Pedro Inácio Rodrigues Pontes, Vitor Mendes, Sergio Amaral, Felipe D'Avilla

Trabalho 02: Estouradores de CLT

Belo Horizonte, Brasil

2024

1 Introdução

1.1 Do Objetivo

O seguinte trabalho foi construído com base no seguinte imperativo:

"O objetivo deste trabalho é implementar um player que se movimenta pelo caminho mais rápido no mapa, utilizando o algoritmo de Dijkstra. O player deve ser capaz de se mover por diferentes tipos de terreno (grama, areia, água) e evitar obstáculos (pedras, cactus, corais). Um barco será colocado aleatoriamente no mapa, permitindo que o player navegue na água com velocidade dobrada. Adicionalmente, a implementação do algoritmo A* para a movimentação do player será recompensada com pontos extras."

Ou seja, o objetivo é construir um ambiente com um player que se movimenta por um grid infinito, que tem diferentes terrenos, auxiliado pelos algoritmos *A e Dijkstra para alcançar um ponto utilizando possivelmente o menor caminho.

1.2 Dos Requisitos

Os requisitos se extendem por quatro campos principais:

- Implementação do Player
- Implementação do Algoritmo de Dijkstra
- Adicionar um Barco
- Ações no Mapa

Cada um deles foi obedecido a partir de imperativos, dados a seguir.

1.2.1 Implementação do Player

- "Crie uma classe ‘Player‘ que inclua atributos para posição, velocidade e um indicador de posse do barco.
- Inicialize o player em uma posição padrão no mapa (por exemplo, no centro).
- O player deve se mover do ponto inicial até o ponto clicado no mapa, utilizando o algoritmo de Dijkstra.
- O player pode andar na grama e na areia, mas a velocidade na areia é reduzida pela metade. O player pode viajar na água apenas quando está de posse de um barco, viajando na água com o dobro da velocidade de quando anda na grama.
- O player não pode atravessar obstáculos (pedras, cactus, corais).

Para a implementação da velocidade, considere que o movimento do player é de 1 bloco do grid por segundo quando na grama, 0,5 blocos no grid por segundo quando na areia e 2 blocos do grid por segundo quando na água com barco."

1.2.2 Implementação do Algoritmo de Dijkstra

- "Implemente o algoritmo de Dijkstra para calcular o caminho mais curto.
- Utilize pesos para os diferentes tipos de terreno, o peso de uma aresta é dada pela média do peso do dois vértices que formam a aresta, O pedo do vértice é dado pelo tipo de terreno:
  - Água: 1 com barco e infinito sem barco
  - Grama: 2
  - Areia: 3"

1.2.3 Adicionar um Barco

- "Coloque um barco em uma posição aleatória do mapa, mas não mais distante do que 100 blocos do ponto inicial do player.
- Se o player pegar o barco, ele pode se mover na água."

1.2.4 Ações no Mapa

- "Ao apertar a tecla 'P', o mapa deve ser centralizado no player."

1.3 Dos Objetivos de Aprendizagem

- Entender as diferenças entre os algoritmos *A e Dijkstra.
- Entender as implementações e usos dos algoritmos de menor caminho no mundo real, como no Waze e Google Maps.
- Aplicar o conhecimento de grids para trabalhar em um ambiente com grid infinito.

2 Desenvolvimento

2.1 Route

2.1.1 Objetivo

Classe criada para resolver o problema de criar as menores rotas para o jogador usar, com tal podendo escolher Dijkstra ou *A para encontrar as rotas. Também resolve o problema

de armazenar as rotas que o jogador criar por si.Cumpre o requisito da implementação dos algoritmos de menor caminho e parte da implementação do player (gera a sequência de movimentos que ele seguirá para fazer a menor rota de A até B)

2.1.2 Atributos

```
int onRoute; // Route status, 0 - off, 1 - ON, 2 - haveOrigin, 3 -
    haveDestiny, 4 - chose Dijstraka or A*, 5 - execute
int lines, cols;
int fromX, fromY, toX, toY; //Grid
int startX, startY;
int destinyV, originV;
int verticesNum; //quantidade de blocos presentes na tela
float[][] adjMatrix;
ArrayList<Integer> path;
float cost, time;
char alg;
```

- onRoute: Armazena o estado da rota: 0 - off, 1 - on, 2 - tem origem, 3 - tem destino, 4 - escolher algoritmo, 5 - executar
- lines cols: Armazenam o número de linhas e colunas do mapa.
- fromX fromY: Armazenam as coordenadas x e y da origem da rota, relacionados ao mapa.
- toX toY: Armazenam as coordenadas x e y do destino da rota, relacionados ao mapa.
- startX e startY: Armazena a coordenada inferior esquerda do retângulo da rota. Tal coisa é feita porque o retângulo sempre é desenhado com base no seu vérice inferior esquerdo. O cáculo para ela é feito em setLimites().
- destinyV, originV: Armazenam os vértices de origem de destino da rota, relacionados à matriz adjacente criada para o caminho.
- verticesNum: quantidade de blocos presentes na tela.
- adjMatrix: Matriz adjacente criada para representar o caminho. É um retângulo criado a partir dos vértices de origem e destino do caminho.
- path: ArrayList contendo o caminho gerado pela máquina ou jogador da origem até o fim da rota.
- cost: Armazena o custo de se passar por certa rota, a qual é a rota que é instanciada como objeto da presente classe.

- time: Armazena o tempo gasto para realizar certa rota. Não é efetivamente trabalhada em Route, apenas em Game, principalmente, e Widgets.
- alg: Armazena o algoritmo escolhido. 'a' = *A 'd' = Dijkstra.

2.1.3 Métodos

Segue a listagem dos métodos presentes em route:

- Route() –> Construtor
- void traceRoute(char m) –> Traça uma rota baseado no caminho definido e algoritmo escolhido.
- Route clone(char c) –> Faz uma cópia da rota. Usada dentro do jogo, para haver uma route para player e uma cópia para a máquina.
- void getCoord(int x, int y) –> Define as coordenadas de início e fim da rota. O método infere por si se é a origem ou destino.
- void setLimits(flaot r) –> Seta os limites do novo grid criado a partir do retângulo que tem como vértices a origem de destino da rota. Define o número de linhas e colunas
- int getVertice(int gridX, int gridY) –> Retorna o vértice correpondente a uma posição x, y no grid.
- int getGridX(int vertice) –> Retorna a coordenada x do grid correpondente a um determinado vértice.
- int getGridY(int vertice) –> Retorna a coordenada y do grid correpondente a um determinado vértice.
- void setAdjMatrix() –> Define a matriz adjacente gerada a partir do retângulo de vértices início e fim da rota.
- void generateAdjMatrix() –> Gera uma matriz adjacente a partir da estrutura definida em setAdjMatrix.
- float costCalculator() –> Define o custo do caminho escolhido.
- void dijstraka() –> Utiliza o algoritmo de Dijkstra para definir a menor rota da origem até destino definidos.
- void aStar() –> Utiliza o algoritmo *A para definir a menor rota da origem até destino definidos.
- void display() –> Desenha o caminho criado. Usa uma cor predefinida.

- void display(color c) –> Desenha o caminho criado. Usa a cor dada como parâmetro.
- void makeWay() –> Define como o player deverá se movimentar a partir do caminho gerado.
- void off() –> Define o fim da presente rota, parando o movimento do player, resetando a classe Route e rota do jogo.

2.1.3.1 Route

```
public Route(){
  this.path = new ArrayList<Integer>();
  this.onRoute = 0;
  this.cost = 0;
}
```

Construtor. Apenas inicializa o estágio da rota (onRoute) e o custo dela como 0. Cria uma ArrayList vazia para path.

2.1.3.2 traceRoute

```
 void traceRoute(char m){
   float range = 0.5;
     do{
      setLimits(range);
      setAdjMatrix();
      generateAdjMatrix();
      this.alg = m;
      if (alg == 'a') aStar();
      if (alg == 'd') dijstraka();
      range += 0.5;
     }while (path.isEmpty() && range < 5 && m != '0');
   costCalculator();
 }
```

range corresponde à área de busca que é incrementada cada vez que não se consegue encontrar um caminho viável da origem ao destino. O limite máximo dela é 5, após isso é encerrada a tentativa de encontrar uma rota. Em cada iteração do loop feito enquanto o caminho não foi definido, o range ainda é menor que 5 ou o algoritmo de menor caminho não foi escolhido pelo jogador, é criada uma matriz adjacente da rota, com range ditando o quanto ela irá ser aumentada, perceba que setLimits() é o que realmente leva à definição do tamanho da matriz. O algoritmo também é inicializado como não escolhido, esperando ser

escolhido para chamar ou o *A ou o Dijkstra para tentar encontrar o menor caminho. No fim do loop, que significa que não foi encontrado um caminho viável, range é incrementado em 0.5. Após tudo isso, costCalculator é chamado para determinar o custo/dificuldade de passar pelo caminho definido.

2.1.3.3 setAdjMatrix

```
void setAdjMatrix(){
  this.adjMatrix = new float[verticesNum][verticesNum];
  this.originV = getVertice(fromX, fromY);
  this.destinyV = getVertice(toX, toY);
}
```

Inicializa a matriz adjacente a partir do número de vértices/blocos presentes nela. Inicializa originV e destinV a partir do retorno da função getVertice para a área no mapa onde começa e termina a rota.

2.1.3.4 generateAdjMatrix

```
void generateAdjMatrix(){
  for (int i=0; i<verticesNum; i++){
    for (int j=0; j<verticesNum; j++){
      float vi = map.getTileValue(getGridX(i), getGridY(i));
      float vj = map.getTileValue(getGridX(j), getGridY(j)); //clculo para
          obter elemento em grid a partir de um valor iterador
      vi = (vi==6) ? 1 : vi;
      vj = (vj==6) ? 1 : vj;
      if (((abs(getGridX(i) - getGridX(j)) == 1 && getGridY(i) - getGridY(j)
          == 0) || (abs(getGridY(i) - getGridY(j)) == 1 && getGridX(i) -
          getGridX(j) == 0) ) && adjMatrix[i][j]==0 &&
          (player.allowedTiles.contains(int(vi)) &&
          player.allowedTiles.contains(int(vj)))){ //ligao entre mesmo
          elemento, vizinhos, se for agua tem q ter barco
        adjMatrix[i][j] = ((vi + vj) / 2) + 1; //mdia dos pesos de dois
            vrtices gera peso de aresta, TileValue+1
        adjMatrix[j][i] = adjMatrix[i][j];
      }
    }
  }
}
```

São tratados os valores/pesos para cada vértice, sendo considerado se o jogador possui barco ou não. É verificada a permissão de movimentação do jogador pelo vértice e verificada a vizinhança do mesmo. Após isso é preenchida a matriz de adjacência.

2.1.3.5 dijstraka

```
void dijstraka(){
  float [] dist = new float[verticesNum];
  int [] anterior = new int[verticesNum];
  Arrays.fill(dist, Float.POSITIVE_INFINITY);
  Arrays.fill(anterior, -1);

  dist[originV] = 0;
  float[] Q = new float[verticesNum];

  done: for (int k = 0; k<verticesNum; k++){
    int u=-1;
    float udist = Float.POSITIVE_INFINITY;

    for (int v = 0; v < verticesNum; v++){
      if(Q[v] == 0 && dist[v] < udist){
        u = v;
        udist = dist[v];
      }
    }

    u = abs(u);

    Q[u] = 1;
    for (int v = 0; v < verticesNum; v++){
      if(u==v || adjMatrix[u][v] == 0) continue;

      float alt = udist + adjMatrix[u][v];

      if(alt < dist[v]){
        dist[v] = alt;
        anterior[v] = u;

     if (v == destinyV){
      while (v != -1){
        path.add(v);
        v = anterior[v];
```

```
      }
      Collections.reverse(path);
      break done;
    }

      }
    }
  }
}
```

Variáveis

dist: corresponde a um array de floats que armazena o menor caminho de cada vértice até a origem. É inicializado com Float.POSITIVE_INFINITY para esse fim, assim, os vértices que não foram analisados terão uma distância infinita inicial, como pregado na teoria do algoritmo.

anterior: corresponde a um array de ints que armazena o vértice anterior a cada vértice do caminho mais curto. Inicializado com -1 por ser desconhecido o vértice anterior ao do menor caminho.

Q: é um array de floats que funciona como uma priorityQuee para armazenar os vértices que devem ser processados. Variáveis e inicialização

Loop principal

O loop principal é iterado a partir do número de vértices.

- int u é inicializado com -1. udist é inicializado com valor infinito.
- Dentro do loop interno, é feita uma verificação de se o vértice em questão (Q[v]) já foi processado e tem distância menor até que a origem do vértice atual com menor distância até ela. Em caso dos parâmetros serem atendidos, o valor de u é atualizado com v. E udist recebe a distância da origem até v.
- o valor de u é convertido para absoluto para evitar problemas de sinal.

Processamento do vértice u

Q[u] é marcado com 1 para indicar que o vértice u já foi processado.

É criado um loop interno para verificar se o vértice v é adjacente a u. Em caso afirmativo, é calculada a distância alternativa (alt) entre u e v, somando a distância mínima de u até a origem com a distância de u a v. Se alt for menor que a distância mínima atual de v, a distância mínima de v é atualizada com alt e u recebe o vértice anterior de v.

Condição de parada

Se o vértice v for igual ao destino (destinyV). Após isso é construído o menor caminho, que seria uma interpretação dos resultados obtidos pelo algoritmo até o momento.

Interpretação dos resultados

É construído o menor caminho adicionando os vértices do caminho mais curto à lista path e a revertendo, pois a lista original vai do destino à origem e é desejado o contrário, da origem ao destino.

O algoritmo sai do loop principal com break done.

2.1.3.6 aStar

```
void aStar() {
      // lista aberta
      PriorityQueue<float[]> queue = new
          PriorityQueue<>(Comparator.comparingDouble(a -> a[0]));
      queue.add(new float[]{0, originV});

      float[] dist = new float[verticesNum];
      int[] anterior = new int[verticesNum];
      Arrays.fill(dist, Float.POSITIVE_INFINITY);
      Arrays.fill(anterior, -1);

      dist[originV] = 0;

      done: while (!queue.isEmpty()) {
          // pega o vertice da queue com menor dist (vertice, dist)
          float[] current = queue.poll();
          int currentNode = (int) current[1];

          // chegou no destinyV
          if (currentNode == destinyV) {
              while (currentNode != -1) {
                  path.add(currentNode);
                  currentNode = anterior[currentNode];
              }
              Collections.reverse(path);
              break done;
          }

          // marca na lista "fechada"
```

```
            dist[currentNode] = current[0];

            // itera sobre vizinhos do n atual
            for (int neighbor = 0; neighbor < verticesNum; neighbor++) {
                if (adjMatrix[currentNode][neighbor] > 0) { // se tiver conexo
                    // calcula tentativa do vizinho at origem, atualDist mais
                        distAtVizinho
                    float tentativedist = current[0] +
                        adjMatrix[currentNode][neighbor];

                    // se tentativa for menor que vizinhoDist (at origem) atualize
                    if (tentativedist < dist[neighbor]) {
                        dist[neighbor] = tentativedist;
                        anterior[neighbor] = currentNode;
                        // euclidian heuristic
                        queue.add(new float[]{tentativedist +
                            sqrt(pow(getGridX(neighbor) - getGridX(destinyV), 2) +
                            pow(getGridY(neighbor) - getGridY(destinyV),2)),
                            neighbor});
                    }
                }
            }
        }
}
```

Variáveis

- queue: uma PriorityQueue que armazena os vértices a serem visitados, ordenados pela distância estimada do vértice até o destino.
- dist: array que armazena a distância mínima até cada vértice.
- anterior: armazena o vértice anterior de cada vértice do caminho mais curto.
- path: armazena o caminho mais curto encontrado.

Inicialização

- queue é inicializada com o vértice de origem valendo uma distância de 0.
- dist é armazenado com infinito para cada vértice, exceto o de origem, que recebe 0.
- anterior é inicializado com -1.

Loop Principal

Obs: Distância estimada = distância real mais eurística de Euclides até o destino. A PriorityQueue é ordenada com os menores valores em seu topo, e os maiores no seu fundo.

Enquanto queue não estiver vazia, o vértice com menor distância estimada é removido da fila.

A distância até o vértice removido é marcada como fechada, não sendo mais considerada.

Os vizinhos do vértice em questão são iterados. Se o vizinho possuir uma aresta com ele, a distância alternativa até o vizinho é calculada, e se esta for menor que a menor distância conhecida até o vizinho, tal será atualizada com a distância alternativa. O vizinho é adicionado à PriorityQueue com a distância estimada.

Se o vértice removido for o destino, o caminho é construído a partir dos resultados do algoritmo, depois invertido, para apontar da origem para o destino, e a função acaba.

Heurística de Euclides

Utilizada para estimar a distância até o vértice destino. É calculada como a distância do vértice observado até o destino, mas sem obedecer ao grafo, sendo a menor distância direta do ponto do vértice até o ponto da origem.

Considerações Finais

O A* é basicamente um dijkstra com o peso de cada vértice tendo adicionada a Heurística de Euclides, diminuindo o campo de vértices a serem analisados na maioria dos casos.

2.2 Widgets

2.2.1 Objetivo

Classe criada para resolver o problema da criação e funcionalidade dos botões do jogo, os quais são os responsáveis pelo jogador conseguir interagir com a maioria das funcionalidades do jogo, como o modo Waze e o modo Game.

2.2.2 Atributos

```
private String origB, destB;
private float startX, largBloco, altBloco;
private float startX1 = 10, startY = 10;
private float largBloco1 = width/2 - startX1, altBloco1 = 60;
private float largBloco2 = 54, altBloco2 = 54;
```

- startX: define o eixo X inicial de um botão, é constantemente alterado no código conforme são construídos botões diferentes, pois cada um tem seu eixo X inicial específico.
- startY: define o eixo y inicial de cada botão.
- largBloco: define a largura de um botão.
- altBloco: define a altura de um botão.
- startX1, largBloco1, largBloco2, altBloco1 altBloco2: Constantes que definem alturas e larguras padrões para os botões, usadas pela alta repetição de certas proporções nestes.
- origB destB: armazenam informações sobre o tipo do bloco de origem/destino de certa rota. Ex: Grama, Areia.

2.2.3 Métodos

Segue a listagem dos métodos presentes em widgets:

- private void display() –> Exibe cada botão de acordo com o momento do jogo e interação do jogador.
- boolean btnTrigger(int x, int y) –> Retorna true e atualiza o estágio/informações do jogo quando certo botão é apertado.
- private void gameBtn() –> Desenha o botão "Game".
- private void routeBtn() –> Desenha o botão "Waze".
- private void simulateBtn() –> Desenha o botão "Simulate route".
- private void goBtn() –> Desenha o botão "GO".
- private void stopBtn() –> Desenha o botão "STOP".
- private void aStarBtn() –> Desenha o botão "A*".
- private void dijstrakaBtn() –> Desenha o botão "Dijstraka".
- private void drawButton(String text, color c, float txtX, float txtY) –> Facilita o desenho de botões fazendo tal a partir apenas da passagem de seus parâmetros de texto, cor, e coodenadas x e y do texto.
- private void routeInfo() –> Exibe a informação sobre a rota, detalhando a coordenada do início, a coordenada do fim e o tipo do bloco em que essas coordenadas estão localizadas.
- private void timer(Route r) –> Exibe o tempo gasto pelo player e o tempo gasto pela máquina ao entrar no modo "Game".

- private void gameInfo() –> Exibe as informações dos jogadores no modo "Game".
- private void playerInfo() –> Exibe as informações do player no modo "Game".
- private void machineInfo() –> Exibe as informações da máquina (player 2) no modo "Game".
- private boolean gameBtnV(int x, int y) –> Pequeno guia de como jogar no modo "Game"
- private boolean routeBtnV(int x, int y) –> Retorna se o botão route foi clicado.
- private boolean simulateBtnV(int x, int y) –> Retorna se o botão simulate route foi clicado.
- private boolean goBtnV(int x, int y) –> Retorna se o botão go foi clicado.
- private boolean stopBtnV(int x, int y) –> Retorna se o botão stop foi clicado.
- private boolean aStarBtnV(int x, int y) –> Retorna se o botão *A foi clicado.
- private boolean dijstrakaBtnV(int x, int y) –> Retorna se o botão dijstraka foi clicado.
- private boolean isWithinBounds(int x, int y, float startX, float largBloco, float altBloco) –> A partir do uso de parâmetros, simplifica o ato de descobrir se um botão foi clicado, retornando tal ação. Usada no return de todos os métodos dessa classe terminados em "BtnV".
- void drawX(int screenX, int screenY) –> Desenha um X, utilizado para marcar a rota de início e a rota do fim.

2.3 Player

2.3.1 Objetivo

A classe `Player` é responsável por representar o jogador no mapa e controlar seu movimento e interações com o ambiente. Ela lida com o movimento do jogador, a verificação de condições do terreno, a manipulação da embarcação e a atualização da posição na tela.

2.3.2 Atributos

```
boolean stop;
int moving;
int posX, posY; //grid
int posTile;
float screenPosX, screenPosY;
float vel;
```

```
float offsetX, offsetY;
ArrayList<Integer> allowedTiles, movs;
boolean boat;
```

- `boolean stop` –> Indica se o jogador deve parar de se mover.
- `int moving` –> Indica a direção do movimento atual (cima, baixo, esquerda, direita).
- `int posX, posY` –> Coordenadas atuais do jogador na grade do mapa.
- `int posTile` –> Representa o valor do tile atual onde o jogador está posicionado.
- `float screenPosX, screenPosY` –> Posições do jogador na tela, baseadas em sua posição no mapa.
- `float vel` –> Velocidade do jogador que dependendo do solo muda.
- `float offsetX, offsetY` –> Deslocamento do jogador para suavizar o movimento de uma tile para outra.
- `ArrayList<Integer> allowedTiles` –> Lista dos tipos de terrenos nos quais o jogador pode se mover.
- `ArrayList<Integer> movs` –> Movimentação do player.
- `boolean boat` –> Indica se o jogador está em um barco.

2.3.3 Métodos

- `void update()` –> Atualiza a posição do jogador na tela e verifica se ele pode se mover, ajustando sua velocidade conforme o terreno em que ele está. Também chama o método de movimento caso o jogador esteja em deslocamento.
- `void setPlayer()` –> Define uma posição inicial aleatória para o jogador sem que ele apareça em uma célula não permitida ou com muitos obstáculos.
- `void display()` –> Exibe o jogador na tela na sua posição atual como um círculo vermelho, ajustando-se ao deslocamento atual (offset) para facilitar a transição entre tiles.
- `void move(ArrayList<Integer> movs)` –> Recebe uma lista de movimentos e atribui ao jogador, fazendo o se mover.
- `void checkEdges()` –> Verifica se o jogador se aproximou das bordas da tela. Caso isso ocorra, ajusta a visualização do mapa para manter o jogador dentro de uma área central da tela.

- `void dropBoat()` –> Faz o jogador "dropar"o barco no chão.
- `void move(int movs)` –> Inicia o movimento do jogador em uma direção específica, se ele não estiver já em movimento. O movimento é adicionado à lista de movimentos a serem realizados.
- `void movePlayer()` –> Executa o movimento do jogador de acordo com o valor atual de `moving`. O deslocamento é feito gradualmente, usando os valores de `offsetX` e `offsetY` para suavizar a animação. Quando o jogador atinge o próximo tile, o movimento é concluído.
- `void stop()` –> Para o movimento do jogador, ajustando os deslocamentos para os valores exatos dos limites de tiles. Limpa a lista de movimentos pendentes.

2.3.4 Principais Métodos (Explicação Detalhada)

2.3.4.1 update

```
void update(){
  this.display();

  this.screenPosX = map.screenPosX(posX);
  this.screenPosY = map.screenPosY(posY);

  posTile = map.getTileValue(posX, posY);
  if (movs !=null || moving != 0) movePlayer();
  if (map.renderized && allowedTiles.contains(posTile))
    if (!(route.onRoute > 0 && route.onRoute < 3)) this.checkEdges();
    switch (posTile){
      case 0:
        this.vel = 2;
        break;

      case 1:
        this.vel = 1;
        break;

      case 2:
        this.vel = 0.5;
        break;

      case 6:
        boat = true;
```

```
        allowedTiles.add(0);
        break;
    }
  }
```

- `void update()` –> O método `update()` Atualiza a situação do jogo, ele começa chamando o método display(), que é responsável por renderizar a representação visual do jogador na tela. Em seguida, atualiza as posições em tela do jogador com base em suas coordenadas de grid (posX e posY), usando os métodos map.screenPosX() e map.screenPosY(). O método então determina o tipo de tile em que o jogador está localizado chamando map.getTileValue(). Se há comandos de movimento pendentes (movs não é nulo) ou se o jogador está em movimento (moving != 0), o método movePlayer() é invocado para processar e aplicar esses comandos de movimento. Após isso, o método verifica se o mapa foi renderizado (map.renderized) e se o tipo de tile atual é permitido para o jogador (allowedTiles.contains(posTile)). Se essas condições forem verdadeiras e o jogador não estiver atualmente em um estado de rota específica (route.onRoute não está entre 1 e 2), o método checkEdges() é chamado para ajustar a posição do jogador com base nas bordas do mapa. Chegando no switch case ele verifica o tipo do solo e altera a velocidade, caso o jogador passe no barco as variaveis relacionadas a este são alteradas para que o jogo continue com sua funcionalidade.

2.3.4.2 movePlayer

```
void movePlayer(){
if (moving == 0 && !movs.isEmpty()){
  moving = movs.get(0);
  movs.remove(0);
}else if (moving == 0 && movs.isEmpty()) stop = true;
float off = (vel/60) * tileSize * 2 * velocidade;
switch (moving){
  case 1: //up
    offsetY -= off;
    break;
  case -1: //down
    offsetY += off;
    break;
  case 2: //left
    offsetX -= off;
    break;
  case -2: //right
```

```
        offsetX += off;
        break;
    }
```

- `void movePlayer()` –> gerencia o movimento do jogador, determinando a direção e a quantidade de deslocamento com base nas entradas atuais e na velocidade do jogador. Inicialmente, ele verifica se o jogador não está em movimento (moving == 0) e se há comandos de movimento pendentes (!movs.isEmpty()). Se essas condições forem verdadeiras, o próximo comando de movimento é retirado da lista movs e atribuído à variável moving. Caso contrário, se não há mais comandos de movimento e o jogador não está se movendo, a flag stop é definida como true, indicando que o jogador deve parar. A seguir, o método calcula o deslocamento do jogador em pixels com base na velocidade (vel), no tamanho do tile (tileSize), e na velocidade global (velocidade). Esse deslocamento é então aplicado aos offsets offsetX e offsetY, que determinam a mudança na posição do jogador. Dependendo da direção do movimento (moving), o método ajusta offsetX e offsetY para refletir o movimento para cima (1), para baixo (-1), para a esquerda (2), ou para a direita (-2). Esses ajustes são essenciais para atualizar a posição do jogador na tela, levando em consideração a direção do movimento e a velocidade calculada.

2.4 Game

2.4.1 Objetivo

A classe `Game` é responsável por gerenciar a lógica e o fluxo do jogo. Ela controla o progresso das partidas, incluindo o movimento dos jogadores e o funcionamento das rotas, seja para o jogador ou para a máquina. Essa classe é a base para o controle dos estágios do jogo e para a execução dos algoritmos de rota (Waze) e da competição entre o jogador e a máquina.

2.4.2 Atributos

- `int stage`: Define o estágio atual do jogo. Ao decorrer da gameplay ele é alterado.
- `boolean done`: Indica se o jogo foi finalizado.
- `boolean reading`: Indica se o jogo está no processo de leitura de uma rota.
- `Integer[] from`: Armazena as coordenadas iniciais da rota.
- `Integer[] to`: Armazena as coordenadas finais da rota.
- `ArrayList<Integer[]> way`: Lista que guarda os pontos do caminho selecionado pelo jogador durante a criação da rota.

- `Route playerRoute`: É a rota do jogador, tendo as coordenadas e as informações necessárias para calcular e exibir o percurso.
- `Route machineRoute`: Representa o caminho da máquina, calculada para competir com o do jogador.

Explicação dos Atributos

- `stage`: Fundamental para definir o estado atual do jogo e determinar a lógica que será executada em cada momento. Ex.: Se o jogador está traçando sua rota, se a máquina está simulando ou se o jogo já foi encerrado.
- `done`: Controla o término do jogo. É importante para determinar se o sistema deve parar ou continuar com novas ações.
- `reading`: Controla se o jogador está no processo de definir uma rota, permitindo a interação com o mapa.
- `from` e `to`: Cruciais para definir o ponto de partida e o destino final da rota traçada pelo jogador, usados na construção da rota.
- `way`: Lista de pontos (coordenadas) que formam a rota traçada pelo jogador. Auxilia no desenho da rota e no cálculo da rota final.
- `playerRoute` e `machineRoute`: Guardam as rotas do jogador e da máquina, respectivamente, e são usadas para desenhar, calcular e comparar tempos.

2.4.3 Métodos

- `Game()` –> Construtor da classe.
- `void display()` –> Desenha na tela a rota do jogador ou da máquina, dependendo do estágio do jogo, além da animação.
- `void start()` –> Inicia o processo de leitura de uma rota pelo jogador e define o estágio inicial do jogo.
- `void addToRoute(Integer x, Integer y)` –> Adiciona um ponto à rota do jogador, verificando se o movimento é válido.
- `void done()` –> Finaliza o processo de criação da rota, salvando os pontos iniciais e finais e chamando a conversão da rota do jogador.
- `void getAlg(char c)` –> Copia a rota do jogador para a rota da máquina, aplicando o algoritmo de busca (Waze) correspondente.

- `void convertInRoute()` –> Converte a rota traçada pelo jogador em um formato que pode ser usado para calcular o percurso e o custo da rota.
- `void animation()` –> Controla as animações e as transições entre os estágios do jogo, como o movimento da rota do jogador e da máquina.
- `void result(int c)` –> Exibe o resultado do jogo, indicando se o jogador venceu, perdeu ou empatou com a máquina.
- `void off()` –> Reinicia o jogo quando ele termina.

2.4.4 Principais Métodos (Explicação Detalhada)

2.4.4.1 display

```
  void display(){
  if (stage == 7) playerRoute.display(color(100, 100, 255, 230));
  else if (stage > 4) machineRoute.display();
  else
    for (Integer i[] : way){
      int x = map.screenPosX(i[0]), y = map.screenPosY(i[1]);
      fill(100, 100, 255, 230);
      ellipse(x, y, tileSize/3, tileSize/3);
      if (way.indexOf(i) == way.size() - 1) trigger.drawX(x, y);
    }
  animation();
}
```

2.4.4.2 `void display()`:

Como dito anteriormente este método é responsável pela parte gráfica do game. Dependendo do estágio, ele desenha a rota traçada pelo jogador ou a rota da máquina na tela. Além disso, é responsável por exibir o ponto final da rota do jogador e também inicia a animação das rotas, criando um feedback visual importante para o jogador.

2.4.4.3 addToRoute

```
  void addToRoute(Integer x, Integer y){
  Integer[] current = new Integer[]{x,y};
  Integer[] last = (way.isEmpty()) ? null : way.get(way.size()-1);
  if (reading)
    if (way.isEmpty())
```

```
        way.add(current);
      else if ( (abs(last[0] - current[0] ) == 0 && abs(last[1] - current[1] )
          == 1) || (abs(last[0] - current[0] ) == 1 && abs(last[1] - current[1]
          ) == 0) ){
        way.add(current);
      }
    }
```

2.4.4.4 `void addToRoute(Integer x, Integer y)`:

Este é a base para a funcionalidade de desenhar a rota. Ele recebe as coordenadas de um novo ponto e verifica se ele pode ser adicionado à rota com base nas regras de movimentação (distância de um bloco). Isso garante que o jogador crie uma rota válida para o jogo prosseguir. Sua importância reside no fato de ser o principal meio de entrada do jogador para o sistema de rotas.

2.4.5 Comentário sobre a classe

A classe `Game` é a peça central que une todas as funcionalidades do jogo, desde a criação da rota pelo jogador, até a simulação da rota pela máquina e a exibição dos resultados. A manipulação dos estágios do jogo, junto com a capacidade de desenhar as rotas e calcular os custos, torna essa classe essencial para o funcionamento adequado do jogo.

2.5 Main

2.5.1 Objetivo

A classe principal (Main) é responsável por iniciar o jogo, configurar as variáveis globais e controlar o fluxo de execução do programa. Nela, são instanciadas as classes principais que compõem o jogo e chamadas as funções responsáveis pela interação do jogador com o mapa e os demais elementos.

2.5.2 Instanciação de Objetos

- `Map map` –> Instancia o objeto `map` que gerencia os chunks, tiles e a visualização do mundo do jogo.
- `Trigger trigger` –> Instancia o objeto `trigger` que lida com eventos e interações do jogador no jogo.
- `Player player` –> Instancia o objeto `player`, representando o personagem principal controlado pelo jogador.

- `Route route` –> Instancia o objeto `route`, responsável pelo sistema de rotas que o jogador pode seguir.
- `Game game` –> Instancia o objeto `game`, que gerencia o estado geral do jogo e seus objetivos.

2.5.3 Fluxo de Execução

- `void setup()` –> A função `setup()` é chamada no início da execução do jogo e é responsável pela inicialização de variáveis, criação dos objetos principais (`map`, `player`, `route`, `trigger` e `game`) e pelo reset do mapa com a posição inicial do jogador.
- `void draw()` –> A função `draw()` é chamada em loop para renderizar os elementos do jogo na tela. Quando o jogo começa (`start == true`), ela desenha o mapa e outros elementos, como a rota e os triggers. Também chama a função de atualização do jogador.
- `void mouseDragged()` –> Detecta o movimento do mouse quando o botão é mantido pressionado. Essa função permite arrastar o mapa, ajustando sua posição conforme o movimento do mouse.
- `void mouseReleased()` –> Quando o botão do mouse é liberado, verifica se o jogo já começou e, se sim, chama as funções apropriadas para lidar com cliques nos botões ou para interagir com o mapa (selecionando tiles ou iniciando uma rota).
- `void keyPressed()` –> Detecta quando uma tecla é pressionada e executa diferentes ações dependendo da tecla. Isso inclui comandos para movimentar o jogador (`W, A, S, D`), confirmar pontos da rota, ativar cheats e centrar o mapa na posição do jogador.

2.5.4 Funções Importantes

- `setup()` –> Inicializa os principais elementos do jogo, como o mapa e o jogador, e define a posição inicial no mundo gerado aleatoriamente.
- `draw()` –> Controla o ciclo de renderização, atualizando a tela a cada frame e mantendo a interação contínua entre o jogador e o ambiente.
- `keyPressed()` –> Garante a responsividade do jogo às entradas do teclado, permitindo que o jogador mova-se pelo mapa, configure rotas e interaja com objetos no ambiente.

2.5.5 Comentário

A classe main atua como o coração do programa, coordenando as interações entre o jogador e o mapa, bem como as rotinas de renderização e movimentação. Ela utiliza objetos instanciados de outras classes para criar um mundo dinâmico e interativo, além de capturar

eventos de entrada, como cliques e teclas pressionadas, para modificar o estado do jogo em tempo real.

2.6 Map

A classe `Map`, fornecida inicialmente com funcionalidades básicas, foi modificada pelo nosso grupo para atender às necessidades do projeto, garantindo maior controle sobre a visualização do mapa, novos elementos interativos e melhorias na performance. Abaixo, discutimos as principais mudanças realizadas.

2.6.1 Objetivo

- A classe Map é responsável por gerenciar e exibir o ambiente de jogo em uma área de visualização dinâmica. Ela calcula quais partes do mapa, representadas por chunks, precisam ser exibidas com base no deslocamento atual e nas dimensões da tela. A classe cuida da atualização da posição do mapa através de métodos de arrasto e redefinição, além de permitir a conversão entre coordenadas de tela e coordenadas de grid. Além disso, a classe Map interage com a classe Chunk para garantir que o terreno e os objetos sejam renderizados corretamente, e gerencia aspectos adicionais, como a colocação de um barco no mapa.

2.6.2 1. Atributos Adicionados

Dois novos atributos foram adicionados à classe para gerenciar a renderização e a lógica de inserção de novos elementos no mapa:

- `renderized` (booleano): Utilizado para verificar se os chunks já foram renderizados, otimizando a execução ao evitar recriações desnecessárias.
- `setBoat` (booleano): Utilizado no controle de posicionamento de um barco aleatório no mapa, garantindo que ele seja colocado em uma área específica.

2.6.3 2. Alterações no Método `display()`

O método `display()` foi modificado para aumentar a eficiência e possibilitar novas interações:

- O cálculo das variáveis `endX` e `endY` foi ajustado para adicionar uma margem extra (`+10`) na renderização dos chunks, garantindo que áreas adjacentes à visualização atual sejam carregadas corretamente.
- O uso da variável `renderized` foi introduzido para otimizar a criação de novos chunks, evitando a recriação após a primeira renderização.

- Foi adicionada uma chamada ao novo método `setBoat()` no final da renderização, para posicionar um barco em uma tile adequada.

2.6.4 3. Novo Método `setBoat()`

Este método foi implementado para adicionar um barco aleatoriamente no mapa. Ele garante que o barco seja colocado em uma tile de água, definida pela densidade da região onde a tile está localizada.

Listing 1 – Método `setBoat()` para posicionar o barco em tiles de água.

```
public void setBoat(){
    this.setBoat = true;
    while (setBoat){
      int pX = player.posX, pY = player.posY;
      PVector randGrid = new PVector((int)random(pX - (width/tileSize/2),
                           pX + (width/tileSize/2)),
                           (int)random(pY - (height/tileSize/2),
                           pY + (height/tileSize/2.5)));

      Object[] ct = this.getChunkTile((int)randGrid.x, (int)randGrid.y);
      Chunk BoatChunk = chunks.get((String)ct[0]);
      PVector tile = (PVector) ct[1];

      // Caso densidade predominante seja agua, posiciona o barco
      if (BoatChunk.density != 0){
        BoatChunk.beforeBoat = BoatChunk.tiles[(int)tile.x][(int)tile.y];
        BoatChunk.tiles[(int)tile.x][(int)tile.y] = 6;
        this.setBoat = false;
      }
    }
}
```

Explicação:

- O método gera coordenadas aleatórias dentro de uma área definida em torno do jogador, usando a função `random()` para calcular essas coordenadas.
- As coordenadas da tile são obtidas por meio do método `getChunkTile()`, que retorna a chave do chunk e as coordenadas locais da tile.
- A densidade do chunk é verificada para garantir que a tile está em uma área de água (densidade não nula), onde o barco pode ser colocado. Após o barco ser posicionado, o valor da tile é alterado para 6.

Esse método foi adicionado para simular a presença de um barco de forma dinâmica no mapa, sempre garantindo que ele seja posicionado em uma área apropriada.

2.6.5 4. Alterações no Método `drag()`

O método `drag()` foi modificado para evitar que o mapa seja deslocado para fora dos limites, garantindo que o jogador não veja áreas inválidas do mapa.

Listing 2 – Método `drag()` modificado para evitar deslocamentos indevidos.

```
void drag(float _offsetX, float _offsetY) {
    if(this.gridPosX(0) < 0 || this.gridPosY(0) < 0){
        if(_offsetX <= 0 && _offsetY<=0){
            offsetX += _offsetX;
            offsetY += _offsetY;
        }
    } else {
        offsetX += _offsetX;
        offsetY += _offsetY;
    }
}
```

Explicação:

- O método agora verifica se a posição do grid é válida (`gridPosX(0)` e `gridPosY(0)`), impedindo o deslocamento se o valor for menor que 0. Isso garante que o jogador não possa "arrastar"o mapa para áreas inexistentes.
- Apenas quando os valores de `offsetX` e `offsetY` são válidos, o deslocamento é aplicado. Caso contrário, o método restringe o movimento, preservando a integridade da visualização do mapa.

Essa modificação foi feita para melhorar a jogabilidade, evitando que o jogador visualize áreas inválidas ou fora dos limites do mapa.

2.6.6 5. Novos Métodos `reset()`

Foram adicionadas duas versões do método `reset()` para facilitar a centralização da visualização do mapa:

- A primeira versão redefine o deslocamento para centralizar o mapa na tela.
- A segunda versão redefine o deslocamento em torno de uma tile específica no grid, permitindo centralizar o mapa com base nas coordenadas da tile.

Essa funcionalidade oferece mais controle ao jogador, permitindo que ele possa recentralizar rapidamente o mapa.

2.6.7 6. Método `getChunkTile()`

O método `getChunkTile()` foi adicionado para facilitar a obtenção das coordenadas locais de uma tile dentro de um chunk. Ele retorna tanto a chave do chunk quanto as coordenadas da tile, modularizando a lógica de busca e permitindo o reaproveitamento do código.

2.7 Chunk

2.7.1 Objetivo

- A classe Chunk gerencia e representa uma parte do terreno do jogo, dividida em uma matriz de tiles. Seu principal objetivo é gerar e exibir diferentes tipos de terreno e obstáculos, como água, grama e areia, utilizando funções de ruído para criar variação natural. Ela também cuida da visualização eficiente dos tiles visíveis na tela e permite modificar o terreno, por exemplo, ao interagir com objetos como barcos.

2.7.2 Alterações na Classe

Foram realizadas as seguintes alterações principais na classe `Chunk`:

- **Adição do Atributo `density`**: Adicionamos o atributo `density` para representar a densidade predominante do terreno em um chunk. Esse atributo pode assumir valores que indicam se o terreno é predominantemente água, grama ou areia.
- **Novo Atributo `beforeBoat`**: Introduzimos o atributo `beforeBoat` para armazenar o valor do tile antes de ser ocupado por um barco. Esse atributo permite restaurar corretamente o tipo de terreno original quando o barco é removido.

2.7.3 Métodos Relevantes

2.7.3.1 `generateChunk()`

O método `generateChunk()` foi revisado para melhorar a diversidade e a precisão dos terrenos gerados. As principais mudanças incluem:

Listing 3 – Método `generateChunk()` Modificado

```
void generateChunk() {
    int[] count = new int[3];
    for (int x = 0; x < chunkSize / tileSize; x++) {
```

```
        for (int y = 0; y < chunkSize / tileSize; y++) {
            float noise = noise((chunkX * chunkSize + x * tileSize) * 0.002,
                                (chunkY * chunkSize + y * tileSize) * 0.002);
            if (noise < 0.4) {
                count[0]++;
                tiles[x][y] = 0; // gua
            } else if (noise < 0.6) {
                count[1]++;
                tiles[x][y] = 1; // grama
            } else {
                count[2]++;
                tiles[x][y] = 2; // areia
            }

            if (random(1) < 0.005) {
                if (tiles[x][y] == 0) tiles[x][y] = 3; // coral
                else if (tiles[x][y] == 1) tiles[x][y] = 4; // pedra
                else if (tiles[x][y] == 2) tiles[x][y] = 5; // cacto
            }
        }
    }

    if (count[0] > count[1] && count[0] > count[2]) {
        this.density = 0; // predominncia de gua
    } else if (count[1] > count[2]) {
        this.density = 1; // predominncia de grama
    } else {
        this.density = 2; // predominncia de areia
    }
}
```

Explicação: O método `generateChunk()` é responsável por preencher a matriz `tiles` com valores que representam diferentes tipos de terreno e obstáculos. As alterações incluem o uso de um fator de escala mais fino na função de ruído (0.002) para criar uma variação mais detalhada no terreno. Além disso, o método agora conta a quantidade de cada tipo de terreno para definir o valor do atributo `density`, que reflete o tipo predominante de terreno no chunk.

2.7.3.2 `display()`

O método `display()` também foi atualizado para melhorar a visualização dos terrenos e obstáculos no jogo.

Listing 4 – Método `display()` Modificado

```
void display(float offsetX, float offsetY) {
    noStroke();
    for (int x = 0; x < chunkSize / tileSize; x++) {
        for (int y = 0; y < chunkSize / tileSize; y++) {
            float screenX = chunkX * chunkSize + x * tileSize + offsetX;
            float screenY = chunkY * chunkSize + y * tileSize + offsetY;

            if (screenX + tileSize < 0 || screenX > width || screenY + tileSize
                < 0 || screenY > height) {
                continue;
            }

            switch(tiles[x][y]) {
                case 0: fill(#42a5f5); break; // gua
                case 1: fill(#99ff88); break; // grama
                case 2: fill(#ffd780); break; // areia
                case 3: fill(#BA14C6); break; // coral
                case 4: fill(#a7b1c1); break; // pedra
                case 5: fill(#55cc44); break; // cacto
                case 6: // barco
                    if (!player.boat) {
                        fill(151, 51, 0);
                    } else {
                        tiles[x][y] = this.beforeBoat;
                    }
                    break;
                case 9: fill(0); break; // preto
            }
            noStroke();
            rect(screenX, screenY, tileSize, tileSize);
        }
    }
}
```

Explicação: O método `display()` é responsável por desenhar os tiles do chunk na tela. As alterações incluem a adição de uma verificação para exibir a cor do barco, caso ele esteja presente. O método agora também garante que os tiles fora da tela sejam ignorados, otimizando a performance ao evitar a renderização desnecessária.

3 Resultados

Figura 1 – Menu Inicial

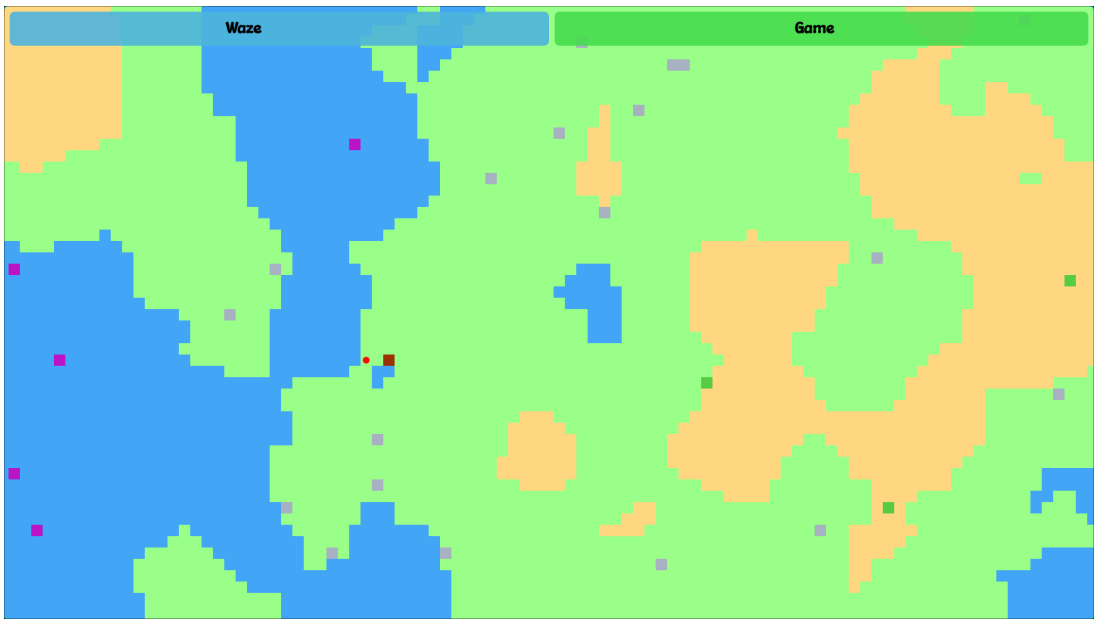

Figura 2 – Player impossibilitado de passar na água por não estar com barco (posicionado a sua direita)

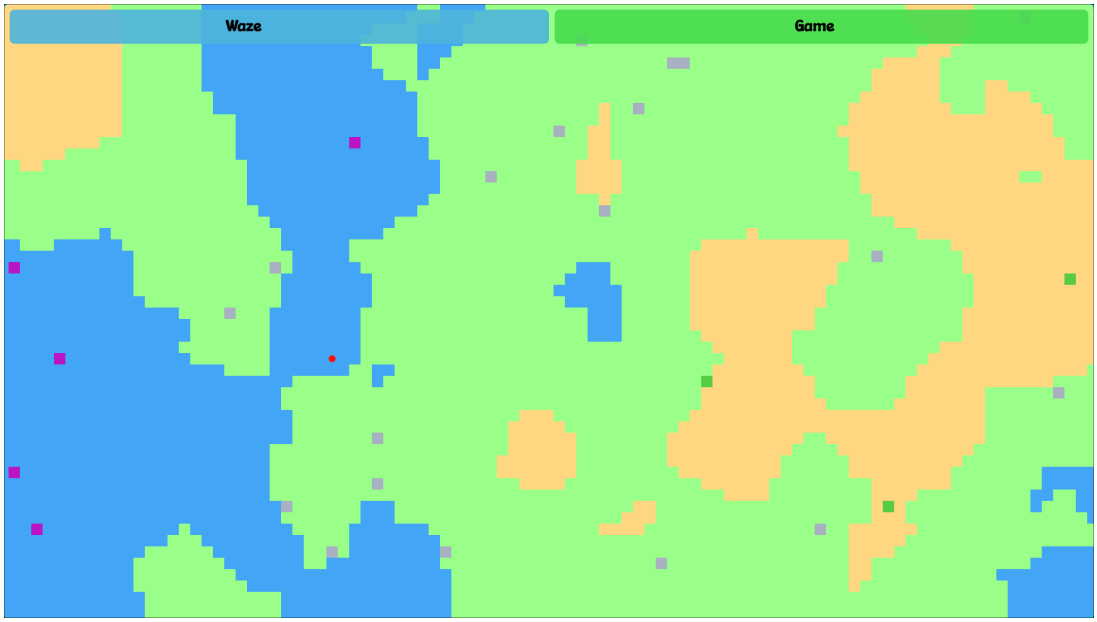

Figura 3 – Player com barco na água

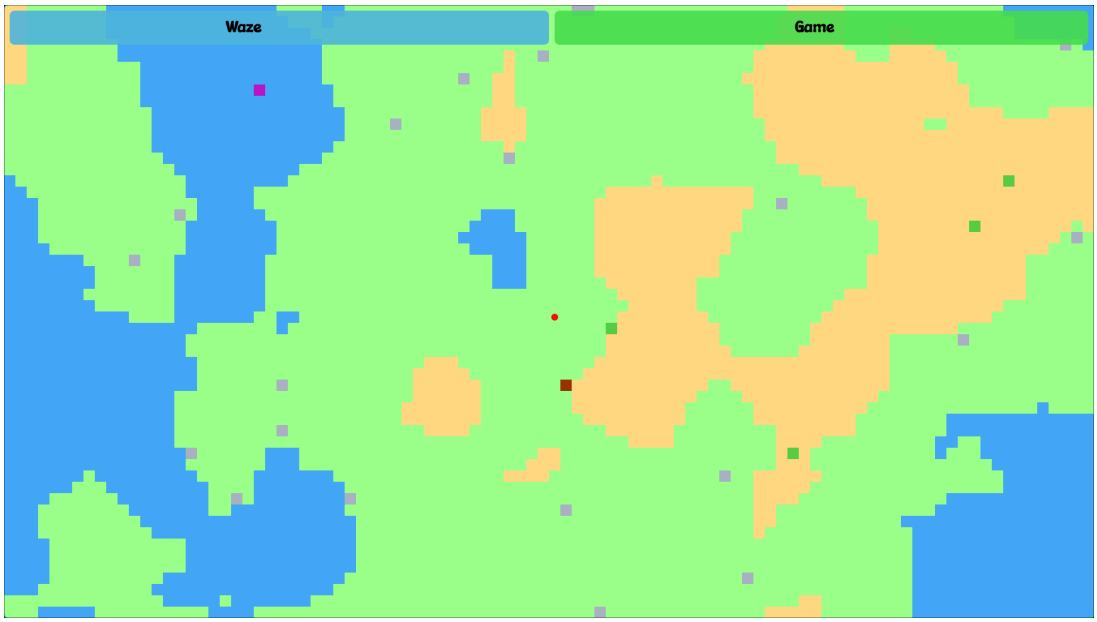

Figura 4 – Menu de escolha entre os modos de jogo

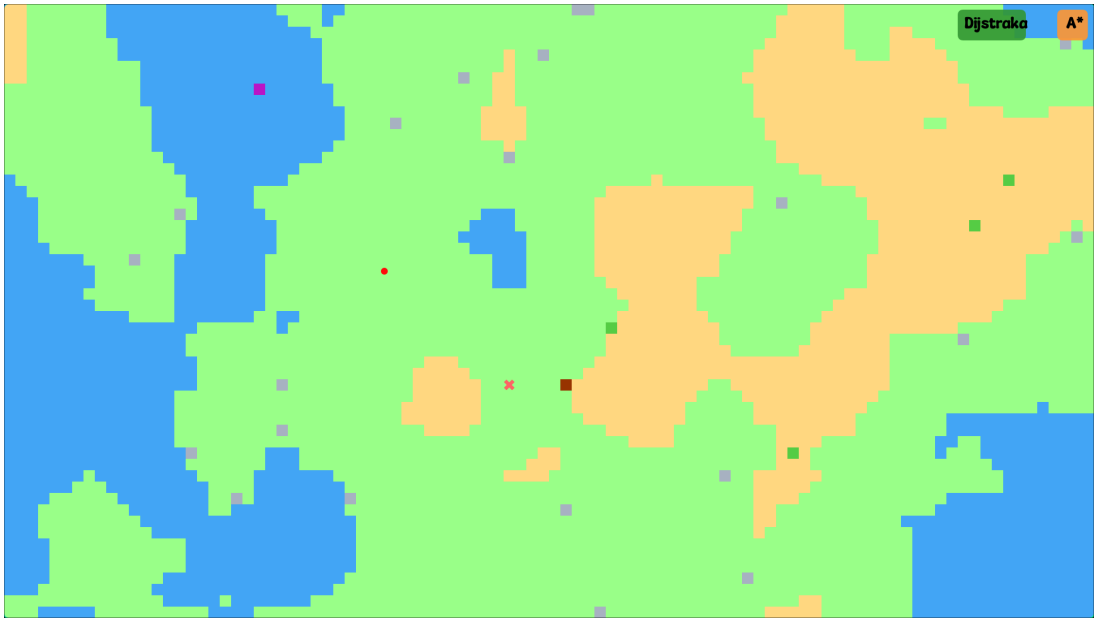

Figura 5 – Escolha de algoritmo no Waze

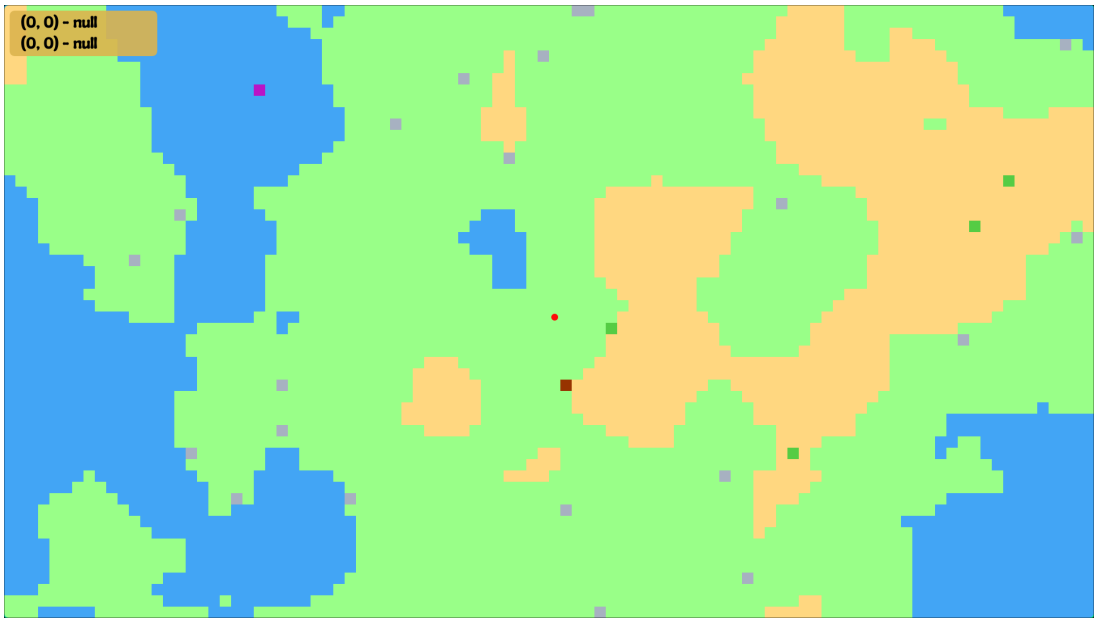

Figura 6 – Waze início

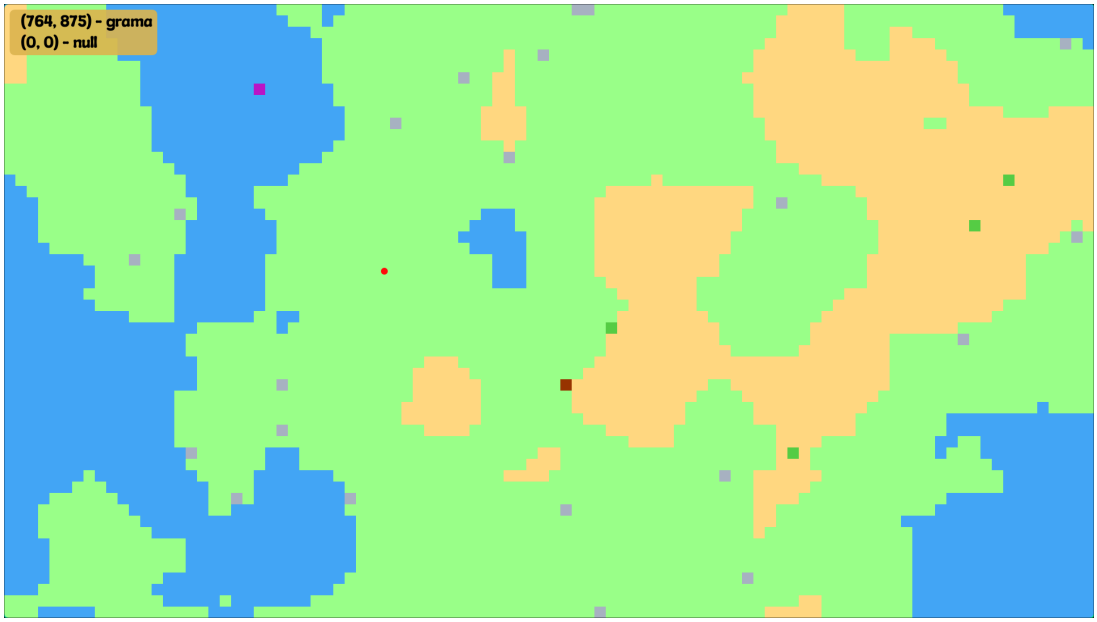

Figura 7 – Waze com origem selecionada

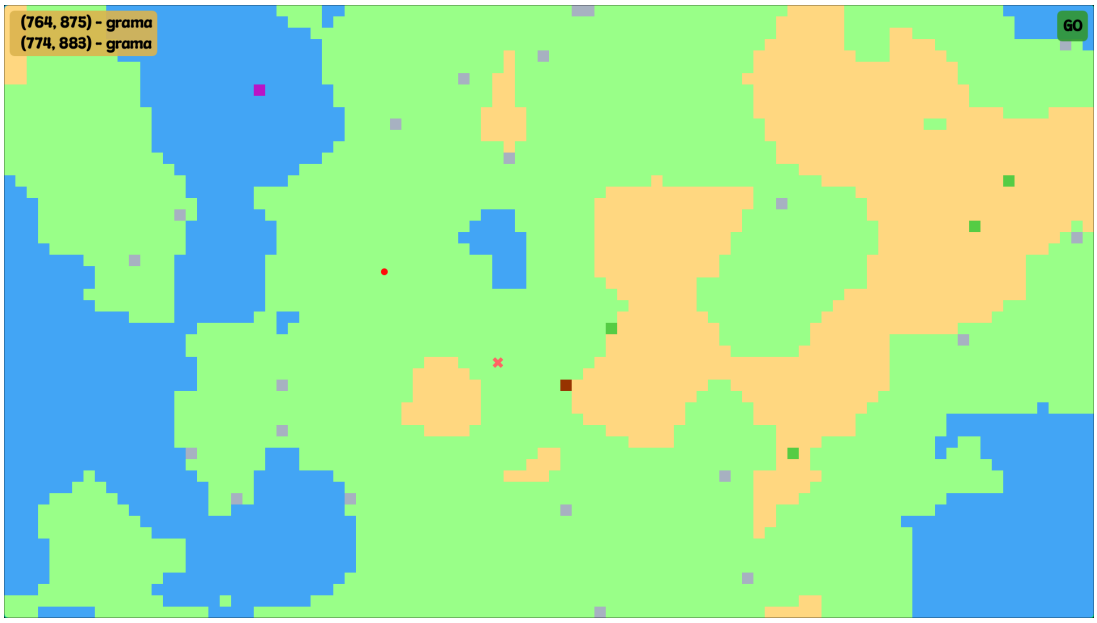

Figura 8 – Waze com destino selecionado

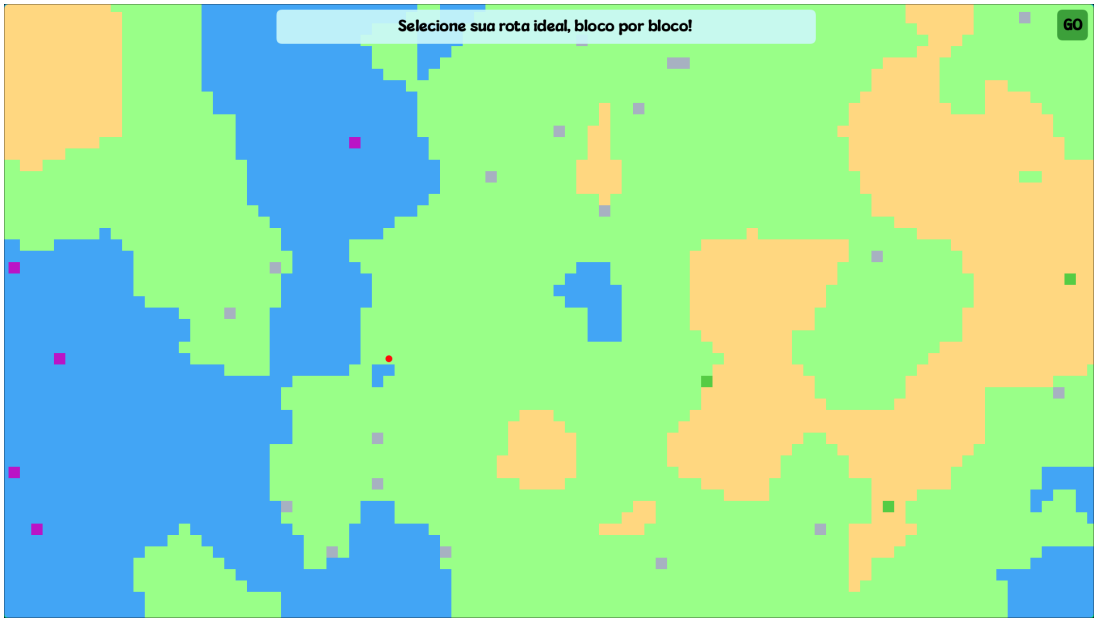

Figura 9 – Pré-game

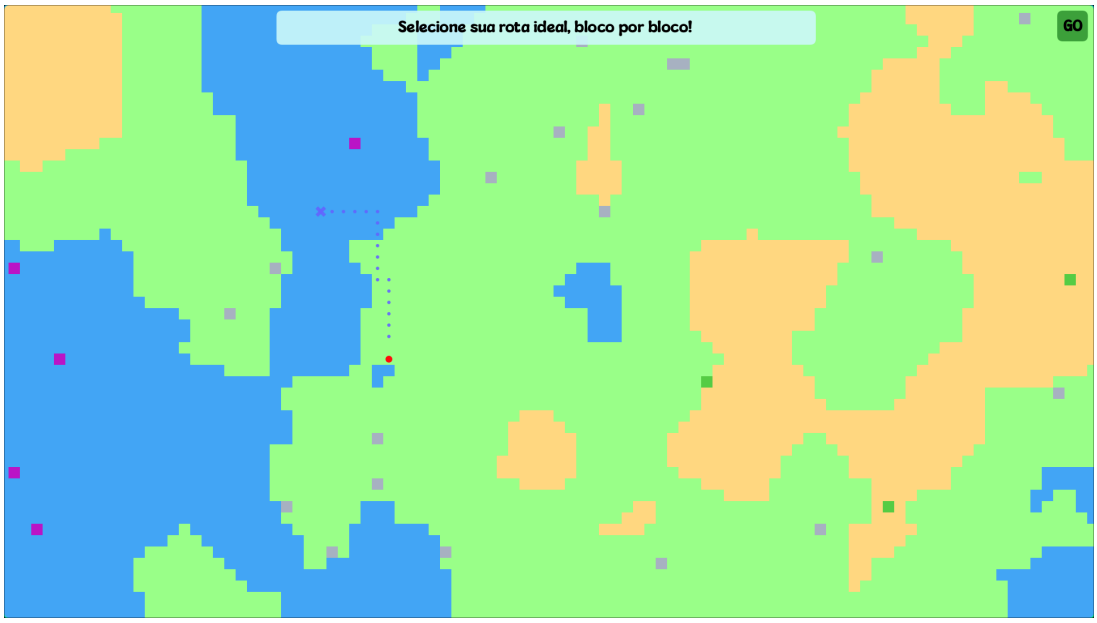

Figura 10 – Game com rota selecionada

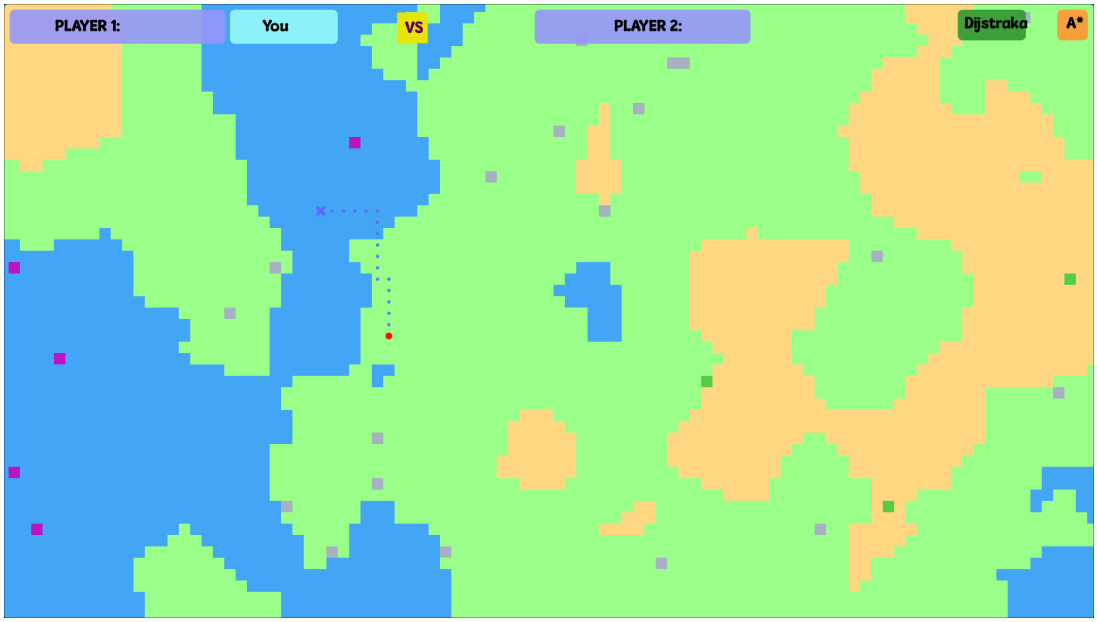

Figura 11 – Tela de início Game

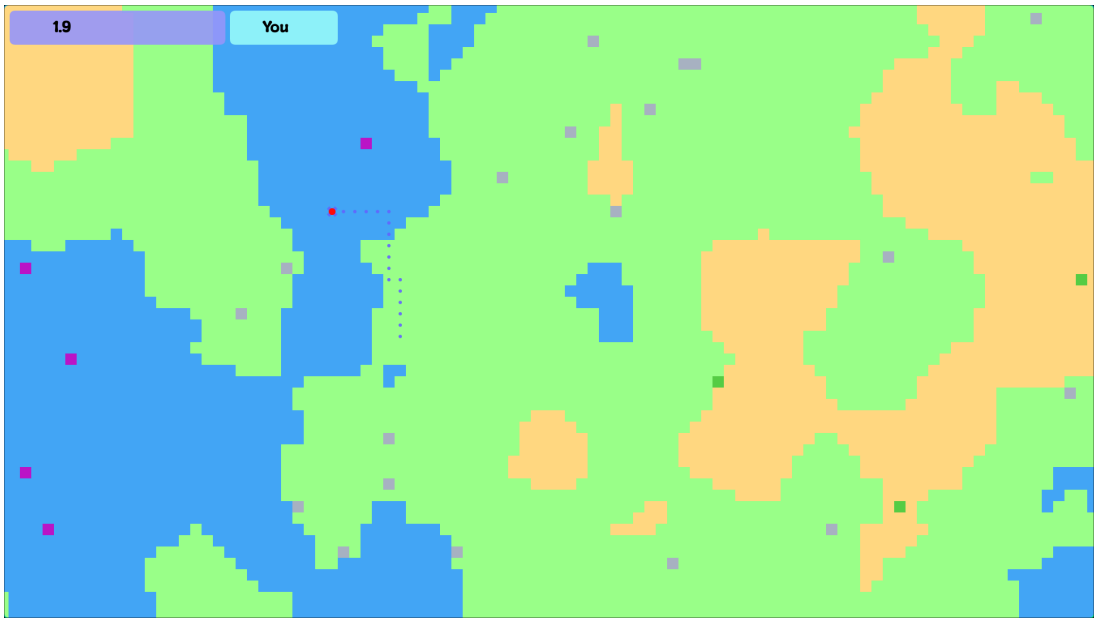

Figura 12 – Game: Vez do Player

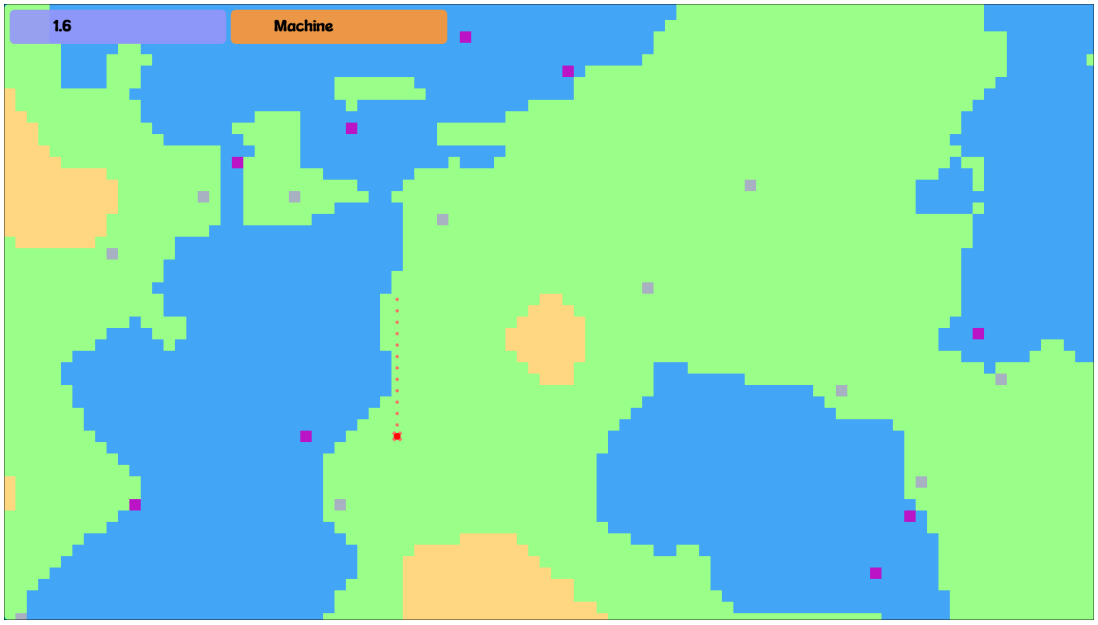

Figura 13 – Game: Vez do algortimo escolhido

Figura 14 – Game: Derrota

Figura 15 – Game: Empate

Figura 16 – Game: Vitória

Vale a pena observar que a vitória só ocorre quando é selecionada um rota em que a origem e o destino estão na mesma linha, mas é mais veloz se deslocar usando os blocos que não estão nessa linha. Nesse caso, só é fornecido o retângulo do vértice origem ao destino, não sendo vistos os blocos mais vantajosos fora dele.

4 Conclusão

O presente relatório apresentou a implementação de um jogo que simula um ambiente de navegação em um mapa com diferentes tipos de terreno e obstáculos. O jogo foi desenvolvido utilizando a linguagem de programação Java e a biblioteca Processing.

A implementação do jogo incluiu a criação de uma classe Map que gerencia o mapa e os chunks, uma classe Chunk que representa uma parte do terreno, uma classe Player que controla o movimento do jogador, uma classe Route que calcula o caminho mais curto entre dois pontos e uma classe Game que gerencia o fluxo do jogo.

O jogo foi testado e apresentou resultados satisfatórios, com a capacidade de gerar mapas aleatórios, calcular caminhos mais curtos e simular o movimento do jogador.

4.1 Principais contribuições

Implementação de um jogo que simula um ambiente de navegação em um mapa com diferentes tipos de terreno e obstáculos. Criação de uma classe Map que gerencia o mapa e os chunks. Criação de uma classe Chunk que representa uma parte do terreno. Criação de uma classe Player que controla o movimento do jogador. Criação de uma classe Route que calcula o caminho mais curto entre dois pontos. Criação de uma classe Game que gerencia o fluxo do jogo.

4.2 Limites e sugestões para possíveis melhorias

A implementação do jogo pode ser melhorada com a adição de mais recursos, como a capacidade de salvar e carregar jogos. A classe Map pode ser melhorada com a adição de mais tipos de terreno e obstáculos. A classe Chunk pode ser melhorada com a adição de mais detalhes, como a capacidade de gerar terrenos mais complexos. A classe Player pode ser melhorada com a adição de mais recursos, como a capacidade de saltar ou usar itens. A classe Route pode ser melhorada com a melhora da feature que faz o algoritmo só testar as possibilidade dentro do retângulo criado do vértice origem ao destino, fazendo ele testar pontos um poucos mais amplos para possibilitar que o menor caminho seja sempre alcançado.

4.3 Conclusão final

O presente relatório apresentou a implementação de um jogo que simula um ambiente de navegação em um mapa com diferentes tipos de terreno e obstáculos, mais o uso de algoritmos de menor caminho, fazendo os presentes autores se aprofundarem na implementação prática, mais ligada ao mundo real, desses algoritmos. A implementação do jogo foi satisfatória e apresentou resultados positivos. No entanto, há espaço para melhorias e sugestões para futuras melhorias foram apresentadas.