Procedural Animation with Genetic Algorithms and Physics Simulation

Bruno Feijó   Marco Aurélio Pacheco*   Pedro Luchini

PUC-Rio, Dept. of Computer Science, Brazil
* PUC-Rio, Dept. of Electrical Engineering, Brazil

Abstract

This paper describes a method for automatically animating a virtual character based on the topology of its musculoskeletal system. Plausible and realistic movements can be computed for any kind of character; no assumptions are made about its body structure. Arbitrarily-shaped creatures or robots, thus, learn how to move in the same way real-life animals did: through evolution.

Keywords: artificial intelligence, biologically-inspired computing, procedural animation, genetic algorithms, physics simulation

Authors' contact:
{bruno,pluchini}@inf.puc-rio.br
*marco@ele.puc-rio.br

1. Introduction

Procedurally-generated content has become important in the past years as a means of reducing designer workload and producing games with smaller footprints. Additionally, these techniques are useful when dealing with unpredictable input from the player, such as user-generated content.

This paper proposes a method of animating characters whose bodies are represented by a series of constraints (bones, joints) and force actuators (muscles) driven by physics simulation. The character's "brain," then, will send a sequence of commands to each muscle, causing the body to move. This way, the animation can be modeled as a list of instructions (a program) that is executed by its body.

Clearly, characters with different body structures will need different programs to animate properly. It is possible, nevertheless, to establish very simple and broad goals for a class of animations regardless of the character's body structure, such as "maximize horizontal speed" to describe a running animation.

This approach to character animation has the following characteristics:

- The animation is simple to represent;
- Displaying and evaluating the animation is straightforward;
- Creating the animation itself, however, is not.

These three traits suggest the employment of *genetic algorithms*, an optimization technique well-suited for finding approximate solutions based on heuristics.

2. Related Work

There is an extensive list of publications related to character physics. Particularly worthy of note are [Jakobsen 2001] and [Witkin and Kass 1988] who focus on constraints capable of simulating bones, joints, and other structural elements of a live character.

Genetic algorithms have been used by [Machado and Cardoso 2002] and [Cope 2005] in the field of artificial creativity with remarkable success. They have shown that the pseudo-random nature of G.A.s can create aesthetically-pleasing works in a manner that traditional algorithms cannot, with surprising results that often startle the developers themselves.

3. Modeling the Character

Since we are trying to simulate real-life biologic systems, it makes sense that we use the same components that are found in real-life animals to represent the virtual character: bones, muscles, joints, etc. These components can be divided in two categories: *passive* and *active*.

3.1 Passive Components

Passive components are those that cannot be controlled by the character's brain. Just like we are unable to change our bones' length, the character should not be allowed to change his. Some examples of passive components are:

- **Bones** can be represented by a simple bilateral constraint between two particles, which is easy and fast to implement with Verlet integration [Jakobsen 2001]. If a more robust physics simulation engine is available (and needed), more complex bone structures can be represented by rigid bodies.
- **Joints** keep bones connected to each other. [Baltman and Radeztsky Jr 2004] propose a method for constraining the angle between bones, as well as their spatial orientation.

3.2 Active Components

Active components are those that can be controlled by the brain.

- A **spring** applies linear force when contracted or stretched past its natural length (Figure 1). To achieve locomotion, the brain can send commands to change the spring's natural length (Figure 2).

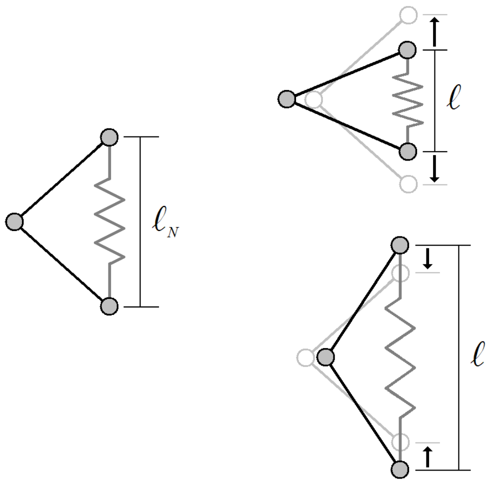


Figure 1: A spring exerts a force whenever it is stretched or compressed. As per Hooke's law, the force is proportional to the spring's deformation (ℓ_N - ℓ) and its force constant (k).

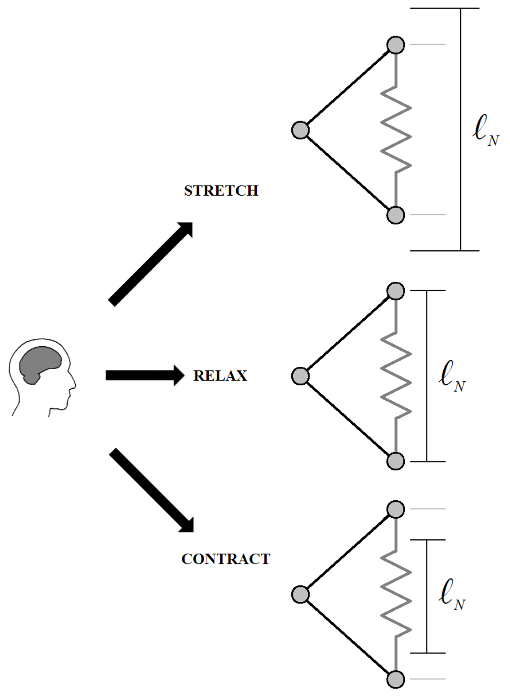


Figure 2: The brain sends commands to the spring, telling it to change its natural length. This pulls the bones together (by decreasing ℓ_N) or pushes them apart (by increasing ℓ_N).

Springs are a rather crude approximation of how muscles really work. However, they are useful as a simulation of abduction/adduction movements (such as pulling the knees together), which would otherwise require very complex modeling.

- A **torsion spring** is the angular equivalent of a spring. It connects to a joint (the junction of two bones) and applies a torque when it is contracted or stretched past its natural angle (Figure 3). The brain can send commands to change the torsion spring's natural angle.

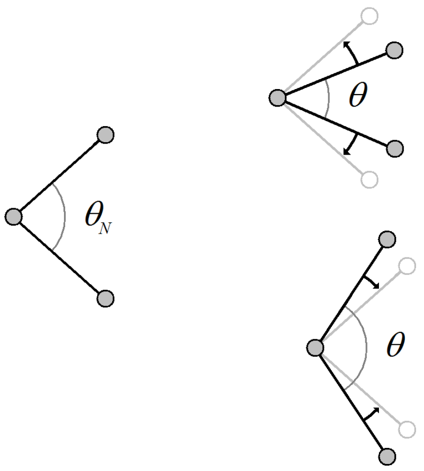


Figure 3: A torsion spring exerts a torque whenever it is stretched or compressed. As per the angular version of Hooke's law, the torque is proportional to the torsion spring's angular deformation (θ_N - θ) and its force constant (k).

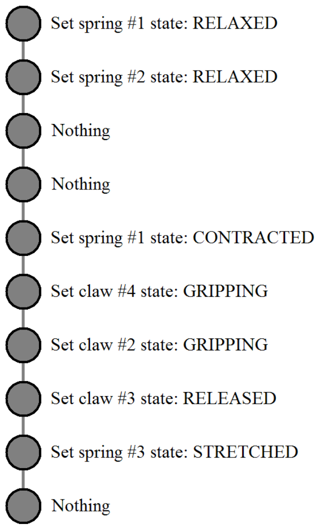
- A useful (if somewhat unrealistic) component to have in a character is a **claw**, that is, an extremity that is capable of gripping walls and floors to anchor the body's movement. When the brain sends a *grip* command to the claw, it will "stick" to whatever surface it is currently touching (and thus become unable to be moved from that spot). When the brain sends the claw a *release* command, it will cease being "sticky." A claw that is in a "sticky" state but is not touching any surface is unaffected.

Springs and torsion springs have a theoretically infinite number of possible states – any real number can be set as their natural length or angle. Our test application simplified it down to only three states: The *relaxed* state will set the natural length to the value it had at the beginning of the simulation, the *contracted* state will set the natural length to half that value, and the *stretched* state will set it to twice that value. It was a somewhat arbitrary choice; depending on the application, this set-up might take a lot of tweaking to achieve good results.

4. Representing the Animation

Because the character moves as a result of the commands issued by its brain, the animation is represented by a *sequence of commands* that are issued one after the other. This is very similar to imperative programming languages (such as assembly, or machine code), so we will refer to these commands as *op-codes*. We can think of the animation as a program that is executed one op-code at a time.

There is no reason to pre-define the size of the program – let the genetic algorithm figure that out on its own –, so the sequence should be allowed to have a variable length. Depending on the animation being generated, it may also be necessary to loop the program – restarting from the first op-code when the end of the program is reached – to simulate an endless cycle (such as a walking animation).



Set spring #1 state: RELAXED
Set spring #2 state: RELAXED
Nothing
Nothing
Set spring #1 state: CONTRACTED
Set claw #4 state: GRIPPING
Set claw #2 state: GRIPPING
Set claw #3 state: RELEASED
Set spring #3 state: STRETCHED
Nothing

Figure 4: A short animation with ten op-codes.

5. Playing Back the Animation

Physics simulation systems are notoriously fickle when it comes to accuracy and consistency. It is often the case that running the same simulation on different machines will result in different outcomes.

Numeric methods (such as those used in real-time physics simulation) are inherently inaccurate, but we can at least ensure they are *consistent* by employing what [Valente et al. 2005] describe as the "fixed-frequency deterministic game loop." This technique is based on the fact that if we always update the game state at a fixed number of times per second (with the same delta-time, or *dt*), then the simulation will consistently produce the same result:

```
accumDT ← 0
function GameTick()
   dt ← time elapsed since last frame
   accumDT ← accumDT + dt
   while accumDT ≥ FIXED_DT do
      GameStateStep(FIXED_DT)
      accumDT ← accumDT - FIXED_DT
   end while
end function
```

To play back the animation, one must define a time interval between op-code executions, and then run the animation program side-by-side with the physics simulation:

```
accumOpDT ← 0
function GameStateStep(dt)
   // Run animation program:
   accumOpDT ← accumOpDT + dt
   while accumOpDT ≥ OPCODE_DT do
      Take next op-code from the program.
      Interpret its command.
      Modify the state of springs/claws.
      accumOpDT ← accumOpDT - OPCODE_DT
   end while

   // Run physics simulation:
   PhysicsStep(dt)
end function
```

The exact values of `FIXED_DT` and `OPCODE_DT` should be adjusted to suit the application. Our test program used `FIXED_DT` = 1/500 and `OPCODE_DT` = 1/200 (in seconds), which ensured a high degree of accuracy in the physics simulation.

6. Creating the Animation

Now that the animation can be represented, stored, and played back, all that is left to do is understand how it is created. This is where genetic algorithms come in.

6.1 An Introduction to Genetic Algorithms

A genetic algorithm is a search technique used to find optimal (or approximate) solutions to a problem based on a heuristic measure of the solution's quality.

Each solution is represented as an individual in a population of candidate solutions. Each individual holds a *chromosome*, a representation of its solution; the chromosome can be evaluated by an *objective function* to determine its *fitness* (a real number).

At first, a population of individuals is created with random chromosomes, and each of them is evaluated by the objective function to discover its fitness. Then, some individuals of the population are selected to be modified and included in a new population of individuals, which forms the next *generation* of solutions. These steps are repeated until either a pre-set number of generations have been evolved, or a desired fitness score has been reached.

The fitness of an individual determines the probability that it will be selected for modification and inclusion in the next generation. Thus, individuals with high fitness are more likely have their genes included in the final solution.

Individuals can be modified by either *crossover* or *mutation*.

- **Crossover** is an operation where the chromosomes from two individuals (parents) are combined. This results in two new individuals (children) with genetic material that is related, but not identical, to the originals'. The children then replace their parents in the new population.
- **Mutation** is an operation where a single individual has its chromosome modified in some random manner.

Crossover is typically set to occur with a high probability, usually 80% or more. Mutation, on the other hand, has a deleterious effect on the evolution if it happens too frequently; it is usually set to occur with a low probability, rarely exceeding 10%.

There is an entire field of research dedicated to genetic algorithms, with several existing techniques to improve their performance. A detailed look into that field is beyond the scope of this paper; for a more thorough introduction we recommend [Davis 1991].

6.2 The Animation Defined in G.A. Terms

The solution we are trying to optimize is the animation program. In our G.A., then, the chromosome is represented by a sequence of op-codes. Each individual in the population can be evaluated by playing back the animation; the fitness score depends on what type of animation is being sought. Some examples:

- **Walking cycle:** Run the animation program for a few seconds, looping it every time it reaches the last op-code. The fitness score is the total horizontal displacement of the character.
- **Jumping animation:** Run the animation program for a few seconds, without looping it. The fitness score is the maximum height achieved by the character.

The crossover modifier of choice is the *one-point crossover* (see Figure 5). It is useful because it recombines the two animations without completely destroying the parents' sequencing of op-codes.

Mutation modifiers can be as random as the application requires. Here are some suggestions:

- **Constructive mutator:** Inserts random op-codes at random points in the chromosome.
- **Destructive mutator:** Removes op-codes from random points in the chromosome.
- **Replacing mutator:** Takes an op-code at a random point and replaces it with another random op-code.
- **Swapping mutator:** Takes two op-codes at random points and swaps their positions.

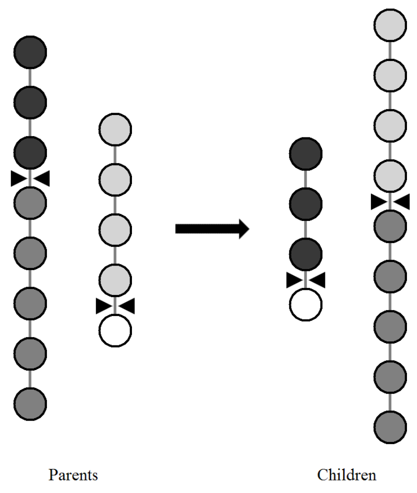


Figure 5: One-point crossover. Each of the parent chromosomes is "cut" at a random point to create two sub-sequences. By swapping the sub-sequences, two new chromosomes are created.

6.3 Encoding Op-Codes in the Chromosome

An op-code can be represented by something as simple as a single integer number. Let us imagine a character whose body is composed of 2 springs and 3 claws. The op-codes can be represented thusly:

1. Set spring #1 state: CONTRACTED
2. Set spring #1 state: RELAXED
3. Set spring #1 state: STRETCHED
4. Set spring #2 state: CONTRACTED
5. Set spring #2 state: RELAXED
6. Set spring #2 state: STRETCHED
7. Set claw #1 state: GRIPPING
8. Set claw #1 state: RELEASED
9. Set claw #2 state: GRIPPING
10. Set claw #2 state: RELEASED
11. Set claw #3 state: GRIPPING
12. Set claw #3 state: RELEASED

That is, there are twelve valid op-codes for that character. Generalizing for a character with N_S springs and N_C claws, there will be $3N_S + 2N_C$ valid op-codes. (Naturally, this quantity will be different if the character includes other active components, such as torsion springs.) Any integer number greater than the number of valid op-codes can be interpreted as a "nothing" op-code.

"Nothing" op-codes are extremely important during the evolution of an animation program. A sequence of op-codes that is excessively dense with commands will cause the character to thrash its limbs in every direction. At worst, this will prevent the character from actually moving; at best, the result will be unsightly.

We suggest that one third of the op-codes in the programs be "nothing" op-codes; this ratio produced the best results in our test application. Every time a random op-code had to be created (in the initialization of the first generation, in the constructive mutator, and in the replacing mutator), our application called the following function:

```
function MakeRandomOpCode()
   return 1.5 * (3*$N_S$ + 2*$N_C$)
end function
```

7. Conclusion

We have shown that animations can be represented by a sequence of commands sent from a character's brain to its body. There aren't any traditional algorithms capable of deriving this sequence of commands, so we suggest the use of a genetic algorithm instead. As is the hallmark of artificial intelligence, the results are somewhat unpredictable, startling, and require significant tweaking before they look "just right." On the other hand, this approach is extremely versatile, being capable of animating any kind of character with very little input from a human operator.

One clear disadvantage of the method proposed is that the generated animations will only apply to the environment where they were evolved – a walking animation that evolved on a flat floor will be useless if the character is confronted with a staircase. As a direction for future work, the authors propose incorporating *senses* into the model, turning it into an implementation of the genetic programming paradigm [Koza 1992]. Specifically, *touch* (whether a part of the character is touching a surface) and *proprioception* (the angles of the character's joints) are simple to implement and can be used as inputs for the animation program.

Acknowledgements

The authors would like to thank Waldemar Celes for his assistance with implementing the physics simulation system we used in the test application.

The software for this work used the GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology.

References

JAKOBSEN, T., 2001. Advanced Character Physics. *Proceedings of the 2001 Game Developers Conference.*

WITKIN, A. AND KASS, M., 1988. Spacetime constraints. *Proceedings of the 15th International Conference on Computer Graphics and Interactive Techniques*, 159-168.

MACHADO, P. AND CARDOSO, A., 2002. All the Truth About NevAr. *Applied Intelligence*, 16 (2), 101-118.

COPE, D., 2005. Computer Models of Musical Creativity. MIT Press.

BALTMAN, R. AND RADEZTSKY JR, R., 2004. Verlet integration and constraints in a six degree of freedom rigid body physics simulation. *Game Developers Conference 2004.*

VALENTE, L., CONCI, A. AND FEIJÓ, B., 2005. Real Time Game Loop Models for Single-Player Computer Games. *Conference Proceedings of the 4th Brazilian Workshop on Computer Games and Digital Entertainment*, 89-99.

DAVIS, L., 1991. *Handbook of Genetic Algorithms*. New York: Van Norstrand Reinhold.

KOZA, J., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.