Licenciatura Engenharia Informática e Multimédia

Instituto Superior de Engenharia de Lisboa

Ano letivo 2024/2025

Modelação e Simulação de Sistemas Naturais

Docente: Arnaldo Abrantes

Projeto Final

Nome: Pedro Marques Numero: 51959

Nome: Gianni Floriddia Numero: 51945

Turma 32D

Data: 27 de Dezembro de 2024

INDICE

Lista de Figuras

Acrónimos

GUI– *Graphical User Interface*

Capítulo 1 – Introdução

1.1 Considerações Iniciais

Neste projeto final da disciplina de Modelação e Simulação de Sistemas Naturais (MSSN), desenvolvemos uma simulação interativa em formato de jogo baseada no clássico "Pedra, Papel ou Tesoura".

A proposta consiste em agentes autónomos, representados como Boids, que se movem através de um ambiente gerado proceduralmente. Este ambiente é composto por diferentes tipos de terrenos, cada um com propriedades específicas que podem influenciar os Boids, causando-lhes dano dependendo da localização.

Ao interagirem, os Boids aplicam as regras do jogo "Pedra, Papel ou Tesoura", determinando o resultado de cada colisão com base nas regras tradicionais do jogo. Este mecanismo de interação reflete conceitos de competição

O projeto utiliza a modelação baseada em agentes para simular comportamentos individuais e de grupo, a geração procedural de terrenos para criar um ambiente dinâmico e variado, e a simulação de movimentos baseada nas leis da física para determinar o deslocamento dos Boids.

1.2 Criação da ideia

A ideia para o projeto começou numa das aulas práticas finais do semestre nas quais o professor nos deu diversas ideias sobre o que fazer. Achamos que a ideia de cria um ecossistema com animais/extraterrestres/seres vivos no geral era algo muito comum e que todos os grupos iriam fazer algo do género.

Tínhamos duas ideias de projeto e para escolhermos uma decidimos jogar ao “Pedra, Papel Tesoura” e surgiu então uma terceira ideia um jogo que ajudasse dois amigos a fazer decisões.

Chegamos por fim á conclusão que daria para fazer um jogo do próprio pedra papel tesoura com Boids.

1.3. Diagramas UML

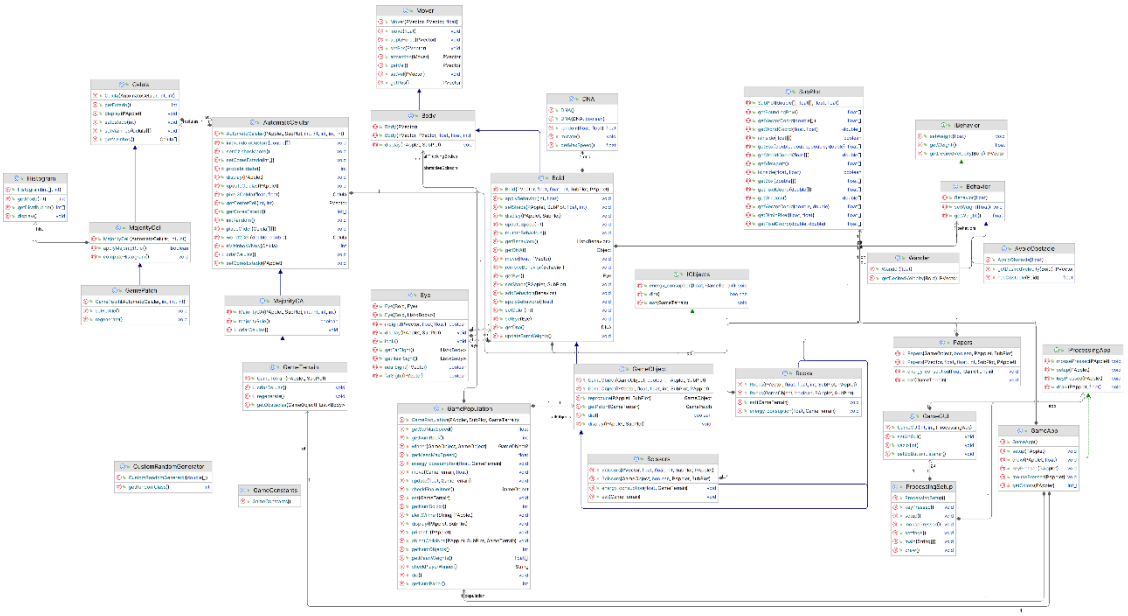

Figura 1: UML 1

O diagrama UML apresentado ilustra a arquitetura do sistema desenvolvido, destacando os principais componentes, suas interações e relações hierárquicas.

Devido à complexidade do projeto, apresentamos também um segundo diagrama de UML simplificado:

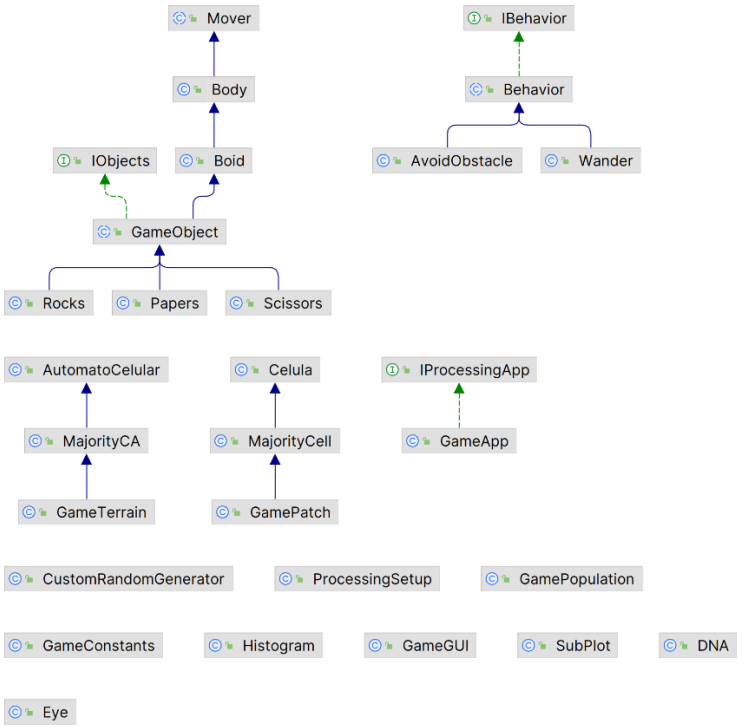

Figura 2: UML 2

No núcleo do sistema, destacam-se as classes base como Body e Boid desenvolvidas ao longo do semestre, que encapsulam atributos e comportamentos fundamentais, como movimento e posicionamento. Estas classes servem como base para entidades específicas do jogo, como Rocks, Papers e Scissors, que herdam funcionalidades básicas e adicionam comportamentos específicos.

Além disso, classes como GameTerrain, GamePopulation e GamePatch são responsáveis por modelar o ambiente de simulação e gerenciar a interação entre os agentes e o espaço.

Outros componentes, como Histogram e DNA, apoiam funcionalidades auxiliares, fornecendo dados analíticos e parâmetros de genética para os agentes. A interação do sistema com o utilizador é gerida por meio de classes como GameApp e GameGUI, que tratam da interface gráfica e da lógica de execução do jogo.

Capítulo 2 – Regras do jogo

Para que o jogo funcionasse corretamente algumas regras tiveram de ser criadas e para isto criamos uma classe "GameConstants" com constantes/regras necessárias para o trabalho.
(Valores tidos como "default" podem ser alterados pelos jogadores)

As regras usadas no modelo são os seguintes:

- Número inicial de cada objetos (default): 15;
- Percentagens do terreno inicial:
    - Água: 19.5%;
    - Obstáculos: 15%;
    - Fértil: 0%;
    - Relva: 23%;
    - Pedregulhos: 19.5%;
    - Terreno Infértil: 23%.
- Tempo de regeneração da relva: 20 a 60 segundos;
- Número máximo de objetos: 250;
- Vida inicial/Máxima dos objetos: 500;
- Danos recebidos:
    - Papeis:
        - Nos Obstáculos: Vida total;
        - Na Água: 2;
    - Pedras:
        - Nos Obstáculos: Vida total;
        - Na Água: Vida total;
        - Na Relva: 1;
    - Tesouras:
        - Nos Obstáculos: Vida total;
        - Na Água: Vida total;
        - Nos Pedregulhos: 1;
- Dano do terreno (default): desativado;
- Balanceamentos para os objetos:
    - Os Papeis são os únicos objetos que andam na água, no entanto perdem um pouco de vida;
    - As Pedras recuperam vida ao andar em Pedregulhos:
        - Vida recuperada = 2;
    - As Tesouras possuem mais facilidade em apanhar Papeis visto que estes não são parados pela Água então as Tesouras cortam a relva (que dá dano ás Pedras) de modo a balancear;

Capítulo 3 – Criação do Mapa

O terreno do jogo foi criado utilizando uma geração procedural baseada em autómatos celulares e na aplicação da regra da maioria. Esta técnica permite simular um ambiente dinâmico e visualmente interessante, gerado de forma automática.

Para implementar esta funcionalidade, a classe GameTerrain foi desenvolvida como uma extensão de MajorityCA, que é responsável por gerir o comportamento global do autómato celular. Cada célula individual do terreno é representada por uma instância da classe GamePatch, que estende MajorityCell, incorporando as regras locais do autómato.

A regra da maioria define que o estado de uma célula em cada iteração é atualizado com base no estado predominante entre os seus vizinhos, criando padrões complexos e coesos no terreno. Este método não só facilita a criação de terrenos variados como também assegura consistência nos detalhes gerados procedimentalmente.

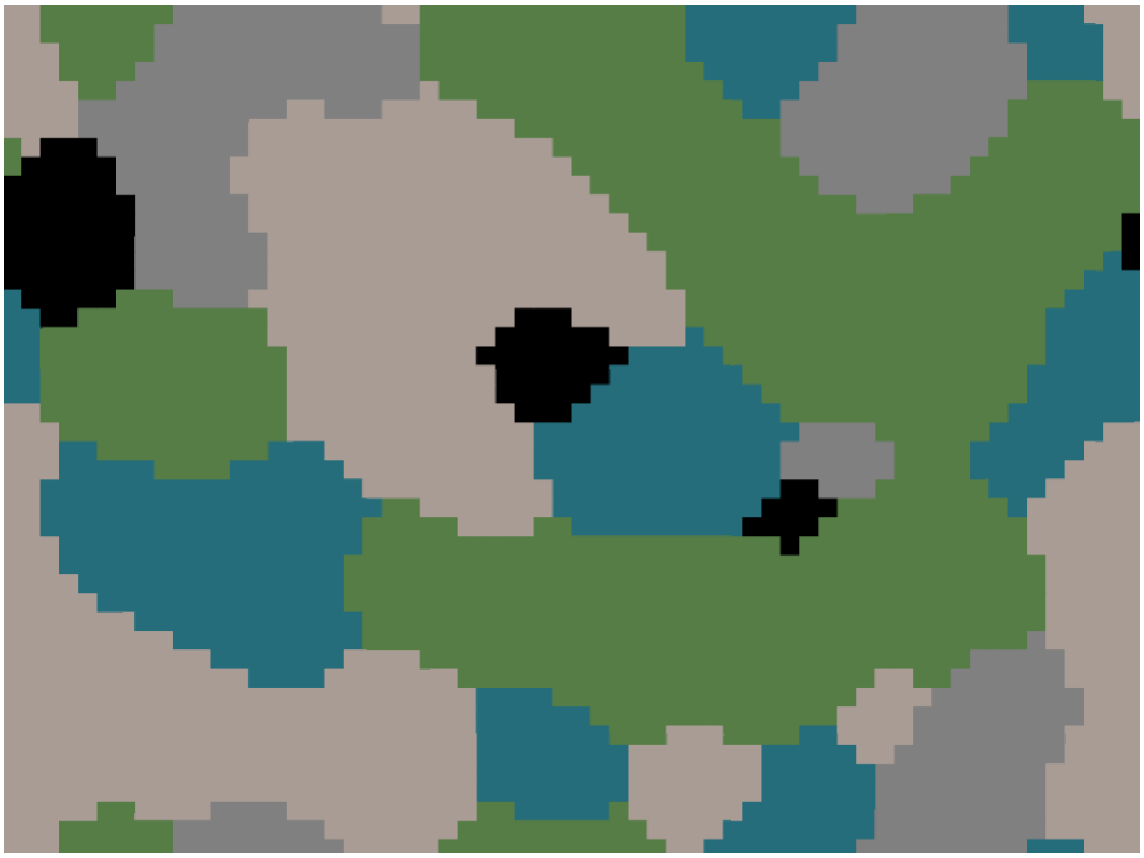

Figura 3: Mapa criado com regra da maioria

A classe GameTerrain difere apenas pela adição de três métodos adicionais relativamente a MajorityCA:

- createCells() – método que passa por todas as linhas e colunas e cria novas células GamePatch.
- regenerate() – itera por todas as células e aplica o método regenerate() da classe GamePatch descrito seguidamente.
- getObstacles(GameObject objeto) – itera por todas as células e define quais são obstáculos, considerando o tipo de objeto:

```
public List<Body> getObstacles(GameObject objeto) {
    List<Body> bodies = new ArrayList<Body>();
    for (int i = 0; i < GameConstants.NROWS; i++){
        for (int j = 0; j < GameConstants.NCOLS; j++) {
            if (objeto instanceof Scissors  || objeto instanceof Rocks){
                if (/*estado == obstacles || estado == water*/){
                    Body b = new Body(this.getCenterCell(i, j));
                    bodies.add(b);
                }
            } else {
                if (celulas[i][j].getEstado() == GC.PatchType.OBSTACLES.ordinal()){
                    Body b = new Body(this.getCenterCell(i, j));
                    bodies.add(b);
                }
            }
        }
    }
    return bodies;
}
```

A classe GamePatch tem dois métodos adicionais em comparação com a classe base:

- setFertile() – Utilizada para definir uma célula como fértil.
- regenerate() – Utilizada para regenerar uma célula para relva, com base no tempo:

```
public void regenerate(){
   if (estado == GameConstants.PatchType.FERTILE.ordinal() &&
System.currentTimeMillis() > (eatenTime + timeToGrow))
      estado = GameConstants.PatchType.GRASS.ordinal();
}
```

Capítulo 4 – Pedras, Papeis e Tesouras

GameObject – Todos os objetos (pedras, papeis e tesouras) são GameObjects.

Esta é a classe base dos objetos e como tal possui o seguinte construtor:

```
protected GameObject(GameObject o, boolean mutate, PApplet p, SubPlot plt){
   super(o.pos, o.mass, o.radius, o.color, plt, p);
   for (Behavior b : o.behaviors)
      this.addBehavior(b);
   if (o.eye != null)
      eye = new Eye(this, o.eye);
   dna = new DNA(o.dna, mutate);
}
```

Cada objeto possui posição, massa, raio, cor, behaviors, eye e Dna. Para alem disso, a classe GameObject possui também 4 métodos internos que são:

- Public GamePatch getPatch (GameTerrain terrain){};
- Public boolean die (){};
- Public GameObject reproduce (PAppelt p, SubPlot plt){};
- Public void display (PApplet p, SubPlot plt){};

getPatch():

Método que retorna a célula onde o Objeto se encontra.

```
public GamePatch getPatch(GameTerrain terrain){
   return (GamePatch) terrain.world2Cell(pos.x, pos.y);
}
```

die():

Método que indica se o Objeto deve morrer ou não.

```
public boolean die() {
   return (energy < 0);
}
```

reproduce():

Método chamado quando dois objetos colidem. O método é chamado através de algum objeto dos tipos Pedra, Papel ou Tesoura (vencedor do embate) e origina um novo objeto semelhante.

```
public GameObject reproduce(PApplet p, SubPlot plt) {
    GameObject o = new Rocks(this, false, p, plt);
    if (this instanceof Scissors)
        o = new Scissors(this, false, p , plt);
    else if (this instanceof Papers)
        o = new Papers(this, false, p, plt);
    else if (this instanceof  Rocks)
        o = new Rocks(this, false, p, plt);
    else
        System.out.println("AVISO: Objeto" + o.getClass() + " não pode ser reproduzido!");
    return o;
}
```

display():

Método que dá display do Objeto no Mapa, já com o emoji definido para o seu tipo.

```
public void display(PApplet p, SubPlot plt) {
    p.pushMatrix();
    float[] pp = plt.getPixelCoord(pos.x, pos.y);
    p.translate(pp[0], pp[1]);
    p.rotate(-vel.heading());

    if (emoji != null) {
        float scaleFactor = .1f;
        p.scale(scaleFactor);
        p.imageMode(PApplet.CENTER);
        p.image(emoji, 0, 0);
    } else {
        System.out.println("AVISO: Imagens não carregadas!");
        float scaleFactor = .5f;
        p.scale(scaleFactor);
        super.display(p, plt);
    }
    p.popMatrix();
}
```

4.1 Pedras

Pedras são Objetos que têm vantagem sobre as Tesouras e desvantagem sobre os Papeis. Estas estendem GameObject e possuem o seguinte construtor:

```
protected Rocks(PVector pos, float mass, float radius, int color, SubPlot plt, PApplet
parent) {
   super(pos, mass, radius, color, plt, parent);
   this.parent = parent;
   this.plt = plt;
   energy = GameConstants.INIT_OBJECT_ENERGY;
   this.emoji =
parent.loadImage("C:\\Users\\USER\\Desktop\\ProjetoMSSN\\ProjetoMSSN\\src\\images\
\moyai.png");
}
```

Possui também um construtor alternativo que serve quando o Objeto é criado através de reprodução imitando o pai protected Rocks(GameObject 0, boolean mutate,…).

Por fim, possui um método energy_consuption(float dt, GameTerrain terrain) que aplica as regras do jogo retirando e dando energia ao Objeto.

```
public void energy_consuption(float dt, GameTerrain terrain) {
   GamePatch patch = (GamePatch) terrain.world2Cell(pos.x, pos.y);
   if (patch.getEstado() == GameConstants.PatchType.OBSTACLES.ordinal() ||
patch.getEstado() == GameConstants.PatchType.WATER.ordinal()){
      energy = -1f;
   } else if (patch.getEstado() == GameConstants.PatchType.GRASS.ordinal()){
      energy -= 1f;
   } else if (patch.getEstado() == GameConstants.PatchType.ROCKS.ordinal())
      energy += 2f;
}
```

Figura 4- Aparência Pedras

4.2 Papeis

Papeis são Objetos que têm vantagem sobre as Pedras e desvantagem sobre as Tesouras. Estas estendem GameObject e possuem o seguinte construtor:

```
protected Papers(PVector pos, float mass, float radius, int color, SubPlot plt, PApplet
parent) {
   super(pos, mass, radius, color, plt, parent);
   this.parent = parent;
   this.plt = plt;
   energy = GameConstants.INIT_OBJECT_ENERGY;
   this.emoji =
parent.loadImage("C:\\Users\\USER\\Desktop\\ProjetoMSSN\\ProjetoMSSN\\src\\images\
\page_with_curl.png");
}
```

Possui também um construtor alternativo que serve quando o Objeto é criado através de reprodução imitando o pai protected Papers(GameObject 0, boolean mutate,…).

Por fim, possui um método energy_consuption(float dt, GameTerrain terrain) que aplica as regras do jogo retirando e dando energia ao Objeto.

```
public void energy_consuption(float dt, GameTerrain terrain) {
   GamePatch patch = (GamePatch) terrain.world2Cell(pos.x, pos.y);
   if (patch.getEstado() == GameConstants.PatchType.OBSTACLES.ordinal()){
      energy = -1f;
   } else if (patch.getEstado() == GameConstants.PatchType.WATER.ordinal()){
      energy -= 2f;
   }
}
```

Figura 5 - Aparência Papeis

4.3 Tesouras

Tesouras são Objetos que têm vantagem sobre as Papeis e desvantagem sobre as Pedras. Estas estendem GameObject e possuem o seguinte construtor:

```
protected Scissors(PVector pos, float mass, float radius, int color, SubPlot plt, PApplet
parent) {
   super(pos, mass, radius, color, plt, parent);
   this.parent = parent;
   this.plt = plt;
   energy = GameConstants.INIT_OBJECT_ENERGY;
   this.emoji =
parent.loadImage("C:\\Users\\USER\\Desktop\\ProjetoMSSN\\ProjetoMSSN\\src\\images\
\scissors.png");
}
```

Possui também um construtor alternativo que serve quando o Objeto é criado através de reprodução imitando o pai protected Scissors(GameObject 0, boolean mutate,…).

Por fim, possui um método energy_consuption(float dt, GameTerrain terrain) que aplica as regras do jogo retirando e dando energia ao Objeto.

```
public void energy_consuption(float dt, GameTerrain terrain) {
   GamePatch patch = (GamePatch) terrain.world2Cell(pos.x, pos.y);
   if (patch.getEstado() == GameConstants.PatchType.OBSTACLES.ordinal() ||
patch.getEstado() == GameConstants.PatchType.WATER.ordinal()){
      energy = -1f;
   } else if (patch.getEstado() == GameConstants.PatchType.ROCKS.ordinal()){
      energy -= 1f;
   }
}
```

Figura 6 - Aparência Tesouras

Capítulo 5 – População

A população do jogo é constituída por todos os GameObjects esta é uma das classes principais do projeto sendo aqui que se realizam métodos como o aviso e avaliação do vencedor, o desgaste da vida aos objetos por parte do terreno, a morte dos objetos e o corte da relva por parte das Tesouras.

No construtor da GamePopulation, são criados 3 objetos que servem para definir todos os Obstáculos/locais proibidos de cada tipo de objetos, alem disso, são inicializados todos os objetos iniciais com "n" número de cada (n=número definido pelos jogadores) atribuindo-lhes as características predefinidas nas GameConstants como a cor, são também adicionados os Behaviors como o Wander e o AvoidObstacles e um olho para que os Objetos tenham noção do que se passa ao redor. Por fim, os objetos são adicionados ao arrayList allObjects que serve para podermos aceder a todos os objetos quando necessário.

Construtor:

```
public GamePopulation(PApplet p, SubPlot plt, GameTerrain terrain){

    window = plt.getWindow();
    allObjects = new ArrayList<GameObject>();
    this.count = 0;

    Scissors exempleScissor = new Scissors(new PVector(p.random((float) window[0],
(float) window[1]), p.random((float) window[2], (float) window[3])),
GameConstants.OBJECT_MASS, GameConstants.OBJECT_SIZE,
p.color(GameConstants.SCIZOR_COLOR[0], GameConstants.SCIZOR_COLOR[1],
GameConstants.SCIZOR_COLOR[2]), plt, p);
    Rocks exempleRocks = new Rocks(new PVector(p.random((float) window[0], (float)
window[1]), p.random((float) window[2], (float) window[3])),
GameConstants.OBJECT_MASS, GameConstants.OBJECT_SIZE,
p.color(GameConstants.ROCK_COLOR[0], GameConstants.ROCK_COLOR[1],
GameConstants.ROCK_COLOR[2]), plt, p);
    Papers exemplePapers = new Papers(new PVector(p.random((float) window[0], (float)
window[1]), p.random((float) window[2], (float) window[3])),
GameConstants.OBJECT_MASS, GameConstants.OBJECT_SIZE,
p.color(GameConstants.PAPER_COLOR[0], GameConstants.PAPER_COLOR[1],
GameConstants.PAPER_COLOR[2]), plt, p);

    obstaclesScissors = terrain.getObstacles(exempleScissor);
    obstaclesRocks = terrain.getObstacles(exempleRocks);
    obstaclesPapers = terrain.getObstacles(exemplePapers);

    for (int i = 0; i < GameConstants.INIT_EACH_POPULATION; i++){
        PVector pos = new PVector(p.random((float) window[0], (float) window[1]),
p.random((float) window[2], (float) window[3]));
        GamePatch patch = (GamePatch) terrain.world2Cell(pos.x, pos.y);
        while (patch.getEstado() == GameConstants.PatchType.WATER.ordinal() ||
patch.getEstado() == GameConstants.PatchType.OBSTACLES.ordinal() ||
patch.getEstado() == GameConstants.PatchType.ROCKS.ordinal()){
```

```
        pos = new PVector(p.random((float) window[0], (float) window[1]),
p.random((float) window[2], (float) window[3]));
        patch = (GamePatch) terrain.world2Cell(pos.x, pos.y);
      }
      int colorScissor = p.color(GameConstants.SCIZOR_COLOR[0],
GameConstants.SCIZOR_COLOR[1], GameConstants.SCIZOR_COLOR[2]);
      GameObject tesoura = new Scissors(pos, GameConstants.OBJECT_MASS,
GameConstants.OBJECT_SIZE, colorScissor, plt, p);
      tesoura.addBehavior(new Wander(1));
      tesoura.addBehavior(new AvoidObstacle(50));
      Eye eye = new Eye(tesoura, obstaclesScissors);
      tesoura.setEye(eye);
      allObjects.add(tesoura);
   }
…(igual para outros objetos)…
```

O método **update()** é chamado a cada frame e neste ocorre o seguinte:

- ❖ Move(terrain,dt) – aplica os Behaviors a todos os objetos da população (incluindo os novos criados via colisão entre os anteriores);
- ❖ Eat(terrain) – corta as células de relva por onde as Tesouras passam;
- ❖ Energy_consumption(dt,terrain) – caso o dano do terreno estiver ativo por opção dos jogadores, realiza as regras de dano causado pelo terreno;
- ❖ Die() – mata os Objetos que já não possuírem vida.

objectColisions: método que avalia o vencedor do embate entre os objetos verifica os vencedores (usando método winner(GameObject 01, GameObject 02)) e reproduz os mesmos removendo os perdedores.

```
public void objectColisions(PApplet p, SubPlot plt, GameTerrain terrain){
   GameObject wins;
   if (allObjects.size() < GameConstants.MAX_NUM_OBJECTS){
      for (int i = 0; i < allObjects.size(); i++){
         GameObject o1 = allObjects.get(i);
         for (int j = i+1; j< allObjects.size(); j++){
            GameObject o2 = allObjects.get(j);
            if (o1.getPatch(terrain)==o2.getPatch(terrain)){
               wins = winner(o1,o2);
               if (wins == o1){
                  allObjects.remove(o2);
                  allObjects.add(wins.reproduce(p, plt));
               } else if (wins == o2) {
                  allObjects.remove(o1);
                  allObjects.add(wins.reproduce(p, plt));
               }
               break;
            }
         }
      }
   }
}
```

Outros métodos relevantes:

- Public GameObject checkFinalWinner():
  - Verifica se existem objetos no ArrayList allObjects diferentes, se não retorna o tipo de objeto Vencedor do jogo.

```
public GameObject checkFinalWinner() {
    if (allObjects.isEmpty()) {
        return null;
    }

    GameObject firstObject = allObjects.get(0);
    for (GameObject obj : allObjects) {
        if (!obj.getClass().equals(firstObject.getClass()))
{
            return null;
        }
    }

    return firstObject;
}
```

- Public void printInfo(PApplet p):
  - Imprime informações na consola como número de cada objeto e vencedor final.
- Public String checkPlayerWinner():
  - Retorna o jogador que venceu, ou empate.

```
private String checkPlayerWinner(){
    String vencedor = "";
    if (checkFinalWinner() instanceof Papers)
        vencedor = "Papel";
    else if (checkFinalWinner() instanceof Rocks)
        vencedor = "Pedra";
    else if (checkFinalWinner() instanceof Scissors)
        vencedor = "Tesoura";

    if (vencedor.equals(GameConstants.Player1))
        return "Player1";
    else if (vencedor.equals(GameConstants.Player2))
        return "Player2";
    else
        return "Empate";
}
```

- Public void alertWinner(String vencedor PApplet p):
  - Realiza um alerta na tela de vencedor.

Capítulo 6 – GUI / Lobby

Para que o jogo tivesse uma melhor ligação com os jogadores decidimos criar um lobby onde estes poderiam escolher o seu objeto, o número de cada objeto e decidir se o dano do terreno deve estar ativo ou não.

Queríamos realizar algo visualmente mais cativante e completo, no entanto não teríamos tempo do terminar, então optámos por usar algo funcional e com o qual sabíamos trabalhar por termos obtido conhecimentos noutras cadeiras, o Java Swing.

Usámos então um grid layout com alguns botões, caixa de texto e comboBox para obtermos valores já indicados acima.

Figura 7: Lobby do jogo

6.1. Aparência do jogo

Com todas as componentes, o jogo fica, portanto, com a seguinte aparência:

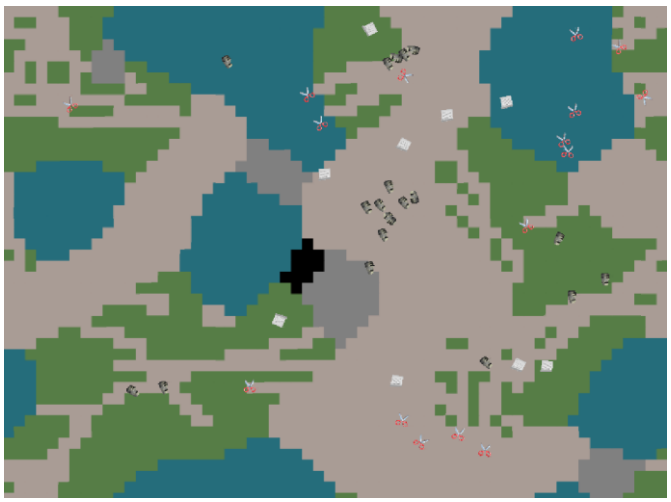

Figura 8:Aparência do Jogo

Capítulo 7 – Análise de Dados

Foi criada posteriormente uma classe PopulationLogger, que guarda a informação da quantidade de cada objeto de x em x tempo num ficheiro .csv para análise de dados. Neste caso utilizamos o Excel para geração de gráficos.

7.1. Sem dano do terreno

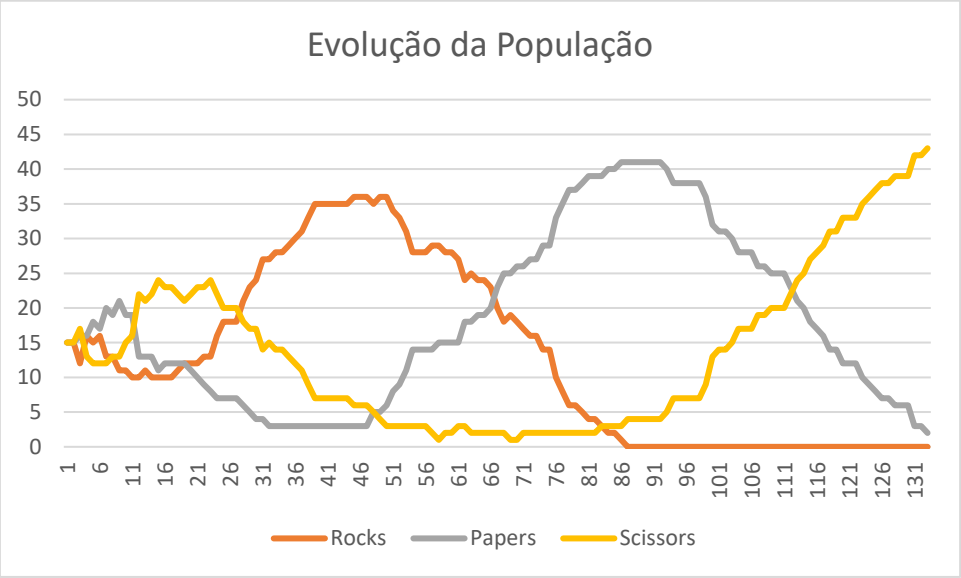

Figura 9:Evolução da População sem Dano

Podemos concluir que se o dano no terreno não estiver ativado, o jogo fica num Loop, até que um dos objetos finalmente morra por completo, e aí sim, um dos dois restantes ganhará. Enquanto isso não acontecer, o jogo vai ficar neste Loop contínuo

Portanto, visto que a não há nenhum fator contribuinte para o resultado, podemos considerar que cada objeto tem uma probabilidade aproximada de 33.33% de ganhar.

7.2. Com dano do terreno

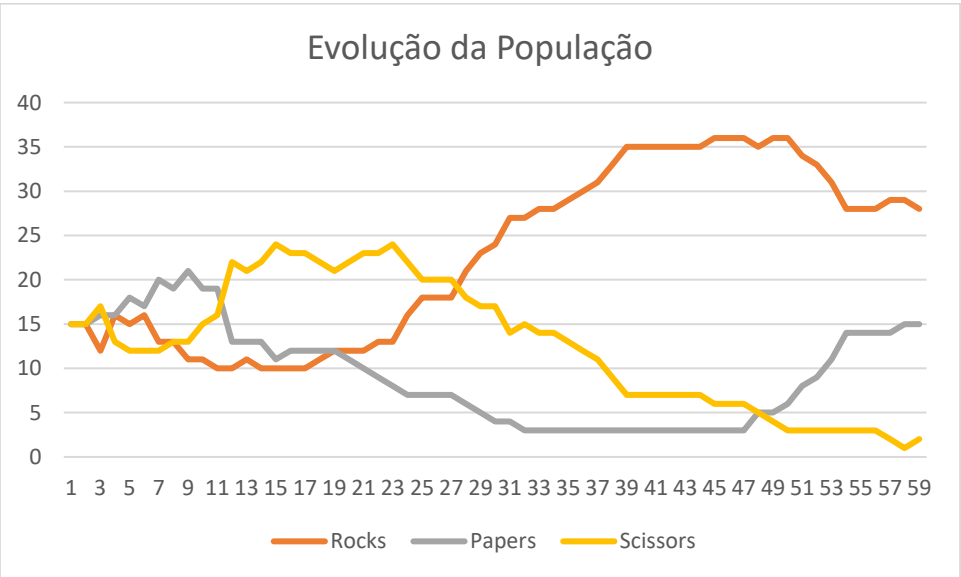

Figura 10:Evolução da População com Dano

Neste caso, conseguimos observar que o resultado não é tão afetado pelas regras tradicionais do jogo, mas sim por morte de dano, pois verifica-se uma diminuição do nível geral dos objetos. Deste modo, a disposição aleatória do mapa será um grande fator para o resultado.

Foi também feita uma contagem de resultados correndo o jogo 10 vezes:

- Com 5 objetos verificou-se que as pedras ganharam 3 vezes, o papel 2 vezes e a tesoura 5 vezes.
- Com 15 objetos verificou-se que as pedras ganharam 2 vezes, o papel 4 vezes e a tesoura 4 vezes.
- Com 30 objetos verificou-se que as pedras ganharam 5 vezes, o papel 2 vezes e a tesoura 3 vezes.

Podemos assumir assim que, a probabilidade de um objeto ganhar é realmente definida pela disposição do mapa.

8. Conclusões

O desenvolvimento deste projeto permitiu uma aplicação prática e abrangente dos conceitos fundamentais da disciplina de Modelação e Simulação de Sistemas Naturais, consolidando conhecimentos teóricos através da criação de um sistema dinâmico complexo. A implementação do jogo, com a integração de comportamentos adaptativos e a análise de dados obtidos, demonstrou a importância de ferramentas como a simulação computacional para compreender interações entre agentes num ambiente virtual. Além disso, o trabalho destacou o papel crítico da parametrização do sistema, como o dano do terreno, no comportamento global do modelo. Este processo não só aprofundou a nossa capacidade de programar e modelar sistemas realistas, mas também nos proporcionou uma visão prática das implicações e desafios da modelação de sistemas naturais, fomentando competências essenciais em análise, pensamento crítico e resolução de problemas.