Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e Multimédia

Modelação e Simulação de Sistemas Naturais

Ano Letivo 2024/2025 – 3º Semestre

Produção de Conteúdos Multimédia

Docente: Docente: Paulo Vieira

Projeto Final - Relatório

Turma 31N

Duarte Fonseca – Nº50825

Índice

1.Introdução

Este projeto tem como objetivo desenvolver um jogo recorrendo à linguagem de programação *Processing* e à linguagem de programação *JAVA* e à reutilização de código já desenvolvido nos trabalhos anteriores da cadeira.
Como pedido no enunciado final, implementou-se vários dos temas e tópicos que se aprendeu a Modelação e Simulação de Sistemas Naturais. Estes temas são:

- Agentes Autômatos e Boids,
- Sistemas de Partículas,
- Geração procedural: Autómato Celular
- Simulação Física

O jogo é um jogo de simulação física do gênero roguelite, action shooter e sci-fi.

Neste relatório começa-se por explicar o *design pattern* aplicado em jogos, seguido pelo modelo geral da aplicação. Depois é explicado com maior pormenor a implementação de vários elementos como o Jogador, Inimigos, Geração de nível, etc. Em cada capítulo de implementação é primeiro demonstrado a modelação do código: *UML,* seguido pela explicação da sua implementação. Por fim explica-se algumas técnicas de optimização tomadas e este relatório termina com o capítulo das Conclusões.

2. Padrões de Desenho

2.1. Game Loop

Neste capítulo quando se refere a *Game Loop* é no sentido estrito do ciclo de atualização dos elementos do jogo como física, gráficos, etc. Neste sentido não se esta a referir ao termo Game Loop no contexto de design de jogos que exemplifica as formas de interação repetitiva que o player tem no jogo.
No início do desenvolvimento dos jogos, era comum o Game Loop ser apenas um ciclo que é atualizado constantemente à velocidade que a máquina permite como exemplificado na seguinte figura :

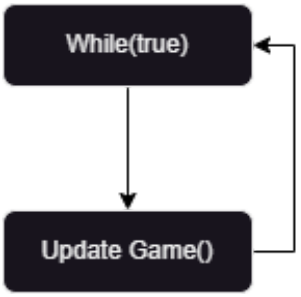


Figura 1. Diagrama exemplo de um *Game Loop*

Este tipo de abordagem tem um problema: Introduz bugs relacionados com o tempo que cada frame demora a ser gerado, que é afetado tanto pelo quão complexo é a física a ser simulada como pelos elementos gráficos. Por exemplo, no jogo space Invaders, a dificuldade aumenta à medida que os aliens são mortos e removidos do ecrã/jogo, o processador tem menos ″trabalho a fazer″, por ter que simular menos inimigos. Deste modo a lógica do jogo é atualizada de forma mais rápida.

Neste contexto, outro tipo de bug introduzido a nível de física é que para a mesma caminhada de um personagem numa simulação ou jogo, o tempo que demora a fazer um percurso à mesma velocidade é diferente dependendo do quão rápido um computador é. Em jogos de um jogador isto normalmente não introduz muitos erros, sendo que uma forma fácil de corrigir é limitar o número máximo de frames gerados. No entanto, num jogo multijogador os bugs mencionados acima são mais agravantes pois pretende-se que os jogadores tenham a mesma experiência para computadores com especificações semelhantes. A seguinte figura tenta exemplificar a dessincronização ao nível da física, quando se tem computadores com potências diferentes:

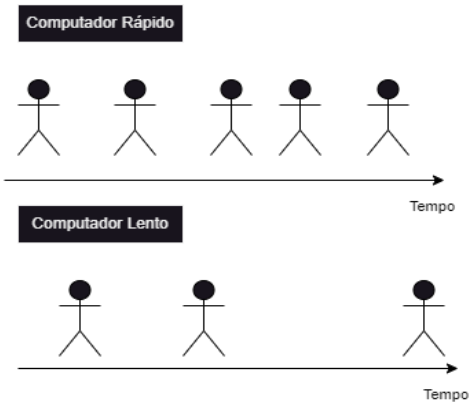


Figura 2. Exemplo de um bug de simulação de física.

Todos os jogos têm Game Loops, nem que seja um tão simples como o referido anteriormente. No entanto, o padrão de desenho *Game Loop* mais aceita atualmente é o que procura remover este problema ao separar a lógica física dos elementos gráficos.
Para resolver este problema, ao invés de o jogo ser atualizado, linha de código a linha de código, demorado mais ou menos dependendo do quão complexo as instruções executadas são como num programa tradicional, introduz-se o conceito de ciclo de jogo. Cada ciclo tem em conta o tempo que cada frame demora a ser atualizado: Se demorar mais tempo são feitos mais ciclos, se demorar menos tempo são feitos menos ciclos. Ao tempo que cada frame demora a ser renderizado denomina-se, por convenção, *deltaTime*.

O diagrama do Game Loop Referido é o seguinte:

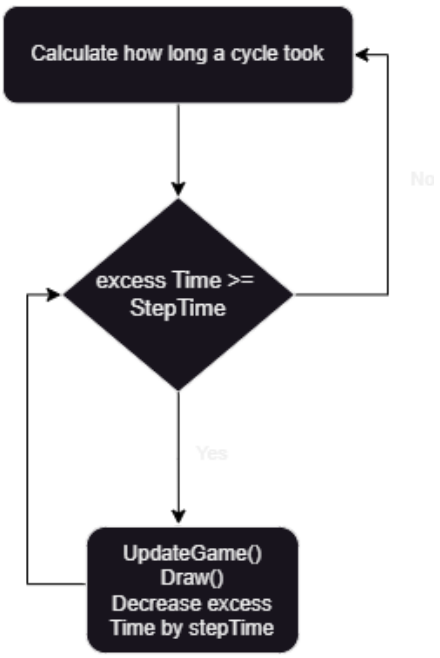


Figura 3. Diagrama exemplo de um *Game Loop* que usa deltaTime

A lógica física ao estar dependente do tempo entre cada frame gerado, o jogo acaba por compensar os atrasos. Ou seja, para o exemplo da caminha anterior, se o computador for mais lento o personagem faz menos passos, se o computador for mais rápido a personagem faz mais passos. No entanto, em ambos os computadores o personagem faz o mesmo percurso no mesmo tempo, como mostrado na seguinte figura:

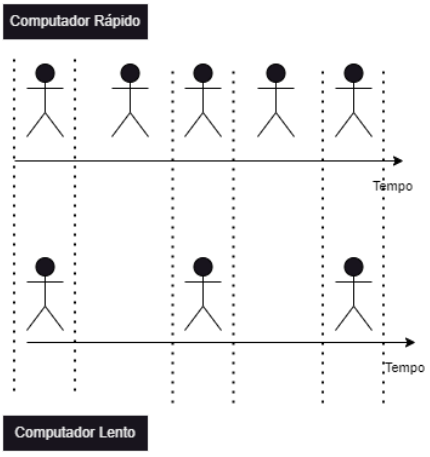


Figura 4. Exemplo de simulação de física com *deltaTime.*

3. Desenvolvimento da Aplicação

Neste capítulo, refere-se de uma forma simples como a aplicação foi estruturada e o desenvolvimento das classes mais abstratas. Como já referido no capítulo de Introdução, a realização deste projeto utilizou-se a linguagem de programação *JAVA* e a biblioteca *Processing*. Deste modo este jogo foi feito sem uso de motores de jogo, que são ferramentas muito úteis que facilitam o desenvolvimento de jogos ao ter forma de representar elementos gráficos e simulação de física. O que tornou o desenvolvimento deste jogo uma tarefa complexa. Foram desenvolvidas classes que permitem a simulação de físicas de corpos rígidos como a classe *RigidBody* e para a componente gráfica a do jogo utilizou-se biblioteca Processing. A modelação deste jogo foi um processo muito iterativo pelo que ao longo do desenvolvimento de jogo quando se foi adicionando novo conteúdo, foi preciso remodelar algumas partes. No entanto, a implementação da maior parte das classes foi feita de forma muito modelada, o que facilitou o processo de desenvolvimento. O seguinte UML exemplifica o modelo do jogo:

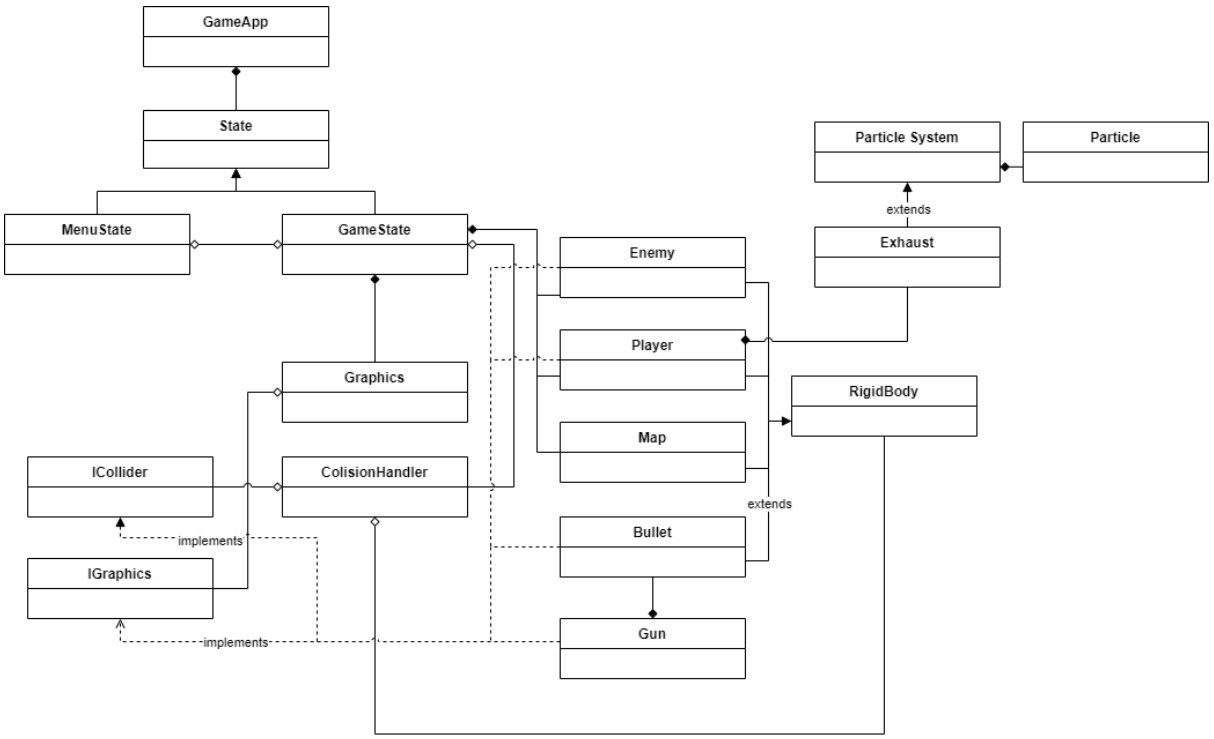


Figura 5. Modelo UML do jogo

Nos próximos subcapítulos irá aprofundar-se com mais detalhe no desenvolvimento de cada módulo do jogo, neste uml optou-se por ocultar a maior parte das propriedades e métodos de cada classe para permitir a visualização de como cada classe interage entre si, e como já referido, nos próximos capítulos explicar detalhadamente cada classe e as suas dependências.

3.1 RigidBody:

A classe RigidBody como a classe Player que será explicada no próximo capítulo, foram as primeiras classes a serem desenvolvidas. Como já referido, não há simulação de física já feita nas ferramentas de desenvolvimento utilizadas. Deste modo o RigidBody é uma classe abstrata que tem como função simular movimentos físicos já a colisão é controlada pela classe *CollisionHandler*. Para isto tem atributos como os vetores posição, velocidade e aceleração. Também tem atributos como massa, velocidade máxima, e raio. Neste jogo não é prioritária uma simulação realista de movimento, mas sim um movimento divertido e rápido. Deste modo, sempre que o player começa o seu movimento há uma aceleração extra para permitir que o player rapidamente chegue a altas velocidades. Já desde o ínicio de desenvolvimento que se pretendia que o player e os inimigos fossem naves no espaço, logo o atrito não é muito elevado.
O diagrama seguinte pretende exemplificar a implementação do movimento no caso mais geral, sendo que a classe Player e a classe Enemy diferem no cálculo da aceleração no entanto a base da movimentação é a mesma:

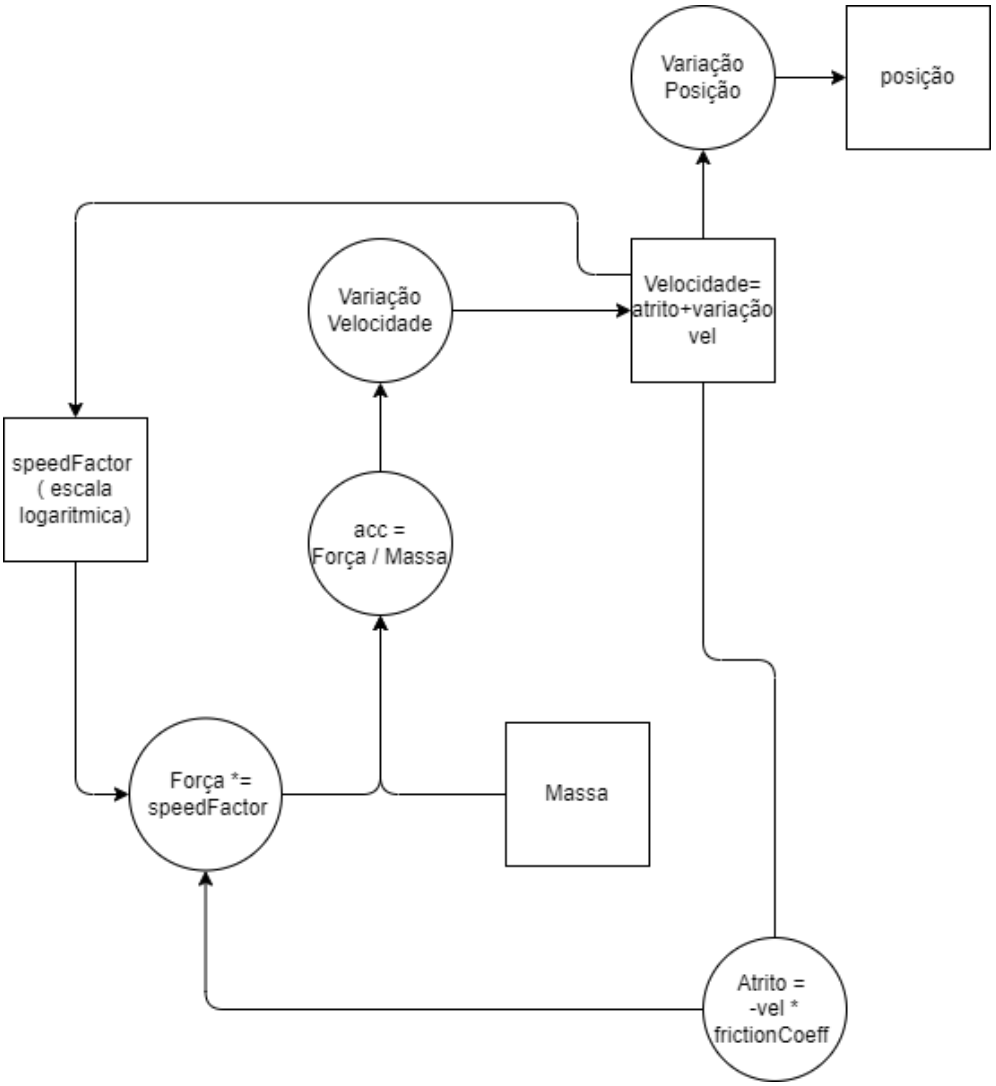


Figura 6. Diagrama Stock and Flow da Movimentação

O movimento implementando tem como inspiração os jogos que usam o motor de jogo Source. Nestes jogos o movimento do jogador tem uma aceleração extra no início do movimento, fazendo com que a velocidade no início seja rápida, e gradualmente o aumento da velocidade é menor até chegar ao limite máximo. Também há força de atrito pelo que se o jogador parar de mexer, o boneco a ser controlado só pará passado uns segundos.

A movimentação de RigidBody para RigidBody difere de como a aceleração é calculada. Por exemplo a aceleração do player tem em conta as teclas pressionadas, já a aceleração do inimigo depende do comportamento do boid. Deste modo fez-se um caso geral em que a força é definida por cada filho:

Listagem 1

```
public void move(float dt, ControlType ct, PApplet p) {
        switch (ct) {
            case POSITION:
                break;

            case VELOCITY:
                vel.add(acc.mult(dt));
                pos.add(PVector.mult(vel, dt));
                // acc.mult(0);
                break;

            case FORCE:
                force(dt, p);
                break;
        }
    }
```

Neste trabalho o movimento através de posição não foi implementado. Esta classe também permite a distinção da posição de ecrã e a posição de world:

Listagem 2

```
    public PVector getWorldPosition() {
        PVector posScreen = pos.copy();
        posScreen.x /= MapSettings.CELL_SIZE;
        posScreen.y /= MapSettings.CELL_SIZE;
        return posScreen.copy();
    }

    public PVector getScreenPosition() {
        return pos.copy();
    }
```

Como o mundo é uma grelha que resulta da divisão do ecrã com a resolução 1920x1080 em células de tamanho 16x16, A conversão é feita ao dividir a posição atual pelo tamanho da célula.

Assim fica implementada com sucesso a classe rigidBody pois permite que os corpos se movimentam no espaço. O modo de movimentar muda ligeiramente de corpo para corpo pelo que cada filho define a função abstrata force() de acordo com o necessário para o seu movimento. O movimento está dependente do deltaTime.

3.2 Player:

A classe Player representa o objeto que será controlado pelo jogador. O jogador controla esta classe através do teclado, usando as teclas típicas de controle de movimento: WASD e para disparar o botão esquerdo do rato.

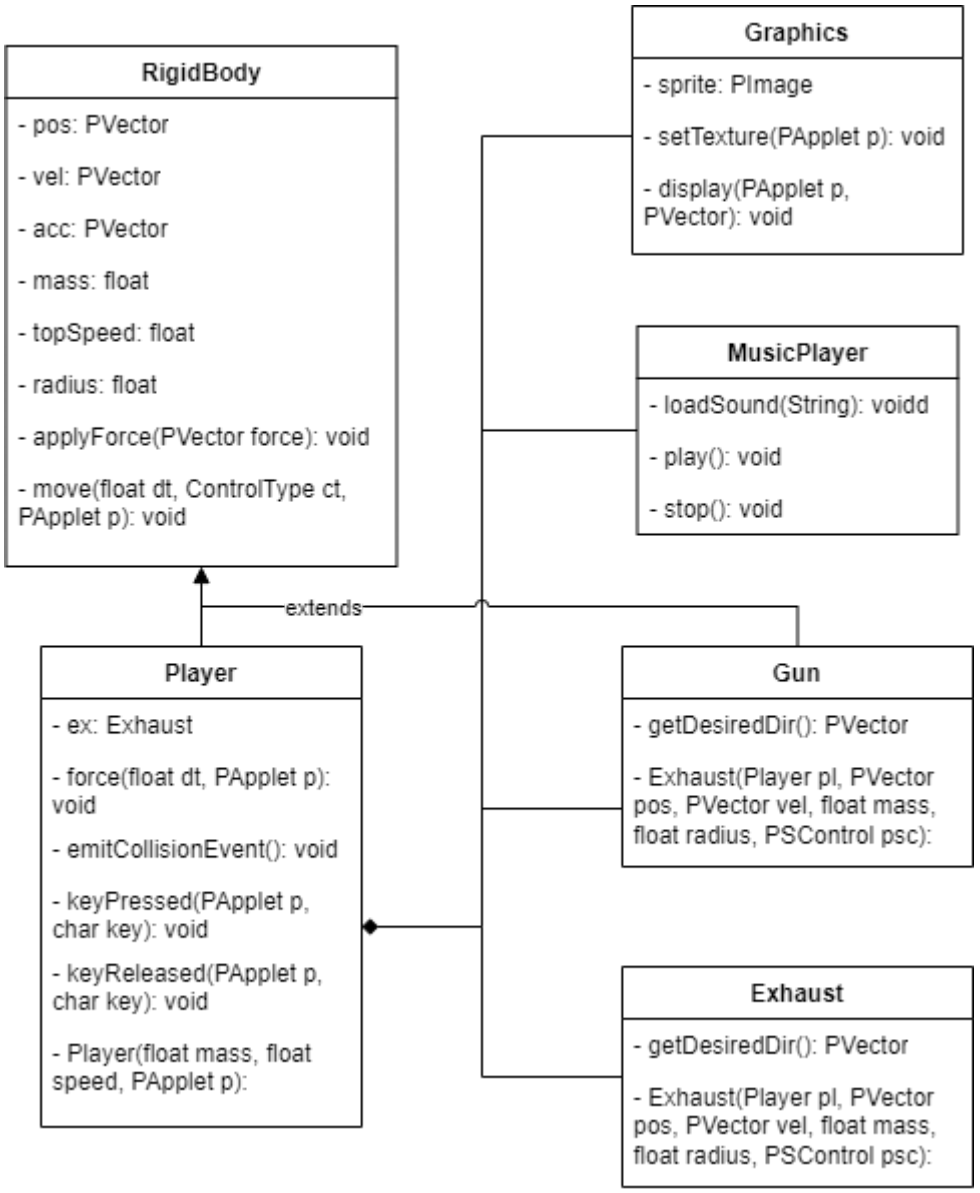


Figura 7. Modelo UML da classe *Player*

A maior diferença do player para as outras entidades do jogo é que o player recebe o input do jogador, para isto a GameApp passa o input recebido do teclado para a classe Player que guarda o valor das teclas pressionadas numa lista. A aceleração é calculada consoante o vetor de direção que é calculado consoante as teclas pressionadas. O tratamento do input é o seguinte:

Listagem 3

```
   public void keyPressed(PApplet p, char key) {
        if (key == 'w' || key == 'a' || key == 's' || key == 'd')
            keyBeingPressed.add(p.key);

    }

    public void keyReleased(PApplet p, char key) {

        if (keyBeingPressed.contains((Character) key))
            keyBeingPressed.remove((Character) key);
        else
            return;

    }
```

O vetor de direção é calculado da seguinte forma:

Listagem 4

```
protected PVector desiredDir() {
        PVector desiredDir = new PVector();
        for (Character key : keyBeingPressed)
            desiredDir.add(dictDir.get(key));

        wishedDir = desiredDir.copy().normalize();
        return wishedDir;
    }
```

Tendo o vetor de direção calculado o movimento do player é calculado da seguinte forma:

Listagem 5

```
   @Override
    protected void force(float dt, PApplet p) {
        PVector frictionForce = PVector.mult(vel, -frictionCoeff);
        displayVectors(p, d, frictionForce, 255, 255, 255);

        vel.add(frictionForce);
        pos.add(PVector.mult(vel, dt));
        vel.add(PVector.mult(acc, dt));

        PVector newForce = PVector.mult(desiredDir(), speed *
accelerationBoost);


        // Apply logarithmic acceleration decay based on current speed
        float speedFactor = 1.5f - PApplet.map(vel.mag(), 0, maxSpeed,
0, 1);
        newForce.mult(speedFactor);



        applyForce(newForce);

        vel.limit(maxSpeed);


        ex.run(dt, p);
        gun.onShooting(shooting, p);
        gun.update(dt, p);

        if ((int) vel.mag() >= 10){
            mp.play();
            mp.amp(0.4f); }
        else
            mp.stop();

    }
```

3.3. Inimigos

Os inimigos do jogo são agentes autônomos, neste caso, agentes boids. Deste modo, estes inimigos têm uma "inteligência" muito básica mas que decidem por si o que querem fazer de acordo com os comportamentos pré-estabelecidos para a classe. Assim como o jogador, os inimigos são uma entidade física que têm colisão e movimentam-se pelo mundo. Por este motivo tanto a classe RigidBody e CollisionHandler são reutilizadas para a simulação da física e colisões. A estrutura do código é a seguinte:

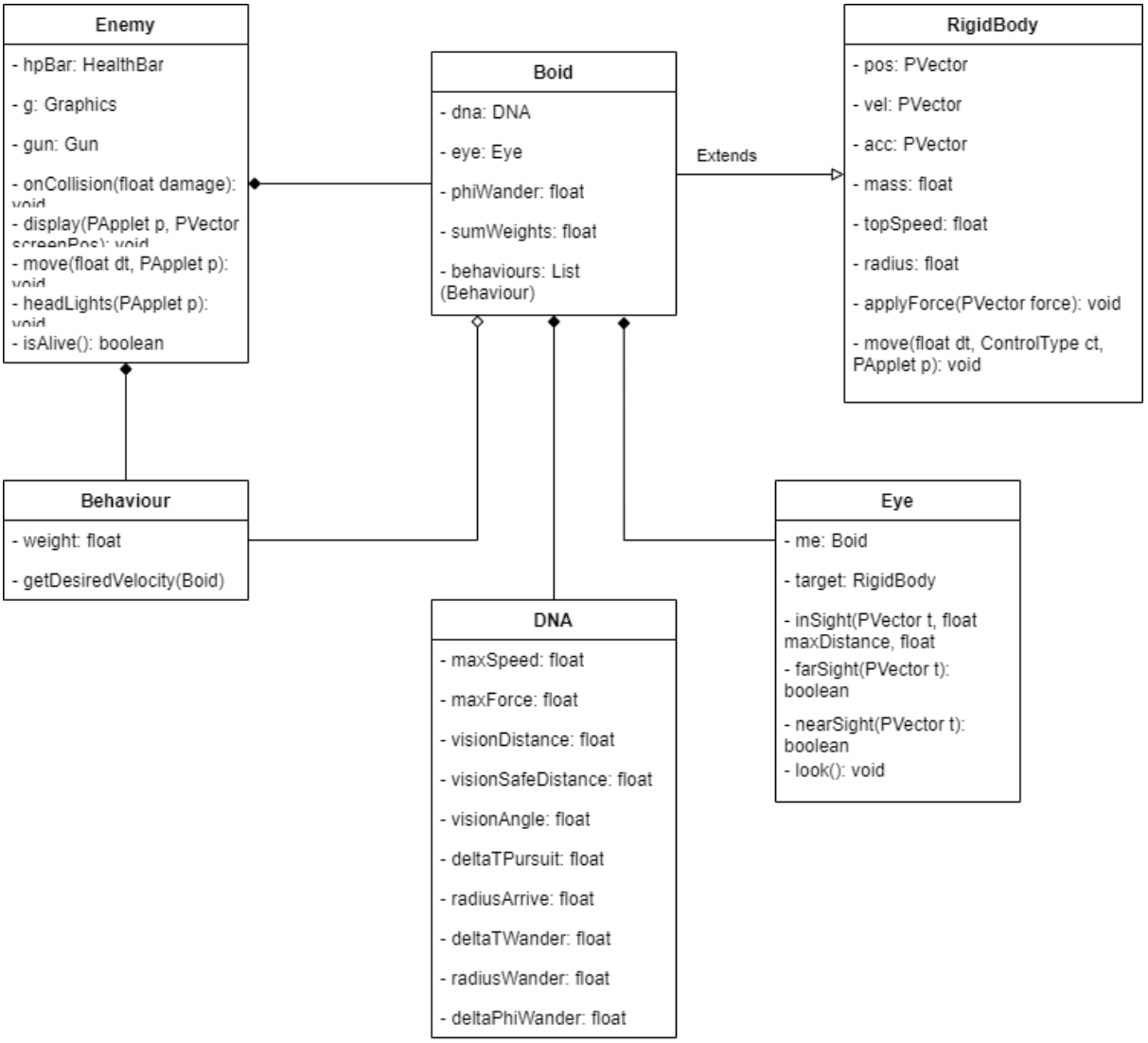


Figura 10. *UML da classe Enemy*

Dos comportamentos que foram implementados ao longo do semestre como mostra a seguinte figura:

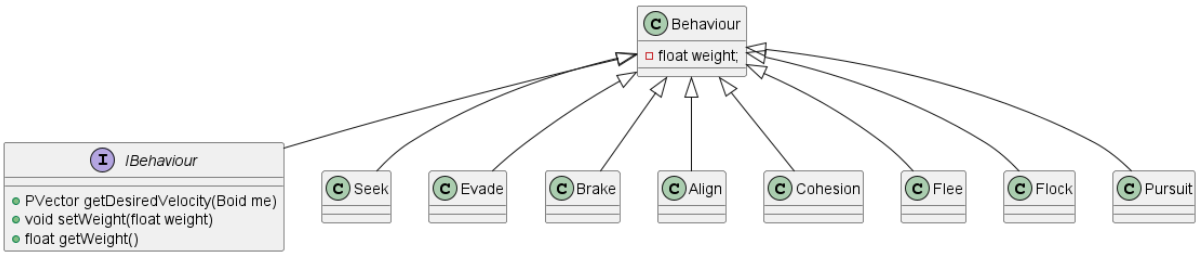


Figura 10. *UML da classe Enemy*

Foram utilizados os comportamentos Seek e Wander. O comportamento seek calcula a posição do player e atualiza a sua velocidade consoante a posição deste. No comportamento do wander é escolhido um ponto aleatório duma circunferência com centro na posição atual do boid.

A implementação destes comportamentos é a seguinte:

Listagem 6

```
   @Override
    public PVector getDesiredVelocity(Boid me) {
        RigidBody bodyTarget = me.eye.target;
        if (me.getEye().inSight(me.eye.target.getWorldPosition(),
me.dna.getVisionDistance(), me.dna.getVisionAngle()))
            return PVector.sub(bodyTarget.getScreenPosition(),
me.getScreenPosition());
        else
            return new PVector();
    }
```

Listagem 7

```
   @Override
    public PVector getDesiredVelocity(Boid me) {
        PVector center = me.getWorldPosition();
        center.add(me.getVelocity().mult(me.dna.getDeltaTWander()));

        float x = me.dna.getRadiusWander() * (float)
Math.cos(me.phiWander);
        float y = me.dna.getRadiusWander() * (float)
Math.sin(me.phiWander);
        PVector target = new PVector(x,y);
        target.add(center);
        me.phiWander += 2 *  (Math.random()-0.5 ) *
me.dna.getDeltaPhiWander();
```

```
        return PVector.sub(target, me.getWorldPosition());
    }
```

Para além destes comportamentos, se o player estiver na visão do inimigo este irá disparar contra o player.
A implementação do inimigo foi feita com sucesso, o inimigo demonstra comportamentos complexos a partir de comportamentos simples.

3.4 Spawner de Entidades

Para permitir que o spawn de inimigos e do player seja feita de forma aleatória e facilitar a geração de níveis foi criado a classe EnemyManager. Os métodos que permitem o spawn do player e do inimigo são o seguinte:

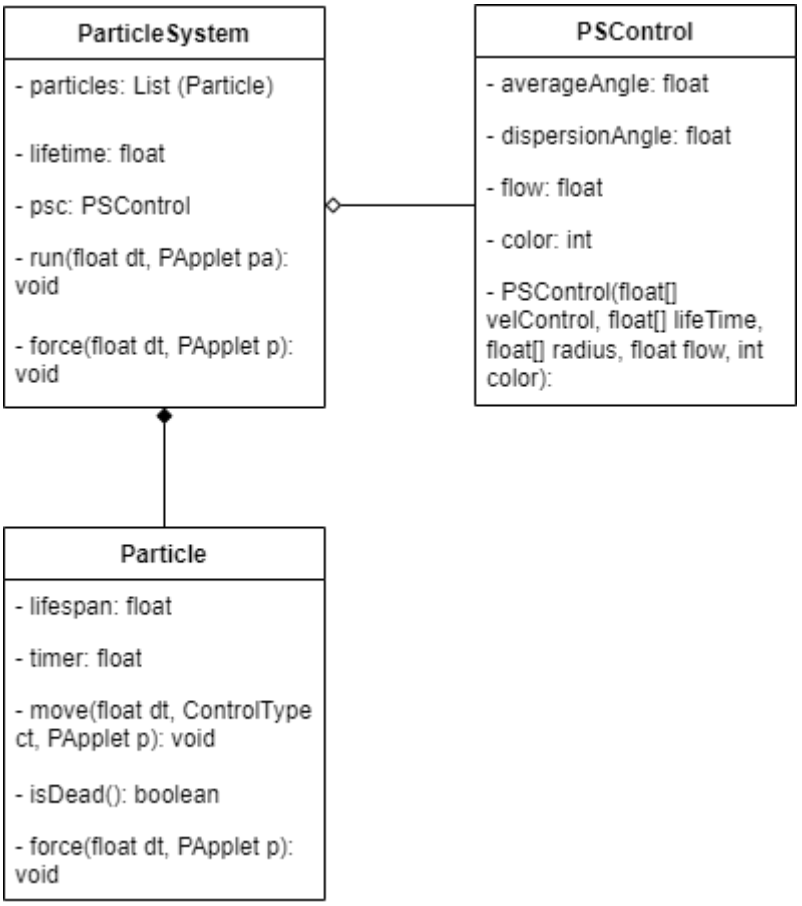
Listagem 8

```
  public Player spawnPlayer(PApplet p) {
        int sum = 0;
        for (int x = (int) p.random(2, col); x < col; x++) {
            for (int y = (int) p.random(2, line); y < line; y++) {
                if (map[x][y].getState() > 0.5f) {
                    sum++;
                    if (sum >= 4) {
                        playerSpawn = new PVector(x *
MapSettings.CELL_SIZE, y * MapSettings.CELL_SIZE);
                        Player player = new Player(playerSpawn, 50f,
p);
                        trackingBodies.add(player);

                        return player;
                    }
                }

            }
        }

        return null;

    }

    public ArrayList<Enemy> spawnEnemies(int enemyAmount, PApplet p) {
        enemiesSpawned = new ArrayList<>();
        int enemySpawned = 0;
        int sum = 0; // spawn on an average of 4 empty cells
        int maxAttempts = 1000; // Avoid infinite loops
        int attempts = 0;

        while (enemySpawned < enemyAmount && attempts < maxAttempts) {
            int x = (int) p.random(2, col);
            int y = (int) p.random(2, line);

            // Check if the tile is valid for enemy spawn
            if (map[x][y].getState() > 0.5f &&
                    !isInside(x, y, (int) playerSpawn.x /
```

```
MapSettings.CELL_SIZE,
                            (int) playerSpawn.y / MapSettings.CELL_SIZE,
safeRadius)) {
                sum++;
                if (sum >= 8) {//bruteforcing it
                    Enemy enemy = new Enemy(new PVector(x *
MapSettings.CELL_SIZE, y * MapSettings.CELL_SIZE),
                            p.random(50, 200), MapSettings.CELL_SIZE,
p);
                    Eye eye = new Eye(enemy, trackingBodies);
                    enemy.setEye(eye);

                    trackingBodies.add(enemy);

                    enemiesSpawned.add(enemy);
                    enemySpawned++;
                    sum = 0;
                }
            }

            attempts++;
        }

        if (attempts >= maxAttempts) {
            System.out.println("Warning: Could not spawn all enemies
within max attempts.");
        }

        return enemiesSpawned;
    }
```

3.5 Sistema de Partículas - Exaustor

O exaustor da nave é um sistema de partículas que foi reutilizado de trabalhos anteriores. O sistema de partículas segue o seguinte modelo uml:



A classe de exaustor serve como abstração entre o sistema de partículas e as entidades do jogo.
O sistema de partículas só é ativo se houver movimento da nave:

Listagem 9

```
public void run(float dt, PApplet pa) { // as par

        super.move(dt, ControlType.FORCE, pa);
        if (isMoving)
            addParticles(dt);
        for (int i = particles.size() - 1; i >= 0; i--) {
            Particle p = particles.get(i);
            p.move(dt, ControlType.VELOCITY, pa);
            if (p.isDead()) {

                particles.remove(i);
            }
        }
```

```
    }
```

O sistema em sí sofre forças físicas enquanto que as partículas não

```
   @Override
    protected void force(float dt, PApplet p) {
        PVector frictionForce = PVector.mult(vel, -frictionCoeff);
        displayVectors(p, d, frictionForce, 255, 255, 255);
        displayVectors(p, d, getDesiredDir(), 0, 255, 0);
        vel.add(frictionForce);
        pos.add(PVector.mult(vel, dt));
        vel.add(PVector.mult(acc, dt));

        PVector newForce = PVector.mult(getDesiredDir(), speed *
accelerationBoost);

        // Apply logarithmic acceleration decay based on current speed
        float speedFactor = 1.5f - PApplet.map(vel.mag(), 0, maxSpeed,
0, 1);
        newForce.mult(speedFactor);

        applyForce(newForce);

        vel.limit(maxSpeed);

        if (vel.mag() >= movingTreshold)
            isMoving = true;
        else {
            isMoving = false;
        }
    }
```

4. Geração de Níveis

Os níveis são gerados de forma procedural. Utilizou-se a técnica estudada dos Autómatos Celulares para gerar estruturas semelhantes a cavernas. Para formar o mapa, as células mortas representam as zonas com colisões e as células vivas as zonas sem colisões, tendo assim dois estados: {0,1}. Deste modo implementou-se a classe *Cell*, classe *CellullarAutomataGenerator* e a classe *Map*. A modelação destas classes é a seguinte:

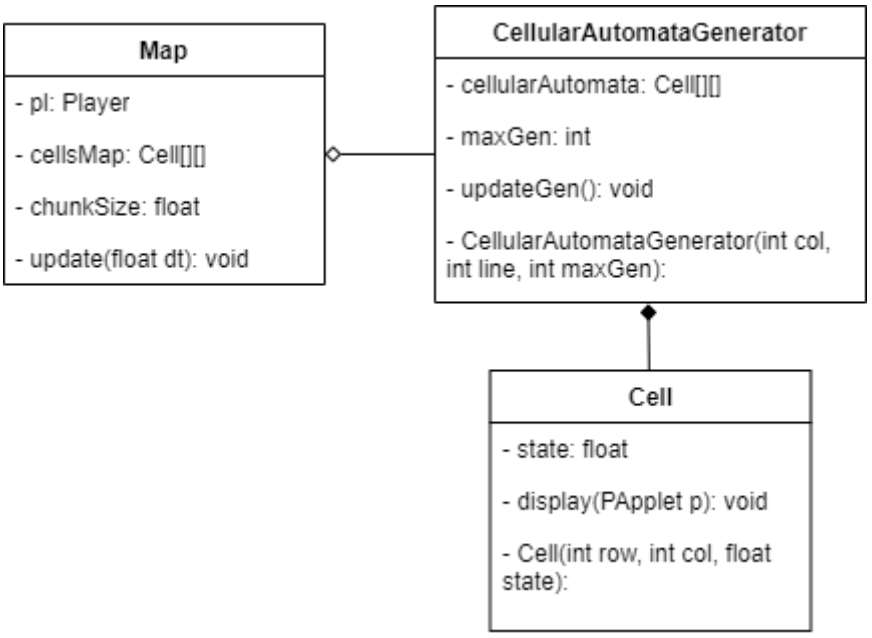


Figura 7. Modelo UML da classe *Map*

Após a geração do nível através do *CellullarAutomataGenerator* a classe map fica com a informação do mapa gerado. Para gerar o mapa é aleatoriamente inserido células vivas ou mortas. Depois de geração em geração, o estado de uma célula é atualizado de acordo com o estado das células à sua volta: se houver mais células vivas que o valor mínimo necessário, a célula permanece viva, se não morre.

Para o cálculo das células vizinhas é necessário percorrer à volta da célula num raio - *r,* definido por vizinhança de Moore que é definido pela seguinte fórmula:

$$(2r + 1)^2 - 1$$

Uma forma de visualizar a vizinhança de uma célula para raio 1 (2*1+1)2-1 = 8 é a seguinte:

	x - 1	x	x + 1
y - 1	x -1 , y - 1	x , y - 1	x + 1 , y - 1
y	x - 1 , y	x , y	x + 1 , y
y + 1	x - 1 , y + 1	x , y + 1	x + 1 , y + 1

Figura 8. Modelo UML do jogo

Para a geração do Autómato Celular, o algoritmo imA figura acima mostra como se pode obter uma vizinhança de uma célula com raio 1. No entanto ao implementar este método é preciso ter cuidado com os cantos pois, as células estão armazenadas num array, e se a célula (x,y) for por exemplo a primeira célula, quando se aceder a célula (x-1,y-1), esta não existe no array. Uma solução, e a implementada, é ignorar estas zonas.

O algoritmo implementado é o seguinte:

1. Preencher o mapa de forma aleatória com células vivas ou mortas.
2. Calcular o estado seguinte de uma célula: se tiver 2 vizinhos sobrevive ou nasce, senão morre.

A implementação deste algoritmo em código é a seguinte:

Listagem 10

```
   private void resetAutomata(PApplet p) {
        cells = new Cell[col][line];
        Random random = new Random();

        for (int x = 0; x < col; ++x) {
            for (int y = 0; y < line; ++y) {
                int state = (random.nextFloat() < fillPercent ? 0 : 1);
                cells[x][y] = new Cell(x, y, state, p);
            }
        }
    }
```

```
   private int GetNeighbourCellCount(int x, int y) {
        int neighbourCellCount = 0;
        int[] dx = { -1, 0, 1, -1, 1, -1, 0, 1 }; // -1,0 e 1 há 8
pontos possiveis (-1,-1), (0,-1), (1,-1) etc
        int[] dy = { -1, -1, -1, 0, 0, 1, 1, 1 };

        for (int i = 0; i < dx.length; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if (nx >= 0 && nx < col && ny >= 0 && ny < line) {
                neighbourCellCount += cells[nx][ny].getState();
            }
        }
        return neighbourCellCount;
    }

    public void updateGen(PApplet p) {
        if (currentGen < maxGen) {
            Cell[][] caBuffer = new Cell[col][line];

            for (int x = 0; x < col; ++x) {
                for (int y = 0; y < line; ++y) {
                    int liveCellCount = GetNeighbourCellCount(x, y);

                    int newState = liveCellCount >
liveNeighboursRequired ? 1 : 0;
                    caBuffer[x][y] = new Cell(x, y, newState, p);
                }
            }
```

```
            cells = caBuffer; // Replace the entire grid with the
updated one
            currentGen++;
        }
    }
```

Após o mapa ser gerado este é copiado para a classe Map que controla o mapa até o fim do jogo. Uma otimização feita é que o mapa é mostrado de chunks em chunks:

Listagem 11

```
   @Override
    public void display(PApplet p, PVector screenPos) {
        p.pushStyle();

        PVector plWorldCoord = pl.getWorldPosition();

        int startX = (int) (plWorldCoord.x - chunkSize);
        int startY = (int) (plWorldCoord.y - chunkSize);
        int chunkX = (int) (plWorldCoord.x + chunkSize);
        int chunkY = (int) (plWorldCoord.y + chunkSize);

        for (int x = startX; x < chunkX; x++)
            for (int y = startY; y < chunkY; y++) {
                int xIdx = x;
                int yIdx = y;
                if (xIdx <= 0)
                    xIdx = 0;
                if (xIdx >= p.width / MapSettings.CELL_SIZE)
                    xIdx = p.width / MapSettings.CELL_SIZE - 1;
                if (yIdx <= 0)
                    yIdx = 0;
                if (yIdx >= p.height / MapSettings.CELL_SIZE)
                    yIdx = p.height / MapSettings.CELL_SIZE - 1;
                cellsMap[xIdx][yIdx].display(p);// x * cellSize, y *
cellSize); // uma unidade neste mundo e 8x8 que o
                                                // tamanho da celula. If
statements to handle borders

            }

        p.popStyle();
    }
```

5. Sistema de colisões

No sistema de colisões decidiu-se implementar um sistema que faz a detecção de frame a frame recorrendo a posição do objeto. Isto significa que o sistema implementado funciona e é bastante eficiente, no entanto não é muito preciso. Em teoria, se os objetos forem rápidos o suficiente podem entrar dentro de objetos sólidos, pois não se está a ter em conta a próxima posição do objeto, apenas a posição atual, este fenómeno denomina-se como *tunneling.* No entanto, no jogo à velocidade que os objetos se movem, isto não acontece.
Todos os Objetos que se mexem podem ser representados por um círculo. Para dois círculos com o mesmo raio e que estejam lado a lado, sem espaço vazio entre si, é fácil de visualizar que a soma dos seus raios é igual ao diâmetro de um dos círculos, como se pode ver na seguinte figura:

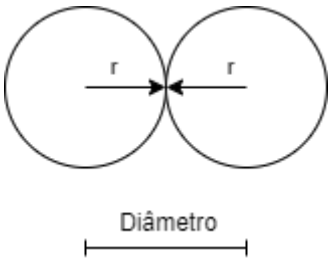


Figura 9. Exemplo de círculos lado a lado

No exemplo acima, os círculos estão a tocar-se mas ainda não estão sobrepostos. Deste modo, a soma dos raios, que neste exemplo é igual ao diâmetro de um dos círculos por terem o mesmo raio, é a menor distância que os círculos podem estar entre si sem estarem sobrepostos e consequentemente sem haver colisão. Isto significa que se calcularmos a distância entre os círculos e a qualquer momento esta distância é menor que a soma dos raios, então os círculos estão sobrepostos e há colisão. Esta forma de pensar funciona para o jogo pois todos os corpos que implementam a classe *RigidBody* possuem um raio e são representados por um círculo. A figura na próxima página demonstra dois cenários diferentes de colisão:

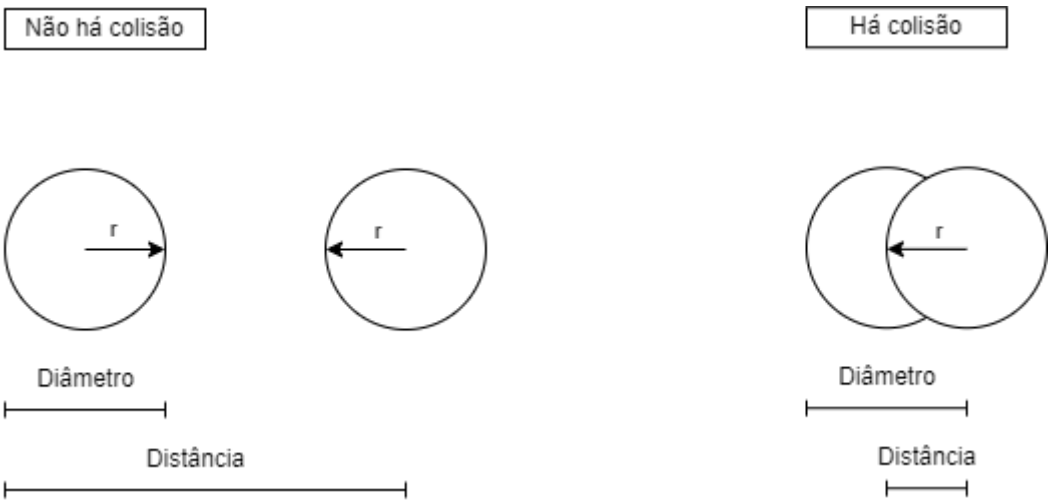


Figura 9. Exemplo de círculos

No exemplo referido anteriormente os círculos têm o mesmo raio para a visualização do problema ser mais fácil, no entanto, esta fórmula funciona para círculos com raio diferente. Por outro lado as células do mapa também têm colisões, mas não são círculos mas sim quadrados. Felizmente esta solução também funciona entre quadrados e círculos, basta pensar no lado do quadrado a dividir por 2 como o raio do quadrado, como pode ser observado na figura abaixo:

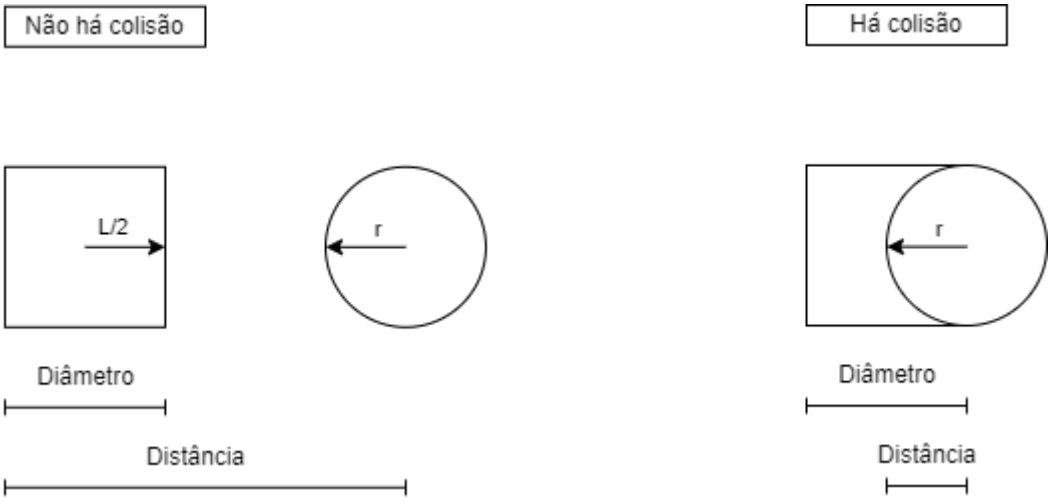


Figura 9. Exemplo de círculo e quadrado

É fácil de observar que perde-se um pouco de precisão ao usar este sistema de detectar colisões, pois a diagonal do quadrado é maior que metade do lado do quadrado.. Para implementar este sistema, como o mapa é constituído por quadrados que são objetos da classe célula decidiu-se dividir o sistema de colisões em duas verificações: uma para as colisões entre os objetos da classe *RigidBody* e outra para as colisões entre a classe *RigidBody*

e a classe *Cell.* O sistema implementado, mesmo tendo em conta as desvantagens, é mais que suficiente para o tipo de jogo.

O método que verifica a colisão entre objetos RigidBody é a seguinte:

Listagem 12. *Listagem do método* checkBodyCollision

```
private void checkBodyCollision(RigidBody currentRigidBody, ArrayList<RigidBody> rbList)
{
        for (RigidBody nextRigidBody : rbList) {
            if (currentRigidBody.equals(nextRigidBody))
                continue;

            int dist = Utils.intDist((int) currentRigidBody.getWorldPosition().x,
                    (int) currentRigidBody.getWorldPosition().y, (int)
nextRigidBody.getWorldPosition().x,
                    (int) nextRigidBody.getWorldPosition().y);

            int sumRadius = (currentRigidBody.getRadius() / MapSettings.CELL_SIZE +
nextRigidBody.getRadius())
                    / MapSettings.CELL_SIZE;

            if (dist <= sumRadius) {
                /* Collision Resolution*/

    }
```

Infelizmente no final do projeto foi preciso adicionar mais e mais condições pelo que talvez com mais abstração fosse possível evitar os vários if-statements da resolução de colisões. Pelo menos manteve-se um código legível e funcional.

O método que verifica a colisão entre o objeto RigidBody e as células do mapa é a seguinte:

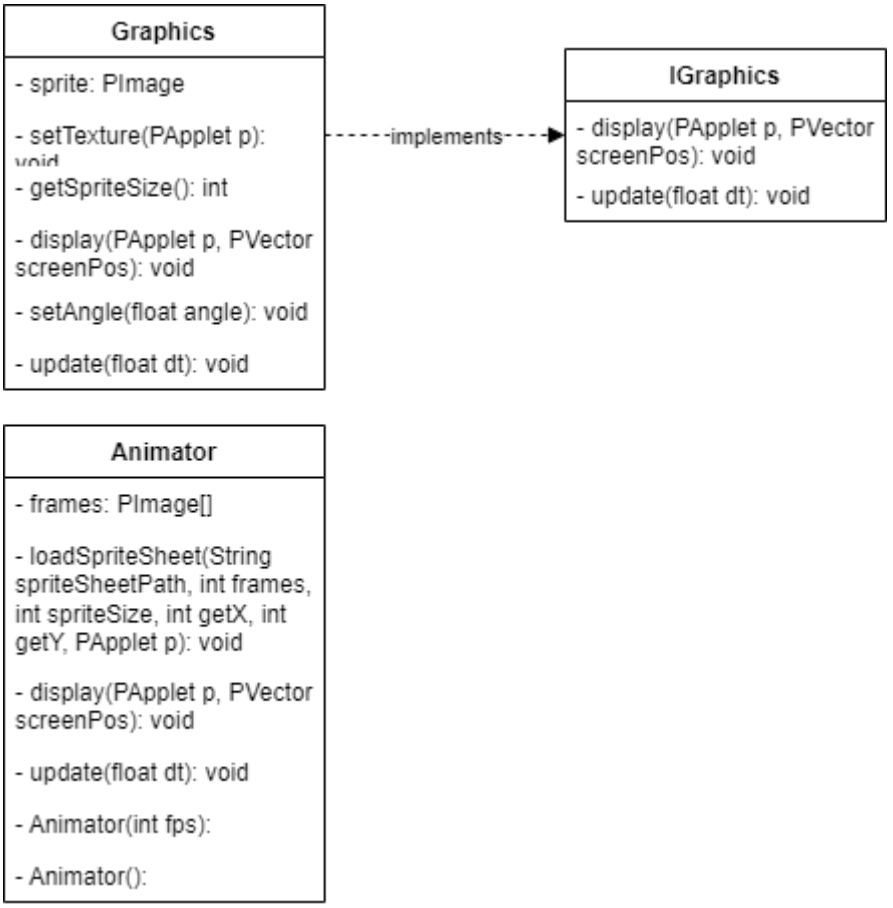
Listagem 13. *Listagem do método* `checkMapCollision`

```
public void checkMapCollision(RigidBody b, Iterator<RigidBody>
bodiesIterator, PApplet p) {
        PVector bodyWorldPos = b.getWorldPosition();
        if (b instanceof Bullet && outOfBounds(bodyWorldPos)) {
            removeBulletCollision()

            return;
        } else if (outOfBounds(bodyWorldPos)) {
            b.multVel(-1);
            return;
        }

        Cell cell = map.getCell((int) bodyWorldPos.x, (int)
bodyWorldPos.y);
        // cell.displayCollider(p); debugging
        int bodyX = (int) bodyWorldPos.x, bodyY = (int) bodyWorldPos.y;
        int cellX = cell.getX(), cellY = cell.getY();

        int dist = Utils.intDist(cellX, cellY, bodyX, bodyY);
        int sumRadius = (MapSettings.CELL_SIZE / 2 + b.getRadius()) /
MapSettings.CELL_SIZE;

        if (dist <= sumRadius && cell.getState() <= 0.5f) {
            if (b instanceof Bullet) {
                removeBulletCollision()
            }

            b.multVel(-1);
        }
    }
```

Como pode-se ver no método acima, para além das colisões já referidas, a classe CollisionHandler é responsável por manter os objetos em jogo dentro do mapa, pelo que se aproximarem das bordas e tentarem sair são empurrados para dentro. Também é responsável por desligar a colisão de uma bala que tenha colidido com um corpo e marcar a bala como desativada para que a arma que disparou a mesma bala possa depois remover a bala de jogo.

6. Gráficos

Assim como foi feito para o sistema de colisões, optou-se por criar uma classe que controla-se a maior parte das responsabilidades gráficas. Para alcançar este fim realizou-se as classes Graphics e Animator e a interface IGraphics como pode-se observar no uml:



A classe graphics não é muito complexa, para mostrar uma imagem primeiro é preciso carregar o sprite e de seguida para expor no ecrã é necessário chamar a função display que a sua implementação é a seguinte:

Listagem 14

```
@Override
 public void display(PApplet p, PVector screenPos) {
     p.pushMatrix();
     p.imageMode(PApplet.CENTER);

     p.translate(screenPos.x, screenPos.y);
     p.rotate(angle);
```

```
        p.image(sprite, 0, 0);

        p.popMatrix();

    }
```

7. Arte Utilizada

A arte gerada em jogos 2d tem tipicamente tamanhos de potências de dois, sendo o comprimento e altura tipicamente iguais por exemplo: um sprite pode ser 32*32 ou 32*64 ou 128*128 mas o comprimento e a altura costumam ser potências de 2 para facilitar o uso em várias resoluções. No início do projeto foram criados dois designs: um para a nave do jogador e um para a nave do inimigo exemplificados nas seguintes figuras:

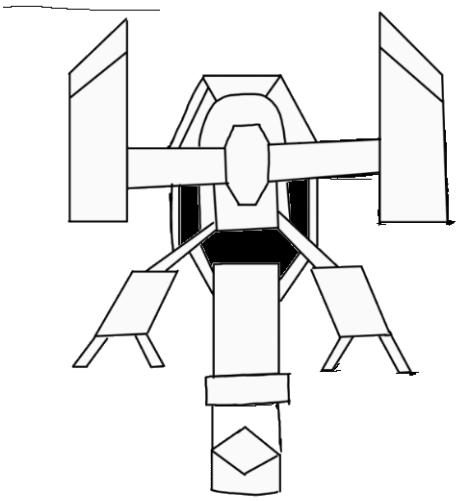


Figura 11: Design Inicial do Inimigo

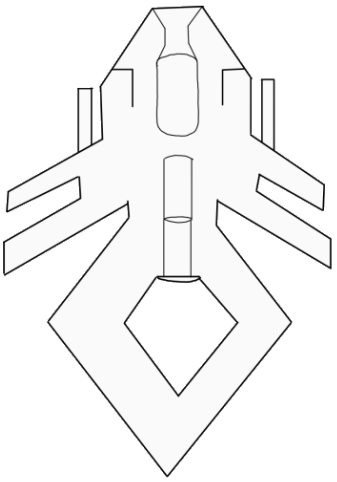


Figura 12: Design Inicial do Jogador

Estes designs ao serem passados para formato digital tem artefatos, no entanto não foram corrigidos pois já estava decidido que o espaço seria preto, o que esconderia estes artefatos.
No entanto estes designs não foram utilizados pois optou-se por mudar para arte 2d PixelArt e utilizar arte já feita disponibilizada de forma gratuita em sites como itch.io

9. Multiplayer

Foi adicionado um modo de jogo multiplayer antes da discussão do trabalho final. O modo multiplayer consiste em dois jogadores ambos controlados pelo teclado. O objetivo é os players lutarem entre si

8. Conclusões

Neste projeto foi feito um jogo com 2 modos de jogo diferentes. Várias vezes durante o processo de desenvolvimento foi necessario alterar o scope do jogo reduzindo-o. No entanto, o jogo final ficou funcional e esteticamente bonito. Foi reutilizando bastante código dos trabalhos anteriores.

9. Bibliografia/Webgrafia

1. Daniel Shiffman. The Nature Of Code. https://natureofcode.com/
2. RogueBasin Basic BSP Dungeon Generation. https://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation
3. Wikipedia contributors. Cellular automaton. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Cellular_automaton.
4. Bronsons procedural generation with CA https://bronsonzgeb.com/index.php/2022/01/30/procedural-generation-with-cellular-automata/
5. Wikipedia contributors. Autonomous agent. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Autonomous_agent.
6. Wikipedia contributors. Boids. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Boids
7. RogueBasin CA Cave levels http://www.roguebasin.com/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels
8. Wikipedia contributors. FloodFill: https://en.wikipedia.org/wiki/Flood_fill
9. Vídeos disponibilizados no moodle da cadeira de MSSN
10. Size of Sprites https://www.youtube.com/watch?v=MrPoCGHM80E
11. Collision Detection https://www.youtube.com/watch?v=oOEnWQZIePs
12. Building Collision Simulations https://www.youtube.com/watch?v=eED4bSkYCB8
13. Chunks https://www.youtube.com/watch?v=DIYTWcq3wpY
14. 2d Camera https://www.youtube.com/watch?v=DIYTWcq3wpY
15. Screen to world coords https://www.youtube.com/watch?v=coEbwmQaoe4

16. **Sprite sheets for 2D animation in p5j:**

    https://www.youtube.com/watch?v=eE65ody9MdI&ab_channel=morejpeg

17. **Random Tree Generation:**

    https://www.youtube.com/watch?v=DTrB8Qqt14Q&ab_channel=VivekGupta

18. **Source Game Engine: https://developer.valvesoftware.com/wiki/Source**