Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Llama Farm

Brief Overview

In our project, we want to simulate a llama farm using advanced algorithmic techniques. We will be using Finite State Machines as the “Artificial Intelligence” for the llamas within our farm. Given a particular farm environment, we want to be able to study the progression of a population of llamas over time. In order to make this progession interesting, we will attribute a certain quantity of “genes” to these llamas. As time progresses within a certain environments, the llamas will compete with each other for resources, with some llamas dying every generation. The surviving llamas will pass on their genes to the next generation of llamas, slowly evolving a random llama population into a population fit for survival within their given environment.

At the end of this project, we want to have created an entertaining game-like platform in which users can create a population of llamas and watch as it progresses over time. Hopefully, this will have taught us how to create basic artificial intelligence using Finite State Machines for video games or more general uses, as well as how to use genetic programming to optimize a problem (in this case, llama survival).

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Feature Specification

When we start our project, we want to start with simply creating the basic algorithms that we will be utilizing to perform llama life simulation. The following selection of items represents the very minimal core features of our project.

- Create a "genome" for the llamas that represents various metrics that will eventually be used to drive the finite state machine. We have found various resources on how to create a genetic algorithm.
    - Implementation Overview: http://www.geneticprogramming.com/Tutorial/
    - Research Paper: Langdon, William B., et al. "Genetic programming: An introduction and tutorial, with a survey of techniques and applications." *Computational Intelligence: A Compendium.* Springer Berlin Heidelberg, 2008. 927-1028.
    - Theory Overview: http://www.ai-junkie.com/ga/intro/gat1.html
    - Tutorial: http://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php
- Create an algorithm that simulates meiosis, creating four "daughter" genomes from an original llama genome. We want to closely simulate real-life meiosis, as seen on this (http://en.wikipedia.org/wiki/Meiosis) Wikipedia page.
    - Including crossing over and random mutations to increase the genetic variability of a population
- Create a finite state machine that will be used for each llama that decides on the probability of choosing the next state based on the llama's genome
    - Overview of FSMs: http://en.wikipedia.org/wiki/Finite-state_machine

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

  - Theory: http://www.ai-junkie.com/architecture/state_driven/tut_state1.html
  - Application to Games: http://gameprogrammingpatterns.com/state.html
- Create a basic environment in which llamas can be placed into and interact with. This will be relatively easy for us after the completion of Problem Set 7
- Create a simulation that will drive the llamas within an environment over multiple generations

After these core features are out of the way, we want to have some slightly advanced extensions that will allow us to better analyze the progression of a llama population.

- Create an interface to track the presence of certain genes within a certain population over the course of the simulation. This interface will most likely plot the average level of a gene within the population over time.
  - Track what the optimal level of a gene is for a certain environment. Is a gene that codes for aggression less favorable in certain settings?
- Allow the user to change the environment type in the middle of a simulation, forcing llamas already adapted to one type of environment to adapt to a different one,
  - What would happen if llamas raised in a rainforest were all of a sudden transported to the middle of a desert?
- Allow the user to interact directly with the simulation
  - Forcibly kill llamas
  - Grow food for llamas to eat
  - Duplicate llamas
  - There's almost an endless number of things that we could implement here

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Finally, we have a few ideas that we are almost 100% certain that we will not have time to implement. If we do manage to get a majority of our advanced extensions out of the way, we hope to work on these.

- Create a second genetic algorithm that will control the food species of the world. Instead of having food simply appear at random based on a parameter given for the environment (Deserts have a 10% chance of food appearing, rainforests have a 90% chance), the plants that make up the food species will decide whether or not to multiply based on their genetic programming, which is affected by the environment just as the llama programming is.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Technical Specification

We plan to complete our final project using Java. We chose Java as a programming language because we wanted to use Object Oriented Programming, which would make it simple for us to break apart our work into multiple, easy-to-digest segments. We also wanted a language that made it easy to run a graphics-based simulation. Java, with the java.lang.Runnable interface, allows us to create a program that runs on a Thread and updates itself based on a given Frames per Second.

The following list details the Java Classes that we will create for this project along with their intended use.

- Genome Class – This class will simply contain a genome, as well as various methods that will allow genomes to undergo meiosis and combine with other genomes. The genome class can work standalone without any specific species and will simply operate on Strings that represent the genome of a species.
- Llama Class – This class will represent a single llama. Each llama will contain a genome as well as a finite-state machine representing all of the llama's states. The finite state machine of this class is where most of the "decision making" of the simulation will be done. The specification of a Llama's genome as well as the finite state machine will be present further down in this document.
- World Class – This class will represent a world and will contain specifications of the environment, as well as the llamas within it. It will be able to pass information about food, boundaries, and nearby llamas to any llama within the world. The simulation class will utilize the world class in order to run the simulation.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

- Application Wrapper – This class will simply create a Container for the Graphical Interface for our project. After doing so, it will have pretty much no other purpose.
- Graphics Panel Class – This is the main window that runs the simulation. It will extend from javax.swing.JPanel and implement java.lang.Runnable as well as various mouse and key listeners in order to allow for user interaction. It will update itself 30 times a second (the simulation will run at 30 FPS) and hold a World Class that will contain all the other classes.

Note that none of this is final and that we may choose to add more classes later in order to expand on the functionality of our Llama Farm simulation.

One advantage of the way the above classes are organized is that we can fully utilize the OOP Concept of encapsulation. Encapsulation allows for the packaging of data and functions into a single class. Other classes can simply call functions of one class while being assured that the implementation within the other class runs fine. This will allow us to break our problem up so that each of us can work on a different aspect of the final. For example, one partner could work on the genome class and begin implementing a Meiosis() method. The other partner can just assume that Genome.Meiosis() works as intended a program some other function, say Llama.BreedWith(Llama other). Within Llama.BreedWith, the second partner can simply call the Meiosis() method to retrieve new possible genomes. If this method is broken, it can simply be updated later without changing any code in the Llama class.

Next, we will explain the precise methods by which the Genome, Llama, World, and Panel Classes will function. We do not need to worry too much about the Application Wrapper class within this technical specification.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Genome Class

For this algorithm, we will be implementing a form of double stranded DNA. Each gene will be represented by 2 strands of multiple bits. For example, a gene that encodes for Llama Laziness could be encoded by the following:

<u>Llama 1</u>

Laziness: 11101

Laziness: 10111

<u>Llama 2</u>

Laziness: 00001

Laziness: 01000

As you can see, the two strands are independent from each other (unlike in normal DNA). While we may eventually decide to change this, we are now assuming that the Llama's "Laziness Index" is a function of the proportion of 1s in the genome. Llama 1 above would have a Laziness Index of 80% while Llama 2 would have a Laziness Index of 20%.

During Llama reproduction, the following events will happen for each llama in order to simulate meiosis.

1. The DNA strands will dissociate from each other.
2. Crossing over may happen. During crossing over, the two strands are mixed together with each other. If Llama 2's genome were to undergo crossing over at the 3rd position, it would end up with the following 2 DNA strands: 00000 and 01001.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

3. Mutation may happen. We will assign a random chance that one of the bits will flip. This will ensure that there is some semblance of population variability.

After that is done, the two llamas will randomly recombine their genomes in order to create 2 new baby llamas. If Llama 1 and 2 were to breed with each other (assuming no crossover in Llama 1, crossover at the 3rd loci in Llama 2, and no mutation), two possible child outcomes are as follows.

Child 1

Laziness: 11101

Laziness: 00000

Child 2

Laziness: 10111

Laziness: 01001

Child 1 will have a Laziness Index of 40% and Child 2 will have a Laziness index of 60%. In each iteration of the world class, some of these children will die, leaving the remainder to recombine and create new llamas. Over time, it is expected for the average Laziness Index of the entire llama population to reach an optimal value.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Llama Class – Llama Genome

Each llama will have the following genes. Note that this list is still a work in progress and may change dramatically in the final iteration of our project. These genes will influence one of two things – the Llama's finite state machine, or the Llama's "Cosmetic Attributes."

Finite State Machine-Influencing Genes

- Hunger Drive Index – Represents the possibility that a llama will decide to search for food. Note that our llamas, as opposed to normal animals, will eat food based on the hunger drive regardless of their hunger state. A llama that has almost no energy will not eat if its hunger drive isn't high enough. Likewise, a llama with a very large hunger drive may eat even when it is full. As you can see, both of these extreme scenarios are evolutionarily unfavorable.
- Laziness Index – A lazy llama uses less energy. However, a lazy llama is also less likely to escape from another llama that wishes to kill it. This index represents the probability that an idle llama will decide to start walking. A llama that is walking around can escape from nearby llamas, whereas a llama that is sitting down is unable to do so.
- Violence Index – A violent llama is more likely to attack other llamas, killing them and removing them from the gene pool. Violent llamas use a lot of energy while attacking, but may remove nearby llamas that are competing against it for food. Thus, violence may actually be beneficial for the llamas. However, indiscriminately killing all other llamas is also not the best of ideas as a violent llama could run out of energy very quickly.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Cosmetic Attribute Genes

- Metabolism Index – A llama's metabolism represents the rate at which a llama consumes energy. A llama with a high Metabolism Index will continuously lose energy at a rapid pace, whereas a llama with a low Metabolism Index will lose energy more slowly. Though a high metabolism seems like a disadvantage (The llamas in this world don't need to worry about becoming fat, unlike normal humans), a high metabolism also grants llamas various perks. For example, a high metabolism gives a llama high speed.
- Style Index – A llama's style index represents how fashionable a llama is. This will affect various aspects of llama decision making. For example, aggressive llamas may choose to attack stylish llamas more often out of jealousy. Passive llamas, on the other hand, may become so overwhelmed by the style of a fashionable llama that they simply get stunned and enter a random walking pattern, forgetting everything even if they were seeking food near the stylish llama. A stylish llama also looks cooler, which offers it not that much practical functionality (just like human fashion).

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Llama Class – Llama Finite State Machine

A finite state machine is simply a model that represents various states that a llama can be in, as well as the chances that a llama will shift to another state within the machine. In our llamas, the finite state machine will contain the following states. Note that this list is also still a work-in-progress.

- Idle State – Llamas in this state literally do nothing. They just sit still and chew grass or something like that. They use very little energy in this state. However, llamas in the idle state are unable to immediate enter a running state to escape from other llamas that attack them. Lazy llamas are also less likely to get up and start searching for food, and might die due to their laziness.
- Food-Seeking State – Llamas in this state have acted on their hunger drive and are now searching for food. They will keep searching until they have either found food, are killed by an attacking llama, or stunned by a super-stylish llama.
- Eating State – Llamas in this state have successfully found food and now are eating it. They will remain here until they have finished eating their food and will return to the idle state once done.
- Aggressive State – Llamas in the aggressive state have exceeded their violence index and now have a serious inclination to commit crimes, like killing another llama. They will walk around and seek out llamas to kill.
- Attacking State – A llama in the attack state has just exited the aggressive state or has exited the random walk state in response to a nearby aggressive llama. A llama in the attacking state will continue to attack until it has either died or

finished destroying its target. If it wins the fight, then it will go back into the idle state.

- Running Away State – A llama will enter this state from the Random Walk state if it senses a nearby aggressive llama and decides that it is too meek to continue gracing the presence of the aggressor. It will then continue to run until it regresses to the Random Walk state.
- Random Walk State – The random walk state is the general “Ready” state for all llamas. A llama will use up energy in the Random Walk state, but is also ready to immediately enter an attacking or running away position. A randomly walking llama is also more likely to randomly chance upon food, which it will gladly eat.
- Death State – A llama can enter the Death State from literally any other state. To do so, it must run out of energy. In our simulation, energy also represents the health of a llama. Thus, when llamas are in combat, they are losing health, which is also energy. The death state also doesn’t really count as a Llama state, as a llama in the death state is technically non-existent, with its genes lost forever and ever.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

World Class

The World Class is the class that the main Graphics Panel Class will use to run the simulation. Each instance of the World Class will represent some kind of environment. Environments will have the following parameters:

- Energy Usage – Represents how much energy each llama will use each “turn.” For example, deserts or polar environments would have a high energy usage, whereas more temperate ecosystems would have a low energy usage.
- Arability – Represents the chance that a new unit of food will appear on the map in a “turn.” One example of a highly arable ecosystem is a rainforest. On the other hand, a desert would have a very low arability, which a savannah would be somewhere in between.
- Initial Genome – Though this isn’t particularly related to the environment in a physical sense, we are including it in the World Class for our simulation. The initial genome can either be set as random, or forced to be towards a certain value. This will help us answer questions such as:
    - What is the absolute optimum genome for a given ecosystem?
    - How does this optimum change if the llamas in the ecosystem have a penchant for violence (High starting Violence Index)?
    - What about a penchant for laziness?

Each World Class will also have a Tick() function within it. The main graphics panel calls the Tick() whenever it needs to progress a unit of time forwards in the simulation. More about the running of the simulation will be described in the next section.

Panel Class (AKA The Main Simulation)

The Panel Class, which controls the World, implements java.lang.Runnable. What this means is that Java will call the run() method of the Panel Class once the application begins to run. Our run() function will be like the following:

```
public void run()
{
     Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
     isRunning = true;
     while (isRunning)
     {
          gameUpdate();
          gameRender();
          paintScreen();
          try
          {
               Thread.sleep(1000 / framesPerSecond);
          }
          catch (InterruptedException ex)
          {
               repaint();
          }
     Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
     }
     System.exit(0);
}
```

What the above code does is loop constantly, running Update, Render, and Paint until the simulation is finished running. After running is complete, the program then quits. The Render and Paint functions will simply draw the current state of the world using double-buffering, and aren't quite complicated from an algorithmic standpoint. These two functions are one of the last "core" features that we will implement. The gameUpdate() function, however, is where most of the heavy lifting of our simulation will happen. In the next section, we will detail the specifications of gameUpdate().

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

GameUpdate()

GameUpdate() is called once every frame and will represent the progression of the simulation. Earlier on in the code, GameInit() is called to set up all necessary variables for GameUpdate(). Below is a model on how GameUpdate() runs.

1. Check to see if a generation has completed.
    a. If 50% of the llamas within a generation are dead, then a generation is considered to be complete. The remaining llamas undergo meiosis, spawning new children llama while dying themselves. These new children are then passed to the next generation.
2. Take care of all necessary world needs.
    a. Grow new plants / remove eaten plants
    b. Remove dead llamas from the world
3. Run the world's Tick().
    a. Evaluate every Llama's finite state machine and have them shift to a different state within the machine if necessary. This is where all the llama "Artificial Intelligence" occurs.
    b. Have each llama act based on the action program of its given state.
    c. Update each llamas' attributes based on the results of all action programs.

GameUpdate() will keep running until the user finally stops the simulation after having observed generations upon generations of llama social activity.

Humza Tahir: htahir@college.harvard.edu
Raahil Sha: raahilsha@college.harvard.edu

Next Steps

Below is a checklist for what we hope to accomplish by the next checkpoint!

1. Download Eclipse and set up a Github Directory to sync Project directories between both team members
2. Create the Genome class and implement its basic functionality
    a. Double stranded data structure that can easily cross over and mutate
    b. Meiosis of a certain genome
        i. Includes crossover and mutation
    c. Combination of 2 genomes to create child genomes
3. Set up Application Wrapper and Graphics Panel
    a. These will hold nothing for the moment, but are an integral part of the structure of our program