```
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 || [];

     console.log('useMapRouting - Filtrando waypoints:');
     console.log('Andar atual:', floorId);
     console.log('Total de waypoints:', allWaypoints.length);
     console.log('Waypoints disponíveis:', allWaypoints);
```

```
37
38      currentFloorWaypoints.value = allWaypoints.filter(wp => {
39        const matches = wp.andar === floorId;
40        console.log(`Waypoint ${wp.id}: andar=${wp.andar}, matches=${matches}`);
41        return matches;
42      });
43
44      console.log('Waypoints filtrados para o andar atual:', currentFloorWaypoints.value);
45    }, { deep: true, immediate: true });
46
47
48    // --- Funções Auxiliares Internas ---
49
50    /** Encontra o waypoint mais próximo de um ponto (x, y) *no andar atual* */
51    const findNearestWaypoint = (x, y, andar) => {
52      let nearest = null;
53      let minDistanceSq = Infinity;
54      const waypointsToSearch = currentFloorWaypoints.value; // Usa a lista pré-filtrada
55
56      console.log(`Buscando waypoint mais próximo para (${x}, ${y}) no andar ${andar} entre ${waypointsToSearch.length}
    waypoints.`);
57
58
59      for (const waypoint of waypointsToSearch) {
60        // Não precisa mais checar o andar aqui, já está filtrado
61        const dx = waypoint.x - x;
62        const dy = waypoint.y - y;
63        const distanceSq = dx * dx + dy * dy;
64
65        if (distanceSq < minDistanceSq) {
66          minDistanceSq = distanceSq;
67          nearest = waypoint;
68        }
69      }
70       if (!nearest) {
71          console.warn(`Nenhum waypoint encontrado no andar ${andar}`);
72       }
```

```
    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) {
```

```
147         while (current !== null && current !== undefined && currentMap[current]) {
148             path.unshift(currentMap[current]);
149             if (current === startId) break; // Chegou ao início
150             current = previous[current];
151             if (path.length > currentFloorWaypoints.value.length) { // Safety break
152                  console.error("Erro na reconstrução do caminho - loop infinito?");
153                  return null;
154             }
155         }
156     }
157 
158      // Verifica se o caminho reconstruído começa com o startId correto
159     if (path.length > 0 && path[0]?.id === startId) {
160         return path;
161     } else if (startId === endId && currentMap[startId]) {
162          return [currentMap[startId]]; // Caminho de um ponto só
163     } else {
164         console.warn("Não foi possível reconstruir um caminho válido de", startId, "para", endId);
165         return null; // Caminho não encontrado ou inválido
166     }
167 };
168 
169 /** Cria um segmento de rota (linha) entre dois pontos */
170 const createRouteSegment = (x1, y1, x2, y2) => {
171     const baseWidth = mapDimensions.value?.width || 1;
172     const baseHeight = mapDimensions.value?.height || 1;
173     if (baseWidth <= 1 || baseHeight <= 1) {
174         console.warn("Dimensões do mapa inválidas para criar segmento:", mapDimensions.value);
175         return; // Não cria segmento se dimensões não forem válidas
176     }
177 
178     const dxPercent = x2 - x1;
179     const dyPercent = y2 - y1;
180     const pixelDx = (dxPercent / 100) * baseWidth;
181     const pixelDy = (dyPercent / 100) * baseHeight;
182     const length = Math.sqrt(pixelDx * pixelDx + pixelDy * pixelDy);
183     const angle = Math.atan2(pixelDy, pixelDx) * (180 / Math.PI);
```

```
184
185     // Evita segmentos de comprimento zero ou NaN
186     if (isNaN(length) || length < 0.1) {
187         return;
188     }
189
190
191     routeSegments.value.push({ x: x1, y: y1, length: length, angle: angle });
192   };
193
194   /** Cria uma rota como uma linha reta simples (fallback) */
195   const createSimpleRouteLine = (startPos, endPos) => {
196     routeSegments.value = []; // Limpa segmentos anteriores
197     if (!startPos || !endPos) {
198       hasRoute.value = false;
199       return;
200     }
201     createRouteSegment(startPos.x, startPos.y, endPos.x, endPos.y);
202     hasRoute.value = routeSegments.value.length > 0;
203     routingError.value = "Não foi possível calcular a rota detalhada. Mostrando linha reta.";
204   };
205
206
207   // --- Função Principal de Criação de Rota ---
208   const calculateRoute = () => {
209     routeSegments.value = []; // Limpa rota anterior
210     hasRoute.value = false;
211     routingError.value = null;
212     debugWaypoints.value = []; // Limpa waypoints de debug
213
214     const startPos = userPosition.value;
215     const endLocal = selectedLocal.value;
216     const floorId = currentFloor.value;
217     const waypointsDisponiveis = currentFloorWaypoints.value;
218
219     // Condições para calcular a rota
220     if (!startPos || !endLocal) {
```

```
      console.log("Cálculo de rota abortado: Posição inicial ou local final ausente.");
      return; // Sai se não houver usuário ou destino
    }
    if (endLocal.andar !== floorId) {
        console.log(`Cálculo de rota abortado: Destino (${endLocal.andar}) não está no andar atual (${floorId}).`);
        // Limpa rota existente se o usuário mudou de andar mas o destino ficou selecionado
        return;
    }
     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.";
      createSimpleRouteLine(startPos, endLocal); // Fallback
      return;
    }

    // Adiciona os waypoints do caminho para depuração visual (exceto start/end já adicionados)
    path.forEach(wp => {
        if (wp.id !== startWaypoint.id && wp.id !== endWaypoint.id) {
            debugWaypoints.value.push({ id: `debug_path_${wp.id}`, ...wp })
        }
    });


    // 3. Criar segmentos da rota
    // Segmento: Usuário -> Primeiro Waypoint do Caminho
    createRouteSegment(startPos.x, startPos.y, path[0].x, path[0].y);

    // Segmentos: Entre Waypoints do Caminho
    for (let i = 0; i < path.length - 1; i++) {
      createRouteSegment(path[i].x, path[i].y, path[i + 1].x, path[i + 1].y);
    }

    // Segmento: Último Waypoint do Caminho -> Destino Final (Local)
    // Garante que path[path.length - 1] existe antes de acessar
     if (path.length > 0) {
        createRouteSegment(path[path.length - 1].x, path[path.length - 1].y, endLocal.x, endLocal.y);
     } else {
         // Se o path era apenas o startWaypoint, conecta direto ao endLocal
         createRouteSegment(startWaypoint.x, startWaypoint.y, endLocal.x, endLocal.y);
     }
```

```
294
295
296     hasRoute.value = routeSegments.value.length > 0;
297     if(!hasRoute.value && !routingError.value) { // Só define erro se não houve outro erro antes
298       routingError.value = "Falha ao gerar segmentos da rota.";
299     } else if (hasRoute.value) {
300         routingError.value = null; // Limpa erro se a rota foi gerada com sucesso
301     }
302      console.log("Segmentos da rota calculados:", routeSegments.value.length);
303   };
304
305   // --- Observador ---
306   // Recalcula a rota quando algo relevante muda
307   watch(
308     [userPosition, selectedLocal, currentFloor, mapDimensions, waypointsDataRef], // Adiciona waypointsDataRef
309     calculateRoute,
```