

Genetic Programming Applied to the Armatures of a Human Object

Daniella Massa
James Brouder

June 29, 2021

Abstract & Project Goals

The goal of this project was to use genetic algorithms to create a walk cycle for a Humanoid mesh and Armature. To do this, we used Blender, an open source Python software. The motivation to do this project arose from the fact that Blender contains rigid structures called bones, which can be used to manipulate meshes into various poses to create animations. However, if the bone structure matches that of a real world robot, the bone data can also be exported to the robot to replicate the simulated animation in the real world. If we can evolve the walk cycle from genetic algorithms, this would open the door for robots who envision methods to navigate their environments. Furthermore as all motion on the robot is encoded in bones, these methods could be applied to interact with numerous objects and environments.

Genetic Algorithm Genotype

The genotype for our project was built from a series of dictionaries where the 10 keys in this dictionary are the names of the primary bones in the character (both shown below). The corresponding values at those keys are lists of three integer values which represent rotation across the x, y and z axis for each bone in the armature. Thus by applying the dictionary to the model, we can deform the model into different poses. To turn these dictionaries into animations, we create a list of these dictionaries, with random values for x, y, and z stored in the dictionary. This dictionary encodes the rules for interpolation between poses, or how much the bones will change between poses in the final walk cycle. To create the walk cycle, we begin with a list only containing the seed, and then begin applying each of the rules to the most recent pose in the walk cycle. The first rule is applied to the seed to create the second pose, and the second rule is applied to the second pose, until all the rules are used. At this point, the walk cycle is fully generated. Evaluation of the walk cycle will be discussed later, as it is a complicated process.

Initial Genetic Algorithm Implementation

With methods to generate random members, we then developed methods of crossover and mutation to create new members from existing ones in the population. The crossover method takes two individuals, and selects a random pivot point amongst the dictionary keys. Keys before the pivot remain unchanged, while those after the pivot are swapped. For mutation, an individual either remains the same or mutates based on a passed mutation probability. If the individual is mutated, for each pose dictionary in the walk cycle each x, y, and z rotation of the bones has a random chance of mutating. If mutation at that specific rotational value occurs, a random amount between $1/3$ and $-1/3$ is added to from the x, y or z data.

While the final fitness of our project is the ability to move in simulation, we first tested the genetic algorithm outside of Blender (in Google Colaboratory) to make things simpler. To do this we created an arbitrary fitness function which simply checks every rotational position, (x,y,z), at every pose, and awards a

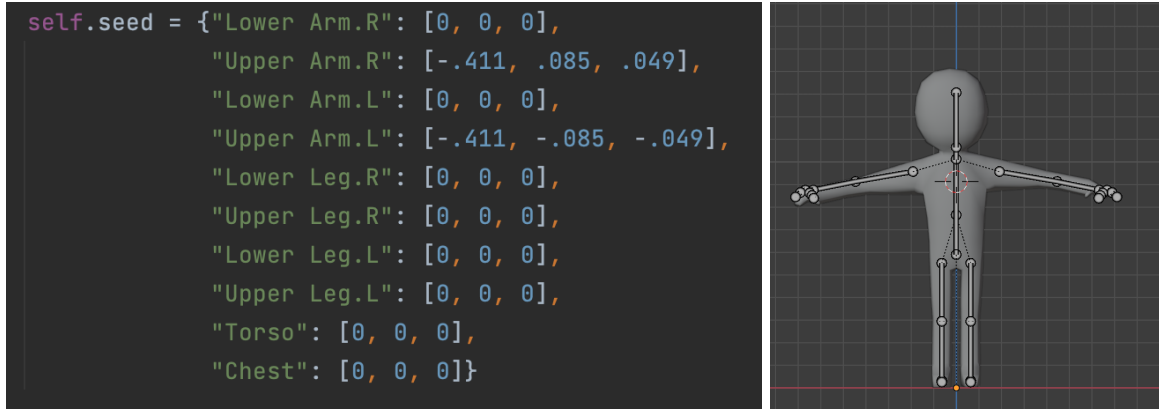


Figure 1: Dictionary of Bones (left) & Model (right)

point if the tuple is ordered. While this is a relatively simple problem, it was sufficient enough to test our GA.

To run our GA, we select a population size, a max number of generations, a mutation rate, a crossover rate, and a number of elites. We generate a population of random individuals of the size as specified by the population size parameter. We then evaluate the fitness of every member in the population, and sort the population by fitness. For the next generation, the number of elites is used to determine how many of the most fit individuals will be copied to the next generation. To create the remaining population, a roulette wheel is used to select groups of two individuals. These individuals are crossed over and mutated using the parameters passed in. These hybridized individuals are then added to the new population. Once the new population is the size of the original population, the generation is complete.

At this point, the population has each member's fitness re-evaluated, and then the population is restored. This process of creating populations is done for however many generations is specified, after which the GA terminates.

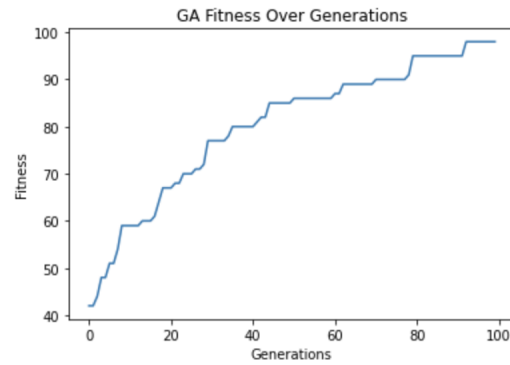


Figure 2: Monotonically Increasing Graph of Fitness

Running the GA with the arbitrary fitness function, we found that the members of the population were monotonically increasing. This was promising as it indicates that our GA is correctly selecting the most fit individuals. We ran several test cases to ensure that this was true with a variety of population sizes, number of maximum generations, number of elites, crossover probabilities and mutation probabilities. The graph (pictured right) shows a monotonically increasing population of 100 genomes, with a number of max generations of 100, number of elites of 20, and mutation and crossover rates at 1.

Generating Fitness From Blender

With the GA functioning on an arbitrary fitness function, we set out to evaluate the walk cycles in the Blender physic engine itself. We first tried simply applying the bones poses to the walk cycle, and then interpolating between the bones. This is the standard method for animation in Blender, and mirrors the way we store data closely. However, when using bones directly, at each frame, the bone data writes explicit x, y, and z locations to each vertex in the humanoid mesh. Since the bones are the only factor contributing to movement, the mesh for our characters was able to phase through the floor and other objects, as it already

was given explicit motion paths. This removes any capacity for the physics engine to factor in, making this workflow unusable.

While directly the motion with solely bones and keyframes did not work, we found that if the bone pose is applied to the mesh as a shape key, changing poses can interact with physics. Shape keys are essentially a saved version of the model, where the mesh has been deformed to the pose specified by the bones. These mesh keys encode data relative to the object's origin, and should the mesh hit an object, the origin is moved accordingly. Thus this mode of animation implementation is valid for our genetic algorithm, as there is real-time feedback from the physics engine.

To convert our animation to a series of shape keys, we created a helper method to create a single shape and then applied it to each pose in the walk cycle. With the poses created, the armature is reset to its original position, as all required bone data is now saved directly as shape keys. To apply the shape keys as an animation, several factors had to be considered. For one, shape keys hold a single parameter called a weight, which is between zero and one. If the weight is one, the shape keys deformation is applied, if the weight is zero, it is not, and values between zero and one interpolate between the mesh poses at weight one and weight zero. Additionally shape keys are defined as deformation from a parent shape key which contains the original mesh data, called the basis shape key. If multiple shape keys have a non-zero weight, their deformation stacks, allowing for drastic mesh deformation, as shown with the pictures below.

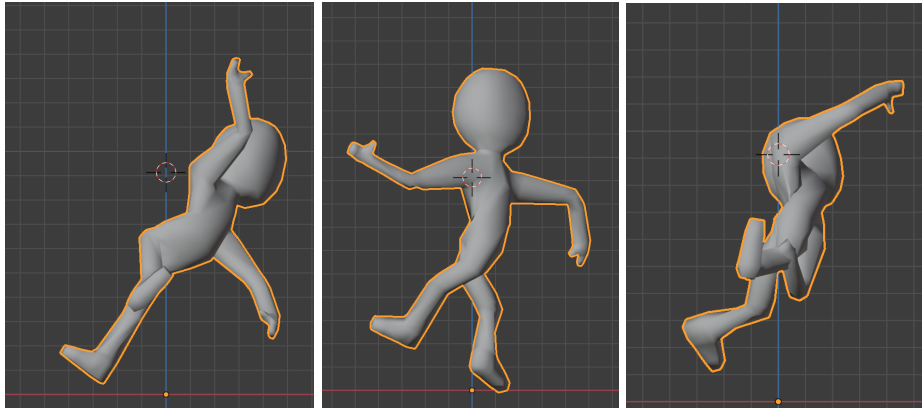


Figure 3: Key 1 Applied (left), Key 2 Applied (middle), Key 3 Applied (right)

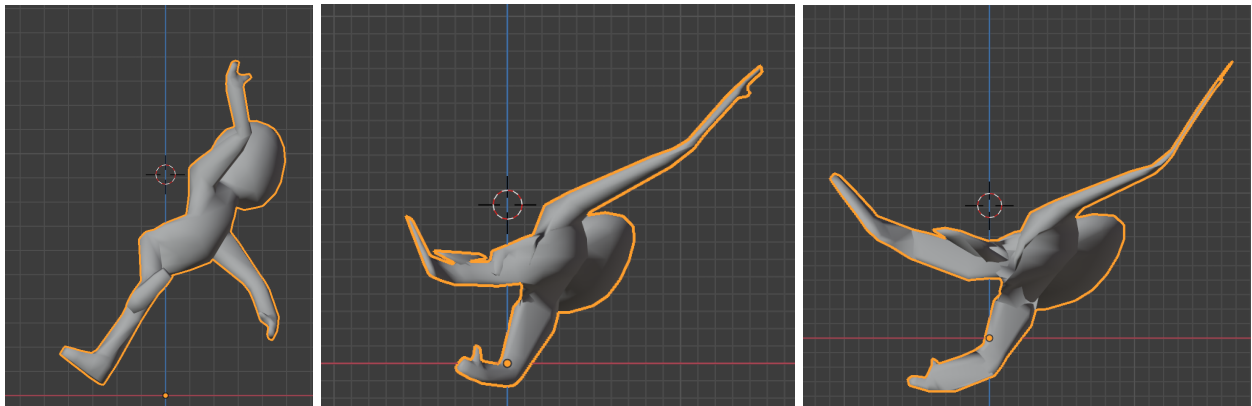


Figure 4: Key 1 Applied (left), Keys 1 & 2 Applied (middle), Keys 1, 2 & 3 Applied (right)

As multiple keys cannot be fully applied simultaneously, we needed to write methods to change the weights of the shape keys at the correct frames. The central premise is that each shape keyed pose is given a specific

frame. At that frame, the weight of the shape key should be one, and all others should be zero. If we set keyframes at each of these poses, Blender can interpolate between the weights in a valid way. This valid interpolation between the seed pose and pose one is shown by the images on the following page, as well as the shape key weights.

With a valid method for motion, we wrote a helper which parses through the poses and sets the keyframes to the correct values of zero and one. The timeline for the animation is shown by the image on the right of the following page. The yellow dots represent keyframes. For each weight value, the set of three dots is a result of the weight being keyframed to 0, then 1, then 0, with 8 frames of separation between. At this point, the animation can be played and the character will follow the same motion paths encoded by the original bone based animation, although now the character can have live feedback from the environment.

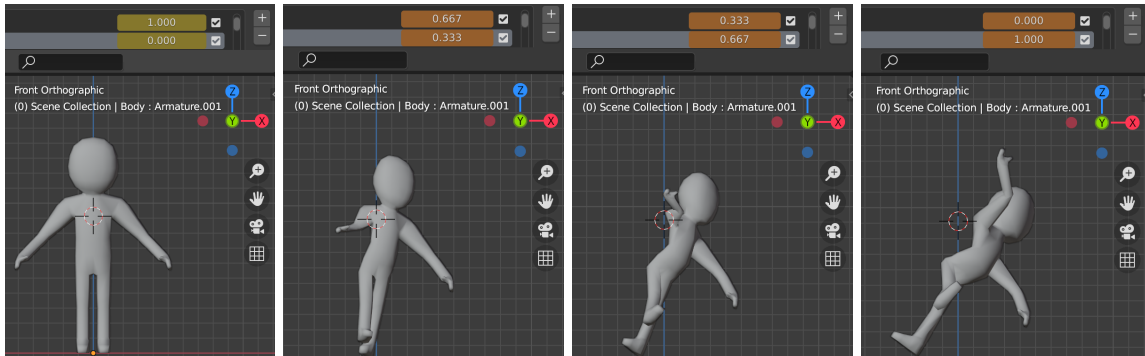


Figure 6: Interpolation Between the Seed Pose and Pose One

With the character now playing the animation in a way where physics is engaged, we parented a plane object to the character's chest bone to track motion. After the animation is played, the x coordinate of the plane becomes the fitness of the individual. However, when evaluating fitness, we do not want to wait for the animation to play in real time, as it is 10 seconds long, 240 frames playing at 24 FPS. To work around this, we iterate through each frame and apply the shape keys f-curve data driver at that frame to the characters mesh. F-curves drivers encode how much the mesh data will change between frames, given the active physics and the framerate of the animation. Thus by applying all the f-curve drivers for each frame in the timeline, we can have the animation reach its final frame, and evaluate fitness, as quickly as possible, instead of being constrained to 10 seconds.

Genetic Algorithms Implemented In Blender

With a method to evaluate the fitness of a Walk Cycle in Blender, we set out to run the GA with the armatures' fitness calculated in Blender. From previous work with robotics in Blender, we know that Blender is slower and more likely to crash than other IDE's. This is a result of the additional Python structures which allow for animation, physics, texturing, and other CAD processes not present in an IDE like PyCharm. As a result, we decided to implement the GA in a way where a helper script uses Blender for the evaluation of fitness, while a master

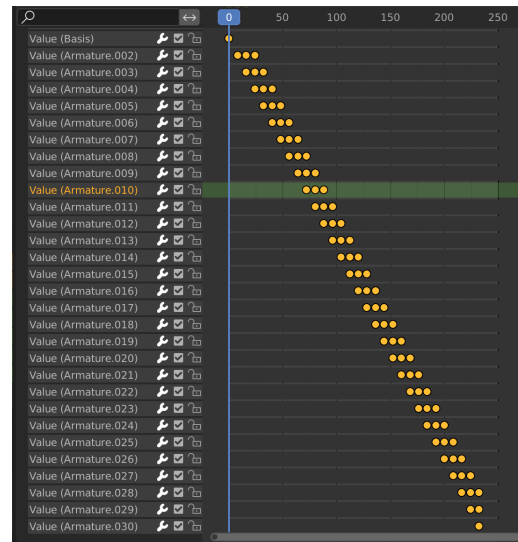


Figure 5: Timeline of Animation

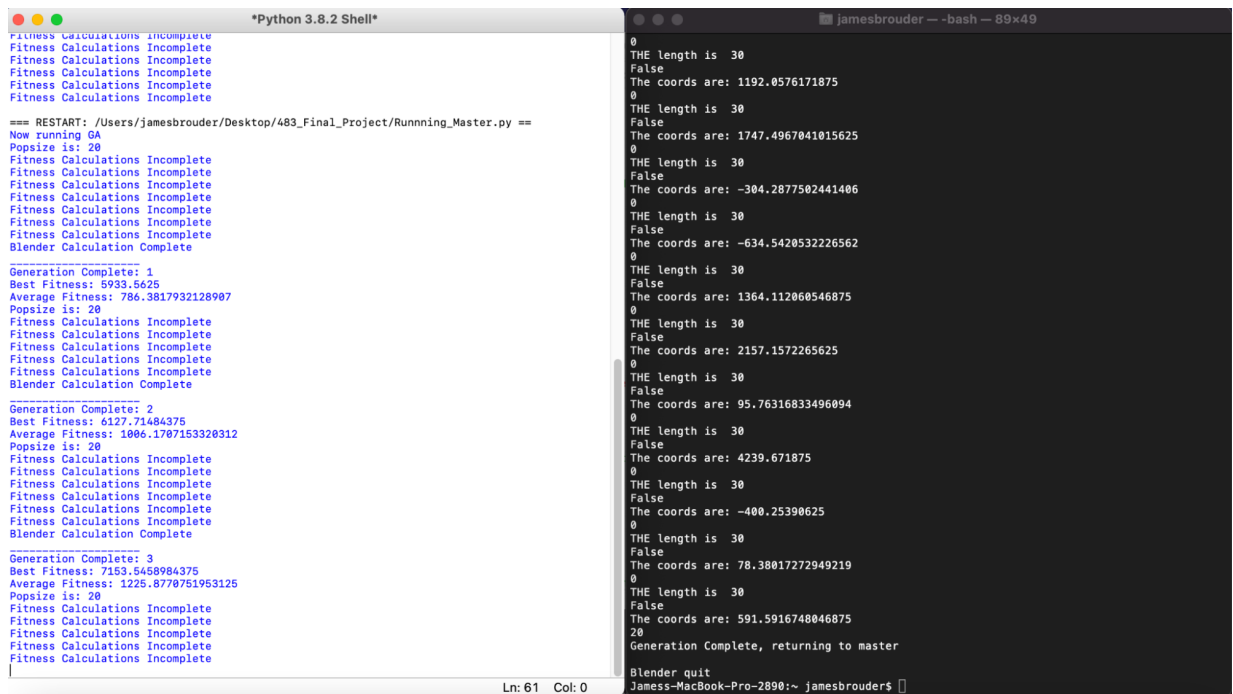
script manages all other GA processes.

To modify our GA accordingly, we used the Pickle module. This module uses a unique text encoding which allows Python to save and load any object type to a text file, by encoding it to a string, and then decoding back to its original type. This was useful as it provided an efficient way to transfer data between the master and the helper script. After the initial population is generated, the list of candidate walk cycles is pickled to a text file, and an accompanying indicator file which is completely empty. The Blender helper is then called via a subprocess, with the master script checking each second to see if the indicator file is empty.

In the Blender helper, the list of WalkCycles is pickled in, and each member has its fitness evaluated. When fitness calculations are complete, the list of candidates, which now have their fitness value, is pickled back into the text file it came from. The Blender helper then writes a single string to the indicator file, and terminates its process. When the master script detects the change in the indicator file it resets the indicator and the list of candidate walk cycles is pickled back into the master script. With fitness evaluated, the master script follows the same processes as our original GA. A set number of elites, the most fit individuals, are carried over to the next population. Then members are randomly selected, weighted by fitness, and mutated and crossed over until the next population is the same size as the original. With our next generation population, we repeat the process of picking the population and calling the Blender helper to gather fitnesses. This process of generating new populations and evaluating their fitnesses in Blender is repeated for however many generations is specified, at which point our GA terminates.

Results From Blender

The results from Blender were partially successful. The Master and Helper scripts interact correctly when the Blender process is called from the command line. However, when the Blender subprocess is run with the OS module or subprocess module, the master script never receives an indicator back. This is likely the result of some file path changing when the script is run by a user compared to being run as a subprocess. By tracing the output of the subprocess, we could likely debug this further, but given time constraints we were not able to. The interactions between the master script and the helper are shown below.



```
*Python 3.8.2 Shell*
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete

=== RESTART: /Users/jamesbrouder/Desktop/483_Final_Project/Running_Master.py ===
Now running GA
Popsize is: 20
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Blender Calculation Complete

Generation Complete: 1
Best Fitness: 5933.5625
Average Fitness: 786.3817932128907
Popsize is: 20
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Blender Calculation Complete

Generation Complete: 2
Best Fitness: 6127.71484375
Average Fitness: 1006.1707153320312
Popsize is: 20
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Blender Calculation Complete

Generation Complete: 3
Best Fitness: 7153.5458984375
Average Fitness: 1225.8770751953125
Popsize is: 20
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Blender Calculation Complete

jamesbrouder -- -bash -- 80x49
0
THE length is 30
False
The coords are: 1192.0576171875
0
THE length is 30
False
The coords are: 1747.4967041015625
0
THE length is 30
False
The coords are: -304.2877502441406
0
THE length is 30
False
The coords are: -634.5420532226562
0
THE length is 30
False
The coords are: 1364.112060546875
0
THE length is 30
False
The coords are: 2157.1572265625
0
THE length is 30
False
The coords are: 95.76316833496094
0
THE length is 30
False
The coords are: 4239.671875
0
THE length is 30
False
The coords are: -400.25390625
0
THE length is 30
False
The coords are: 78.38017272949219
0
THE length is 30
False
The coords are: 591.5916748046875
20
Generation Complete, returning to master
Blender quit
James-MacBook-Pro-2890:~ jamesbrouder$
```

Figure 7: Interactions Between The Master Script and The Helper

As expected, the fitness levels are monotonically increasing through generations. However, we were unable to run extensive trails with a high number of generations as each helper call must be made manually by the user. However, we could glean some useful information from the limited trails we were able to conduct. The image below shows a candidate of fitness 60, who moved 60 Blender units from the origin. For context, the character stands at 14 units tall. As the animation is 10 seconds long, in this trail the humanoid is moving 6 Blender units per second, or roughly .4 of its height per seconds. Inspecting this candidate’s animation more carefully, we can see that it’s velocity is created mostly at frame 97, when it successfully pushes its head through the physical floor, shown via the red line. After this point, the physics engine accounts for clipping by throwing the character backwards, displacing it from the origin and returning. Frames 97 and 110 are shown below.

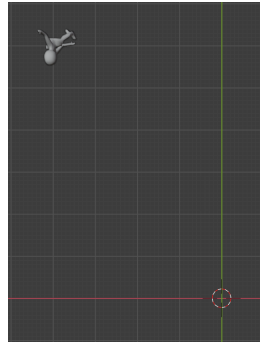


Figure 8: Fitness 60 Individual

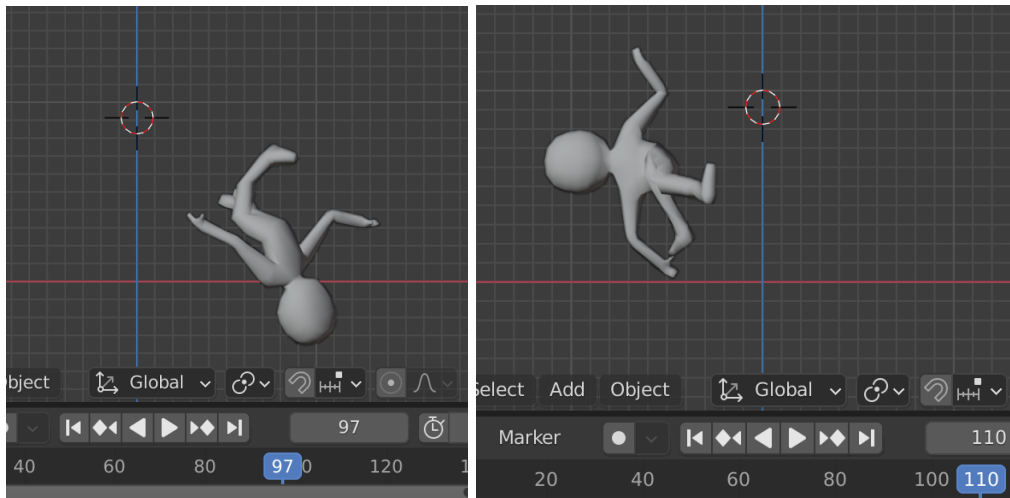


Figure 9: Sample Character & Clipping Example

While this is not the walk cycle we intended, we anticipated this result. As mentioned in Lehmen Et al., genetic algorithms using physics engines often exploit errors instead of developing the desired behavior[4]. In this case, the displacement generated by clipping into the environment is far greater than the displacement generated through simulated friction, which is the means by which natural walking occurs. As a result, the GA prioritizes individuals who break the simulation, rather than those who actually perform viable walk cycles. Our data confirms that this is the case. By generation 3, the average fitness is 1225.8 Blender units, indicating a velocity of 122.8 units per second, for an individual of high 14. Clearly this is not akin to walking but rather the result of clipping individuals being the most fit, and thus the algorithm converges on the best way to clip into the floor and be catapulted.

```

-----
Generation Complete: 3
Best Fitness: 7153.5458984375
Average Fitness: 1225.8770751953125
Popsize is: 20
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete
Fitness Calculations Incomplete

```

To resolve this problem, we would have to refine our fitness function to discourage these cheating behaviors. For one, all walk cycles creating non-manifold interaction between the character and the plane should be discarded, as non-manifold interactions are any two meshes which clip into each other. This would hopefully prevent catapulting individuals. We could also use more data from the tracking plane. If the tracking data was taken at each frame instead of only the final frame, fitness could be defined as a measure of how consistently the x value increases, with a hard cutoff for any individual traveling above a certain speed.

This would hopefully prioritize individuals who move via frictional interactions, as frictional interactions provide the model with a motion path derived from an acceleration force. If the position is derived from an underlying acceleration, it should increase consistently as it's a continuous function.

One last improvement which could be made would be to restrict the motion of certain joints in the simulation. Currently, the joints which move have their full range of x, y, and z rotation available. However, this is not the case for real joints, which have far more restricted ranges of motion. That being said, by locking certain axes of rotation, we can force the model to move with the same ranges of motion as a human. For instance, the leg bone would only be able to move between 0 and roughly 170 degrees along its central axis, with all other rotation paths locked. By restricting motion in this way, we force the GA to explore a search space which more closely resembles actual human motion, and thus any results from this should be more similar to actual human walk cycles.

Related Articles & Further Inspiration

One main source of inspiration for this project came from the “Evolving Virtual Creatures” paper discussed in class. The authors of this paper used a simulated 3-dimensional environment to apply a genetic algorithm that defined an individual’s fitness through behaviors of walking, jumping, swimming, and following[7]. Although this paper related heavily to our concept, our experiment differed in the fact that we did not use neural networks to manipulate behavior; instead we used Blender made objects like meshes and bones. We also decided to focus on one human behavior, instead of four. Another article we found is titled “Genetic Programming Evolution of Controllers for 3-D Character Animation.” This paper highlights more specifics involving animation techniques: such as key framing and inputting degrees of freedom. The authors evolved a dynamic controller that generates plausible motion for an animation (which essentially got rid of the tedious task animators need to do when inputting degrees of freedom). The authors eventually evolved controllers to make an animation of the jumping Pixar lamp[3]. This paper was extremely helpful for visualizing our experiment because it provided a strong background of how DOFs are interpolated through animation.

A few other papers that related to our research were “Memetic Evolution for Generic Full-Body Inverse Kinematics in Robotics and Animation”, “Evolving Dynamic Gaits on a Physical Robot”, and “Evolving Humanoids: Using Artificial Evolution as an Aid in the Design of Humanoid Robots”. The first paper discussed using evolutionary computing to solve inverse kinematics, using methods of mutation, recombination, adaption, and elitism exploitation[9]. The second paper explained how genetic algorithms can be applied beyond virtual simulation, and to the physical gaits of a robot[11]. The authors of the third paper generated a humanoid robot, in which they applied evolutionary techniques both in simulation and in the physical world, to the humanoid’s body and “brain”[2].

Some papers we found that were specifically helpful in terms of understanding animation with genetic programming were “Evolutionary computing method in 3D animation modeling cooperative design”, “An evolutionary computing approach for solving key frame extraction problem in video analytics”, “An Evolutionary Computing Approach to Solve Object Identification Problem for Fall Detection in Computer Vision-Based Video Surveillance Applications”, “3D Motion Estimation of Human By Genetic Algorithm”, and “Simulation of intelligent target hitting in obstructed path using physical body animation and genetic algorithm”. The first paper explains how 3D animations can be generated from ACIS rule expression through binary trees, each representing the x-axis, y-axis, and z-axis; the animations were generated through crossover, mutation, and selection[10]. The second paper discussed the Differential Evolution algorithm, and how it works significantly better than the SSIM algorithm when extracting key frames from three different videos[1]. The third paper also talked about the Differential Evolution algorithm, but it was implemented in a framework to detect an object within a given image, which was later used to detect the fall of elderly people in a surveillance application[8]. The authors of the fourth paper used a genetic algorithm to determine scalar factors of human joints for animation[5], and the authors of the fifth paper applied a genetic algorithm to train an object to find the correct path of a target while utilizing concepts of physical body animation[6].

References

- [1] Kevin Thomas Abraham, M Ashwin, Darshak Sundar, Tharic Ashoor, and G Jeyakumar. An evolutionary computing approach for solving key frame extraction problem in video analytics. In *2017 International Conference on Communication and Signal Processing (ICCSP)*, pages 1615–1619. IEEE, 2017.
- [2] Malachy Eaton. Evolving humanoids: Using artificial evolution as an aid in the design of humanoid robots. In *Frontiers in evolutionary robotics*. IntechOpen, 2008.
- [3] Larry Gritz and James K Hahn. Genetic programming evolution of controllers for 3-d character animation. *Genetic Programming*, 97:2, 1997.
- [4] Joel Lehman, Jeff Clune, Dusan Misevic, Christoph Adami, Lee Altenberg, Julie Beaulieu, Peter J Bentley, Samuel Bernard, Guillaume Beslon, David M Bryson, et al. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *Artificial life*, 26(2):274–306, 2020.
- [5] Chunhong Pan and Songde Ma. 3d motion estimation of human by genetic algorithm. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, volume 1, pages 159–163. IEEE, 2000.
- [6] Shivendra Shivani and Shailendra Tiwari. Simulation of intelligent target hitting in obstructed path using physical body animation and genetic algorithm. *Multimedia Tools and Applications*, 78(8):9763–9790, 2019.
- [7] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, 1994.
- [8] Katamneni Vinaya Sree and G Jeyakumar. An evolutionary computing approach to solve object identification problem for fall detection in computer vision-based video surveillance applications. In *Recent Advances on Memetic Algorithms and its Applications in Image Processing*, pages 1–18. Springer, 2020.
- [9] Sebastian Starke, Norman Hendrich, and Jianwei Zhang. Memetic evolution for generic full-body inverse kinematics in robotics and animation. *IEEE Transactions on Evolutionary Computation*, 23(3):406–420, 2018.
- [10] Hanchao Yu, Hong Liu, and Xiaopeng Yang. Evolutionary computing method in 3d animation modeling cooperative design. In *Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 296–303. IEEE, 2011.
- [11] Viktor Zykov, Josh Bongard, and Hod Lipson. Evolving dynamic gaits on a physical robot. In *Proceedings of Genetic and Evolutionary Computation Conference, late breaking paper, GECCO*, volume 4, 2004.