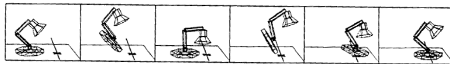
Evolving Motion

Scott H. Douglas[1], Chris R. Klaiber[2], and Shawn "Schmitty" Smith[3]
February 23, 2005
RIT
Rochester NY



1. Introduction

This paper addresses the animator's dilemma of the automation versus control trade-off. That is, the animator wishes to retain control over the final work, while also automating as much as possible to reduce the amount of time it takes to produce the final product. In animation, a degree of freedom is any value that an animator has to set. This paper attempts to provide a better answer to the the problem by evolving motion for a number of degrees of freedom.

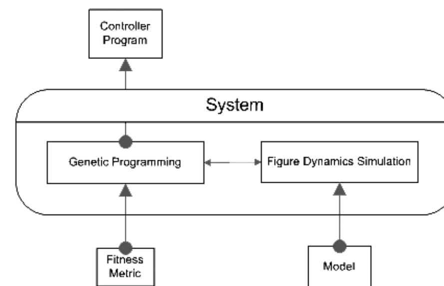
2. Functional Description

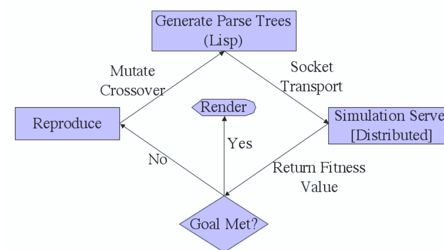
This paper uses a Genetic Programming (GP) approach to the animator's dilemma. It is based on the use of Genetic Algorithms for problem solving. In a Genetic Algorithm (GA), there is a population consisting of many individual bit strings, or permutations. In GP, the population is instead filled with program code. These programs are expression trees representing functions and their arguments, in a nested fashion. This paper introduces a method of applying the genetic operators present in a GA to the control programs in the population of a GP to create articulated motion.

3. Software Design

3.1. System Overview

A genetic programming system similar to that described by Gritz and Hahn[1] was implemented. The system used in Klaiber[2] was used as a basis, and the necessary modifications were made to support genetic programming. Since individuals are permutations of expression trees, Lisp proved an ideal language for the GA code. This allowed individuals to be stored and manipulated as if if they were Lisp data. For fitness evaluation, efficiency and compatibility with graphics libraries was a must. To this end, C++ was used for the graphics simulation code. The GP code interfaces with the simulation code via TCP/IP and standard network sockets. Although it was not implemented, it would be possible to convert the system to a distributed application that performs fitness evaluations simultaneously by connecting to multiple instances of the simulation server.





Because the final product is an animation, the system is not required to produce a result in real time. Therefore, use of a GA for this purpose is feasible.

3.2. Default Parameters

- Cap – 2000
- Mutation Rate – 0.0
- Population Size – 100
- Tournament Size – 2

[1]Send correspondence to shd0326@cs.rit.edu.
[2]Send correspondence to cxk1970@cs.rit.edu.
[3]Send correspondence to schmitty@csh.rit.edu.

3.3. Individual Representation

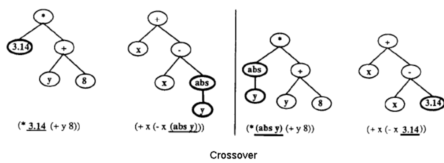
Individuals are Lisp expression trees. That is, they are of the form (function argument ...), where function is one of the functions +, -, $\times$, ifltz, or %. The function ifltz is a branching function and takes three arguments, returning the value of the second or third depending on whether the first evaluates to true or false, respectively. The function % represents safe division, such that when the denominator is zero, a 1 is returned as the value of the function. All functions take either two or three arguments, each of which may be a terminal or another function sub tree.

Terminals consist of a number of variables, and random constants between 0 and 360. Each variable is given the value of the angle of a bone in the system at evaluation time.

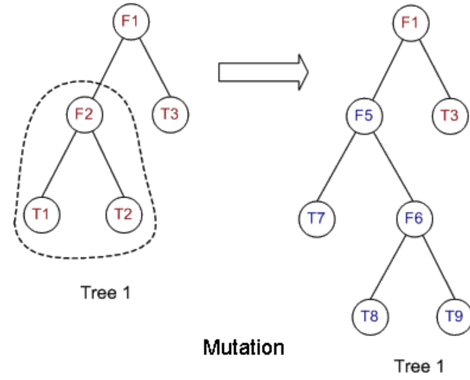
3.4. GP System

The GP implementation is straightforward. Mutation occurs by picking a randomly selected sub tree, and building a random new sub tree at that location. The crossover method takes two parents and randomly selects and swaps a sub tree on each of them to create two children.

This implementation suits itself well to Lisp, the expression trees were trivial to create, and treated by Lisp as Lisp code or data. CMUCL was used because it is an efficient and practical Lisp implementation. The resulting trees were also easily parsable by the simulation server. Because of the client/server architecture, the Lisp server can be paused during execution, and allows for querying or modifying of variables in the running system. This allows the user to look at the current best parse tree, its fitness, or how many fitness evaluations have occurred. Parameters, such as the address of the simulation server, can also be altered in mid-run. Once the Lisp client is resumed, it proceeds from where it left off.



Crossover

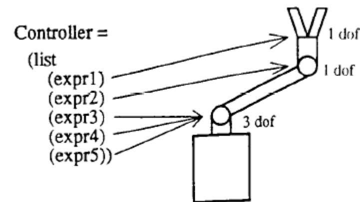


Mutation

The parse trees consist of terminals, variables, or functions. A terminal is a constant value. A variable node contains a value that will be replaced at run time based on the current state of the system. Currently, only joint angle may be used for this value. Functions are parse trees that evaluate their parameters. For more curvy motion, other cyclic operators could have been added, such as sine or cosine.

3.5. Simulation

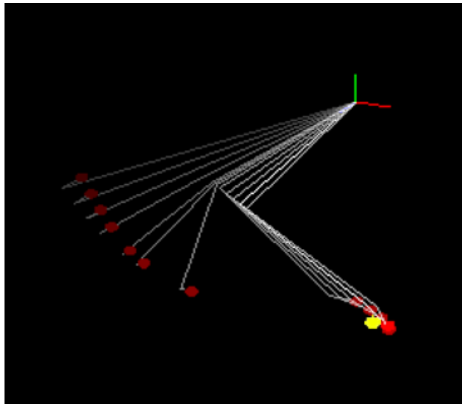
The simulation server was created in C++ because of speed and easy access to the OpenGL API. It waits for a connection on port $30,002$. Once it receives a connection, it begins parsing until it reaches the end of a parse tree. The parse tree is then parsed into a tree structure of nodes. All the nodes implement an interface. The interface allows all nodes to be evaluated to a float. To this end, the crossover and mutation could easily be performed on this structure because any tree can be replaced by any other tree. The simulation then begins. The bones are updated based on constrained dynamics. Each bone has up to three degrees of freedom to be updated. Each bone structure has a set of parse trees associated with it called a controller program.

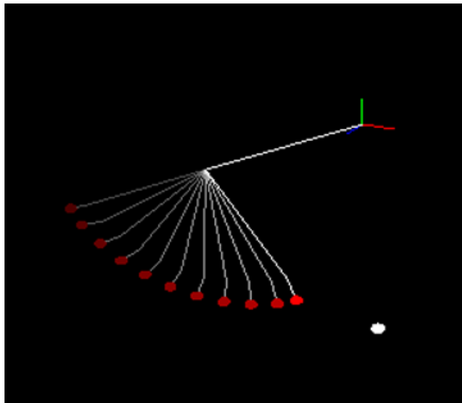


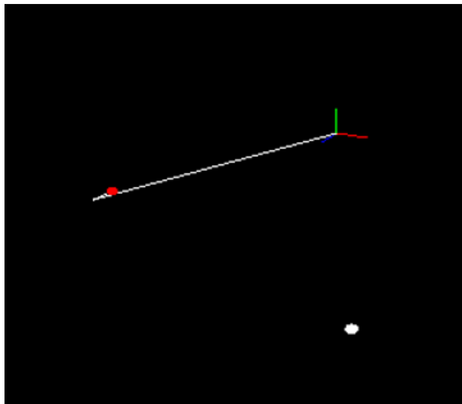
The controller program is evaluated at each time step, and the result number is used as a force applied to the bones at that time step. All matrix transformations are handled using the OpenGL API. This makes modeling hierarchical structures fairly easy.

The bone system used for our project consisted of three bones modeled after a human arm. Constraints on the range of motion for the bones was also modeled after human arm constraints.

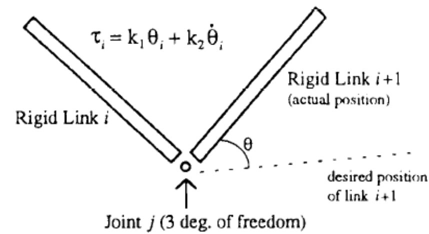
The bone structure's goal is to move from its initial configuration into a configuration where the hand of the arm is touching a goal point. An example bone structure and some evolved motions are included below.







In the Gritz and Hahn paper, joints were implemented as rigid links. The controller program provided a desired position for the rotation of the joint. The difference between the desired and the actual position created torque on the rigid link. They used a dampening spring to affect the motion.



In our simulation we decided to use the controller program output as a force to be applied to the bone at that time step.

3.6. Code Organization

A single Lisp source file contains all the code required for the GP system. To begin the application, run (ga (make-gaparam)) in an ANSI-compatible Lisp. The Trivial Sockets[1] package by Edi Weitz must be loaded, since it is required for network communication.

The simulation code is split into a number of classes. A node interface defines the methods needed to be an operator node in the simulation's representation of the parse tree. Each operator implements the interface and uses the convention OperatorNameNode. The Server class deals with reading a parse tree from a socket, and later writing a fitness back over the socket. The Parser classes parse the data received into a tree of Node pointers. The control program class is a data structure that holds multiple parse trees that map to the different degrees of freedom of the object. The DOF class contains a minimum, maximum, actual, and desired value. The desired value isn't used right now, but would be if our implementation of joint movement mimicked the implementation in the Gritz and Hahn paper more closely. A Bone class defines each bone and contains data about how it can rotate. The sim class contains the program entry point and performs the logistics of getting the simulation to work. The simrun class is used for giving a quick idea of how a control program will look. This is run in realtime. The simvid class runs the simulation, and writes out the video frames to raw files. A Perl script, process.pl, converts the raw files to jpegs and assembles them into a movie file.

4. Testing

Testing was done in two parts, one for each of the two portions of the system. For the GP portion, testing was trivial, since previous testing proved that the framework worked properly, the remaining work was to test the new additions. The additions were the networked fitness function, the individual creation routines, and the crossover and mutation

[1] `http://www.cliki.net/trivial-sockets`

routines. The fitness function was run standalone in an interpreter. Contrived individuals were sent to the simulation server, and the returned values checked by hand. Creation was checked by creating a number of individuals and verifying that they contained the proper amount of joints and were in the proper format to be interpreted by the simulation. Each function required the proper amount of arguments. Crossover and mutation were checked by hand to verify that the number of joints and function arguments were preserved.

The simulation code was verified in two chief ways. First, a number of angles were checked to verify that the arm would appear much like a human arm. Also, the point to reach was verified to be within the reach of the arm. The parsing code was checked by example, reading in a sample tree, and then printing out the in-memory structure it was parsed into. The network server code was verified by using telnet to connect and send data.

5. Experimentation

The mutation rate and population size of the genetic programming algorithm were varied to see what effect they had on the number of fitness evaluations required to find a solution. Since fitness evaluations consume the largest amount of processor time, they are a good candidate for optimization. Figure 2 and Figure 3 vary the mutation rate over different population sizes. Figure 1 varies the population size. All three graphs are based on the default parameters. For each setting, the genetic program was run five times. The average was recorded and used to plot a data point. As the mutation rate was varied, the resulting number of fitness evaluations also varied greatly. There was no clear best setting for a mutation rate. Some researchers argue that mutation is entirely unnecessary, as crossover is already destructive enough. As the population size was varied, a sweet spot was discovered at around population size thirty.

Figure 1 varies the population size to see what effect it has on the number of fitness evaluations required to solve the problem. The best population sizes were 30, 50, and 80.

Figure 2 varies the mutation rate to see how it affects the number of fitness evaluations required when using a population of size 100. The sweet spots for mutation rate were 0.004 and 0.026. Figure 3 varies the mutation rate to see how it affects the number of fitness evaluations required when using a population of size 50. The sweet spot for this graph was at 0.012.

Although the system is effective at achieving results, optimizing the system is almost impossible. In most of the graphs shown, there is no clear winner for mutation rate or population size. No matter how the independent variable is altered, the results have a high variance. Pinpointing a good guess of the best variable settings would require a larger number of runs of the system, which isn't feasible. Since the animator will want to create different animations, the system will constantly change. The system works, but since it has such a dynamic nature, it is hard to find concrete optimal settings. To find truly optimal parameters, we would need a genetic algorithm to evolve parameters to this genetic algorithm. Basically, art is still art. We haven't converted it completely to a science.

Figure 4 shows the successive fitness evaluations of a run that did not converge completely within the fitness evaluation cap, but that clearly could have converged over time. By either increasing the mutation rate or increasing the cap for fitness evaluations, a solution may have been reached.

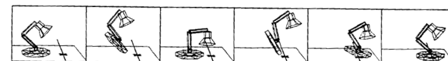
Figure 5 shows run that converged just after $1,600$ fitness evaluations. The bulk of the points plotted have a clear downward slope. Figure 6 shows a run that converged just after $1,000$ fitness evaluations. The graph shows a slight shift downward of points plotted, but it is not as clear in this graph as in most graphs.

Figure 7 shows a run that did not converge completely within the fitness evaluation cap. This run slightly converges over time, but never finds the solution. The minimum plots between fitness evaluations 600 and $2,000$ are all similar. Increased mutation would probably help a solution to converge.

Almost all of the plots showed strong bands around 90, 50, and 5. The 90 band consists of parse trees that get devastated during crossover and mutation. The band at 50 is most likely due to the specific problem set we are working with. There are probably multiple bone paths that lead to having the same fitness. This would account for the bunching. There is also a weaker trend that emerges at around 40 that can be explained the same way. The 5 band consists of parse trees that are converging towards the goal. If there were no bands, then the genetic programming would be doing random search.

6. Previous Results

Larry Gritz and J.K. Hahn were successfully able to evolve figure motion. In one example, the evolved motion for Luxo to jump to a desired point.



The resulting controller program was fairly simple. The controller programs evolved by our system are usually much more complex.

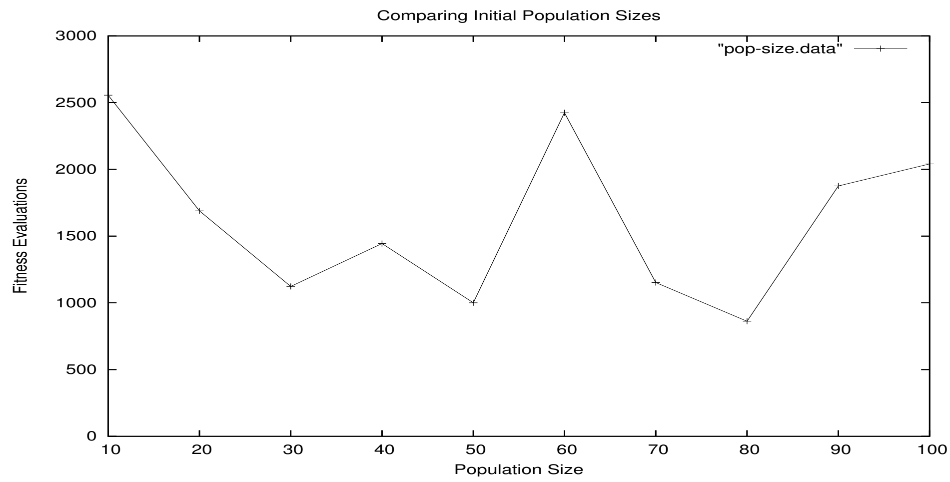


Figure 1: *The population size is varied to observe its effect on the number of fitness evaluations required to find a solution.*

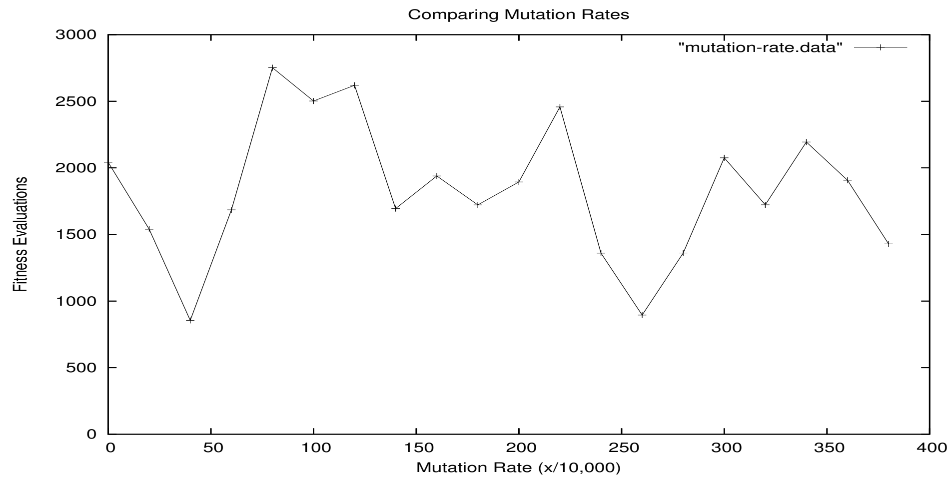


Figure 2: *The mutation rate is varied to observe its effect on the number of fitness evaluations required to find a solution. Here the population size was set at 100, and the default parameters are used.*

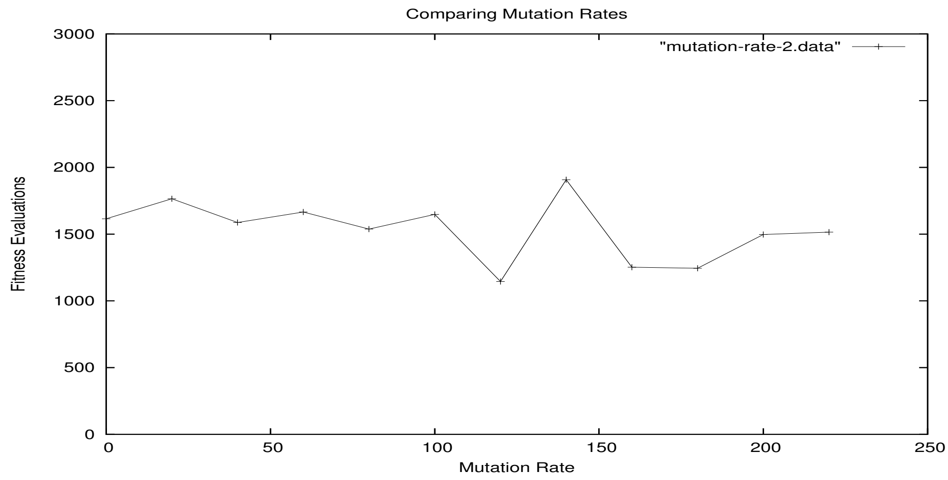


Figure 3: *The mutation rate is varied to observe its effect on the number of fitness evaluations required to find a solution. Here the population size was set at 30, otherwise the default parameters are used.*

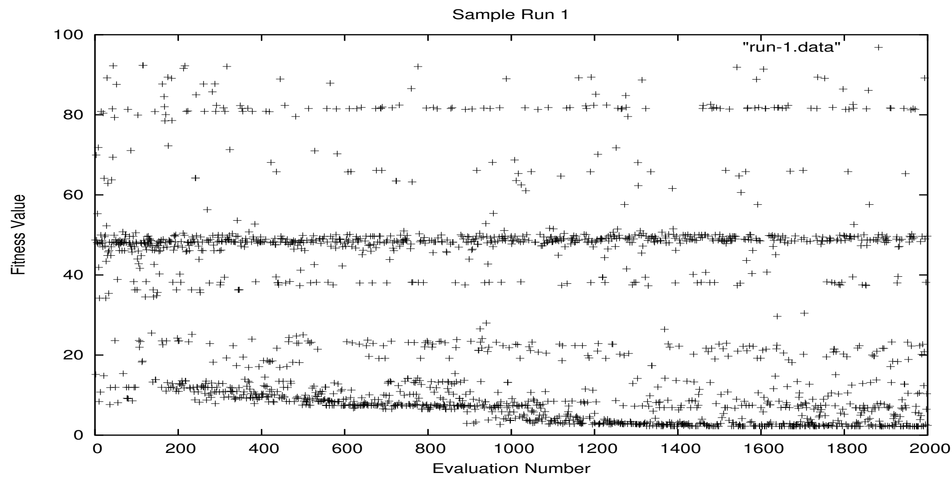


Figure 4: *The fitness value of every fitness evaluation performed for an individual that did not converge within the maximum number of allowed evaluations.*

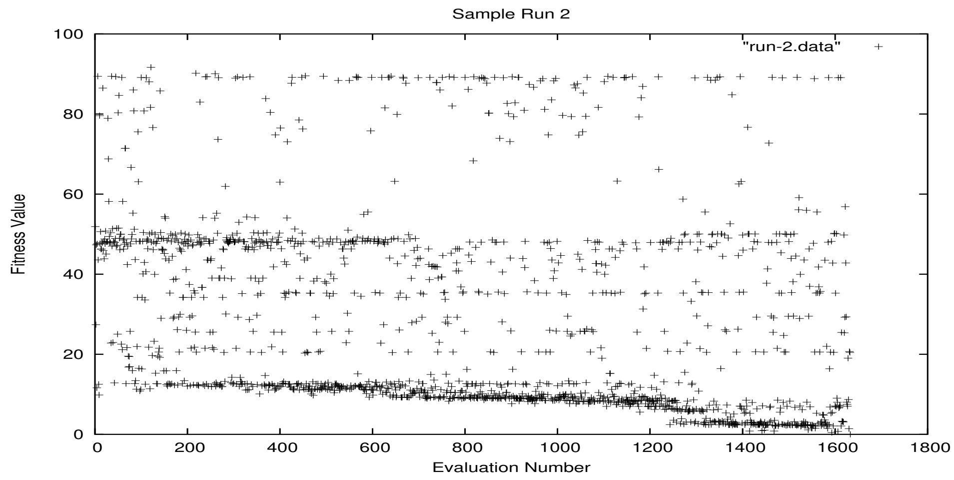


Figure 5: *The fitness value of every fitness evaluation performed for an individual that required approximately 1,600 evaluations to converge.*

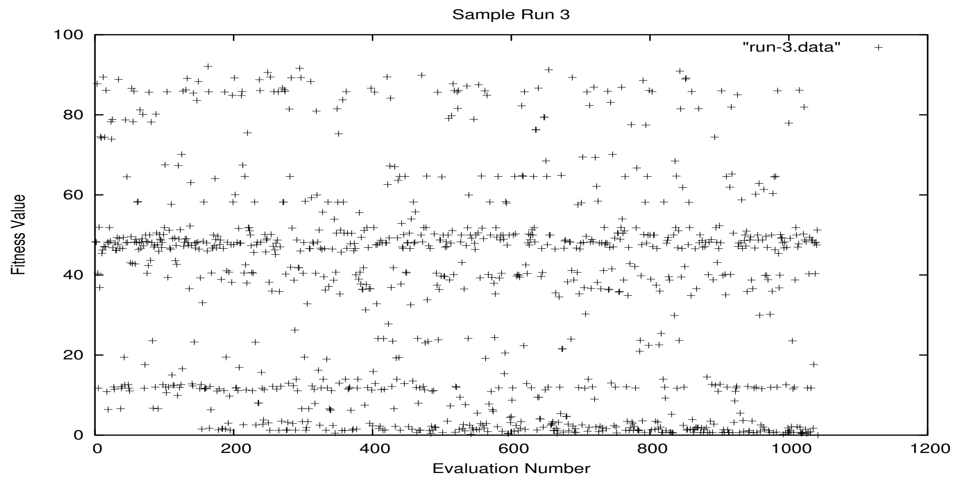


Figure 6: *The fitness value of every fitness evaluation performed for an individual that required approximately 1,000 evaluations to converge.*

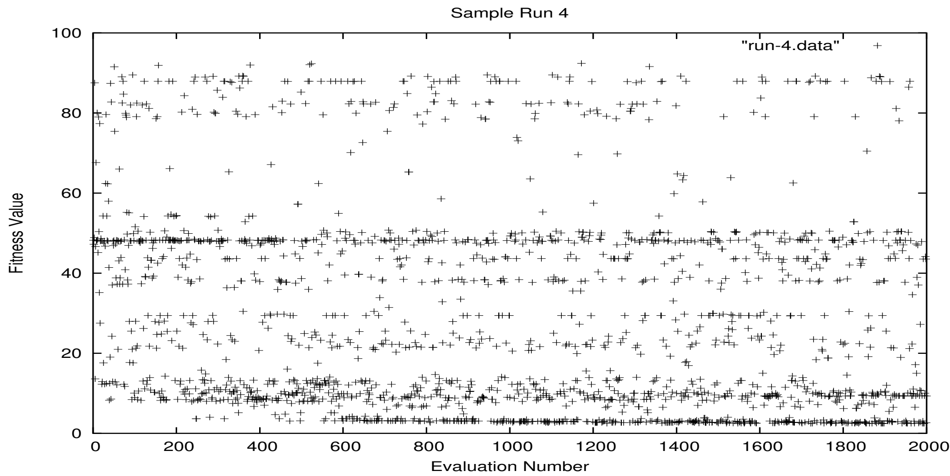
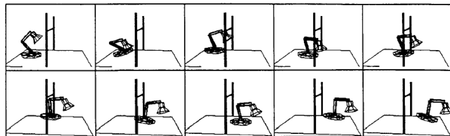


Figure 7: *The fitness value of every fitness evaluation performed for an individual that did not converge within the maximum number of allowed evaluations.*

```
(list
  (- (ifltz a0 pz a2) (% a1 t))
  (ifltz (- (- (ifltz a0 pz a2) (% a1 t)) (ifltz (+ a2 15.4963) (ifltz a0 pz a2) (% a1 a2))) (ifltz (% (- s0 vx) (+ a2 s0)) (% (- s0
     vx) (+ a2 s0)) (+ (- a1 a1) (- a1 t))) (% (+ a1 vz) (ifltz (* pz a1) (% a0 pz) (% px a0))))
  (- (ifltz s0 (ifltz (- (% vx a0) (% vx vx)) (- (% vx a0) (- (+ -28.4382 t) (% a1 t))) (* 16.5266 s0)) vz) (+ a1 vz)))
```

To further refine the motion that is evolved, Gritz and Hahn introduced the concept of style points. After some basic motion is evolved, the system applied a negative fitness for the figure disobeying style guidelines, such as colliding with objects. Using this method, they were able to get Luxo to limbo underneath a bar without touching it.



Following the precedence set by the Gritz-Hahn paper, this project has produced the foundation required to implement a fully articulated figure. Future enhancements will recreate the accomplishments of Gritz and Hahn: to create motion for specific tasks, which appear physically plausible and appealing to viewers.

7. Related Work

One area of study in genetic algorithms is interactive evaluation of fitness. Some researchers have used the user as a fitness function to evolve solutions that the user likes best. One example, genetic images, evolves images the user finds aesthetically pleasing.

Other researchers have used genetic algorithms to evolve creatures to perform certain tasks, such as walking, swimming, flying, or jumping. Some researchers go as far as to co-evolve the creature's brain and body. Basically, the brain and body are two independent systems that interact with each other. As the body adapts to performing a task, the brain must evolve to use the new body more efficiently. As the brain evolves, the body must further adapt to take advantage of the new abilities of the brain.

8. Conclusion

Genetic programming has both scientific and artistic applications. While the artistic applications are more abstract and harder to study, they provide a rich depth of experience. Genetic programming is a versatile tool that can be applied to many diverse problem spaces. The system implemented in this paper was successfully able to evolve motion.

The lack of convergence for some runs was perhaps due to the cap on fitness evaluations. However, because each fitness evaluation took a considerable amount of time, such a cap was necessary. The goal of this paper was reached, however, and convincing motion was successfully evolved via genetic programming.

8.1. Future Enhancements

This project successfully implemented the concepts described in the Gritz-Hahn paper. The algorithm to create genetic programs worked smoothly and efficiently. However, like all development, there are aspects of this project which could be enhanced in future implementations.

8.1.1. Figure Motion

A worthwhile addition to this project would be the implementation of figure motion throughout the entirety of a humanoid model. While the initial work provided figure motion in an extremity, such as an arm or leg, it would be interesting to extend the model to an entire skeletal system–thusly opening the possibility of attaching the figure motion to a standardized set of models.

8.1.2. Physics

Another modification of particular worth would be the implementation of a more accurate physics system. By forcing a more strict adherence to physical law the resulting animations will have a more natural appearance. Unfortunately a number of complexities arise from the addition physical constraints.

Another physical consideration is the simulation of muscles. As a means of locomotion, muscles interact with bones, which in turn interact with an object in the world (e.g. a floor). Simulating this effect can be accomplished via springs. By connecting individual 'bones' with a spring the model with be able to properly dampen the motion to give the appearance of natural motion. Upon further investigation it was discovered that the same dampening effect could be recreated without the explicit use of springs. A non-linear interpolation of the motion resulting from torques and forces would provide a more than adequate substitution for the springs.

8.1.3. Toolkit

Additionally, providing simplified control mechanisms to this project would allow for a much more "user-friendly" method of interacting with the animations. Of particular interest would be the ability to specify the degrees of freedom at each joint, crossover type and mutation rates, and a fitness function. Once access to these essential variables have been granted the interface could be integrated into popular computer animation software (e.g. Maya). Moreover, the ability to chain multiple genetically derived motions into a single animation with complex movement would be extremely useful to animators.

8.1.4. More Experimentation

Lastly, a more thorough exploration of the problem space would enhance the robustness of this simulation. Of particular interest would be altering the form and nature of cross-over, assigning the initial command program size, determining and optimizing operator sets, and the implementation of style points.

References

[1] L. Gritz and J. K. Hahn, "Genetic Programming for Articulated Figure Motion," The Journal of Visualization and Computer Animation, Vol. 6, 129-142 (1995)

[2] Klaiber, Christopher R., "Exploring Permutation-based Genetic Algorithms via the N-Queens Problem," Rochester Institute of Technology, New York, 2005.

[3] Ik Soo Lim and Daniel Thalmann, "Solve Customer's Problems: Interactive Evolution for Tinkering with Computer Animation," 404-407 (2000)

[4] Yoon-Sik Shim and Chang-Hun Kim, "Generating Flying Creatures using Body-Brain Co-Evolution," Eurographics/SIGGRAPH Symposium on Computer Animation, 276-285 (2003)

[5] Karl Sims, "Evolving virtual creatures," Proceedings of the 21st annual conference on Computer graphics and interactive techniques, 15-22 (July 1994)

[6] Karl Sims, "Artificial Evolution for Computer Graphics," ACM SIGGRAPH Computer Graphics, Proceedings of the 18th annual conference on Computer graphics and interactive techniques, 319-328 (July 1991)