### **Documentation: Smart Boats**

**Introduction: Description of Project**

The nice thing about using genetic algorithms in a project is that example code can easily be ported from application to application. The core mechanics of selection and reproduction don’t need to change. There are, however, three key components to genetic algorithms that one has to customize for each use, being the Population size and mutation rate, The fitness function, Genotype and Phenotype”.

For our smart boats project, we took inspiration from Jer Thorp, a Canadian data artist from Vancouver, British Columbia, who released a genetic algorithms example on his blog entitled “Smart Rockets”. Jer points out that NASA uses evolutionary computing techniques to solve all sorts of problems, from satellite antenna design to rocket firing patterns. This inspired him to create a Flash demonstration of evolving rockets. We took the same idea but instead we visualized them as boats because well, we (Saad and Khalid) both like boats.

Here is a description of the scenario, a population of boats launch from the bottom of the screen with the goal of hitting a target at the top of the screen (with obstacles blocking a straight-line path). Each boat is equipped with five thrusters of variable strength and direction. The thrusters don’t fire all at once and continuously; rather, they fire one at a time in a custom sequence. Our boats will have only one thruster, and this thruster will be able to fire in any direction with any strength for every frame of animation. This isn’t particularly realistic, but it will make building out the framework a little easier.

**Process:**

We start off by making our basic Mover class:

class Boats {

// A boat has three vectors: location, velocity, acceleration.

PVector location;

Pvector velocity;

Pvector acceleration;

// Accumulating forces into acceleration (Newton’s 2nd law)

void applyForce(PVector f) {

acceleration.add(f);

}

// Our simple physics model (Euler integration) void update() {

void update()

// Velocity changes according to acceleration.

velocity.add(acceleration);

// Location changes according to velocity.

location.add(velocity);

acceleration.mult(0);

}

}

Using the above framework, we can implement our smart boat by saying that for every frame of animation, we call “applyForce()” with a new force. The “thruster” applies a single force to the boat each time through the” draw()” function.

Let’s now go through the three keys to programming our own custom genetic algorithm example as outlined in the previous section.

* **Key #1: Population size and mutation rate**

We can actually hold off on this first key for the moment. Our strategy will be to pick some reasonable numbers (a population of 100 boats, mutation rate of 1%) and build out the system, playing with these numbers once we have our sketch up and running.

* **Key #2: The fitness function**

Playing around with the mutation rate or population total is pretty easy and involves little more than typing numbers in your sketch. The real hard work of a developing a genetic algorithm is in writing a fitness function. If you cannot define your problem’s goals and evaluate numerically how well those goals have been achieved, then you will not have successful evolution in your simulation.

The goal of a boat reaching a target. In other words, the closer a boat gets to the target, the higher the fitness. Fitness is inversely proportional to distance: the smaller the distance, the greater the fitness; the greater the distance, the smaller the fitness.

void fitness() {

// How close did we get?

float d = PVector.dist(location,target);

// Fitness is inversely proportional to distance.

fitness = pow(1/d,2);

}

This is a pretty simple fitness function. By using one divided by distance, large distances become small numbers and small distances become large. Each boat has a thruster that fires in a variable direction with a variable magnitude in each frame. And so, we need a “PVector” for each frame of animation. Our genotype, the data required to encode the boat’s behavior, is therefore an array of “PVectors”. To make a slight improvement to our program however, we make the fitness function exponential by raising the distance “d” to the power of 2.

* **Key #3: Genotype and Phenotype**

The final key to designing your own genetic algorithm relates to how you choose to encode the properties of your system. What are you trying to express, and how can you translate that expression into a bunch of numbers? What is the genotype and phenotype?

Each boat has a thruster that fires in a variable direction with a variable magnitude in each frame. And so we need a PVector for each frame of animation. Our genotype, the data required to encode the boat’s behavior, is therefore an array of PVectors.

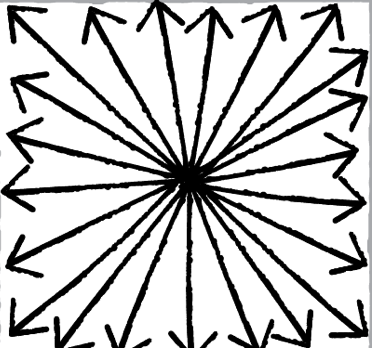
class DNA {

PVector[] genes;

we have an array of characters and picked a random character for each element of the array. Additionally, we initialize a DNA sequence as an array of random “Pvectors”. Now, your instinct in creating a random PVector might be as follows:

PVector v = new PVector(random(-1,1),random(-1,1));

However, if we were to draw every single possible vector we might pick, the result would fill a square (picture attached below). In this case, it probably doesn’t matter, but there is a slight bias to diagonals here given that a “PVector” from the center of a square to a corner is longer than a purely vertical or horizontal one.



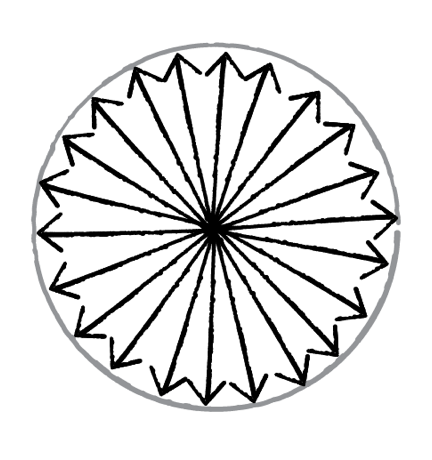
So instead, we picked a random angle and make a ”Pvector” of length one from that angle, giving us a circle (see picture below). This could be easily done with a Polar to Cartesian conversion, but a quicker path to the result is just to use PVectors Random2D()” function

for (int i = 0; i < genes.length; i++) {

// Making a PVector from a random angle

genes[i] = PVector.random2D();

}



A “Pvector” of length one is actually going to be quite a large force, as forces are applied to acceleration, which accumulates into velocity thirty times per second. So we added a variable that is a maximum force that scales all the “Pvectors” in our DNA Class. This will control the thruster power.

class DNA {

// The genetic sequence is an array of PVectors.

PVector[] genes;

// How strong can the thrusters be?

float maxforce = 0.1;

DNA() {

// We need a PVector for every frame of the Boat’s life.

genes = new PVector[lifetime];

for (int i = 0; i < genes.length; i++) {

genes[i] = PVector.random2D();

// Scaling the PVectors randomly, but no stronger than maximum force

genes[i].mult(random(0, maxforce));

}

}

We also created an array of “Pvectors” with length lifetime. We need a ”Pvector” for each frame of the boat’s life, and the above assumes the existence of a global variable lifetime that stores the total number of frames in each generation’s life cycle.

The expression of this array of “Pvectors”, the phenotype, is a” Boat” Class All we did was add an instance of a DNA object to the class. The fitness variable will also live here. Only the Boat object knows how to compute its distance to the target, and therefore the fitness function will live here in the phenotype as well.

class Boat {

// A Boat has DNA.

DNA dna;

// A Boat has fitness.

float fitness;

PVector location;

PVector velocity;

PVector acceleration;

}

We are using the DNA class to march through the array of “Pvectors”, and applying them one at a time as a force to the boat. To do this, we also had to add an integer that acts as a counter to walk through the array.

int geneCounter = 0;

void run() {

// Apply a force from the genes array.

applyForce(dna.genes[geneCounter]);

// Go to the next force in the genes array.

geneCounter++;

// Update the Boat’s physics.

update();

}

We now have our DNA class (genotype) and our Boatclass (phenotype). So, we then moved forward to our Population class, which manages an array of boats and has the functionality for selection and reproduction.

class Population {

// Population has variables to keep track of mutation rate, current population array, mating pool, and number of generations.

float mutationRate;

Boat[] population;

ArrayList<Boat> matingPool;

int generations;

// These functions haven’t changed, so no need to go through the code again.

void fitness () {}

void selection () {}

void reproduction () {}

}

The boats, however, need to live for a period of time before they can be evaluated; they need to be given a chance to make their attempt at reaching the target. Therefore, we added one more function to the Population class that runs the physics simulation itself.

void live () {

for (int i = 0; i < population.length; i++) {

// The run function takes care of the forces, updating the Boat’s location, and displaying it.

population[i].run();

}

}

Finally, we’re ready for the setup() and draw() functions. We implement the steps of the genetic algorithm in the appropriate order by calling the functions in the Population class.

population.fitness();

population.selection();

population.reproduction();

All in all, the steps of our program works as follows:

1. Create a population of boats
2. Let the boats live for N frames
3. Evolve the next generation
   * Selection
   * Reproduction
4. Return to Step #2