Synthesis of Incidental Detail as Composable Components in a Functional Language

Richard Roberts, Timothy Jones, John Lewis
School of Engineering and Computer Science
Victoria University of Wellington
New Zealand
{riro, tim, jplewis}@ecs.vuw.ac.nz

***Abstract*—Designs of real-world objects must include small-scale details in order to appear plausible. Often the overall character of this 'incidental' detail matters, but the exact shape and placement of each feature is unimportant. The creation of these details often consumes artist time when pattern generation techniques could automate the process instead, but many current pattern generation just shift the effort from manual modeling to custom per-object programming. Machine learning approaches to pattern synthesis are promising, but successful efforts have been mainly restricted to stochastic patterns. In this paper we investigate how detail patterns may be encoded using techniques from functional programming, and contribute a DSL for describing and composing these details. By allowing pure descriptions to be seamlessly composed together we produce a high-level process for creating structured and semi-structured patterns.**

I. Introduction

Pattern analysis and synthesis are topics that have long been prevalent in computer graphics. The general problem is that pattern production is a complex aesthetic domain, and pattern design may (or may not) feature numerous descriptive traits such as structured, stroked, clustered, organic, repetitive, or entropic. Models for pattern categorization and creation demand both generality and simplicity [1, p. 1630].

Patterns are important in modeling, texturing, and animation. They are commonly used in the construction of city layouts, architecture, natural environments, and fur and skin. Unfortunately, pattern production can be both time-consuming and repetitive for artists. The parameters required to describe a given pattern can quickly become too numerous to control, or too specific to allow for a broad range of outputs.

Approaches towards simplistic and generalized pattern analysis and synthesis have been a focus of recent research. Current techniques offer methods to parameterise repetitive elements in stroke or pixel based patterns, which can be used to generate outputs featuring similar elements, as described in section II. Some methods have been quite successful at describing and replicating patterns, but often for only a restricted subset of the aesthetic domain. The more general solutions become too complex to be intuitively communicated through the interface.

We contribute a novel approach that provides a generalized and intuitive workflow for pattern production. We focus on automation of repetitive processes, where stylistic and compositional control will not be inhibited. An artist uses meaningful descriptions to iteratively expand a simple set of coordinates into complex structured or semi-structured data, representative of a convincing pattern that can appear organic, structured, repetitive, stochastic or any mixture of the above. This approach is equivalent to a hierarchical shape grammar, though expressed in terms of higher-order functions as described in Section III. It can rapidly author highly complex patterns, enabling a multitude of detailing and texturing applications.

Our primary application is the construction of fine scale details; however, since our approach can also produce stochastic textures, the range of applications is not limited to these examples. The construction of these details using our system is efficient and intuitive.

II. Related Work

Research on the computer synthesis of patterns has spanned more than five decades [2] and occurs in different fields including biology [3], physics [4], mathematics [5] and computer graphics [6]. Our brief survey will group this literature into approaches appropriate for random versus structured patterns. Informally speaking, random patterns are those in which there is no exact symmetry or repetition of pattern elements, and conversely, structured patterns are those which do have symmetries.

Random pattern synthesis approaches have been used to simulate textures, terrains, camouflage, and other phenomena. Random process 'noise' models [7] appear as a primitive in computer graphics shading languages and are used to approximately simulate dirt, dust, clouds, and other unstructured textures. Random fractals [8] are are a well-known special case of noise, having a self similar power spectrum of the form $1/\omega^p$. These techniques are not generally appropriate for structured textures. The popular and powerful nonparametric sampling texture synthesis approaches [9], [10] have resulted in many improvements and extensions [11]. These approaches synthesize new pixels by sampling from a nonparametric distribution obtained by finding regions in a reference texture that are similar to the neighborhood being synthesized. They can simulate both unstructured and 'semistructured' textures and can generate very high quality textures, however the synthesis also fails to produce plausible textures in some cases. Random textures have also been modeled as a machine learning problem [1], [12], [13].

Prototypical examples of structured patterns are the pattern of rivets or similar details on a train, robot, or spaceship. Simple patterns such as a line or grid of objects are easily

simulated using loops and translations. Group theory provides an appropriate framework for studying the possible types of simple patterns [14]. More complex structured patterns have been generated using special-purpose algorithmic approaches. For example, L-systems [15] successfully model plants and plant growth. Shape grammars have been developed to model cities and city growth, among other things [16], [17].

The general notion of 'algorithm' of course subsumes all random and structural pattern generation approaches. The problem with exploring the general space of algorithms, however, is that it is simply too large. For example, an eight byte program is too tiny to be useful. The number of distinct 8-byte programs is 2^{64}. Imagining a current machine with a roughly four gigahertz clock rate (approximately 2^{32} operations per second), running each program for N steps would require $2^{32} \cdot N$ seconds, which is roughly 136 years for each step. Genetic algorithms typically address this complexity by introducing custom instructions or languages designed to increase the percentage of useful programs [18]. The size of the programs that can be explored is nonetheless severely limited.

In this work we attempt an alternate and quite general algorithmic approach, using the concept of higher order functions to *invent* pattern synthesis algorithms. We will show that higher order functions are a useful way to explore the 'space' of structured and semi-structured patterns.

III. Method

Our implementation describes detail as a nesting of progressively finer incidental detail, represented as a forest of iterations. Each iteration indicates properties such as the geometry to create and where to place it. This structure is converted to operations in the Maya modelling tool to produce a final model. We provide a mechanism for composing detail trees in either direction, so that each one can easily be reused as finer detail in even larger trees or refined by applying further fine detail on specific branches.

The construction of the detail trees takes the form of a domain specific language in Haskell. We take advantage of Haskell's curried function encoding [19] to automatically produce tree builders, and then specialise these builders into the DSL using free monads [20]. While the declaration of the detail generation is currently bound to this DSL in Haskell, the encoding does not require techniques unique to either language, and allows for generating the detail information from a different (potentially graphical) user interface. We aim to eventually allow the detail to be generated automatically, and then displayed and modified by an artist.

A. Representing Detail Generation

We represent a complete detail generation process as a forest of general trees whose elements describe a step of the iteration to apply to its parent shape in the tree. Each iteration produces new geometry, selected from a set of simple three-dimensional primitives, a technique for positioning the geometry on its parent, the amount to scale down the detail relative to its parent, and the new direction of each shape's up-vector.

The technique for selecting points to generate detail on a parent shape depends on the primitive shape that will be the parent of the detail, and include selecting opposite faces of a cube on a relative axis, selecting a number of points in a loop around the curved surface of a cylinder or sphere, and so on. A simplified example of a detail tree with geometry and positioning as elements in a forest, completed by adding an initial shape as the root node, is provided in Figure 1.

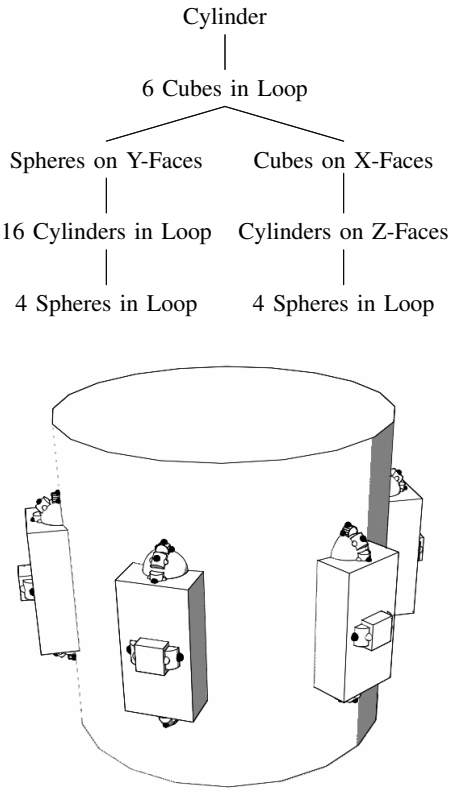


Fig. 1: A simplified detail tree and the resulting model

Encoding the construction of these forests in Haskell has a curious outcome. Haskell functions that require multiple parameters are *curried* [19]. The technique of currying is derived from the observation that a function with multiple parameters can be equivalently represented by defining it to only take a single argument and instead return a new function that takes the second argument. The outcome is that rather than writing $f(x, y)$, we write $f(x)(y)$.

If we represent an entire iteration of detail with a type $Detail$, then a constructor $Forest.singleton$ for a forest containing a single tree has two parameters: the detail of the top node, and a forest of children. In its curried form, the constructor will have the following type:

$$Detail \rightarrow (Forest\ Detail \rightarrow Forest\ Detail)$$

The outcome of applying the constructor to *just* a $Detail$ value is a function from a forest to a larger forest. We can use this function as a value, where it acts as a functional 'builder' and can be composed with other forest builders using standard function composition to produce builders of larger forests. The final detail generation tree can be produced by applying the builder to an empty forest, and placing an initial shape as the root node of the resulting forest.

$$
\begin{aligned}
&\mathit{firstBranch} = \mathbf{do}\\
&\quad \mathit{detail}\ \mathit{Sphere}\quad (\mathit{CubeFaces}\ \mathit{yAxis})\ 0.6\\
&\quad \mathit{detail}\ \mathit{Cylinder}\ (\mathit{SphereLoop}\ 8\ 0)\quad 0.3\\
&\mathit{secondBranch} = \mathbf{do}\\
&\quad \mathit{detail}\ \mathit{Cube}\quad\ (\mathit{CubeFaces}\ \mathit{xAxis})\ 0.4\\
&\quad \mathit{detail}\ \mathit{Cylinder}\ (\mathit{CubeFaces}\ \mathit{zAxis})\ 0.7\\
&\mathit{applyDetail} = \mathbf{do}\\
&\quad \mathit{detail}\ \mathit{Cube}\quad (\mathit{CylinderLoop}\ 6\ 0)\ (0.2, 0.4, 0.2)\\
&\quad \mathit{branch}\ [\mathit{firstBranch}, \mathit{secondBranch}]\\
&\quad \mathit{detail}\ \mathit{Sphere}\ (\mathit{CylinderLoop}\ 4\ 0)\ 0.4
\end{aligned}
$$

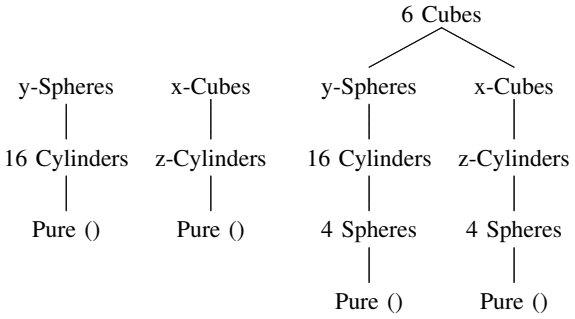


Fig. 2: DSL encoding of Fig. 1 and resulting trees

We recursively apply a transformation to this tree to generate Maya commands in Python and build the model described by the representation. The higher-order detail generation is both concise and strongly typed because of its Haskell implementation.

The use of higher order functions was inspired by McDermott et al. [21], who demonstrate the advantages of the approach within the aesthetic domains of architectural modeling and music synthesis, and by Lewis et al. [22], which uses higher order functions to synthesise distinctive icons for desktop interfaces. Two of the advantages identified are non-entropic mutations and compressible phenotypes, entailing that data can be manipulated without losing coherence and that the grammars are efficient to maintain. This contributes a generalized and dynamic approach to synthesis, enabling a broad range of outputs.

B. Free Monads and a DSL for Tree Generation

While the interface for building detail forests given above is useful, the higher-order function suffers from the limitation that once the detail generation of the children is applied, the result is a complete forest that can only have further detail applied to it by manually traversing the tree structure. A preferable interface for generating detail would allow us to continue adding smaller detail until the whole tree is ready to be processed.

To that end, we have taken advantage of a common Haskell technique for generating DSLs, wrapping a modified form of the detail tree encoding into a *free monad* [20]. While a formal description of free monads in Haskell is outside the scope of this paper, they are in a practical sense a general technique for creating a recursive, monadic structure. Monads are an important interface for sequencing computations in Haskell, and allow use of the **do** notation, which sequences code imperatively with indentation-specific syntax.

The result of wrapping the detail tree encoding in a free monad produces a type $\mathit{DetailGen}$, which can be thought of as representing a forest of Detail which may contain 'pure' leaves acting as placeholders, indicating that the detail may be continued at those points. The monadic interface can be used to 'bind' a subtree to the existing tree, replacing each of the pure nodes with a copy of the subtree. The result is that each line of code in the DSL specifies the values to appear at a particular depth in the tree, attached to the detail specified by the line above.

We have defined our DSL as a set of utility functions on top of this structure. Further detail can be applied at the pure nodes in the tree with the detail function, which leaves a new pure node at the bottom of the new detail node.

$$\mathit{detail} :: \mathit{Shape} \rightarrow \mathit{Selection} \rightarrow \mathit{Scale} \rightarrow \mathit{DetailGen}\ ()$$

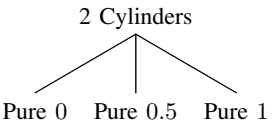
Each pure node can be replaced by a forest of detail generations with branch, or a group of pure nodes with $\mathit{pureBranch}$:

$$
\begin{aligned}
&\mathit{branch} :: [\mathit{DetailGen}\ a] \rightarrow \mathit{DetailGen}\ a\\
&\mathit{pureBranch} :: [a] \rightarrow \mathit{DetailGen}\ a
\end{aligned}
$$

Finally, a path may choose to prevent further detail generation below it with done, which is equivalent to an empty branch: $\mathit{branch}\ [\,]$.

The detail generation can now be written in the DSL. This is demonstrated in Figure 2. The use of $\mathit{branch1}$ and $\mathit{branch2}$ demonstrates the composability of $\mathit{DetailGen}$: single and multiple iterations are indistinguishable from one another, and a tree of detail generation can be defined and reused any number of times in a larger tree. Applying more detail

$$
\begin{aligned}
&\mathit{segment} = \mathbf{do}\\
&\quad \mathit{detail}\ \mathit{Cylinder}\ (\mathit{CubeFaces}\ \mathit{xAxis})\ 0.5\\
&\quad \mathit{pureBranch}\ [0, 0.5, 1]
\end{aligned}
$$



$$
\begin{aligned}
&\mathit{cubeSegment} = \mathbf{do}\\
&\quad \mathit{height} \leftarrow \mathit{segment}\\
&\quad \mathit{detail}\ \mathit{Cube}\ (\mathit{CylinderLoop}\ 4\ \mathit{height})\ 0.2
\end{aligned}
$$

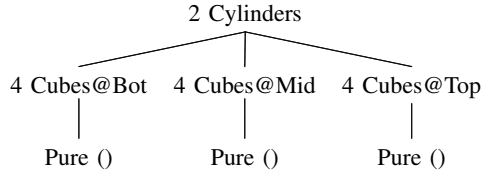


Fig. 3: An example of using values in pure nodes

simply moves the pure nodes in the tree. This allows for the development of libraries of detail generation trees which can be reused in any context.

The real power of this DSL is that the use of free monads causes values to inhabit the pure nodes. Previously, when a subtree was bound to an existing tree the same subtree appeared in place of each pure node. The monadic interface allows the shape and contents of the resulting subtree to depend on the value in the pure node instead. This is demonstrated in Figure 3, where values in pure nodes are used to apply the same detail at different heights on a cylinder. The DSL achieves this by creating three pure nodes after the cylinder in the generation tree, with each pure node containing a different height proportion to apply detail at.

While monads are often associated with side-effects, the DSL consists of entirely pure code. Large passes of detail generation can be run in parallel without interference. As the DSL is already implemented in a monad, it would be fairly trivial to also add more detailed mechanics to the language, such as the ability to generate details randomly.

The only important functionality that is missing from this package is that there is no guarantee that the method for selecting points to add detail on a shape is actually valid for that shape. The DSL exacerbates this problem, as multiple pure nodes in a single tree may apply detail to completely different shapes. Solving this problem while preserving type safety involves non-standard Haskell extensions, and is beyond the scope of this paper.

IV. Results

To demonstrate the outcome of using the DSL for pattern synthesis, we have generated two model sets with detail generation. The first example in Figure 4 features an intentionally designed structure, while the example in Figure 5 using similar *Detail* types repetitively. The presented images were produced by generating executable Python scripts from the finalised *DetailGen*. These scripts automatically build a Maya scene file, which was then rendered using 3Delight.

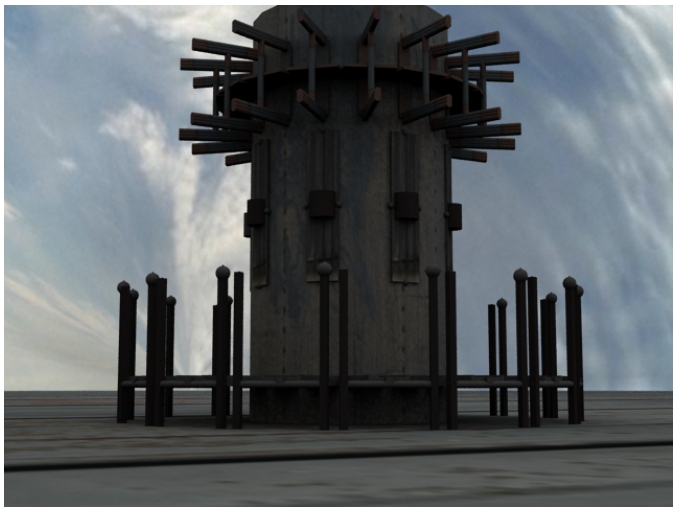


Fig. 4: A model created with a custom *DetailTree*

Objects intended to appear intentionally designed may require meaningful details at all scales. Creating an object akin to the one in Figure 4 is simple when using the *Detail* types. The central cylinder is the root detail, which emits different patterns of details looping horizontally around itself. This includes the walkway-like plateau, the plated middle area, and the fencing at the base. Each of these objects generate further details using their own *Detail* values. This model required about 10 minutes of experimentation in the DSL to author.

Figure 5 is an example of repetitively nesting a small set of detail generation trees. Using a hierarchical approach to cluster the *DetailGen* values enables fast production of complicated geometry. An artist is then able to focus on aesthetic refinement as opposed to manual creation, enabling serendipitous creativity. The minor changes to the repetitive structure are expressed simply in the DSL, without requiring repetitive code from the author.

Our system ensures robust repetition when iteration and mutation is required; any attribute of the pattern can be mutated at any point in the tree. For instance, the piston-like connections between the base and cylinder objects are scaled differently per row. Since nesting *DetailGen* trees is powerful, an artist may wish to build a library of expansions, effectively building their own grammars for pattern expansion. The construction of an effective interactive interface, outlined in Section V, aims to encourage this.

V. Conclusion

We have presented a technique for pattern categorization and synthesis using higher-order functions in Haskell, and demonstrated how this interface can be generalised into a tree building DSL with free monads. This allows for a broad range of pattern possibilities while ensuring composability of individual detail generation computations.

Due to the sheer size of the pattern aesthetic domain, general solutions quickly become complex to interact with. The advantages of the higher order functions reduce the impact of this issue in our approach, but do not solve it. Artist time is still required to produce and adjust the code written in the DSL, but the language lends itself to extensibility in producing initial programs randomly. A more approachable interface for interacting with the generated model is also a future goal, allowing non-technical artists to interact with the package and directly manipulate nodes in the detail generation tree.

A. Future Direction

The DSL in its current state is not helpful to non-technical artists and still requires each piece of detail to be manually described at some point in the development process. As discussed in Section III-B, extending the language with further features, such as generating random detail, would be relatively trivial, but extending the generation process for more general use through a visual interface for manipulating the language would be most desirable.

The output pattern data can be readily applied to modeling and texturing, but also has other useful applications. In future research the authors aim to further the vocabulary of our system, which may enable the three-dimensional output to be used as input to simulation software and animation techniques.

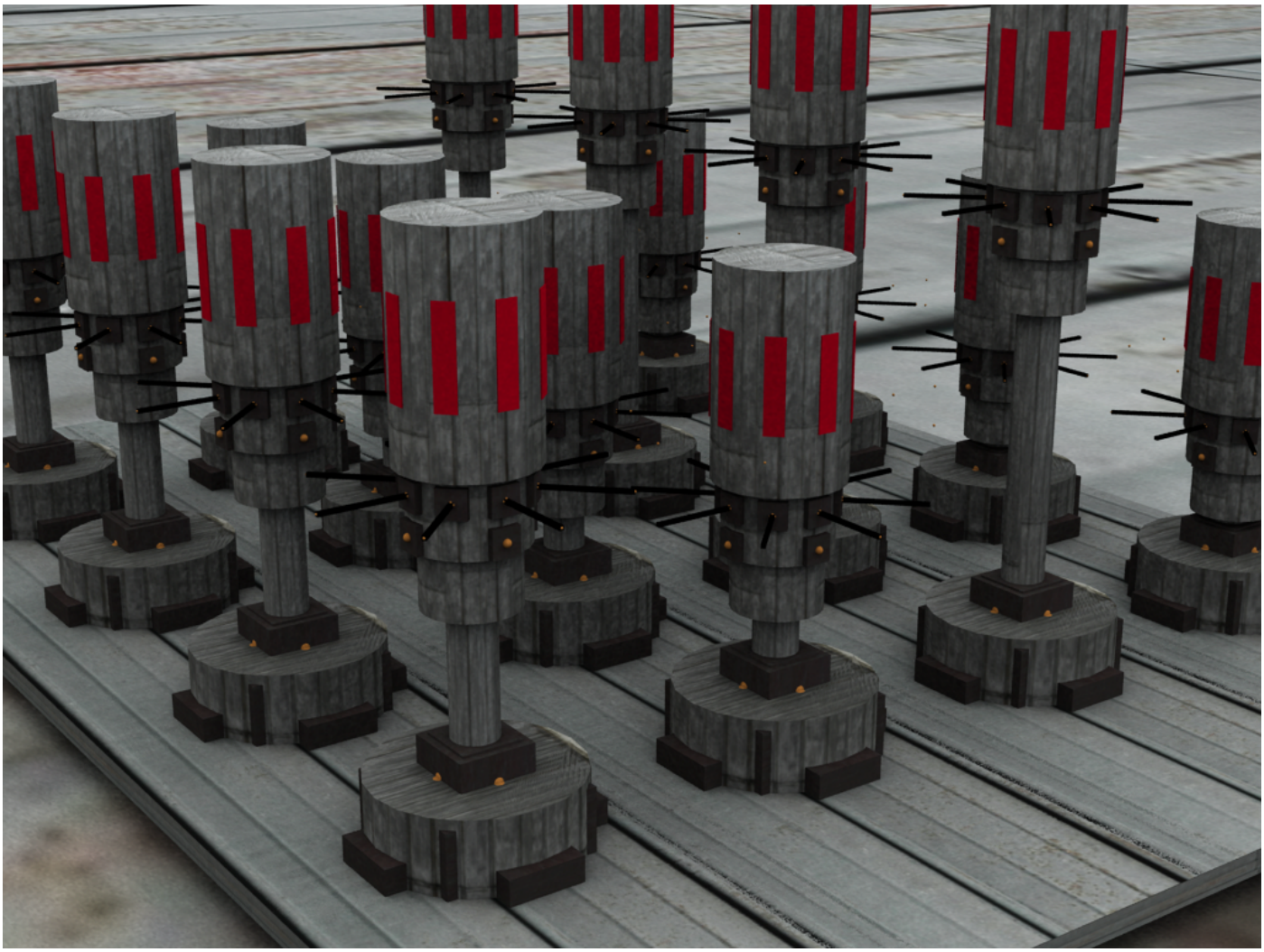


Fig. 5: Variance in the pure values of a *DetailGen* tree can produce controlled differences in the generated patterns

There is plenty of potential to take advantage of other Haskell features, laziness in particular. Because Haskell data structures are not eagerly evaluated, the detail generation tree can branch infinitely. Given a level of detail to generate, we can trim the possibly infinite tree down to that depth before producing the model. As adjustments are made, this detail level could be increased or decreased as desired.

While generating random detail inside of the DSL would be fairly trivial to implement, a more effective method would be to integrate training data into the computation. Artist adjustments could be translated into learned preferences, and the random generation could then be biased towards this preference.

References

[1] S. C. Zhu, Y. N. Wu, and D. Mumford, "Minimax entropy principle and its application to texture modeling," *Neural Computation*, vol. 9, pp. 1627–1660, 1997.

[2] A. Turing, "The chemical basis of morphogenesis," *Philosophical Transactions of the Royal Society B*, vol. 237, pp. 37–72, 1952.

[3] P. Maini and H. Othmer, *Mathematical Models for Biological Pattern Formation*, ser. Ima Volumes in Mathematics and Its Applications. Springer-Verlag GmbH, 2001.

[4] M. C. Cross and P. C. Hohenberg, "Pattern formation outside of equilibrium," *Reviews of Modern Physics*, vol. 65, no. 3, p. 851, 1993.

[5] R. Peng and M.-x. Wang, "On pattern formation in the gray-scott model," *Science in China Series A: Mathematics*, vol. 50, pp. 377–386, 2007.

[6] A. Witkin and M. Kass, "Reaction-diffusion textures," in *Proc. SIGGRAPH*, 1991, pp. 299–308.

[7] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. S. Ebert, J. Lewis, K. Perlin, and M. Zwicker, "A survey of procedural noise functions." *Comput. Graph. Forum*, vol. 29, no. 8, pp. 2579–2600, 2010.

[8] B. Mandelbrot, *The Fractal Geometry of Nature*. San Francisco: Freeman, 1983.

[9] A. A. Efros and T. K. Leung, "Texture synthesis by non-parametric sampling," in *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ser. ICCV '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 1033–.

[10] L.-Y. Wei and M. Levoy, "Fast texture synthesis using tree-structured vector quantization," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 479–488.

[11] P. Barla, S. Breslav, J. Thollot, F. X. Sillion, and L. Markosian, "Stroke pattern analysis and synthesis," in *Proc. of Eurographics 2006 : Computer Graphics Forum*, ser. 663-671, E. Gröller and L. Szirmay-

Kalos, Eds., vol. 25. Vienna, Autriche: ACM, 2006.

[12] J. P. Lewis, "Creation by refinement: A creativity paradigm for gradient descent learning networks," in *International Conference on Neural Networks*, vol. 2. New York: IEEE, 1988, pp. 229–33.

[13] W. Baxter and K. Anjyo, "Latent doodle space," *Computer Graphics Forum*, vol. 25, pp. 477–485, 2006.

[14] H. Weyl, *Symmetry*, ser. Princeton science library. Princeton University Press, 1952.

[15] P. Prusinkiewicz and J. Hanan, *Lindenmayer Systems, Fractals, and Plants*. New York: Springer Verlag, 1989.

[16] Y. Li, F. Bao, E. Zhang, Y. Kobayashi, and P. Wonka, "Geometry synthesis on surfaces using field-guided shape grammars," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 2, pp. 231–243, 2011.

[17] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 301–308. [Online]. Available: http://doi.acm.org/10.1145/383259.383292

[18] K. Sims, "Artificial evolution for computer graphics," in *SIGGRAPH*, 1991, pp. 319–328.

[19] S. P. Jones, R. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler, "Report on the programming language Haskell 98," *SIGPLAN Notices*, 1998.

[20] J. Adámek, S. Milius, N. Bowler, and P. B. Levy, "Coproducts of monads on set," in *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, ser. LICS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 45–54. [Online]. Available: http://dx.doi.org/10.1109/LICS.2012.16

[21] J. McDermott, J. Byrne, J. Swafford, M. O'Neill, and A. Brabazon, "Higher-order functions in aesthetic ec encodings," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, 2010, pp. 1–8.

[22] J. Lewis, R. Rosenholtz, N. Fong, and U. Neumann, "Visualids: automatic distinctive icons for desktop interfaces," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 416–423, 2004.