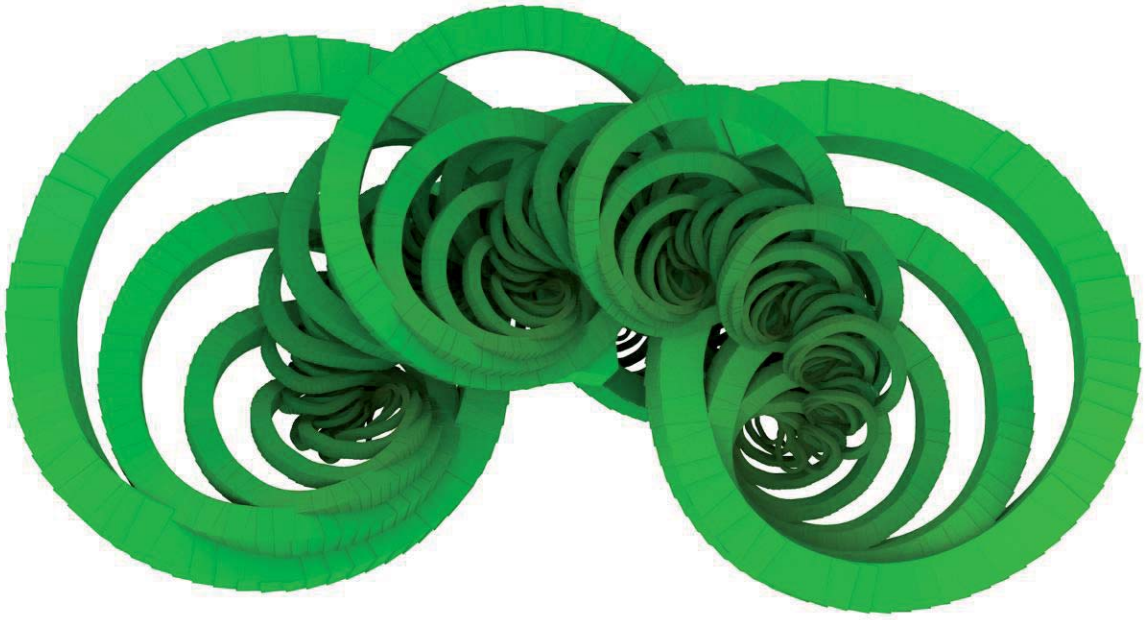

Figure 19: Stochastic self-similarity. The principal form of the ring system above is stochastic, yet the system above copies itself on many scales [18].

Constrained systems and Generative Art

The first part of this paper discussed how systems of various complexity (expressive power) can be generated using formal grammars, and how this led to design grammars and Structure Synth.
So why restrict Context Free Art and Structure Synth to context-free systems? It is well known that it can be computationally hard [19] to *analyze* the structure of context-sensitive systems, but it is not much harder to *generate* structures based on a more powerful grammar. The grammar in Structure Synth could easily be extended to context-sensitive systems. However, context-free systems have the nice property of being complex enough to be interesting, while not being omnipotent (in the general-purpose programming languages sense), making them very suitable for *generative art*:
Even though no definitive definition of 'generative art' exists, it has been suggested that generative art is about creating and exploring systems, without being too much in control (see e.g. [20]). When generating structures, it should not be possible to anticipate how a given structure turns out by looking at the rules. There should be a sense of non-determinism and surprise in the result. The system needs not to necessarily be driven by random choices in order to achieve this – the Mandelbrot set is a good example of this: nobody would have been able to imagine how complex images a simple system like "$z \rightarrow z^2+c$" could create, yet there is nothing stochastic in the generation of Mandelbrot sets. Choosing

to work within a restricted rule system is a way to give up some control and to be forced to think differently. It becomes necessary to explore and work within the limitations of the system, which may lead to interesting and unexpected results.
More generic languages, for instance the popular Java-based Processing environment [21], have no limitations in expressiveness[8]. Does this mean that Processing is not suitable for generative art, because of its universal expressive power? Well, the answer is of course that Processing is very suitable and is widely used by the generative art community. In fact, any Structure Synth or Context Free Art system could be created in Processing/Java because of this universal power. However, Processing is also suitable for many other applications, such as Data Visualization and other non-generative tasks. Context Free Art and Structure Synth on the other hand force you to explore generative systems.

Conclusions and future work

This paper introduced Structure Synth and described its heritage from Chomsky's grammars and the Context Free Design Grammar by Chris Coyne. It has been argued (but not formally proved) that these systems are closely related to stochastic context-free bracketed Lindenmayer-systems, but different from procedural programming languages. Finally, the potential benefits from working with constrained systems have been discussed.
The next version of Structure Synth will focus on two new features: a new internal raytracer for creating high-resolution output directly in Structure Synth, without having to use external third-party software. It will also include automation and scripting of the structure creation using a built-in JavaScript interpreter: this will make it possible to vary internal grammar parameters and create animations. Integration with other programs (such as VVVV [22] and Blender [23]) is in progress and there will likely be better integration with other software systems as well.
On a longer time frame, there are several ideas that might be pursued. One possibility is to extend Structure Synth to make it suitable for live performances - by making it possible to interact with and control the model building in real-time. Another idea is to implement topological operations on grid meshes (using operations such as those used by TopMod [24]) instead of working with fixed primitives. Finally, several people have suggested a user interface for automatically creating a set of 'mutated' systems, making it possible to evolve the systems in a direction supervised by the user (evolutionary art / design).

Acknowledgements

I would like to thank René Thomsen and Kamma O. Hansen for commenting on and proofreading this paper. I would also like to thank the users of Structure Synth who have provided valuable and encouraging feedback. In particular, I would like to thank the Structure Synth Flickr community for many interesting discussions and suggestions.

References

[1] N. Chomsky (1956): Three models for the description of language. IRE Transactions on Information Theory (2): 113–124
[2] J. Stribling, M. Krohn, D. Aguayo (2005): SCIgen - an automatic cs paper generator, pdos.lcs.mit.edu.
[3] R. Mosallahnezhad (2007): Cooperative, Compact Algorithms for Randomized Algorithms. Applied Mathematics and Computation, Elsevier.
[4] P. Prusinkiewicz, A. Lindenmayer (1990): "The Algorithmic Beauty of Plants."
[5] Chris Coyne's CFDG description: http://www.chriscoyne.com/cfdg/.

[8] Java, like all other general-purpose programming languages, is Turing complete - meaning they can express arbitrary computations. This places their output in the most powerful 'Type 0' category of the Chomsky Hierarchy [19].

[6] C. Coyne, M. Lentczner, J. Horigan: Context Free Art. http://www.contextfreeart.org/
[7] Menger variations at Flickr: http://www.flickr.com/photos/syntopia/3199111727/
[8] Sunflow. http://sunflow.sourceforge.net/
[9] POV-Ray - The Persistence of Vision Raytracer. http://www.povray.org/
[10] Nokia Qt. http://qt.nokia.com/
[11] OpenGL. http://www.opengl.org/
[12] Mersenne Twister. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
[13] GPL and LGPL. http://www.gnu.org/licenses/gpl.html
[14] Structure Synth at SourceForge: http://structuresynth.sourceforge.net/
[15] Nabla System at Flickr: http://www.flickr.com/photos/syntopia/3401189363/
[16] Deterministic versus Stochastic system: http://www.flickr.com/photos/syntopia/3909688627/ and http://www.flickr.com/photos/syntopia/2244330735/
[17] Nouveau System. http://www.flickr.com/photos/syntopia/3571168238/
[18] Synctor. http://www.flickr.com/photos/syntopia/3272377029/
[19] J. C. Martin (1991): Introduction to Languages and the Theory of Computation. McGraw-Hill.
[20] P. Galanter (2003): What is Generative Art? Complexity Theory as a Context for Art Theory. Generative Art 2003 Conference.
[21] Processing 1.0. http://processing.org/
[22] VVVV: a multipurpose toolkit. http://vvvv.org
[23] Blender - open source 3D content creation suite. http://www.blender.org/
[24] TopMod3D - topological mesh modeler. http://www.topmod3d.org/