Gabriel Meira, Lucas Gabriel, Thiago Calebe, Vítor Paixão

Trabalho 2
Laboratório de AEDS

Belo Horizonte, Brasil

2024

1 Introdução

O objetivo deste trabalho é implementar um player que se movimenta pelo caminho mais rápido no mapa, utilizando um algoritmo de pathfinding. O player deve ser capaz de se mover por diferentes tipos de terreno (grama, areia, água) e evitar obstáculos (pedras, cactus, corais). Um barco deverá ser obtido durante o jogo, permitindo que o player navegue na água com velocidade dobrada.

2 Desenvolvimento

2.1 A classe Player

Figura 1 – Sprite do jogador

Iniciamos o trabalho com a implementação de uma classe `Player`. O código a seguir mostra uma versão simplificada dessa classe:

Listing 1 – Simplificação da classe Player

```
1 class Player {
2   PVector pos, origem, destino;
3   Stack<PVector> caminho;
4   int caminhoIndex;
5   float velocidade, velocidadeFator = 5;
6   int[][] grid;
7   boolean hasBoat, hasMap, flipped;
8   PImage sprite, boatSprite;
9   int woodsGotten = 0;
```

```
10    float animation = 0;
11
12    // Construtor
13    Player(float x, float y){...}
14
15    // Definir grid de busca
16    void setGrid() {...}
17
18    // Algoritmo A*
19    Stack<PVector> aEstrela(PVector destino) {...}
20
21    // Traducao de coordenadas
22    PVector translateGridPosition(PVector gridPosition) {...}
23    PVector translateToGridPosition(PVector mapPosition) {...}
24
25    // Atualizar e exibir o player
26    void update() {...}
27    void show() {...}
28  }
```

Os métodos setGrid e aEstrela serão usados na lógica de pathfinding.

2.2 Algoritmo de pathfinding

Escolhemos o A* para ser nosso algoritmo de pathfinding. Como parâmetro para o algoritmo, iremos passar uma matriz quadrada contendo os valores de cada bloco. O tamanho da matriz é definido pela variável `areaDeBusca`. Para encontrar esse valor calculammos o eixo com a maior distância entre o player e o mouse, e então adicionamos um valor de tolerância extra para garantir a eficácia do algoritmo.

Listing 2 – Cálculo da área de busca

```
1  void mouseReleased() {
2    if (mouseButton == LEFT) {
3      int deltaX = (int) abs(player.pos.x - map.gridPosX(mouseX));
4      int deltaY = (int) abs(player.pos.y - map.gridPosY(mouseY));
5
6      areaDeBusca = ((deltaX > deltaY) ? deltaX : deltaY) * 2 +
           toleranceRange;
7
8      player.setGrid();
9      PVector m = player.translateToGridPosition(new PVector(map.
           gridPosX(mouseX), map.gridPosY(mouseY)));
```

```
10      player.destino = new PVector(map.gridPosX(mouseX), map.
            gridPosY(mouseY));
11      player.caminho = player.aEstrela(m);
12
13      updateScreen();
14    }
15  }
```

A matriz grid então é criada a partir dos valores que estão dentro da área de busca que definimos.

```
1  void setGrid() {
2    grid = new int[areaDeBusca][areaDeBusca];
3    origem = new PVector(pos.x, pos.y);
4    for (int x = 0; x < areaDeBusca; ++x) {
5      for (int y = 0; y < areaDeBusca; ++y) {
6        grid[x][y] = map.getTileValue(-areaDeBusca/2+x +(int)origem
             .x, -areaDeBusca/2+y+(int)origem.y);
7      }
8    }
9  }
```

Os métodos player.translateToGridPosition() e player.translateGridPosition() seguem os mesmos moldes dos métodos gridPos() e screenPos() da classe Map. Entretanto, ao invés de relacionar a posição da tela à posição no mapa, eles relacionam a posição no mapa à posição na matriz de busca do algoritmo A*.

```
1       PVector translateGridPosition(PVector gridPosition) {
2      // Coordenadas globais no grid do mapa
3      int globalX = (int)origem.x - areaDeBusca / 2 + (int)
           gridPosition.x;
4      int globalY = (int)origem.y - areaDeBusca / 2 + (int)
           gridPosition.y;
5      return new PVector(globalX, globalY);
6    }
7
8    PVector translateToGridPosition(PVector mapPosition) {
9      // Calcula a posicao local no grid do Player
10     int localX = (int)mapPosition.x - (int) origem.x +
           areaDeBusca / 2;
11     int localY = (int)mapPosition.y - (int) origem.y +
           areaDeBusca / 2;
12
```

```
13      // Certifica-se de que as coordenadas estao dentro da area de
            busca
14      localX = constrain(localX, 0, areaDeBusca - 1);
15      localY = constrain(localY, 0, areaDeBusca - 1);
16
17      return new PVector(localX, localY);
18    }
```

Essa matriz então é dada como parâmetro para o método `aEstrela`. Seu funcionamento gira em torno de dois tipos de valores, os pesos, valores de cada tile, e a distância real do atual tile ao destino. Os pesos são definidos pela letra G e a distância, H, enquanto sua soma, valor de avaliação, como F. Para rastrear o caminho, o algoritmo faz uso de duas listas, uma aberta e a outra fechada, sendo uma de tiles que não foram explorados e a outra que foram, juntamente com outra lista que guarda de qual nó um outro nó veio. O programa prioriza selecionar tiles que possuem um baixo F, aqueles seriam mais próximos do destino. Segue o código da implementação:

Listing 3 – Implementação do algoritmo A*

```
1       Stack<PVector> aEstrela(PVector destino) {
2      if (obst.contains(WATER) && obst.contains(SHALLOW_WATER)) {
3        obst.set(obst.indexOf(WATER), -1);
4        obst.set(obst.indexOf(SHALLOW_WATER), -1);
5      }
6      caminhoIndex = 0;
7      caminho = new Stack<PVector>();
8
9      //Valor do peso
10     HashMap<PVector, Float> gScore = new HashMap<>();
11     //Valor da distancia
12     HashMap<PVector, Float> hScore = new HashMap<>();
13     //Soma do peso e a distancia (valor final)
14     HashMap<PVector, Float> fScore = new HashMap<>();
15     //No que leva ao outro
16     HashMap<PVector, PVector> pais = new HashMap<>();
17
18     // Inicializa as listas de abertos e fechados
19     PriorityQueue<PVector> abertos = new PriorityQueue<>(new
           Comparator<PVector>() {
20       public int compare(PVector p1, PVector p2) {
21         return Float.compare(fScore.getOrDefault(p1, Float.
               MAX_VALUE), fScore.getOrDefault(p2, Float.MAX_VALUE));
22       }
```

```
23      }
24      );
25
26      HashSet<PVector> fechados = new HashSet<>();
27
28      // Adiciona a posicao inicial (centro da grid) aos abertos
29      PVector inicio = new PVector(areaDeBusca / 2, areaDeBusca /
           2);
30      abertos.add(inicio);
31
32      // Inicializa os scores
33      gScore.put(inicio, 0.0f);
34      hScore.put(inicio, dist(inicio.x, inicio.y, destino.x,
           destino.y));
35      fScore.put(inicio, hScore.get(inicio));
36
37      while (!abertos.isEmpty()) {
38        // Encontra o nodo com o menor fScore (PriorityQueue faz
             isso automaticamente)
39        PVector atual = abertos.poll();
40
41        // Se o nodo atual e o destino, reconstruir o caminho
42        if (atual.equals(destino)) {
43          Stack<PVector> caminhoAux = new Stack<PVector>();
44          while (pais.containsKey(atual)) {
45            caminhoAux.add(atual);
46            atual = pais.get(atual);
47          }
48          caminhoAux.add(inicio); // Adiciona o inicio ao caminho
49          Collections.reverse(caminhoAux); // Inverte o caminho
               para comecar do inicio
50          return caminhoAux;
51        }
52
53        // Move o nodo atual dos abertos para os fechados
54        fechados.add(atual);
55
56        // Verifica os vizinhos ortogonais (nao diagonais)
57        int[] dx = { -1, 1, 0, 0 };
58        int[] dy = { 0, 0, -1, 1 };
59
60        for (int k = 0; k < 4; k++) {
```

```
61          PVector vizinho = new PVector(atual.x + dx[k], atual.y +
                dy[k]);
62          PVector gridVizinho = translateGridPosition(vizinho);
63          if (!isObstacle(map.getTileValue((int)gridVizinho.x, (int
                )gridVizinho.y))) {
64
65            if (vizinho.x < 0 || vizinho.x >= areaDeBusca ||
                  vizinho.y < 0 || vizinho.y >= areaDeBusca) continue;
66            if (fechados.contains(vizinho)) continue;
67
68            float vizinhoValue = map.getTileValue((int)gridVizinho.
                  x, (int)gridVizinho.y);
69
70            if (hasBoat && (vizinhoValue == WATER || vizinhoValue
                  == SHALLOW_WATER)) {
71              vizinhoValue = 0.5;
72            }
73
74
75
76            float tentativeGScore = dist(atual, vizinho)*
                  vizinhoValue;
77
78            if (!abertos.contains(vizinho) || tentativeGScore <
                  gScore.getOrDefault(vizinho, Float.MAX_VALUE)) {
79              // Atualiza o caminho para o vizinho
80              pais.put(vizinho, atual);
81              gScore.put(vizinho, tentativeGScore);
82              hScore.put(vizinho, dist(vizinho.x, vizinho.y,
                    destino.x, destino.y));
83              fScore.put(vizinho, gScore.get(vizinho) + hScore.get(
                    vizinho));
84
85              // Adiciona o vizinho a lista de abertos
86              if (!abertos.contains(vizinho)) {
87                abertos.add(vizinho);
88              }
89            }
90          }
91        }
92      }
93
```

```
94      setGrid();
95      caminhoIndex = caminho.size();
96      // Retorna uma lista vazia se nao houver caminho
97      return new Stack<>();
98    }
```

Dado a velocidade do jogador, definida pelo tipo de tile em que ele está, ele se move mais rápido ou mais devagar: na grama, ele se move proporcionalmente a uma velocidade de tamanho 1, enquanto na areia de ½ e na água de 2. Para que isso seja visualizado é usado uma pequena condição em que se checa se a velocidade do jogador condiz com o tempo de jogo, como pode ser visto abaixo:

Listing 4 – Controle da velocidade

```
1     if (time % player.velocidade == 0) player.update();
```

Tendo a condição estabelecida, uma função dentro de player é chamada, a de `update()`. Em `update()`, o programa irá passar por cada posição definida pelo caminho dado pelo algoritmo A*, seguindo um índice para rastrear em qual posição o jogador está. Segue o código implementado abaixo:

Listing 5 – O método player.update()

```
1  if (caminhoIndex<caminho.size()) {
2        //Traduz a posicao da grid de busca para a grid do map
3        PVector aux = translateGridPosition(caminho.get(
            caminhoIndex));
4        //Remove a posicao anterior
5        caminho.set(constrain(caminhoIndex-1, 0, caminho.size()),
            new PVector(-1, -1));
6
7        //Checa se o tile e agua
8        float value = map.getTileValue((int)aux.x, (int)aux.y);
9        if (value==WATER || value==SHALLOW_WATER) value = .5;
10
11       //Checa para qual lado o jogador esta virado
12       if (pos.x-aux.x < 0) flipped = true;
13       else if (pos.x-aux.x > 0)flipped = false;
14
15       //Define sua posicao e atualiza a tela
16       pos = aux;
17       updateScreen();
18
19       //Recalcula sua velocidade para o proximo tile
```

```
20        velocidade = value*velocidadeFator;
21        ++caminhoIndex;
22      } else {
23        setGrid();
24        caminhoIndex = 0;
25        caminho = new Stack<PVector>();
26      }
```

2.3 Centralização do player

Uma booleana cameraSeguindo foi criada para indicar se a camêra está ou não seguindo o player. Ela é ativada ao soltar a tecla 'P'.

Listing 6 – Ativação da câmera

```
1 void keyReleased() {
2   switch (key) {
3   case 'p':
4     if (cameraSeguindo) cameraSeguindo = false;
5     else cameraSeguindo = true;
6     break;
7   }
8 }
```

Na função `draw()`, o offset da câmera é ajustado para a posição atual do personagem.

```
1 if (cameraSeguindo) {
2   offset.x = width / 2 - (int) player.pos.x * tileSize;
3   offset.y = height / 2 - (int) player.pos.y * tileSize;
4 }
```

2.4 Barco

Utilizamos outra abordagem para a criação do barco, que não envolve coloca-lo em uma posição aléatória no mapa para ser obtido. Ao invés disso, fizemos com que o barco fosse construido após obter madeiras espalhadas pelo mapa. Para rastrear a posse do barco, foi usada a booleana `hasBoat`, que é atributo de `Player`

Listing 7 – A classe Boat

```
1 class Boat{
2   PVector pos;
3
4   Boat(float x, float y){
```

```
5       pos = new PVector(x, y);
6     }
7
8     void show() {
9       float screenX = pos.x * tileSize + offset.x;
10      float screenY = pos.y * tileSize + offset.y;
11      noStroke();
12      fill(#984712);
13      rect(screenX, screenY, tileSize, tileSize);
14    }
15  }
```

Figura 2 – Sprite do jogador com um barco

2.5 Definição de um objetivo

Para combinar com a temática de pirata, escolhemos como objetivo do jogo encontar um tesouro enterrado. Para isso, é o jogador precisa coletar um número de madeiras espalhadas pelo mapa para construir um barco. Em seguida, ele deve coletar um mapa que revelará a posição do tesouro, e por fim coletar o tesouro e ganhar o jogo.

A classes Treasure, Paper e Wood são semelhantes em atributos e métodos, se diferenciando apenas pelo sprite usado. Todos os itens são colocados no mapa no início do jogo, porém o mapa só é mostrado após coletar as madeiras, e o baú só é mostrado ao coletar o mapa.

Listing 8 – A classe Treasure

```
class Treasure {
  PVector pos;
  boolean visible;

  Treasure(float x, float y) {
    pos = new PVector(x, y);
    visible = true;
  }

  void show() {
    if (visible) {
      float screenX = pos.x * tileSize + offset.x;
      float screenY = pos.y * tileSize + offset.y;

      fill(0);
      PVector icon = new PVector(constrain(screenX, tileSize,
         width-tileSize*2), constrain(screenY, tileSize, height-
         tileSize*2));
      imageMode(CENTER);
      image(treasureSprite, icon.x+tileSize/2.0, icon.y+tileSize
         /2.0, tileSize*1.5, tileSize*1.5);

      if (screenX < 0 || screenX > width || screenY < 0 ||
         screenY > height) {
        noFill();
        stroke(255);
        strokeWeight(3);
        ellipse(icon.x+tileSize/2.0, icon.y+tileSize/2.0,
           tileSize*2.0, tileSize*2.0);
      }
    }
  }
}
```

Após o baú é coletado, uma mensagem de vitória aparece na tela.

Listing 9 – Tela final

```
void endingScreen(){
  background(#110B27);
  fill(255);
  image(treasureSprite, width/2.0, height/2.0-width/15.0, width
     /10.0, width/10.0);
```

```
5     text("VOCE ENCONTROU O TESOURO!", width/2.0-width/7.0, height
          /2.0);
6   }
```

Figura 3 – Tela final

3 Resultados

Para a função draw(), optamos por ecrever o mínimo de código possível e priorizar chamadas de função, aplicando os conceitos de modularização:

Listing 10 – Função Draw()

```
1   void draw() {
2     if (!endGame) {
3       updateScreen();
4
5       if (mousePressed && mouseButton == RIGHT) {
6         map.drag((width / 2.0 - mouseX) / 10.0, (height / 2.0 -
              mouseY) / 10.0);
7       }
8
9       if (time%player.velocidade==0) player.update();
10
11      if (cameraSeguindo) {
12        offset.x = width / 2 - (int) player.pos.x * tileSize;
13        offset.y = height / 2 - (int) player.pos.y * tileSize;
14      }
15
16      if (player.woodsGotten == nWood) {
17        player.hasBoat = true;
18        player.woodsGotten = 0;
19      }
20
21      if (player.hasBoat) {
```

```
22        paper.show();
23      }
24
25      if (player.hasMap) {
26        treasure.show();
27      }
28
29      showWoodCounter();
30    }else{
31      endingScreen();
32    }
33    ++time;
34  }
```

A seguir, apresentamos algumas mídias que demonstram diferentes estados do jogo:

Vídeo do jogo em funcionamento

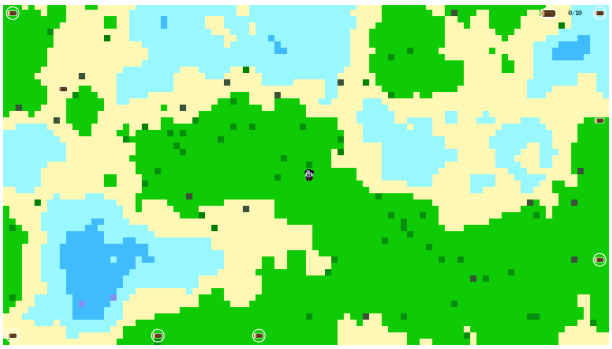

Figura 4 – Plano incial.

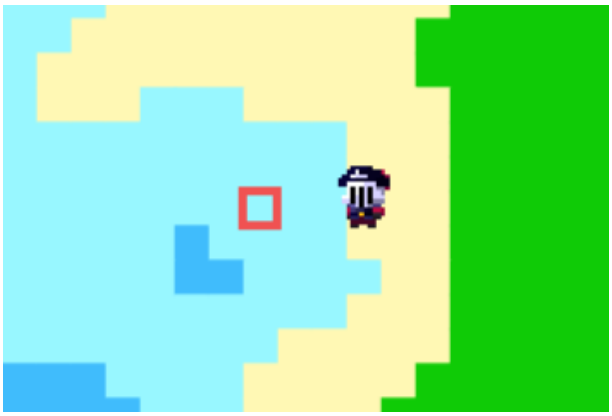

Figura 5 – Jogador não consegue nadar.

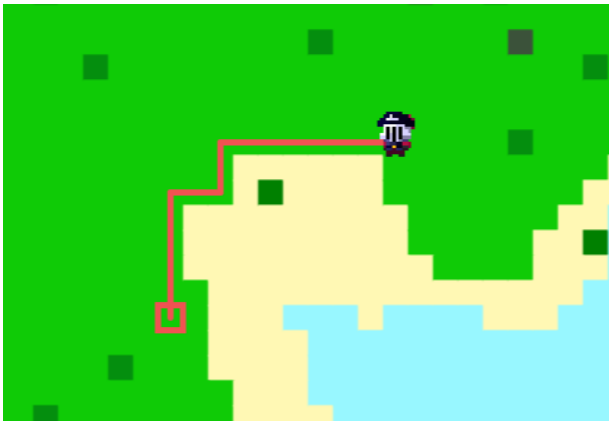

Figura 6 – Jogador calculando menor caminho.

Figura 7 – Jogador com barco.

4 Conclusão

Os resultados encontrados mostraram alcançar todos os objetivos do propostos do experimento. Usamos da liberdade criativa que nos foi concedida para melhorar alguns aspectos de implementação proposta incialmente, mas ainda demonstrando todos os requisitos técnicos.

O maior desafio encontrado se resumiu em adaptar o algoritmo A* para o contexto do programa. Este foi superado após trabalharmos em conjunto para corrigir erros e encontrar soluções criativas para implementação.

A experiência adquirida com a prática se mostrou valiosa, consolidando nossos conhecimentos sobre estruturas de dados como o hashMap e o algoritmo A*.