Physical Character Animation Using Machine Learning

Final Report

Team 4

Client/Advisor: Jim Lathrop

Rob Quinn - Project lead, Sim lead programmer, client communications

Joe Sogard - Web lead, Back-End programmer, Back-End QA

Joe Kuczek - Full stack web, SCRUM master

Luke Oetken - Simulation programmer, Status reporter

Andrew McKeighan - Simulation programmer

Kenneth Black - Simulation programmer, Machine Learning

sdmay18-04@iastate.edu

https://sdmay18-04.sd.ece.iastate.edu/

Table of Contents

1 Revised Project Design

1.1 Problem Statement

In modern video games there is an increasing need for high-fidelity physical and physics based animations for characters. These animations can be extremely time consuming and expensive to make using traditional methods such as hand-keying or motion capture, and require many professional animators. Also, most existing procedural animation tools only work on human skeletons.

1.2 Operating Environment

This is a virtual simulation rendered on the Unity 3d game engine. The engine creates most of the graphical interface, with our code being written on separate script components written in C#.

1.3 Intended Users and Intended uses

Our intended users are game developers who need animations for their characters. Many games today use physical animations where the movement of the character is driven by physics. Other games that use keyed animations may use physics simulation to make their animations realistic. Animations and especially walk cycles are time consuming and may require a dedicated professional to make them. Our application will take characters and simulate them to learn physically plausible movements which game developers can then use in their game.

1.4 Limitations

Limitations:

- There is a fixed amount of locomotion behaviors before the animal enters the behavior simulation, including walk, run, and sit.
- There are at least 3 animal species to test such as dog, snake, and inchworm.
- Limited by the amount of training data.

2 Implementation

This project creates physical animations for video game characters using genetic algorithms. Characters have realistic 3D physicality and learn coordinated muscle-based motion to satisfy a goal such as creating a walk cycle.

2.1 Character setup

Characters come from a 3D modeling application such as Maya. Our tools generate a skeleton for the character based on the animation bones. The skeleton includes collision objects and physics such as joints and masses. Adjustments can be made using the editor tools as needed.

2.2 Character muscles

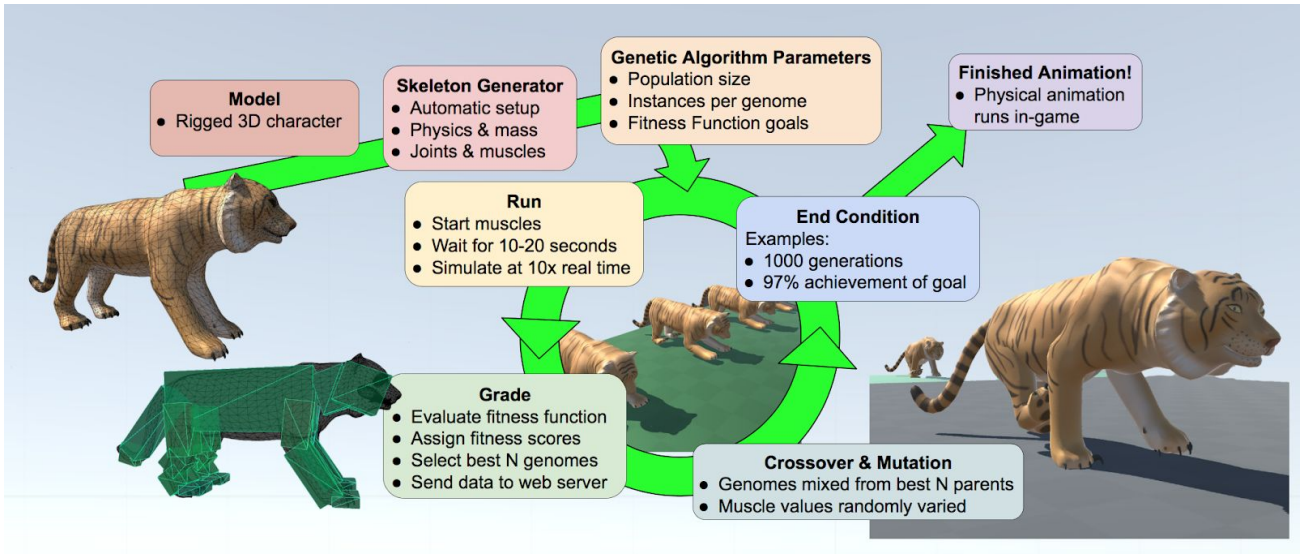
Characters have muscles on each joint. The muscles move using a torque equation that follows a sine wave with 4 parameters: amplitude, offset, center, and frequency. Muscle strength is scaled based on the animals weight and the size of the bone.

2.3 Genomes

Genomes are the name for our serialized data. The data for each muscle is serialized into one object called the “Brain Control Block” along with the name of the character, the generation, and the fitness score it received.

2.4 Genetic Algorithm

We implemented a genetic algorithm for our machine learning. This diagram shows how the process of using the genetic algorithm works.



1. Genetic algorithm parameters
    a. Population size is the number of genomes that are tested in each run (a run is indicated by the loop in the diagram).
    b. Instances per genome will make multiple characters for the same genome in the same run. In a perfect simulation they would be identical but due to very small physics inconsistencies depending on the position and due to floating point precision sometimes characters perform better or worse that they normally should. To smooth out these inconsistencies in the physics engine we make multiple copies of each genome and average their scores to find a more accurate score that better represents that genome.
    c. Fitness functions define which characters are the best of the run. The function is based on user defined parameters such as how far the character moves, their elevation, and the difference from where they started. Each parameter has a weight to define which goals are more important. Fitness functions are created as Scriptable Object files that go in the Assets folder of the project.

2. Run
    a. The muscles run based on their 4 parameters and the sine function.
    b. The simulation will go for as long as the user has defined, 10 to 20 seconds is good for a walk cycle.
    c. The user can speed up the simulation using the number keys to make the simulation go as fast as their computer can calculate the physics frames.
3. Grade
    a. Each genome is evaluated based on the fitness function.
    b. For multiple instances of the same genome, they are averaged.
    c. The best N genomes (as defined by the user) are saved as the best of that generation.
    d. The best genomes are sent to the web server.
4. Crossover & Mutation
    a. Some child genomes are created from genetic crossover - random mix of genome values from two parent genomes.
    b. Other child genomes are created by mutating the genomes of the best performers.
5. End Condition
    a. The simulation ends when either the set maximum number of generations is reached, or a set goal percentage is reached.
6. Finished Animation
    a. When the simulation is complete, the resulting best genome can be used to animate a character in a game.

3 Testing and Testing Results

3.1 Simulation Testing

Testing on the results of the simulation is done manually and through debug logging. It would be infeasible for us to develop automated testing to check whether the machine learning algorithm is running correctly, as results are subjective. Therefore to test our algorithm, a tester runs the simulation and checks whether the creatures are walking in a way that looks good. Throughout the code there are also several points where logging occurs, which helps show the tester what is happening within the simulation. This testing is done frequently, roughly half a dozen times a week for the entirety of the development process.

3.2 Web API Testing

Testing scripts were developed to verify the use of the back-end web API allowing access to the database. Using PHPUnit 5.7.27 for testing suite. Master testing execution script is found at /var/www/test/run_tests, which runs PHPUnit testing scripts. Results of these tests are displayed to the user on command line. These PHPUnit scripts test every aspect (Create, Read, Delete, and sometimes Update) of each of the four tables in the database

schema (Action, Family, Species, Generation). By executing this script a user can, at any moment, test whether the web API is working as predicted. These tests have been developed by the Back-End Quality Assurance Master and have been run regularly after every change to the API code.

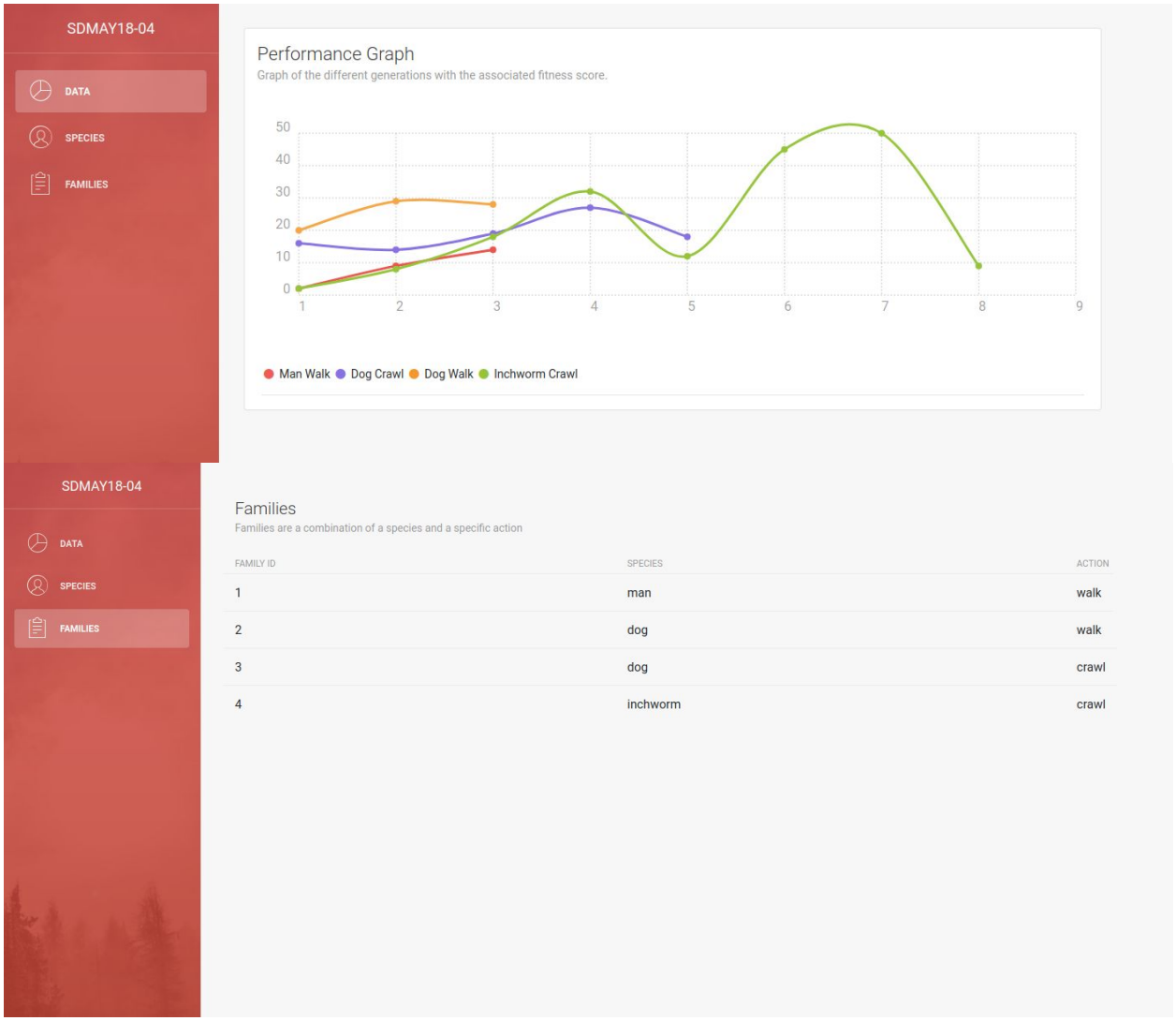
Testing occurred before and after integration of the simulation to web API interaction. The method for this interaction is creating a Generation and adding to the database. Because of lack of time, this testing was done manually by running a C Sharp script and checking whether it is accurately reflected in the database. Testing occurred often during development of this interaction and regularly after it was put into the Dev environment.

4 Appendix I - Operating Manual

4.1 Simulation

1. The simulation is tested and compatible with Unity version 2017.3.
2. Open the “Main” scene, this can be found in the Project Explorer.
3. Select “Runner” in the game hierarchy. There should be a component in the Inspector called “Run Locomotion GA”.
4. To select the animal you would like to perform the simulation with, find the prefab in the Project Explorer. Then drag the animal into the “Body” variable found in the “Run Locomotion GA”.
5. There are many parameters that can be set in the runner game component.
    a. Under the “GA parameters” header there are three variables, N, K, and I. N specifies the number of animals that will be spawned and mutated for each best genome saved in each iteration. K specifies the number of best genomes kept each iteration. I specifies how many times to duplicate each genome spawned. Thus the total number of animals spawned in the simulation equals N*K*I.
    b. Under the “Sim parameters” header is the animal prefab to be used for the simulation, max number of generations, spacing between each animal, and the time for each iteration to take.
    c. The current generation number can be found under the “Sim runtime” header while the simulation is running.
    d. The fitness function asset to be used in the simulation is specified under the “Fitness Function” header.
6. After all parameters are set as desired, the simulation can be started by simply running the game. The speed of the simulation can be controlled with the number keys 1-5, with 5 being the fastest speed. The simulation will handle saving all data and sending information to the web server.

4.2 Web Front End



Displayed above are screenshots of the front-end interface. The website utilizes the Web API in section 4.3 to perform basic data retrieval operations. The website is built using Bootstrap, jQuery, and Vue.js. We first retrieve data from the Web API and then propagate it to our Vue model. The website is fully responsive so tables and graphs can be viewed on any device.

The family and species pages' just list a table of all the different families and species. The dashboard page shows the graph of different generations and how they perform after a certain number of iterations. Performance is measured based on a fitness score. The primary factor that the fitness score is based on is the total distance traveled.

4.3 Web API

The web API can be tested using something like CURL, Postman, or in specific cases just a web browser. For each of the following requests, take the URL "http://sdmay18-04.ece.iastate.edu/" and append the URL provided in the request guide. Use the Request type and send the appropriate data. The response type is provided. THis guide can be found at /var/www/html/php/api_guide.md.

4.3.1 Species Functionality

Delete a species:

Request: POST
URL: /php/species/delete.php
Data:
```
{
        "species_id":##
}
```

Get all species:

Request: GET
URL: /php/species/get.php
Data: {}
Response Format:
```
[
        {
                "species_id": "2",
                "species_name": "dog"
        },
        {
                "species_id": "3",
                "species_name": "inchworm"
        },
        ...
]
```

Get a single species:

Request: GET
URL: /php/species/get.php
Data:
```
{
        "species_id":##
}
```

Response Format:

```
{
        "species_id": "##",
        "species_name": "dog"
}
```

Create a new species:

Request: POST
URL: /php/species/new.php
Data:

```
{
        "species_name":"XXX"
}
```

Response Format:

```
{
        "species_id": "5",
        "species_name": "XXX"
}
```

4.3.2 Family Functionality

Delete a family:

Request: POST
URL: /php/family/delete.php
Data:

```
{
        "family_id":##
}
```

Get all families:

Request: GET
URL: /php/family/get.php
Data: {}
Response Format:

```
[
        {
    "family_id": "1",
    "species_id": "1",
    "action_id": "1"
  },
  {
    "family_id": "2",
```

```
        "species_id": "2",
        "action_id": "1"
    },
            ...
]
```

Get a single family:

```
Request: GET
URL: /php/family/get.php
Data:
{
            "family_id":##
}
Response Format:
{
    "family_id": "##",
    "species_id": "1",
    "action_id": "1"
}
```

Create a new family:

```
Request: POST
URL: /php/family/new.php
Data:
{
            "species_id":##,
            "action_id":##
}
Response Format:
{
    "family_id": "1",
    "species_id": "##",
    "action_id": "##"
}
```

4.3.3 Action Functionality

Delete an action:

```
Request: POST
URL: /php/action/delete.php
Data:
```

```
{
        "action_id":##
}
```

Get all actions:

Request: GET
URL: /php/action/get.php
Data: {}
Response Format:

```
[
        {
      "action_id": "2",
      "action_name": "crawl"
   },
   {
      "action_id": "3",
      "action_name": "leap"
   },
        ...
]
```

Get a single action:

Request: GET
URL: /php/action/get.php
Data:

```
{
        "action_id":##
}
```

Response Format:

```
{
        "action_id": "##",
        "action_name": "crawl"
}
```

Create a new action:

Request: POST
URL: /php/action/new.php
Data:

```
{
        "action_name":"XXX"
}
```

Response Format:
{
"action_id": "5",
"action_name": "XXX"
}

4.3.4 Generation Functionality

Create a new generation:

Request: POST
URL: /php/generation/new.php
Data:
{
"family_id" : "XXX"
"generation_number" : "XXX"
"fitness_score" : "XXX"
"muscle_genome" : "XXX"
}
Response Format:
{
"family_id" : "XXX"
"generation_number" : "XXX"
"fitness_score" : "XXX"
"muscle_genome" : "XXX"
}

Read one/multiple generations:

Request: GET
URL: /php/generation/get.php
Data:
{
"family_id" : [X,X,X,X],
"generation" : "XXX",
"gen_min" : "XXX",
"gen_max" : "XXX",
"fitness_min" : "XXX",
"fitness_max" : "XXX"
}
Data Explanation:
family_id is an array of id numbers of specified families **[optional]**
generation is the specific generation number of item **[optional]**
gen_min is the minimum generation number inclusive (not used if generation specified) **[optional]**

gen_max is the maximum generation number inclusive (not used if generation specified) **[optional]**
fitness_min is the minimum fitness score number inclusive **[optional]**
fitness_max is the maximum fitness score number inclusive **[optional]**
Each successive parameter provided is treated as "gen_min=20 AND family_id=2 AND ...", so unions are not supported
Response Format:

```
[
        {
                "family_id" : "XXX",
                "generation_number" : "XXX",
                "fitness_score" : "XXX",
                "muscle_genome" : "XXX"
        },
        ...
]
```

4.4 API Testing

Testing for the API can be found in a series of bash and PHPUnit scripts. PHPUnit version used is 5.7. The master script which executes all PHPUnit tests is found at /var/www/test/run_tests on the server. Execution of this script will run the PHPUnit scripts contained in /var/www/test/scripts/, which are modularized to one test file for each table in the schema. These can also be run individually using PHP 5.6.32. WHen a test script is run it will display the result of running each appropriate CRUD function for the respective table.