Evolutionary Design of Objects Using Scene Graphs

Marc Ebner

Universität Würzburg, Lehrstuhl für Informatik II
Am Hubland, 97074 Würzburg, Germany
ebner@informatik.uni-wuerzburg.de
http://www2.informatik.uni-wuerzburg.de/staff/ebner/welcome.html

Abstract. One of the main issues in evolutionary design is how to create three-dimensional shape. The representation needs to be general enough such that all possible shapes can be created, yet it has to be evolvable. That is, parent and offspring must be related. Small changes to the genotype should lead to small changes of the fitness of an individual. We have explored the use of scene graphs to evolve three-dimensional shapes. Two different scene graph representations are analyzed, the scene graph representation used by OpenInventor and the scene graph representation used by VRML. Both representations use internal floating point variables to specify three-dimensional vectors, rotation axes and rotation angles. The internal parameters are initially chosen at random, then remain fixed during the run. We also experimented with an evolution strategy to adapt the internal variables. Experimental results are presented for the evolution of a wind turbine. The VRML representation produced better results.

1 Motivation

Evolutionary algorithms can be used to find structures which are optimal for a given problem. In evolutionary design [4–6], each individual represents a particular shape. Genetic operators are used to change these shapes. Selection is used to find structures which are suited for the given problem. The main question to address is which representation to use. Which representation is best suited to find shape? The representation has be be able to create all possible shapes. In addition to this requirement the representation should be chosen such that parent and offspring are closely related. If the fitness of parent and offspring are completely uncorrelated we have a random fitness landscape and evolution degrades to random search.

So far, a number of different representations have been used in evolutionary design. They range from direct encoding of shapes to generative encodings such as L-Systems [7–9, 13]. Several different representations for three-dimensional objects are known from the field of computer graphics [17]. Those representations were created to model three-dimensional objects and visualize them in a virtual environment. Some of these representations may also be used for evolutionary design. In addition to being able to create any possible shape these representations need to be evolvable, that is, parent and child have to be related. We are investigating a representation known from computer graphics, a scene graph, for evolutionary design.

2 Scene Graph

A computer graphics object or entire scene can be stored in a scene graph. A scene graph is an ordered collection of nodes. Several different types of nodes such as shape nodes, property nodes, transformation nodes, and group nodes exist. Shape nodes are used to define elementary shapes such as spheres, cubes, cylinders and cones. Property nodes can be used to define the look of the elementary shapes. The shapes are placed at the origin of a local coordinate system. In order to combine different shapes and to position these shapes relative to each other we need transformation and group nodes. Transformation nodes change the current coordinate system. Using transformation nodes we can rotate, translate or scale the coordinate system any way we like. Group nodes can be used to merge different scene graphs. The contents of the individual scene graphs are placed in the same coordinate system.

Two different types of scene graphs are commonly used. One is used by OpenInventor [18], an object-oriented library for 3D graphics, the other one is used in the virtual reality modeling language (VRML) [1]. In OpenInventor transformation nodes are used as outer nodes of the tree. The scene graph is displayed by executing the action stored in the root node and then traversing each child in turn. Transformation nodes change the current coordinate system. All nodes which are executed after a particular transformation node are influenced by this transformation. To limit the influence of the transformation nodes a special type of group node, called separator node, exists. This group node stores the current coordinate system on a stack before traversing its children. After all children have been traversed, the current coordinate system is restored. Objects commonly have a separator at the root of the tree. Thus, such objects can be combined easily to create a larger scene.

The scene graph used in VRML differs from the one used by OpenInventor in that transformation nodes are also separator nodes. In VRML transformation nodes are used as inner nodes of the tree. They change the current coordinate system but this change only affects the children of the transformation node. Thus, the distinction between separator nodes and group nodes is not necessary in VRML. Figure 1 shows the differences between an OpenInventor and a VRML scene graph which represent the same object. Let us start with the VRML scene graph. The object consists of 4 cubes (A, B, C, and D). In VRML the object is constructed by positioning cube A (node 5) using a transformation node (node 2). Next, cube B (node 3) is placed at the origin. The transformation stored in node 2 only affects node 5 but does not affect any other nodes. Cube C is positioned relative to cube B and cube D is positioned relative to cube B. This is achieved by positioning cube C (node 6) using a transformation node (node 4)

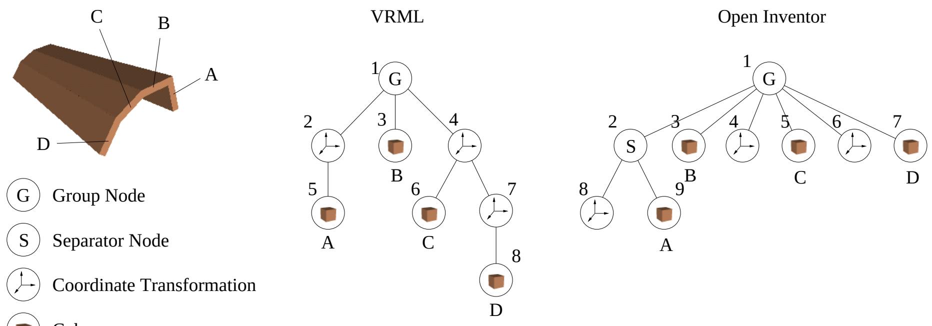

Fig. 1. Differences in the representation of the same object between VRML and OpenInventor. The same object is represented once with a VRML scene graph and once with an OpenInventor scene graph.

followed by an additional transformation node (node 7) which positions cube D (node 8) relative to the coordinate system of cube C. The transformation stored in node 4 affects all nodes below it (nodes 6, 7, and 8). Thus, cube D (node 8) is affected by two transformations (node 4 and 7).

The OpenInventor scene graph has a different structure. Cube A (node 9) is positioned using a transformation node (node 8). The effect of this transformation is encapsulated using a separator node (node 2). Thus, the transformation stored in node 8 only affects node 9. It does not affect any other nodes. Next, cube B (node 3) is positioned at the origin. Cube C is positioned relative to cube B and cube D is positioned relative to cube D. The transformation stored at node 4 changes the coordinate system and affects nodes 5, 6, and 7. The transformation stored at node 6 only affects node 7. Thus, cube C (node 5) is only affected by the transformation node 4, whereas cube D (node 7) is affected by both transformations (node 4 and 6).

Both scene graphs are trees and of course we can use genetic programming to evolve such scene graphs.

3 Evolving Scene Graphs

We have used genetic programming [3, 10–12] to evolve scene graphs. Each node is either a terminal symbol or an elementary function depending on the number of arguments. Table 1 shows the elementary functions and terminal symbols used for the OpenInventor scene graphs. The elementary functions and terminal symbols used for the VRML scene graphs are similar except that the group node is not used and nodes `Translate`, `TranslateX`, `TranslateY`, `TranslateZ`, `Rotate`, `RotateX`, `RotateY`, `RotateZ` take two arguments. These transformations save the current transformation matrix on a stack, evaluate their children and then restore the original transformation matrix. In a sense, they work like combined separator and transformation nodes. We also added a no-operation node to be able to add only a single subtree to a transformation node.

Table 1. Elementary functions and terminal symbols used for OpenInventor scene graphs

Name	Internal Vars.	Args.	Function
`Group`	none	2	Combines objects stored in subtrees.
`Separator`	none	2	Save current transformation matrix on stack, evaluate subtrees, pop transformation matrix from stack.
`Translate`	`dx`, `dy`, `dz`	0	Translates the coordinate system by $[dx, dy, dz]^T$.
`TranslateX`	`dx`	0	Translates the coordinate system by $[dx, 0, 0]^T$.
`TranslateY`	`dy`	0	Translates the coordinate system by $[0, dy, 0]^T$.
`TranslateZ`	`dz`	0	Translates the coordinate system by $[0, 0, dz]^T$.
`Rotate`	`nx`, `ny`, `nz`, `a`	0	Rotates the coordinate system by the angle `a` around the vector $[nx, ny, nz]^T$.
`RotateX`	`a`	0	Rotates the coordinate system around the X-Axis.
`RotateY`	`a`	0	Rotates the coordinate system around the Y-Axis.
`RotateZ`	`a`	0	Rotates the coordinate system around the Z-Axis.
`Cube`	`w`, `h`, `d`	0	Places a cube with dimensions `w`×`h`×`d` at the current position.
`Sphere`	`r`	0	Places a sphere with radius `r` at the current position.
`Cylinder`	`r`, `h`	0	Places a cylinder with radius `r` and height `h` at the current position.

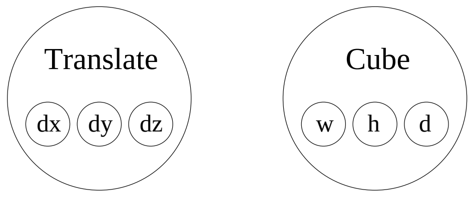

Fig. 2. Each node also contains a number of internal variables. For instance, a translation node contains a vector and a shape node describing a cube contains the size of the cube.

Floating point variables may be stored inside each node (Figure 2). For a shape node describing a cube, these variables are used to specify the width, height, and depth of the cube. For a transformation node the variables are used to specify the vector which translates the coordinate system. Thus, the topology of the object is defined by the tree structure of the individual which is evolved using genetic programming. The internal variables are initially selected from a random range suitable for each node. The values are set whenever a new node is created and then left unchanged during the course of the run. If a new subtree is created by the mutation operator, new nodes with new internal variables are generated.

These variables are thus similar to the standard ephemeral constants except that they are internal to a node. The internal variables are always exchanged with the node during crossover.

Mutation, and crossover operators are defined as usual. The mutation operator first selects an individual. Next, a node is randomly selected. Internal nodes are selected with a probability of 90%, external nodes are selected with a probability of 10%. Then this node is replaced by a newly generated subtree. The subtree is generated with the grow method with a maximum depth of 6. If the number of nodes of the resulting individual is larger than 1000 or the depth of the tree is larger than 17, we add the parent to the next population instead. The crossover operator chooses two individuals. A random node is selected in each individual and the two subtrees are swapped. Again, internal nodes are selected with a probability of 90%, external nodes are selected with a probability of 10%. Constraints are imposed on the size of the individuals. If the number of nodes of an offspring is larger than 1000 or the depth of the tree is larger than 17, we add the parent to the next population instead.

4 Evolving the Blades of a Wind Turbine

As a sample problem we have chosen to evolve the blades of a horizontal-axis wind turbine. We should note that our goal is to investigate the evolvability of different representations for evolutionary design. We are not interested in actually finding an optimal shape which can be used for a real wind turbine. Indeed, the resulting shape looks more like a water than a wind turbine.

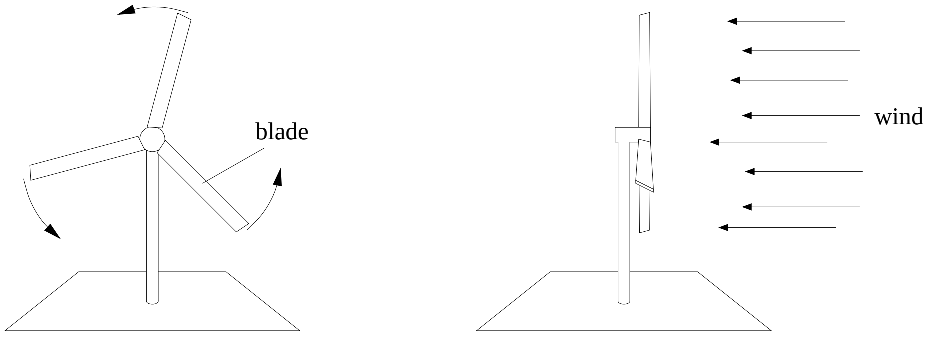

Fig. 3. A wind turbine is constructed from three blades which are rotated around a center.

To simulate the virtual mechanics we have used ODE (Open Dynamics Engine) which was developed by Smith [16]. ODE is a library for simulating rigid body dynamics in VR environments. ODE supports collision detection and collision response of primitive geometry objects such as cubes, spheres, capped cylinders and planes. A capped cylinder is a cylinder with a half-sphere cap at each end. This type of cylinder is used by ODE because it makes collision detection easier.

A single individuals represents the shape of a blade of the turbine. The turbine is constructed by adding three of the blades to a base (Figure 3). Each blade is rotated by 120°compared to the previous bade. A real turbine is moved by wind passing over both surfaces of an airfoil shaped blade [2]. This causes pressure differences between the top and the bottom surfaces of the blades. We model wind as a fixed number of small spherical particles. The particles are placed in front of the blade at random locations within a certain radius around the axis of the turbine.

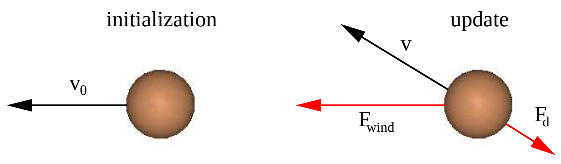

Fig. 4. Initialization and update of wind particles. Wind particles initially have a starting velocity of v_0. During each time step we apply a force F_{wind} and a damping force $F_d = -\alpha_d v$ to the particle.

Each particle initially has a starting velocity of v_0 (Figure 4). The simulation is carried out for a fixed number of steps. During each step we apply a force F_{wind} to each wind particle. A damping force F_d is also applied. This damping force is directly proportional to the velocity of the particle $F_d = -\alpha_d v$. Thus, the total force acting on a single wind particle is given by

$$F = F_{\mathrm{wind}} + F_d = F_{\mathrm{wind}} - \alpha_d v.$$

Collisions between wind particles and the turbine's blades are detected by ODE. ODE calculates forces in response to collisions between the blades and the wind particles. We don't allow collisions between wind particles as this is computationally intractable. The wind particles are only allowed to move within the volume of a virtual cylinder placed around the turbine. If a wind particle leaves this cylinder it is again placed at a random location in front of the turbine. Fitness of an individual is defined as the average rotational energy of the rotor:

$$\mathrm{fitness} = \frac{1}{\mathrm{steps}} \sum_{i=1}^{\mathrm{steps}} \frac{1}{2} \omega_i I \omega_i$$

where steps is the number of steps the simulation was done, I is the inertia matrix of the turbine's rotor, and ω_i is the rotational velocity of the rotor at time step i.

5 Experiments

Experiments were carried out with the following parameters. The rotor was placed at a height of 15 units above the ground plane. Maximum wind radius was set to 10 units. Wind particles were created 6 units in front of the rotor. Blades whose bounding box extends outside the virtual cylinder receive a fitness of zero. Blades which are thicker than 4 units also receive a fitness of zero. Spheres with radius 0.1 units were used as wind particles. The number of wind particles was set to 100. Each particle has a starting velocity $v_0 = 1.5$. Wind force F_{wind} was set to 0.03 units and α_d, which defines the amount of drag, was set to 0.02. The simulation was carried out for 3000 steps.

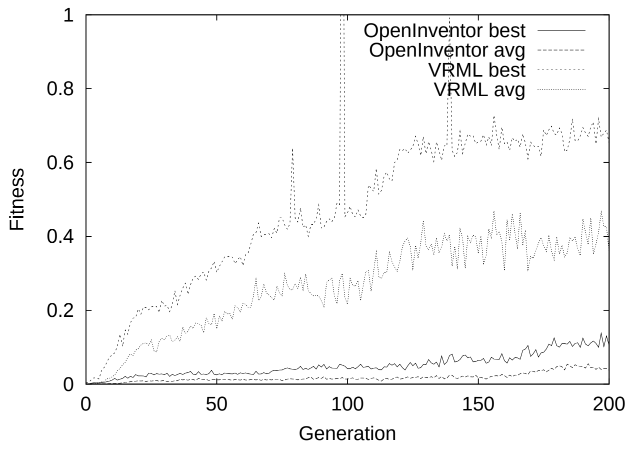

Fig. 5. Fitness statistics are shown for two experiments with a population size of 50 individuals. One experiment used the OpenInventor scene graph to represent the blades of a wind turbine. The other experiment used the VRML scene graph. The peak at generation 98 (maximum fitness of 1.957) resulted from a wind particle being caught between the blade and the base of the wind turbine. This led to a large rotational velocity of the rotor as the wind particle was freed.

A population size of 50 individuals with tournament selection of size 7 was used. The best individual was always copied into the next generation. The population was initialized using ramped half-and-half initialization with maximum depths between 2 and 6. Mutation and crossover were applied with a probability of 50% each. Figure 5 shows the results for both representations. Two evolved wind turbine designs are shown in Figure 6. The wind turbine on the left was evolved using the OpenInventor representation, the wind turbine on the right was evolved using the VRML representation. The wind turbine which was evolved using the VRML representation has a much higher fitness (0.652) in comparison to the turbine which was evolved using the OpenInventor representation (0.105).

We then performed additional experiments to see if the VRML representation is significantly better than the OpenInventor representation. To exclude the possibility that a particle was caught between the blade and the base of the wind turbine we increased the distance between the base and the rotor. Due to the amount of time required to evaluate a single run we were only able to perform 10 runs for the OpenInventor representation and 10 runs for the VRML representation. A single run takes between one and three days to complete. For each setup we compared the maximum fitness achieved after 200 generations with a population size of 50 individuals. A t-test was applied to analyze the results. The results with the VRML representation were significantly better than the OpenInventor representation (t=2.413).

Each subtree of the VRML scene graph is a substructure of the blade which is encapsulated and can be exchanged using the crossover operation. With the OpenInventor representation the entire design may be disrupted by a single mutation if group nodes are used instead of separator nodes. A single mutation can have a large impact because transformations may influence other subtrees.

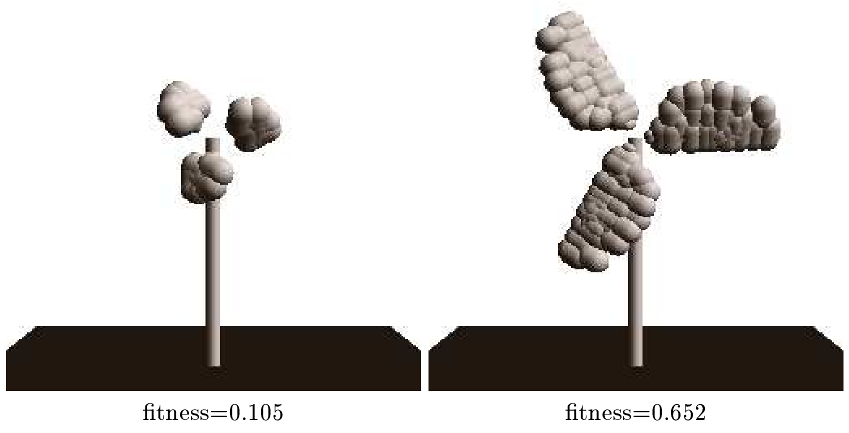

Fig. 6. Two evolved designs for the blades of a wind turbine. The one on the left was evolved using the OpenInventor scene graph. The one on the right was evolved using the VRML scene graph.

We also experimented with the use of an evolution strategy [14, 15] to adapt the values of the internal parameters. The evolution strategy was nested inside a genetic programming loop. Rechenberg [14] describes a method called structure evolution, where continuous parameters are adapted in an inner loop and discrete parameters are adapted in an outer loop. The same approach was followed here. Genetic programming is used to adapt the topology of the objects and an evolution strategy is used to make small changes to the structure. The genetic programming loop was executed every n-th time step. We denote this type of algorithm as $\mathrm{GP}_n/\mathrm{ES}$.

Evolution strategies use Gaussian mutations to change the variables. Let $[v_1, ..., v_n]$ be n real valued variables. This vector is mutated by adding a vector of Gaussian mutations. Each variable v_i has an additional parameter δ_i which specifies the standard deviation which will be used to mutate the variable. The standard deviations are adapted by multiplying the standard deviations with $e^{N(0,\tau)}$ where $N(0, \tau)$ is a Gaussian distributed random number with standard deviation τ. For our experiments we have set $\tau = 0.05$. This process automatically adapts the step size of the mutation. Step sizes which lead to large improvements are propagated into the next generation.

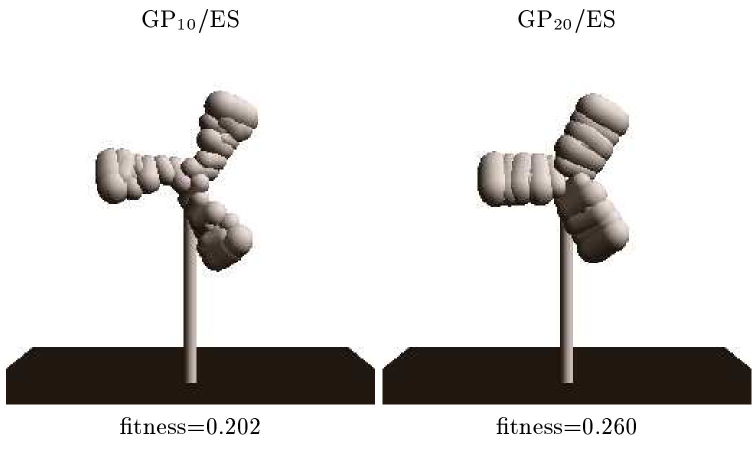
$\mathrm{GP}_{10}/\mathrm{ES}$
$\mathrm{GP}_{20}/\mathrm{ES}$
fitness=0.202
fitness=0.260

Fig. 7. Evolved wind turbines using a mixed GP/ES algorithm for 200 generations.

Figure 7 shows the results for wind turbines which were evolved using the nested GP/ES algorithm. The wind turbine on the left was evolved using a $\mathrm{GP}_{10}/\mathrm{ES}$-algorithm, the wind turbine on the right was evolved using a $\mathrm{GP}_{20}/\mathrm{ES}$-algorithm. We again performed 10 runs for each setup. The full set of statistics is shown in Table 2. The results of different setups were compared using a t-test. Table 3 shows if differences were significant at $\alpha = 0.05$.

Figure 8 shows the maximum fitness for all ten runs for each setup. A possible explanation for these results is that bigger improvements could be achieved by varying the topology of the shape as opposed to fine-tuning the shape. If we apply an ES-mutation, all internal parameters are mutated at the same time. This causes the overall shape to change a little from one generation to the next. However, if we apply a GP-mutation, the change is local to a subtree. A part of the structure may be added or removed. Larger fitness gains could be achieved by locally changing the shape as opposed to varying the entire structure.

Table 2. Average and standard deviation of maximum fitness achieved. Only 10 runs could be performed for each algorithm due to the amount of time required to complete a single run

Algorithm	μ	σ
IV GP	0.146944	0.051362
IV GP_{10}/ES	0.130383	0.083100
IV GP_{20}/ES	0.162535	0.129536
VRML GP	0.341237	0.249383
VRML GP_{10}/ES	0.205339	0.148970
VRML GP_{20}/ES	0.269060	0.200728

Table 3. Comparison of results using a t-test

Algorithm 1	Algorithm 2	$\|T\|$	significance at $\alpha = 0.05$
IV GP	IV GP_{20}/ES	0.536067	not significant
IV GP	IV GP_{10}/ES	0.353820	not significant
IV GP_{20}/ES	IV GP_{10}/ES	0.660644	not significant
VRML GP	VRML GP_{20}/ES	1.479385	not significant
VRML GP	VRML GP_{10}/ES	0.712965	not significant
VRML GP_{20}/ES	VRML GP_{10}/ES	0.806116	not significant
IV GP	VRML GP	2.413064	significant
IV GP_{20}/ES	VRML GP_{20}/ES	1.389561	not significant
IV GP_{10}/ES	VRML GP_{10}/ES	1.410075	not significant

Additional experiments were made with a population size of 100 individuals. Figure 9 shows evolved wind turbines after 100 generations. The first wind turbine was evolved using the OpenInventor representation. The second, third, and fourth wind turbines were evolved using the VRML representation. The second wind turbine was evolved using the simple GP algorithm with fixed internal variables. The third wind turbine was evolved using a GP_{10}/ES-algorithm. The fourth wind turbine was evolved using a GP_{20}/ES-algorithm.

6 Conclusion

Two different scene graph representations for evolutionary design were analyzed. OpenInventor's scene graph is more volatile in comparison to a VRML scene graph. If group nodes are used, a single mutation may completely change the overall shape of the object. Subtrees of a VRML representation are automatically encapsulated and may be extracted and placed at other locations of the structure. Floating point variables which specify three-dimensional vectors, rotation axis or rotation angles are stored inside the nodes of the tree representation. These variables are initialized with random values from a specific range. After initialization, they remain unchanged for the life of the node. Significantly bet-

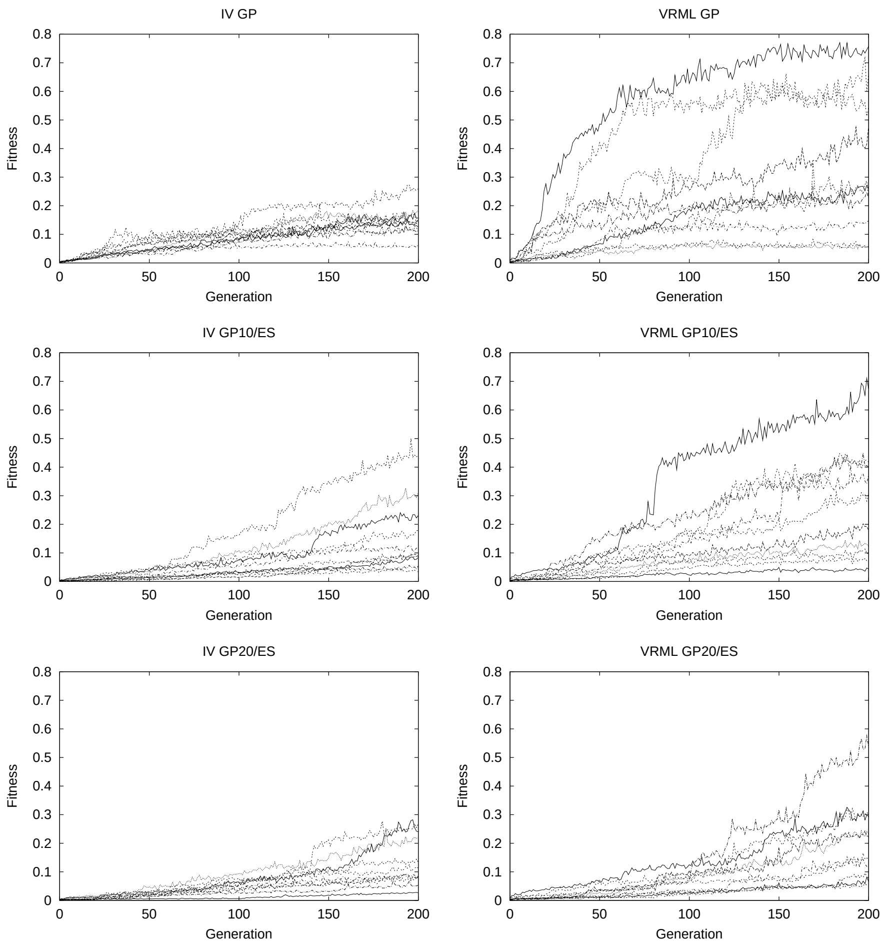

Fig. 8. Fitness of best individual. Ten runs were performed for each setup.

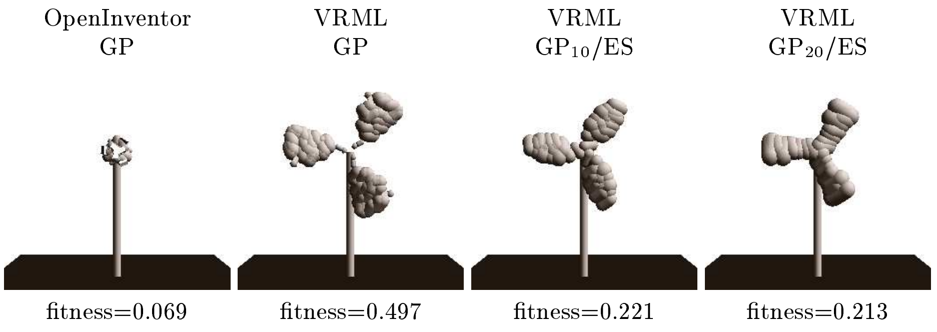

Fig. 9. Results for a population size of 100 individuals after 100 generations.

ter results were achieved with the VRML representation in comparison to the OpenInventor representation if fixed internal variables were used.

We also experimented with a mixed GP/ES algorithm where the values of the internal variables are evolved using an evolution strategy. The overall topology of the shape was evolved using genetic programming. However, this approach did not lead to significantly better results.

References

1. A. L. Ames, D. R. Nadeau, and J. L. Moreland. *VRML 2.0 Sourcebook.* John Wiley & Sons, Inc., NY, 2nd edition, 1997.
2. American Wind Energy Association. The most frequently asked questions about wind energy, 2002.
3. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming - An Introduction: On The Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann Publishers, San Francisco, CA, 1998.
4. P. J. Bentley, ed. *Evolutionary Design by Computers.* Morgan Kaufmann Publishers, May 1999.
5. P. J. Bentley and D. W. Corne, eds. *Creative Evolutionary Systems.* Morgan Kaufmann Publishers, Jan. 2001.
6. P. J. Bentley. *Generic Evolutionary Design of Solid Objects using a Genetic Algorithm.* PhD thesis, Division of Computing and Control Systems, School of Engineering, The University of Huddersfield, 1996.
7. T. Broughton, A. Tan, and P. S. Coates. The use of genetic programming in exploring 3D design worlds. In R. Junge, ed., *CAAD futures 1997. Proc. of the 7th Int. Conf. on Computer Aides Architectural Design Futures, Munich, Germany, 4-6 Aug.*, pp. 885–915, Dordrecht, 1997. Kluwer Academic Publishers.
8. P. Coates, T. Broughton, and H. Jackson. Exploring three-dimensional design worlds using Lindenmeyer systems and genetic programming. In P. J. Bentley, ed., *Evolutionary Design by Computers*, pp. 323–341. Morgan Kaufmann, 1999.
9. G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proc. of the 2001 Congress on Evolutionary Computation, COEX, Seoul, Korea*, pp. 600–607. IEEE Press, 2001.
10. J. R. Koza. *Genetic Programming. On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, MA, 1992.
11. J. R. Koza. *Genetic Programming II. Automatic Discovery of Reusable Programs.* The MIT Press, Cambridge, MA, 1994.
12. J.R. Koza, F.H. Bennett III, D. Andre, and M.A. Keane. *Genetic Programming III. Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, 1999.
13. S. Kumar and P. Bentley. The ABCs of evolutionary design: Investigating the evolvability of embryogenies for morphogenesis. In *GECCO-99 Late Breaking Papers*, pp. 164–170, 1999.
14. I. Rechenberg. *Evolutionsstrategie '94.* frommann-holzboog, Stuttgart, 1994.
15. H.-P. Schwefel. *Evolution and Optimum Seeking.* John Wiley & Sons, NY, 1995.
16. R. Smith. *Open Dynamics Engine v0.03 User Guide*, Dec. 2001.
17. A. Watt. *3D Computer Graphics.* Addison-Wesley, Harlow, England, 2000.
18. J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2.* Addison-Wesley, Reading, MA, 1994.