**SELF-DRIVEN CAR USING GENITIC ALGORITHM AND NEURON NETWORK**

Student: Hai Nguyen

Professor: Carl Mummert

1. **Introduction**

This program simulates an auto-driven car learning to drive through the track. It uses Neural Network to make decision of turning right or turning left and uses Genetic Algorithm to train the Neural Network; makes improvement each time the Car fails finishing the track.

1. **Source Code Explanation**
   1. **Framework**

JavaSE-1.8

* 1. **Program logics**
     1. **FPS and main flow**

Every computer has different speeds so we need a mechanism to normalize that; make the game runs at the same speed in any computer. We have 2 main methods: *update* and *render*. Usually, the game runs at 60fps so the *update* method will be called 60 times per second and the *render* will be called as fast as the computer’s speed.



* + 1. **Sequence Diagram of EntityManager’s *update* method**



* 1. **Graphics**
     1. **Screen class**



The Screen class handles all graphic renderings for the program. There are 2 ways to render in this program which are either by manipulating individual pixels or by using built-in functions of java library. For the map, manipulating individual pixels is used and for the other objects such as Car Sprite, Sensor Line and Sensor Circle, built-in functions of java are used to gain better graphics. This class has these main fields: *width, height, pixels, xOffset, yOffset* and *graphics.* If we want to manipulate individual pixel, we will change data in this *pixels* array*.* Two values *xOffset* and *yOffset* changes when the Car moves around. Because we always need to see the Car on screen so basically, the Car will stand at a fixed position on screen, we just need to adjust the offset of screen to make us feel like the Car is moving. There are 5 main functions in this class:

* + - 1. ***renderTile***

The *renderTile* method renders a Tile on screen. It requires arguments *x, y* coordinates of Tile and the Tile.

* + - 1. ***renderCar***

The *renderCar* method renders the Car object. It requires arguments *x, y* which are coordinates of the Car, the *angle* which the Car is heading and the Sprite for the image of the Car.

* + - 1. ***renderLine***

The method *renderLine* requires the arguments *color* and *x, y* for start point and end point to render the line between these points with a specific color.

* + - 1. ***renderCircle***

The method *renderCircle* requires arguments *x, y* coordinate for the center of circle, the radius *r* and the *color* for this Circle.

* + - 1. ***dispose***

The *dispose* method is used to dispose the object *graphics* to release the resource after rendering.

* + 1. **Tile class**



The map uses 16x16 tiles to tile the screen. There are 3 kind of tiles in this program: GrassTile, BrickTile and VoidTile and they are all inherited from Tile class. GrassTile and VoidTile are non-collided tiles; meanwhile, the BrickTile is collided. The BrickTile is used to build the track’s wall so the Car can detect the collision. The VoidTile will be used to cover places where there is nothing to render on screen.

* + 1. **TileCoordinate class**

This class is in charge of converting Tile coordinate to pixel coordinate. Each unit in Tile coordinate is equal to 16 pixels which is the size of Sprite in this program.

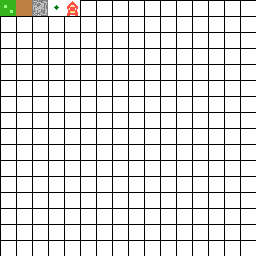
* + 1. **Sprite class**



Sprite object holds pixels of 16x16 image located in the Sprite Sheet. Sprite is used to design for objects in the program such as Car, Brick or Grass. Tile can either load Sprite image or just simply fill in itself by color. In this program, VoidTile is just simply filled in by a color meanwhile Brick, Gras and Car Sprite will load image from SpriteSheet. The background of Sprite which is WHITE color will not be render.

* + 1. **SpriteSheet class**

SpriteSheet contains multiple small 16x16 image of Sprites. The following image is the SpriteSheet.



* + 1. **Level class**



Level is a contract map of tiles. A level loads an image file, bases on the color of individual pixel to determine which Tile should be rendered. One pixel in the image will be replaced by 16 pixels of Tile in the program when it is rendered on screen. In this program, the color 0x00FF00 (Green) will represent the GrassTile and the color 0xFFD800 (Dark Yellow) will represent the BrickTile; any other colors that are not defined will be replaced by VoidTile. The SpawnLevel class is inherited from Level class and it overrides the method *loadLevel.* This method will load an image from the path and store data in the array *pixels.* After that, the colors on the image will be replaced by the Tile to render on screen. The following is the sample of a Level.



* 1. **Entity, Car and Mob class**



Entity is parent class of Car and Mob c­lass. It has 2 fields *x*, *y* coordinates and getters, setters for *x* and *y*.

Mob is inherited from Entity and has 2 functions which are *move* and *isCollided*. The Mob object can only move to a new position when there is no collision at that position. To detect collision, we will detect the Tile at that position. If the Tile is solid, then there is collision and vice versa. Mob also has a Sprite to render on screen.

Car is inherited from Mob class. It has information about *angle* of where the car is heading; *deltaDistance* which is about the distance that the car has just moved; 5 sensors EAST, NORTH EAST, NORTH, WEST and NORTH WEST have mission of detecting the distances from the car to the wall while the Car is moving around the track and *intersections* which are the intersected points between the sensor and the track. The intersected points will be represented by a yellow circle. After the distances are detected, they will be normalized and sent to neural network to get out decision. The neural network is attached to the car to receive information from 5 sensors. Main methods of Car entity:

* + 1. **buildFeelers**

This method will calculate the coordinates of the head and tail of each feeler. All the feelers’ tails will have the same coordinate with the car’s current position. Other feelers’ heads will be calculated based on the current heading *angle* of the car.

* + 1. **detectFeelerIntersection**

To detect the intersection between a feeler and the wall, first we build a line go through 2 points head and tail of that feeler. After having the line’s information, we will iterate from the tail to the head of that feeler. Here we need help from function *getTile* of Level object, we will pass the coordinates of the points on the line to this function to get out the Tile object. If the Tile object is not null and is solid, then we have intersection between the feeler and the wall right at this point.

* + 1. **update**

The car will constantly build the feelers, detect the intersections with the wall then send data to Neural Network. The Neural Network will call its *update* method then give out the information about left force and right force which help car turn right or left. The car uses this data to calculate the *angle* that it will turn, the *deltaDistance* that it will move then send to its *move* method.

* + 1. **render**

After 2 methods *buildFeelers* and *detectFeelerIntersection* are called, this method will use the output of these 2 method to draw the Car itself, its sensors’ lines and intersection circle.

* 1. **Inputs**

The Keyboard class is in charge of input; it implements the KeyListener interface. There are few key shortcuts in this program. ESCAPE key is to stop the program. R key is to restart the simulation. UP and DOWN key are to increase and decrease FPS. SPACE key is to force the genetic algorithm changing the population of genomes. N key is to change the map.

* 1. **Genetic Algorithm**

Genetic Algorithm is used to train neuron network, helps the neuron network give a better decision.

* + 1. **HelperFunction class**

This class contains static functions that will be use globally in program. It has method called *Sigmoid* which will normalize a value, give out another value between -1 and 1. *RandomSigmoid* method will make a random number between -1 and 1 and *getValueInRange* method will give out a value in a given range.

* + 1. **Genome class**



A genome has 3 information: *ID*, *fitness* and *weights*. The *fitness* is the distance which the car has finished without collision and *weights* is the list of random Sigmoid value which is from -1 to 1.

* + 1. **GeneticAlgorithm class**



This class has a list of Genomes called *population.*  Each *population* has good genomes with high fitness. Those genomes will be used to mixed up and mutated to create a new population of genomes. This class has the following main methods:

* + - 1. **generateNewPopulation**

This function will generate new *population* of genomes. The genomes’ *weights* will be randomly assigned Sigmoid value, *ID* and *fitness* will be assignedto 0.

* + - 1. **breedPopulation**

This function will generate a new *population* using old *population.* First, 4 genomes with high fitness will be chosen from the old *population*. After that, we will mutate and cross over 4 of them together and add to new *population*. The left positions of new *population* will be filled up by new random Genome.

* + - 1. **crossOver**

crossOver is a function that mixes up 2 genomes and create 2 other new genomes. To mix it, first we randomly choose the cross over point then split 2 genomes into 4 parts from that point then mix 4 parts together.

|  |  |
| --- | --- |
| Parent1: xxx xyyxx | Parent2: yxy xyxyx |
| Parent1: xxx(part1)-xyyxx(part2) | Parent2: yxy(part3)-xyxyx(part4) |
| Child1: xxx-xyxyx (new genome1) | Child2: yxy-xyyxx (new genome2) |

* + - 1. ***mutate* function**

Mutate is a function that randomly pick up a gene in a genome and assign a new value which is the sum of a random Sigmoid value multiplied by constant MAX\_PERMUTATION = 0.3 plus weight of that gene if the Sigmoid value is smaller than MUTATION\_RATE = 0.15.

|  |  |
| --- | --- |
| Parent1: xxx xyyxx | Parent2: yxy xyxyx |
| Parent1: xxx-xyyxx | Parent2: yxy-xyxyx |
| Child1: xxx-xyxyx | Child1: yxy-xyyxx |
| Mutated Child1: xxx-xyZyx | Mutated Child2: yxy-xyyxZ |

* 1. **Neural Network**
     1. **Neuron class**



Neuron is the basic element in Neuron Network. It has 2 fields which are the *numberOfInputs* coming to a neuron and values of those inputs which are called *weights*. In this program, neuron will receive data from genomes and stores into its *weights*. These *weights* will be used by neurons layers to evaluate output.

* + 1. **NeuronsLayer class**



NeuronsLayer contains a list of neurons. It will use those neurons’ *weights* to evaluate and give out the outputs. There are 2 types of neurons layer. One is Hidden Layer and one is Output Layer. These layers will be managed by Neurons Network. The main method of this class is *evaluate* method. In this method, we will sum up the value of *inputs* multiplied by neuron’s *weights* and add the value of last weight multiplied by a constant BIAS = -1. The purpose of BIAS value is to make sure the output is not 0. After that, the sum will be normalized by *Sigmoid* function and store in the *outputs.*

* + 1. **NeuralNetwork class**



NeuralNetwork class contains 1 Output Layer and 1 or many Hidden Layers as the following picture.

 These are the main methods of this class:

* + - 1. **setInput**

This method receives the inputs from car’s sensors and store them.

* + - 1. **getOutput**

This method receives the index and give out the data at that index.

* + - 1. **update**

The NeuralNetwork constantly receives data from the sensors of the car, passes that data to Hidden Layer to process then transfers output to the Output Layer for second processing. After that, the Output Layer will give out the decision back to the car to make turn.

* + - 1. **fromGenome**

This function will get *weights* from genome of GeneticAlgorithm to store in Neurons Layers.

1. **Program’s statistics**