GENETIC EVOLUTION OF COMPUTER TEXTURES

Lim Teck Sin[1] and Wong Kok Cheong[2]

[1] 59A Science Park Drive, The Fleming, Singapore Science Park, Singapore 118240
tecksin@mail.com
[2] Nanyang Technological University, Singapore
askcwong@ntu.edu.sg

Abstract. A framework for the evolution of computer textures via procedural texture approach is developed. A scanner module is built to extract and save useful details of tokens of the texture programs onto a repository. The tokens are mapped onto linear genetic structures in a postfix manner via recursive descent approach. This allows splicing points to be efficiently computed for execution of genetic operators such as mutation, deletion and allelic combination. Populations of programs are created and images of the programs are generated with a rendering module. The multi-niche crowding and sharing concepts are adopted to facilitate visual selection of textures for the next round of evolution.

1 Introduction

The procedural texture approach entails the development of programs that can produce texture upon evaluation. However, new textures are not easy to develop because it is difficult to formulate functions and program constructs to achieve the desired visual effects. On the other hand, because programs are used to represent textures, evolutionary algorithms can be applied to assist the creation process.

The Texture Evolver framework is developed to evolve programs that are written with RenderMan shading language [1]. The language is chosen because of the similarity to C language and the small set of constructs involved. This facilitates rapid research and development and offers the possibility of applying the results on programs that are written in C, a language which is pervasive in the computing industry. This paper describes the framework and the various modules that are developed to assist user create new textures.

The framework involves a set of modules, namely Scanner, Evolver, Parser, Renderer and Visual Selector. The modules are executed in cycles so as to generate new populations of texture mapping programs (shaders) for genetic operations. The user first selects a shader from a collection. The shader is tokenized by the Scanner and the tokens generated are stored as records in the Token Repository. The next module, Evolver, maps the data of each shader onto linear genetic structures and genetic operators are applied to generate new individuals. The Parser is then used to generate shaders from the individuals before the BMRT renderer is executed to produce images. The images are organised into a two dimensional grid for users to inspect. The user may either choose a shader or crossover a pair of shaders to generate new shaders to evolve the next population. User inputs are required for the selection because it is often difficult to describe the target textures. The framework memorises and stores the user inputs so that knowledge gained can be applied across all the generations of shaders.

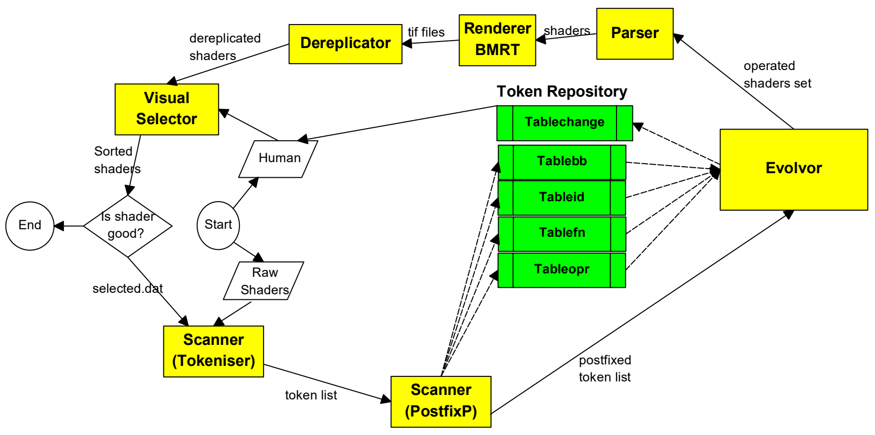

Fig. 1. The evolutionary framework developed for creation of new textures

2 Texture Scanner

The Scanner takes a shader as input and outputs a set of tokens. Each token is an indivisible chunk of codes within the shader. The types of tokens include constants, variables, data types, mathematical operators and functions [2].

The scanner processes each character of a shader just once. It reads a character from the input stream and if it is determined to be the start of a token, the scanner will repeatedly read the next character from the input stream until a terminating character is encountered (e.g. space, commas). The characters read are then returned as a token. The scanner skips unwanted characters (e.g. comments, white space) until the next token is located.

The scanner is able to perform the above mentioned tasks because of the implementation of Finite State Machine (FSM) [3]. Once a character is read, FSM determines if it satisfies any of the edges that are linked to the current state and performs state transition to the next state accordingly. The next section describes how the state transition is performed.

2.1 Recursive Descent Approach (RDA)

The tokens obtained can be divided into terminal and non-terminal tokens. The latter are defined when sequences of tokens are found to match the syntactic rule sets that describe how various program constructs are recursively built on top of the terminal tokens. Each rule set includes a non-terminal token which is used to match the sequences of tokens.

‘Reduction procedures' that can analyse the non-terminal tokens are written with the syntactic rule sets. Such procedures are applied in a recursive manner because non-terminal tokens can contain other non-terminal tokens which in turn need further analysis. It is with the procedures that the scanner ensures that closure property of functions is observed. The tokens are interpreted at a higher syntactic level so that codes can be spliced or exchanged without syntax errors. At the same time, the genetic operators can be made generic because specific syntactic details can be coded separately into the procedures. This makes the system easier to maintain.

2.2 Token Repository

The generated tokens and their syntactic parameters harvested via RDA are saved within the Token Repository. The parameters include

- Type of token which indicates whether a token is an identifier, number, etc.
- Flag to indicate if an identifier is the first occurrence in the shader. This is useful for genetic operations which splice chunks of codes and can thus affect scoping of variables
- Parentheses level of token. This determines the splicing points of the mathematical expression via the Splicing Algorithm (as described in a subsequent section)
- TokenId which acts as a key for genetic operators to search the repository

A set of tables is developed to store the information. One of the tables, ‘Building Block Table' (TableBB) is used to append tokens of evolved shaders. This allows repeated and distributed access of token information by the various modules at different phases of analysis. Similar tables such as TableParen, TableOpr and TableFn are also developed to store information about the parentheses, operators and functions processed.

Another table, ‘Change Table’ saves the operations that are made on tokens. This table acts as a historical log upon which backtracking and Tabu like search can be performed to increase the efficiency of the genetic operators [4].

The careful development of the repository is critical in the creative process, as researchers in music creation have found [5]. Data repositories like TableBB facilitates 'explicit knowledge to explicit knowledge' approach [6]. Information across generations are now stored in a common indexed memory (e.g. on commercial databases) for ease of sharing. Each record of the tables is effectively a meme that allows information to be transferred across populations of shaders [7]. Strategies can be devised to propagate or curb the memes so that new works can be developed effectively. Such generational composition has produced new compositional styles in the domain of music [5].

Before the tokens can be stored in the repository, they have to be transformed such that subsequent genetic operations can be performed efficiently. This is discussed in the next section.

2.3 Postfix Parser

Although the infix notation adopted by the shader language provides a convenient mean for human to visually comprehend the mathematical expressions, it creates problems when the tokens have to be stored on linear genetic structures. The operands that are arranged on both sides of mathematical operators have to be searched in both directions so as to determine the correct ‘limits’ for the expressions. Additional processing resources are thus incurred by genetic operators that have to determine the splicing points of expressions so as to move, append or delete tokens in a syntactically correct manner. There is a need to develop an approach to organise the mathematical tokens such that they can be manipulated easily via the linear genetic structures.

A postfix representation is implemented. The Reverse Polish Notation (RPN) is applied to specify mathematical expressions without the need for parentheses via postfix notation. The RPN method is adapted to feed expression tokens onto a stack. As a result, the operands are organised to be on just one side of the mathematical operators. Splicing of tokens become easier because the limits of a group of tokens can be quickly located by searching along just one direction. It becomes easy to parallelise the parsing process as well. Expressions of a shader can be processed quickly by generating a stack for each expression concurrently.

Both mathematical (in postfix format) and non-mathematical tokens are stored as records in the Token Repository.

3 Evolver

The previous sections describe how useful information derived from the shaders can be preprocessed and stored in a form that genetic operators can work on. This section focuses on the representation of the genetic structures and how the operators assist in the creation of new textures.

3.1 Representation of Genetic Structures

The shaders are represented as linear sequences of tokens (genes). A link list structure is implemented to accommodate the variation in length of shaders and to facilitate easy addition or deletion of tokens that the various genetic operators may perform. At the same time, the structure is designed to allow homologus recombination so that ‘typed genetic operators’ can splice at only genes that have the same types so as to generate syntactically correct solutions [7].

There are a number of reasons for implementing linear structures. Firstly, much evolutionary work has been performed with these structures since 1960s and the results obtained (e.g. niching, sharing) can be readily applied [8]. Also, by keeping the linear structures generic and separate from specific domain implementation, it is feasible to reuse the same genetic system to solve similar problems of other domains, such as music evolution. The complexity involved in implementation of specialised constructs for C, LISP or musical languages can be handled via suitable data structures (e.g. Token Repository) and operators that are built modularly.

Unlike most genetic programming approaches, tree-like structures are not used although they have been widely deployed for LISP programs evolution. Tree structures can represent programs intuitively and at the same time, allow easy manipulation [9]. However, the tree approach is resource intensive [10]. Tokens have to be read and stored in the memory, and parsing has to be performed for the type information that are stored in the nodes. This results in performance overhead and additional coding complexity. In addition, it is not trivial to modify the tree structures to cater for the needs of imperative languages like C, C++ or RenderMan language, which have constructs (e.g. FOR loop) that differ from those of LISP programs.

The next section discusses about the Splicing Algorithm that is used by the operators to determine the limits of mathematical expressions so that genetic operators can be readily work on the genetic structures.

3.2 Splicing Algorithm

The existing RPN methods developed by researchers for genetic operation purposes are insufficient for more complex mathematical expressions of the Renderman language which are often embedded with functions, datatypes and other types of tokens [2]. An algorithm with O(n) time complexity has been devised to solve this.

1. Let the token position of a mathematical operator, MT, be posMT and the parentheses level of MT be lvlMT. The parentheses level of tokens are computed via the Reduction procedures and stored in the Token Repository for reference. A pair of Right Parentheses (RP) and Left Parentheses (LP) has the same level if they enclose the same expression.
2. There are four types of limits to compute for the tokens that are associated with a MT, namely Start of Right Token range (SRT), End of Right Token range (ERT), Start of Left Token range (SLT) and End of Left Token range (ELT). The initial values for the limits are set as SRT=MT+1 and ELT=MT-1.
3. It is easier to compute the limits if there are parentheses available as placeholder to mark the start or end of expression. The algorithm just needs to move left of the token list to find the left parentheses (LP) that is of the same 'level' as RP. The algorithm first search for MT in the postfixed list of tokens that are stored in TableBB. It then checks if there is a Right Parentheses (RP) token immediately after MT.
4. If a RP is present after MT, ERT is set as (RP - 1). The algorithm next proceeds ‘leftwards’ of the expression to hunt for the corresponding Left Parentheses (LP) by moving up the records of TableBB. This LP is to have parentheses level that is equivalent to lvlMT. The search is to conclude if the difference between number of RPs and LPs is 1. The search will also terminate if an equal sign or a comma (of same or lower level) is encountered. Once the search ends, SLT is computed as

(LP + 1). The splicing points for the expression will be one character after LP and one character before RP.

5. If there is no RP after MT, the algorithm becomes more involved. It has to move towards the right to search for a comma, RP, datatype or reserved variable to determine the ERT. SLT is obtained when
   - the level of an identifier, variable or number (IVN) is the same as that of MT, or
   - the number of IVN is equal to (number of MT + 1) for those IVN with level equal to lvlMT
6. Assumes that LP is found and/or the number of RP is equal to number of LP. The algorithm then checks if there is a function name before LP where (level of function = level of LP - 1). If so, SLT is that of function name, otherwise SLT is that of LP. The left splicing point will be (position of SLT - length of SLT + 1).

3.3 Genetic Toolsets

Toolsets of genetic operators have been designed to work on individual or set of tokens for one or more shaders. Many of the operators use the Splicing Algorithm to manipulate tokens. Some of the operators that are originally intended to generate a new individual have also been enhanced such that they are able to generate populations of shaders. All the changes made by the genetic operators are stored in the Change Table of the Token Repository.

3.3.1 Mutation Toolset

This toolset involves a set of operators that perform local changes to a token or a set of tokens. These changes include modification of the parametric ranges, changing of mathematical operators and functions. Given a token, the toolset first searches through the TableBB for more syntactical information about the token, e.g. type of the token. Different mutation operators are then called to handle the various types of syntactic variation. The operators include

- MutateOperator that substitutes a target mathematical operator with another operator that has the same number of operands via a list of operators stored in a table
- MutateFunction that substitutes a function with another function that has the same number of arguments and return type via a list of functions stored in a table
- MutateNumber that takes a number and multiplies it with a random factor

3.3.2 Deletion Toolset

Deletion toolset involves operators that remove a token or a set of tokens from a shader file. This toolset makes the program shorter and may ease any bloating of codes. Codes have to be carefully removed (using the Splicing Algorithm) so that the resultant program is still valid. Some of the syntactic considerations include

 Tokens which are involved in the definition of variables are not deleted, or it may result in syntactic errors if there are multiple occurrence of the same variables
 Set of tokens that is associated with an IF statement cannot be deleted if there is an 'ELSE' after the tokens
 Any dangling 'ELSE', '=' or ',' tokens that are presented before a deleted set of tokens has to be removed
 The whole line of tokens has to be deleted if the equal sign '=' is to be deleted, i.e. if the equal sign is not part of IF or FOR statements

One of the deletion operators has also been developed to delete a whole branch of dual argument mathematical operator.

3.3.3 Combination Toolset

This toolset includes an 'Alleles Combination' operator that has been developed as a form of crossover operator. Given a pair of allelic shaders, the operator searches the TableBB for information about the alleles and generates a new shader that contains both alleles (instead of exchanging genes in usual crossover process). This approach allows incremental combination of features that a user wishes to have for the ideal texture.

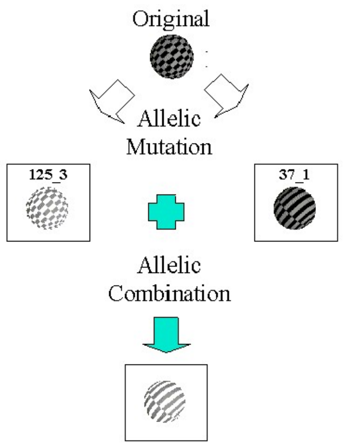

Fig. 2. A checkered shader is operated by the Mutation Toolset to generate a population of allelic shaders for selection. Two of the children, *125_3* and *37_1* produce a lightly shaded texture and a horizontally striped texture respectively. A combination of these shaders results in a texture is lightly shaded and horizontally striped.

3.4 Population Generating Toolset

Some of the mentioned toolsets have been further developed to generate population of new genetic structures from a given structure. This is made feasible by combing through the relevant tokens in the token repository and attempting to evolve each of them into a few variants (alleles). This results in the generation of individuals that differ from each other by an allele. The population size of the new generation is not fixed and is dependent on the size of the shader and the token types involved.

4 Visual Selector

This section discusses how the operated genetic structures are parsed into shaders, and how the images are generated from the shaders and selected by the user for the next round of shader evolution.

4.1 Parser module

The outputs of the genetic operators are genetic individuals that are linear linked list of tokens. A parser is required to 'translate' these lists into shaders that the Renderer module can decipher so as to generate textural images for user to view. The parsing process can be achieved via two approaches.

The first approach regenerates the required shader from the token list that is stored in the Token Repository. A knowledge repository of the syntax has to be implemented so that syntactically correct shaders can be produced. However, this is not trivial as the full syntactic description of the shader language (and other languages) can be involved and tedious to implement.

The second approach modifies the original shader with the relevant changes that have been logged in the Change Table by the genetic operators during the process of allelic shader generation. This approach requires less processing. The resultant files can be generated quickly because only the relevant portions of the shaders have to be changed. Efficiency is important because user performs interactive visual selection and new shaders have to be generated rapidly.

A software package, BMRT is used to generate TIF output of the shaders parsed [11]. These image files are then organised on a two dimensional grid for user to select.

4.2 Selection of shaders

The population of texture images is presented to the user for selection via an adapted form of Multi-Niche Crowding (MNC) so to preserve diversity [12].

Shaders that produce similar textures are clustered (either manually or via a commercial image dereplicator system) and only a representative shader for each cluster is presented for visual selection. This is to facilitate competition among individuals that belong to the same niches. It turns out that this approach speeds up the visual selection process as well because user now has less images to inspect.

The user may select a shader based on the pattern, color, shade or other dimensions (niches). The reasons for the selection are elicited with a user interface and saved into the Token Repository. This knowledge capturing effort allows the texture search to proceed systematically along each niche across all the cycles. At the same time, the user can incrementally build up his specifications for each dimension in a logically partitioned and constrained search space. While the user can choose to change the selection dimension from population to population, the alleles that are preferred for a particular dimension will be tagged accordingly and can be readily mined from the repository. The derived fitness can be deployed to predict and guide the search in an elitist manner so that results can be obtained quickly.

To further accelerate the selection process, the sharing approach is adopted to sort the images before presenting to the user [13]. An image that is generated by many allelic shaders is assigned low fitness as it is a more ‘common’ form. Novelty is an important element of creation and as such, ‘common’ images are ranked lower than the rare ones. Besides using sharing to pick out the more novel shaders, this form of ranking is another attempt to preserve the diversity of alleles found [14].

5 Discussion

The creation of new textures is subjective and involves personal preferences. It is thus difficult to assign a fitness value to each texture and user has to be involved interactively in the selection process. An evolutionary approach is hereby adopted to not only generate new textures automatically, but also traps the implicit knowledge of user so that relevant textures can be built up quickly. The shaders are fragmented by the Scanner into tokens that can be manipulated by both users and computing system across generations of evolution. The Recursive Descent Approach is applied to harvest useful syntactic parameters of the tokens and the mathematical expressions are represented in a postfix format before the tokens are saved into the Token Repository.

The sequences of tokens obtained are represented as linear linked list of chromosomes and toolsets of genetic operators are applied. These operators use the Splicing Algorithm to manipulate of chunks of tokens. A number of the genetic operators are also modified to generate populations of new textures that are allelic. A Parser module and BMRT renderer are then used to regenerate the shaders and images of textures respectively. The Multi-Niche Crowding and Sharing concepts are adopted to facilitate the selection of shaders by user.

It is hoped that this evolutionary approach will accelerate the texture generation process.

Future investigation may involve

- Crossover operators that exchange chunks of tokens so as to produce shaders that have features of both parental shaders
- Contrasting of the genetic structures of similar shaders may uncover differences in the genotypes (tokens). Such tokens maybe an indication of code redundancy
- Current ‘Allelic Combinator’ can only integrate two shaders. But it can be readily scaled up to integrate alleles from more programs. Instead of letting user specify at most two dimensions at a same time, it is possible to let user specify all the dimensions at a go. This will help to reduce the number of evolutionary cycles

 Crossover operator can be developed to exchange token sets between a pair of shaders. This entails the definition of new variables that are introduced into a shader, development of mechanism that can track the scopes of variables, etc.
 The various modules described can be automated via application of an integration system like KOOP so as to increase throughput and productivity of user [15].

References

1. Pixar.: The RenderMan Interface, Version 3.1. San Rafael, California (1989)
2. Upstill, S.: The RenderMan companion: a programmer's guide to realistic computer graphics, Addison-Wesley (1990)
3. Conway, J.H.: Regular algebra and finite machines, Chapman & Hall (1971)
4. Glover F.: Future paths for Integer Programming and Links to Artificial Intelligence, Computers and Operations Research, 5 (1986) 533-549
5. Cope, D.: Experiments in Musical Intelligence, A-R Editions, Inc, Madison, Wisconsin (1996)
6. Nonaka, Ikujiro: Managing innovation as an organisation knowledge creation process. Technology Management and Corporate Strategies: A Tricontinental Perspective, J. Allouche and G. Pogorel (Editors), Elsevier Science B. V(1995)
7. Banzhaf, W., P. Nordin, R. E. Keller and R. D. Francone.: Genetic Programming - An Introduction: on the automatic evolution of computer programs and its applications. Morgan Kaufmann Publishers Inc (1998)
8. Lim Teck Sin.: A GA Approach for Drug Design, MSc Thesis, Oxford University (1995)
9. Koza, J.R.: Genetic Programming. The MIT Press (1992)
10. Keith, M. J. and Martin C. M.: Genetic Programming in C++: Implementation Issues. Chapter 13 of Advances in Genetic Programming, Edited by Kenneth E. Kinnear, Jr., MIT Press (1994)
11. Gritz, Larry and James K. Hahn.: BMRT: A Global Illumination Implementation of the RenderMan Standard. Journal of Graphics Tools, Vol. 1, No. 3 (1996) 29-47
12. Vemuri, V. R. and W. Cedeno: Multi-Niche Crowding for Multi-Modal Search, Practical Handbook of Genetic Algorithm New Frontiers Vol II, Lance Chambers, CRC Press (1995)
13. Deb, K. and D. E. Goldberg: An Investigation of Niche and Species Formation in Genetic Function Optimisation, proc. Third Intl. Conference on Genetic Algorithms, J. D. Schaffer (ed.) Morgan Kaufmann Publishers, San Mateo, CA (1989) 42-50
14. Maxwell, H. John: An Expert System for Harmonising Analysis of Tonal Music. Balaban, Mira, Kemal Ebcioglu and Otto Laske (eds.). Understanding Music with AI: Perspectives on Music Cognition, AAAI Press/MIT Press (1992)
15. Lim Teck Sin: An Introduction to Knowledge and Object Oriented Programming (KOOP) and Technologies Related to High Throughput Collection and Analysis of Data and Knwoledge. www.koooprime.com (2000)