Trabalho Prático 1

Jogo da Vida / DLA

Licenciatura:

Engenharia Informática e Multimédia

Unidade Curricular:

Modelação e Simulação de Sistemas Naturais

Docente:

Arnaldo Abrantes

Alunos:

Henrique Matos nº 48608

João Gonçalves nº 47507

Turma:

33D

Contents

Introdução

Este primeiro trabalho prático, dividido em duas partes (Jogo da Vida e DLA), pretende simular virtualmente processos de evolução de elementos em comunidades e a agregação de objetos caóticos num determinado espaço. Tais simulações foram feitas em Java, utilizando as ferramentas de Processing.

A primeira parte pretende representar o autómato celular, Jogo da Vida (criado em 1970 pelo matemático *John Horton Conway*), cuja finalidade seria estudar a evolução de comunidades ecológicas, aplicando as regras e princípios de ações genéticas e os seus processos de desenvolvimento, chegando à conclusão que estes são inseparáveis no que toca à sustentabilidade dum sistema biológico. Tal jogo explora as hipóteses do desenvolvimento de um sistema com base nos princípios de sobrepopulação e subpovoamento, sendo que em ambos os casos a célula não sobreviverá, necessitando duma vizinhaça sustentável, nem muito sufocante nem muito escassa.

A segunda parte do trabalho representa o processo de DLA (*Diffusion-Limited Aggregation*) cuja finalidade será representar a popular *Brownian Tree*, estudando a agregação de partículas com movimento caótico, num determinado espaço/ambiente cujas forças permitem tal agregação a partículas vizinhas num estado parado e agregado. O modelo de DLA é frequentemente utilizado para o estudo de crescimento dendrítico (ramificado), sendo que com a simulação da agregação de partículas é possível melhor compreender a criação de estruturas como, por exemplo, um floco de neve.

Jogo da Vida

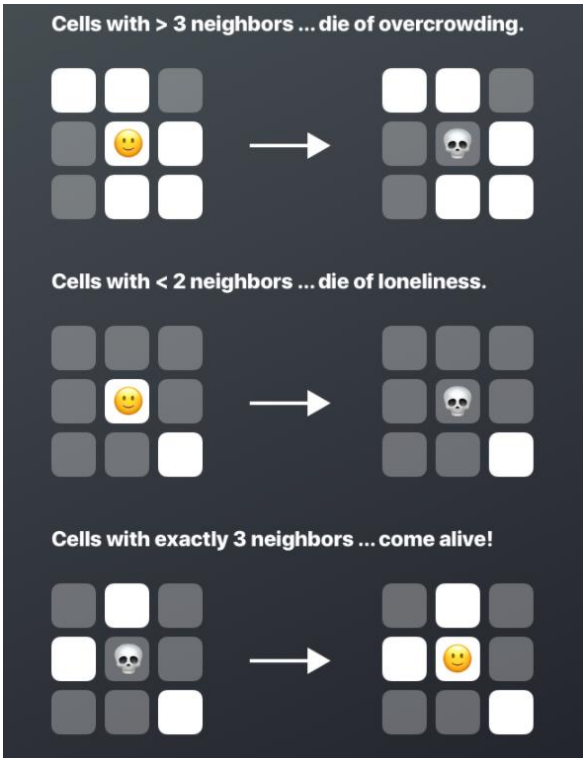
O Jogo da Vida (Conway's Game of Life) criado por John Horton Conway, um matemático britânico, é um jogo que não requer jogadores por ser um autómato celular, sendo apenas preciso uma configuração inicial que irá evoluir através do código implementado com as suas regras.

Cada célula do jogo pode estar "viva" ou "morta" e tem uma vizinhança 2D (geralmente seguindo o modelo de Moore) de raio definido e que irá influenciar o estado da própria célula.

As regras definidas são aplicadas a cada nova "geração" o que permite a sua evolução autónoma.

As regras do jogo são:

- Qualquer célula viva com mais de três vizinhos vivos morre de superpopulação
- Qualquer célula viva com menos de dois vizinhos vivos morre de solidão
- Qualquer célula morta com exatamente três vizinhos vivos se torna uma célula viva
- Qualquer célula viva com dois ou três vizinhos vivos continua no mesmo estado para a próxima geração



Para recriar este clássico jogo foram criadas três classes:

- A classe **Cell**
- A classe **CellularAutomata**
- A classe **GameofLife**

```
public class Cell {
    private int row, col;
    private int state;
    private Cell[] neighbors;
    private CellularAutomata ca;

    public Cell(CellularAutomata ca, int row, int col) {
        this.ca = ca;
        this.row = row;
        this.col = col;
        this.state = 0;
        this.neighbors = null;
    }
```

- Classe Cell

Esta primeira classe representa uma célula e é responsável por guardar os seus atributos individuais.

São definidas as seguintes variáveis:

row e **col** guardam a posição da célula

state indica se está viva ou morta

O array de células **neighbors** guarda a vizinhança da célula

Uma instância **ca** do objeto **CellularAutomata** é inicializada também

```
public class Cell {
    private int row, col;
    private int state;
    private Cell[] neighbors;
    private CellularAutomata ca;

    public Cell(CellularAutomata ca, int row, int col) {
        this.ca = ca;
        this.row = row;
        this.col = col;
        this.state = 0;
        this.neighbors = null;
    }

    public int getRow() {
        return row;
    }
    public int getCol() {
        return col;
    }


    public void setNeighbors(Cell[] neigh) { this.neighbors = neigh; }

    public Cell[] getNeighbors() { return neighbors; }

    public void setState(int state) { this.state = state; }

    public int getState() { return state; }

    public void display(PApplet p) {
        p.fill(ca.getStateColors()[state]);
        p.rect( a: col * ca.getCellWidth(),  b: row * ca.getCellHeight(),
                ca.getCellWidth(), ca.getCellHeight());
    }
```

É ainda implementado um construtor **Cell** da célula que irá ser utlizado pelo autómato celular (**CellularAutomata**) para criar cada célula.

A célula tem um estado inicial de morta, guarda a sua posição no campo de jogo e começa sem vizinhos.

Na restante parte da classe são implementados métodos para guardar e receber os vizinhos da célula (**setNeighbors** e **getNeighbors**), métodos para guardar e receber o estado da célula (**setState** e **getState**) e ainda um método **display** que está encarregue de representar a célula graficamente. Cada célula é um retângulo que será preenchido de cinzento se a célula estiver viva ou branco se estiver morta.

- Classe CellularAutomata

A classe **CellularAutomata** corresponde ao autómato celular e vai conter a informação do autómato e do seu funcionamento

Começando pelos atributos temos:

- **PApplet** – responsável pela representação gráfica
- **nrows** – número de linhas
- **ncols** – número de colunas
- **nStates** – número de estados possíveis (este é um autómato binário então terá apenas dois estados possíveis)
- **radiusNeigh** – raio da vizinhança
- **cells** – array bidimensional de células (grelha do jogo)
- **colors** – cores disponíveis para os estados das células
- **cellWidth** – largura de cada célula
- **cellHeight** – altura de cada célula

Em seguida temos o construtor da classe **CellularAutomata** que passa o número de linhas, o número de colunas, o número de estados e o raio da vizinhança.

```
public CellularAutomata(PApplet p, int nrows, int ncols, int nStates, int radiusNeigh) {
    this.nrows = nrows;
    this.ncols = ncols;
    this.nStates = nStates;
    this.radiusNeigh = radiusNeigh;
    cells = new Cell[nrows][ncols];
    colors = new int[nStates];
    cellWidth = p.width / ncols;
    cellHeight = p.height / nrows;
    createCells();
    setStateColors(p);
}
```

É inicializado o array bidimensional das células cells com tamanho [linhas] [colunas] e também o array **colors** que guarda as cores possíveis para os estados

São definidas as dimensões das células através de duas funções, **cellWidth** para a largura da célula, definida pela dimensão horizontal da aplicação (**p.width**) a dividir pelo número de colunas (**ncols**) e **cellHeight** para a altura da célula, definida pela dimensão vertical da aplicação (**p.height**) a dividir pelo número de linhas (**nrows**).

É no construtor também que é chamada a função **createCells** que cria as células do jogo de acordo com as dimensões do jogo e ainda a função **setStateColors** que define as cores que correspondem a cada estado.

De seguida temos as funções **getCellWidth** e **getCellHeight** usadas para ser possível receber as dimensões das células para uso noutros métodos

Temos um método **setStateColors** que irá fazer corresponder cores a cada estado possível das células. Com a configuração atual temos que ao estado 0, portanto morto, corresponde a cor branca e ao estado 1, para as células vivas, corresponde uma cor cinzenta-escura. O método seguinte é, portanto, **getStateColors** para receber esse array de cores disponíveis.

Definimos também um método **createCells** encarregue de criar as células a ser usadas no jogo, tendo em atenção as dimensões **nrows** e **ncols** da grelha do jogo. Este método chama ainda no final o método **setMooreNeighbors** que define a vizinhança das células.

O método **initRandom** é o responsável pela inicialização da grelha de jogo com as células no seu estado inicial.

Foram feitas alterações adicionais que deixam o utilizador escolher a codificação inicial que prefere.

É pedido o input do utilizador e, se este inserir o número 1 a grelha de jogo será inicializada com as células definidas com um estado aleatório. Se o utilizador pretender ter mais controlo sobre o resultado pode inserir o número 2 o que irá inicializar a grelha com as células todas mortas, sendo possível depois a sua alteração através de um método a ser definido na classe **GameofLife**.

```
public int getCellWidth() { return cellWidth; }

public int getCellHeight() { return cellHeight; }

private void setStateColors(PApplet p) {
    colors[0] = p.color( gray: 255);
    colors[1] = p.color(50,50,50);
}

public int[] getStateColors() { return colors; }

private void createCells() {
    for (int i = 0; i < nrows; i++) {
        for (int j = 0; j < ncols; j++) {
            cells[i][j] = new Cell( ca: this, i, j);
        }
    }
    setMooreNeighbors();
}

public void initRandom() {
    Scanner input = new Scanner(System.in);
    System.out.println("Opção 1: Início random");
    System.out.println("Opção 2: Início personalizado");
    System.out.print("Codificacao: ");
    int number = input.nextInt();
    if(number == 1) {
        for (int i = 0; i < nrows; i++) {
            for (int j = 0; j < ncols; j++) {
                cells[i][j].setState((int) (2* Math.random()));
            }
        }
    }
    if(number == 2) {
        for (int i = 0; i < nrows; i++) {
            for (int j = 0; j < ncols; j++) {
                cells[i][j].setState(0);
            }
        }
    }
    System.out.println("Prima qualquer tecla para iniciar");
}
```

Depois criámos o método **pixel2Cell** que permite retornar a posição em pixéis da célula onde estamos a clicar com o rato.

O método seguinte, **setMooreNeighbors**, chamado após serem criadas as células, é responsável por definir a vizinhança de Moore das células. É percorrido a grelha inteira e, para cada célula individual é definida uma vizinhança de raio 1.

r=1

```
public int[][] activeCells(PApplet p){
    int [][]nActive = new int[nrows][ncols];
    for (int row = 0; row < nrows; ++row){
        for (int col = 0; col < ncols; col++){
            Cell[] neighbors = cells[row][col].getNeighbors();
            int count = 0;

            for (Cell neighbor : neighbors) {
                count += (neighbor.getState() == 1) ? 1 : 0;
            }
            nActive[row][col] = (cells[row][col].getState() != 1) ? (count) : (count - 1);
        }
    }
    return nActive;
}

public void regras(PApplet p) {

    int[][] livingCells = activeCells(p);
    for (int row = 0; row < nrows; ++row){
        for (int col = 0; col < ncols; col++){
            if (livingCells[row][col] == 3 || (cells[row][col].getState() >= 1 && livingCells[row][col]
                cells[row][col].setState(1);
            }
            else {
                cells[row][col].setState(0);
            }
        }
    }
}
```

O método **activeCells** é usado para perceber quais células estão vivas e quais estão mortas e guardar as células vivas num array bidimensional.

O método anterior é importante para que o método **regras**, que define as regras do jogo, funcione.

Utilizando o array obtido através do método **activeCells** este método vai correr a grelha do jogo e, para cada célula, vai aplicar as condições para estar viva ou morta e vai definir o próximo estado correto a cada célula.

As células com 3 vizinhos vivos (independentemente do estado) ou as células vivas com 2 vizinhos vivos vão estar vivas. Caso contrário vão estar mortas.

O último método desta classe é o método **display** que é responsável pela representação das células, chamando para cada célula do array o método **display** da classe anterior **Cell**, atualizando cada célula individualmente.

- Classe GameofLife

A classe **GameofLife** implementa a interface do Processing **IProcessingApp** e dá override aos seus métodos.

São definidos os atributos **nrows** e **ncols** que dão a dimensão da grelha de jogo, **nStates** que define o número de estados que as células podem ter, **radiusNeigh** que define o raio da vizinhança das células, uma instância **ca** de **CellularAutomata** e um variável booleana **start** responsável pelo início do jogo e a continuação e pausa do mesmo.

```
@Override
public void setup(PApplet p) {
    ca = new CellularAutomata(p, nrows, ncols, nStates, radiusNeigh);
    this.start = false;
    ca.initRandom();
}

@Override
public void draw(PApplet p, float dt) {
    p.frameRate( fps: 10);
    ca.display(p);
    if (this.start){
        ca.regras(p);
    }

}

@Override
public void mousePressed(PApplet p) {
    Cell cell = ca.pixel2Cell(p.mouseX, p.mouseY);
    if (cell.getState() == 1) {
        cell.setState(0);
    } else {
        cell.setState(1);
    }
}

@Override
public void keyPressed(PApplet p) {
    this.start = !this.start;
    if (!this.start){
        System.out.println("PAUSED");
    }
    else{
        System.out.println("IN PROGRESS");
    }
}
```

O método **setup** cria uma nova instância de **CellularAutomata** com os atributos **p**, **nrows**, **ncols**, **nStates** e **radiusNeigh**.

A variável **start** é tornada false e é chamado o método **initRandom** da classe **CellularAutomata** para criar a grelha inicial.

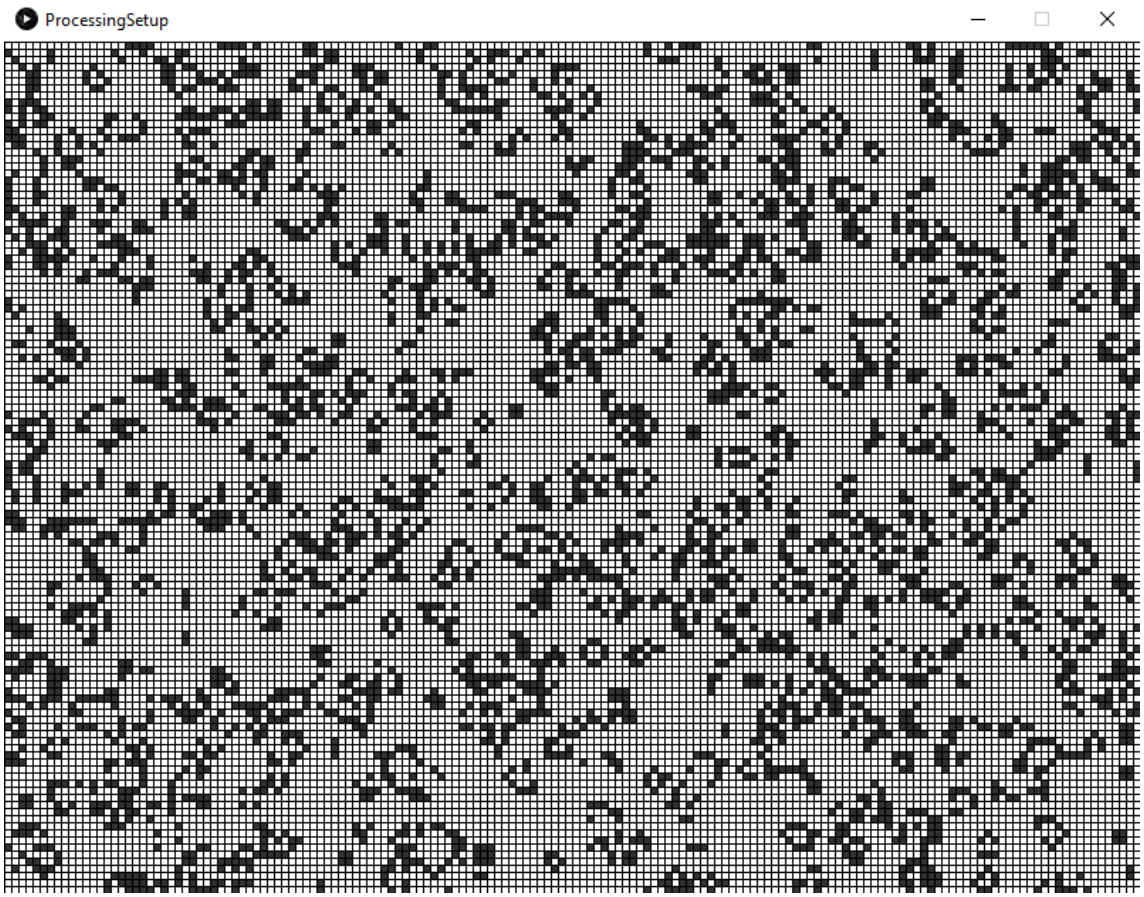
O método **draw** é utilizado para "jogar", definindo o framerate da aplicação e chamando o método **display** de **CellularAutomata** para mostrar a grelha inicial com os estados definidos das células.

Este método **regras** chamado no método **draw** vai constantemente autalizar a grelha de jogo enquanto a variável **start** for verdadeira.

O método **mousePressed** é usado para, através do uso do método **pixel2Cell** da classe **CelullarAutomata**, alterar a célula selecionada através de um clique do rato. Qualquer que seja o estado da célula, se houver um clique do rato esta muda de estado. Esta habilidade pode ser usada durante o jogo no seu início e depois sempre que este estiver pausado.

O último método **keyPressed** deteta se alguma tecla foi pressionada e se sim altera o estado de jogo entre estar em pausa ou estar a correr, podendo ser usado sempre que se quiser.

Este jogo vai correr através da classe **ProcessingSetup** do programa **Processing** e corre como uma **app** no método **main** dessa classe.



Diffusion-Limited Aggregation (DLA)

Dissecando o nome *Diffusion-Limited Aggregation*, entendemos que *Diffusion* refere-se ao movimento difuso e caótico das particulas envolventes, até ao momento em que se agregam umas às outras (*Aggregation*). A junção *Diffusion-Limited*, refere-se ao facto das particulas estarem em baixas concentrações de modo a que não entrem as particulas em contacto umas com as outras, assegurando que a estrutura cresce num todo em vez de ser uma junção de aglomerados, permitido que seja formada a ramificação da mesma.

Para estudar este processo de agregação, criou-se duas classes: a classe **Walker** e a classe **DLA**.

- Classe Walker

Começaremos então com a classe Walker. Esta classe é responsável pelo comportamento individual da particula (Walker), possuindo um construtor e tratando métodos que atualizam o estado do Walker em si e da colisão.

Definimos as variáveis **pos** (posição), **state** (estado), **colour** (cor), **radius** (raio), **num_wanders** (numero de walkers em movimento), **num_stopped** (numero de walkers parados).

```
public Walker(PApplet p){
    //pos = new PVector(p.random(p.width), p.random((p.height)));
    pos = new PVector( x: p.width/2,  y: p.height/2);
    PVector r = PVector.random2D();
    r.mult(p.width/2);
    pos.add(r);
    setState(p, State.WANDER);
}
```

No construtor, definimos a posição inicial do primeiro **Walker**, sendo esta no meio da janela, e definimos o estado como “Wander” para que possa haver movimento nos adjacentes. Definimos também um vetor para orientar a forma da força da agregação, atribuindo o vetor à posição inicial dos walkers, para que se desloquem com o formato de movimento pretendido. Este construtor, tal como os métodos descritos abaixo, passam o argumento **Papplet**, já que esta é a classe responsável pelo tratamento gráfico de todos os elementos em Processing.

Definimos um segundo construtor **Walker**, passando-se como argumento o Papplet e o vetor **pos**. Este construtor é responsável pelo Walker inicial, que se inicia parado, tal como se define com a função *setState*, que será explicada abaixo. Este Walker inicial será o catalizador para a formação ramificada da estrutura simulada.

Definindo então o método **setState**, passamos como argumento **PApplet**, e o enum **State**, que define o estado do Walker. Este estado, usa uma das duas constantes (já que um **enum** é uma

coleção de constantes definidas pelo utilzador). Dentro deste método também se estabeleceu um limite através de intervalos de Walkers parados, em que a cor muda a cada intervalo. O métodod também conta o número de Walkers parados e em movimento.

Desenvolveu-se também um método **getState** de modo a obter o estado do Walker.

Para atuailizar o estado, criou-se o método **updateState**, passando dois argumentos em que um é o **PApplet** e o outro é uma lista de Walkers. Este método irá atualizar o estado do Walker através de deteção de colisão. Através de um ***if***, caso o estado seja já parado, não será necessário fazer nada. Após esta verificação, cria-se um ***for loop*** que cria e percorre a lista de Walkers, adicionado cada Walker à lista, adicionando dentro do loop um ***if*** que define a distância necessária para a deteção de colisão. Caso o Walker esteja em movimento e toque num já parado, muda de estado para **STOPPED**, daí o nome de updateState do método.

Existe também o método **wander** que passa o argumento **PApplet**, responsável pelo movimento dos Walkers, definindo as direções e posições de todos os elementos. Este método faz uso do vetor de forças, usando uma classe do processing chamada **lerp()** que permite modificar o centro de ação e a velocidade de aglomeração (*stickiness*).

Por fim, o último método da classe **Walker** é o método **display** que passa o argumento **PApplet**, que serve para demonstrar os walkers criados, que terão uma forma circular e a cor que será definida pelo programador.

- Classe DLA

Esta classe é responsável pela aglomeração dos Walkers, definidos na classe anterior. Passa as variáveis duma lista de walkers, o número de walkers (**NUM_WALKERS**) e o numero de ações por frame de evento (**NUM_STEPS_PER_FRAME**).

Estabelece-se o método **setup** que passa o argumento **PApplet** em que se define a lista de walkers a ser usada e adiciona-se um walker, criado com o objeto Walker, passado através da classe **Walker**. Cria-se e enche-se esta lista de walkers (passada através dum ArrayList), baseando-se no número de walkers, definido na variável **NUM_WALKERS**.

```
@Override
public void setup(PApplet p) {
    walkers = new ArrayList<Walker>();
    Walker w = new Walker(p, new PVector( x: p.width/2, y: p.height/2));
    walkers.add(w);

    for(int i = 0; i < NUM_WALKERS; i++){
        w = new Walker(p);
        walkers.add(w);
    }

}
```

O método responsável pela concatenação de todos os métodos, dando origem à visualização da simulação das particulas é o método **draw** que passa or argumento **PApplet** e um float **ft**. Este método define a cor de fundo da janela, que neste caso é preto, definida através da classe background em PApplet, com o valor de RGB 0. Aqui definimos uma variável “parados” que será explicada de seguida.

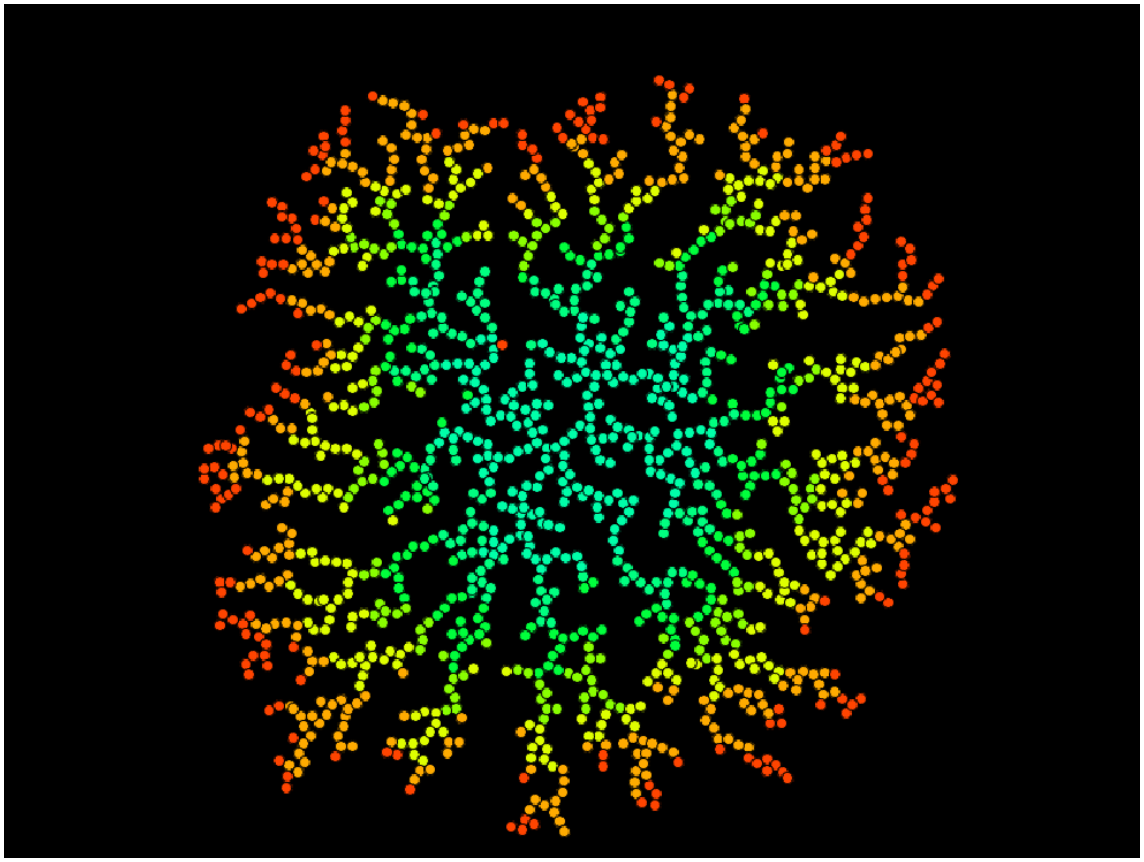
Possui um ***for loop*** com um outro ***nested for loop*** que serve para, caso o estado seja em movimento (WANDER), damos update ao estado do walker. Caso o estado seja parado (PARADO), criamos um segundo ***if*** que caso o numero de walkers parados seja menor que 1200, continuar-se-à a contar o numero de walkers parados, sendo para isso que serve a variável definida anteriormente “parados”.

Fora deste ***for loop*** principal, criamos um novo ***for loop*** que, por cada walker parado, adiciona-se um novo walker à lista de Walkers, para que por cada walker parado, adicione-se um novo, tendo sempre um número de walkers em movimento constante.

Usamos também um outro ***for loop*** que percorre todos os walkers na lista de walkers, para que possamos dar display a todos os walkers, através do método **display** da classe **Walker**.

Por fim, damos ***print*** ao numero de walkers em movimento e parados na consola.

Assim sendo, eis o resultado final:



Temos então acima o resultado final da simulação de particulas, formando uma ***Brownian Tree***, finalizando assim o DLA.