Toward Greater Artistic Control for Interactive Evolution of Images and Animation

David A. Hart

Salt Lake City, UT, USA,
http://dahart.com/

Abstract. We present several practical improvements to the interactive evolution of 2D images, some of which are also applicable to more general genetic programming problems. We introduce tree alignments to improve the animation of evolved images when using genetic cross dissolves. The goal of these improvements is to strengthen the interactive evolution toolset and give the artist greater control and expressive power.

1 Evolving Images

There have been many studies using different kinds of genotypes for evolutionary image synthesis and design. [2–4, 6, 7, 10, 11]. Perhaps the most well known image synthesis technique using interactive evolution was explored in Sims' seminal paper "Artificial Evolution for Computer Graphics" [10].

Sims used symbolic functions as genotypes, and simply evaluated the evolved functions at every pixel to express the genotypes into rendered image phenotypes. Symbolic functions are naturally represented as parse trees, with variables and constants at the leaf nodes, and parametric functions composing the internal branching structure. These tree structured genotypes start small and evolve complexity over time, as opposed to genetic algorithms that define fixed-length genotypes with a flat, linear structure.

1.1 Function Set

For design simplicity, we use a function set for genotypes consisting of only scalar valued functions. We have avoided the use of complex iterative operations, such as image processing operations, or pre-defined fractals.

Most functions in our set are purely functional, as opposed to procedural or imperative, but we do support imperative variable assignment for reasons we will discuss in section 2.5. Variable assignments are not allowed to participate in the selection process, however, because they introduce dependencies that are incompatible with cross-over mating and most mutation strategies.

For the images in this paper, we use arithmetic operators ($+$, $-$, $*$, $/$, $<$, $>$, $==$, $?:$), *sqrt*, *pow*, transcendentals (*sin*, *cos*, *asin*, *acos*, *tan*, *atan*, *exp*, *sqrt*, *log*), 1D-4D perlin noise functions (*noise*, *turbulence*), 1D curve functions (*lerp*, *smoothstep*, *linear*, *bspline*), and aliases a, r and *length* for polar coordinates. Several example genotypes and their corresponding phenotypes are pictured in figure 1.

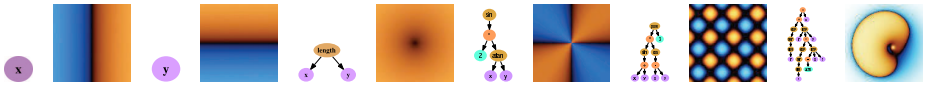

Fig. 1. Example genotypes and their expressed phenotypes: x; y; $length(x, y)$; $sin(sin(2 * atan(x, y)))$; $pow(sin(x + y) * cos(x - y), 3)$; $exp(sqr(log(r))) + lerp(r, (exp(sqr(log(r))) + lerp(-2.778, x, r)), y) + x$

1.2 Dynamic Range

Figure 2 illustrates a common issue with rendering symbolic functions: output values fall outside the range of displayable colors unless a compressing color mapping is used (consider $tan(x)$, for example).

We have found it useful to treat the output from symbolic functions as high dynamic range images, and apply any one of the standard tone operators to the image before display. The user can choose from a selection of tone operators.

In our experience one of the easiest to use has been Reinhard's "simple" operator $L/(1+L)$ [9], which smoothly compresses luminance values in the range $[0, +\infty)$ to $[0, 1]$. The user is of course given exposure and gamma controls.

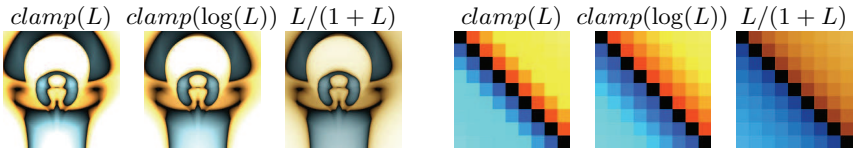

Fig. 2. Tone reproduction of symbolic functions

1.3 Mutations

Mutations on tree structured genotypes can be thought of as regular expression search and replace. For example, below we list the mutation types described by Sims [10] in loose regular expression form. Typically, all mutations are combined into an aggregate mutation by assigning each one a weighted probability, and normalizing the weights to one. In the example below, 'a' matches a node and its sub tree, 'n' matches constants, and 'v' matches variables. treeRand() constructs a random expression sub tree, and nodeRand() constructs a new node with given children.

$$
\begin{aligned}
Tree &: a \rightarrow treeRand() \\
Const &: n \rightarrow n + gaussRand() \\
Var &: v \rightarrow randVar() \\
Func &: a(a_1, a_2, \ldots) \rightarrow nodeRand(a_1[, a_2[, \ldots]*][, treeRand()[, \ldots]*]) \\
Node2arg &: a \rightarrow nodeRand([treeRand()[, \ldots]*], a[, treeRand()[, \ldots]*]) \\
Arg2node &: a(a_1, a_2, \ldots) \rightarrow a_{rand()} \\
Copy &: a \rightarrow treeCopy(getRandSubtree(getRoot(a))
\end{aligned}
$$

1.4 New Mutation Types

Warp Mutations

$a \rightarrow a + treeRand();$ $\quad v \rightarrow v + const(gaussRand());$

$v \rightarrow v + treeRand();$ $\quad v \rightarrow nodeRand(v, treeRand());$

These are useful specializations of the $Node2arg$ mutation type. Examples are shown in figure 3.

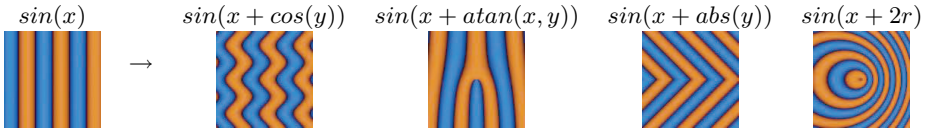

Fig. 3. Example warp mutations

Harmonic Mutations

$T(x, y) \rightarrow T(x, y) + b * T(c * (x - j), d * (y - k))$

Where sub tree $T(x, y)$ matches any function of x and y, any sub tree that contains x and y. b, c and d are random constants (typically $b < 1$ while $c > 1$ and $d > 1$). We often use $c := gaussRand() + 2$, $d := c$, and $b := 1/c$. j and k are optional random phase offsets. More generally, all variables in the sub tree can be mutated in the same fashion, making harmonic mutations applicable to all non-const nodes in the genotype.

This mutation represents the addition of a single harmonic. It is a way of introducing higher complexity and spatial frequencies into an evolving image while preserving the basic structure of the current image (see figure 4). Mandelbrot and IFS fractals are sometimes used to address this problem, though we prefer to avoid them only because their aesthetic styles can be so identifiable and visually dominant.

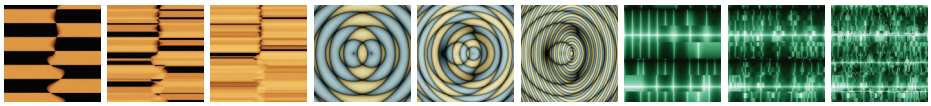

Fig. 4. Example harmonic mutations

1.5 Mutation control

Providing user controls over mutation types, rates and biases has proven extremely useful, given that the human user's "fitness function" is constantly changing. Early in the process, one may want to make very large constructive changes ($Node2Arg$). Later when nearing a finished image, small mutations

are desirable (Var, $Const$). In the middle of the process destructive mutations ($Arg2Node$) can become very useful for simplifying genotypes that are too complex and slow to render, or for distilling an image down to its basic characteristics. But it is not uncommon to want small changes early, or large changes late in the process. Therefore, we feel control over mutations is best left to the user. Every automatic heuristic we tried inevitably reduced usability in some situations. Here are some simple user controls we have provided to address mutation control.

- Global relative mutation probability. A default of 10-25% seems to work well.
- Global absolute mutation rate, e.g. 2, 4, 8 nodes total.
- Control individual mutation types- allow the user to specify "var" mutations only for example.
- Control relative mutation rates for the aggregate mutation- user can change the probability for each mutation type, and the weights are re-normalized.
- Node height bias- mutation probability depends on a node's depth. High probability near the root leads to large changes, while high probability near leaves makes smaller adjustments.

1.6 Genotype transforms

We provide the user with several genotype transforms that change the way genotypes respond to mutation. New variables or function definitions can be created from the current genotype. This expression aliasing is useful for strongly reinforcing the characteristics of an image in later images, without having to manually balance these characteristics throughout the selection process. When fine tuning is required at a later stage, the user can also expand these aliases back into primitives x and y.

Additionally a user can permeate a genotype with (initially benign) constants by inserting offsets and/or scaling factors in all but the const nodes: $a \rightarrow a + 0$; $a \rightarrow 1 * a$; $a \rightarrow 1 * a + 0$. Limiting mutations to constants is a good way to make fine-tuning adjustments. But often constants may not have evolved into every part of a genotype the user might wish to adjust. Inserting constants throughout the tree makes a genotype far more responsive to mutations on constants.

2 Alignments for animating evolved images

Sims' genetic cross dissolve [10] is a technique that aims to blend or morph smoothly between two phenotypes by bridging similarities in the genotypes. Of all the animation techniques Sims describes, the genetic cross dissolve is the most analogous to traditional key framing of poses, and in practice is the easiest to use. Genetic cross dissolves have been used to animate linear fixed length genotypes, such as Draves' evolved fractal flames "Electric Sheep" [3]. This is fairly straightforward when using fixed length genotypes because there is a one-to-one mapping between the parameters.

Finding the similarities between tree structured genotypes, however, has traditionally been a difficult problem. Sims described the genetic cross dissolve as a technique to be applied to "two expressions of similar structure [...] by matching the expressions where they are identical and interpolating between the results where they are different. If the two expressions have different root nodes, a conventional image dissolve will result. If only parts within their structures are different, interesting motions can occur." [10] Sims uses this same matching technique for cross-over mating.

2.1 First-difference alignment

The process of finding matches between two genotypes is called *alignment*. Alignment of genotypes is an entire sub field of computational biology [1, 8]. We will refer to Sims' matching technique as the *first-difference alignment*. By aligning only the identical parts of the genotypes, and stopping at the first encountered difference, any and all similarities underneath are ignored. This can produce animation that is unrelated to the structural evolutionary similarities between the genotypes. In the worst case, when root nodes differ, the result is a static fade as Sims noted, rendering the dissolve ineffective as an animation tool. Unfortunately any two given genotypes are likely to have differing root nodes.

We believe this limitation is severe and unnecessary. There is no technical reason we cannot assign correspondences arbitrarily to non-matching nodes. In fact it is generally desirable to continue matching nodes after encountering a difference between the trees, since the result of the blend will often move rather than fade, as depicted in figure 5.

We now examine ways to relax the similarity constraint on the genetic cross dissolve. We first present several alternative algorithms for computing tree alignments. We then present a generalized evaluation procedure for genetic cross dissolves, using as input a tree alignment. Finally we show some visual results using these algorithms.

2.2 Leaf node alignment

A special case of the first-difference alignment is when both trees have identical node types and structure, and the only differences happen at the leaf nodes. In this case, the alignment is a trivial one-to-one mapping, making it somewhat analogous to cross dissolves between fixed-length genotypes.

Leaf node alignments are particularly easy to use and are useful for animating, so to facilitate this type of alignment, we can construct one a-priori rather than search for it. A pre-aligned leaf node dissolve can be constructed by replacing all leaf nodes (constants and variables) in a tree with a blend node, initially blending each replaced leaf node with a copy of itself (e.g. $x \rightarrow lerp(t, x, x)$). Then to animate the leaf node alignment, mutation and selection proceeds with all operations restricted to modifying only the end point of any blend node in the tree (e.g. $lerp(t, x, x) \rightarrow lerp(t, x, y)$). This way, the user can first evolve a start frame, and then evolve the end frame directly from there, with the dissolve

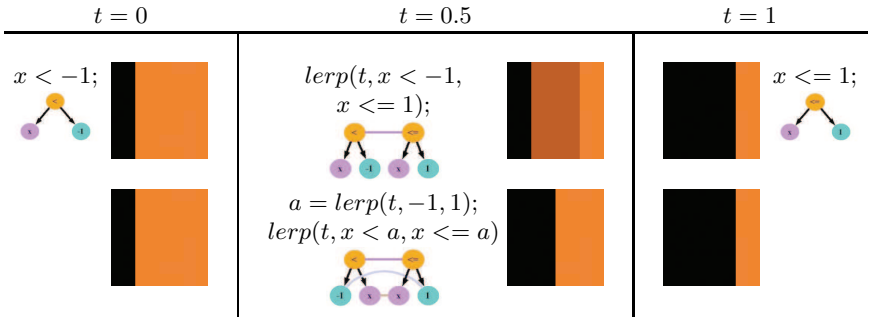

Fig. 5. A simple example of first-difference alignment (top) versus a full alignment (bottom), applied to two similar expressions with differing root nodes. The first-difference alignment produces a static region that fades over time, while the full alignment produces a moving boundary.

automatically built into the expression. With this approach, there is often no need to compute or examine animated sequences during the selection process, which can become painfully slow, only the end frame need be shown.

2.3 Constrained alignment with linear cost

To explore full tree alignments, we first construct a simple ad hoc procedure to assign correspondences between the nodes of two trees. The nodes in a tree can be arranged into rows in a variety of ways. We use the distance from the root node as the node's row number. If both trees are arranged by row, we can assign a mapping between the rows, and then for each corresponding pair of rows, assign a mapping between the nodes, as depicted in figure 6. To guarantee that the dissolve produced using this alignment will be a valid tree, the root nodes are forced to match, and below the root we use an order preserving bijection for both the row mapping between trees and the node mapping between corresponding rows. The number of such bijections between two sets of size m and n, where n is size of the larger set, is $\binom{n}{m}$. We pick one of them at random.

In practice, this alignment performs well on trees that are very similar, but a single random alignment will rarely be optimal, especially for trees that are not very similar. Because of this, and because the alignment can be computed very quickly, we generate a large number of them, each time making different random choices for the row and node mappings. We then *compute a score for each random alignment*, based on the scoring scheme described in the next section, and present a small selection of the best alignments to the user for selection. This allows for some user control over the resulting animated dissolve.

The first advantage of the constrained alignment is in allowing user to select the alignment from a set of randomized choices, which has the very practical advantage of user control, which the other alignment techniques discussed here lack. Selection of the alignment also opens up the possibility of evolving new

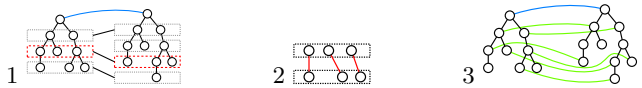

Fig. 6. Constrained alignment. 1) Align rows with a random, order preserving bijection. 2) For each matched row pair, align the nodes similarly. 3) The resulting alignment.

alignment criteria. The second advantage is its sheer simplicity- it is very easy to understand and code, and very fast to execute.

For displaying constrained alignments, we often evaluate only the midpoint (t=0.5) of the blend. Because the user has already seen both endpoints of the dissolve, seeing only the middle frame of the dissolve is usually enough to give a good sense of its quality.

2.4 Optimal alignment with quadratic cost

Jiang et. al. [5] propose an algorithm for optimal alignment of ordered phylogenetic trees, and show that the cost for ordered and unordered binary trees as well as ordered n-ary trees of bounded degree is $O(|T_1| * |T_2|)$ where $|T_1|$ is the number of nodes in tree T_1.

We present a brief summary of their algorithm, and then outline some minor modifications needed in order to use this algorithm for our purposes. The basic idea is to examine all possibilities between a single pair of nodes that may have children, assign a score for each possibility, select the one with the best score, repeat recursively, and show by induction that the final score will be that of the optimal alignment. The alignment itself is computed using backtracking.

Consider figure 7. On the left we examine all possible ways to align a pair of nodes and on the right we examine every way to align a pair of forests of children, assuming the two parents are matched. Note that both of these operations are self-recursive and mutually recursive. It follows that every node in T_1 is compared against every node in T_2, and every forest of children of a node in T_1 is compared against every forest of children of a node in T_2. Intuitively this explains why the cost of the algorithm is $O(|T_1| * |T_2|)$.

The only change needed to align ordered binary trees is to remove the single out-of-order comparison from the forest matching case. The algorithm also easily extends to n-ary trees by expanding the two node-match cases in the forest match test to all possible injections (one-to-one mappings) of nodes between the two forests. Examining all possible injections includes allowing nodes on either side to align with nothing (called a "space"). Jiang et al's scoring metric rewards matching a node with another node, and penalizes matching nodes with spaces, to prefer alignments with fewer spaces whenever possible.

Jiang et al. describe a scoring measure μ as follows. Nodes have only a single attribute, called a label, designated "a", "b", etc.. Spaces are designated by "λ". They define $\mu(a, b) = 0$ when $a = b$, $\mu(a, \lambda) = 1$, $\mu(\lambda, a) = 1$, $\mu(a, b) = 2$ when $a \neq b$.

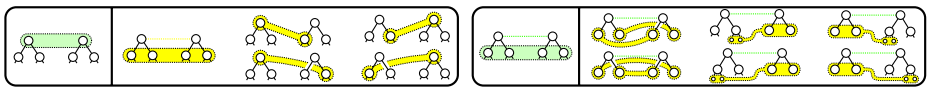

Fig. 7. Optimal binary tree alignment. On the left we consider all possible ways to match a pair of nodes. On the right we consider all possible ways to match the two forests of children of a pair of matched nodes.

In our case, the label represents the node type (e.g. const, var, '+', sin, etc.,) but some nodes have an additional piece of data that must be considered separately. We want two different variables to be less likely to match than two of the same variable, and more likely to match than, say, a variable and a sine function. Therefore, we must modify the scoring measure. The $func$ node type refers to a user-defined function application, and its data is the function definition.

$\mu(a, b) = 0$ when $a = b$, and $a \notin \{const, var, func\}$
$\mu(a, b) = 0$ when $a = b$, and $a \in \{const, var, func\}$, and $a.data = b.data$
$\mu(a, \lambda) = 1$, $\mu(\lambda, a) = 1$
$\mu(a, b) = 1$ when $a = b$, and $a \in \{const, var\}$, and $a.data \neq b.data$
$\mu(a, b) = 2$ when $a \neq b$, or ($a = func$, and $a.data \neq b.data$)

We use unordered alignment for unordered binary nodes, and ordered alignment for ordered binary and n-ary nodes. Addition, for example, is an unordered operation, while subtraction is ordered. This mixed-order alignment scheme clearly still has $O(|T_1| * |T_2|)$ complexity.

The advantages of using optimal alignment are that we get the best correlation of similar structure in the genotypes, under criteria defined by $\mu(a, b)$. The disadvantages include high memory and time complexity, and the lack of user control or the option for selection of the alignment. Examples of this alignment applied to evolved genotypes are shown in figure 10.

2.5 Using an alignment to contruct the dissolve expression

After we align two expression trees, we then need to be able to evaluate it as a dissolve. Sims described a specialized evaluation procedure for the first-difference alignment which traverses the identical parts of both trees simultaneously, but stops descent and performs an interpolation between any differing nodes. [10] Full alignments cannot use this evaluation procedure since they typically cannot be traversed simultaneously.

We instead use the alignment to connect both trees into a single explicit expression for straightforward evaluation. This can be done by creating a blend node for each pair of matched nodes in the alignment, creating a blend node between the root nodes, and then rewiring the tree to allow for imperative evaluation, as depicted in figure 8.

After the dissolve has been generated, we apply a final optimization step to speed evaluation and reduce redundant nodes by traversing the tree and collapsing blend nodes and their two children into a single copy of one of the children, in the case that both child nodes and their sub-trees are identical.

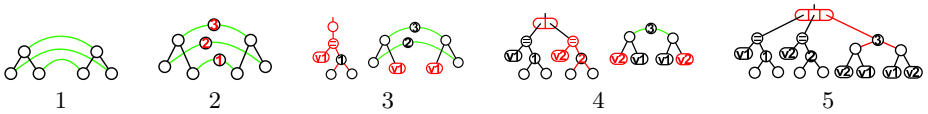

Fig. 8. Using an alignment to construct the imperative evaluation cross dissolve. 1) Align trees and create a blend node for each match. 2) Sort the matches from bottom to top. 3) Cut the bottom-most blend sub-tree and replace connections to it with a new variable. Create a new variable assignment expression and connect the blend sub-tree that was just cut. Append the new variable assignment expression to the end of an ordered list. 4) Repeat step 3 until there is only one blend node left. 5) Append the final blend node to the end of the ordered list. The list's "value" must be defined as the value of the last item in the list.

3 Conclusions and Results

We have presented several tools that improve the artistic controllability of interactive evolution. Two new expressive mutation types were given, and genotype transforms were introduced as a way to make genotypes more receptive to mutation.

We have eliminated previous limitations when blending between evolved images by introducing tree alignment into the genetic cross dissolve process. We have created an algorithm for constructing the dissolve expression using an alignment, and presented several alignment methods, each with different advantages.

References

1. P.J. Bentley and J.P. Wakefield. Hierarchical Crossover in Genetic Algorithms. *Proceedings of the 1st On-line Workshop on Soft Computing (WSC1)*, 1996.
2. R. Dawkins. The Blind Watchmaker. *Harlow Logman*, 1986.
3. S. Draves. The electric sheep screen-saver: A case study in aesthetic evolution. *Applications of Evolutionary Computing, LNCS*, 3449, 2005.
4. G. Greenfield. New Directions for Evolving Expressions. *Bridges: Mathematical Connections in Art, Music, and Science*, pages 29–36, 1998.
5. T. Jiang, L. Wang, and K. Zhang. Alignment of trees- an alternative to tree edit. *Theoretical Computer Science*, 143(1):137–148, 1995.
6. C.G. Johnson and J.J.R. Cardalda. Genetic Algorithms in Visual Art and Music. *Leonardo*, 35(2):175–184, 2002.
7. M. Lewis. *Creating Continuous Design Spaces for Interactive Genetic Algorithms with Layered, Correlated, Pattern Functions.* PhD thesis, PhD thesis, Ohio State University, 2001, 2001.
8. C. Notredame. Recent progresses in multiple sequence alignment: a survey. *Pharmacogenomics*, 3(1):131–144, 2002.
9. E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. Photographic Tone Reproduction for Images. *Proceedings of SIGGRAPH 2002*, pages 267–276, 2002.
10. K. Sims. Artificial evolution for computer graphics. *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 319–328, 1991.
11. S. Todd and W. Latham. The mutation and growth of art by computers. *Evolutionary Design by Computers*, pages 221–250, 1999.

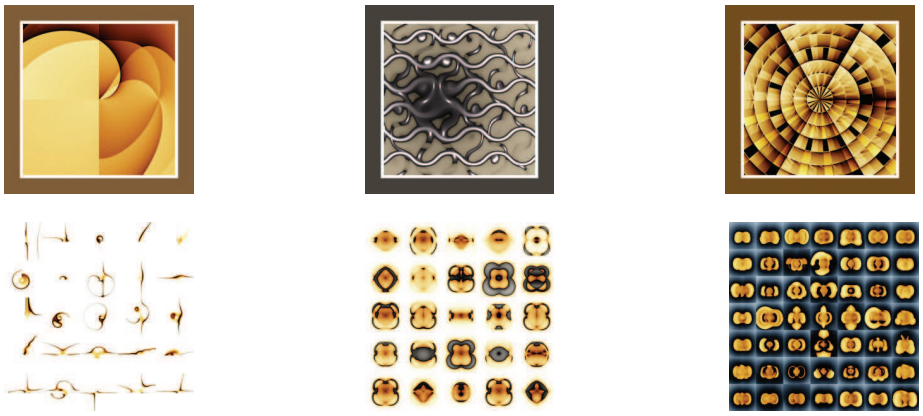

Fig. 9. Top, fully evolved images. Bottom, populations of mutated variations.

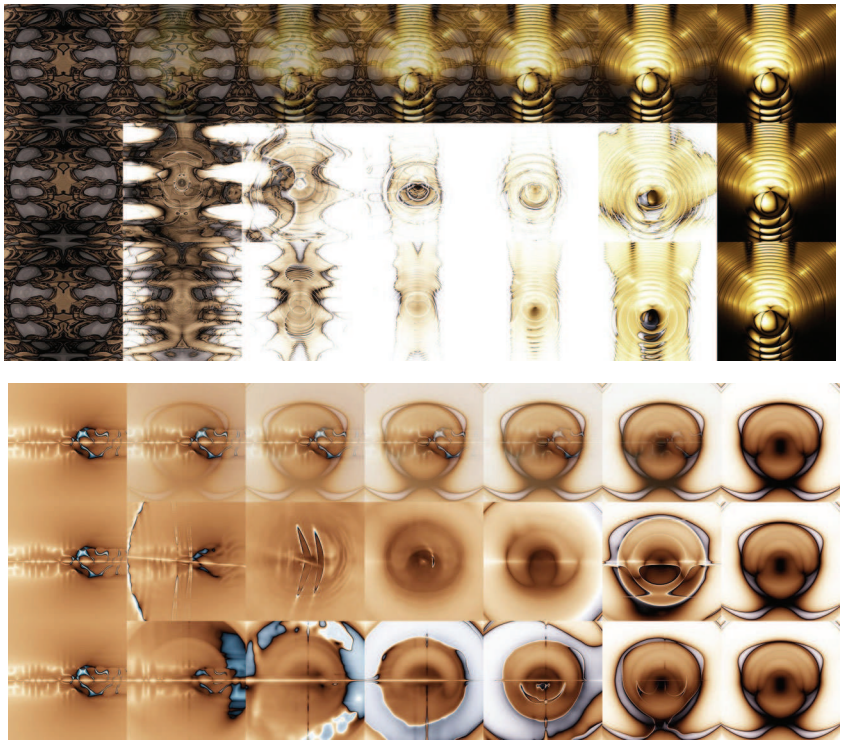

Fig. 10. Two examples of genetic cross dissolve sequences produced using different alignments on two genotypes which did not explicitly share heritage. Top row: First-difference alignment. Only the root nodes were connected, resulting in a static image fade between genotypes. Middle row: Constrained random alignment. Similar to optimal alignment, but with a little bit of (perhaps desirable) random variation. Bottom row: Optimal alignment. This gives us the most visual insight into the structures that are shared between the two genotypes.