Evolving Simple Art-based Games

Simon Colton and Cameron Browne

Computational Creativity Group
Department of Computing, Imperial College London
sgc,camb@doc.ic.ac.uk, http://www.doc.ic.ac.uk/ccg

Abstract. Evolutionary art has a long and distinguished history, and genetic programming is one of only a handful of AI techniques which is used in graphic design and the visual arts. A recent trend in so-called 'new media' art is to design online pieces which are dynamic and have an element of interaction and sometimes simple game-playing aspects. This defines the challenge addressed here: to automatically evolve dynamic, interactive art pieces with game elements. We do this by extending the Avera user-driven evolutionary art system to produce programs which generate spirograph-style images by repeatedly placing, scaling, rotating and colouring geometric objects such as squares and circles. Such images are produced in an inherently causal way which provides the dynamic element to the pieces. We further extend the system to produce programs which react to mouse clicks, and to evolve sequential patterns of clicks for the user to uncover. We wrap the programs in a simple front end which provides the user with feedback on how close they are to uncovering the pattern, adding a lightweight game-playing element to the pieces. The evolved interactive artworks are a preliminary step in the creation of more sophisticated multimedia pieces.

1 Introduction

Broadly speaking, evolutionary art is the process by which programs are evolved which are able to produce artworks. In most cases, the user acts as the fitness function, by selecting and rejecting genotypes (programs) based on their phenotypes (the images they produce). Example projects include Mutator from Latham et. al., where sequences of 3D shape morphing operations are evolved [7], NeVaR from Machado et. al., where programs are evolved to choose each pixel's colour in an image based on its location [4], and the work of Jon McCormack, where L-systems for 3D-modelling of organic forms are evolved [5]. In none of these projects are the artworks generated interactive in any way. In contrast, a major trend in so-called 'new media' approaches to modern visual arts is to build online, dynamic and interactive artworks, often with a puzzle or game element to them. The question we therefore address here is whether it is possible to automatically evolve such artworks, i.e., of a dynamic and interactive nature and with a lightweight game element to them.

To answer this question positively, we extend the Avera evolutionary art system (which is described below) by both extending the sophistication of the programs it evolves and by building a simple front-end to use the programs in an interactive way. As described in section 2, we simulate a well known form of making artworks, namely spirographs, by evolving programs which control the position, rotation, size, colour and transparency of a series of geometric shapes (namely squares, circles and triangles). As described in section 3, we extend this to enable the generation of programs that produce spirographs drawn in a way dependent on where a user clicks in the canvas that the spirograph is being drawn on. In section 4, we describe how this enabled us to add a puzzle element to the pieces, by evolving programs which embed a causal pattern of clicks to be discovered by the user. If the user correctly approximates the timing and position of the required clicks, they are rewarded by generating an aesthetically pleasing spirograph, and we provide a target image to guide his/her clicks. We conclude by describing some of the ways in which we plan to extend this work to produce more sophisticated multi-media art/game pieces.

1.1 Background

Our interactive system is an extension of an earlier framework designed for creative artefact generation, named Avera [3]. This framework proved general enough to accept problems from a range of different domains, yet concise enough to allow for quick prototyping and easy modification. It is based on a variable-sized tree representation to encode solutions, which admits an evolutionary search through crossover and mutation. The framework uses a combination of node-type constraints upon the generated parse trees, and logical constraints for removing unwanted node patterns. The former is based upon the constraints of Montana's strongly typed genetic programming [6], and allows the type systems of programming languages to be respected. Meanwhile, the latter allows for constraints over node dependencies to be expressed. Finally, a method is provided for translating the tree structure used internally to represent solutions into text output, which is analogous to the genotype-phenotype mapping in Genetic Programming. This can be used to convert solutions into entire programs, program classes (in particular, compilable Java classes), scripts or data that can be accepted by other programs, and the behaviour of these programs can be used to evaluate the success of the solution. An example tree and the resulting code are given in the next section.

By attaching a front-end which enables the user to act as a fitness function by choosing between phenotype images, in [3], the authors applied Avera to three evolutionary art projects involving: 1) pixel-based, 2) image filtering and 3) particle-based styles. We now describe an additional mode of operation for interactive use.

2 Evolutionary Setup for Spirographs

The word 'spirograph' is commonly used to describe both an art-producing contraption – most commonly used as a children's toy – and the artworks that

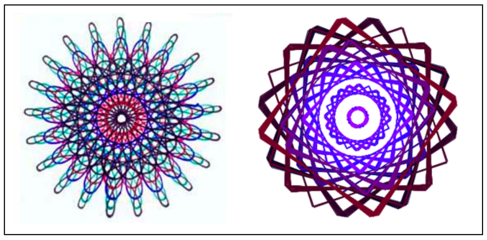


Fig. 1. A hand-drawn spirograph curve from `en.wikipedia.org/wiki/Spirograph`, and an approximate spirograph produced using the repeated placement of squares.

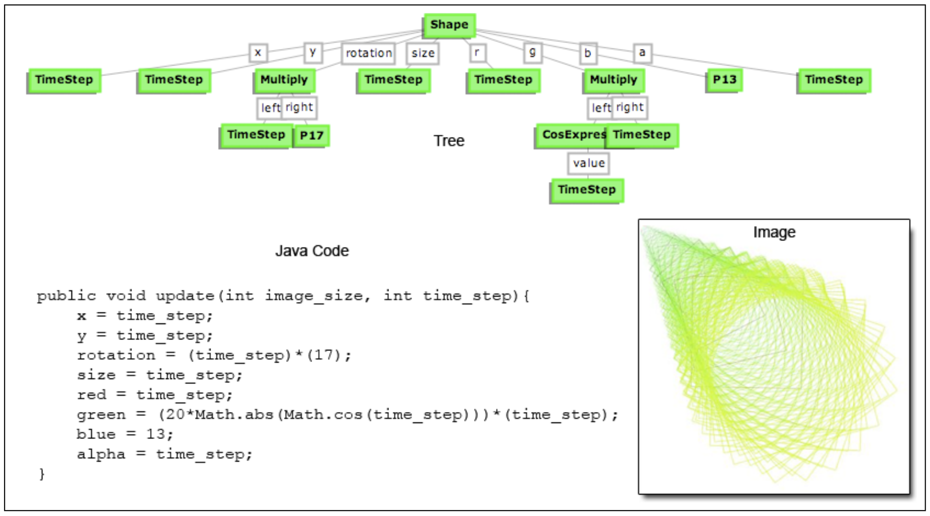
it produces.[1] Introduced in the 1960s, the contraption is able to produce hypotrochoid and epitrochoid curves. While these curves are continuous, the look of spirographs can be approximated by repeatedly drawing geometric objects, including squares, circles and triangles in close proximity to each other. For instance, Figure 1 portrays a spirograph produced by hand using the contraption, and an image consisting of a series of coloured squares. To evolve programs able to generate such approximate spirographs, we used Avera to construct a Java class with the following two methods:

- `initialise:` In this method, the type of shape (square, circle, triangle) is specified, along with an upper limit for a time-step counter. These values do not change as the image is drawn.
- `update:` In this method, the position of the centre, and the scale, rotation, transparency and RGB colour values of a shape are calculated. These values may be constant, or may be a function of the time-step counter and/or the image square size (which we kept at 500 by 500 pixels in all the examples reported here, but which can be varied).

Starting at zero, as the time-step counter increases up to the maximum specified in `initialise`, the shapes change according to the functions in `update`, and if the shapes are drawn onto a canvas, the resulting image approximates a spirograph image. We used the integers between 1 and 20 as terminals, along with a pointer to the time-step counter and the size of the canvas. We also used the binary functions $cos(x)$ and $sin(x)$ as well as addition, multiplication, division and subtraction, and the unary functions: $random(x)$ (able to generate a random number between 1 and x inclusive) and negation. Note that the absolute value of the output of all unary and binary functions was taken, as all the values which define a shape (with the exception of rotation) need to be positive integers. In addition, before drawing the shapes, any RGB values that exceeded 255 were limited to 255. Moreover, for the time-step constant in `initialise`, we take the constant that nt is equated to, and use $200 + (100 * nt)$ time steps.

In Figure 2, we present an example of an evolved spirograph-generating program. In this example, the `initialise` program (not shown) specifies a square

[1] For an online applet which creates spirographs, see the following web site: `http://wordsmith.org/anu/java/spirograph.html`



```
public void update(int image_size, int time_step){
    x = time_step;
    y = time_step;
    rotation = (time_step)*(17);
    size = time_step;
    red = time_step;
    green = (20*Math.abs(Math.cos(time_step)))*(time_step);
    blue = 13;
    alpha = time_step;
}
```

Fig. 2. An evolved tree, its java interpretation and the resulting image.

as the shape to draw, and 250 time steps. As we see, in the `update` method, the x and y co-ordinates are equated to the time step, and hence the square moves from the top left to the bottom right of the canvas. The size of the square increases in line with the time step, as does the rotation and the transparency of the line drawn. The blue value of the square is fixed, but the red value increases with the time step, and the green value is calculated as: $20 * |cos(time\_step)| * time\_step$.

3 Adding Dynamic and Interactive Features

Drawing shapes onto a canvas can be achieved very quickly, but if the process is slowed down – we used a 40ms delay between the placing of subsequent shapes – then the static spirographs essentially become a dynamic, viewable video. The Avera specification was then extended so that the programs could use the (x, y) co-ordinates of a mouse click in the functions which determine the colour, position, rotation, size and transparency of the shapes. A simple graphical user interface was built to dynamically display the spirograph and to process mouse clicks by passing them to the drawing process, so that the evolved `update` method can use the click point to change the shapes as they are drawn, which adds an interactive element to the pieces. The click point defaults to the middle of the canvas (250, 250) until the user clicks the canvas.

As an example, the `update` program embedded in the Java class producing all four images in Figure 5 is given in figure 3. We see that the x and y values are fixed to the centre of the canvas, the rotation of the shape increases with time, the size is proportional to how far down the canvas the user clicks, and the colour is dependent on whereabouts from right to left the user clicks. In practice, without any clicks, this spirograph rotates a blue square with RGB value (10, 10, 250) and of size 250 around the centre point of the canvas. However, if the

```
public void update(int image_size, int time_step, Point click_point){
       x = image_size/2f;
       y = image_size/2f;
       rotation = time_step;
       size = click_point.y;
       red = 10;
       green = 10;
       blue = click_point.x;
       alpha = 255;
}
```

Fig. 3. Evolved update function for the images portrayed in Figure 5

user clicks on a point on the left of the canvas, the colour becomes blacker, going blue again if the user clicks on the right of the canvas. If the user clicks towards the top of the canvas (noting that y co-ordinates *increase* from top to bottom), the square gets smaller, but reaches a size of 500 as the user clicks towards the bottom of the canvas. Note that – for reasons explained later – in the first, third and fourth spirographs of Figure 5, the user clicked along the centre line from the top to the bottom of the canvas. A constraint dictating that the evolved methods must contain some reference to the mouse click point was added to the Avera specification.

4 Adding Game Aspects

A game need not have explicitly stated goals to be enjoyable, according to the recent concept of *ludic engagement* proposed by Gaver et al [1]. Ludic engagement covers a wide range of exploratory, self-motivated activities with which participants engage not to achieve some particular goal but because the process itself is rewarding. The resulting interactions are led by pleasure rather than utility, and can be valuable in helping to explore new relations and to foster an awareness of perspectives. The kinds of art pieces we evolved are meant to be enjoyable to interact with in their own rights. However, we added a game element to further demonstrate the possibilities for our approach, and to hopefully make the pieces more interesting by encouraging the user to interact with the programs for a purpose.

4.1 Introducing a Game Element

To instil a game element, we added to the programs that Avera evolved a notion of a fixed causal/spatial sequence of mouse click points. When viewing the phenotypes in Avera's browser (i.e., when we act as the fitness function in order to evolve spirograph programs), we display them as they would look if the user were to click exactly the specified locations on the canvas at exactly the specified times. This enabled us to evolve spirograph programs which can produce aesthetically pleasing images if the user is able to reproduce the correct set of on-screen clicks. By hiding the click sequence, but giving the user a target image to try to produce through interaction with the spirograph generator, we therefore gave the programs a lightweight game element.

We found that having to uncover an arbitrary set of points at arbitrary times was too difficult, so we opted for fixed sets of fairly natural patterns which dictate the placement of the points, and a fixed set of Morse-code style time intervals which dictate the intervals (in terms of the time steps of the spirograph generator, not actual seconds) that the user should leave between clicks. To enable this, we added two variables to the `initialise` methods that Avera evolved, and stipulated that the values correspond to position/interval sequences respectively. We extended the GUI described above by enabling it to read the evolved Java classes, extract the pair of values representing the sequences and calculate the set of triples $\langle x, y, t\rangle$ which specify that the screen should be clicked at co-ordinate (x, y) at time-step counter t. Then, as the shapes are drawn onto the canvas, the clicking of points are simulated automatically, triggered by the time-step counter reaching certain values.

The position patterns and time interval sequences we implemented for these preliminary tests were as follows:

- *Position patterns:*
corners (4 points), clockwise circle (8 points), anti-clockwise circle (8 points), top-bottom centre line (6 points), left-right centre line (6 points), plus sign (8 points), cross sign (8 points), left-right diagonal from top to bottom (5 points), left-right diagonal from bottom to top (5 points).
- *Time interval sequences:*
$\{10\}, \{10, 20\}, \{10, 40\}, \{10, 20, 30\}, \{10, 20, 40\}, \{10, 20, 10, 30\}$.

As an example, suppose the shape class evolved by Avera specifies that the position pattern variable is set to 2 (signifying an anti-clockwise circle position pattern), and the time-interval variable is set to 2 also (signifying a dot-dash time interval pattern, using the Morse-code analogy). The time intervals signify the length of time the user needs to wait after they have clicked the screen, before clicking the screen again. In Figure 4, we portray the sequence of clicks the user would need to perform in order to recreate the target image.

4.2 Improving Enjoyability

With 9 position and 6 time-interval sequences, there are 54 possible click sequences available for Avera to embed into the Java classes that it evolves, which is enough for trial and error to become an unfavourable option. Hence the user will need to experiment to begin to understand how their clicks affect the drawing of the shapes, and then decide through experimentation which sequence of clicks produce the desired spirograph. We kept the first iteration of the GUI very simple. In addition to the panel allowing the user to click the changing spirograph, the GUI also contained an image of the completed target spirograph, and the user's first click started the time-step counter. They were then required to keep on clicking until the counter reached its maximum, or start again using a

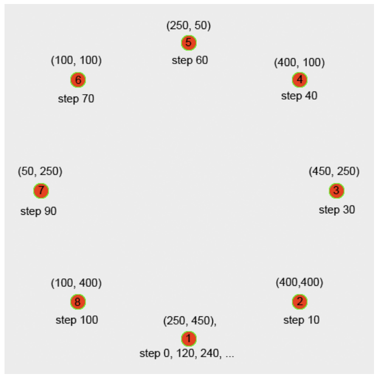


Fig. 4. An example of required click positions in order to recreate a target spirograph.

button we provided. Unfortunately, we found this GUI far too difficult and haptically awkward to recreate the target image. Hence, in a second GUI iteration, we implemented the following improvements:

1. The user is allowed to drag the mouse, and the `update` function driving the spirograph generation is given the mouse drag point just like it is given mouse click points. We found that this encouraged the user to play around with the program much more at the start of a session, when trying to uncover exactly how positions on screen corresponded to shape changes. This was because dragging is essentially equivalent to hundreds of click operations.

2. Once the user has determined what they hope is the correct sequence of clicks, they can restart the program, click the pattern, and end where they started with a right-click operation. This then informs the GUI to repeat the same pattern of clicks until the drawing process ends. This removed the requirement for the user to click until the end of the process, which really limited their enjoyment.

3. We added a button which enabled the user to see a simulation of the correct click sequence, so that they could be given help in recreating the target image.

4. With many examples, where the `update` method used a complicated function involving the click co-ordinates to determine the shape change, it was difficult to comprehend how a click altered the shapes. Hence, we implemented a scoring mechanism which gave the users instant feedback on how close their sequence of clicks has been to the correct sequence. This is calculated as an average over the distance between a user's click and the relevant required click. The required click which is closest to, but causally before, the user's click is taken as the relevant click. At the start of the game, the user can speculatively click around the screen in order to determine where the starting point of the pattern is, and the scoring mechanism gives valuable feedback towards a solution.

5. Even when the user has correctly determined the pattern to click and the correct times at which to click (or they have looked at the solution), it was still impossible to recreate the target image in many cases. This is because some of the spirograph programs were particularly sensitive to slightly wrong clicks

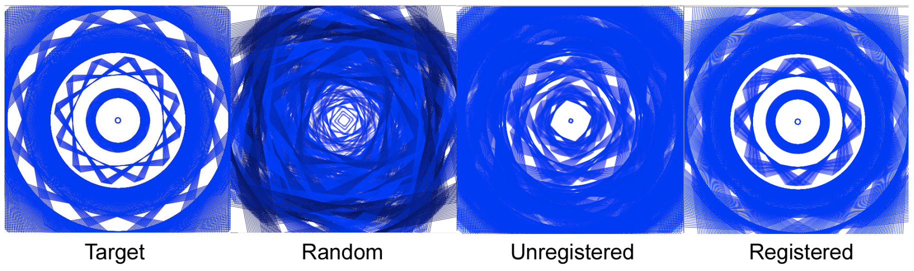


Fig. 5. Target spirograph; spirograph produced via random clicking; spirograph produced by near-correct clicking, but without point registration; spirograph produced by near-correct clicking, with point registration.

(both spatially and causally). This was very dissatisfying for users, as they felt they should be rewarded for their endeavours with a recreated target image. To solve this problem, we perform point *registration*. That is, when the user clicks on a point which is within 20 time steps (either before or after), and within 50 pixels of a correct click, it is mapped to that correct click. Hence, when the user finishes the sequence with a right-click, if they have been nearly right with all of their clicks, the resulting image produced will be almost identical to the target image (with any differences arising from the slightly incorrect early generation of shapes before the repetition was turned on). Figure 5 shows the value of this addition to the GUI. In this example, the correct sequence is the centre line from top to bottom of the canvas, with repeating intervals of 10, then 20 milliseconds. Comparing the image produced by a random set of clicks (image 2 in Figure 5) with the target highlights the fact that some thought is needed to solve the problem. However, in the final two spirographs of Figure 5, the user had uncovered the correct sequence of clicks. In the first case, registration was not enabled, and the resulting spirograph is far from the target image. In the second case, however, registration was enabled, but one of the clicks was not close enough to be mapped to the perfect click (hence it is not a perfect reproduction of the target image, but it does show the value of registration, as the resulting spirograph is visually much closer to the target one than the third image).

6. Finally, we added a slider bar, so that the user can alter the number of milliseconds delay between shape renders, ranging from 0 to 100 milliseconds. This enables them to work slower or faster depending on their abilities and/or the difficulty of the puzzle.

5 Conclusions and Future Work

We have so far performed only limited testing of the interactive and game-playing aspects of the evolved pieces, which prompted the second round of GUI implementations. We plan to perform much more user testing to see if the particular configuration we have is indeed enjoyable, and to see where further improvements can be made: for instance, we hypothesise that mouse drag gestures will provide a more natural input to the pieces than clicks. To do the user testing, we will appeal to internet communities to provide feedback about the pieces, which we will make available as online applets. While the games may be simplistic (albeit quite difficult in some cases), the visual variety of the artworks underlying them is quite marked. To emphasise this, during a single session lasting around 2 hours, we evolved 3000 spirograph programs in 30 generations of 100 individuals. We selected around 150 as our favourites (functionality which Avera enables), and we present 77 of these in the appendix. There is considerable variety in the results; due to the ability to use random numbers, some of the images have a particularly sketchy look, while others include vibrant block colours resembling abstract art, and others still have an unexpected photo-realistic quality.

We hope to have shown that there is much potential for evolutionary search to break through from static visual art into the more dynamic, interactive, puzzle-based world of new media art. We plan to further expand the sophistication of the programs we evolve so that their phenotypes are multimedia art pieces on a par with those written by hundreds of new media artists using packages such as processing [2]. By further adding the ability for Avera to use a fitness function automatically (i.e., bypassing the requirement for a user's inputs, as can be the case with the NeVaR system [4]), we hope to show that computational creativity approaches can be used to generate increasingly complex, valuable and enjoyable artefacts on the border of the art and games worlds.

6 Acknowledgements

This work was supported by EPSRC grant EP/E005713. Thanks to Jeremy Gow for useful input, the anonymous reviewers for helpful suggestions, and to Marc Hull for developing the original Avera system.

References

1. W Gaver, J Bowers, A Boucher, A Law, S Pennington, and B Walker. Electronic furniture for the curious home: Assessing ludic designs in the field. *International Journal of Human-Computer Interaction*, 22(1/2):119–152, 2007.
2. I Greenberg. *Processing: Creative Coding and Computational Art (Foundation)*. Friends of Ed., 2007.
3. M Hull and S Colton. Towards a general framework for program generation in creative domains. In *Proceedings of the 4th International Joint Workshop on Computational Creativity*, 2007.
4. P Machado and A Cardoso. NEvAr – the assessment of an evolutionary art tool. In *Proceedings of the AISB00 Symposium on Creative and Cultural Aspects and Applications of AI and Cognitive Science*, 2000.
5. J McCormack. *Impossible Nature. The Art of Jon McCormack.* Australian Centre for the Moving Image, 2004.
6. D Montana. Strongly typed genetic programming. *Journal of Evolutionary Computation*, 3:199–230, 1995.
7. S Todd and W Latham. *Evolutionary Art and Computers.* Academic Press, 1992.

Appendix: 77 Examples Evolved in a Single Session