Technical Guide

Artificial Life

Name: David Craig

Supervisor: Alistair Sutherland

Student number: 14506617

Date: 18/05/2019

1. Introduction

1.1 Project Overview

The project is an attempt to model some simple single-celled creatures which can interact with each other in a virtual world. A primary goal was to have organisms cooperate and compete with each other. Another goal was to mimic a kind of Darwinian evolution to observe if they cooperate to become complex multicellular organisms and also what kind of traits become prominent over time and over many simulations.

1.2 Motivation

The motivation for this project comes solely for a passion for algorithms and a vision of a Genetic Algorithm that models a society which people can interact with. It is strictly an algorithms and programming practice that I wished to share with others and hoped others would find useful. I was as also largely inspired by the implementations of genetic algorithms (particularly Karl Sims) referenced below.

1.3 Research

The main components of my research were in figuring out how Genetic Algorithms were implemented and how I could code cooperative and competitive qualities to them. The book "An Introduction to Genetic Algorithms" by Melanie Mitchell gave a hugely useful and comprehensive outlook on the history of implementation of game theory and the wide range of diversity in it's uses and implementation. The other component of research was simply how to use HTML Canvas which I learned mainly using w3schools. Although I'd never worked with Javascript before I learned it on the go; learning features as I required them in the project.

1.4 Glossary

Diglets: Individuals in the simulation.

Genes: The variable characteristics of each Diglet.

Digritos: A group of Diglets, which represent a multicellular organism.

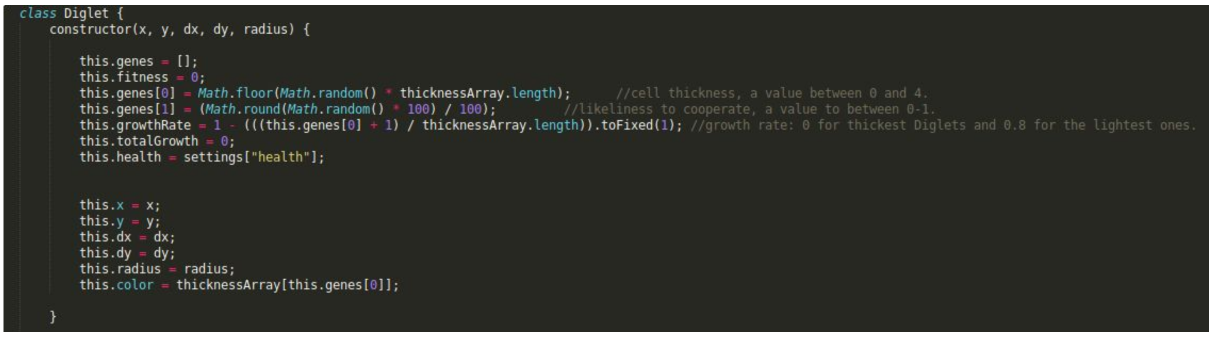
Population: The sum of individuals in the simulation.

Mating Pool: The pool of potential mates.

2. Design

Diglets

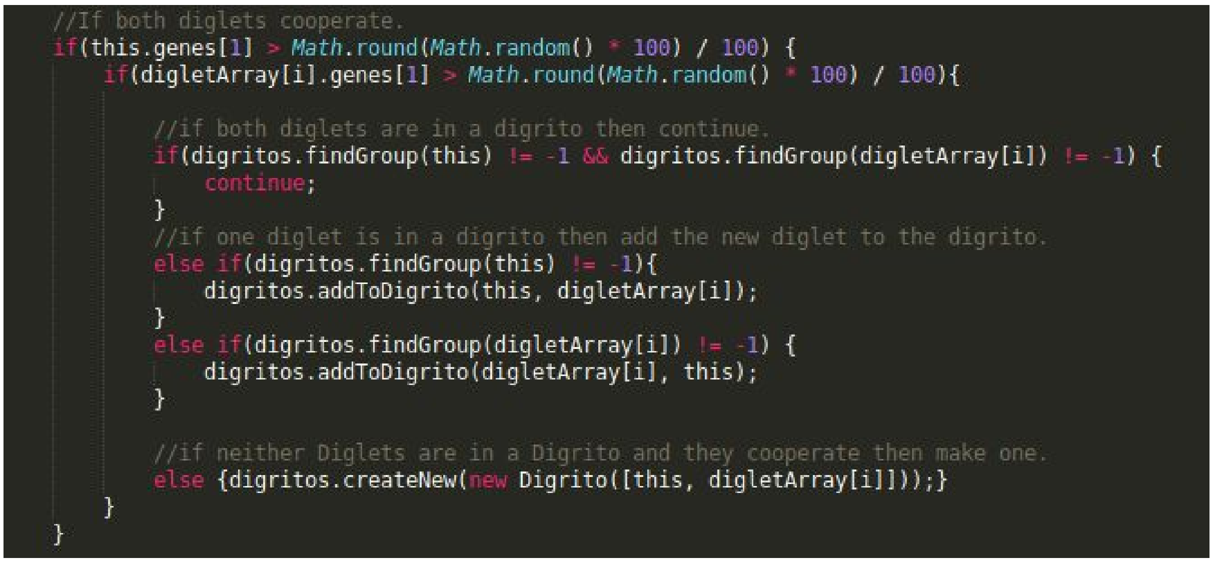
Diglet.js contains information that is applied to individual Diglets. The constructor initialises their properties such as their genes, which are stored in a short array called this.genes and generated essentially using the Math.random() function. Diglets also have a value of fitness used to determine how likely they are to mate; a total growth rate which I use to measure fitness; health which is a value that decays over time proportional to the energy they can generate; and their x and y coordinates, as well as their velocity which are stored as dx and dy. Radius is kept for Diglets as is their color(which is related to their thickness.



```
class Diglet {
    constructor(x, y, dx, dy, radius) {

        this.genes = [];
        this.fitness = 0;
        this.genes[0] = Math.floor(Math.random() * thicknessArray.length);      //cell thickness, a value between 0 and 4.
        this.genes[1] = (Math.round(Math.random() * 100) / 100);         //likeliness to cooperate, a value to between 0-1.
        this.growthRate = 1 - (((this.genes[0] + 1) / thicknessArray.length)).toFixed(1); //growth rate: 0 for thickest Diglets and 0.8 for the lightest ones.
        this.totalGrowth = 0;
        this.health = settings["health"];


        this.x = x;
        this.y = y;
        this.dx = dx;
        this.dy = dy;
        this.radius = radius;
        this.color = thicknessArray[this.genes[0]];

    }
```

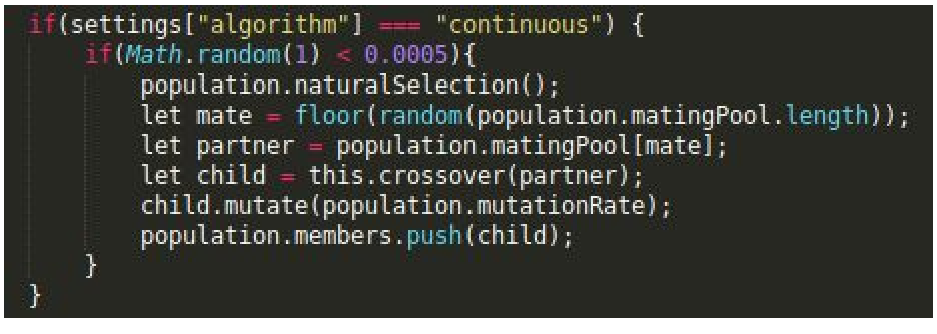
The first gene is the cell “Cell Wall Thickness”. The wall thickness can be any integer from 0-4. Genes with a higher value for cell thickness can consume genes with values lower than their own. while genes with a lower thickness produce energy more rapidly than their thicker counterparts. Genes with a higher thickness appear Darker in the simulation, while those who are lighter appear brighter.

The second Gene is “likeliness to cooperate” which is a random two decimal place value between 0 and 1. In order for two Diglets to cooperate(and join/create a Digrito) they must both pass a test in which their “likeliness to cooperate” value is measured against another randomly generated number between 0 and 1.



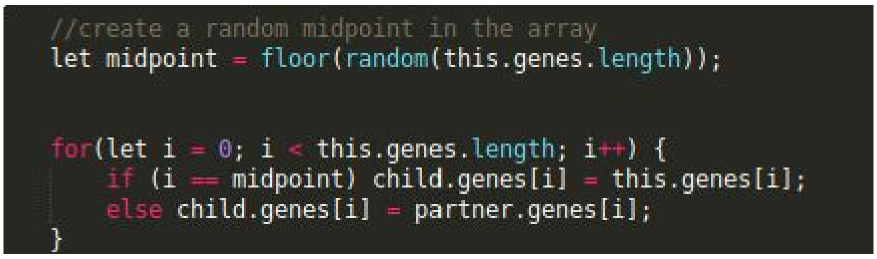
```
//If both diglets cooperate.
if(this.genes[1] > Math.round(Math.random() * 100) / 100) {
    if(digletArray[i].genes[1] > Math.round(Math.random() * 100) / 100){

        //if both diglets are in a digrito then continue.
        if(digritos.findGroup(this) != -1 && digritos.findGroup(digletArray[i]) != -1) {
            continue;
        }
        //if one diglet is in a digrito then add the new diglet to the digrito.
        else if(digritos.findGroup(this) != -1){
            digritos.addToDigrito(this, digletArray[i]);
        }
        else if(digritos.findGroup(digletArray[i]) != -1) {
            digritos.addToDigrito(digletArray[i], this);
        }

        //if neither Diglets are in a Digrito and they cooperate then make one.
        else {digritos.createNew(new Digrito([this, digletArray[i]]));}
    }
}
```

The update() function in Diglet.js is very important as it is responsible for performing all of the actions that can occur when Diglets come into contact with each other as well as updating their coordinates according to their velocity, and checking whether they’ve hit the corner of the screen. There’s too much in this function to take a single screenshot of, but one of the most important parts of this function is the block responsible for generating children when the “continuous” switch is active.



```
if(settings["algorithm"] === "continuous") {
    if(Math.random(1) < 0.0005){
        population.naturalSelection();
        let mate = floor(random(population.matingPool.length));
        let partner = population.matingPool[mate];
        let child = this.crossover(partner);
        child.mutate(population.mutationRate);
        population.members.push(child);
    }
}
```

The above block generates a random number between 0-1. If the number is less than 0.0005 then natural selection is performed on the population(ie. a mating pool is created). A mate is then chosen from the mating pool at random and then crossover occurs to produce a child which then has a small chance of undergoing mutation before entering the population.

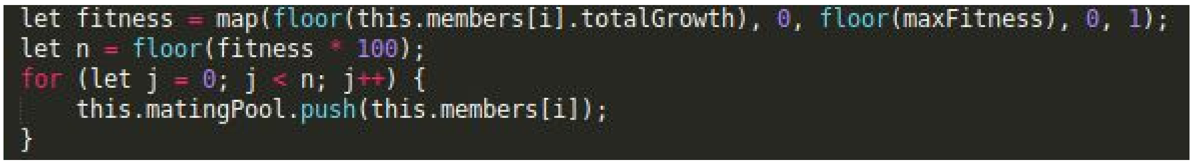


```
//create a random midpoint in the array
let midpoint = floor(random(this.genes.length));


for(let i = 0; i < this.genes.length; i++) {
    if (i == midpoint) child.genes[i] = this.genes[i];
    else child.genes[i] = partner.genes[i];
}
```

Above is a snippet of my crossover function. A midpoint is chosen as a random number between zero and the array length. In most genetic algorithms the midpoint would act as a “split” and every gene index below the midpoint would be inherited from one parent and indexes above the midpoint would be inherited from the other parent; this would be done using greater and less than operators. Because my implementation has only two genes I use equals instead(midpoint will always be zero or one.

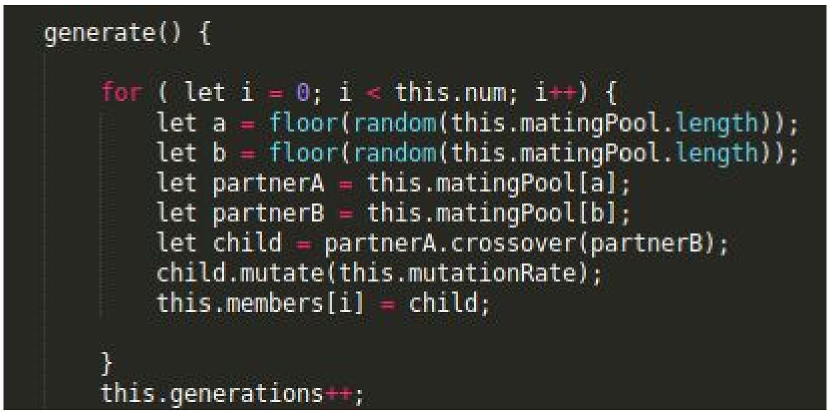
Population

Population.js contains all of the information and methods required to initialize the population. It also contains the natural selection and generate functions. The Natural selection function works by finding the highest value of fitness in the current population. For each individual the their fitness is mapped from “0 - highest fitness” to “0 - 1”, and then added to the mating pool array proportional to its fitness score. When mating occurs a random member of the mating pool is chosen to be a parent. The more times a Diglet is added to the mating pool the higher chance it has to become a potential mate and this is how Natural Selection is expressed in the simulation.



```
let fitness = map(floor(this.members[i].totalGrowth), 0, floor(maxFitness), 0, 1);
let n = floor(fitness * 100);
for (let j = 0; j < n; j++) {
    this.matingPool.push(this.members[i]);
}
```

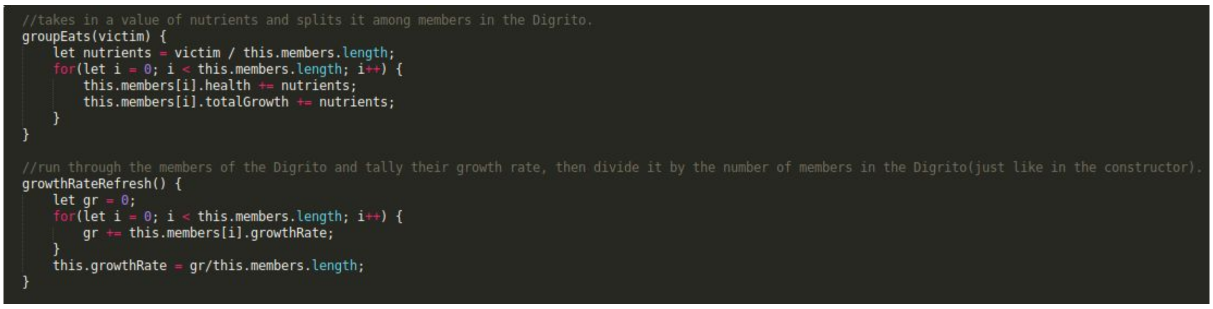
The Generation method in Population.js is the classic method for producing a new generation of solutions in any Genetic Algorithm. It selects two potential parents from the mating pool, performs crossover, mutates the child, and then adds them to the population.



```
generate() {

    for ( let i = 0; i < this.num; i++) {
        let a = floor(random(this.matingPool.length));
        let b = floor(random(this.matingPool.length));
        let partnerA = this.matingPool[a];
        let partnerB = this.matingPool[b];
        let child = partnerA.crossover(partnerB);
        child.mutate(this.mutationRate);
        this.members[i] = child;

    }
    this.generations++;
```

Digritos

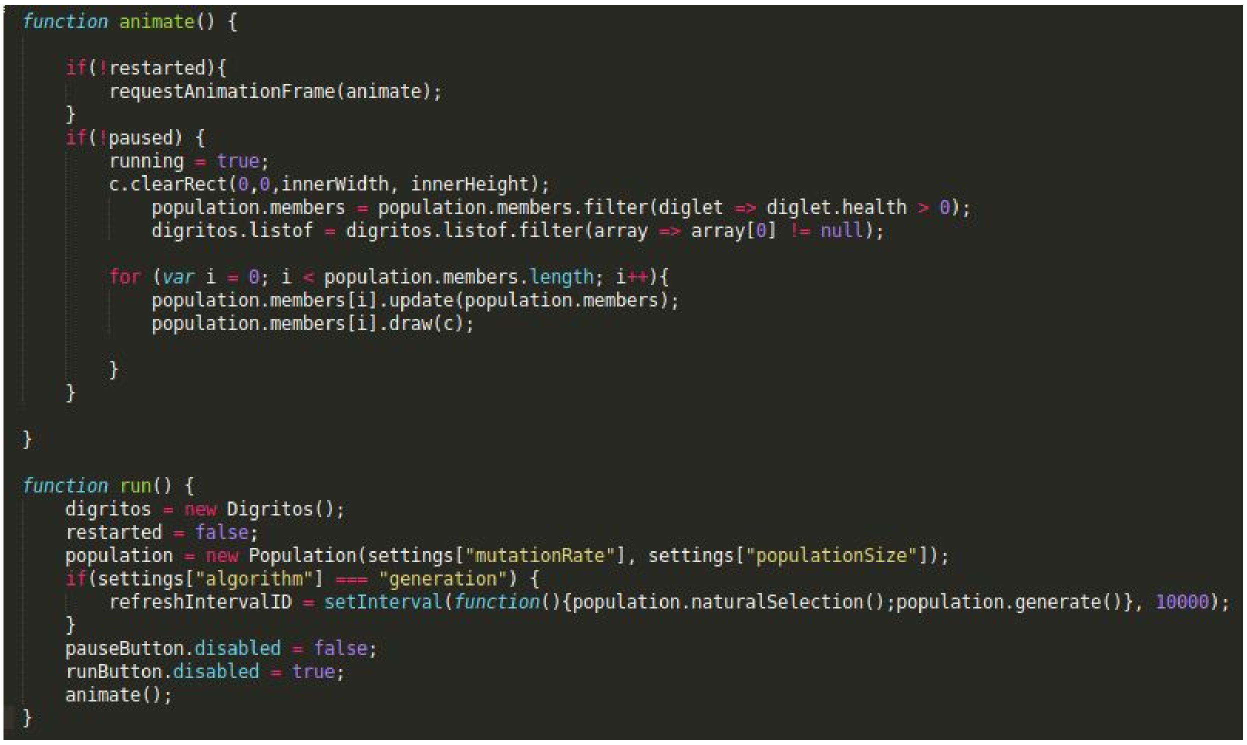
The Digrito.js file is where diglets join to make multicellular organisms and combine their growth rates, share consumption growth, etc.



```
//takes in a value of nutrients and splits it among members in the Digrito.
groupEats(victim) {
    let nutrients = victim / this.members.length;
    for(let i = 0; i < this.members.length; i++) {
        this.members[i].health += nutrients;
        this.members[i].totalGrowth += nutrients;
    }
}

//run through the members of the Digrito and tally their growth rate, then divide it by the number of members in the Digrito(just like in the constructor).
growthRateRefresh() {
    let gr = 0;
    for(let i = 0; i < this.members.length; i++) {
        gr += this.members[i].growthRate;
    }
    this.growthRate = gr/this.members.length;
}
```

Canvas and Index

Canvas.js is the file which is responsible for handing the canvas initialisation, as well as containing some geometry methods and variables used in the simulation. Index.html is the main page which contains the menu and some functions for passing the values from the menu into the Javascript code.



```
function animate() {

    if(!restarted){
        requestAnimationFrame(animate);
    }
    if(!paused) {
        running = true;
        c.clearRect(0,0,innerWidth, innerHeight);
            population.members = population.members.filter(diglet => diglet.health > 0);
            digritos.listof = digritos.listof.filter(array => array[0] != null);

        for (var i = 0; i < population.members.length; i++){
            population.members[i].update(population.members);
            population.members[i].draw(c);

        }
    }

}

function run() {
    digritos = new Digritos();
    restarted = false;
    population = new Population(settings["mutationRate"], settings["populationSize"]);
    if(settings["algorithm"] === "generation") {
        refreshIntervalID = setInterval(function(){population.naturalSelection();population.generate()}, 10000);
    }
    pauseButton.disabled = false;
    runButton.disabled = true;
    animate();
}
```

P5.js

P5.js is a library which I used to floor some numbers, generate some random numbers, and most importantly it does the mapping in my Natural Selection function.

3. Problems Solved

Problems solved involved learning all of the languages used in this project, which I had no prior experience with. The first thing I had to do was learn how to use HTML canvas, which started with drawing straight lines on the canvas and ended with being able to render hundreds of circles of random colors, coordinates, and velocities.These circles also needed to be able to detect other objects and the sides of the canvas and adjust their velocities accordingly.

I had never used Javascript before and though I knew in theory how a genetic algorithm worked in theory I had never coded one before. I had to learn the theory and then put it in to practice in a language I had never used before and so that was a challenge. The feedback for my Genetic Algorithm was displayed using my work on Canvas, which turned out to be a huge advantage as I was able to get rapid visual feedback when editing the Algorithm. Passing values from HTML menus to Javascript and handling Canvas elements as well as real time events was a trickier than expected.

After finishing the basic Genetic Algorithm creating groups of Diglets was also a challenge and creating groups of individuals on the canvas who moved and acted together added a lot more complexity than I initially thought it would, as synchronicity had to be maintained every frame.

4. Results and Future Work

The result of this project is a functional web hosted example of a Genetic Algorithm which implements features of game theory in a society of Single-celled organisms. It was found that while designing the framework of the Algorithm was a challenge in of itself the harder part was balancing the variables to produce a fair ecosystem.

Future work would include further balancing of the algorithm to provide a more stable ecosystem for the Diglets. Some options for new behaviours would also be a fresh improvement, such as option for children of Diglets attached to a Digrito to also be attached to that Digrito. I would also like the option for many more variables such as environment variables that affect energy generation and the behaviours of Diglets, as well as more genetic variation of Diglets. Additionally there are some existing variables that users do not have access to in the current simulation that I would like to add in the future.

5. References

https://p5js.org/

https://www.youtube.com/watch?v=bBt0imn77Zg

https://www.youtube.com/watch?v=ZpW_ojpmTWk