Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia
Informática e Multimédia (LEIM)

MSSN – 21/22

Projecto Final

Professores: Eng.º Arnaldo Abrantes e Eng.º Paulo Vieira

Trabalho realizado por:	número
Miguel Ginga	46292

Lisboa, 20 de Fevereiro de 2022

Índice de matérias

Índice de Imagens

1. Introdução / Objectivos

O projecto final da unidade curricular Modelação e Simulação de Sistemas Naturais consiste em aplicar todos os conhecimentos adquiridos ao longo do semestre e consolidá-los numa aplicação de escolha livre. Sendo assim a escolha caiu sobre a simulação de um ecossistema em JAVA/Processing tendo como base o código desenvolvido nas aulas. Foram introduzidas diversas alterações como diferentes presas com comportamentos diferentes, mutações e uma vertente interactiva.

Num ecossistema teremos então um terreno constituído por vários componentes, entre eles, terreno fértil, alimento, obstáculos e teremos uma população constituída por presas, predadores e, neste caso em particular, a opção de adicionar uma componente humana, um caçador.

Todos estes elementos da população irão ter os mesmos comportamentos, embora os exibam de forma diferente, sendo estes Movimentação, Alimentação, Reprodução e Morte. Isto garante que temos um ecossistema que irá evoluir durante o tempo de acordo com os critérios escolhidos para estes comportamentos.

2. Narrativa da Aplicação

Tal como referido na introdução um ecossistema será composto por um terreno e uma população de agentes que variam entre presas e predadores.

O nosso ecossistema será composto por um terreno que contém áreas de terra fértil e terra desértica, significando que vamos ter terrenos de terra onde irá crescer alimento e outra nunca se irá verificar esta ocorrência. As plantas que crescem no terreno fértil irão gerar frutas que têm a capacidade mutar as presas do terreno, torando-as mais ágeis, sendo que este efeito apenas é verificado na primeira vez que consomem a fruta.

No terreno também teremos áreas aquáticas, especificamente lagos. Nos lagos podemos verificar a existência de peixes que podem ser consumidos pelas presas que voam.

Passando à população do nosso ecossistema teremos então:

- Prey : presas terrestres que vagueiam o terreno, evitando obstáculos e que sofrem uma mutação consumindo a fruta das plantas.

-FlockPrey: presas que voam e têm a capacidade consumir peixes que vivem no lago, tal como se alimentam de fruta também.

-Predator: predadores que se alimentam de presas terrestres, têm a capacidade de atravessar obstáculos, mas perdem velocidade ao fazê-lo. Não só não consomem peixe por não fazer parte da sua alimentação porque não têm as ferramentas necessárias para os consumir, tal como não o conseguem fazer porque embora consigam atravessar os lagos não têm a capacidade motora para caçar peixes enquanto o fazem.

Este ecossistema pressupõe que a morte e nascimento dos agentes está ligada a energia ganha através de consumo de alimento, seja este plantas e frutas para as presas ou presas para os predadores, não havendo afixado um tempo limite para uma morte natural. Um agente ganha energia ao consumir um alimento, se tiver energia suficiente irá se reproduzir, já se a sua energia chegar a 0 o agente morre. Ou seja, quando mais alimento houver, maior será taxa de natalidade e menor será a de mortalidade e no sentido inverso, quanto menor o número de alimentos maior será a taxa de mortalidade e a taxa de natalidade. Isto significa que neste tipo de sistema vamos acabar por ver um ponto de estabilidade para onde as populações irão tender.

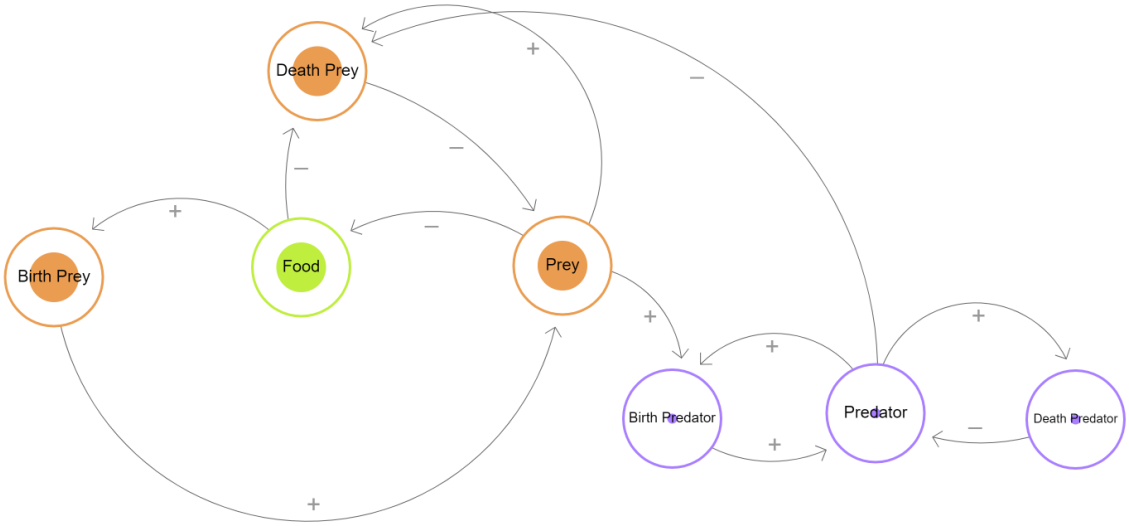


Imagem 1 – Diagrama Causal de Prey e Predator

Através do diagrama causal podemos observar como deverá progredir a evolução da população das presas e predadores ao longo do tempo.

A presa ao alimentar-se, diminui a comida do terreno e provocando o nascimento de novas presas. Quando mais alimento tivermos, maior será a taxa de natalidade e por sentido inverso menor será a taxa de mortalidade porque existe alimento para as presas apenas morrerem de forma natural e não à fome. O aumento da natalidade significa o aumento do número de presas o que acaba por levar à diminuição da comida disponível e consequente diminuição da natalidade e aumento da mortalidade. Ou seja, estamos na presença de um ciclo de feedback negativo em que se procura antigir um nível de estabilidade entre o número de presas e a comida disponível. Verificamos novamente esse tipo de feedback quando olhamos para Prey, Food e Death Prey, garantido a procurar de estabilidade do sistema que referimos.

Quando olhamos para a presa verificamos que quanto mais presas tivermos, maior será a natalidade das presas, provocando tanto um aumento no número de presas como consequentemente da taxa de mortalidade.

Ao englobarmos o efeito que os predadores têm no aumento da taxa de mortalidade da presa verificamos novamente um ciclo de feedback negativo que procura o equilíbrio.

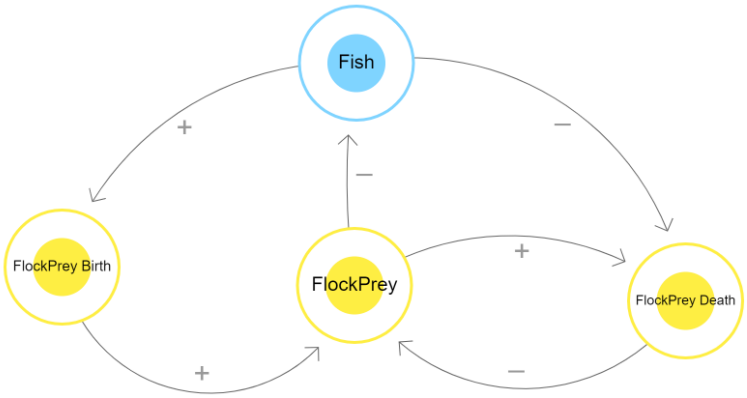


Imagem 2 – Diagrama Causal de FlockPrey e Fish

No diagrama que engloba a evolução do FlockPrey podemos novamente ver a presença dos ciclos negativos que levam à estabilização da população da espécie. Quanto maior for o número de alimento, menor será a taxa de mortalidade e maior será a taxa de natalidade, no entanto se tivermos uma população muito expressiva a quantidade de alimento diminui, diminuindo a taxa de natalidade e aumentando a taxa de mortalidade. Eventualmente iremos chegar a um ponto em que todos este elementos tendem para um valor estável.

3. Apresentação da Aplicação

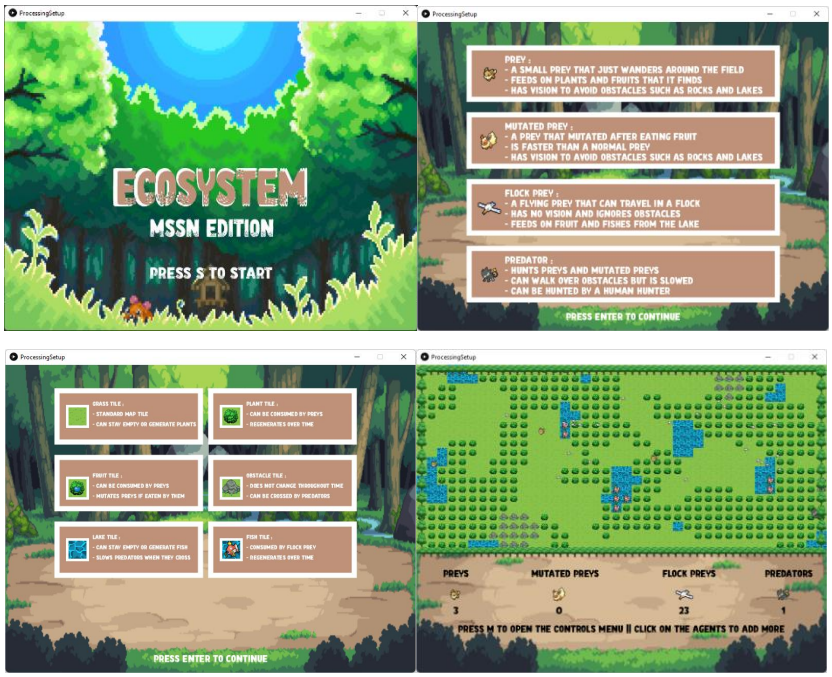


Imagem 3 – Inicialização da Aplicação

Quando iniciamos a aplicação são apresentados dois menus iniciais que irão descrever as componentes da simulação ao utilizador. O primeiro irá indicar quais os agentes que farão parte da simulação e quais os seus principais comportamentos. O segundo irá descrever as especificidades de cada tipo de terreno. Estes serão temas que iremos abordar posteriormente neste relatório.

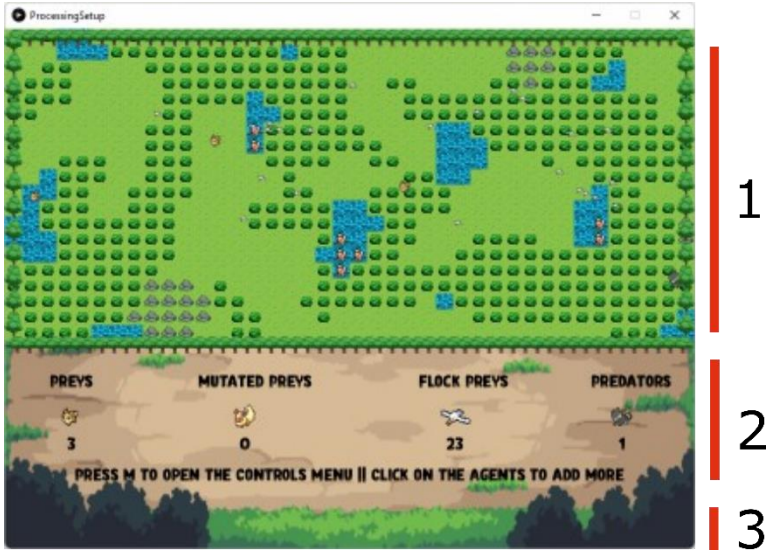


Imagem 4 – Viewports da Aplicação

Na imagem 2 temos assinalada a respectiva divisão da janela de aplicação em Viewports. Iremos ter no total 3 Viewports:

- Viewport 1: Composto pelo tabuleiro principal da aplicação onde terá todo o terreno e agentes a evoluir ao longo do tempo.
- Viewport 2: Irá mostrar o número actual de cada agente presente no terreno, sendo também possível clickar em cada um destes agentes para os adicionar.
- Viewport 3: Embora não esteja visível este viewport irá conter uma barra de energia associada ao agente controlado pelo utilizador, sendo que esta apenas é visível quando este agente estiver presente no terreno.

A aplicação terá ainda as seguintes funcionalidades associadas às teclas indicadas:

- M – abre um menu que exibe todos os controlos da aplicação
- C- abre o menu de agentes que contém informação sobre os mesmos
- T – abre o menu do terreno que contém informação sobre o mesmo
- R – reinicia a aplicação, gerando um novo terreno
- P – inicializa um agente que será controlado pelo utilizador ao clickar no terreno

4.Desenvolvimento da Aplicação

O desenvolvimento da aplicação foi realizado num ambiente JAVA, tendo como base as classes de ecossistemas desenvolvidas ao longo da aula. De seguida iremos abordar as novas classes criadas e alterações feitas às classes existentes ao longo do desenvolvimento da aplicação, tendo este desenvolvimento em conta a narrativa do ecossistema e os comportamentos adequados para cada agente e tipo de terreno.

Antes de iniciar esse processo, primeiro iremos descrever os fundamentos da aplicação. O terreno será gerado de forma procedimental através de um autómato celular que segue a regra da maioria. Os agentes serão boids com comportamentos específicos associados. A aplicação foi desenvolvida usando exclusivamente as bibliotecas gerais de JAVA e do Processing.

4.1. Agentes

4.1.1 Prey

A presa (Prey) será o nosso agente mais comum num ecossistema, alimenta-se à base de plantas e é caçado por predadores (Predator). O construtor deste agente não sofre alterações relativamente às classes que lhe dão origem, excepto num ponto específico. Sendo que a posição da nossa presa é gerada aleatoriamente, e esta tem como comportamento evitar obstáculos, fará sentido esta não nascer no topo de um obstáculo.

```
Patch patch = (Patch) terrain.world2Cell(pos.x, pos.y);
while (patch.getState() == WorldConstants.PatchType.OBSTACLE.ordinal()
        || patch.getState() == WorldConstants.PatchType.LAKE.ordinal()) {
    pos = new PVector(parent.random((float) window[0], (float) window[1]),
            parent.random((float) window[2], (float) window[3]));
    patch = (Patch) terrain.world2Cell(pos.x, pos.y);
}
```

Este troço de código, que será adaptado e reutilizado ao longo do desenvolvimento da aplicação, irá verificar em parte do terreno estará o agente, sendo que se tiver num tipo de terreno que equivale a um dos obstáculos, será gerado um novo vector de posição até que este seja válido.

Outra ligeira alteração que foi feita, de modo que o comportamento de evitar obstáculos se tornasse mais consistente principalmente, foi a ampliação da visão de segurança do agente por 50%.

```
sprite = parent.loadImage("../sprites/eevee.png");
```

Algo que será comum a todos os agentes, será a componente *sprite*, que é um objecto do tipo PImage que irá carregar a imagem representativa de cada agente.

A presa tal como já referido alimenta-se de plantas e das frutas que crescem nas plantas. Para este efeito iremos então reutilizar o código anteriormente referido em que verificamos qual será o tipo de terreno onde se encontra o agente.

```
@Override
public void eat(Terrain terrain) {
    Patch patch = (Patch) terrain.world2Cell(pos.x, pos.y);
    if (patch.getState() == WorldConstants.PatchType.FOOD.ordinal()) {
        energy += WorldConstants.ENERGY_FROM_PLANT;
        patch.setFertile();
    }
    if (patch.getState() == WorldConstants.PatchType.FRUIT.ordinal()) {
        energy += WorldConstants.ENERGY_FROM_PLANT;
        if (!mutate) {
            dna.dnaMutate();
            mutate = true;
            color = parent.color(WorldConstants.PREY_MUTATE_COLOR[0], WorldConstants.PREY_MUTATE_COLOR[1],
                    WorldConstants.PREY_MUTATE_COLOR[2]);
            sprite = parent.loadImage("../sprites/flareon.png");
            sprite.resize(0, 20);
        }

        patch.setFertile();

    }
}
```

Se o tipo de terreno corresponder ao da planta, a presa alimenta-se aumentado a sua energia e alterando o estado do terreno para fértil.

Se a presa consumir a fruta, podemos então ver que o seu DNA irá sofrer uma mutação, traduzida pelo seguinte método na classe DNA:

```
public void dnaMutate() {
    maxSpeed += random(0.3f,0.3f);
}
```

Iremos então observar um aumento da velocidade da presa, e em termos representativos, esta sofrerá também uma alteração no seu visual, efectuado através dos comandos associados ao objecto sprite presentes no troço de código apresentado. E este será o processo que origina as referidas “Mutated Prey”.

```
public Prey reproduce() {
    Prey child = null;
    if (energy > WorldConstants.PREY_ENERGY_TO_REPRODUCE) {
        energy -= WorldConstants.INI_PREY_ENERGY;
        child = new Prey(this, parent, plt);
        if (this.mutate) {
            child.sprite = parent.loadImage("../sprites/flareon.png");
            child.sprite.resize(0, 20);
            child.mutate = true;
        }

    }
    return child;
}
```

Relativamente à reprodução da presa, a única alteração feita aqui, é que se o progenitor for uma presa que sofreu uma mutação, a cria será também uma presa mutada.

```
public Prey reproduceNowPrey() {
    Prey child = new Prey(this, parent, plt);
    return child;

}

public Prey reproduceNowMutatedPrey() {
    Prey child = new Prey(this, parent, plt);
    child.sprite = parent.loadImage("../sprites/flareon.png");
    child.sprite.resize(0, 20);
    child.mutate = true;
    return child;

}
```

Os métodos reproduceNowPrey e reproduceNowMutatedPrey têm como propositado gerar novas presas sem estar preso às restrições de energia que uma presa normal teria. Estes métodos irão servir para gerar novas presas através da interface da aplicação.

4.1.2 FlockPrey

A classe FlockPrey será uma implementação da classe AnimalFlock que por sua vez implementa a interface IAnimal. A classe AnimalFlock não será nada mais de que um clone da classe Flock desenvolvido nas aulas, mas alterado para trabalhar com Animal em vez de Boid.

A principal alteração em termos de métodos, será um método que cria um novo FlockPrey usado no método reproduce deste agente.

```
//gerar um novo animal
public Animal addAnimal() {

    double[] w = plt.getWindow();
    float maxSpeed = p.random(2, 4);
    float x = p.random((float) w[0], (float) w[1]);
    float y = p.random((float) w[2], (float) w[3]);
    Animal b = new Animal(new PVector(x, y), mass, radius, color, p, plt);
    b.sprite = p.loadImage("../sprites/wingull.png");
    b.sprite.resize(0, 10);
    b.getDNA().visionDistance = p.random(10, 10);
    b.getDNA().maxSpeed = maxSpeed;
    b.addBehaviour(new Separate(sacWeights[0]));
    b.addBehaviour(new Align(sacWeights[1]));
    b.addBehaviour(new Cohesion(sacWeights[2]));
    b.energy = WorldConstants.INI_FLOCK_ENERGY;
    return b;
}
```

Relativamente á classe FlockPrey, irá funcionar de forma muito semelhante à classe Prey, sendo que para cada comportamento, eat e reproduce, terá de ser feito um ciclo auxiliar para verificar estes comportamentos para cada agente pertencente ao flock.

4.1.3 Predator

O predador tal como a presa irá extender a classe Animal desenvolvida nas aulas. A principal alteração feita no seu construtor é o aumento da sua velocidade, sendo que é um agente predatório que requer mais energia para se reproduzir e que terá de perseguir as presas para se alimentar.

```
public void eat(Prey prey, Terrain terrain, Population population) {
    Patch patch = (Patch) terrain.world2Cell(pos.x, pos.y);
    if(patch.preyInPatch(prey, terrain)) {
        energy +=  WorldConstants.ENERGY_FROM_PREY;
        population.preyEaten(prey);
    }
}
```

Enquanto as presas verificavam em que tipo de terreno se encontravam para se alimentarem, os predadores têm de verificar se a presa se encontra no mesmo Patch(célula) do terreno que eles. Para este feito foi criada uma classe auxiliar em Patch.

```
//método que recebe uma presa e verifica a sua posição no terreno
//retorna true se algum dos patch das presas for equivalente ao patch actual
public boolean preyInPatch(Body prey, Terrain terrain) {
    Patch patch = (Patch) terrain.world2Cell(prey.getPos().x, prey.getPos().y);
    if (this.equals(patch))
        return true;
    return false;

}
```

Este processo será então feito da seguinte maneira: é verificada qual a célula onde se encontra o predador. Entretanto, na classe Population, que iremos ver mais adiante, este métdo é corrido para todas as presas no terreno. Quando corremos o método preyInPatch, verificamos se a célula onde se encontram qualquer uma destas presas equivale à célula onde se encontrar o predador. Se sim, a presa será então consumida e a energia do predador é aumentada.

```
public void slowed(Terrain terrain) {
    Patch patch = (Patch) terrain.world2Cell(pos.x, pos.y);
    if(!slowed && patch.getState() == WorldConstants.PatchType.LAKE.ordinal() ||
            patch.getState() == WorldConstants.PatchType.LAKEFISH.ordinal()) {
        getDNA().maxSpeed = (float) PApplet.constrain(((float)getDNA().maxSpeed * 0.8f), ((float)maxSpeed*0.8f), maxSpeed);
        slowed = true;
    }
    if(slowed && patch.getState() != WorldConstants.PatchType.LAKE.ordinal() &&
            patch.getState() != WorldConstants.PatchType.LAKEFISH.ordinal() ) {
        getDNA().maxSpeed = (float) (getDNA().maxSpeed * 1.2);
        slowed = false;
    }
```

O método slowed serve para diminuir e repor a velocidade do predador, se este estiver num tipo de terreno equivalente a um lago. Aqui é feito um constrain ao valor, porque algo verificado ao testar este método, é que como este está constantemente a ser chamado e verificado, quando o predador se encontrasse num terreno que o abrandaria a sua velocidade era decrementada consistentemente até ser 0. Ao fazermos o constrain garantimos que esta não cai abaixo do valor pretendido.

4.1.3 Human

O agente Human, será o agente controlado pelo utilizador permite que o mesmo cace predadores. O controlo deste agente é semelhante ao comportamento Seek em que o vector de velocidade desejada é definido pelas coordenadas de onde clickarmos na janela.

```
public PVector getTarget(float x, float y) {
    double[] pp = plt.getWorldCoord(x, y);
    PVector vd = PVector.sub(new PVector((float)pp[0], (float)pp[1]), new PVector(this.getPos().x,this.getPos().y));
    return vd;
}
```

Quanto ao método eat, sendo que fundamentalmente isto será uma opção para o utilizador, optou-se por repor a energia do agente para a sua totalidade sempre que caçar uma presa.

```
public void eat(List<Predator> predators, Terrain terrain, Population population) {
    Patch patch = (Patch) terrain.world2Cell(pos.x, pos.y);
    Predator hunted = null;
    for(Predator p : predators)
    {
        if(patch.preyInPatch(p, terrain)) {
            hunted = p;
            energy = 10f;
        }
    }
    if(hunted!=null)
        population.predatorHunted(hunted);

}
```

Para verificarmos se caçámos uma presa, fazemos a mesma verificação feita nos predadores em que verificamos se na mesma célula onde se encontra o agente se também se encontra um predador. Se consumirmos um predador, este é retirado do terreno através de predadorHunter, um método auxiliar de Population que retira a presa da lista das presas.

```
public boolean die(Terrain terrain) {
    Patch patch = (Patch) terrain.world2Cell(pos.x, pos.y);

    for(int i = 0; i < terrain.nrows; i++) {
        for(int j = 0; j < terrain.ncols; j++) {
                if(terrain.getCells()[i][j].getState() == WorldConstants.PatchType.OBSTACLE.ordinal() ||
                        terrain.getCells()[i][j].getState() == WorldConstants.PatchType.LAKE.ordinal() ||
                        terrain.getCells()[i][j].getState() == WorldConstants.PatchType.LAKEFISH.ordinal()||
                        terrain.getCells()[i][j].getState() == WorldConstants.PatchType.EMPTYLAKE.ordinal())
                    if(terrain.getCells()[i][j].equals(patch))
                        return true;
            }
        }
    if(energy < 0)
        return true;
    return false;
}
```

Quanto à morte do agente, pode ocorrer por dois métodos, a sua energia terminar tal como os outros agentes ou se este colidir com obstáculos, dando assim também um desafio ao utilizador para controlar correctamente o seu agente.

5.Terreno

5.1. Patch

A classe Patch, será então uma extensão de uma célula onde se aplica a regra da maioria, ou por outras palavras, extensão da classe MajorityCell. Sendo que temos vários tipos de terreno e que alguns destes são gerados ao longo do decorrer da aplicação, utilizámos o método regenerate desenvolvido nas aulas para este feito. O método regenerate verifica o tempo actual de reprodução da aplicação e verifica se este é maior que um tempo de crescimento definido por nós, juntamente com um tempo de referência que marcamos.

```
public boolean regenerate() {
    boolean update = false;
    if(!stopGrowing) {
        if (state == PatchType.FERTILE.ordinal() && System.currentTimeMillis() > (eatenTime + timeToGrow)) {
            state = PatchType.FOOD.ordinal();
            aliveTime = System.currentTimeMillis();
            update = true;
        }
    }
    return update;
}
```

Podemos ver aqui algo que não tínhamos presente no método desenvolvido nas aulas que será a variável aliveTime. Esta variável irá ter como uso a verificação de quanto tempo a planta se encontra viva para que se possa gerar fruta nesta.

```
public boolean generateFruit() {
    boolean update = false;
    if(!stopGrowing) {
        if (state == PatchType.FOOD.ordinal() && System.currentTimeMillis() > (aliveTime + timeToGrow * 1.5)) {
            state = PatchType.FRUIT.ordinal();
            update = true;
        }

    }
    return update;
}
```

Como podemos verificar, como método generateFruit verificamos à quanto tempo uma planta se encontra viva comparando com o tempo necessário para gerar uma fruta. O tempo de crescimento da fruta, será superior ao tempo de crescimento de uma planta.

```
public boolean generateFish() {
    boolean update = false;
    if(!stopGrowing) {

        if (state == PatchType.LAKE.ordinal() && System.currentTimeMillis() > (breedTime + timeToGrow / 4)) {
            state = PatchType.LAKEFISH.ordinal();
            update = true;
        }

    }
    return update;
}
```

Para o caso da geração de peixes no lago, teremos novamente uma adaptação do código regenerate, onde iremos ter um breedTime que é marcado pelo sistema e em que o tempo de crescimento será 4 vezes inferior ao tempo normal. Isto porque sendo um dos principais

alimentos dos FlockPrey, e que a fruta demora mais tempo a ser gerada, é necessária uma fonte de alimento consistente para estes, conseguida com o baixo tempo de geração de peixes. Podemos de certa foram dizer, que a geração de peixes, não será necessariamente a reprodução destes, mas que poderá ser o tempo que estes demoram a vir à superfície para que possam ser consumidos pelos agentes.

```
public void setFertile() {
    state = PatchType.FERTILE.ordinal();
    eatenTime = System.currentTimeMillis();
}

public void setBreedable() {
    state = PatchType.LAKE.ordinal();
    breedTime = System.currentTimeMillis();
}
```

Os métodos serFertile e setBreedable, têm o mesmo objectivo, repor o estado da célula e fazer uma referência ao momento em que estas foram respostas para posteriormente ser verificado se estas se encontram prontas para novamente gerar plantas e peixes respectivamente.

5.2. Terrain

A classe Terrain é uma adaptação da classe Terrain desenvolvida nas aulas, extendo então MajorityCa(Autómato Celular de Maioria). Relativamente aos métodos base, é feita uma ligeira alteração ao método getObstacles, que guarda como obstáculos, não só as células que sejam do tipo obstáculo, mas também qualquer uma das células que contenha alguma variação de lago. Esta lista será então referente aos obstáculos das presas, cumprindo então que o facto de estas não poderem atravessar obstáculos e lagos.

```
public List<Body> getFruitsFish() {
    List<Body> bodies = new ArrayList<Body>();
    for (int i = 0; i < nrows; i++) {
        for (int j = 0; j < ncols; j++) {
            if (cells[i][j].getState() == WorldConstants.PatchType.FRUIT.ordinal()
                    || cells[i][j].getState() == WorldConstants.PatchType.LAKEFISH.ordinal()) {
                Body b = new Body(this.getCenterCell(i, j));
                bodies.add(b);
            }
        }
    }
    return bodies;
}
```

Relativamente a novos métodos temos getFruitsFish, que funciona de forma semelhante ao getObstacles mas guarda as células que contêm fruta e peixe. Esta lista será usada como uma lista de targets para os nossos FlockPrey.

Iremos ter um método de regenerate e dois métodos de generate. O método regenerate será semelhante ao método base desenvolvido nas aulas em que verifica o método generate relativo à célula, sendo que se este retornar true, altera o estado da célula. Neste caso de FERTILE para FOOD.

Os métodos generate, generateFruits e generateFish, tal como o nome indicam irão gerar frutas e peixes chamando os métodos respectivos associados à célula. Estes métodos funcionam de forma semelhante ao método regenerate.

```
public void generateFruits(PApplet p, boolean stopGrowing) {
    for (int i = 0; i < nrows; i++) {
        for (int j = 0; j < ncols; j++) {
            ((Patch) cells[i][j]).resumeGrowing(stopGrowing);
            if (((Patch) cells[i][j]).generateFruit())
                cells[i][j].display(p, this.imageHash.get(cells[i][j].getState()));
            ;
        }
    }
}

public void generateFish(PApplet p,boolean stopGrowing) {
    for (int i = 0; i < nrows; i++) {
        for (int j = 0; j < ncols; j++) {
            ((Patch) cells[i][j]).resumeGrowing(stopGrowing);
            if (((Patch) cells[i][j]).generateFish())
                cells[i][j].display(p, this.imageHash.get(cells[i][j].getState()));
            ;
        }
    }
}
```

Podemos ainda verificar nestes métodos a presença de uma variável imageHash.

```
public void imageHash(PApplet p){
    int aux = 0;
    for(String path : paths) {
        PImage img = p.loadImage(path);
        imageHash.put(aux, img);
        aux++;
    }
}
```

A variável imageHash será do tipo HashMap, um tipo de lista que guarda entradas com uma chave associada, em que a chave será o estado da célula e a entrada será o objecto PImage da imagem associada ao estado. Esta variável é inicializada no construtor, tal como o método imageHash antes das células serem geradas. Isto significa também que o autómato celular deverá receber como argumento uma lista de String com os paths das imagens correspondentes aos estados das células.

6. Population

A classe Population foi a que mais alterações sofreu relativamente à classe desenvolvida nas aulas. As populações de presas e predadores são iniciadas através de métodos auxiliares.

```
public void iniPreys(PApplet parent, Terrain terrain, SubPlot plt) {
    obstacles = terrain.getObstacles();
    for (int i = 0; i < WorldConstants.INI_PREY_POPULATION; i++) {
        PVector pos = new PVector(parent.random((float) window[0], (float) window[1]),
                parent.random((float) window[2], (float) window[3]));
        int color = parent.color(WorldConstants.PREY_COLOR[0], WorldConstants.PREY_COLOR[1],
                WorldConstants.PREY_COLOR[2]);
        Prey a = new Prey(pos, WorldConstants.PREY_MASS, WorldConstants.PREY_SIZE, color, parent, plt, terrain,
                window);
        avoid = new AvoidObstacle(10);
        wander = new Wander(1);
        avoidStrong = new AvoidObstacle(100);
        a.addBehaviour(avoid);
        a.addBehaviour(wander);
        Eye eye = new Eye(a, obstacles);
        a.setEye(eye);
        allPreys.add(a);
        targets.add(a);
    }
}
```

O método iniPreys, inicializa a população de presas, criando um número de presas equivalentes ao que definirmos. Estas presas terão como comportamentos AvoidObstacle, para tal como o nome indica evitar os obstáculos e Wander, para vaguearem pelo terreno. Quando criada a presa esta é adicionada a uma lista que contém todas as presas e uma lista que contém todos os alvos dos predadores.

```
public void iniPredators(PApplet parent, Terrain terrain, SubPlot plt) {
    for (int i = 0; i < WorldConstants.INI_PREDATOR_POPULATION; i++) {
        PVector pos = new PVector(parent.random((float) window[0], (float) window[1]),
                parent.random((float) window[2], (float) window[3]));
        int color = parent.color(WorldConstants.PREDATOR_COLOR[0], WorldConstants.PREDATOR_COLOR[1],
                WorldConstants.PREDATOR_COLOR[2]);
        Predator a = new Predator(pos, WorldConstants.PREY_MASS, (float) (WorldConstants.PREY_SIZE * 2), color,
                parent, plt, window);
        Eye eye = new Eye(a, targets);
        a.setEye(eye);
        pursuit = new Pursuit(1);
        a.addBehaviour(pursuit);
        allPredators.add(a);
        // obstacles.add(a);
    }
}
```

O método iniPredators, inicializa a população de predadores, criando um número de predadores equivalente ao que definirmos. O comportamento dos predadores será Pursuit, para perseguir os seus alvos que neste caso serão as presas.

Já o método iniFlockPrey inicializa um Flock de FlockPrey, onde podemos notar aqui que a massa e tamanho de um FlockPrey será metade comparado com os de Prey para simular agentes mais ágeis.

O movimento dos agentes terá mudanças de comportamentos associados com a visão dos mesmos.

```
for (Prey a : allPreys) {

    a.getEye().look();

    if (!a.getEye().getNearSight().isEmpty()) {
        a.removeBehaviour(avoid);
        a.addBehaviour(avoidStrong);

    }
    a.removeBehaviour(avoidStrong);
    a.addBehaviour(avoid);

    a.applyBehaviors(dt);
}
```

Para a presa podemos verificar que esta olha ao seu redor e se no seu campo de visão de segurança verificar que tem um obstáculo substitui o comportamento avoid previamente definido por avoidStrong, que será um comportamento AvoidObstacle com mais peso. Se não verificar nenhuma destas condições volta aos seus comportamentos iniciais.

```
for (Predator a : allPredators) {
    a.slowed(terrain);
    a.getEye().look();
    boolean eyeChanged = false;
    if (!a.getEye().getFarSight().isEmpty()) {
        if (!a.getEye().getNearSight().isEmpty()) {

            a.setEye(new Eye(a, a.getEye().getNearSight()));
            eyeChanged = true;
        }
        if (!eyeChanged)
            a.setEye(new Eye(a, a.getEye().getFarSight()));
    }

    if (targets.isEmpty()) {
        a.removeBehaviour(pursuit);
        a.addBehaviour(new Wander(1));
    }

    a.applyBehaviors(dt);

}
```

O movimento do predador primeiro verifica se este se encontra um piso que o atrase ou não, seguido de olhar ao seu redor. Quando olha ao seu redor verifica se tem presas no seu campo de visão mais próximo ou mais longínquo adequando a sua lista de alvos correspondentemente ao que verificar. Isto ajuda que as presas tenham padrões de movimento diferenciados em vez de perseguirem todos a mesma presa. Também verifica se ainda existem presas para perseguir, sendo que se não houverem simplesmente irá vaguear no terreno e morrer devido à escassez de alimento.

```
if (fruitsFishes != null && fruitsFishes.size() > 0) {
    for (int i = 0; i < flock.getAnimals().size(); i++) {
        Animal a = flock.getAnimals().get(i);
        a.removeBehaviour(new Separate(1f));
        a.addBehaviour(new Separate(10f));
        a.addBehaviour(new Seek(4f));
        a.setEye(new Eye(a, fruitsFishes));
        a.getEye().look();
        boolean eyeChanged = false;
        if (!a.getEye().getFarSight().isEmpty()) {
            if (!a.getEye().getNearSight().isEmpty()) {

                a.setEye(new Eye(a, a.getEye().getNearSight()));
                eyeChanged = true;
            }
            if (!eyeChanged)
                a.setEye(new Eye(a, a.getEye().getFarSight()));
        }

    }
    flock.applyBehavior(dt);

} else {
    for (int i = 0; i < flock.getAnimals().size(); i++) {
        Animal a = flock.getAnimals().get(i);
        a.removeBehaviour(new Seek(4f));
        a.setEye(new Eye(a, flock.animalList2BodyList(flock.getAnimals())));
    }
    flock.applyBehavior(dt);
}
```

Um Flock irá movimentar-se em grupo, no entanto, se um agente verificar que tem alimento no seu campo de visão irá separar-se do grupo para se alimentar. Se verificar que não tem alimentos voltará ao seu comportamento de Flock. Sendo que o Flock se movimenta em grupo isto também irá mostrar como os agentes com maior capacidade de adquirirem alimento serão aqueles que irão prevalecer durante mais tempo.

Relativamente à reprodução o método mantém-se semelhante ao que se baseia sendo que quando uma nova presa é reproduzida, esta é adicionada à lista de alvos e a visão dos predadores é atualizada para ter em conta as novas presas.

```
//método auxiliar chamado em Predator quando este consome uma presa
//remove a presa do terreno
public void preyEaten(Prey prey) {
    allPreys.remove(prey);
    targets.remove(prey);
    for (Animal a : allPredators)
        a.setEye(new Eye(a, targets));

}
```

Quanto à alimentação temos o método auxiliar preyEaten, que será chamado no método eat do predador. Este método irá retirar a presa consumida do Terreno e actualizar a visão dos predadores para não perseguirem uma presa que já não existe.

```
public int getNumAnimals() {
    return allPreys.size();
}

public int getPreyNumber() {
    int aux = 0;
    for (Prey a : allPreys)
        if (!a.mutate)
            aux++;
    return aux;
}

public int getPredatorNumber() {
    return allPredators.size();
}

public int getFlockNumber() {
    return flock.getAnimals().size();
}

public int getMutateNumber() {
    mutateN = 0;
    for (Prey a : allPreys) {
        if (a.mutate)
            mutateN++;
    }
    return mutateN;
}
```

Temos ainda uma lista de métodos auxiliares que especificamente retornam o número de cada tipo de agentes presentes no terreno, servindo para efeitos ilustrativos na aplicação quando esta estiver em execução.

Resumidamente a classe População é responsável por constantemente ir actualizando os nossos agentes aplicando-lhes e verificando todos os comportamentos implementados. De forma a termos uma simulação mais fluída implementámos a actualização da visão dos agentes quando estes se movimentam.

6.EcosystemMSSN

Esta classe será a nossa aplicação que irá implementar a interface IProcessingApp. Aqui iremos reproduzir visualmente a nossa aplicação através das ferramentas do processing para alcançarmos os objectivos pretendidos. Sendo que estamos na presença de um jogo/simulação a inicialização da aplicação fará sentido em ter um ecrã de início, seguindo de dois ecrãs que contenham o menu para descrever o funcionamento dos agentes e do terreno. Tal como referido anteriormente, iremos ter vários viewports, no entanto, para este efeito podemos fazer uso da janela na sua totalidade.

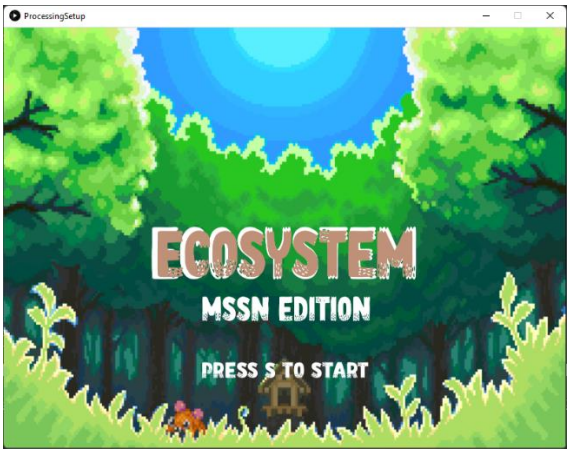


Imagem 5 – Ecrã de Inicio

Este ecrã de inicio será simplesmente uma imagem trabalhada num editor de imagem exterior (InkScape) e definida como imagem de fundo.

```
startBackground = p.loadImage("../sprites/startBackground.png");
startBackground.resize(p.width, p.height);
```

```
if(!start) {
    p.image(startBackground, 0, 0);
    p.pushStyle();
    PFont font = p.createFont("../fonts/CartooNature.ttf",20);
    p.textFont(font);
    p.fill(255);
    p.textSize(32);
    p.textAlign(p.CENTER);
    p.text("Press S to start", p.width/2, 500);
    p.popStyle();
```

Podemos ver aqui como criamos o objecto startBackground que será do tipo PImage que terá como dimensões a totalidade do ecrã. De seguida desenhamos texto no topo da imagem, centrando o mesmo no ponto que será metade do comprimento do ecrã. A variável booleana start, permite que fiquemos neste ecrã até que seja pressionada a tecla ‘S’ que irá mudar o estado da variável para true.

```
}else {
    if(!checkAgents) {
        PImage background = p.loadImage("../sprites/background.png");
        background.resize(p.width, p.height);
        p.background(background);
        agentsMenu(p);

    }
    else if(checkAgents && !checkTiles) {
        PImage background = p.loadImage("../sprites/background.png");
        background.resize(p.width, p.height);
        p.background(background);
        tilesMenu(p);
    }
```

Ultrapassada a condição imposta pela variável start, podemos então desenhar os nossos ecrãs que irão conter os menus descritivos das componentes da simulação. Aqui iremos ter novas variáveis booleanas que permitem que o ecossistema em si não se inicie enquanto ambas não forem verificadas. Com a ordem que definimos as variáveis, primeiro iremos ter o menu de agentes e de seguida o menu de terreno.

```
public void agentsMenu(PApplet p) {
    p.pushStyle();
    PFont font = p.createFont("../fonts/CartooNature.ttf",20);
    p.textFont(font);
    p.fill(0);
    p.textSize(20);
    p.fill(p.color(190,145,120));
    p.stroke(255);
    p.strokeWeight(10);
    p.rect(100, 50, 600, 100);
    PImage img = p.loadImage("../sprites/eevee.png");
    img.resize(0, 40);
    p.image(img, 120, 80);
    p.fill(255);
    p.text("Prey : ", 170, 80);
    p.text("- A small prey that just wanders around the field", 170, 100);
    p.text("- Feeds on plants and fruits that it finds", 170, 120);
    p.text("- Has vision to avoid obstacles such as rocks and lakes", 170, 140);
```

Aqui temos um exemplo de uma das entradas do menu de agentes. Algumas destas dimensões foram definidas por tentativa e erro até que fosse encontrada uma que fornecesse um efeito estético aceitável. Outras foram definidas tendo em conta as dimensões da janela com que estamos a trabalhar (800 x 600). Como podemos ver, criamos um rectângulo onde iremos desenhar a imagem do agente e o texto descritivo do mesmo.

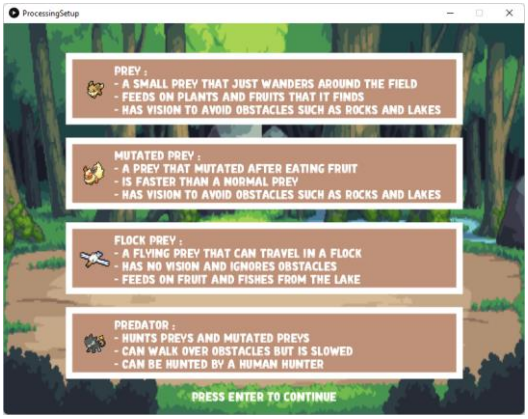


Imagem 6 – Menu de Agentes

```
public void tilesMenu(PApplet p) {
    p.pushStyle();
    PFont font = p.createFont("../fonts/CartooNature.ttf",20);
    p.textFont(font);
    p.fill(0);
    p.textSize(20);
    p.fill(p.color(190,145,120));
    p.stroke(255);
    p.strokeWeight(10);
    p.rect(100, 50, 280, 100);
    PImage img = p.loadImage("../sprites/grass_1.png");
    img.resize(0, 40);
    p.image(img, 120, 80);
    p.noFill();
    p.strokeWeight(5);
    p.rect(120, 80, 40, 40);
    p.fill(255);
    p.textSize(12);
    p.text("Grass Tile : ", 170, 80);
    p.text("- Standard Map Tile", 170, 100);
    p.text("- Can stay empty or generate plants", 170, 120);
```

A técnica para desenhar o menu de Terreno será semelhante à dos Agentes, sendo que aqui utilizamos rectângulos de comprimento menor, divindo de certa forma ao meio um rectângulo que teria um agente.

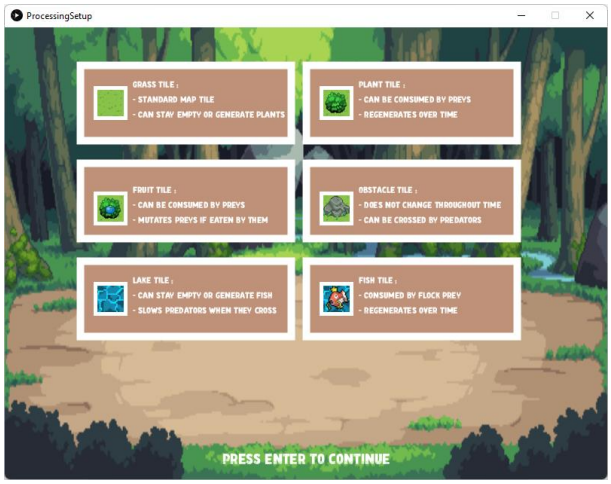


Imagem 7 – Menu de Terreno

Ambas as variáveis que permitem que estes menus se encontrem a ser desenhados no ecrã e que o ecossistema não se inicie, são alteradas quando carregadas na tecla ‘ENTER’.

Como já referido no Viewport 1 teremos então o nosso tabuleiro onde estão representados os nossos agentes e terreno a evoluir ao longo do tempo, sendo nada mais do que os objectos Population e Terrain.

No Viewport 2 vamos ter a classe AnimalCounter, que tal como o nome indica irá representar a quantidade de cada agente presente no terreno.

```
public void animalCount(int prey, int flock, int predator, int mutate){
    hash.put("prey", prey);
    hash.put("flock", flock);
    hash.put("predator", predator);
    hash.put("mutate", mutate);
}
```

O método animalCount nesta classe irá receber a quantidade de cada agente e colocar num objecto já utilizado anteriormente, HashMap, em que a chave será o nome do agente e a entrada a quantidade do mesmo. Estas quantidades são obtidas através dos métodos de contagem auxiliares referidos em Population.

```
PFont font = p.createFont("../fonts/CartooNature.ttf",20);
p.textFont(font);
p.fill(0);
p.textSize(20);
PImage img = p.loadImage("../sprites/eevee.png");
img.resize(0, 30);

float[] pp = plt.getPixelCoord(-8.5, 1);
p.image(img,pp[0], pp[1]);
pp = plt.getPixelCoord(-8.2, -2);
p.text(hash.get("prey"), pp[0], pp[1]);
pp = plt.getPixelCoord(-8.7, 2);
p.text("Preys", pp[0], pp[1]);
```

Podemos verificar o exemplo de uma das entradas do método display desta classe, onde fazemos uso do método getPixelCoord da classe SubPlot para obter as coordenadas respectivas à janela onde estamos a trabalhar relativamente às do viewport. Desenhamos então a representação visual do agente e a sua quantidade presente no terreno.

Imagem 8 – Display da classe AnimalCounter

Também é desenhado uma pequena entrada de texto para indicar ao utilizador como abrir o menu de comandos e que pode pressionar nos agentes para adicionar mais.

No viewport 3 teremos então um objecto da classe PlayerEnergyBar, que irá apresentar a barra de energia do agente controlado pelo utilizador.

```
public void display(float dt, float currentEnergy) {

    float[] pp = plt.getPixelCoord(-8.8, 1);
    float[] pp2 = plt.getPixelCoord(0, 3);

    p.pushStyle();
    p.textAlign(p.CENTER);
    p.text("Player's energy", pp2[0], pp2[1]);
    p.fill(p.color(0, 255, 0));
    float value = p.map(currentEnergy, 0, 10, 0, p.width-100);
    p.rect(pp[0], pp[1], value, 20);


    p.popStyle();


}
```

Para desenharmos uma barra que irá diminuir de acordo com a energia do agente, criamos um rectângulo em que o seu comprimento é mapeado de acordo com a energia do agente, sabendo que esta irá de 0 a 10 e que os valores a utilizar para o rectângulo serão de 0 ao comprimento da janela menos 100 pixels.

Imagem 9 – Display da classe PlayerEnergyBar

Como referido podemos aumentar o número de agentes ao pressionar sobre a imagem dos mesmos no Viewport 2.

```
float[] pp = pltCounter.getPixelCoord(-8.5, 1);
if(x >= pp[0] && x <= pp[0]+30 && y >= pp[1] && y <= pp[1]+30) {
    population.reporduceNowPrey();
}

pp = pltCounter.getPixelCoord(-3.4, 1);
if(x >= pp[0] && x <= pp[0]+30 && y >= pp[1] && y <= pp[1]+30) {
    population.reporduceNowMutatedPrey();
}

pp = pltCounter.getPixelCoord(2.5, 1);
if(x >= pp[0] && x <= pp[0]+30 && y >= pp[1] && y <= pp[1]+30) {
    population.flock.getAnimals().add(population.flock.addAnimal());
}

pp = pltCounter.getPixelCoord(7.5, 1);
if(x >= pp[0] && x <= pp[0]+30 && y >= pp[1] && y <= pp[1]+30) {
    population.reproduceNowPredator();
}
```

Isto é alcançado no método mousePressed ao definirmos as áreas onde estarão inseridas as imagens e chamado os métodos de reprodução instantânea construídos em Population e nas classes dos respectivos Agentes.

Imagem 10 – Ecossistema na Aplicação

No viewport que contém o ecossistema existe um rebordo e árvores no mesmo, que é construído através de imagens trabalhadas no InkScape que são desenhadas nas bordas da janela.

```
p.image(treeLine, 0, 0);
p.image(treeLineVert, 0, 0);
p.image(treeLineVert, p.width-20, 0);
p.image(treeLine, 0, 355);
```

Para finalizar esta classe iremos descrever resumidamente o que fazemos dentro do nosso método draw após o ecossistema ser inicializado.

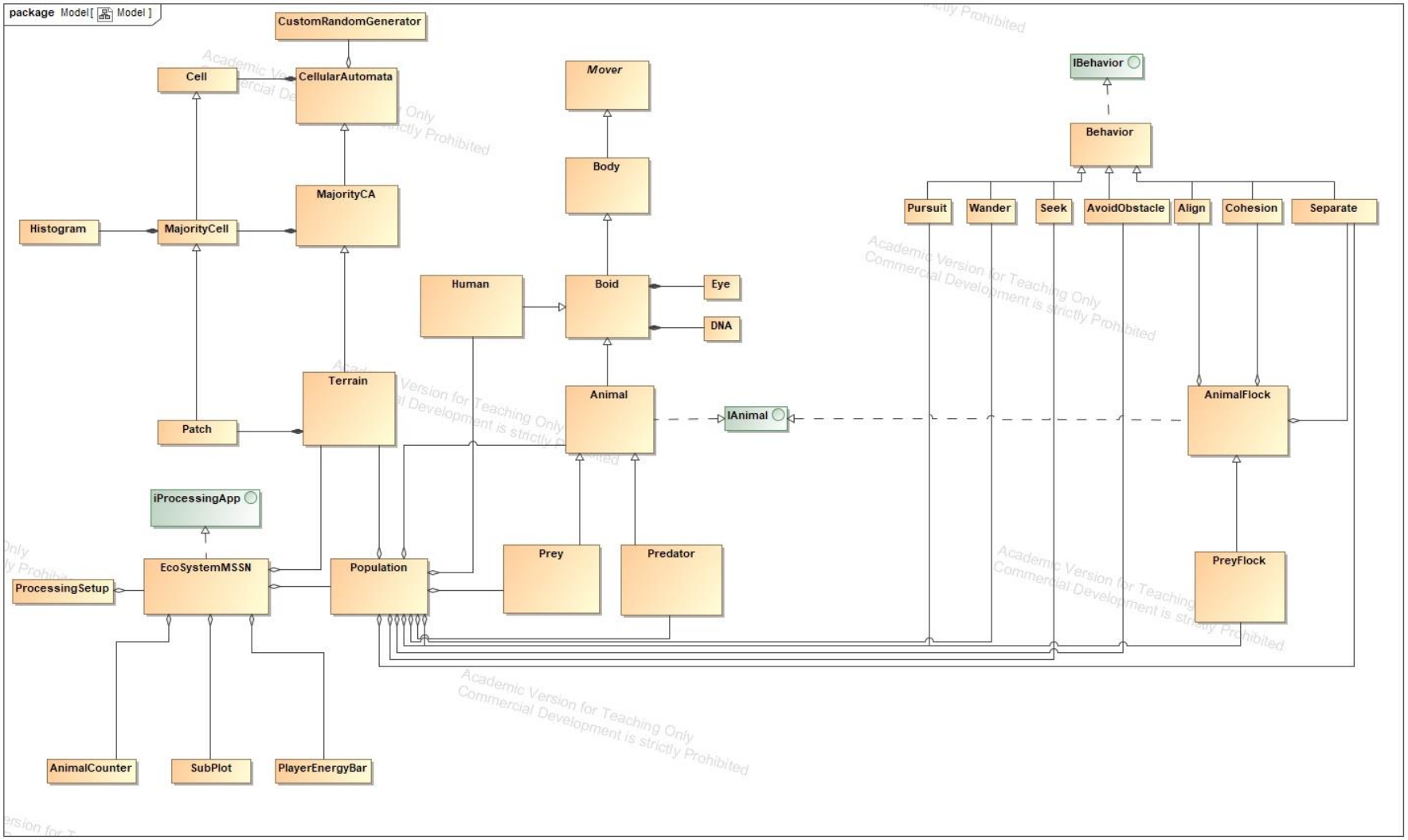
```
ac.animalCount(population.getPreyNumber(), population.getFlockNumber(), population.getPredatorNumber(),
        population.getMutateNumber());
ac.display(p, pltCounter);
if (population.hunter != null)
    playerBar.display(dt, population.hunter.energy);
terrain.regenerate(p, false);
terrain.generateFruits(p, false);
terrain.generateFish(p, false);
population.update(dt, terrain);
terrain.display(p);
if (population.hunter != null)
    population.hunter.move(dt, vd);
population.display(p, plt);
p.image(treeLine, 0, 0);
p.image(treeLineVert, 0, 0);
p.image(treeLineVert, p.width-20, 0);
p.image(treeLine, 0, 355);
if(controlMenu)
    controlMenu(p);
if(agentsMenu)
    agentsMenu(p);
if(tilesMenu )
    tilesMenu(p);
```

Sendo que no processing os objectos são desenhados por cima do que estiver no ecrã primeiro lançamos o AnimalCounter que irá conter a imagem de fundo da janela. De seguida verificamos se existe o agente do utilizador no terreno para representar a sua barra de energia.

Chamamos todos os métodos necessários para actualizar o terreno e executarem todos os comportamentos associados aos agentes.

Temos ainda a opção de que a qualquer instante podermos verificar os menus da aplicação sem interromper a evolução do ecossistema.

8.Diagrama UML



9. Resultados Obtidos

Imagem 11 – Simulação em Condições Normais

Como já referido a natureza de um terreno gerado de forma procedimental onde até as posições de origem dos agentes são aleatórios leva a que nenhuma simulação seja igual a uma outra e também como mencionado a natureza deste sistema vai levar a que o mesmo tenda para um ponto estável em termos das populações do mesmo. Neste caso verificamos um sistema em que as populações estabilizaram nos valores demonstrados no contador das mesmas, não havendo mudanças relevantes. Também como seria de esperada a presa original já não se encontra no terreno, tendo como motivos a melhor adaptação da presa mutada para não só evitar predadores mas também para consumir de forma mais rápida e eficiente comida, deixando menos alimenta para uma presa mais lenta.

Relativamente à execução da aplicação e dos comportamentos verificamos que a execução é em grande parte fluída resultado apenas em algumas perdas de frames em partes mais avançadas da mesma. Outro ponto que leva perdas de frames é o desenhar do menu de terreno, devido às imagens que está a carregar.

Quanto aos comportamentos dos agentes, consideramos que os comportamentos executados estão correctos, apenas notando que por vezes, e acreditamos que isto seja devido à velocidade que circulam, as presas consigam atravessar obstáculos. Foi optado por não colocar qualquer tipo de penalização porque mesmo ao conseguirem atravessar os obstáculos a sua velocidade fica muito reduzida comparativamente com as dos predadores, considerando então que já é uma penalização suficiente para este bug.

Imagem 12 – Simulação sem predadores

Numa simulação onde não temos predadores, verificamos por um lado um maior crescimento do número de presas, como a prevalência da espécie original, comprovando o que tínhamos identificado anteriormente que a presa original seria mais suscetível a ser caçada pelos predadores.

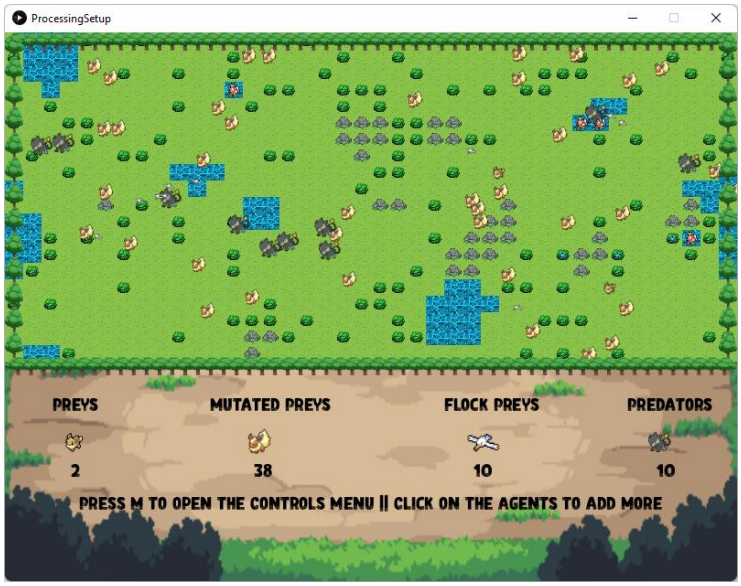


Imagem 13 – Simulação com uma população inicial de presas elevadas

Outro caso que verificamos é adicionar um número elevado de presas no início da vida do ecossistema, resultando que ao longo de tempo, devido à sua maior presença comparado com o número de frutas disponíveis ainda haja algumas presas da espécie original no terreno.

10.Conclusões

O resultado final obtido neste projecto é algo que consideramos satisfatório, foi elaborado um ecossistema onde são implementados todos os comportamentos necessários para uma população de agentes evoluir ao longo do tempo tal como a evolução do próprio terreno. O uso do autómato celular da maioria provou-se numa ferramenta extremamente útil para a geração procedimental do terreno podendo até haver resultados diferentes tendo em conta o tipo de layout produzido, tornando assim todas as execuções da aplicação únicas.

A aplicação do conhecimento em Processing adquirido ao longo do semestre provou-se útil para o desenvolvimento de uma aplicação com a representação visual pretendida. Foi ainda possível expandir este conhecimento com várias pesquisas na biblioteca do Processing para ou usar novos objectos da mesma ou aprofundar os conhecimentos de métodos anteriormente utilizados.

No entanto no contexto do projecto houve duas variáveis que gostaríamos de ter implementado, sendo elas a possibilidade pausar o ecossistema e outra a implementação de presas aquáticas que também fosse agentes. Aliás essa foi a primeira ideia de implementação, em que o FlockPrey seria um predador que caça os peixes gerados nos lagos. No entanto esta implementação causava que os peixes saíssem dos lagos e permanecessem em movimento no terreno enquanto não fossem consumidos, desrespeitando o comportamento que esse tipo de agente deveria ter.

Relativamente à pausa do ecossistema, as soluções que foram testadas nunca tiveram o objectivo pretendido sendo que os agentes pausavam os seus comportamentos, mas o terreno desenvolvia-se, isto porque o terreno tem uma espécie de relógio interno associado a cada célula, tendo sido efectuadas várias alternativas ou para preservar este relógio ou o actualizar, sempre sem sucesso.

Excluindo estas duas situações, ficámos satisfeitos com o resultado final do projecto que pensamos representar a nossa visão e também respeitar o que deverá ser um ecossistema, podendo não só ter uma simulação base, como testarmos vários cenários onde podemos alterar o número de cada tipo de agentes verificando todo o tipo de cenários estudados na unidade curricular, comprovando a importância de modelos e simulações para a projecção de vários elementos da sociedade como a conhecemos.