On Artificial Intelligence and Motivation in Computer Graphics Education

Christian-A. Bohn   Jan Oliver Steinbach

Wedel University of Applied Sciences
Feldstraße 143, D–22880 Wedel, FR Germany
Tel. +49(0)4103/804840, E-Mail: bo@fh-wedel.de

Abstract
When educating artificial intelligence topics in the field of computer graphics, it has always been a challenge to generate a sufficient amount of motivation for driving students to investigate concepts of artificial intelligence to greater detail. From our experiences, this can often be achieved if students are forced to implement their knowledge in a playful manner, i.e, to implement applications which focus on fascinating instead of ”serious” results.

This work presents a re-implementations of two problems which have successfully been solved by Karl Sims in the early 1990s [Sim91, Sim94]. The approaches use *genetic algorithms (GA)* since solutions are driven by adjusting a huge set of abstract parameters which users are hardly capable of tuning it interactively. Despite the fact that the problems to be solved can hardly be applied in real world, the results are very interesting and inspiring and emphasize the fascinating idea behind it.

The two ideas were, first, the automatic invention of abstract creatures consisting of connected rigid bodies, which are capable of swimming under water efficiently, and second, the automatic creation of aesthetic pictures. Both approaches are driven by the representation of the problem under consideration by a genetic string and to let them evolve trough a GA. The first is led by a fitness function which judges from motion velocity, the second takes the user’s taste as fitness criterion by letting him decide which evolutionary transformations are applied at which selection of individuals (pictures) from the actual generation.

Apart from the original works, we present new aspects of solving the problems described above. These aspects focus on the optimization of the original algorithms, on increasing their flexibility, and on their adaption to actual hardware.

Keywords: genetic algorithm, computer graphics, evolutionary computation, animation, modeling, articulated figures, rigid body animation, image processing, textures

1 Introduction

Genetic algorithms (GA) were proposed by John Holland in the 1970s and they are based on Darwin's theory of the *Survival of the Fittest*. GA are a useful tool for iteratively finding near-optimal solutions within an arbitrary search space [Mit99]. They are usually applied to search spaces of vast dimensions where it is hardly possible to find the optimal solution due to the huge complexity of deterministic search algorithms.
In contrast to classical techniques which take a start vector and modify its components according to a certain gradient criterion, GA take a set of random vectors, take them as genom strings, and let them develop by simulating evolution as it happens in nature, i.e., mimicking recombination of a pairs of vectors and mutation of singles.
The vital challenges in realizing a genetic algorithm are, first, to find a suitable coding of the problem under consideration as a string-like data structure similar to a genome string — the *genotype*. Second, from the genotype the *phenotype* — the "real" solution of the problem — has to be developed which often can be a very time-consuming procedure since usually millions of evolutionary transformations have to be accomplished until a valuable solution has been found. Third, like in nature, a *fitness function* is required which judges from the quality of a phenotype if an element from the set of solutions should be selected for procreation to the next generation. For a more detailed description of genetic algorithms we recommend [Mit99].

In the following, we describe two re-implementations of two problems which have successfully been solved by Karl Sims in the early 1990s [Sim94, Sim91]. The first GA (section 2) is on inventing virtual creatures consisting of connected rigid bodies. The GA invents a body topology of a creature together with the motion mechanism in a way that it is capable of swimming in an under water environment efficiently (fitness criterion). The second GA approach (section 3.1) works on genoms which carry mathematical expressions from which procedural pictures are calculated. The fitness criterion in this case is represented by the user who interactively selects individuals from the actual generation.

2 Evolved Articulated Figures

In this section, creatures are developed, i.e., their bodies which are sets of connected rigid bodies together with a specific neural network which controls a creature's movements through stimulating virtual muscles. Here, movement is focussed on swimming effectively in an underwater-environment in a simulated three-dimensional physical world.
We developed our algorithm by using the language *Haskell* [Hud00] which is a pure functional programming language based on lazy evaluation. Graphics output was realized using *HOpenGl* which is a Haskell binding for *OpenGL*. Haskell allows the definition of complex data structures in a very simple and elegant way which — together with its outstanding ability of function composition — makes it the first choice as an implementation language for this application.

2.1 Genotype

2.1.1 Creature Morphology

The morphology of a creature is represented by a directed graph. Each node relates to a rigid body element and these elements are connected by joints forming the whole body of the creature. The graph enables efficient parent-child-relationships like recursion and symmetry between rigid bodies. Recursion is needed if a node element reoccurs as one of its own children (like limbs of a snake). Symmetry is used to duplicate an element symmetrically (for pairs of legs, arms, etc.). This enables reusing a rigid body without creating

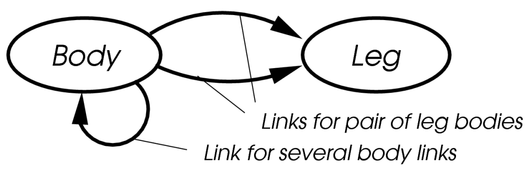

Figure 1: Example for a snake-like creature with two legs at the last limb. The topology is realized by only two real object instantiations with several links for each rigid body.

a new object for each instance (see Figure 1 for example).

We define two types of joints which connect rigid bodies of the creature: rotational and translational, each of them with one degree of freedom. Each joint has a specific force coefficient which determines if a symmetric children is affected in the same or in the opposite manner (by a positiv or a negativ value). For example, two wings of a flying bird are assumed to being moved similar and in the same direction whereas legs of a walking monkey move in opposite directions.

The graph is implemented as an n-dimensional tree where each node is a singleton. Duplicated nodes (due to recursion or symmetry) are not copied. Instead, links are realized as imported attributes which contain information about children, recursion, and symmetry. From the Haskell's point of view, a children's attribute is a list of tuples: $\texttt{children} = [(\texttt{id}, \texttt{pos})]$ with the position `pos` of a certain child and its identifier `id`. The recursion attribute is defined as $\texttt{recursion} = [(\texttt{pos}, \texttt{scale})]$ which virtually copies the node to `pos` and scales it according to a coefficient `scale`. Symmetry is defined as $\texttt{symmetry} = [(\texttt{id}, \texttt{axis})]$ where `axis` defines the rotation centrum which is used to generate the symmetric, second element. The advantage of this concept is that every node to be transformed during evolution processes has to be transformed only one time, i.e., in this case, duplications are avoided.

Every node stores information about the type of joint connection to the parent's body. It defines the joint's position (relative to the node's center of mass), its type, its main axis and the according force coefficients. Additionally each joint has a mobility term which limits its movement to a certain amount, like joints in nature are limited by its maximum muscles extent.

2.1.2 Creature Control

Every body element has a nested local neural network which controls muscles affecting the child elements. Additionally, there is one central neural network which may seen as the creature's brain. The whole neural system consists of three different node types: sensors, effectors, and neurons. A sensor gathers information from the environment and passes it to neurons and effectors. A neuron represents a mathematical function with several inputs (from sensors or neurons) and calculates one output value which can be used as an input for other neurons or effectors. An effector is used to directly control a muscle. Its output corresponds to the force acting on a specific joint.

Implementation of node communication uses relative identifiers. Every node of a local network is able to communicate with local nodes or the nodes of the brain. In Haskell, the communication channel `relId` is defined as a tuple: $\texttt{relId} = (\texttt{morphId}, \texttt{controlId})$ where `morphId` references a body node and the `controlId` references a node of a neural network, i.e., $(-2, n)$ references the n^{th} node of the brain, $(-1, n)$ the n^{th} node of the parents' neural network, $(0, n)$ the n^{th} node of the same neural network, and (m, n) references the n^{th} node of the neural network of child m.

Using relative references has the advantage that, when applying evolutional processes, they do not necessarily need to be updated if the children of a rigid body node change. Additionally, since a neuron's function is implemented as $\texttt{NeuralFunc} :: [\texttt{Double}] \to \texttt{Double}$, it is independent from its number of inputs, i.e., after an evolution step functions are still valid even without an explicit adaption process.

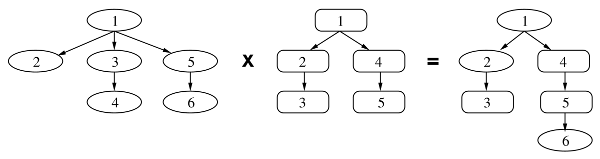

Figure 2: Example for recombination of two parent graphs.

2.2 Evolution of the Genotype

2.2.1 Recombination

A child arises from combining the genotype of a pair of parent graphs. Recombination combines topological elements of the creature body as well as their behavior. First, all nodes of the parents' graphs are numbered according to an LR traversal. Then two random values n and m are chosen and the first n nodes of the first parent are taken while the first n nodes of the second are dropped. The taken nodes become the first part of the child. After that m nodes of the second parent are taken and m nodes of the first are dropped.
This process continues until no nodes are left. Every taken node is inserted at the same position in the child graph like it had in the parent graph in order to guarantee that a child of the root element remains child of this element, i.e., a leg remains a leg (see Figure 2 for an example).
After this, validity of the new creatures' references are proven. Children related attributes of a body node have to be modified if the number of child nodes has changed. Since these attributes are lists, elements can simply be removed or added. New list elements are created at random constrained by an additional validity check.

2.2.2 Mutation

For each element in the body, probability terms for mutation are defined. According to them an element is either, first, deleted, second, a child is added, or, third, the element is modified. Whenever an element is removed all references to them are also deleted. Modification of an elements' attributes are divided into the following six individual steps: modification of

1. joint axis and joint position,
2. body size,
3. joint mobility,
4. force coefficient,
5. recursion, and
6. symmetry.

Each of them underlies certain restrictions, i.e., a predefined list of valid values are taken for mutation only.

2.3 Phenotype

The morphology of the phenotype consists of an n-dimensional tree of rigid bodies. In the first step of the transformation of the genotype to the phenotype, the genotype tree is expanded, i.e., every virtual node is instantiated by (copied to) a real object.
The neural network is implemented as a list together with a map storing the state of each element of the network during simulation of the creature's movements. Thus, at each key frame a phenotype of a creature is generated by, first, the traversal of the tree and second, by concurrently reading the state parameters from the map.

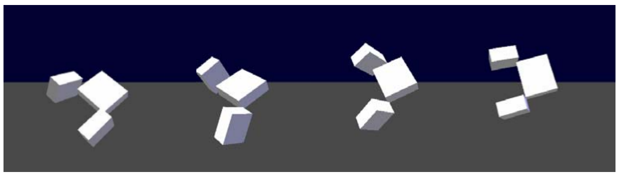

Figure 3: An animation sequence (time increases from left to right) of a creature generated by the presented GA approach.

2.4 Fitness Function

The amount of fitness of a creature is determined by its velocity when moving under water. The simulation is based on the *articulated body method* [Fea87] which calculates joint accelerations based on position, velocity and external forces on a body. The new positions and velocities of each body element are calculated by using the *Runge-Kutta-Nyström* integration technique [Dan02]. Collision detection is realized using *object oriented bounding boxes* [Par02].

3 Genetic Pictures

In our second example of "playing with genetic algorithms", we developed a technique to generate very complex, visually aesthetic two-dimensional procedural images.
The main point in our work is that the according procedures are generated by an evolutionary process where the fitness function is realized by the interaction of a user through a graphical user interface (GUI). The user selects pictures from a generation which "look nice" and these are used to create the next generation.
We implement this technique on a usual PC running a *C++* development environment. From our point of view, using *C++*, i.e., using its pointer mechanism, is the only possibility for implementing such an approach efficiently. We give some hints for proving this statement in the following. Also, in contrast to existing implementations, we chose an object-oriented approach where each chromosome is a single object connected through pointers to other elements of the genotype. Because of that and the strict distinction between genotype (data keeping) and phenotype (data analysis) we are able to keep any modification of the genome on a constant level through all generations.

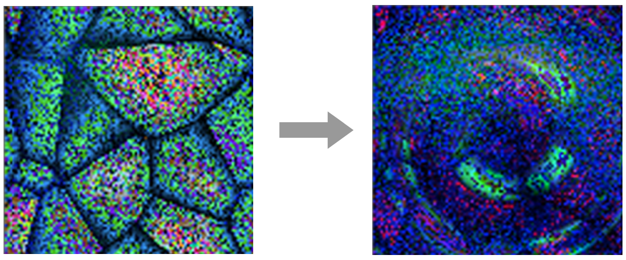

Figure 4: A genom evolving by head rotation.

3.1 Genotype

The genotype describes the whole genetic information of an individual, whereas the phenotype is the visual appearance of shape, color and pattern determined by the genotype. The genotype is represented by a genome consisting of several chromosome strings with constant length. Single chromosomes within these strings are instantiated by mathematical expressions as trees of different shapes and sizes. A chromosome is the smallest unit of the genetic code and consists of operations (function with $n \in \mathbb{N}_0$ arguments) or constant values like constants, pixels positions, color codes, a pixel map, or gradients). We limit procedure types to being constant factors (`ZeroArgChrom`), one-argument operations (`OneArgChrom`), or two-argument operations (`TwoArgChrom`). Examples related to the number of parameters are

- `ZeroArgChrom`: constant values, noise-, turbulence-, or ramp generators,
- `OneArgChrom`: absolute, sine, cosine, swirl transformation,
- `TwoArgChrom`: plus, modulo, xor, warp-swirl transformation.

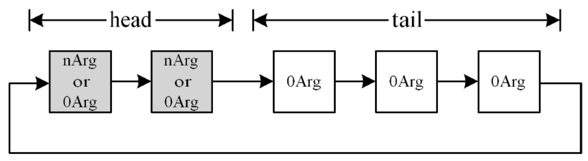

Figure 5: Structure of a single chromosome string with head size of 2.

Each chromosome string consists of a head and a tail whereas the overall length of the tail is $t = h * (n - 1) + 1$ with h the length of the head and n the number of arguments of the chromosome with the highest number of arguments (see Figure 5). The head may contain `TwoArgChrom`, `OneArgChrom` (n-`ArgChrom`), as well as `ZeroArgChrom` whereas the tail must contain `ZeroArgChrom` exclusively. Chromosome strings are implicitly represented by ring lists.
Sims' original technique is based on evaluating and mutating symbolic expressions returning a calculated color for each pixel coordinate (x, y). In contrast, we decided to formulate all imaging operations as functions over the whole image by mapping scalar-, vector-values, or pixel maps to a unique image class.
Each genome may consist of part-evaluation trees (chromosomes) of any size, shape and complexity which contain unused nodes (see Figure 6). Such evaluation trees arise from the evolutionary process of dividing and reorganizing chromosomes like they were strings without any internal semantics.
Unused information in a chromosome string does not influence the actual phenotype but it may influence later generations if it becomes an active part of a chromosome string ("*coding chromosomes*", see Figure 6).

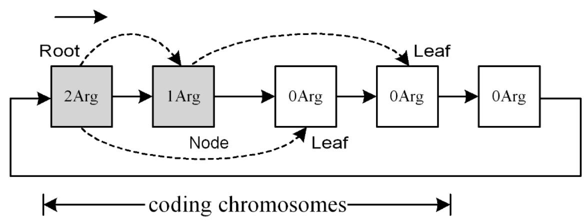

Figure 6: Example of a partial evaluation tree where the outer right chromosome is unused.

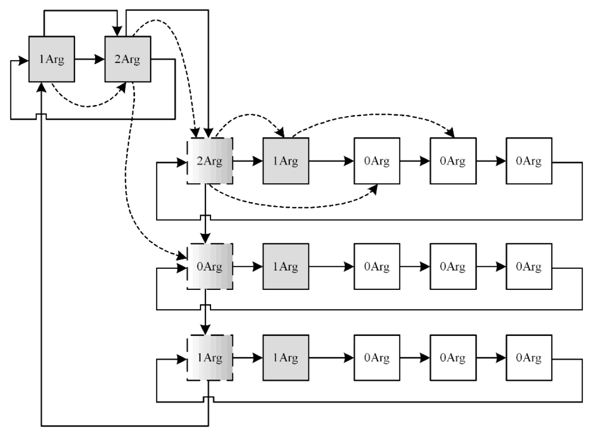

Figure 7: An entire genome with a head size of two and three chromosome strings. The string on the top carries two `PreHeader` chromosomes which both are active, in the middle and on the bottom, two partial chromosome strings are exposed, the lowest of them is completely inactive.

Individual chromosome strings are connected by a so-called `PreHeader`, which is a ring consisting of head chromosomes (the root tree of the genome). The chromosomes of the `PreHeader` can be seen as another ring which contains all chromosome strings belonging to a certain genome (see Figure 7).

3.2 Phenotype

The phenotype is the result of a walk through the genome's expression tree, beginning with the root, i.e., evaluating the functional term which the tree defines. Like in nature, individuals vary in shape, color, and pattern only due to the order and structure of the chromosomes. Each evaluated chromosome returns either a scalar (grey scale value), a vector (RGB value), or a matrix (pixel map). To guarantee validity of the input parameters of a certain function, all functions are defined in a way that they deliver a data type which equals the most complex type of its input data. Thus, a function of a matrix and a scalar delivers a matrix, whereas a vector and a scalar as inputs results in a vector as output.

3.3 Evolution of the Genotype

3.3.1 Mutation

Mutation generates new genotypes from a single selected parent by either rearranging the order of chromosomes or by substitution of one or more chromosomes in a randomly chosen chromosome string. Since our fitness function is represented by the user, he will decide about the genom which will mutate and how it will change its chromosomes. The user can chose one of the following operations.

- **Transposition:** Two randomly selected chromosomes within one chromosome string (without the root) are either interchanged or the first chromosome is replaced by a clone of the second chromosome. To maintain consistency of the resulting strings both selected chromosomes must remain either part of the head or of the tail.

- **Root Transposition:** The first chromosome of a selected chromosome string (root) is replaced either by a second randomly selected chromosome of the head of the same string or by a clone of the second chromosome.

- **Substitution:** A randomly selected chromosome of a chromosome string is replaced by a another randomly generated chromosome according to the first chromosomes' location (head or tail).

- **Head/Tail Rotation:** A chromosome string is separated into two individual strings of only head and of only tail chromosomes. Then, one of these is rotated several times in advance to putting both together again forming a single chromosome string.

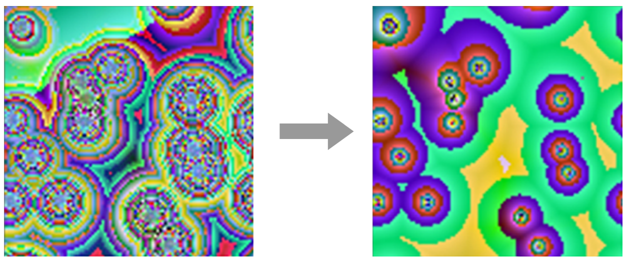
Figure 8: Substitution followed by a tail rotation.

3.3.2 Recombination

Recombination obtains a new genotype from two user-selected parents' genotypes and it results in two child individuals with mixed genetic information from both parents. Here, the user may chose between three different ways of recombination as follows.

- **Crossover:** One position inside of a pair of chromosome strings is chosen which divides both chromosome strings in a head and a tail part. Then, head and tail of the two strings are interchanged. The two newly generated chromosome strings inherit features of both parent strings. To avoid emphasizing properties of one parent, crossover can be repeated with a different head and tail separation.

- **Random Copy:** Two individual chromosomes at arbitrary positions in each chromosome string are selected and copied into the child's chromosome string.

- **Random Interpolation:** Two chromosomes are selected from two strings

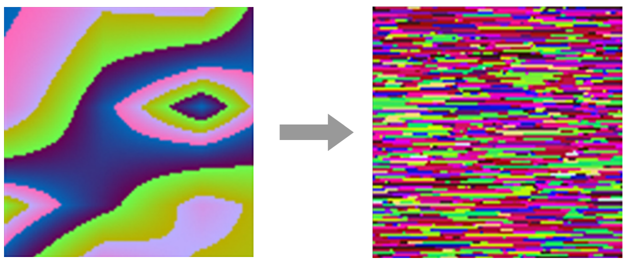
Figure 9: Head rotation followed by transposition.

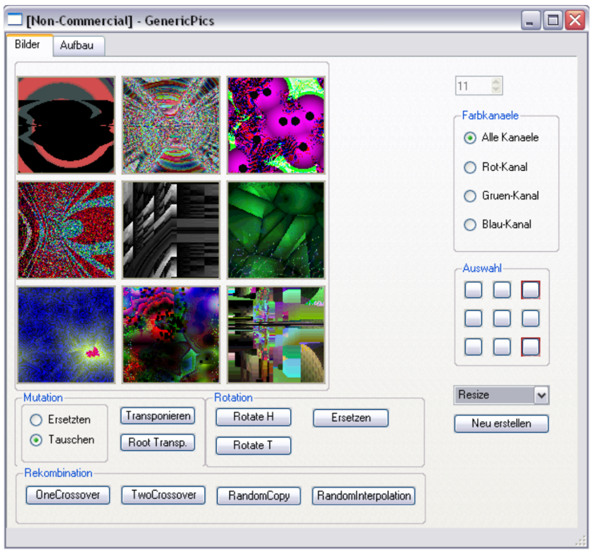

Figure 10: The GUI for choosing mutation or recombination.

and linearly interpolated, where the interpolation parameter is determined at random.

3.4 Fitness Function

The program was written using the *Qt*-library as user interface. Figure 10 shows a screen shot of the GUI which enables the user to act as the fitness function of the evolution.

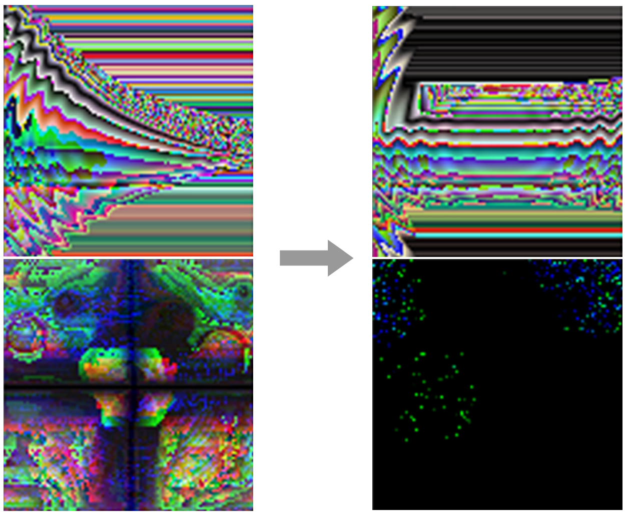

Figure 11: Random copy between the pair of images on the left resulting in the right column of pictures.

4 Results and Summary

We presented two implementations which expose the fascinating facilities typical in most GA approaches.

4.1 Evolved Articulated Figures

In section 2, a system is described that generates three-dimensional creatures which are able to swim under water. This process is totally autonomous. No user efforts are needed. The presented data structure for describing the genotype provides virtually unlimited possibilities concerning variations of body shape and behavior. Figure 3 shows a creature which was developed by the presented GA. Four key frames of a swimming animation are exposed. Further experiments led to several interesting results concerning the facilities of the GA. Although the algorithm runs automatically it seems to be strongly depending on the user's initial parameter settings. For example, a vital parameter is the ratio of crossover and mutation operations. On the one hand, frequent usage of mutation should support fast development of complex topologies and behaviors, but, on the other hand, this tends to destroy complex patterns of behavior. Another parameter that has huge influence on getting satisfying results is the probability of modifying elements during mutation. Whereas varying only a few values has a little effect on the evolution, modifying many can stop evolution by destroying reasonable development progresses. Thus, creatures of a new population are usually not more successful than its predecessors.
Other fascinating discoveries arise from the kind of the GA's handling of programming mistakes in the physical simulation. Blocking mobility of joints lead, for example, to creatures which use their legs like propellers. Concerning future work, there are various possibilities for improvement of the presented system. Most important seems to be the implementation of more complex joint types with more than one degree of freedom. For

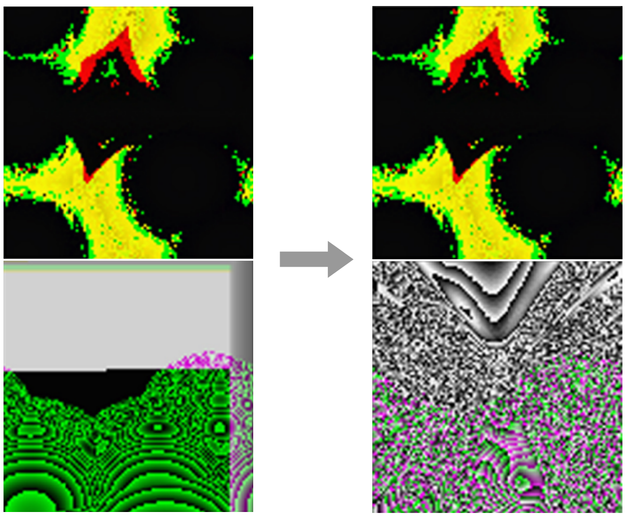

Figure 12: One point crossover between the two images on the left resulting in the images on the right. Only the phenotype of the lower picture is modified.

example, with the actual implementation creatures are hardly able to move with constant velocity since body elements that generate repulsion forces can not be rotated for reducing dragging when pulling them back.

4.2 Genetic Pictures

Figures 4 through 13 expose some phenotypes generated by the approach from section 3.1. Although these images cannot be seen as a prove of the genetic operations, some picture seem to expose a connection between the phenotype and the evolutionary transformations.
The described techniques of storing, representing and manipulating genetic information for procedural texture generation allows for a wide variety of unique individuals on a high level of possible image characteristics. The flexible mechanism of chromosome strings which form separate partial evaluation trees capable of being arranged and combined to more complex structures, enables simple extensibility by new modification routines and chromosomes.

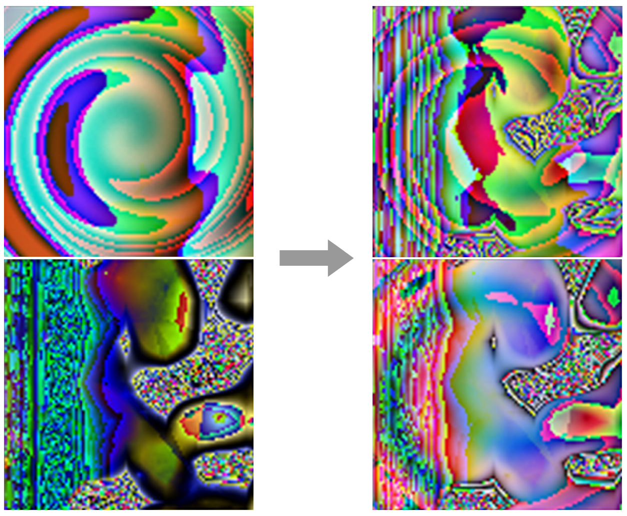

Figure 13: On the right, two interpolations between the left pair of images.

4 References

[Dan02] Jürgen Dankert. Numerische methoden. World Wide Web, 2002. Hamburg University.

[Fea87] Roy Featherstone. *Robot Dynamics Algorithms*. Kluwer Academic Publishers, Norwell, 1987.

[Hud00] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, Cambridge, 2000.

[Mit99] M. Mitchell. An introduction to genetic algorithms. 1999.

[Par02] Rick Parent. *Computer Animation*. Morgan Kaufmann Publishers, San Francisc, 2002.

[Sim91] K. Sims. Artificial evolution of computer graphics. *ACM SIGGRAPH '91 Conference Proceedings*, 25(4):319–328, 1991.

[Sim94] K. Sims. Evolving virtual creatures. *Computer Graphics, Annual Conference Series, siggraph*, pages 15–22, 1994.