src/composables/useMapRouting.js

```
// src/composables/useMapRouting.js
import { ref, watch, computed } from 'vue';
// Não importa mais getWaypoints daqui

export function useMapRouting(userPosition, selectedLocal, currentFloor, mapScale, mapDimensions, waypointsDataRef) {
  const routeSegments = ref([]); // Array de { x, y, length, angle }
  const hasRoute = ref(false);
  const debugWaypoints = ref([]); // Para visualizar waypoints no mapa
  const routingError = ref(null);

  // Mapa de waypoints (ID -> waypoint object) - será atualizado pelo watcher
  const waypointMap = ref({});
  const currentFloorWaypoints = ref([]); // Waypoints apenas do andar atual

  // Observa o Ref de waypoints passado como argumento e atualiza o mapa interno
  watch(waypointsDataRef, (newWaypointsData) => {
    console.log("Waypoints recebidos/atualizados no useMapRouting:", newWaypointsData);
    const newMap = {};
    if (Array.isArray(newWaypointsData)) {
      newWaypointsData.forEach(wp => { newMap[wp.id] = wp; });
    }
    waypointMap.value = newMap;
    // Dispara um recálculo da rota se os waypoints mudarem e já houver usuário/destino
    if (userPosition.value && selectedLocal.value) {
        calculateRoute();
    }
  }, { deep: true, immediate: true }); // immediate: true para processar os waypoints iniciais

   // Filtra waypoints do andar atual reativamente
  watch([currentFloor, waypointsDataRef], () => {
     const floorId = currentFloor.value;
     const allWaypoints = waypointsDataRef.value || [];
     currentFloorWaypoints.value = allWaypoints.filter(wp => wp.andar === floorId);
     // Opcional: recalcular rota se o andar mudar (já coberto pelo watcher principal?)
```

```
  }, { deep: true, immediate: true });


  // --- Funções Auxiliares Internas ---

  /** Encontra o waypoint mais próximo de um ponto (x, y) *no andar atual* */
  const findNearestWaypoint = (x, y, andar) => {
    let nearest = null;
    let minDistanceSq = Infinity;
    const waypointsToSearch = currentFloorWaypoints.value; // Usa a lista pré-filtrada

    console.log(`Buscando waypoint mais próximo para (${x}, ${y}) no andar ${andar} entre ${waypointsToSearch.length} waypoints.`);


    for (const waypoint of waypointsToSearch) {
      // Não precisa mais checar o andar aqui, já está filtrado
      const dx = waypoint.x - x;
      const dy = waypoint.y - y;
      const distanceSq = dx * dx + dy * dy;

      if (distanceSq < minDistanceSq) {
        minDistanceSq = distanceSq;
        nearest = waypoint;
      }
    }
     if (!nearest) {
        console.warn(`Nenhum waypoint encontrado no andar ${andar}`);
     }
    return nearest;
  };

  /** Algoritmo de Dijkstra para encontrar o caminho mais curto entre waypoints *no mesmo andar* */
  // TODO: Adaptar para multi-andar se necessário usando 'conectaAndar'
  const findShortestPath = (startId, endId) => {
    const distances = {};
    const previous = {};
```

```
    const unvisited = new Set();
    const currentMap = waypointMap.value; // Usa o mapa reativo

    // Inicialização apenas para o andar atual
    currentFloorWaypoints.value.forEach(wp => {
        distances[wp.id] = wp.id === startId ? 0 : Infinity;
        previous[wp.id] = null;
        unvisited.add(wp.id);
    });

    if (!currentMap[startId] || distances[startId] === undefined) {
        console.error("Waypoint inicial inválido ou não pertence ao andar atual:", startId);
        routingError.value = "Ponto de partida inválido para roteamento.";
        return null;
    }
     if (!currentMap[endId] || distances[endId] === undefined) {
        console.error("Waypoint final inválido ou não pertence ao andar atual:", endId);
        routingError.value = "Ponto de destino inválido para roteamento.";
        return null;
    }

    while (unvisited.size > 0) {
      let currentId = null;
      let smallestDistance = Infinity;
      for (const nodeId of unvisited) {
        if (distances[nodeId] < smallestDistance) {
          smallestDistance = distances[nodeId];
          currentId = nodeId;
        }
      }

      if (currentId === null || currentId === endId || smallestDistance === Infinity) break;

      unvisited.delete(currentId);
      const currentWaypoint = currentMap[currentId];

      if (!currentWaypoint || !currentWaypoint.connections) {
```

```
          console.warn(`Waypoint ${currentId} sem conexões ou inválido.`);
          continue;
        }

        // Iterar sobre vizinhos (conexões)
        for (const neighborId of currentWaypoint.connections) {
          const neighbor = currentMap[neighborId];

          // Considera apenas vizinhos NO MESMO ANDAR (limitação atual)
          if (!neighbor || neighbor.andar !== currentFloor.value) continue;

          // Calcula distância euclidiana
          const dx = currentWaypoint.x - neighbor.x;
          const dy = currentWaypoint.y - neighbor.y;
          const distance = Math.sqrt(dx * dx + dy * dy);
          const totalDistance = distances[currentId] + distance;

          if (totalDistance < distances[neighborId]) {
            distances[neighborId] = totalDistance;
            previous[neighborId] = currentId;
          }
        }
      } // Fim do while

      // Reconstruir caminho
      const path = [];
      let current = endId;
      // Verifica se o destino foi alcançado
      if (previous[endId] !== undefined || startId === endId) {
          while (current !== null && current !== undefined && currentMap[current]) {
              path.unshift(currentMap[current]);
              if (current === startId) break; // Chegou ao início
              current = previous[current];
              if (path.length > currentFloorWaypoints.value.length) { // Safety break
                   console.error("Erro na reconstrução do caminho - loop infinito?");
                   return null;
              }
```

```
        }
    }

     // Verifica se o caminho reconstruído começa com o startId correto
    if (path.length > 0 && path[0]?.id === startId) {
        return path;
    } else if (startId === endId && currentMap[startId]) {
         return [currentMap[startId]]; // Caminho de um ponto só
    } else {
        console.warn("Não foi possível reconstruir um caminho válido de", startId, "para", endId);
        return null; // Caminho não encontrado ou inválido
    }
};

/** Cria um segmento de rota (linha) entre dois pontos */
const createRouteSegment = (x1, y1, x2, y2) => {
    const baseWidth = mapDimensions.value?.width || 1;
    const baseHeight = mapDimensions.value?.height || 1;
    if (baseWidth <= 1 || baseHeight <= 1) {
        console.warn("Dimensões do mapa inválidas para criar segmento:", mapDimensions.value);
        return; // Não cria segmento se dimensões não forem válidas
    }

    const dxPercent = x2 - x1;
    const dyPercent = y2 - y1;
    const pixelDx = (dxPercent / 100) * baseWidth;
    const pixelDy = (dyPercent / 100) * baseHeight;
    const length = Math.sqrt(pixelDx * pixelDx + pixelDy * pixelDy);
    const angle = Math.atan2(pixelDy, pixelDx) * (180 / Math.PI);

    // Evita segmentos de comprimento zero ou NaN
    if (isNaN(length) || length < 0.1) {
        return;
    }


    routeSegments.value.push({ x: x1, y: y1, length: length, angle: angle });
```

```
182   };
183
184   /** Cria uma rota como uma linha reta simples (fallback) */
185   const createSimpleRouteLine = (startPos, endPos) => {
186     routeSegments.value = []; // Limpa segmentos anteriores
187     if (!startPos || !endPos) {
188       hasRoute.value = false;
189       return;
190     }
191     createRouteSegment(startPos.x, startPos.y, endPos.x, endPos.y);
192     hasRoute.value = routeSegments.value.length > 0;
193     routingError.value = "Não foi possível calcular a rota detalhada. Mostrando linha reta.";
194   };
195
196
197   // --- Função Principal de Criação de Rota ---
198   const calculateRoute = () => {
199     routeSegments.value = []; // Limpa rota anterior
200     hasRoute.value = false;
201     routingError.value = null;
202     debugWaypoints.value = []; // Limpa waypoints de debug
203
204     const startPos = userPosition.value;
205     const endLocal = selectedLocal.value;
206     const floorId = currentFloor.value;
207     const waypointsDisponiveis = currentFloorWaypoints.value;
208
209     // Condições para calcular a rota
210     if (!startPos || !endLocal) {
211       console.log("Cálculo de rota abortado: Posição inicial ou local final ausente.");
212       return; // Sai se não houver usuário ou destino
213     }
214     if (endLocal.andar !== floorId) {
215         console.log(`Cálculo de rota abortado: Destino (${endLocal.andar}) não está no andar atual (${floorId}).`);
216         // Limpa rota existente se o usuário mudou de andar mas o destino ficou selecionado
217         return;
218     }
```

```
     if (waypointsDisponiveis.length === 0) {
         console.warn("Cálculo de rota abortado: Nenhum waypoint disponível para o andar", floorId);
         routingError.value = `Nenhum waypoint encontrado para o andar ${floorId}. Impossível rotear.`;
         return;
     }


    // 1. Encontrar waypoints mais próximos
    const startWaypoint = findNearestWaypoint(startPos.x, startPos.y, floorId);
    const endWaypoint = findNearestWaypoint(endLocal.x, endLocal.y, floorId);

    if (!startWaypoint || !endWaypoint) {
      console.warn("Não foi possível encontrar waypoints próximos para o início ou fim no andar", floorId);
      routingError.value = "Waypoints próximos não encontrados.";
      createSimpleRouteLine(startPos, endLocal); // Fallback para linha reta
      return;
    }

    console.log("Waypoint inicial mais próximo:", startWaypoint.id);
    console.log("Waypoint final mais próximo:", endWaypoint.id);
    debugWaypoints.value.push({ id: 'debug_start_wp', ...startWaypoint });
    debugWaypoints.value.push({ id: 'debug_end_wp', ...endWaypoint });

    // 2. Encontrar caminho mais curto entre waypoints
    let path = null;
    if (startWaypoint.id === endWaypoint.id) {
      path = [startWaypoint]; // Caminho contém apenas um waypoint
      console.log("Waypoints inicial e final são os mesmos.");
    } else {
      path = findShortestPath(startWaypoint.id, endWaypoint.id);
      console.log("Caminho encontrado:", path?.map(p => p.id));
    }

    if (!path || path.length === 0) {
      console.warn(`Não foi possível encontrar um caminho entre ${startWaypoint.id} e ${endWaypoint.id}. Usando linha reta.`);
      routingError.value = "Não foi possível calcular a rota entre os pontos.";
```

```
255       createSimpleRouteLine(startPos, endLocal); // Fallback
256       return;
257     }
258
259     // Adiciona os waypoints do caminho para depuração visual (exceto start/end já adicionados)
260     path.forEach(wp => {
261         if (wp.id !== startWaypoint.id && wp.id !== endWaypoint.id) {
262             debugWaypoints.value.push({ id: `debug_path_${wp.id}`, ...wp })
263         }
264     });
265
266
267     // 3. Criar segmentos da rota
268     // Segmento: Usuário -> Primeiro Waypoint do Caminho
269     createRouteSegment(startPos.x, startPos.y, path[0].x, path[0].y);
270
271     // Segmentos: Entre Waypoints do Caminho
272     for (let i = 0; i < path.length - 1; i++) {
273       createRouteSegment(path[i].x, path[i].y, path[i + 1].x, path[i + 1].y);
274     }
275
276     // Segmento: Último Waypoint do Caminho -> Destino Final (Local)
277     // Garante que path[path.length - 1] existe antes de acessar
278      if (path.length > 0) {
279         createRouteSegment(path[path.length - 1].x, path[path.length - 1].y, endLocal.x, endLocal.y);
280      } else {
281          // Se o path era apenas o startWaypoint, conecta direto ao endLocal
282          createRouteSegment(startWaypoint.x, startWaypoint.y, endLocal.x, endLocal.y);
283      }
284
285
286     hasRoute.value = routeSegments.value.length > 0;
287     if(!hasRoute.value && !routingError.value) { // Só define erro se não houve outro erro antes
288       routingError.value = "Falha ao gerar segmentos da rota.";
289     } else if (hasRoute.value) {
290         routingError.value = null; // Limpa erro se a rota foi gerada com sucesso
291     }
```

```
292       console.log("Segmentos da rota calculados:", routeSegments.value.length);
293     };
294
295     // --- Observador ---
296     // Recalcula a rota quando algo relevante muda
297     watch(
298       [userPosition, selectedLocal, currentFloor, mapDimensions, waypointsDataRef], // Adiciona waypointsDataRef
299       calculateRoute,
300       { deep: true, immediate: false } // Roda após a montagem inicial e dados carregados
301     );
302
303     // --- Retorno ---
304     return {
305       routeSegments,
306       hasRoute,
307       debugWaypoints,
308       routingError,
309       calculateRoute, // Permite chamar manualmente
310     };
311 }
```