Procedural texture generation based on Genetic Programming

Vinícius M. D'Assunção
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais
Belo Horizonte, Brazil
viniciusvmda@gmail.com

Flávio R. S. Coutinho
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais
Belo Horizonte, Brazil
fegemo@cefetmg.br

***Abstract*—The texture is one of the elements that give a realistic aspect to an object in a game or animation. Textures can be drawn by designers and also can be defined mathematically as a function. This method is also known as procedural texture generation. In this context, we present a procedural generator based on Genetic Programming that provides a set of operations capable of generating an image with similar characteristics given a sample image but not necessarily with the same features. This approach allowed us to create a tree formed by a set of image manipulation operations. Besides, we created a framework for procedural texture generation since we implemented several image manipulation operators.**

***Index Terms*—procedural texture, Genetic Programming**

I. Introduction

The realistic appearance of objects in a digital image created on the computer is the result of lighting, shading, and also the texture applied on it. The texture consists of modifying the appearance of an object's surface, using an image, function, or other data source [1].

Images repeated along the surface of an object can be used as textures. Besides, textures can be described mathematically by a function so that they are automatically generated, instead of created by coloring their texture elements (texels). This technique is known as procedural texture generation [2].

Procedural textures are an interesting alternative to image textures because they have a more compact representation, occupying a much smaller storage space, and also because it is easier to generate small variations [3]. However, instead of being created by someone with knowledge of art, they are generated as a combination of transformations, which is not typically part of the artists' domain. Therefore, some works have proposed to generate these textures in an automated way from an example image texture. Among the most common approaches are those based on Genetic Programming [4], [5] and those based on Convolutional Neural Networks [6], [7].

The creation of textures from another image can be applied in computer games. For generating different textures for an object, it would only be necessary to change the parameters of the operation so that small changes are made in the output image. With that, it would be possible to create many objects with a different look, but following the same pattern. Considering a platform game with several levels, it would be possible, for example, to create single textures for different levels using only one image as an example, changing only the parameters. It would provide more visual variety to the game.

In this study, we present a procedural texture generator based on Genetic Programming that receives an image as input and produces an image with similar characteristics but not necessarily identical, and the transformation tree that generated the output. Also, we present a framework for the procedural texture generation with a series of image manipulation operations.

II. Theoretical Foundation

Textures give a more realistic look to a surface where instead of using a single color on the entire surface, it is possible to map a 2D image on the 2D or 3D surface. In addition to textures based on an image scanned or manufactured by an artist, there are also procedural textures, which are textures generated from an algorithm or a mathematical model [8].

Procedural textures have some advantages over image textures. Procedural representation does not have a fixed resolution and is more compact. Among its disadvantages, programming can be complex and the result can be an unexpected texture since its results are the consequence of the application of several computational transformations. In addition, generating a procedural texture can be slower than accessing a texture already stored on the disk [8].

There are several techniques for the procedural generation of textures. Among them, we can highlight: Pseudo-random Number Generators; Image filtering; Space Algorithms; Modeling and Simulation of Complex Systems; and Artificial Intelligence [9].

Briefly, the Genetic Programming algorithm maintains a population of programs, and each one is assigned a fitness value that indicates how well it is capable of solving the problem. Then, selection methods choose the programs who are best fit. Then, genetic operators are applied to modify them to converge to a solution, generating the next generation of individuals. The program with the best fitness value among the population is chosen as the solution at the end of the execution [10].

III. Related Works

Gentropy [4] uses Genetic Programming to find a program capable of generating a procedural image similar to a pro-

vided sample image. In this implementation, each individual represents an expression that returns a vector representing a pixel of the image, that is, a vector of three positions that correspond to the intensity of the colors red, green, and blue. The operators used to return a pixel vector or a scalar that can be used as a parameter for another operator. The evolution process is guided by feature tests that evaluate color, shape, and smoothness.

Despite not using images that could be useful for games as input, such as tree barks, ground, and coatings, the generated images were quite close to the input, with aptitudes between 60% and 78%. The average execution time was a negative point due to the hardware limitations at the time (2002). The algorithm took around 52 hours to evolve the set of operations for generating the image.

Our study differs from Gentropy [4] by representing individuals as expressions that return a full image, instead of a pixel. With that, it is possible to use image processing techniques in the image transformation process, applying filters that reduce image noise, for example.

IV. Architecture

Briefly, the procedural texture generation algorithm receives a sample image as input. Then the population is initialized with trees formed from the set of image operators, parameter operators, and terminals. For each tree, the program calculates the fitness value, which indicates how well the generated image relates to the input image. Then, the algorithm uses selection methods for choosing programs with the best fitness values. Afterwards, genetic operators are applied to modify them to converge to a solution, generating the next generation of individuals. At the end of the algorithm execution, it chooses the tree with the best fitness value among the population as a solution.

We used the Python programming language and the DEAP[1] library in our implementation. This library provides resources and implementations of Evolutionary Computing techniques. In addition, we used some functions from the OpenCV[2] library to build the image manipulation operators.

A. Terminals

Terminals are used as parameters of the operators. The set of terminals can contain both constants and variables. We defined a set formed only by constants, containing the first nine elements in the Fibonacci sequence starting from the number 1 and having no repeated elements. In addition, there is the $WHITE$ matrix, which consists of a completely white image. We chose the Fibonacci sequence based on experimentation.

B. Operators

The set of operators contains image operators and parameter operators. Image operators receive images or floating points as arguments, process them on the received image or in a blank image, and return the resulting image. The image operators we used are:

$noise$: corresponds to Perlin noise [11], which generates noise in a coherent structure.

$stripes$: generates stripes on the image.

$checkerboard$: creates a checkerboard texture in black and white over the image.

rgb: adds color to the image.

$sumImg$: gives the weighted sum of two images.

$bilateral$: applies a filter that removes noise from the image while preserving the edges.

$erode$: corresponds to a morphological operator that is able to reduce the boundaries of the objects.

$dilate$: increases the boundaries of objects.

The parameter operators perform operations with the terminals, so that it is possible to generate different parameters for the image operators. We used the following set of operators: sum, sub, $mult$, div, mod, log, sen, cos, avg, min, and max.

C. Genetic Programming Algorithm

The program initializes the population with a fixed set of individuals. Then, for each generation, one of the genetic operators is applied until it generates a fixed set of children. Then the children's fitness is calculated and the selection is made over the population and the children generated until it produces a new set of individuals with a fixed size.

The initial population of programs is generated using the *ramped-half-and-half* method [12] that consists of using the methods $full$ and $grow$ for random population generation. The $grow$ method creates 50% of individuals in the population by choosing the nodes of the tree at random where the size of the tree varies within the defined minimum and maximum range. The $full$ method creates the other 50% of the population by generating random trees of a defined size. The fitness function is a combination of two fitness values, one based on the generated image shape, and the other based on the color.

The fitness value based on the shape is given by the $getOrbMatch(img)$ function, which uses the Canny operator [13] for generating an image that highlights the edges. Then, the ORB [14] function identifies the key points of the generated image. Having the key points of both the sample image and the image being evaluated, the Brute-Force Matcher [15] method compares the distances between the key points of the two images. Finally, the sum of these distances is divided by the maximum value of the distance (product between the height and width of the images) multiplied by the number of key points. Thus, the higher the fitness value, the closer will the generated image be to the target image in terms of shape. Fig. 1 shows the fitness calculation steps in which the Canny detector and the ORB function were applied.

The function $getColourHistogramMatching(img)$ gives the color-based fitness value. It returns the difference between the histograms of the target image and the generated image.

[1]DEAP: https://deap.readthedocs.io/en/master/.
[2]OpenCV: https://opencv.org/.

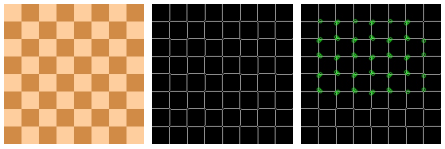

Fig. 1. From left to right: the sample image; the Canny operator output; the key points detected by the ORB function in green.

Such difference is in the range of $[0, 1]$ where the higher the similarity value, the closer will the generated image be from the target image in terms of color.

Equation 1 represents the fitness evaluation function $f(i, g)$, where img_i is the image generated by the individual i in the generation g.

$$f(i,g) = 0.9 \cdot getOrbMatch(img_i) \\ +0.1 \cdot getColourHistogramMatching(img_i) \quad (1)$$

We weighted the image shape component to privilege it over the color component, so it considers images similar even if the color differs or the output colors are not in the same proportion. For example, considering the first image of Fig. 1 as an input, if a checkerboard texture was generated with the block colors alternated, both would be similar when comparing them in terms of shape, whereas if we use a fitness function based on the distance between the colors of the pixels, the two images would not be considered similar since all the blocks would have a different color.

The genetic operators are reproduction, crossing, and mutation. The algorithm applies them to individuals after the selection. The reproduction method selects an individual and copies it to the new population. The crossing method selects a node randomly in the tree of two individuals and switches their subtree, that is, the node and all its children. The chosen mutation method is the uniform, which randomly selects a node in the individual's tree and exchanges the subtree for a randomly generated subtree of minimum and maximum defined size. The end condition of the algorithm is the maximum number of generations.

The algorithm output is an expression composed of the terminals, described in Section IV-A and the operators described in Section IV-B. This expression returns a three-dimensional array in the format $width \times height \times RGB\_vector$.

V. Results

The machine used to run the tests has an Intel Core i5-5200U CPU @ 2.20GHz CPU with 2 cores and 8 GB RAM. The operating system used was 64-bit Linux Mint 19.2.

We performed the tests using 6 sample images and the program ran twice for each sample. The initial population was 150 individuals and the algorithm ran for 15 generations.

A. Qualitative

Fig. 2 shows all the input images and their respective outputs for each test, and Fig. 3 shows one of the trees generated in the tests. Visually analyzing the outputs, we can observe that the textures of checkerboard, granite, and grass were similar to their respective input images, but with some differences. The generated checkerboard textures even have more artistic elements than the original image. On the other hand, the texture generated from bricks, sky, and water were very different from the input images, as the algorithm does not have any operator capable of generating rectangles in an alternating pattern like those present in the brick texture. In addition, the detected edges of the brick texture may cause many false positives to be generated since the generated image has borders in close regions and they do not respect the shape of a rectangle.

The sky and water textures are very light and do not have well-defined edges. This hindered the shape evaluation since it is based on this property. The water texture has very few visible edges after applying the Canny filter. On the other hand, the applied filter maximized the details of the sky texture. This created many key points, which ended up generating a lot of false positives.

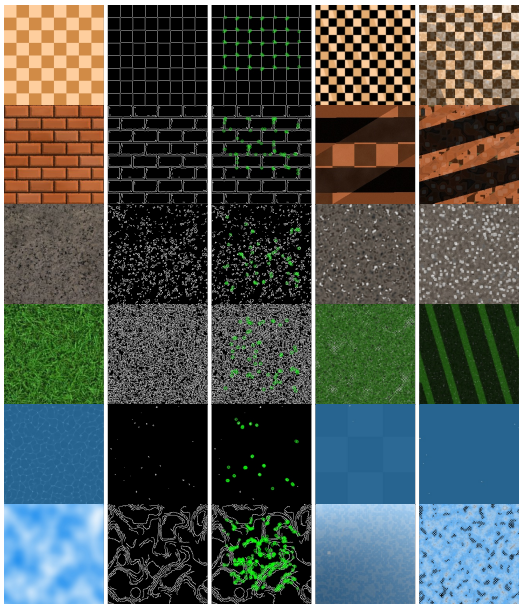

Fig. 2. The first column refers to the input image, the second shows the Canny operator output, the third column highlights in green the key points detected by the ORB function, and last two columns show the output image for the two algorithm executions. The rows represent the following test cases: checkerboard; bricks; granite; grass; water; and sky.

B. Quantitative

We present important evaluation metric for the algorithm in Fig. 4: the likelihood of evolution leap [16]. This metric informs the progress between successive generations, in which the number of generations is averaged in which the best individual has greater fitness than the best individuals from previous generations. The test scenarios are in a range between approximately 53% and 67%, which indicates that the population is constantly evolving. The test scenario that has

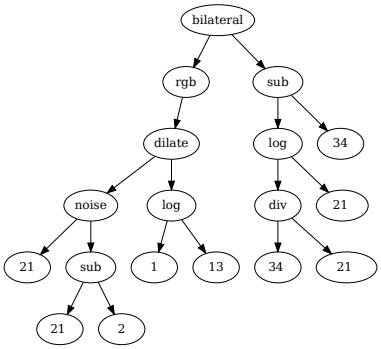

Fig. 3. Generated tree for the first granite output texture in Fig. 2. Generated expression: bilateral(rgb(dilate(noise(21, sub(21, 2)), log(1, 13))), sub(log(div(34, 21), 21), 34)).

the worst progress rate is the water texture, precisely because it has a few key points after applying the Canny function, as shown in Fig. 2. Thus, there are not many operations to be done that can make this image look like the original image.

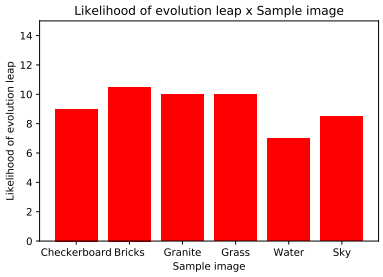

Fig. 4. Likelihood of evolution leap per sample image graph.

VI. Conclusion

The objective of the present work was to create a program that receives an input image and generates a set of operations capable of generating an image with similar characteristics, but not necessarily equal to it.

Given the challenge of evaluating whether an image is similar to another, part of this analysis had to be done visually and some generated images were very similar to the input images, adding even more artistic elements and therefore, can be used in the process of creating textures for games. The artist could use a texture from his image library to generate similar textures, and he could also edit the parameters and operators of the generated expression to attend his demand.

The present work also created a framework for procedural texture generation, since all the operators created can be reused to implement other approaches. In addition, we used another strategy for procedural image generation based on genetic programming in which the expression output resulting from the algorithm is capable of generating an image, instead of generating only a vector representing a pixel (as in [4]). This enabled the use of image processing operators such as the bilateral filter and the erosion and dilation operators, which made it possible to generate more realistic images.

However, the algorithm has limitations about the complexity of the input image, not being able to generate forms that are not covered by its set of operators. The algorithm is also unable to generate textures from images that do not have well-defined edges or that have low contrast. In addition, it has a high computational cost for manipulating matrices in most of its operators and for having to compile the expressions whenever an individual is evaluated.

As future work, we suggest implementing parallel programming to decrease the algorithm execution time, the use of some evolutionary model or algorithm to find out what would be the best input parameters, the removal of the operator that adds color to the texture, working with grayscale and adding the color only after generating the texture, and the use of another fitness function when the algorithm detects that an image does not have well-defined edges since the method used based on the shape cannot interpret that kind of image. A comparative study could also be done to find out which fitness function has the best results.

References

[1] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. Crc Press, 2019.

[2] D. Hearn, M. P. Baker, and W. R. Carithers, *Computer graphics with OpenGL*. Upper Saddle River, NJ: Pearson Prentice Hall,, 2014.

[3] D. Ginsburg, B. Purnomo, D. Shreiner, and A. Munshi, *OpenGL ES 3.0 programming guide*. Addison-Wesley Professional, 2014.

[4] A. L. Wiens and B. J. Ross, "Gentropy: evolving 2d textures," *Computers & Graphics*, vol. 26, no. 1, pp. 75–88, 2002.

[5] A. Hewgill and B. J. Ross, "Procedural 3d texture synthesis using genetic programming," *Computers & Graphics*, vol. 28, no. 4, pp. 569–584, 2004.

[6] L. Gatys, A. S. Ecker, and M. Bethge, "Texture synthesis using convolutional neural networks," in *Advances in neural information processing systems*, 2015, pp. 262–270.

[7] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. S. Lempitsky, "Texture networks: Feed-forward synthesis of textures and stylized images." in *ICML*, vol. 1, no. 2, 2016, p. 4.

[8] D. S. Ebert and F. K. Musgrave, *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.

[9] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, no. 1, p. 1, 2013.

[10] A. Pozo, A. d. F. Cavalheiro, C. Ishida, E. Spinosa, and E. M. Rodrigues, "Computação evolutiva," *Universidade Federal do Paraná, 61p.(Grupo de Pesquisas em Computação Evolutiva, Departamento de Informática-Universidade Federal do Paraná)*, 2005.

[11] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[12] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[13] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[14] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*. Ieee, 2011, pp. 2564–2571.

[15] A. Jakubović and J. Velagić, "Image feature matching and object detection using brute-force matchers," in *2018 International Symposium ELMAR*. IEEE, 2018, pp. 83–86.

[16] K. Sugihara, "Measures for performance evaluation of genetic algorithms," in *Proc. 3rd. joint Conference on Information Sciences*. Citeseer, 1997, pp. 172–175.