GUILHERME DE LIMA OTTONI

PLANEJAMENTO DE TRAJETÓRIAS PARA ROBÔS MÓVEIS

Projeto de Graduação apresentado ao Curso de Engenharia de Computação, Fundação Universidade Federal do Rio Grande.

Orientador: Prof. Dr. Walter Fetter Lages

Rio Grande

Janeiro de 2000

Guilherme de Lima Ottoni

Planejamento de Trajetórias para Robôs Móveis

Projeto de Graduação apresentado ao Curso de Engenharia de Computação, Fundação Universidade Federal do Rio Grande.

Orientador: Prof. Dr. Walter Fetter Lages

Rio Grande

Janeiro de 2000

SUMÁRIO

LISTA DE ILUSTRAÇÕES

RESUMO

Na área de robótica, um dos objetivos que têm sido buscados nos últimos anos é o de robôs autônomos, ou seja, robôs que possuam um certo grau de autonomia no desempenho das tarefas para as quais foi projetado. Um dos problemas relacionados a robôs móveis é o de planejamento de trajetórias, no qual deseja-se apenas informar de que configuração inicial até qual outra configuração espera-se que um determinado objeto seja movimentado, sendo que o robô deve ser capaz de escolher de que forma isto será feito. Um caso particular deste problema é quando se tem um robô móvel, e o objeto que deseja-se mover é o próprio robô. Neste trabalho é feita uma revisão dos principais métodos genéricos para tratar este problema. A seguir, é apresentado o projeto e a implementação de um sistema de planejamento de trajetórias em ambientes desconhecidos, o qual opera em tempo real. Ele é baseado no método da decomposição em células aproximada. O modelo implementado foi testado em simulação e a seguir no robô móvel Twil do Departamento de Física da FURG. Os resultados obtidos foram bastante satisfatórios, mostrando-se o método utilizado suficientemente eficiente.

ABSTRACT

In Robotics, one of the ultimate goals in the last years has been for autonomous robots, i.e. robots which have some autonomy in developing their work. One of the problems related to autonomous robots is path planning, in which we just want to inform the robot about the initial configuration of an object and the goal configuration we want it to reach, and let the robot choose how it is going to be done. A particular case of this problem is when the robot is a mobile one, and the object we want it to move is himself. In this work, it's going to be studied the principal generic methods for this problem. Later it's presented the project and implementation of a real time path planning system, intended to work on unknown environments. It is based on the approximated cell decomposition method. The implemented model was tested on simulation, and later on Twil, a mobile robot developed on the FURG's Physics Department. The results were very satisfactory, and the method proved to be enough efficient.

CAPÍTULO I

INTRODUÇÃO

Um importante objetivo na área de robótica é a criação de robôs autônomos. Tais robôs devem aceitar descrições de alto nível das tarefas que eles devem fazer, sem a necessidade de maiores intervenções humanas. As descrições de entrada especificam o que o usuário deseja que seja feito, e não como proceder para fazê-lo. Estes robôs são equipados com atuadores e sensores sob controle de um sistema de computação.

Progressos relacionados a robôs autônomos são de grande interesse para uma grande variedade de aplicações, como por exemplo automação industrial, construção, exploração espacial, trabalhos submarinos, lavagem externa de aviões, assistência a deficientes e cirurgias [1][2]. Além disto, são de grande interesse técnico, principalmente para a área da Ciência da Computação, uma vez que surgem vários problemas relacionados a esta. Por exemplo, freqüentemente encontram-se problemas de busca em grafos, otimização, geometria computacional, inteligência artificial, entre outras sub-áreas da computação.

O desenvolvimento da tecnologia necessária para robôs autônomos engloba algumas ramificações como raciocínio automatizado, percepção e controle, surgindo vários problemas importantes. Entre eles, encontra-se o planejamento de trajetórias. Este problema pode ser formulado, primeiramente, conforme segue: "Como pode um robô decidir que movimentos ele deve fazer de forma a alcançar uma configuração desejada de determinados objetos?" Embora os métodos estudados aqui sejam válidos na maioria dos casos, focaremos sempre em um caso particular deste problema. Trata-se do caso de um robô móvel onde o objeto que se deseja levar a uma determinada configuração é o próprio robô. Por configuração deve-se aqui entender posição e direção do robô.

À primeira vista, o problema do planejamento de trajetórias para robôs pode parecer algo relativamente simples. Assim como em problemas relacionados à percepção, isto se deve à inteligência elementar que as pessoas possuem e que usam inconscientemente na sua interação com o ambiente que as cerca. Tarefas aparentemente simples, como por exemplo montagem de um equipamento, servir uma xícara de café e movimentar-se dentro de um prédio, tornam-se extremamente difíceis de serem reproduzidas em robôs controlados por computador. Será visto neste trabalho que geralmente são necessárias diversos conhecimentos matemáticos e algorítmicos não triviais para construir-se um planejador de trajetórias razoavelmente geral e confiável.

As pesquisas sobre planejamento de movimentos começaram nos anos 60, nos estágios iniciais do desenvolvimento de robôs controlados por computador. Contudo, maiores esforços nesta área foram feitos a partir dos anos 80. Nos últimos anos, conhecimentos práticos e teóricos têm surgido mais rapidamente devido ao trabalho conjunto de pesquisadores de áreas como Inteligência Artificial, Teoria da Computação, Matemática e Engenharia Mecânica. Além da produção de métodos efetivos de planejamento de movimentos, este trabalho conjunto contribuiu para o avanço no conhecimento sobre estruturas matemáticas de problemas, e também para o entendimento da complexidade computacional inerente a eles.

O problema de planejamento de trajetórias pode abranger diversos aspectos como o planejamento de movimentos entre obstáculos móveis, coordenação do movimento de vários robôs, raciocínio sobre incerteza para construção de estratégias confiáveis de movimentos baseados em sensores, e consideração de modelos físicos. Portanto, planejamento de movimentos pode englobar a consideração de restrições geométricas, além de restrições físicas e temporais.

Um planejador automático de trajetórias pode ser utilizado de várias formas. Ele pode fazer parte de um sistema *off-line* de programação de um robô, de forma a simplificar a tarefa de descrever trajetórias de robôs. Pode também fazer parte de um sistema interativo de movimentação de um robô. No presente trabalho, dá-se ênfase especial a este último caso. Neste, torna-se importante a interação do planejamento de movimentos com outros problemas relacionados.

O primeiro destes problemas relacionados é o do sensoriamento do ambiente ao redor do robô. Isto torna-se importante quando se trata de problemas mais realísticos, pois, neste caso, não se pode assumir que possui-se de antemão todos os conhecimentos necessários sobre o ambiente, e ainda, com a devida exatidão. Há muitos problemas reais nos quais a geometria do espaço de trabalho pode ser apenas parcialmente conhecida no momento do planejamento. Como esta tarefa requer que o robô tenha um modelo do ambiente, quanto mais incompleto o modelo, mais limitado será o papel do planejamento.

Um outro problema relacionado ao planejamento de movimentos é o do controle de movimento em tempo real. A tarefa básica do controlador de tempo real é fazer com que o robô execute de forma adequada os movimentos previamente planejados, como por exemplo seguir uma trajetória.

Neste trabalho, será feita inicialmente uma breve revisão dos principais métodos de planejamento de trajetórias, indicando suas vantagens e situações nas quais cada um deles é mais adequado. A seguir, descreve-se o sistema de planejamento de trajetórias implementado. Trata-se de um planejador para funcionar em tempo real, interagindo com o sensoriamento dos obstáculos, bem como com o controlador de tempo real. Finalmente, são mostrados alguns resultados obtidos e faremos uma análise do sistema implementado.

CAPÍTULO II

CONCEITOS BÁSICOS

Primeiramente, serão introduzidas algumas notações as quais serão úteis no restante deste trabalho.

- O robô é denominado *A*, e é modelado, em geral, como um polígono para o caso do problema em duas dimensões, e como um objeto sólido no caso de três dimensões.
- Os obstáculos são denominados B_1, B_2,..., B_n, e são modelados da mesma forma que o robô.
- O espaço de trabalho do robô é denotado por *W* e é modelado como um espaço euclidiano $\mathbf{R}^n$, com n = 2 ou 3 (**R** representa o conjunto dos números reais).
- O espaço de configurações do robô *A* é denotado por *C*. Um elemento de *C* (ou seja, uma configuração) é denotado por *q*. A região em *W* ocupada por *A* na configuração *q* é denotada por *A(q)*.
- Um obstáculo *B* no espaço de trabalho é mapeado para uma região do espaço de configurações chamada obstáculo-C e denotada *CB*.

II.1. Problema Básico

Será definido agora um problema básico de planejamento de movimentos, com o intuito de se apresentar alguns conceitos e técnicas básicas, visualizando-os de forma mais clara.

Neste problema, assumi-se que o robô é o único objeto em movimento no espaço de trabalho, e ignorar-se-á as propriedades dinâmicas do robô. Restringi-se também os movimentos a aqueles que não envolvem contato, de forma que as interações provenientes de dois objetos físicos em contato podem ser desconsideradas. Estas restrições transformam o problema em questão em um problema puramente geométrico. Ainda faz-se a simplificação de que o robô é formado por um único objeto rígido cujos pontos são fixos uns em relação aos outros. Os movimentos do robô são restringidos apenas pela presença dos obstáculos.

Assim, o problema básico de planejamento de movimentos, resultante das simplificações acima, é o seguinte:

"Seja A o robô (um único objeto rígido) que pode movimentar-se em um espaço euclidiano W, denominado espaço de trabalho e representado como R^n *(n = 2 ou 3). Sejam* B_1, B_2, ..., B_m *os obstáculos (objetos rígidos) distribuídos em W. Assumir que as geometrias do robô e dos obstáculos, além das localizações destes últimos, sejam conhecidas com a devida precisão. O problema então é: dadas as configurações (posição e orientação) inicial e objetivo de A em W, gerar um caminho T que especifique uma seqüência contínua de posições e orientações de A, iniciando na configuração inicial e terminado na configuração objetivo, evitando contatos de A com os obstáculos B. Caso não exista tal caminho, deve-se declarar falha no processo."*

É claro que este problema está relativamente simplificado. Entretanto, mesmo assim não se trata de um problema simples. E ainda, a resolução mesmo deste problema mais básico possui interesse prático [3].

II.2. Noção de Caminho

Um caminho de A, partindo de uma configuração q_{inic} até uma configuração q_{fim}, é uma mapeamento contínuo:

$$T: [0,1] \rightarrow C \quad ,$$

onde:

C: espaço de configurações

$T(0) = q_{inic}$

$T(1) = q_{fim}$

Para que T seja contínuo, temos que, para todo $s \in [0,1]$, todas as variáveis que definem a configuração $T(s)$ devem tender às suas correspondentes em s_0 quando s tende a s_0.

II.3. Obstáculos no Espaço de Configuração

A definição anterior de caminho não leva em consideração os obstáculos. Será caracterizado agora o conjunto de caminhos os quais são solução para o problema básico na presença de obstáculos.

Cada obstáculo B_i, i =1..m, no espaço de trabalho W é mapeado em C para uma região:

$$CB_i = \{ q \in C \mid A(q) \cap B_i \neq \emptyset \}$$

Define-se então o conceito de espaço livre de configurações:

$$C_{livre} = C - \bigcup CB_i \quad , i = 1..m$$

E cada configuração deste conjunto é dita uma configuração livre.

Um caminho livre entre duas configurações livres q_{inic} e q_{fim} é um mapeamento contínuo T: [0,1] → C_{livre}, com $T(0) = q_{inic}$ e $T(1) = q_{fim}$. Duas configurações pertencem ao mesmo *componente conexo* de C_{livre} se, e somente se, eles são conectados por algum caminho livre.

Dadas as configurações livres inicial e final, o problema básico de planejamento de movimentos é gerar um caminho livre entre estas duas configurações, caso elas pertençam ao mesmo componente conexo, ou declarar falha em caso contrário.

II.4. Expansão de Obstáculos

Quando o robô A resume-se a um único ponto, não faz sentido falar a respeito de orientação. Neste caso, assim como W, o espaço C de configurações de A é uma cópia de $\mathbf{R}^n$, e é portanto um espaço euclidiano. Além disto, os obstáculos são idênticos em C e em W.

Quando A é um disco ou um objeto de dimensões conhecidas que pode apenas transladar (sem rotacionar), o espaço de configurações é também $\mathbf{R}^n$. Mas, nestes casos, os obstáculos em C são obtidos por expansão dos obstáculos em W de acordo com a forma de A, conforme ilustrado na figura 1.

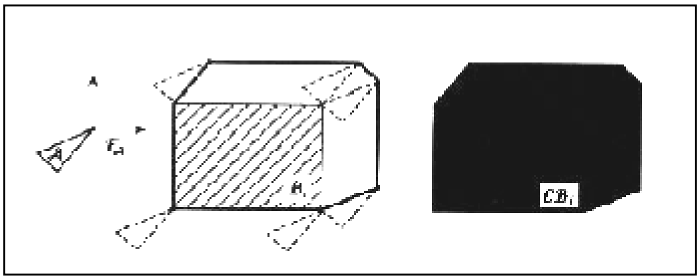


Figura 1: Expansão de obstáculos.

Esta técnica é de grande relevância neste trabalho, dado que o robô no qual são implementados os métodos aqui apresentados possui vista de topo circular. Isto permite que seja feita a expansão dos obstáculos, e depois disto pode-se considerar o robô como sendo um único ponto (o seu centro) para efeito do planejamento das trajetórias. Desta forma, uma configuração na qual o centro do robô não tocar nos obstáculos expandidos corresponderá a uma configuração real na qual certamente o robô não colide com os obstáculos do ambiente.

CAPÍTULO III

TÉCNICAS DE PLANEJAMENTO DE TRAJETÓRIAS

Existe um grande número de métodos para resolver problemas gerais de planejamento de movimentos. Entretanto, nem todos resolvem todas as generalizações deste problema. Por exemplo, alguns métodos necessitam que o espaço de trabalho seja bidimensional e os obstáculos poligonais. Apesar de muitas diferenças externas, os métodos são baseados em algumas técnicas gerais: *roadmap*, decomposição em células e campo potencial. A seguir, são brevemente descritos cada um destes métodos, além de ilustrados com simples exemplos em espaço de trabalho de duas dimensões com obstáculos poligonais, sendo o robô um polígono (ou ponto) que pode transladar neste espaço.

III.1. Roadmap

A técnica geral de *roadmap* para planejamento de trajetórias consiste em capturar a conectividade do espaço livre do ambiente em uma rede de curvas de uma dimensão, denominada *roadmap*. Uma vez que esta rede tenha sido obtida, ela é vista como um conjunto padrão de caminhos. O planejamento de trajetórias reduz-se então a conectar as posições iniciais e finais do robô ao *roadmap* e buscar neste um caminho entre estes dois pontos. Se existir um tal caminho, ele será dado pela concatenação de três sub-caminhos: um conectando a posição inicial até algum ponto do *roadmap*, outro sub-caminho pertencente ao *roadmap*, e finalmente um sub-caminho que leve do último ponto escolhido do *roadmap* até a posição desejada.

Vários métodos baseados nesta idéia geral foram propostos. Eles computam diferentes tipos de *roadmaps*, denominados grafo de visibilidade, diagrama de Voronoi, rede de caminho livre, e silhueta. Neste trabalho, apenas os grafos de visibilidade serão tratados com mais detalhes.

III.1.1. Grafos de Visibilidade

Considera-se um objeto poligonal (o robô) A que pode transladar em orientações fixas entre obstáculos poligonais em um espaço de trabalho bidimensional $W = \mathbf{R}^2$. Portanto, o espaço de configurações de A é $C = \mathbf{R}^2$ e a região de obstáculos de C, CB (união de todos os obstáculos de C) é uma região poligonal em $\mathbf{R}^2$. O espaço livre C_{livre} é igual a $C - CB$. São conhecidos algoritmos para calcular os limites de CB (e portanto de C_{livre}) a partir das descrições de A e dos obstáculos. Entretanto, como o objetivo final é o de vislumbrar a aplicação das técnicas aqui apresentadas no caso particular (robô com vista de topo circular), para o qual os limites de CB podem ser calculados facilmente a partir da técnica de expansão dos obstáculos apresentada anteriormente, não serão vistos mais detalhes a respeito deste problema. Para maiores detalhes, vide Latombe [3].

O princípio do método do grafo de visibilidade é construir um caminho livre como uma simples linha poligonal conectando as configurações inicial (q_{inic}) e final (q_{fim}) através de vértices de CB. Facilmente pode-se observar que existirá um caminho livre entre duas determinadas configurações se e somente se houver uma linha poligonal unindo q_{inic} a q_{fim} passando pelos vértices em CB. Isto implica em que, para encontrar-se um caminho livre entre quaisquer duas configurações, basta que o planejador considere apenas as linhas retas em que conectem os vértices de CB, além das posições inicial e final. E ainda, se existir algum caminho entre as configurações inicial e final, o menor caminho possível será constituído por um subconjunto destas linhas em consideração.

Define-se agora o que vem a ser o grafo de visibilidade. Grafo de visibilidade é um grafo G não-dirigido, e com as seguintes características:

- os vértices de G são q_{inic}, q_{fim} e os vértices de CB
- dois vértices de G estão conectados por uma aresta se e somente se eles o segmento de reta unindo eles for uma aresta de CB, ou se este segmento estiver totalmente contido em C_{livre}, com a possível exceção de suas extremidades

A figura 2 mostra o grafo de visibilidade para um espaço de configurações simples. Todos os vértices de *CB* são arestas de *G*. A linha mais espessa indica um caminho escolhido.

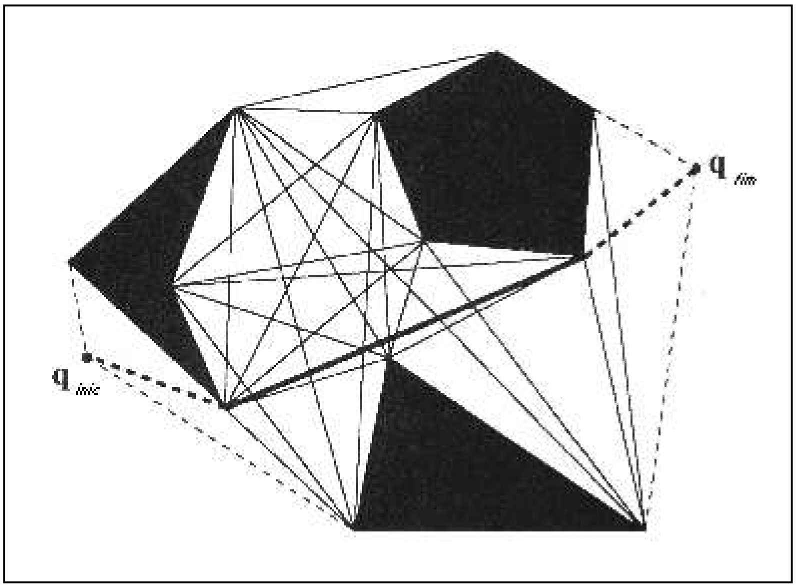


Figura 2: Grafo de visibilidade de um espaço de configurações bidimensional com obstáculos poligonais.

O algoritmo do método do grafo de visibilidade é o seguinte:

1. Construa o grafo de visibilidade *G*
2. Busque em *G* um caminho partindo de q_{inic} até q_{fim}
3. Se existir tal caminho, retorne-o; caso contrário, indique falha no processo

Um algoritmo simples para construção do grafo *G* consiste em considerar todos os pares (*u*, *v*) de vértices em *G*. Se *u* e *v* forem extremidades de um mesmo segmento de *CB*, então a aresta (*u*, *v*) é inserida em *G*. Caso contrário, deve-se analisar a interseção de *CB* com do segmento de reta aberto (sem considerar as extremidades) que liga os pontos correspondentes a *u* e *v*. A aresta (*u*, *v*) pertencerá a *G* se e somente se esta interseção for nula. Este algoritmo possui complexidade $O(n^3)$, onde *n* é o número total de vértices de *CB*.

Este algoritmo pode ser melhorado pelo uso de uma técnica denominada varredura, amplamente utilizada em algoritmos de geometria computacional. Este algoritmo permite que o grafo de visibilidade seja construído em tempo O($n^2 \log n$). Há ainda algoritmos que resolvem este problema em O(n^2), como o proposto por Welzl [9].

Após construído o grafo de visibilidade, deve-se então utilizar algum algoritmo de busca para escolha de um caminho. Pode-se aqui utilizar tanto algoritmos exatos (que garantem que a melhor solução possível será encontrada), quanto algoritmos heurísticos (que em geral levam a uma solução boa, mas não necessariamente ótima, porém com um custo computacional menor). Utilizando-se o algoritmo clássico de Dijkstra para o menor caminho [5], o processo de busca pode ser feito em tempo O(n^2). Este é um algoritmo exato, levando a um caminho tão curto quanto possível.

O método do grafo de visibilidade pode ter sua performance melhorada se percebermos que algumas arestas de *G*, criado conforme descrito anteriormente, nunca serão utilizadas. Dados dois obstáculos, são úteis apenas as arestas que determinam segmentos tangentes a ambos os obstáculos, conforme ilustrado na figura 3:

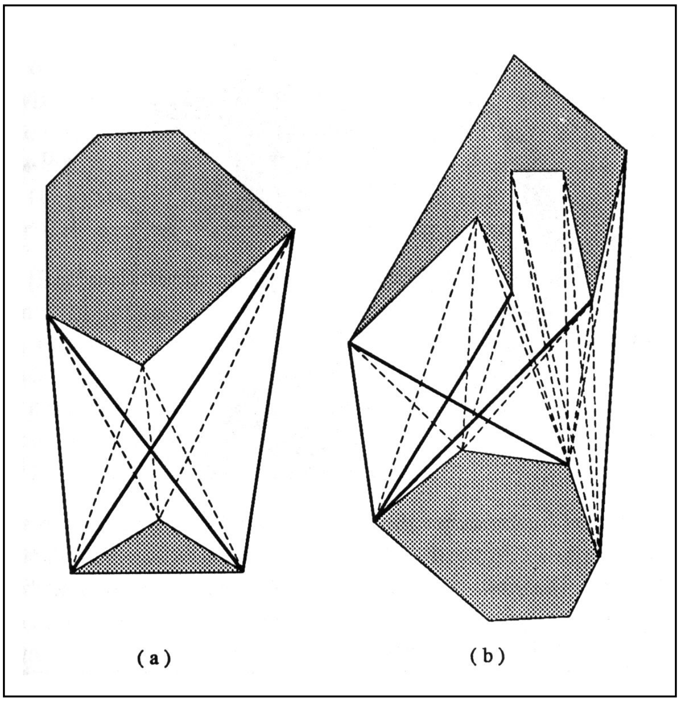


Figura 3: Segmentos tangentes entre dois polígonos.

Os segmentos de reta tangentes aos polígonos encontram-se desenhados em linhas cheias. Em linhas tracejadas, temos as arestas que são eliminadas pela aplicação desta técnica. Nesta figura, pode-se ainda verificar duas propriedades dos segmentos tangentes:

- Entre dois polígonos convexos disjuntos, existem exatamente quatro segmentos tangentes. Em relação a dois destes segmentos, ambos os polígonos encontram-se no mesmo lado; em relação aos outros dois, cada polígono encontra-se de um lado da reta suporte do segmento.
- Se *v* for um vértice côncavo então *v* não faz parte de nenhum segmento tangente.

Ao grafo *G'* obtido pela eliminação dos segmentos não-tangentes, dá-se o nome de *grafo de visibilidade reduzido.* E ainda, se existir em *G* um caminho unindo q_{inic} e q_{fim}, *G'* conterá as arestas correspondentes ao seu caminho mínimo. Na figura 4, está ilustrado o grafo de visibilidade reduzido correspondente ao grafo da figura 2.

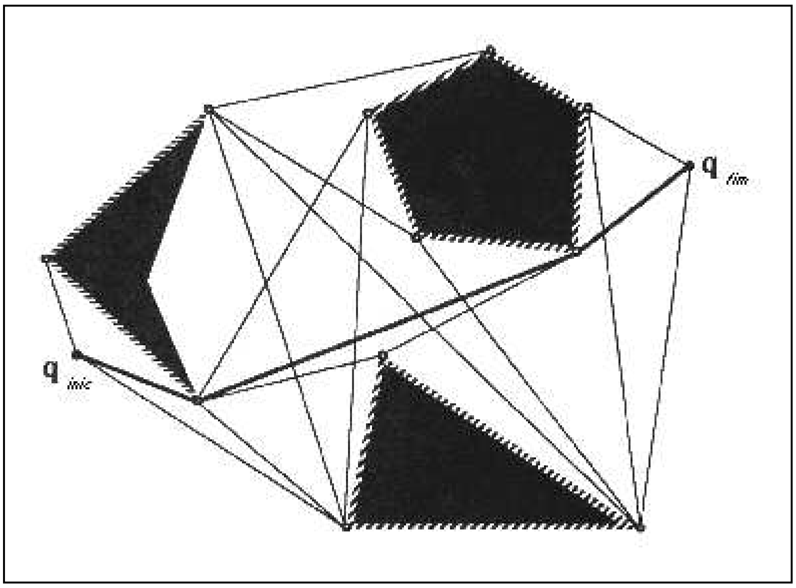


Figura 4: Grafo de visibilidade reduzido, correspondente ao grafo de visibilidade da figura 2.

São conhecidos algoritmos O(log n_1 + log n_2) para calcular os segmentos tangentes de dois polígonos convexos com n_1 e n_2 vértices, respectivamente.

O método do grafo de visibilidade pode ser estendido para o caso no qual os obstáculos-C são polígonos generalizados. Por polígonos generalizados, quer-se dizer regiões limitadas por linhas retas e/ou arcos de círculos. Este tipo de obstáculos-C são encontrados em casos quando *A* é um polígono generalizado que translada em orientações fixas entre obstáculos os quais também são modelados como polígonos generalizados. Esta extensão é particularmente útil quando os obstáculos no espaço de trabalho são melhor modelados como polígonos generalizados, o que acontece freqüentemente no caso de robôs móveis. Isto também é útil quando tem-se obstáculos-C poligonais os quais são expandidos em polígonos generalizados por um disco de raio ε para garantir-se uma distância mínima entre os pontos da trajetória gerada e os obstáculos reais.

III.2. Decomposição em Células

O método de decomposição em células consiste em dividir o espaço livre do robô em regiões simples (células), de forma que um caminho entre quaisquer duas configurações em uma mesma célula possa ser facilmente obtido. Um grafo não-dirigido representando a relação de adjacência entre as células é construído, e é então efetuada uma busca no mesmo. Este grafo é denominado *grafo de conectividade*. Os vértices que compõem este grafo são as células extraídas do espaço livre do robô. Há uma aresta entre dois vértices se e somente se as células correspondentes a eles são adjacentes. O resultado da busca efetuada é uma seqüência de células denominada *canal.* Um caminho contínuo pode ser computado a partir do canal.

Os métodos baseados em decomposição em células podem ser divididos ainda em *exatos* e *aproximados*:

- Métodos exatos de decomposição em células decompõem o espaço livre em um conjunto de células cuja união cobre exatamente o espaço livre.
- Métodos aproximados de decomposição em células dividem o espaço livre em um conjunto de células de forma predefinida cuja união está estritamente contida no espaço livre.

Uma diferença que decorre diretamente das definições acima é a seguinte. Os métodos exatos são ditos *completos*, isto é, eles permitem que um caminho entre duas quaisquer configurações seja obtido sempre que for possível, dado que seja utilizado um algoritmo de busca apropriado. Por outro lado, os métodos aproximados podem não ser completos, pois podem, dependendo da precisão utilizada, não encontrar um caminho entre duas configurações mesmo que este exista. Entretanto, na maioria das vezes, a precisão destes métodos pode ser ajustada arbitrariamente, a custo de espaço de armazenamento e tempo de processamento. Contudo, os métodos aproximados são mais simples, sendo os que são utilizados com maior freqüência na prática.

A seguir, são apresentadas com mais detalhes ambas as classes de métodos de decomposição em células. Pode-se adiantar que o programa desenvolvido neste projeto é baseado no método aproximado de decomposição em células, conforme será apresentado no capítulo IV.

III.2.1. Método Exato de Decomposição em Células

O princípio do método exato de decomposição em células é primeiramente decompor o espaço livre do robô em uma coleção de regiões que não se sobrepõem, e cuja união é exatamente C_{livre}. Depois, é construído o grafo de conectividade, representando a relação de adjacência entre as células, e então é feita uma busca no mesmo. Se existir um caminho entre as configurações inicial e final, será gerado um canal (seqüência de células, sendo cada uma delas adjacente à anterior e à posterior), unindo as células correspondentes às configurações inicial e final. Finalmente, é construído um caminho a partir desta seqüência.

Todavia, nem todas as decomposições em células são apropriadas. Por exemplo, uma visão extrema consideraria todo o espaço livre como uma única célula, mas assim a obtenção de um caminho dentro desta célula recairia no problema original. Por isto, é desejado que as células geradas tenham as seguintes características:

1. A geometria de cada célula deve ser simples o bastante para tornar fácil a obtenção de um caminho entre duas configurações dentro de uma mesma célula.
2. Não deve ser difícil testar a adjacência de duas células quaisquer, bem como encontrar uma caminho que cruze a fronteira entre duas células adjacentes.

Neste trabalho, é apresentado apenas um método baseado na decomposição exata em células. Este método se aplica a problemas nos quais o espaço de configurações C é bidimensional e os obstáculos-C são poligonais. O método apresentado considera o robô como um único ponto, e é baseado na decomposição poligonal do espaço livre. Há outros métodos mais elaborados, como por exemplo o apresentado por [6], que considera o robô como um segmento de reta.

III.2.1.1. Decomposição em Células em Espaços de Configurações Poligonais

Será agora apresentado o método exato de decomposição em células no caso em que $C = \mathbf{R}^2$ e que a região dos obstáculos-C CB (união dos obstáculos-C) forma uma região poligonal em C. A figura 5 ilustra um espaço de configurações com estas características.

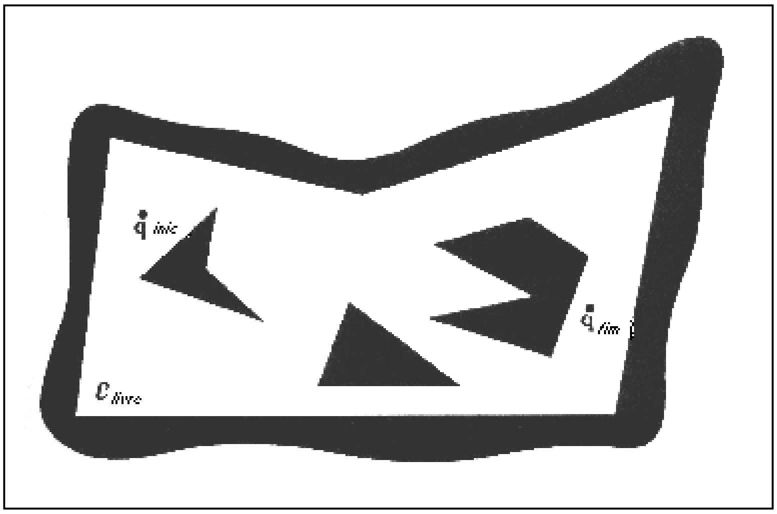


Figura 5: Espaço de configurações bidimensional com obstáculos poligonais, utilizado para exemplificar o método exato de decomposição em células.

A seguir, tem-se as definições da decomposição de C_{livre} e do grafo de conectividade associado.

Uma decomposição poligonal convexa K de C_{livre} é uma coleção finita de polígonos convexos (células), tal que os interiores de quaisquer duas células não possuem interseção, e a união de todas as células é igual a C_{livre}. Duas células k e k' de K são adjacentes se e somente se $k \cap k'$ é um segmento de reta de comprimento não nulo.

O grafo de conectividade G associado com uma decomposição poligonal convexa K de C_{livre} é o grafo não-dirigido assim especificado:

- Os vértices de G são as células de K.
- Dois vértices de G possuem uma aresta entre si se e somente se as células correspondentes são adjacentes.

A figura número 6 mostra uma decomposição poligonal convexa do espaço de configurações apresentado na figura 5, além do grafo de conectividade associado.

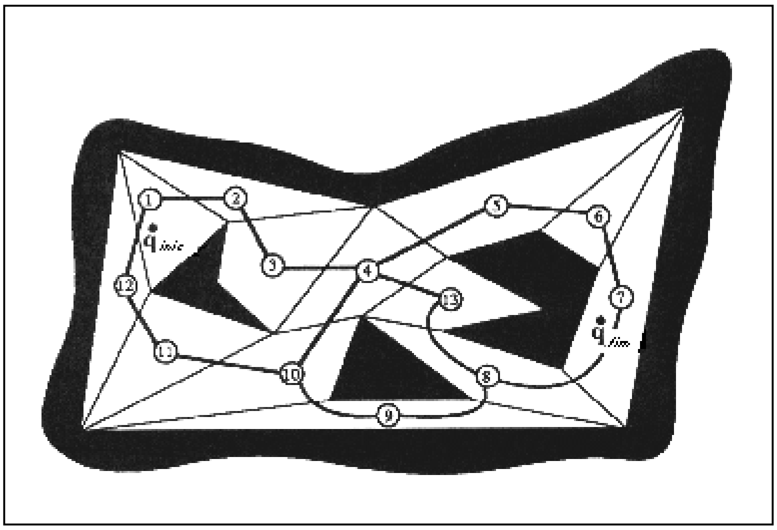


Figura 6: Espaço livre decomposto de forma exata em um conjunto de células poligonais.

Seja uma configuração inicial q_{inic} e uma configuração final q_{fim} em C_{livre}. O algoritmo de decomposição exata em células para planejamento de um caminho conectando estas duas configurações é o seguinte:

1. Gerar uma decomposição poligonal convexa K para C_{livre}.
2. Construir o grafo de conectividade G associado a K.
3. Buscar em G uma seqüência de células adjacentes entre q_{inic} e q_{fim}.

4. Se a busca obtiver sucesso, retornar a seqüência gerada de células; caso contrário, retornar falha.

A saída deste algoritmo é portanto uma seqüência $k_1, k_2, ..., k_p$ de células tais que $q_{inic} \in k_1$, $q_{fim} \in k_p$, e para cada j ($1 <= j < p$), k_j e k_{j+1} são adjacentes. Esta seqüência compõem o canal. Por exemplo, na figura 6, q_{inic} está na célula 1, e q_{fim} na célula 7. Um possível canal entre estas células seria a seqüência das células 1, 2, 3, 4, 13 ,8 e 7.

O interior do canal está totalmente compreendido dentro do espaço livre. Considere β_j como sendo o segmento de reta que separa as células k_j e k_{j+1} da seqüência. Uma forma simples de gerar um caminho livre contido no interior do canal produzido pela busca seria considerar os pontos médios Q_j dos segmentos β_j, para todo segmento β_j, e conectar q_{inic} a q_{fim} através de uma linha poligonal cujos vértices são Q_1, ..Q_{p-1}. Como as células são polígonos convexos, é imediato verificar que o caminho assim construído estará totalmente contido no espaço livre. Esta construção está ilustra da na figura 7.

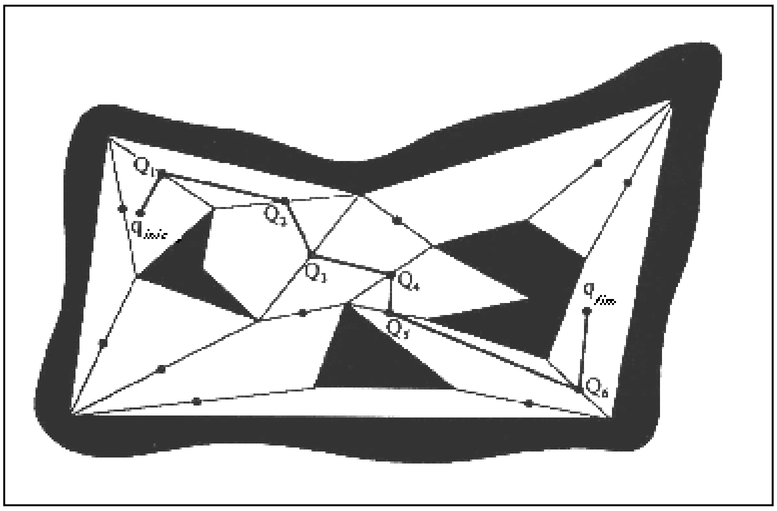


Figura 7: Escolha de um caminho poligonal a partir do canal.

A decomposição convexa ótima de um polígono côncavo, minimizando o número de polígonos, é um problema clássico em geometria computacional, tendo sido propostos vários algoritmos com este propósito, como por exemplo um algoritmo baseado na técnica *sweeping-line*, cuja complexidade de tempo é O(n log n), onde n é o número de vértices do polígono original [3]. Entretanto, quando o polígono pode conter furos (com ocorre em ambientes com obstáculos poligonais), este problema torna-se NP-difícil [7]. Entretanto, uma decomposição não ótima para este caso pode se gerada eficientemente, conforme apresenta [3].

Cabe mencionar ainda que este método pode ser estendido para o caso tridimensional, com obstáculos poliédricos [3].

III.2.2. Método Aproximado de Decomposição em Células

O método de decomposição em células aproximada consiste, assim como o método exato, em representar o espaço livre do robô como um conjunto de células. A diferença entre estes dois métodos reside no fato de que no método aproximado as células possuem uma forma simples, definida previamente. Como conseqüência desta característica, tem-se que em geral não é possível modelar o ambiente na forma exata como ele é, sendo necessário que algumas aproximações sejam feitas. Isto deve ser feito de uma forma conservadora, ou seja, de maneira que garanta que o robô não colida com os obstáculos. Por este motivo, o espaço livre modelado aproximadamente através das células tem que estar estritamente contido no espaço livre real do robô. Pode decorrer disto que não seja encontrado algum caminho entre duas configurações, embora exista. Devido a esta característica, o método aproximado de decomposição em células é dito não completo, ao contrário do método exato.

As principais razões do uso deste método são:

1. fazer a decomposição do ambiente em células através da iteração da mesma computação, que será simples por causa da forma das células
2. não ser sensível, de certa forma, a computações numericamente aproximadas

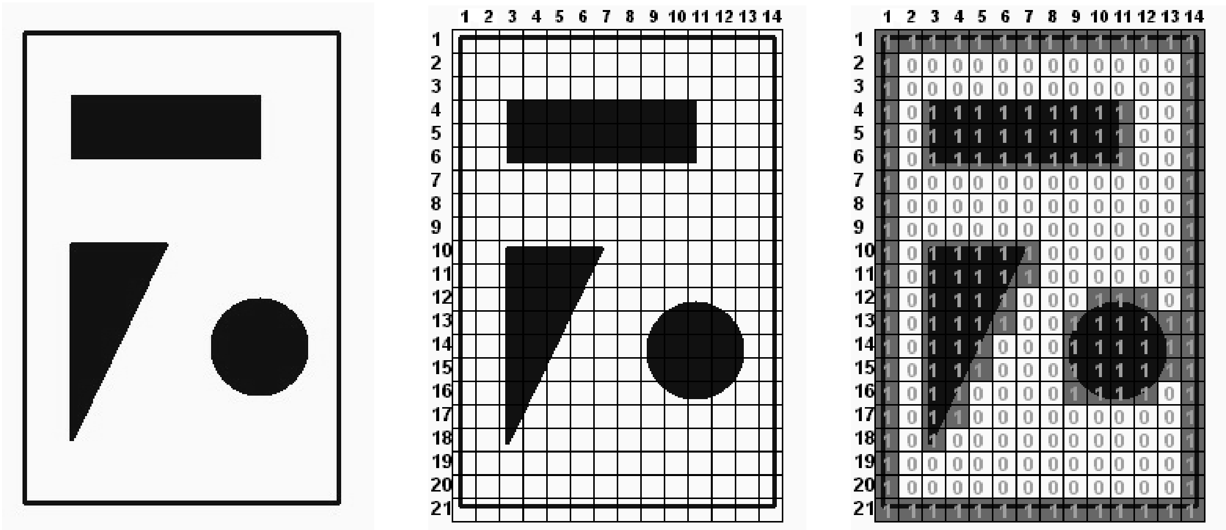
Assim, o método aproximado geralmente dá origem a sistemas mais simples de planejamento de trajetórias do que os que utilizam o método exato. E por isto, o método aproximado tem sido mais utilizado na prática. Vale acrescentar ainda que com o método aproximado, pode-se ajustar a precisão conforme desejado mudando, para tanto, apenas o tamanho das células. Por estas razões, o modelo de planejamento de trajetórias implementado neste projeto baseia-se no método de decomposição aproximada em células.

Entretanto, é importante salientar que possibilidade de ajuste da precisão deste método está intimamente relacionada à quantidade de espaço e de tempo de execução do planejador de trajetórias. Não deve-se definir previamente o tamanho das células, uma vez que este deve ser ajustado com base em diversos parâmetros. Como foi dito, células de tamanho grande reduzem a quantidade de memória necessária e, principalmente, o tempo de execução, o que é muito importante em sistemas que operam em tempo real, como o proposto neste trabalho. Por outro lado, células muito células muito grandes podem causar uma perda de precisão indesejada, fazendo com que não sejam encontradas trajetórias mesmo quando for possível.

Teoricamente, este método pode ser aplicado a casos mais gerais do problema de planejamento de movimentos. Porém, na prática, ele é normalmente aplicado a problemas de dimensões relativamente pequenas (até quatro) [3]. Isto ocorre por causa das complexidades de tempo e de espaço dos algoritmos envolvidos em função das dimensões do problema. Nestes casos mais simples, como no caso de um robô poligonal que pode transladar e rotacionar livremente entre obstáculos poligonais em um espaço de trabalho de duas dimensões, este método leva vantagem em relação aos demais.

Afora estas diferença mencionadas acima, o método aproximado é semelhante ao método exato. Ou seja, após ser feita a decomposição do ambiente em células, deve-se construir o grafo de conectividade correspondente, e buscar neste uma seqüência de células, na qual células subseqüentes sejam adjacentes, e escolher uma trajetória que vá desde a primeira célula (onde encontra-se q_{inic}) até a última (onde está q_{fim}). Portanto, exemplificaremos como pode-se fazer a decomposição aproximada em células, numa forma bastante simples. Para o restante do processo, de uma forma geral, valem as técnicas apresentadas anteriormente para o método exato.

Uma forma bastante simples de representar o ambiente e seus obstáculos é através de um mapa de *bits*. Este mapa deve conter um número de dimensões igual ao da modelagem utilizada para o problema. Utiliza-se aqui o problema bidimensional, sendo o mapa de *bits* portanto uma matriz de duas dimensões. O ambiente real, incluindo seus obstáculos, está ilustrado na figura 8.



(a) (b) (c)

Figura 8: Decomposição aproximada em células

Adota-se que cada célula representará uma região quadrada, de lado arbitrário. Então cada posição do mapa de *bits* conterá 1 se a interseção da célula correspondente com os obstáculos for não nula. A figura 8 ilustra isto. O planejamento da trajetória deve então escolher uma certa seqüência de células marcadas com 0.

Verifica-se que com este método, não há custo adicional em armazenar o grafo, e o custo de testar se duas células são adjacentes é muito baixo. Isto pode ser testado simplesmente verificando-se as diferenças dos índices correspondentes às células no mapa de *bits*. Se todas estas diferenças forem nulas, exceto uma, a qual deve ser unitária, então diz-se que as células são adjacentes. Caso contrário, elas não o são.

III.3. Campo Potencial

O último método tratado neste trabalho é o método do campo potencial. Neste, o espaço de configurações é discretizado em uma fina grade regular de configurações. A cada posição desta grade é associado um valor de uma função, com a qual pode-se fazer a analogia de um campo potencial. Então, como o tamanho da grade será bastante grande (pois ela é fina), são utilizados métodos heurísticos para encontrar um caminho no amplo espaço de buscas resultante.

A metáfora que a terminologia sugere é que o robô, representado por um ponto no espaço de configurações, seja uma partícula movendo-se sob a influência de um campo potencial artificial gerado obstáculos. Este campo provoca a repulsão do robô. Por outro lado, a configuração objetivo gera um campo potencial que atrai o robô. A idéia então é que, a cada configuração, a direção do movimento do robô seja determinada pela força resultante proveniente do campo potencial nesta determinada configuração.

A figura 9 ilustra a noção de potencial repulsivo e atrativo, bem como a combinação deles. O potencial atrativo (b) é um parabolóide com ponto de mínimo localizado na configuração objetivo. Já o potencial repulsivo (c) é diferente de zero somente a partir de uma determinada distância dos obstáculos, aumentando com a proximidade destes. O caminho entre a origem e o destino (e) é construído pela direção oposta à do gradiente do potencial resultante (d) em cada configuração. Em (f) temos ilustrada uma matriz das orientações do vetor gradiente de campo negado, que são as orientações das forças artificiais induzidas pelo campo potencial.

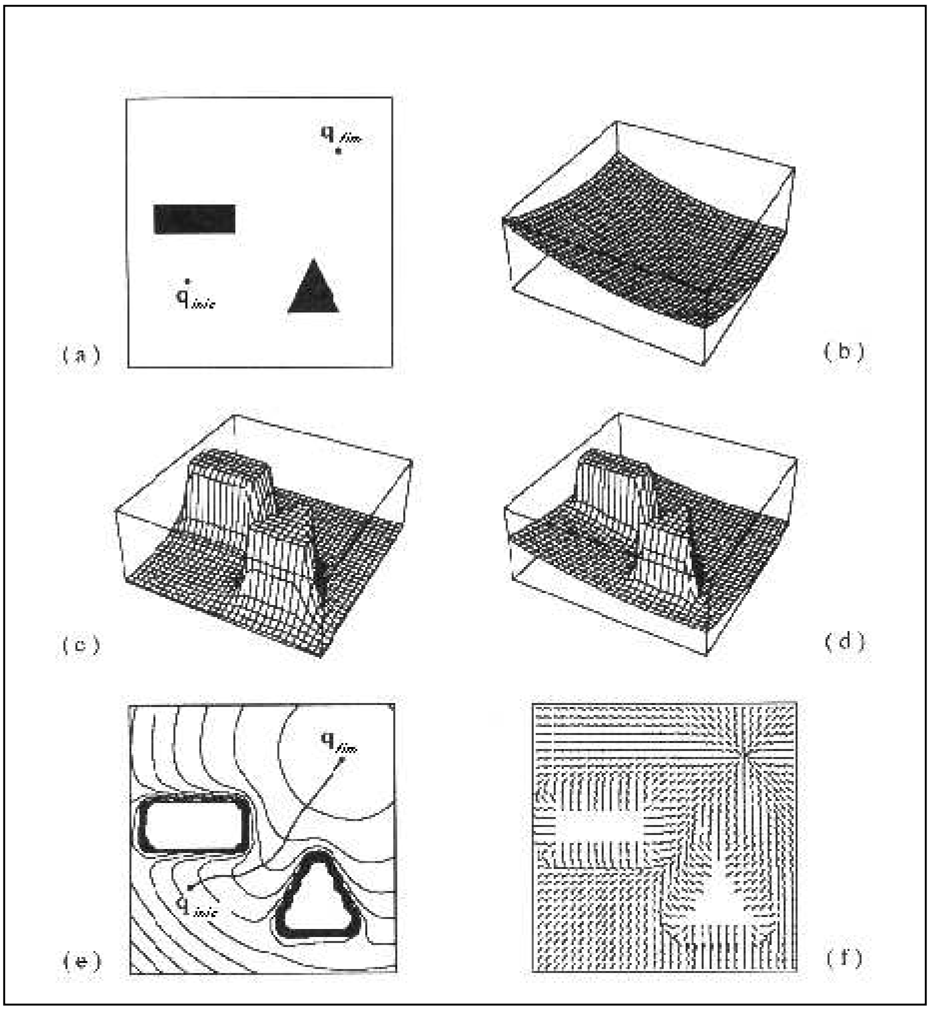


Figura 9: Método do campo potencial

Em comparação com outros métodos, o de campo potencial pode ser bastante eficiente. Porém, eles possuem uma grande desvantagem. Como eles são baseados em métodos rápidos de otimização, eles podem levar o robô a uma configuração que seja um ponto de mínimo local da função potencial, diferente da configuração almejada. Uma forma de atacar este problema é pela construção de funções potenciais que não possuam mínimos locais, pelo menos no subconjunto conectado do espaço livre onde se encontra a configuração objetivo. Contudo, isto quase nunca é uma tarefa fácil, principalmente em espaços de configurações com vários obstáculos. Uma outra técnica empregada é adicionar, ao esquema básico de campo potencial, um mecanismo suficientemente poderoso para escapar de mínimos locais. Há vários métodos de otimização para este tipo de problemas, em especial as meta-heurísticas, como *simulated annealing*, algoritmos genéticos, GRASP (*Greedy Randomized Adaptative Search Procedures*), busca tabu, e times assíncronos [8]. Entretanto, não há nenhum método que garanta, para qualquer função de otimização, que não se cairá em um mínimo local.

CAPÍTULO IV

O MODELO IMPLEMENTADO

Neste capítulo, é apresentado o sistema de planejamento de trajetórias implementado neste projeto. Inicialmente, implementou-se o método do grafo de visibilidade. Foi verificado, através de simulações, a eficiência deste método quando tem-se uma descrição bastante completa do ambiente. Entretanto, decidiu-se neste projeto implementar um método que fosse bastante adequado para casos em que não se tem informações completas (ou mesmo nenhuma informação) sobre o ambiente no qual o robô se encontra. Nestes casos, o método do grafo de visibilidade não se mostrou suficientemente eficiente para compor um planejador de trajetórias em tempo real. Isto se deve ao fato de que neste método é preciso ter-se uma descrição geométrica dos obstáculos, a qual se torna custosa de ser obtida quando se trata da descoberta de obstáculos a partir de leituras de sonares, como é o caso.

Assim, optou-se por implementar o planejador de trajetórias baseado no método de decomposição aproximada em células, o qual se mostra mais adequado à situação a qual nos propusemos a tratar. Será visto com mais detalhes agora de que se trata esta situação. O robô móvel Twil, objetivo da aplicação deste projeto, possui dois sonares ultra-sônicos que são utilizados para detecção de obstáculos, e para os quais já havia uma série de rotinas desenvolvidas [14][18]. Além disto, também já estavam implementados alguns controladores para este robô, que possui duas rodas não orientáveis acopladas a motores [13][15][16][17]. Assim, percebeu-se que seria útil se fosse desenvolvido um planejador de trajetórias para trabalhar em tempo real, interagindo tanto com os sonares para detecção dos obstáculos, bem como com o controlador para fornecimento da trajetória. E então isto foi o que nos propusemos a fazer. Além disto, foi implementado um esquema para permitir que o usuário possa fornecer *a priori* a localização de obstáculos. Desta forma, seria possível para o robô tanto navegar em ambientes sobre os quais se tem pouco ou nenhum conhecimento prévio, como movimentar-se de forma eficiente em ambientes previamente conhecidos.

IV.1. Robô Móvel Twil

Na figura 10, temos uma foto do robô móvel Twil, que foi desenvolvido no setor de Eletro-eletrônica do Departamento de Física desta Universidade.

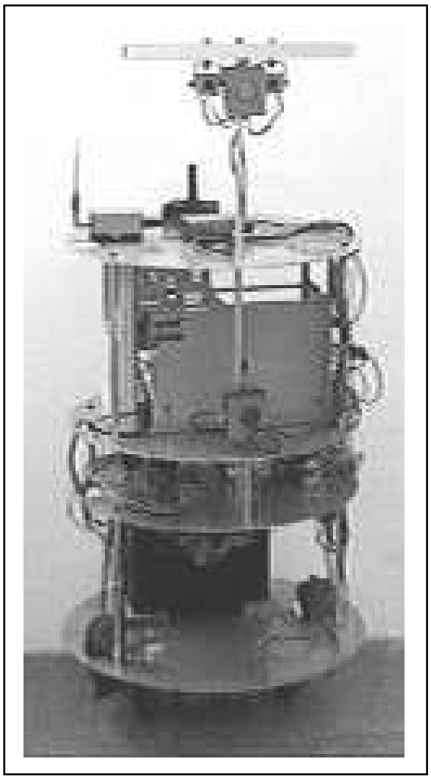


Figura 10: Robô móvel Twil

Suas principais características são:

- Possui geometria cilíndrica, com raio de 30 cm e altura de 1,35 m
- Dois sonares ultra-sônicos
- Duas câmeras de vídeo
- Duas rodas não orientáveis acopladas a motores
- Um computador PC com processador Pentium MMX 233 MHz, com sistema operacional Linux
- Uma placa de rede Ethernet
- Um modem sem fio
- Um joystick para movimentação manual do robô
- Uma bateria
- Placas de interface com os vários dispositivos

• Uma bússola digital

A figura 11 mostra o diagrama de blocos deste robô. Nesta figura pode ser observado como os diversos dispositivos interagem entre si.

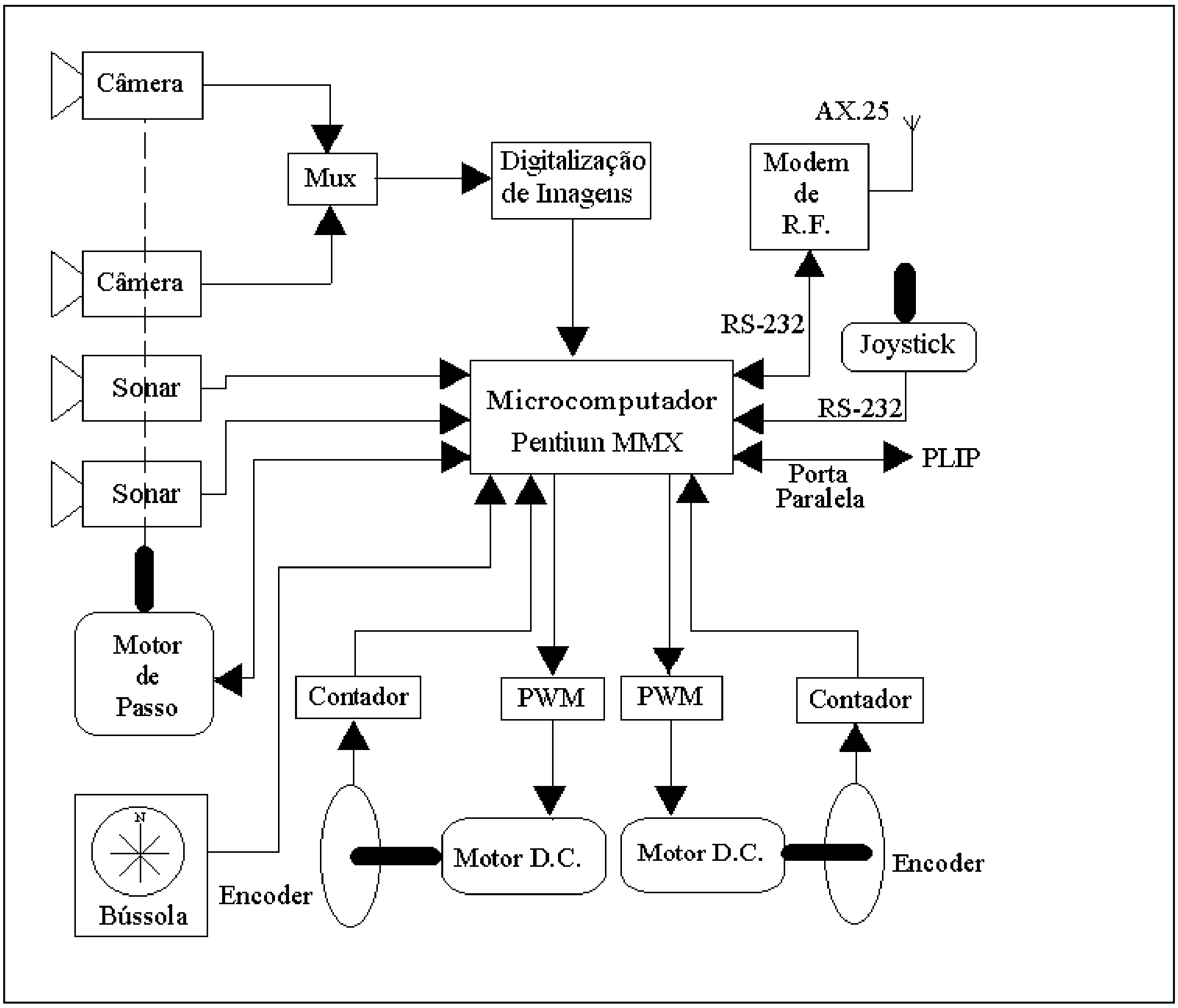


Figura 11: Diagrama de blocos do robô Twil.

IV.2.Método

Como mencionado anteriormente, o método de planejamento de trajetórias utilizado foi o de decomposição em células, sendo usado para tanto um mapa de bits para armazenar um ambiente retangular de tamanho ajustável, e com células quadradas de lado que também pode ser configurado. Além disto, devido à geometria favorável do robô, pode-se aplicar o processo de expansão de obstáculos anteriormente apresentado, e tratar o problema como o de um robô puntual movimentando-se em um ambiente bidimensional. E ainda, o robô pode ser rotacionado sem que haja movimento de translação, o que permite que seja desconsiderada a sua orientação no espaço de configurações, obtendo assim um problema com duas dimensões.

Para efetuar a busca de uma trajetória no espaço de configurações, foi utilizado um algoritmo de busca em largura. Esta escolha foi feita com base em diversas características, como eficiência, garantia de encontrar uma solução sempre que possível, e facilidade de implementação. Considera-se duas células adjacentes conforme ilustrado na figura 12. As células em branco são parte do espaço livre do robô, enquanto que as escuras indicam a presença de um obstáculo. O robô inicialmente encontra-se na célula central. Então ele poderá se mover para qualquer uma das células adjacentes a esta célula inicial. Duas células são adjacentes caso elas possuam um lado comum e nenhuma delas esteja marcada como obstáculo.

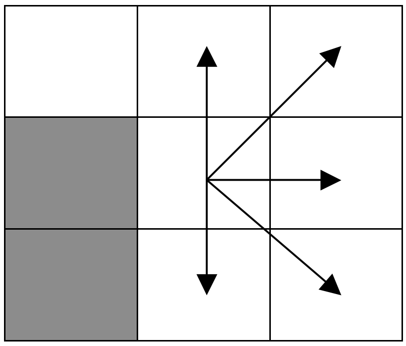


Figura 12 : Células vizinhas (adjacentes).

Além disto, foi feita ainda a seguinte consideração. Duas células não marcadas como obstáculo serão consideradas adjacentes se elas possuírem um vértice em comum e as duas células que possuem lado adjacente a ambas as primeiras também não forem marcadas como obstáculo.

IV.3. Modelo de Implementação

O sistema foi desenvolvido em linguagem C, utilizando-se *threads* [11] para modelar cada uma das tarefas envolvidas no processo. A sincronização entre os *threads* foi feita através de semáforos. Os *threads* implementados foram os seguintes:

- *Thread* Principal: inicializa os demais *threads* e os semáforos, além de outras inicializações;
- *Thread* Gerador de Trajetória: faz a geração da trajetória a ser seguida pelo robô;
- *Thread* Fornecedor de Trajetória: passa a trajetória de controle para o controlador, aguardando que ele termine de fornecê-la;
- *Thread* Controlador: faz o controle da trajetória do robô, de acordo com o planejamento anterior;
- *Thread* Detector de Obstáculos: solicita leituras do sonar e marca no mapa de *bits* os obstáculos encontrados; invoca o Gerador de Trajetórias;
- *Thread* Leitor do Sonar: faz uma leitura do ambiente utilizando o sonar e detecta as regiões de profundidade constantes [14]

Destes, o *thread* controlador [13] e o leitor do sonar [14] foram adaptados de programas já desenvolvidos para o robô Twil para interagir com os demais *threads*.

A estrutura deste sistema com múltiplos *threads* está ilustrada na figura 13. Os *threads* estão representados pelas bolhas. Os retângulos representam os dispositivos físicos do robô que interagem com o sistema desenvolvido. As setas representam o fluxo de comunicação entre eles. O funcionamento destes *threads*, inclusive a comunicação entre eles, serão descritos mais adiante. Antes disto, será explicado como foi desenvolvido o simulador utilizado para testar este modelo de implementação projetado.

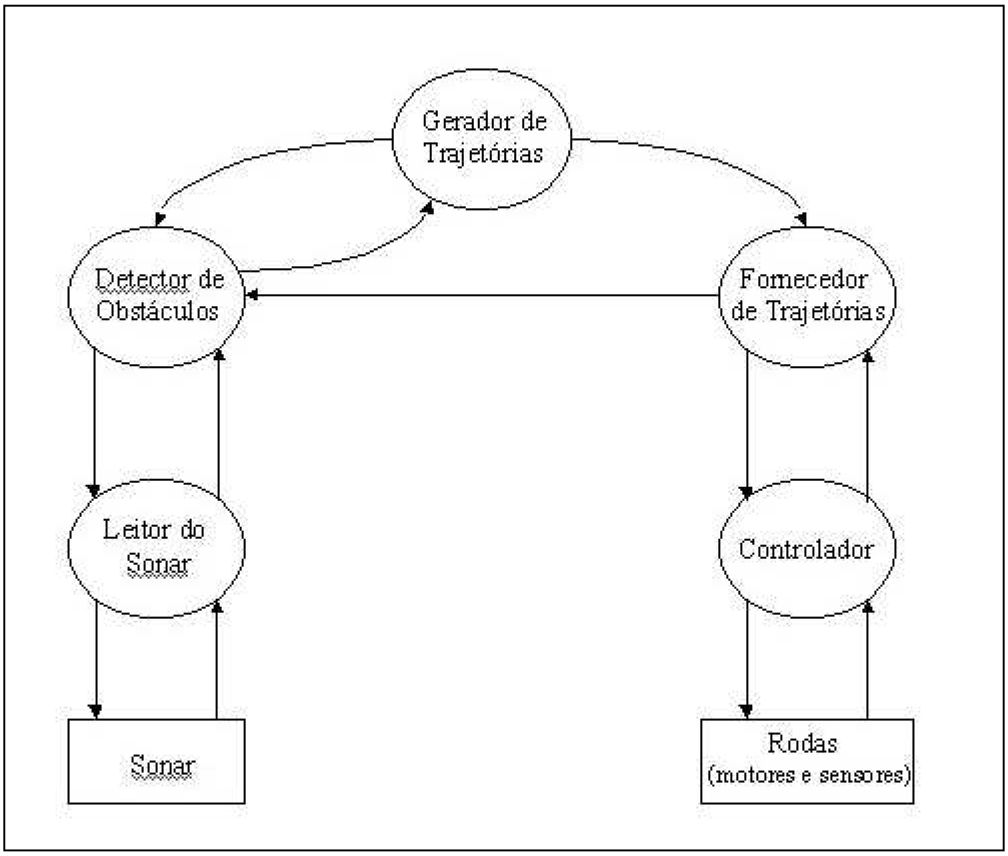


Figura 13: Estrutura de *threads* do modelo implementado

IV.4. Simulador

Para testar e validar o modelo que se estava implementando, desenvolveu-se um simulador. Neste, os *threads* de leitura do sonar e controle eram simulados por um outro *thread*, o qual tinha conhecimento do ambiente real que se estava simulando. Assim, este *thread* fornecia os obstáculos que seriam detectáveis a partir de cada posição pelo sonar, e também simulava que o robô seguia a trajetória, aproveitando desta situação para já verificar se ocorreria alguma colisão do robô com os obstáculos do ambiente.

Inicialmente, o simulador foi construído assumindo que as leituras do sonar poderiam ser feitas com o robô em movimento. Entretanto, as rotinas que estavam disponíveis para serem utilizadas no momento em que foi necessário neste projeto funcionam apenas para o robô parado, pois não levam em consideração os movimentos realizados pelo robô durante a varredura do sonar. Por este motivo, tive-ve que alterar um pouco o sistema que havia sido implementado. Passou a ser necessário que o robô, apesar de ter toda uma trajetória até o seu objetivo já planejada, movimente-se somente até uma posição na qual ele pode ter certeza de que não há obstáculos, ou seja, uma posição que esteja dentro da faixa de alcance do sonar. Daí então é feita uma nova varredura do ambiente e replanejada a trajetória do robô. Caso não haja algum obstáculo novo que não tivesse sido considerado no planejamento anterior, o novo trecho de trajetória a ser seguida será o mesmo do planejamento anterior. Certamente esta forma de implementação não é a mais desejada, pois é gasto tempo extra para o robô parar e efetuar nova varredura do ambiente através do sonar. Porém, é o que pode ser obtido com base nas rotinas que se encontravam desenvolvidas para o Twil. Contudo, os programas desenvolvidos neste projeto podem ser alterados, sem grande custo, para voltarem a operar como anteriormente, isto é, efetuando a varredura do ambiente com o uso do sonar, bem como replanejando a trajetória do robô, ao mesmo tempo em que o robô se movimenta. A figura 13, tem-se um exemplo dos resultados obtidos em simulação com o método antigo.

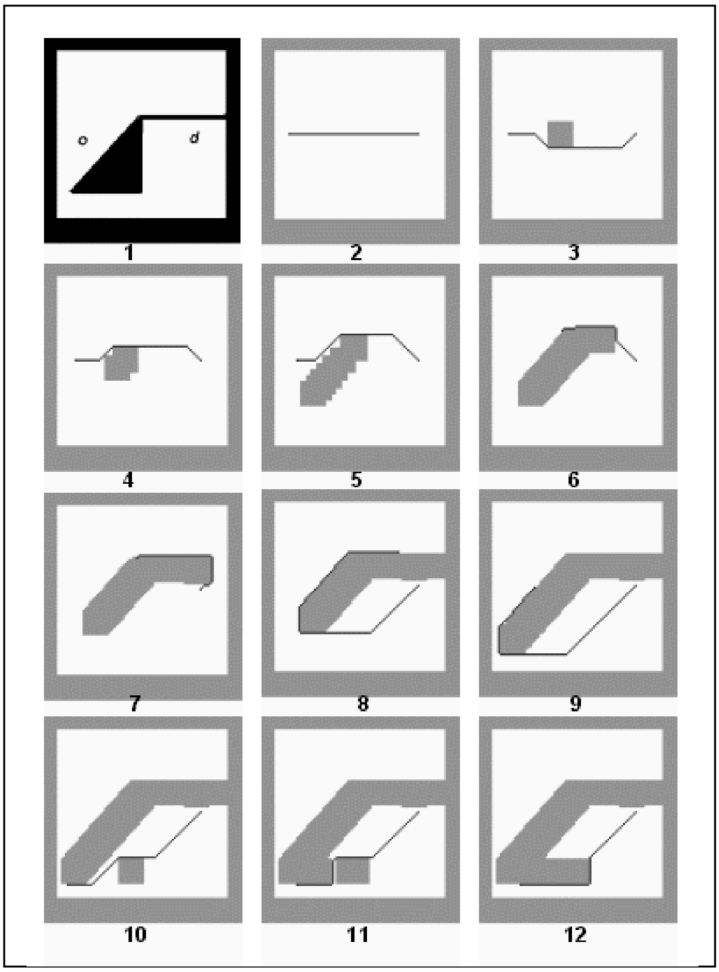


Figura 14: Exemplo de simulação com detecção de obstáculos simultânea ao movimento.

O ponto marcado com *O* é a origem, ou seja, o ponto inicial do robô. O ponto *D* é o de destino. A figura 14.1 é o ambiente real que está sendo simulado. Na figura 14.2, tem-se em cinza claro os obstáculos que foram informados *a priori*, e em preto a primeira trajetória gerada pelo planejador de trajetórias. À medida que o robô vai se movimentando, novos obstáculos vão sendo identificados no ambiente. Estes vão sendo marcados no mapa de *bits* pelo Detector de Obstáculos. Pode-se perceber que, ao ser identificado um ponto no ambiente que seja parte de um obstáculo, toda uma região quadrada de lado igual ao diâmetro do robô e centrada no ponto detectado é marcada no mapa como sendo obstáculo. Isto é o que corresponde à expansão dos obstáculos, e que permite que se considere o robô como um único ponto, o seu centro. Se o obstáculo detectado bloquear a trajetória que está sendo seguida pelo robô, então o Detector de Obstáculos invoca o Gerador de Trajetórias para que este escolha um novo caminho, considerando agora este novo obstáculo. A figura 14.12 ilustra a última trajetória gerada: esta consegue ser fornecida completamente ao robô (em simulação), sem que obstáculos sobre ela sejam identificados pelo Detector de Obstáculos.

Será mostrado agora como se comporta o simulador para o caso em que o robô deve parar para fazer a varredura do ambiente. A figura 15, a seguir, ilustra os resultados para o mesmo exemplo ilustrado acima. Nesta figura 15.a, encontra-se ilustrado o ambiente real que está sendo simulado, bem como a posição original e a de destino do robô. No presente caso, inicialmente é feita uma varredura do ambiente, cujo resultado obtido é mostrado em 15.b. Então é gerada uma primeira trajetória com base nestes dados obtidos, traçada em preto na figura 15.c. Esta trajetória vai desde a posição original do robô até a posição desejada. Entretanto, ela será seguida apenas até o ponto em que pode-se garantir não haver obstáculos, ou seja, até a última posição da trajetória gerada a qual pode ser visualizada a partir da posição atual do robô. Ao ser alcançada esta posição, é feita nova varredura do ambiente para de obstáculos não detectáveis a partir da posição anterior. Estes obstáculos são marcados no mapa de *bits* (15.c). Na figura 15.d, o robô movimentou-se mais um pouco e fez nova varredura do ambiente, verificando a não existência de passagem por aquela direção, e planejando portanto uma trajetória pelo outro lado dos obstáculos já detectados. Nas figuras 15.e a 15.l, o robô vai alternadamente gerando trajetórias, seguindo-as parcialmente, e detectando novos obstáculos, até finalmente alcançar a posição de destino.

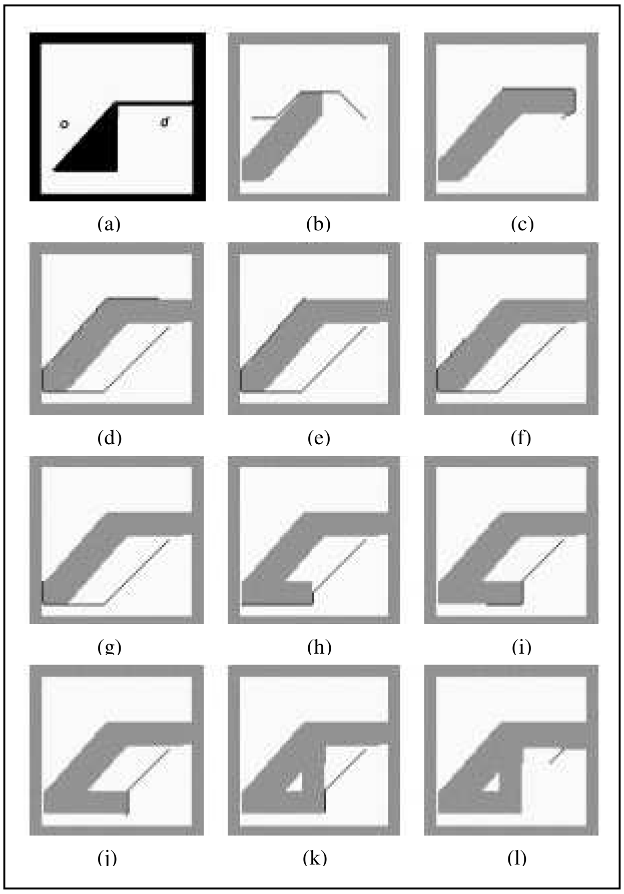


Figura 15: Exemplo de simulação com detecção de obstáculos feita com o robô parado.

Comparando-se este novo simulador com o apresentado anteriormente, observamos que este, apesar de requerer um pouco mais de tempo (pois o robô precisa parar para efetuar leituras do sonar), obteve resultados bastante semelhantes aos anteriores e, portanto, satisfatórios.

IV.5. Implementação

Será visto agora, de forma mais detalhada, a implementação de cada um dos *threads* que compõem a implementação deste projeto. A divisão deste programa em vários *threads* foi feita por alguns motivos principais. Em primeiro lugar, para permitir que diversas tarefas relacionadas com o planejamento, detecção de obstáculos e controle possam ser executadas concorrentemente. Porém, com a alteração que se fez necessária para que as leituras do sonar sejam feitas apenas com o robô parado, acabou ocorrendo que alguns destes *threads* precisam esperar pela execução de outros, não sendo possível que tantos *threads* quantos planejados anteriormente estejam em "execução simultânea". Entretanto, preservou-se a estrutura original dos *threads* até mesmo para facilitar as alterações para a detecção de obstáculos e planejamento de trajetórias simultâneos ao movimento. Uma outra justificativa para a divisão em *threads* é a modularização do programa, permitindo que cada parte dele detenha-se em efetuar um pequeno número de tarefas. Por fim, a biblioteca de *threads* utilizada permite que sejam atribuídas prioridades para cada um dos *threads* de um processo, bem como a escolha de qual algoritmo de escalonamento deve ser utilizado. Estes fatores são importantes para o ajuste do programa para executar em tempo real, particularmente necessário por parte do controle.

Como já foi mencionado, a sincronização entre os *threads* é feita através de semáforos. De uma forma geral, cada *thread* possui um semáforo associado a ele. Quando um *thread* precisa ficar bloqueado a espera de outro, ele executa um *sem_wait* no seu semáforo. Quando um *thread* quer desbloquear aquele, colocando-o em execução, ele deve executar uma chamada *sem_post* tendo como parâmetro o semáforo associado ao *thread* que se deseja desbloquear. Outros semáforos, utilizados com outra finalidades, como executar alguma operação atomicamente, serão explicados juntamente com os *threads* envolvidos.

IV.5.1. Thread *Principal*

Este *thread* é o que começa a ser executado, e é ele quem cria os demais. Além disto, ele é o que cria e inicializa todos os semáforos utilizados para sincronização entre todos os *threads*. Também é ele que finaliza todos os semáforos quando do término da execução do programa. Para que isto seja feito, é preciso garantir que todos os demais *threads* já tenham acabado de executar. O esquema utilizado então para fazer esta sincronização é o seguinte. Ao criar todos os outros *threads*, o Principal efetua uma série de operações *sem_wait* em um semáforo com esta finalidade específica. O número de vez que esta operação é executada é igual ao número de *threads*, sem contar o Principal. Cada um dos outros *threads*, imediatamente antes de terminar sua execução, executa uma operação de *sem_post* neste mesmo semáforo. Assim, quando o Principal for desbloqueado do último *sem_wait*, é garantido que todos os outros *threads* já encerraram sua execução, não necessitando mais dos semáforos. Então o Principal desaloca todos os semáforos e também termina sua execução.

Uma outra tarefa que também é responsabilidade do *thread* Principal é a inicialização do mapa de *bits* do ambiente, que representa o espaço de configurações. Primeiramente este mapa é inicializando com zeros para indicar que todo o ambiente está livre, sem obstáculos. Depois disto, é feita a leitura do arquivo de entrada com extensão *.da*, o qual contém a *d*escrição do *a*mbiente fornecida pelo usuário. Este arquivo descreve cada obstáculo como um polígono, a partir das coordenadas de seus vértices. Cada lado de cada um destes obstáculos é então marcado mapa de *bits* do ambiente. A função utilizada para marcar cada um destes lados utiliza o algoritmo de Bressenham para traçado de retas [10]. Para cada ponto a ser marcado é invocada a função *marca_pto*, também utilizada para marcar os obstáculos identificados a partir do sonar, a qual marca, na verdade, um quadrado com centro localizado no ponto fornecido e lado igual ao diâmetro do robô (método da expansão dos obstáculos).

IV.5.2. Thread *Detector de Obstáculos*

Este *thread*, assim como todos os restantes, tem na sua função principal um laço, do qual só se sai quando um de dois possíveis eventos ocorrer: ou a configuração objetivo é alcançada (indicado pela variável *alcancou_obj*), ou verifica-se que o mesmo não pode ser atingido (indicado pela variável *impossivel_gerar_traj*) porque a configuração inicial e a objetivo não pertencem ao mesmo componente conexo do espaço de configurações (e portanto do grafo associado).

Neste seu laço principal de execução, o Detector de Obstáculos segue a seguinte seqüência normal de passos (assumindo que nenhuma das condições de parada foi alcançada).

1. Desbloqueia o *thread* de leitura do sonar
2. Espera que o *thread* de leitura do sonar execute
3. Marca no mapa os obstáculos identificados pelo *thread* de leitura do sonar
4. Desbloqueia o *thread* de Geração de Trajetórias
5. Põe-se a esperar no semáforo *sem_do* até ser desbloqueado, o que pode ocorrer por dois motivos diferentes:
   - O Gerador de Trajetórias planejou uma trajetória que passa por pontos que localizam-se num ângulo fora da faixa de leitura normal do sonar. Isto porque normalmente não é feita uma varredura de 360° pelo leitor sonar, mas sim numa faixa com amplitude predefinida (estamos utilizando 180°), visando diminuir o tempo de execução.
   - O Gerador de Trajetórias planejou uma trajetória que não caiu no caso anterior, a qual foi fornecida adequadamente ao robô, que por sua vez já encontra-se no final da mesma (não necessariamente na posição objetivada, conforme veremos mais adiante). Neste caso, é atualizada a variável *origem* a qual contém a posição a partir da qual é gerada uma nova trajetória pelo Gerador de Trajetórias.
6. Finalmente, é testado se a posição contida em origem coincide com a posição objetivo. Em caso afirmativo, a variável *alcancou_obj* passa a ter valor lógico verdadeiro.

IV.5.3. Thread *Gerador de Trajetórias*

Este *thread* é o responsável pela geração da trajetória de referência a ser seguida pelo robô, a fim de atingir a configuração objetivo. Cabe salientar que, apesar de ser gerada uma trajetória completa até a configuração objetivada, esta não será necessariamente seguida na sua totalidade. Isto porque nos propusemos a trabalhar com incerteza. De início, nas posições onde não se sabe da existência de obstáculos, assume-se que estes não existem. Desta forma, pode ser gerada uma trajetória que passe em posições que contenham obstáculos, mas que entretanto ainda não podem ser detectados. Por este motivo, a parte da trajetória que será fornecida será apenas aquela localizada a uma distância no máximo igual ao alcance do sonar, a partir do ponto no qual foi feita a leitura deste, e cujo todos os pontos sejam visíveis a partir desta posição (possa ser traçada uma linha reta entre cada ponto e a posição atual, sem que se passe por qualquer ponto marcado no mapa como obstáculo). Ainda, se algum ponto da trajetória assim gerada se localizar fora da faixa na qual foi feita a leitura do sonar, então é solicitada que nova varredura seja feita, agora em 360°, para garantir-se que não há obstáculos nas posições que serão fornecidas ao controlador. O motivo pelo qual são geradas trajetórias completas, até a configuração final, é a obtenção da melhor trajetória (de acordo com os critérios adotados) possível, de posse apenas das informações atualmente conhecidas.

Desta forma, os passos executados normalmente por este *thread* no seu laço principal de execução, são os descritos a seguir. Vale lembrar que o fluxo de execução sairá deste *loop* apenas quando a posição objetivo for alcançada, ou for verificado que não há caminho que leve até ela.

1. Espera no semáforo *sem_gt*. Quem executará o comando para desbloquear o Gerador de Trajetórias será o *thread* Detector de Obstáculos.
2. Geração de uma trajetória, a partir da posição atual até a objetivo, ou verificação da impossibilidade de geração. O método de busca utilizado é busca em largura no espaço livre (representado pelo mapa de *bits*), conforme já mencionado.
    2.1. Se foi possível gerar tal trajetória, então está é podada no ponto em que deixa de ser visível a partir da posição atual (como mencionado acima).
    2.2. Caso contrário, é setada a variável *impossivel_gerar_traj*.
3. Se nenhuma das condições de saída está ativada, e a trajetória gerada possui algum ponto fora da faixa de leitura normal do sonar, e a última leitura não foi feita em 360°, então é indicado na variável *ler_toda_volta* que a próxima leitura do sonar deve ser feita em 360° e o *thread* Detector de Obstáculos é desbloqueado.

   Caso contrário, o *thread* Fornecedor de Trajetória é que é desbloqueado, para que a trajetória planejada seja passada para o controlador.

IV.5.4. Thread *Fornecedor de Trajetórias*

A tarefa principal deste *thread* é disponibilizar a trajetória recentemente escolhida pelo Gerador de Trajetórias para o *thread* controlador. Assim, os passos normalmente executados por este *thread*, dentro de seu laço principal de execução, são:

1. Bloqueia no semáforo *sem_ft* até que o *thread* Gerador de Trajetórias termine de gerar uma nova trajetória
2. Atualiza a trajetória utilizada pelo controlador, armazenada no vetor *traj_ctrl* e com número de elementos *tamtraj_ctrl*. É importante notar que isto deve ser feito atomicamente. Para tanto, utiliza-se o semáforo *sem_traj_ctrl*, que controla o acesso às regiões críticas, efetuando uma exclusão mútua entre este *thread* e o do controlador. Neste momento, também é feita a transformação das coordenadas das posições da trajetória, que são passadas de um sistema de referência global para o sistema de referência do robô (que tem origem situada na posição inicial do robô).
3. Verifica se o *thread* controlador já se encontra em execução. Em caso negativo, coloca-o em execução através de um *sem_post* no semáforo *sem_ctrl*.
4. Põe-se a esperar no semáforo *sem_ft*, até que o controlador utilize completamente a trajetória recentemente fornecida.
5. Espera durante um tempo de acomodação correspondente a um erro menor do que 2%, para garantir que o robô esteja estável. Este tempo é calculado a partir do modelo de referência utilizado pelo controlador.
6. Efetua um *sem_post* no semáforo *sem_do*, indicando para o Detector de Obstáculos que a trajetória já foi fornecida, e que agora deve ser feita nova leitura do ambiente para identificar novos obstáculos.

IV.5.5. Thread *Leitor do Sonar*

Este *thread* foi adaptado de um programa desenvolvido por [14]. Ele efetua uma seqüência de leituras do sonar superior do robô Twil. A diferença angular entre duas leituras consecutivas é de $1{,}8^{\circ}$. Para ser capturada a distância do obstáculo mais próximo em cada direção, é disparado o sonar e então espera-se um tempo correspondente à distância máxima na qual se espera que um obstáculo seja identificado. Colocando-se uma distância muito grande, a espera também será elevada, consumindo um tempo aquém do desejado. Entretanto, se a distância ajustada for muito pequena, o eco proveniente de uma leitura anterior afetará a leitura subseqüente. Tendo-se em mente estas considerações, e através de algumas experimentações, ajustou-se o tempo de espera para um valor que se mostrou adequado. Após serem feitas amostras nas diversas direções, é utilizado um algoritmo para identificar regiões de profundidade constante, para eliminar erros inerentes à natureza deste método de identificação de obstáculos. O resultado deste algoritmo é então passado ao *thread* Detector de Obstáculos, que é quem marcará os obstáculos no mapa de *bits* do ambiente, conforme já foi apresentado.

IV.5.6. Thread *Controlador*

Este *thread* foi adaptado de um programa baseado no controle descrito em [13]. Trata-se de um esquema de controle linearizante por realimentação de estados. Entretanto, no controlador aqui utilizado, leva-se em consideração apenas o equacionamento cinemático do robô para geração do sinal de controle. Este *thread* é executado em tempo real, com um período de 50 ms, o que é implementado através de um *timer*.

Será apresentada agora a forma pela qual a trajetória gerada é fornecida ao controlador ao longo do tempo. Sobre a trajetória, é aplicada um perfil de velocidade constante, cujo valor é também um parâmetro fornecido pelo usuário. Como os pontos da trajetória são relativamente próximos (5 cm nos testes aqui apresentados), não foi necessário fazermos interpolação, sendo sempre fornecido o ponto da trajetória gerada que mais se aproxima da posição na qual o robô deveria estar num determinado instante. Quando a última posição da trajetória que está sendo seguida é alcançada, o *thread* fornecedor de trajetórias é avisado. Este desbloqueará o detector de obstáculos, que novamente solicitará que seja feita uma varredura do ambiente através do sonar, e assim por diante, fechando o ciclo de funcionamento do programa desenvolvido como um todo. Nota-se que até que seja gerada uma nova trajetória, o *thread* controlador permanece fornecendo ao robô a última posição da trajetória recentemente seguida, de forma que ele permaneça parado.

CAPÍTULO V

RESULTADOS EXPERIMENTAIS

Após os testes em simulação serem efetuados, e dos *threads* de interação com os sonares e o controlador serem implementados, o programa desenvolvido foi finalmente testado no robô móvel Twil. Os resultados obtidos foram bastante satisfatórios, sendo que o robô adquiriu assim capacidade de locomover-se autonomamente por entre obstáculos detectados pelo uso de seus sonares, conforme veremos mais adiante em um exemplo de teste prático desenvolvido. Apenas alguns problemas na detecção dos obstáculos foram identificados, como já era esperado, dada a natureza do método utilizado (uso de sonares). Notou-se também que a performance de tempo do modelo como um todo ficou prejudicada, devido à necessidade de o robô interromper seu movimento para efetuar a identificação dos obstáculos através dos sonares.

Segue um exemplo significativo dos testes práticos utilizados, ilustrado na figura 16. Este teste foi efetuado em um dos laboratórios de eletro-eletrônica da FURG. Apesar de relativamente pequeno, este laboratório serviu de base para testes bastante realísticos, pois os obstáculos eram bastante reais, contendo várias irregularidades, o que causa ruído na detecção dos mesmos através do uso de sonares. A figura 16.(a) mostra apenas o contorno real deste laboratório, mas sem pequenas reentrâncias, as quais não podem ser identificadas através de sonares, mas que não são significativas dadas as dimensões do robô. Cada uma das figuras seguintes ilustra, para cada vez que o robô parou para detectar obstáculos, o mapa de bits do ambiente até então identificado e a trajetória completa gerada (mas que não será seguida até o final, conforme já explicado). Neste caso de teste, nenhum obstáculo do ambiente foi informado *a priori* ao robô. Por este motivo, verificamos a eficácia no trabalho desenvolvido neste projeto, dados os resultados satisfatórios ilustrados na figura 16.

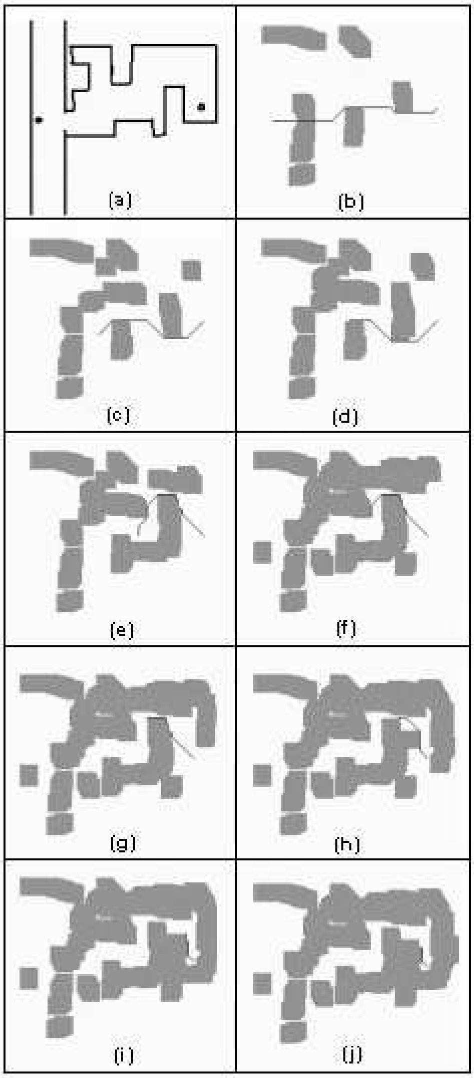


Figura 16: Teste realizado num dos laboratórios de eletro-eletrônica

Finalmente, na figura 17, tem-se um último exemplo de experimento. Este teste foi feito no laboratório de Eletrotécnica do Departamento de Física da FURG. Lembramos que este foi o objetivo estabelecido na proposta inicial deste projeto. Isto porque trata-se de uma sala bastante grande (18 x 10 m) e que possui vários obstáculos. A distância entre as posições inicial e a objetivo era de aproximadamente 13 metros. O resultado obtido foi satisfatório no que tange ao alcance da posição desejada e planejamento da trajetória. Contudo, o tempo que o robô levou para atingir o seu objetivo foi relativamente elevado. Isto deveu-se ao grande número de vezes que ele precisou parar para detectar novos obstáculos (20 vezes). Percebeu-se também em outros testes que, com o método implementado, o robô às vezes não tenta passar em locais pelos quais ele poderia. Isto se deve à imprecisão dos sonares, e também à folga utilizada no raio do robô informado ao programa, necessária como medida de segurança em decorrência do primeiro problema.

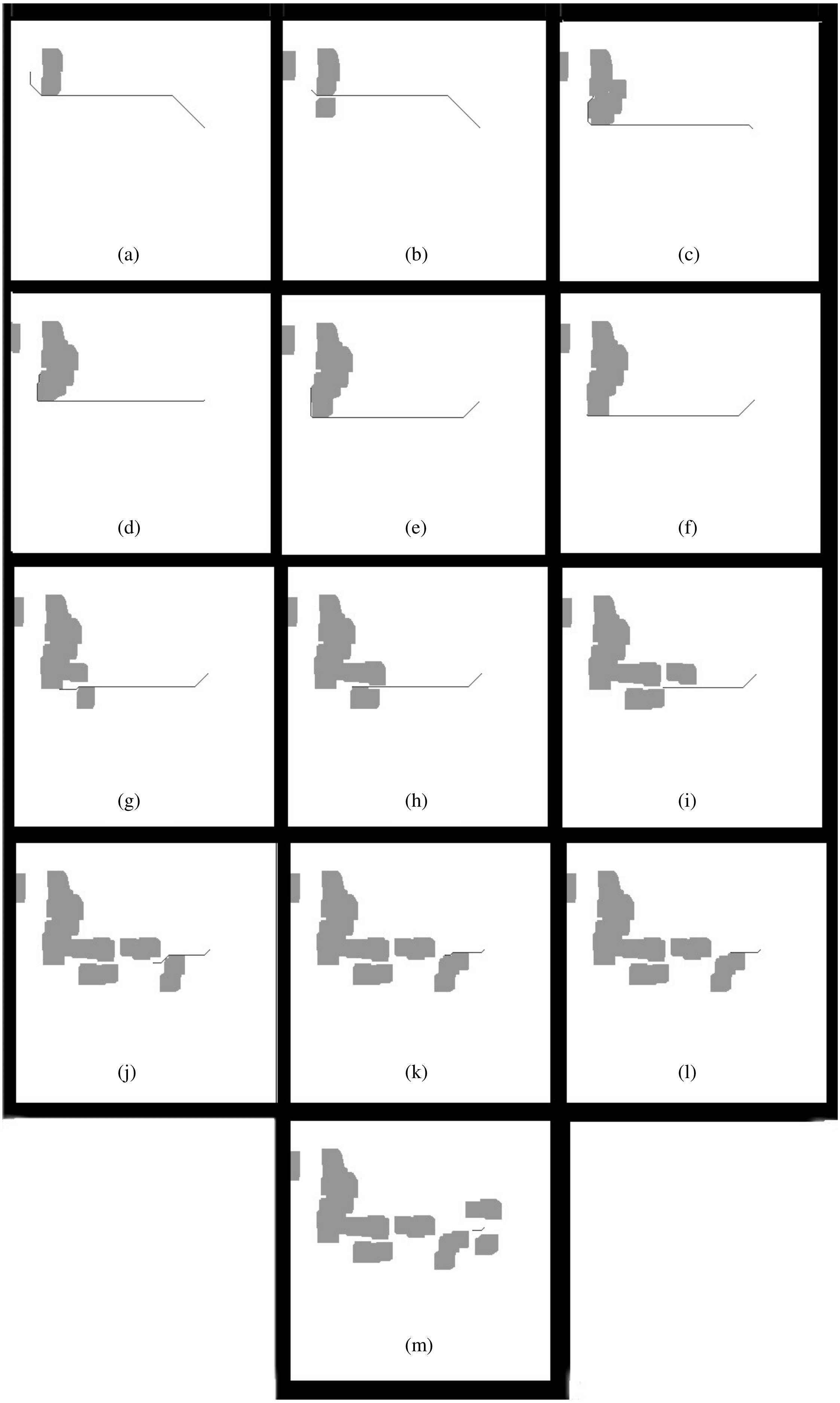


Figura 17: Teste realizado no laboratório de Eletrotécnica do Departamento de Física

CAPÍTULO VI

CONCLUSÃO

No presente trabalho foram estudadas diferentes técnicas para planejamento de trajetórias de robôs. A partir disto, escolheu-se o método que se considerou mais adequado ao problema específico que foi proposto, a saber, o caso em que as informações sobre o ambiente são incompletas, sendo portanto necessário um planejador de trajetórias para funcionar em tempo real. A implementação baseou-se no método de decomposição em células aproximada, e foi desenvolvida em linguagem C sob uma plataforma Linux. Primeiramente desenvolveu-se um simulador para avaliar o método escolhido frente a situações do tipo proposto. Isto foi bastante válido, pois pôde-se fazer alguns testes e ajustes necessários antes de efetivamente aplicar o programa desenvolvido no robô real. Depois disto, foi construído o programa efetivo, fazendo-se então a interface do planejador de trajetórias com os sonares, para detectar os obstáculos, e com o controlador, para fornecimento da trajetória. Conforme os resultados apresentados, tanto de simulação quanto reais, pôde ser verificada a eficiência do sistema desenvolvido. O principal problema encontrado foi referente ao uso dos sonares para detecção dos obstáculos, dada a sua imprecisão inerente. Entretanto, o sistema desenvolvido é suficientemente genérico, de forma que possa ser utilizado juntamente com outro método de detecção de obstáculos.

ANEXO 1

ESTRUTURA DOS ARQUIVOS DE ENTRADA DO PROGRAMA

Os arquivos de entrada são dois, um descrevendo o ambiente real (podendo não conter informação alguma), e outro descrevendo alguns parâmetros, conforme será visto.

Arquivo de Parâmetros

Este arquivo deve conter a seguinte estrutura:

Raio do Robô (em metros)

Velocidade do Robô (em metros por segundo)

Ângulo inicial do robô no sistema de referências global (em graus)

$X_{inicial}$ $Y_{inicial}$ (posição inicial do robô no sistema de referências global; em metros)

X_{final} Y_{final} (posição final do robô no sistema de referências global; em metros)

Arquivo de Descrição do Ambiente

A primeira linha deve conter o número de obstáculos do ambiente descritos no arquivo. A seguir, para cada obstáculo, deve haver uma linha contendo o número de vértices do obstáculo, seguida por uma linha para cada um dos vértices. Os vértices devem ser fornecidos seqüencialmente em sentido horário ou anti-horário, sendo cada vértice descrito pelas suas coordenadas X e Y em metros, respectivamente, no sistema de referências global.

ANEXO 2

CÓDIGO DO PROGRAMA DESENVOLVIDO

```
/*************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  defs.h  - Arquivo de defincoes de constantes,
                      declaracoes de tipos e
                      declaracao de variaveis globais da classe externa


 *************************************************************************/

#ifndef _DEFS_H

#define _DEFS_H

#include <pthread.h>
#include <semaphore.h>

#define GERAR_LOGS
#define GERAR_IMAGENS
// #define MOSTRAR_GRAFICOS

#define PRECISAO        0.05    // em metros

#define LARGURA         15.0  // em metros
#define COMPRIMENTO     15.0  // em metros

#define TETO(X)         ((int)((X)+1.0-1e-5))

#define MESMO_PTOI(P1,P2) (P1.x == P2.x && P1.y == P2.y)

#define MAXX            (TETO(LARGURA     / PRECISAO))
#define MAXY            (TETO(COMPRIMENTO / PRECISAO))

#define VALIDO_PTOI(X,Y)  (X >= 0 && X < maxx && Y >= 0 && Y < maxy)

typedef unsigned char   byte;

typedef struct
        {
          float x,y;
        } ptof_t;

typedef struct
        {
          int x,y;
        } ptoi_t;

// variaveis a serem compartilhadas entre todos os processos

extern byte           mapa[MAXX][MAXY];
extern ptof_t         traj[MAXX*MAXY];      // traj do GT pro FT
extern ptof_t         traj_ctrl[MAXX*MAXY]; // traj do FT pro CTRL
extern uint           tamtraj;       // nro de ptos da trajetoria armazenada em traj
extern uint           tamtraj_ctrl;    // nro de ptos da trajetoria armazenada em
traj_ctrl
extern uint           itraj;         // posic atual; ult posic fornecida ao controlador
```

```
extern ptof_t  origem,         // pto de inicio de cada trajetoria gerada trajetoria
(muda!)
               origem_global,  // pto inicial do robo em todo o processo
               destino;        // pto de destino da trajetoria (objetivo)

extern int impossivel_gerar_traj;
extern int alcancou_obj;
extern int ler_toda_volta;

extern float raio,             // raio do robo
             veloc,
             ang_inic;

extern ptof_t posicao;       // posicao atual do robo; por enq eh atualizada pelo sim
extern float  direcao;       // direcao atual do robo; por enq eh atualizada pelo sim

#define ALCANCE_SONAR                1.5    // 4.0 metros
#define ANG_PASSO_MOTOR_SONAR        1.8    // graus
#define TEMPO_LEITURA_SONAR          0.025  // segundos; tempo necessario para o som ir
e voltar 4m (ALCANCE_SONAR)
#define TEMPO_MOV_PASSO_MOTOR_SONAR  0.001  // segundos
#define AMPLITUDE_LEITURA_NORMAL_SONAR  95.0   // graus
#define DIST_CENTRO_SONAR            0.20    // metros

typedef struct
{
  ptof_t pos;
  float ang,dist;
} dados_sonar_t;

extern dados_sonar_t list_obst[210]; // uma folguinha nao faz mal pra ninguem :)
extern int tam_list_obst;

extern char nome_arq_amb_real[30];   /* nome do arquivo que contem o ambiente real,
                                 para fins de simulacao */

extern unsigned char fig[MAXY][MAXX];   /* para gerar arqgif */

extern int maxx;
extern int maxy;

extern pthread_t thread_gt, thread_ft, thread_do, thread_sonar, thread_ctrl;

extern sem_t sem_gt, sem_ft, sem_do, sem_sonar, sem_princ, sem_ctrl, sem_traj_ctrl;

#define OBSTACULO       127     // valor colocado nos bytes correspondentes a obstaculos
#define MARC_MASC       0x80    // mascara para indicar que pto jah foi visitado na
busca
#define DESMARC_MASC    0x7F    // mascara para desmarcar os ptos do mapa

#define INFINITO      1000.0


#endif
```

```
/***********************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  var.h  - Arquivo de declaracao das variaveis globais. Incluido pelo
           arquivo celldec.c


 ***********************************************************************/

#include "defs.h"
#include <pthread.h>
#include <semaphore.h>

// variaveis a serem compartilhadas entre todos os processos

byte           mapa[MAXX][MAXY];
ptof_t         traj[MAXX*MAXY];      // traj do GT pro FT
ptof_t         traj_ctrl[MAXX*MAXY]; // traj do FT pro CTRL
uint           tamtraj;       // nro de ptos da trajetoria armazenada em traj
uint           tamtraj_ctrl;   // nro de ptos da trajetoria armazenada em traj_ctrl
uint           itraj;         // posic atual; ult posic fornecida ao controlador
ptof_t  origem,         // pto de inicio de cada trajetoria gerada trajetoria (muda!)
        origem_global,  // pto inicial do robo em todo o processo
        destino;        // pto de destino da trajetoria (objetivo)

int impossivel_gerar_traj;
int alcancou_obj;
int ler_toda_volta;

float raio,             // raio do robo
      veloc,
      ang_inic;

ptof_t posicao;              // posicao atual do robo; por enq eh atualizada pelo sim
float  direcao;              // direcao atual do robo; por enq eh atualizada pelo sim

dados_sonar_t list_obst[210]; // uma folguinha nao faz mal pra ninguem :)
int tam_list_obst;

char nome_arq_amb_real[30];   /* nome do arquivo que contem o ambiente real,
                                 para fins de simulacao */

unsigned char fig[MAXY][MAXX];   /* para gerar arqgif */

int maxx = MAXX;
int maxy = MAXY;

pthread_t thread_gt, thread_ft, thread_do, thread_sonar, thread_ctrl;

sem_t sem_gt, sem_ft, sem_do, sem_sonar, sem_princ, sem_ctrl, sem_traj_ctrl;
```

```
/***************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao

               Planejamento de Trajetorias para Robos Moveis
               ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages


  geometria.h  - Arquivo de definicoes de tipos e constantes,
               e prototipos de funcoes geometricas

 ***************************************************************************/
#ifndef __GEOMETRIA_H
#define __GEOMETRIA_H

#define F_IGUAL(X,Y)      (fabs((X)-(Y)) < 1e-3)
#define F_MENORIGUAL(X,Y) (((X) < (Y)) || F_IGUAL(X,Y))
#define F_MAIORIGUAL(X,Y) (((X) > (Y)) || F_IGUAL(X,Y))
#define PTO_IGUAL(P1,P2)  (F_IGUAL((P1).x,(P2).x) && F_IGUAL((P1).y,(P2).y))

#define SQR(X)        ((X)*(X))
#define DIST(P1,P2) (sqrt( SQR((P1).x - (P2).x) + SQR((P1).y - (P2).y) ))

#define GRAUS_PARA_RAD(X) ((X)*M_PI/180.0)
#define RAD_PARA_GRAUS(X) ((X)*180.0/M_PI)

// codigos de retorno da func intersec_retas(...)
#define CONCORRENTES 0
#define PARALELAS    -1
#define MESMARETA    -2

typedef struct PTO                    // ED para armazenar um ponto no plano
{
  float x,y;
} Pto;

typedef Pto Vetor;                // a ED de um vetor eh a mesma de um ponto

extern Vetor vetor(Pto orig, Pto dest);

extern float modulo(Vetor v);

extern Vetor mult_vet_esc(Vetor v, float e);

extern Vetor soma_vet(Vetor v1, Vetor v2);

extern float prod_esc(Vetor v1, Vetor v2);

extern float prod_vet(Vetor v1, Vetor v2);

extern float angulo(Vetor v1, Vetor v2); // retorna o angulo em radianos entre os
vetores

extern Vetor rot90(Vetor v);         // rotaciona v em 90 graus

extern Vetor rot_90(Vetor v);        // rotaciona v em -90 graus

extern Vetor rot(Vetor r, float ang);        // rotaciona v em ang radianos

extern int pert_pto_reta(Pto a, Pto b, Pto c); // verifica se os 3 ptos sao colineares

extern int pert_pto_seg(Pto a, Pto b, Pto c);

extern int intersec_retas(Pto p1, Pto p2, Pto q1, Pto q2, Pto * pi);
                                  // *pi: pto de interseccao, se concorrentes

extern int intersec_segmtos(Pto a, Pto b, Pto p, Pto q, Pto * pi1, Pto * pi2);
// retorna o nro de parametros de retorno utilizados (pi1, pi2)
// se concorrentes -> retorna em pi1 o pto comum
// se mais de um pto em comum -> retorna em pi1 e pi2 os extremos do segmento comum

#endif
```

```
/*************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  celldec.c  - Arquivo Principal (Metodo Cell Decomposition)


 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <vga.h>
#include <unistd.h>
#include "defs.h"
#include "var.h"
#include "geometria.h"

#include <pthread.h>
#include <semaphore.h>

extern void detect_obst(void);
extern void ft(void);
extern void gt(void);
extern void le_sonar(void);
extern void controlador(void);

void sai_sintaxe()
{
  printf("Sintaxe:\n\tcelldec -p <arq_param> -r <arq_amb_real> [ -a <arq_descr_amb> | -s
<arq_amb_salvo> ]\n\n");
  exit(1);
}

void erro_sintaxe()
{
  printf("\nErro de sintaxe.\n");
  sai_sintaxe();
}

void erro(char * msgerro)
{
  printf("\n%s\n",msgerro);
  exit(1);
}

void erro_abrir_arq(char * nomearq)
{
  printf("\nErro ao abrir arquivo %s\n",nomearq);
  sai_sintaxe();
}

void carrega_parametros(char * nomearq)
{
  FILE * arqpar;
  if ((arqpar = fopen(nomearq,"rt")) == NULL)
    erro_abrir_arq(nomearq);
  fscanf(arqpar,"%f %f %f %f %f %f
%f",&raio,&veloc,&ang_inic,&(origem.x),&(origem.y),&(destino.x),&(destino.y));
  origem_global = origem;
  direcao = ang_inic;
  fclose(arqpar);
}

ptoi_t pto_mapa(ptof_t ptof)
```

```
{
  ptoi_t ptoi;
  ptoi.x = (uint)(ptof.x/PRECISAO);
  ptoi.y = (uint)(ptof.y/PRECISAO);
  return(ptoi);
}

void marca_pto(ptoi_t p)
{
  int raioi=((int)ceil(raio/PRECISAO));

  int x,y;
  int xi = p.x-raioi >= 0 ? p.x-raioi : 0;
  int xf = p.x+raioi < maxx ? p.x+raioi : maxx-1;
  int yi = p.y-raioi >= 0 ? p.y-raioi : 0;
  int yf = p.y+raioi < maxy ? p.y+raioi : maxy-1;
  for (x=xi; x <= xf; x++)
    for (y=yi; y <= yf; y++)
      mapa[x][y] = OBSTACULO;
}

void marca_linhaX(ptoi_t p1, ptoi_t p2)
{
  uint passo=(int)ceil(raio/PRECISAO),cont=passo;
  int s=0;
  int dx = p2.x - p1.x;
  int dy = p2.y - p1.y;
  int yincr = dy > 0? 1 : -1;
  int dxdiv2 = dx >> 1;
  ptoi_t p = p1;
  dy = abs(dy);
  marca_pto(p);
  while (p.x < p2.x)
  {
    p.x++;
    s += dy;
    if (s > dxdiv2)
    {
      s -= dx;
      p.y += yincr;
    }
    if (cont == passo)
    {
      cont=1;
      marca_pto(p);
    }
    else
      cont++;
  }
  if (cont > 1)  /* se ultimo ponto marcado nao foi o final entao marca ele */
    marca_pto(p);
}

void marca_linhaY(ptoi_t p1, ptoi_t p2)
{
  int s=0;
  int dx = p2.x - p1.x;
  int dy = p2.y - p1.y;
  int xincr = dx > 0? 1 : -1;
  int dydiv2 = dy >> 1;
  ptoi_t p = p1;
  dx = abs(dx);
  marca_pto(p);
  while (p.y < p2.y)
  {
    p.y++;
    s += dx;
    if (s > dydiv2)
    {
      s -= dy;
      p.x += xincr;
    }
    marca_pto(p);
  }
}

void marca_linha(ptoi_t p1, ptoi_t p2)
```

```
{
  int dx = abs(p2.x - p1.x);
  int dy = abs(p2.y - p1.y);

  if (dx > dy)
    if (p1.x < p2.x)
      marca_linhaX(p1,p2);
    else
      marca_linhaX(p2,p1);
  else
    if (p1.y < p2.y)
      marca_linhaY(p1,p2);
    else
      marca_linhaY(p2,p1);
}

void carrega_descr_amb(char * nomearq)
{
  int nobsts,o,nverts,v;
  ptof_t pf,pinicf;
  ptoi_t pi,pinici,panti;
  FILE * arqda;
  if ((arqda = fopen(nomearq,"rt")) == NULL)
    erro_abrir_arq(nomearq);
  fscanf(arqda,"%d",&nobsts);
  for (o=0; o < nobsts; o++)
  {
    fscanf(arqda,"%d %f %f",&nverts,&pinicf.x,&pinicf.y);
    panti = pinici = pto_mapa(pinicf);
    for (v=1; v < nverts; v++, panti=pi)
    {
      fscanf(arqda,"%f %f",&pf.x,&pf.y);
      pi = pto_mapa(pf);
      marca_linha(panti,pi);
    }
    marca_linha(pi,pinici);
  }
  fclose(arqda);
}

void carrega_amb_salvo(char * nomearq)
{
  FILE * arqas;
  if ((arqas = fopen(nomearq,"rb")) == NULL)
    erro_abrir_arq(nomearq);
  fread(mapa,sizeof(mapa),1,arqas);
  fclose(arqas);
}

int main(int argc, char * argv[])
{
  pthread_attr_t do_attr, gt_attr, ft_attr, ctrl_attr, sonar_attr;
  struct sched_param do_sched_param, gt_sched_param, ft_sched_param, ctrl_sched_param,
sonar_sched_param;

  int descr_amb=0, amb_salvo=0, i;
  if (argc == 1)
    sai_sintaxe();
  if (argc < 3)
    erro_sintaxe();
  memset(mapa,0,sizeof(mapa));
  itraj = 0;
  nome_arq_amb_real[0] = '\0';
  for (i=1; i < argc; i+=2)
  {
    if (i == argc-1)
      erro_sintaxe();
    if (strcmp(argv[i],"-p") == 0)
      carrega_parametros(argv[i+1]);
    else
      if (strcmp(argv[i],"-a") == 0)
      {
        if (descr_amb || amb_salvo)
          erro_sintaxe();
        descr_amb = 1;
        carrega_descr_amb(argv[i+1]);
      }
```

```
    else
    {
      if (strcmp(argv[i],"-s") == 0)
      {
        if (descr_amb || amb_salvo)
          erro_sintaxe();
        amb_salvo = 1;
        carrega_amb_salvo(argv[i+1]);
      }
      else
        if (strcmp(argv[i],"-r") == 0)
          strcpy(nome_arq_amb_real,argv[i+1]);
        else
          erro_sintaxe();
    }
}
setvbuf(stdout,NULL,_IONBF,0);

impossivel_gerar_traj = 0;
alcancou_obj = 0;
ler_toda_volta = 0;

printf("Vou inic os semaforos\n");
sem_init(&sem_princ,0,0);
sem_init(&sem_gt,0,0);
sem_init(&sem_ft,0,0);
sem_init(&sem_do,0,0);
sem_init(&sem_sonar,0,0);
sem_init(&sem_ctrl,0,0);
sem_init(&sem_traj_ctrl,0,1);
printf("Vou criar os threads\n");

pthread_attr_init(&do_attr);
pthread_attr_setschedpolicy(&do_attr,SCHED_RR);
pthread_attr_setdetachstate(&do_attr,PTHREAD_CREATE_DETACHED);
do_sched_param.sched_priority=4;
pthread_attr_setschedparam(&do_attr,&do_sched_param);

pthread_attr_init(&gt_attr);
pthread_attr_setschedpolicy(&gt_attr,SCHED_RR);
pthread_attr_setdetachstate(&gt_attr,PTHREAD_CREATE_DETACHED);
gt_sched_param.sched_priority=4;
pthread_attr_setschedparam(&gt_attr,&gt_sched_param);

pthread_attr_init(&ft_attr);
pthread_attr_setschedpolicy(&ft_attr,SCHED_RR);
pthread_attr_setdetachstate(&ft_attr,PTHREAD_CREATE_DETACHED);
ft_sched_param.sched_priority=4;
pthread_attr_setschedparam(&ft_attr,&ft_sched_param);

pthread_attr_init(&ctrl_attr);
pthread_attr_setschedpolicy(&ctrl_attr,SCHED_RR);
pthread_attr_setdetachstate(&ctrl_attr,PTHREAD_CREATE_DETACHED);
ctrl_sched_param.sched_priority=0;
pthread_attr_setschedparam(&ctrl_attr,&ctrl_sched_param);

if (pthread_create(&thread_gt,&gt_attr,(void *(*)(void *))gt,NULL) != 0)
  printf("Erro criando thread 1\n");
if (pthread_create(&thread_ft,&ft_attr,(void *(*)(void *))ft,NULL) != 0)
  printf("Erro criando thread 2\n");
if (pthread_create(&thread_do,&do_attr,(void *(*)(void *))detect_obst,NULL) != 0)
  printf("Erro criando thread 3\n");
if (pthread_create(&thread_sonar,&sonar_attr,(void *(*)(void *))le_sonar,NULL) != 0)
  printf("Erro criando thread 4\n");
if (pthread_create(&thread_ctrl,&ctrl_attr,(void *(*)(void *))controlador,NULL) != 0)
  printf("Erro criando thread 5\n");
printf("Vou esperar pelos UPs\n");
sem_wait(&sem_princ);
printf("Recebi um UP\n");
sem_wait(&sem_princ);
printf("Recebi um UP\n");
sem_wait(&sem_princ);
printf("Recebi um UP\n");
sem_wait(&sem_princ);
printf("Recebi um UP\n");
sem_wait(&sem_princ);
printf("Recebi um UP\n");
```

```
  printf("Principal terminado!\n");
  sem_destroy(&sem_princ);
  sem_destroy(&sem_gt);
  sem_destroy(&sem_ft);
  sem_destroy(&sem_do);
  sem_destroy(&sem_sonar);
  sem_destroy(&sem_ctrl);
  sem_destroy(&sem_traj_ctrl);
  return(0);
}
```

```
/***************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  gt.c  - Arquivo do Gerador de Trajetorias


 **************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <vga.h>
#include <unistd.h>
#include "defs.h"
#include "geometria.h"

#include <pthread.h>
#include <semaphore.h>

extern ptoi_t pto_mapa(ptof_t ptof);

FILE * gtlog;

/******************* ROTINAS GRAFICAS BASICAS **********************/

void inicializa_modografico()
{
#ifdef MOSTRAR_GRAFICOS
  fprintf(gtlog,"Vou inic o modo grafico\n");
  if (vga_init() != 0)
  {
    printf("Erro no vga_init()\n");
    exit(1);
  }
  vga_setmode(G640x480x16);
  fprintf(gtlog,"Inicializei o modo grafico\n");
#endif
}

void finaliza_modografico()
{
#ifdef MOSTRAR_GRAFICOS
  vga_setmode(TEXT);
#endif
}

/*****************************************************************/

void desmarca_mapa()
{
  int x,y;
  for (x=0; x < maxx; x++)
    for (y=0; y < maxy; y++)
      mapa[x][y] &= DESMARC_MASC;
}

int gera_trajetoria()
{
  struct
  {
    ptoi_t p;
    int iant;
  } fila[MAXX*MAXY];
  uint ifila=0,ffila=1,i;
  int x,y;
```

```
  int dx[4] = {0,1,0,-1};
  int dy[4] = {1,0,-1,0};
  ptoi_t desti = pto_mapa(destino);
  ptof_t auxptof;

  fprintf(gtlog,"Vou gerar uma trajetoria: (%0.2f,%0.2f) ->
(%0.2f,%0.2f)\n",origem.x,origem.y,destino.x,destino.y);
  fila[0].p = pto_mapa(origem);
  fprintf(gtlog,"fila[0]: (%d,%d)     desti:
(%d,%d)\n",fila[0].p.x,fila[0].p.y,desti.x,desti.y);
  fila[0].iant = -1;
  mapa[fila[0].p.x][fila[0].p.y] = 0;
  while (!MESMO_PTOI(fila[ifila].p,desti) && ifila < ffila)
  {
/*    fprintf(gtlog,"Prim da Fila: (%d,%d)\n",fila[ifila].p.x,fila[ifila].p.y); */
    for (i=0; i < 4; i++)
    {
      x = fila[ifila].p.x+dx[i];
      y = fila[ifila].p.y+dy[i];
      if (VALIDO_PTOI(x,y) && (mapa[x][y] == 0))
      {
        mapa[x][y] |= MARC_MASC;
        fila[ffila].p.x = x;
        fila[ffila].p.y = y;
        fila[ffila].iant = ifila;
        ffila++;
      }
    }
    for (i=0; i < 4; i++)
    {
      x = fila[ifila].p.x+dx[i]+dx[(i+1)%4];
      y = fila[ifila].p.y+dy[i]+dy[(i+1)%4];
      if (VALIDO_PTOI(x,y) && (mapa[x][y] == 0))
      {
        mapa[x][y] |= MARC_MASC;
        fila[ffila].p.x = x;
        fila[ffila].p.y = y;
        fila[ffila].iant = ifila;
        ffila++;
      }
    }
    ifila++;
  }
  if (MESMO_PTOI(fila[ifila].p,desti))  /* encontrou um caminho...*/
  {
    fprintf(gtlog,"Encontrei caminho!!!\nTrajetoria invertida:
(%d,%d)\n",fila[ifila].p.x,fila[ifila].p.y);
    tamtraj = 1;
    i = ifila;
    traj[0].x = fila[i].p.x*PRECISAO + PRECISAO/2;
    traj[0].y = fila[i].p.y*PRECISAO + PRECISAO/2;
    while (fila[i].iant != -1)
    {
      i = fila[i].iant;
      traj[tamtraj].x = fila[i].p.x*PRECISAO + PRECISAO/2;
      traj[tamtraj].y = fila[i].p.y*PRECISAO + PRECISAO/2;
      tamtraj++;
      fprintf(gtlog,"(%d,%d)\n",fila[i].p.x,fila[i].p.y);
    }
    for (i=0; i < (tamtraj)/2; i++)
    {
      auxptof = traj[i];
      traj[i] = traj[(tamtraj)-i-1];
      traj[(tamtraj)-i-1] = auxptof;
    }
  }
  else
  {
    printf("\n\n\n\n\n\n\nNao ha' caminho unindo origem e destino!\n");
    return(0);
  }
  fprintf(gtlog,"GeraTrajet: Terminei. Vou desmarcar o mapa\n");
  desmarca_mapa();
  return(1);
}

void mostra_ambiente()
```

```
{
  int x,y;
  #ifdef MOSTRAR_GRAFICOS
    vga_setcolor(7);
  #endif
  for (y=0; y < maxy; y++)
    for (x=0; x < maxx; x++)
    {
      if (mapa[x][y] == OBSTACULO)
      {
        #ifdef MOSTRAR_GRAFICOS
          vga_setcolor(7);
        #endif
        fig[y][x] = 150;
      }
      else
      {
        #ifdef MOSTRAR_GRAFICOS
          vga_setcolor(0);
        #endif
      }
      #ifdef MOSTRAR_GRAFICOS
        vga_drawpixel(100+x,100+y);
      #endif
    }
}

#define COR_TRAJETORIA  15

void mostra_origdest()
{
  ptoi_t ptoi;
  ptoi = pto_mapa(origem);
#ifdef MOSTRAR_GRAFICOS
  vga_setcolor(11);
  vga_drawpixel(100+ptoi.x,100+ptoi.y);
  vga_drawline(100+ptoi.x-5,100+ptoi.y-5,100+ptoi.x+5,100+ptoi.y+5);
  vga_drawline(100+ptoi.x-5,100+ptoi.y+5,100+ptoi.x+5,100+ptoi.y-5);
#endif
  fig[ptoi.y][ptoi.x] = 0;
  ptoi = pto_mapa(destino);
  fig[ptoi.y][ptoi.x] = 20;
#ifdef MOSTRAR_GRAFICOS
  vga_setcolor(13);
  vga_drawpixel(100+ptoi.x,100+ptoi.y);
  vga_drawline(100+ptoi.x-5,100+ptoi.y-5,100+ptoi.x+5,100+ptoi.y+5);
  vga_drawline(100+ptoi.x-5,100+ptoi.y+5,100+ptoi.x+5,100+ptoi.y-5);
#endif
}

void mostra_trajetoria()
{
  int i;
  ptoi_t ptoi;
#ifdef MOSTRAR_GRAFICOS
  vga_setcolor(COR_TRAJETORIA);
#endif
  for (i=0; i < tamtraj; i++)
  {
    ptoi = pto_mapa(traj[i]);
#ifdef MOSTRAR_GRAFICOS
    vga_drawpixel(100+ptoi.x,100+ptoi.y);
#endif
    fig[ptoi.y][ptoi.x] = 50;
  }
}


int visivel_linhaX(ptoi_t p1, ptoi_t p2)
{
  uint passo=(int)ceil(raio/PRECISAO),cont=passo;
  int s=0;
  int dx = p2.x - p1.x;
  int dy = p2.y - p1.y;
  int yincr = dy > 0? 1 : -1;
  int dxdiv2 = dx >> 1;
  ptoi_t p = p1;
```

```
  dy = abs(dy);
  if (mapa[p.x][p.y] == OBSTACULO)
    return(0);
  while (p.x < p2.x)
  {
    p.x++;
    s += dy;
    if (s > dxdiv2)
    {
      s -= dx;
      p.y += yincr;
    }
    if (cont == passo)
    {
      cont=1;
      if (mapa[p.x][p.y] == OBSTACULO)
        return(0);
    }
    else
      cont++;
  }
  if (cont > 1)  /* se ultimo ponto marcado nao foi o final entao marca ele */
    if (mapa[p.x][p.y] == OBSTACULO)
      return(0);
  return(1);
}

int visivel_linhaY(ptoi_t p1, ptoi_t p2)
{
  int s=0;
  int dx = p2.x - p1.x;
  int dy = p2.y - p1.y;
  int xincr = dx > 0? 1 : -1;
  int dydiv2 = dy >> 1;
  ptoi_t p = p1;
  dx = abs(dx);
  if (mapa[p.x][p.y] == OBSTACULO)
    return(0);
  while (p.y < p2.y)
  {
    p.y++;
    s += dx;
    if (s > dydiv2)
    {
      s -= dy;
      p.x += xincr;
    }
    if (mapa[p.x][p.y] == OBSTACULO)
      return(0);
  }
}

int visivel_linha(ptoi_t p1, ptoi_t p2)
{
  int dx = abs(p2.x - p1.x);
  int dy = abs(p2.y - p1.y);
  if (dx > dy)
    if (p1.x < p2.x)
      return(visivel_linhaX(p1,p2));
    else
      return(visivel_linhaX(p2,p1));
  else
    if (p1.y < p2.y)
      return(visivel_linhaY(p1,p2));
    else
      return(visivel_linhaY(p2,p1));
}

void gt_termina()
{
  fclose(gtlog);
  sem_post(&sem_princ);
}

void _vga_getch(void)
{
#ifdef MOSTRAR_GRAFICOS
```

```
  vga_getch();
#endif
}

float transf_ang(float ang)
// transforma um ang de 0..2*PI para -PI..PI
{
  return(ang <= M_PI? ang : ang - 2*M_PI);
}

int traj_fora_faixa_leit()
{
  int i;
  Vetor vet_dir;
  Pto _posic,_ptotraj;

  _posic.x = posicao.x;
  _posic.y = posicao.y;
  vet_dir.x = cos(GRAUS_PARA_RAD(direcao));
  vet_dir.y = sin(GRAUS_PARA_RAD(direcao));
  for (i=1; i < tamtraj; i++)
  {
    _ptotraj.x = traj[i].x;
    _ptotraj.y = traj[i].y;
    if (fabs(transf_ang(angulo(vet_dir,vetor(_posic,_ptotraj)))) >
GRAUS_PARA_RAD(AMPLITUDE_LEITURA_NORMAL_SONAR))
    {
      fprintf(gtlog,"ACHOU TRAJ FORA DA FAIXA: posicao = (%.2f,%.2f)  direcao = %.2f
graus\n",posicao.x,posicao.y,direcao);
      fprintf(gtlog,"   Ponto = (%.2f,%.2f)\n",_ptotraj.x,_ptotraj.y);
      fprintf(gtlog," vet_dir = (%.2f,%.2f)   angulo = %.2f rad   ang_lim =
%.2f\n\n",vet_dir.x,vet_dir.y,fabs(angulo(vet_dir,vetor(_posic,_ptotraj))),GRAUS_PARA_RA
D(AMPLITUDE_LEITURA_NORMAL_SONAR));
      return(1);
    }
  }
  return(0);
}

void gt(void)  /* gerador de trajetorias */
{
  int x,y;
  int gerou_traj;
  int cont, contptos;
  FILE * arqfig;
  char nomearq[30], s[30];
  unsigned char rmap[256], gmap[256], bmap[256];
  ptoi_t orig, dest;

  for (cont = 0; cont < 256; cont++)
  {
    rmap[cont] = gmap[cont] = bmap[cont] = cont;
  }
  cont = 1;

#ifdef  GERAR_LOGS
  gtlog = fopen("gt.log","wt");
#else
  gtlog = fopen("/dev/null","wt");
#endif
  setvbuf(gtlog,NULL,_IONBF,0);

  inicializa_modografico();

  fprintf(gtlog,"maxx=%d   maxy=%d\n",maxx,maxy);
  while (1)
  {
    sem_wait(&sem_gt);
    fprintf(gtlog,"ORIGEM = (%.2f,%.2f)  DESTINO =
(%0.2f,%0.2f)\n",origem.x,origem.y,destino.x,destino.y);

     /* mostra mapa*/
        orig = pto_mapa(origem);
        dest = pto_mapa(destino);
        for (y=0; y < maxy; y++)
        {
          for (x=0; x < maxx; x++)
```

```
          if (x == orig.x && y == orig.y)
            if (mapa[x][y] == OBSTACULO)
              fprintf(gtlog,"M");
            else
              fprintf(gtlog,"O");
          else
            if (x == dest.x && y == dest.y)
              fprintf(gtlog,"D");
            else
              fprintf(gtlog,"%c",mapa[x][y] == OBSTACULO? 'X' : ' ');
        fprintf(gtlog,"\n");
      }
    if (!alcancou_obj)
    {
      gerou_traj = gera_trajetoria();
        #ifdef GERAR_IMAGENS
          strcpy(nomearq,"fig");
          sprintf(s,"%d",cont++);
          strcat(nomearq,s);
          strcat(nomearq,".gif");
          arqfig = fopen(nomearq,"wb");
          memset(fig,255,sizeof(fig));
        #else
          arqfig = fopen("/dev/null","wb");
        #endif
        mostra_ambiente();
        _vga_getch();
        mostra_trajetoria();
        mostra_origdest();
        _vga_getch();
        writegif(arqfig,fig,maxx,maxy,rmap,gmap,bmap,256,1,"");
        fclose(arqfig);

      if (gerou_traj)
      {
        for (contptos=0; (contptos < ALCANCE_SONAR/(1.4142*PRECISAO)) && (contptos <
tamtraj); contptos++)
        {
          if (!visivel_linha(pto_mapa(traj[0]),pto_mapa(traj[contptos])))  // verifica
se traj[cont] eh visivel a partir de traj[0]
            break;
        }
        tamtraj = contptos;
        fprintf(gtlog,"NOVO TAMTRAJ = %d\n",tamtraj);
        if (tamtraj < 2)
        {
          fprintf(gtlog,"Deu merda no visivel_linha!  tamtraj = %d\n",tamtraj);
        }
      }
      else
      {
        impossivel_gerar_traj = 1;
        fprintf(gtlog,"Nao consegui gerar uma trajetoria\n");
        mostra_ambiente();
        _vga_getch();
        fprintf(gtlog,"Vou mostrar orig e dest\n");
        mostra_origdest();
        _vga_getch();
      }
    }
    if (!alcancou_obj && !impossivel_gerar_traj && traj_fora_faixa_leit() &&
!ler_toda_volta)
    {
      fprintf(gtlog,"Eh preciso ler toda a volta!\n");
      ler_toda_volta = 1;
      sem_post(&sem_do);
    }
    else  // soh se nao precisar fazer leitura total do sonar
    {
      ler_toda_volta = 0;
      fprintf(gtlog,"\n ENVIANDO UP PRO FT!!!\n");
      sem_post(&sem_ft);
    }
    if (alcancou_obj)
      break;
    if (impossivel_gerar_traj)
      break;
```

```
  }
  fprintf(gtlog,"Sai do laco\n");
  if (alcancou_obj)
    fprintf(gtlog,"   Com sucesso\n");
  else
    fprintf(gtlog,"   Sem sucesso\n");
#ifdef MOSTRAR_GRAFICOS
  fprintf(gtlog,"Esperando getch para sair\n");
  _vga_getch();
  finaliza_modografico();
#endif
  gt_termina();
}
```

```
/***********************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


               Planejamento de Trajetorias para Robos Moveis
               ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  do.c  - Arquivo do Detector de Obstaculos


 ***********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <vga.h>
#include <unistd.h>
#include "defs.h"
#include "geometria.h"

#include <pthread.h>
#include <semaphore.h>

extern ptoi_t pto_mapa(ptof_t ptof);

FILE * dolog;

void do_termina()
{
  fprintf(dolog,"DO: Vou terminar\n");
  fclose(dolog);
  sem_post(&sem_princ);
}

void detect_obst()    /* detector de obstaculos */
{
  ptof_t pf;
  ptoi_t pi, pt, pdest=pto_mapa(destino);
  int i, o;
  int x,y;
  int prim_vez=1;

#ifdef GERAR_LOGS
  dolog = fopen("do.log","wt");
#else
  dolog = fopen("/dev/null","wt");
#endif
  setvbuf(dolog,NULL,_IONBF,0);

  fprintf(dolog,"DO: Vou entrar no loop\n");

  while (1)
  {
    sem_post(&sem_sonar);
    fprintf(dolog,"DO: Acordei o SIM\n");
    if (impossivel_gerar_traj)
      break;
    if (!alcancou_obj)
    {
      fprintf(dolog,"DO: Vou esperar pelo SIM\n");
      sem_wait(&sem_do);  // espera pelo SIM
      fprintf(dolog,"DO: Acordei\n");
      for (o=0; o < tam_list_obst; o++)
      {
     fprintf(dolog,"pos = (%.2f,%.2f)   ang = %.2f  dist =
%.2f\n",list_obst[o].pos.x,list_obst[o].pos.y,list_obst[o].ang,list_obst[o].dist);
        pi = pto_mapa(list_obst[o].pos);
        if (list_obst[o].dist < INFINITO)
        {
```

```
          fprintf(dolog,"Detectou Obstaculo\n");
          pf.x = list_obst[o].pos.x +
list_obst[o].dist*cos(GRAUS_PARA_RAD(list_obst[o].ang));
          pf.y = list_obst[o].pos.y +
list_obst[o].dist*sin(GRAUS_PARA_RAD(list_obst[o].ang));
          fprintf(dolog,"\nSonar enviou detectou obstaculo (%.4f,%.4f)\n",pf.x,pf.y);
          pi = pto_mapa(pf);
          marca_pto(pi);

        }
      }
    /* mostra mapa*/
      for (y=0; y < maxy; y++)
      {
        for (x=0; x < maxx; x++)
          fprintf(dolog,"%c",mapa[x][y] == OBSTACULO? 'X' : ' ');
        fprintf(dolog,"\n");
      }
    }
    fprintf(dolog,"Vou acordar o GT\n");
    sem_post(&sem_gt);
    if (alcancou_obj)
      break;
    fprintf(dolog,"Vou esperar pelo FT ou pelo GT\n");
    sem_wait(&sem_do);  // espera ft ou gt (q pode pedir para ler_toda_volta)
    if (!ler_toda_volta)  // entao quem me acordou foi o ft; vou atualizar origem
      origem = traj[tamtraj-1];
    fprintf(dolog,"Testando se objetivo alcancado: origem =
(%.2f,%.2f)\n",origem.x,origem.y);
    if (MESMO_PTOI(pto_mapa(origem),pdest))
    {
      fprintf(dolog,"\nOBJETIVO ALCANCADO!!!\n");
      alcancou_obj = 1;
    }
  }
  do_termina();
}
```

```c
/*************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  ft.c  - Arquivo do Fornecedor de Trajetorias


 ***********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <vga.h>
#include <unistd.h>
#include "defs.h"
#include "geometria.h"

#include <pthread.h>
#include <semaphore.h>

// variaveis no controlador
extern int acordou_ft;
extern double tinic, treal;

FILE * ftlog;

void ft_termina()
{
  fprintf(ftlog,"FT: Vou Terminar\n");
  fclose(ftlog);
  sem_post(&sem_princ);
}

ptof_t pto_ctrl(ptof_t ptraj)
// transforma as coordenadas do sistema de referencia adotado para o do robo
{
  ptof_t pctrl, paux;
  float ang = GRAUS_PARA_RAD(ang_inic);

  paux.x = ptraj.x - origem_global.x;   // translacao p/ posicao original
  paux.y = ptraj.y - origem_global.y;

  pctrl.x =  cos(ang)*paux.x + sin(ang)*paux.y;  // rotacao de ang_inic
  pctrl.y = -sin(ang)*paux.x + cos(ang)*paux.y;

fprintf(ftlog,"(%.2f,%.2f)  -->  (%.2f,%.2f)\n",ptraj.x,ptraj.y,pctrl.x,pctrl.y);
  return(pctrl);
}

/* ERRADA!!!
ptof_t pto_sist_orig(ptof_t pctrl)
// transforma as coordenadas do sistema de referencia do robo para o adotado
{
  ptof_t ptraj;
  ptraj.y = pctrl.x + origem_global.y + raio;
  ptraj.x = -(pctrl.y - origem_global.x);
  return(ptraj);
}
*/

void atualiza_traj_ctrl(void)
{
  int i;
  sem_wait(&sem_traj_ctrl);
  for (i=0; i < tamtraj; i++)
  {
```

```
    // passando as coordenadas da traj gerada para o sist de coordenadas do robo
    traj_ctrl[i] = pto_ctrl(traj[i]);
  }
  acordou_ft = 0;
  tinic = treal;
  tamtraj_ctrl = tamtraj;
  sem_post(&sem_traj_ctrl);
}

void ft()      /* fornecedor de trajetoria */
{
  int ctrl_acordado = 0;

#ifdef  GERAR_LOGS
  ftlog = fopen("ft.log","wt");
#else
  ftlog = fopen("/dev/null","wt");
#endif
  setvbuf(ftlog,NULL,_IONBF,0);

  while (1)
  {
    fprintf(ftlog,"Vou esperar trajetoria\n");
    sem_wait(&sem_ft);
    if (!alcancou_obj && !impossivel_gerar_traj)
    {
      fprintf(ftlog,"Recebi trajetoria\n");
      atualiza_traj_ctrl();
    }
    if (!ctrl_acordado)
    {
      sem_post(&sem_ctrl);
      ctrl_acordado = 1;
    }
    if (!alcancou_obj && !impossivel_gerar_traj)
    {
      fprintf(ftlog,"Vou esperar pelo controlador\n");
      sem_wait(&sem_ft);      // espera que o controlador forneca toda a trajetoria
      fprintf(ftlog,"Vou esperar o robo parar\n");
      espera_robo_parar(ftlog);
    }
    sem_post(&sem_do);
    if (alcancou_obj || impossivel_gerar_traj)
      break;
    fprintf(ftlog,"ORIGEM = (%0.2f,%0.2f)  DESTINO = (%0.2f,%0.2f)  TAMTRAJ = 
%d\n",traj[0].x,traj[0].y,traj[tamtraj-1].x,traj[tamtraj-1].y,tamtraj);
  }
  ft_termina();
}
```

```
/************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  le_sonar.c  - Arquivo do thread de leitura do sonar


 ***********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include "defs.h"
#include "geometria.h"

#include <pthread.h>
#include <semaphore.h>

extern void le_sonar(void);
extern void inic_sonar(void);
extern void fim_sonar(void);

#define RAIO_REAL     (raio - 0.10)

FILE * sonarlog;

extern void erro(char *);

extern ptoi_t pto_mapa(ptof_t ptof);

void sonar_termina()
{
  fprintf(sonarlog,"SIM: Vou teminar; entrei no sonar_termina\n");
  fclose(sonarlog);
  sem_post(&sem_princ);
}

void le_sonar()
{
  ptof_t posic;
  double ang, a,
        delta;  /* leitura do sonar vai de -delta ateh +delta */
  double dist;
  Vetor dir;
  ptoi_t pi;
  int cont=0, i;
  int x,y;
  int prim_vez = 1;
  ang = 0.0;

#ifdef  GERAR_LOGS
  sonarlog = fopen("sonar.log","wt");
#else
  sonarlog = fopen("/dev/null","wt");
#endif
  setvbuf(sonarlog,NULL,_IONBF,0);

  inic_sonar();

  while (1)
  {
    sem_wait(&sem_sonar); // solicitacao de leitura do sonar
    if (alcancou_obj || impossivel_gerar_traj)
        break;
    if (prim_vez)
```

```
    {
      prim_vez = 0;
      posic = origem; // que serah origem_global
    }
    else
    {
      if (!ler_toda_volta)
        posic = traj[tamtraj-1];
      // se eh para ler toda a volta, entao nao se moveu e continua na mesma posicao
    }

    posicao = posic; // posicao eh global, a ser usada pelas demais funcoes
// a atualizacao da direcao agora eh feita pelo controlador!
    cont = 0;
    if (ler_toda_volta)
      delta = 180.0;
    else
      delta = AMPLITUDE_LEITURA_NORMAL_SONAR;

    sonar_rpc();
    sem_post(&sem_do);
  }
  fim_sonar();
  sonar_termina();
}
```

```
/***************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  geometria.c  - Arquivo de Funcoes Geometricas


 ***************************************************************************/

// FUNCOES GEOMETRICAS ********************************************

#include <math.h>
#include "geometria.h"

Vetor vetor(Pto orig, Pto dest)
{
  Vetor v;
  v.x = dest.x - orig.x;
  v.y = dest.y - orig.y;
  return(v);
}

float modulo(Vetor v)
{
  return(sqrt(v.x*v.x + v.y*v.y));
}

Vetor mult_vet_esc(Vetor v, float e)
{
  v.x *= e;
  v.y *= e;
  return(v);
}

Vetor soma_vet(Vetor v1, Vetor v2)
{
  Vetor s;
  s.x = v1.x + v2.x;
  s.y = v1.y + v2.y;
  return(s);
}

float prod_esc(Vetor v1, Vetor v2)
{
  return(v1.x * v2.x + v1.y * v2.y);
}

float prod_vet(Vetor v1, Vetor v2)
{
  return(v1.x * v2.y - v1.y * v2.x);
}

float angulo(Vetor v1, Vetor v2)
/* retorna o angulo em radianos entre os vetores
   o angulo eh medido a partir de v1, em sentido anti-horario ateh v2 */
{
  float modv1 = modulo(v1);
  float modv2 = modulo(v2);
  float pv;
  if (modv1*modv2 == 0.0)
    return(0.0);
  if ((pv = prod_vet(v1,v2)) == 0.0)
    return(prod_esc(v1,v2) > 0.0 ? 0.0 : M_PI);
  return(pv > 0.0 ? acos(prod_esc(v1,v2)/(modv1*modv2)) : 2*M_PI -
acos(prod_esc(v1,v2)/(modv1*modv2)));
}
```

```
Vetor rot90(Vetor v)                              // rotaciona v em 90 graus
{
  Vetor r;
  r.x = -v.y;
  r.y = v.x;
  return(r);
}

Vetor rot_90(Vetor v)                             // rotaciona v em -90 graus
{
  Vetor r;
  r.x = v.y;
  r.y = -v.x;
  return(r);
}

Vetor rot(Vetor v, float ang)
{
  Vetor r;
  r.x = v.x*cos(ang) - v.y*sin(ang);
  r.y = v.x*sin(ang) + v.y*cos(ang);
  return(r);
}

int pert_pto_reta(Pto a, Pto b, Pto c) // verifica se os 3 ptos sao colineares
{
  return(F_IGUAL(prod_vet(vetor(a,b),vetor(a,c)),0.0));
}

int pert_pto_seg(Pto a, Pto b, Pto c)
{
  if (pert_pto_reta(a,b,c))
    return( F_MENORIGUAL (prod_esc(vetor(c,a),vetor(c,b)),0.0));
  return(0);
}

#define CONCORRENTES 0
#define PARALELAS    -1
#define MESMARETA    -2

int intersec_retas(Pto p1, Pto p2, Pto q1, Pto q2, Pto * pi)
                                  // *pi: pto de interseccao, se concorrentes
{
  Vetor d1, d2;
  double k;
  if (F_IGUAL(prod_vet(d1 = vetor(p1,p2), d2 = vetor(q1,q2)),0.0))
    return(pert_pto_reta(p1,p2,q1)?MESMARETA:PARALELAS);
  k = prod_vet(vetor(p1,q1),d1) / prod_vet(d1,d2);
  pi->x = q1.x + k * d2.x;
  pi->y = q1.y + k * d2.y;
  return(CONCORRENTES);
}

int intersec_segmtos(Pto a, Pto b, Pto p, Pto q, Pto * pi1, Pto * pi2)
// retorna o nro de parametros de retorno utilizados (pi1, pi2)
// se concorrentes -> retorna em pi1 o pto comum
// se mais de um pto em comum -> retorna em pi1 e pi2 os extremos do segmento comum
{
  switch(intersec_retas(a,b,p,q,pi1))
  {
    case PARALELAS    : return(0);
    case CONCORRENTES : return(pert_pto_seg(a,b,*pi1) && pert_pto_seg(p,q,*pi1));
    case MESMARETA    : if (pert_pto_seg(p,q,a))
                        {
                          *pi1 = a;
                          *pi2 = pert_pto_seg(p,q,b)?b:(pert_pto_seg(a,b,p)?p:q);
                        }
                        else
                          if (pert_pto_seg(p,q,b))
                          {
                            *pi1 = b;
                            *pi2 = pert_pto_seg(a,b,p)?p:q;
                          }
                          else
                            if (pert_pto_seg(a,b,p))
                            {
```

```
                              *pi1 = p;
                              *pi2 = q;
                            }
                            else
                              return(0);
                        return(PTO_IGUAL(*pi1,*pi2)?1:2);
  }
  return(0);  // nunca deve chegar aqui!
}
```

```
/*************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  sonar.cpp  - Arquivo de interface e leitura efetiva dos sonares,
               adaptado a partir de programa desenvolvido por Felipe Renon

 ***************************************************************************/
#include <iostream.h>
#include <gif.h>
#include<unistd.h>
#include <conio.h>
#include <ciostream.h>
#include<iostream.h>
#include<delay.h>
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include <time.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <twldrive.h>

#include "defs.h"
#include "geometria.h"

#define DIST_MIN 0.26
#define DIST_MAX 6.0  // em metros; distancia maxima que espera-se que
                      // detecte um obstaculo; se detectar algo depois
                      // disto, atrapalharah a leitura seguinte

#define DELAY ((int)(2*1000*DIST_MAX/343.2))// delay em ms referente a DIST_MAX

float delta,_angulo;

#define PRECISION 0.50
#define PADROES 4
#define ARC 2*PI

#define RED 254
#define GREEN 253
#define BLUE 252

#define PEN 3


extern ROVER_DRIVER rov;
extern FILE * sonarlog;

void bline(int x1, int y1, int x2, int y2, char color, unsigned char pic[])
{
/* Desenha uma linha (x1,y1)-(x2,y2) de cor 'color' no bitmap 'pic' */
        short int dx, dy, sdx, sdy, x, y, px, py;

        dx = x2 - x1;
        dy = y2 - y1;

        sdx = (dx < 0) ? -1 : 1;
        sdy = (dy < 0) ? -1 : 1;

        dx = sdx * dx + 1;
        dy = sdy * dy + 1;
```

```
        x = y = 0;

        px = x1;
        py = y1;

        if (dx >= dy) {
                for (x = 0; x < dx; x++) {
                        pic[py*640 + px] = color;
                        y += dy;
                        if (y >= dx) {
                                y -= dx;
                                py += sdy;
                        }
                        px += sdx;
                }
        } else {
                for (y = 0; y < dy; y++) {
                        pic[py*640 + px] = color;
                        x += dx;
                        if (x >= dy) {
                                x -= dy;
                                px += sdx;
                        }
                        py += sdy;
                }
        }
}


void circle(int x, int y, int raio, unsigned char pic[])
{
/* Circulo de centro x,y e raio 'raio' no bitmap 'pic' */
        int cor=BLUE;
        int w=640;
        int xoff=0;
        int yoff=raio;
        int balance= -raio;
        do
        {
                *(pic+(y+yoff)*w+(x+xoff))=cor;
                *(pic+(y+yoff)*w+(x-xoff))=cor;
                *(pic+(y-yoff)*w+(x-xoff))=cor;
                *(pic+(y-yoff)*w+(x+xoff))=cor;

                *(pic+(y+xoff)*w+(x-yoff))=cor;
                *(pic+(y+xoff)*w+(x+yoff))=cor;
                *(pic+(y-xoff)*w+(x+yoff))=cor;
                *(pic+(y-xoff)*w+(x-yoff))=cor;

                if ((balance += xoff++ + xoff) >= 0)
                        balance -= --yoff + yoff;
        } while (xoff <= yoff);
}

void plot_point(int x, int y, char pic[])
{
/* Ponto simples x,y no bitmap 'pic' */
        *(pic+y*640+x)=0;
}

extern "C" void inic_sonar(void);

void inic_sonar(void)
{
  rov.on();
}

extern "C" void fim_sonar(void);
void fim_sonar(void)
{
//  rov.off();  quem desligara sera o controlador
}

extern "C" void sonar_rpc(void);

void sonar_rpc(void)
```

```
{
   char nome_arq_fig[20] = "figsonar.gif";
/* Tamanho da imagem */
        const int w=640;
        const int h=480;
/* Constantes para gerar GIF */
        unsigned char *rmap=new unsigned char[256];
        unsigned char *gmap=new unsigned char[256];
        unsigned char *bmap=new unsigned char[256];
        const int numcols=256;
        const int colorstyle=0;
        char *comment="Descricao de RPCs";

        FILE *fp=fopen(nome_arq_fig,"wb");
        if(!fp)
        {
                printf("Can't open output file:%s\n",nome_arq_fig);
                return;
        }

        unsigned char *pic=new unsigned char [w*h];
        if(!pic)
        {
                printf("Can't allocate memory\n");
                return;
        }

        int i;
        for(i=0;i < 256;i++)
        {
                rmap[i]=i;
                gmap[i]=i;
                bmap[i]=i;
        }

         rmap[254]=255;
         gmap[254]=0;
         bmap[254]=0;

         rmap[253]=0;
         gmap[253]=255;
         bmap[253]=0;

         rmap[252]=0;
         gmap[252]=0;
         bmap[252]=255;

        int x,y;

/* Apaga (pinta) o bitmap na memoria */
        for(y=0;y < h;y++) for(x=0;x < w;x++) *(pic+y*w+x)=255;

        float _angulo;
        int ang;
        float roh[201];
         float rpc[201];

/* Circunferencias de referencia */
        circle(320,240,50,pic);
        circle(320,240,150,pic);
        circle(320,240,100,pic);
        circle(320,240,200,pic);


         fprintf(sonarlog,"Vou dar reset\n");
         cout << "* Inicializando posicao inicial\n";
        rov.pmotor.reset();

/* De 0 a 200, gira o pescoco para cobrir 360 graus, colocando no array
roh[] as distancias */


// ajusta delta
        if (ler_toda_volta)
                delta = 180.0;
        else
                delta = AMPLITUDE_LEITURA_NORMAL_SONAR;
```

```
        tam_list_obst = 2*(int)(delta/ANG_PASSO_MOTOR_SONAR);

// posiciona em -delta
         fprintf(sonarlog,"Vou para posicao inicial\n");
        for (i=0;i<tam_list_obst/2;i++)
        {
                rov.pmotor.step(0);
                delay(10);
        }

        cout << "* Efetuando varredura\n";

         fprintf(sonarlog,"Vou fazer a varredura\n");
        for (i=0;i<tam_list_obst;i++)
        {
                if (ler_toda_volta)
                  rov.tsonar.trig();
                else
                  rov.bsonar.trig();
                delay(DELAY);
                if (ler_toda_volta)
                  roh[i]=(rov.tsonar.utof()*343.2*1e-6)/2*50;
                else
                  roh[i]=(rov.bsonar.utof()*343.2*1e-6)/2*50;
printf("roh  %.2f\n",roh[i]);

                  // inicializa todas as RPC's com o tamanho do primeiro circulo
                  rpc[i]=0;
                rov.pmotor.step(1);
        }

        cout << "* Reset \n";
         /* Retorna o giro */
        for (i=0;i<tam_list_obst/2;i++)
        {
                rov.pmotor.step(0);
                delay(10);
        }
        rov.pmotor.reset();

/* Gera a imagem a partir de roh[], decompondo as distancias em suas
respectivas componentes XY */

        roh[tam_list_obst]=roh[tam_list_obst-1];

/* Analisa as possiveis RPC's */

        cout <<  "* localizando RPC's\n";

float media, n_arc=0, ultimo =0;
int n=0, inic_ang=0, fim_ang=0;
int n_parede=0;
char copy_data=0;

        for (int i=0; i<tam_list_obst; i++)
         {
                if ((ultimo!=0)&&(fabs(ultimo-roh[i])<=PRECISION))
                  {
                        if (n==0) inic_ang=i-1;
                          n++;
                          n_arc =n_arc + PI*roh[i-1]/100;
                          if ((n>=PADROES)||(n_arc>=ARC))
                          {
                                copy_data=1;
                                fim_ang=i;
                          }
                  }
                  else
                  {
                        media = roh[inic_ang];
                        if ((copy_data) || ((i==199)&&(copy_data!=0)))
                          {
                                copy_data=0;
                                    n_parede++;
                                  /* calcula a media da RPC */
                                  for (int k=inic_ang; k<=fim_ang; k++) media =
(media+roh[k])/2;
```

```
                                for (int k=inic_ang; k<=fim_ang; k++)
                                {
                                      rpc[k]= media;
                                }
                        }
                        n=0;
                        media=0;
                        n_arc=0;
                }
                ultimo=roh[i];
        }
        media=roh[inic_ang];
        if (copy_data)
        {
                copy_data=0;
                 n_parede++;
                /* calcula a media da RPC */
                for (int k=inic_ang; k<=fim_ang; k++) media = (media+roh[k])/2;
                for (int k=inic_ang; k<=fim_ang; k++)
                {
                        rpc[k]= media;
                }
        }

        cout << "Total de " << n_parede << " rpc(s) encontrada(s)\n\n";
       int x2,y2;
        /* tracando linhas entre pontos vizinhos    */
       for (ang=0;ang<tam_list_obst;ang++)
       {
printf("          %.2f\n",rpc[ang]);
           _angulo=ang*1.8/180*PI;
           x=320+int(rint(roh[ang]*cos(_angulo)));
           if (x<0) x=0;
           if (x>639) x=639;
           y=240-int(rint(roh[ang]*sin(_angulo)));
           if (y<0) y=0;
           if (y>479) y=479;
           _angulo=(ang+1)*1.8/180*PI;
           x2=320+int(rint(roh[ang+1]*cos(_angulo)));
           if (x2<0) x2=0;
           if (x2>639) x2=639;
           y2=240-int(rint(roh[ang+1]*sin(_angulo)));
           if (y2<0) y2=0;
           if (y2>479) y2=479;
           bline(x,y,x2,y2,0,pic);
       }

        /* tracando linhas entre as RPC's           */
       for (ang=0;ang<tam_list_obst;ang++)
       {
              /* Passa filtro                        */
              if ((rpc[ang]!= 0)&&(rpc[ang+1]!=0)&&(rpc[ang]==rpc[ang+1]))
               for (i=1; i <= PEN; i++)
               {
                      _angulo=ang*1.8/180*PI;
                      x=320+int(rint((rpc[ang]+i)*cos(_angulo)));
                      if (x<0) x=0;
                      if (x>639) x=639;
                      y=240-int(rint((rpc[ang]+i)*sin(_angulo)));
                      if (y<0) y=0;
                      if (y>479) y=479;
                      _angulo=(ang+1)*1.8/180*PI;
                      x2=320+int(rint((rpc[ang+1]+i)*cos(_angulo)));
                      if (x2<0) x2=0;
                      if (x2>639) x2=639;
                      y2=240-int(rint((rpc[ang+1]+i)*sin(_angulo)));
                      if (y2<0) y2=0;
                      if (y2>479) y2=479;
                      bline(x,y,x2,y2,RED,pic);
               }
       }

        /* Traca vetor da trajetoria */
       float maior = rpc[0];
        int k1=0, k2=0;
        for (i=1; i<tam_list_obst; i++)
        {
```

```
                if (maior <= rpc[i])
                 {
                        k1 = i;
                        if (maior==rpc[i]) k2++;
                        else k2=0;
                        maior=rpc[i];
                 }
         }

        _angulo=(2*k1-k2)*0.9/180*PI;
        x=320+int(rint((maior)*cos(_angulo)));
        if (x<0) x=0;
        if (x>639) x=639;
        y=240-int(rint((maior)*sin(_angulo)));
        if (y<0) y=0;
        if (y>479) y=479;
        _angulo=(ang+1)*1.8/180*PI;
        x2=320;
        if (x2<0) x2=0;
        if (x2>639) x2=639;
        y2=240;
        if (y2<0) y2=0;
        if (y2>479) y2=479;
        bline(x,y,x2,y2,GREEN,pic);


         /* Escreve 'pic' para o formato GIF */

//      cout << "Gerando arquivo " << argv[1] << "\n";
         cout << "\n\nARCO da RPC                  : " << k2*1.8;
         cout <<   "\nlocalizacao <Dist,ang>       : ( " << maior/50 << ", " << (2*k1-
k2)*0.9 << ").";
         cout <<   "\nCoordenada cartesiana  (x,y) : ( " << maior/50*sin(_angulo) << ", "
<< maior/50*cos(_angulo) <<").";
        writegif(fp,pic,w,h,rmap,gmap,bmap,numcols,colorstyle,comment);

        fclose(fp);
        free(pic);

// cria list_obst

        _angulo = direcao - delta;

         ptof_t pos_sonar;

         pos_sonar.x = posicao.x + DIST_CENTRO_SONAR*cos(GRAUS_PARA_RAD(direcao));
         pos_sonar.y = posicao.y + DIST_CENTRO_SONAR*sin(GRAUS_PARA_RAD(direcao));

        for (i=0; i < tam_list_obst; i++)
        {
                 rpc[i] = rpc[i] / 50;
printf("%.2f\n",rpc[i]);
                list_obst[i].pos = pos_sonar;
                list_obst[i].ang = _angulo;
                if (rpc[i] > DIST_MIN && rpc[i] < ALCANCE_SONAR)
                       list_obst[i].dist = rpc[i];
                else
                       list_obst[i].dist = INFINITO;

                _angulo += ANG_PASSO_MOTOR_SONAR;
//              printf("P=(%.2f,%.2f)",list_obst[i].pos.x,list_obst[i].pos.y);
                printf("A=%.2f   D=%.2f \n",list_obst[i].ang,list_obst[i].dist);
        }
}
```

```
/**************************************************************************
  Fundacao Universidade Federal do Rio Grande
  Engenharia de Computaco
  Projeto de Graduacao


                 Planejamento de Trajetorias para Robos Moveis
                 ---------------------------------------------

  Guilherme de Lima Ottoni

  Orientacao: Prof. Dr. Walter Fetter Lages



  twlckin.cpp  - Arquivo do Controlador - Modelo Cinematico - Robo Twil
                 Adaptado a partir de programa desenvolvido por Walter Lages

 **************************************************************************/

#define REAL_TIME

#ifdef REAL_TIME
#include <fcntl.h>
#include <unistd.h>
#endif

#include <ciostream.h>

#include <conio.h>
#include <fstream.h>

#include <contain.h>

#include <twldrive.h>

#include <twlparam.h>
#include <twlpath.h>
#include <twltype.h>

#ifdef REAL_TIME

#include <rtc.h>

#endif

#include <control.h>


#include <semaphore.h>
#include <pthread.h>
#include <delay.h>

#include "geometria.h"
#include "defs.h"
#include <stdio.h>


#define ALPHA0 1
#define ALPHA1 ALPHA0

int t_acom = (int)(1000*(3*2*M_PI/ALPHA0));

ROVER_DRIVER rov;


static const double ST=0.05;

FILE * ctrllog;

// variaveis para fornecimento da trajetoria

int acordou_ft;
double tinic=0.0, treal=0.0;

struct ODOMETRY
```

```
{
        int npr;
        int npl;
        double ur;
        double ul;
        double x;
        double y;
        double phi;
};


cvector path(double t_atual)
{
        cvector p(2);
         double t;
         uint itraj;

        sem_wait(&sem_traj_ctrl);

         t = tinic;
         itraj = 0;
         while (t < t_atual && itraj < tamtraj_ctrl-1)
         {
           t += DIST(traj_ctrl[itraj],traj_ctrl[itraj+1])/veloc;
           itraj++;
         }
         p[0] = traj_ctrl[itraj].x;
         p[1] = traj_ctrl[itraj].y;

         fprintf(ctrllog,"tamtraj_ctrl = %d   PATH =
(%.2f,%.2f)\n",tamtraj_ctrl,p[0],p[1]);
         if (itraj == tamtraj_ctrl-1 && !acordou_ft)
         {
           fprintf(ctrllog,"Acordando o FT! T = %.3f\n",t_atual);
           sem_post(&sem_ft);
           acordou_ft = 1;
         }

        sem_post(&sem_traj_ctrl);

        return p;
}


// vehicle dead-reckoning (center of wheels)
//
// p(t+1)=dr(p(t),u(t))
cvector dr(const cvector &p,const cvector &u)
{
        double deltaD=(u[0]+u[1])/2.0;
        double deltatheta=(u[0]-u[1])/WHEELBASE;
        double deltatheta2=deltatheta/2.0;

        double deltasigma=(deltatheta==0.0)? deltaD:deltaD*sin(deltatheta2)/deltatheta2;

        cvector p1(3);

        p1[0]=deltasigma*cos(p[2]+deltatheta2);
        p1[1]=deltasigma*sin(p[2]+deltatheta2);
        p1[2]=deltatheta;

        return p+p1;
}

// vehicle kinematic transform (center of wheels position->center of mass positon)
//
// cm=cw2cm(cw)
cvector cw2cm(const cvector &cw)
{
        const double d=CENTER2WHEEL_AXIS;

        cvector cm(3);

        cm[0]=cw[0]-d*cos(cw[2]);
        cm[1]=cw[1]-d*sin(cw[2]);
        cm[2]=cw[2];
```

```
        return cm;
}


// vehicle kinematic transform (center of mass position->center of wheels positon)
//
// cw=cm2cw(cm)
cvector cm2cw(const cvector &cm)
{
        const double d=CENTER2WHEEL_AXIS;

        cvector cw(3);

        cw[0]=cm[0]+d*cos(cm[2]);
        cw[1]=cm[1]+d*sin(cm[2]);
        cw[2]=cm[2];

        return cw;
}

// vehicle system output function
//
// y(t)=h(x)
cvector h(const cvector &x)
{
        cvector y(2);

        double s=sin(x[2]);
        double c=cos(x[2]);

        y[0]=x[0]+Xcr*c-Ycr*s;
        y[1]=x[1]+Xcr*s+Ycr*c;

        return y;
}

// vehicle system output function with orientation
//
// y(t)=ho(q)=ho(x)
cvector ho(const cvector &q)
{
        cvector y(3);

        double s=sin(q[2]);
        double c=cos(q[2]);

        y[0]=q[0]+Xcr*c-Ycr*s;
        y[1]=q[1]+Xcr*s+Ycr*c;
        y[2]=q[2];

        return y;
}

// invbeta(x)
matrix invbeta(const cvector &x)
{

        const double c=WHEEL_RADIUS/(2.0*AXIS2WHEEL);
        const double b=AXIS2WHEEL;
        const double d=CENTER2WHEEL_AXIS;

        double sphi=sin(x[2]);
        double cphi=cos(x[2]);

        matrix B(2,2);

        B[0][0]=(d-Xcr)*cphi+(b+Ycr)*sphi;
        B[0][1]=-((b+Ycr)*cphi+(-d+Xcr)*sphi);

        B[1][0]=-((-d+Xcr)*cphi+(b-Ycr)*sphi);
        B[1][1]=(b-Ycr)*cphi+(d-Xcr)*sphi;

        return 1/(2.0*c*b*(d-Xcr))*B;
}
```

```
// Model reference
// .
// ym(t)=g(x,u)
cvector g(const cvector &x,const cvector &u)
{
	cvector G(2);

	G[0]=-ALPHA0*x[0]+ALPHA0*u[0];
	G[1]=-ALPHA1*x[1]+ALPHA1*u[1];

	return G;
}

extern "C" void espera_robo_parar(FILE * arqlog);

void espera_robo_parar(FILE * arqlog)
{
  fprintf(arqlog,"delay em miliseg: %d\n",t_acom);
  delay(t_acom);
}

extern "C" void controlador(void);

void controlador(void)
{

	PTRLIST<struct ODOMETRY> odolist;


// initial state

	cvector x(3);
	x[0]=X0;
	x[1]=Y0;
	x[2]=PHI0;


// odomety initial conditions

	cvector xo=cm2cw(x);

// Model reference initial conditions

	cvector ym=h(x);

// Reference path

	cvector yr(3);
	yr[0]=0;
	yr[1]=0;
	yr[2]=0;
//	LINE_PATH path(yr,1,5.5e-2);	//ym=y

//	LINE_PATH path(yr,2.0,10e-2);	//ym=y


	cvector rgain(3);
	rgain[0]=2.5;		// 2.52;
	rgain[1]=3.25;		 // 0.756*3;
	rgain[2]=0.05;		// 0.189;

	cvector lgain=rgain;

	double ur=0.0;
	double ul=0.0;

	PID_CONTROLLER rpid(rgain[0],rgain[1],rgain[2]);
	PID_CONTROLLER lpid(lgain[0],lgain[1],rgain[2]);

	rpid.saturate(1,-11.7,11.7);
	lpid.saturate(1,-11.7,11.7);


	int error=0;

	int xs=wherex();
	int ys=wherey();
```

```
#ifdef  GERAR_LOGS
  ctrllog = fopen("ctrl.log","wt");
#else
  ctrllog = fopen("/dev/null","wt");
#endif
  setvbuf(ctrllog,NULL,_IONBF,0);


        fprintf(ctrllog,"\nVou aguardar no semaforo sem_ctrl\n");
        sem_wait(&sem_ctrl);


#ifdef REAL_TIME
       volatile int flag=1;

       int timer=open("/dev/rtctimer",O_RDWR);
       unsigned long blocktime;
       const unsigned long STj=(unsigned long)(ST*1000.0*1024.0/1000+0.5);
       write(timer,&STj,sizeof(STj));
#endif

//     int nmax=(int)(path.time()/ST*5);

       for(int n=0; !alcancou_obj && !impossivel_gerar_traj; n++)
       {
              double t=n*ST;

              treal = t;



#ifdef REAL_TIME
              if(n)
              {
                     if(flag)
                     {
                            cout << "\n\nOverrun!\n";
                            return ;
                     }
              }
              read(timer,&blocktime,sizeof(blocktime));
              flag=(blocktime <= 1)? 1:0;
              write(timer,&STj,sizeof(STj));
#endif

              cvector p=path(t);

              int npr=rov.rtacho.read();
              rov.rtacho.clear();

              int npl=rov.ltacho.read();
              rov.ltacho.clear();

              double rvel=npr*2.0*M_PI/rov.rtacho.PULSES/ST;
              double lvel=npl*2.0*M_PI/rov.ltacho.PULSES/ST;

              cvector u(2);
              u[0]=rvel*WHEEL_RADIUS*ST;
              u[1]=lvel*WHEEL_RADIUS*ST;

              xo=dr(xo,u);
              x=cw2cm(xo);

              cvector y=h(x);
              cvector coord_robo=ho(x);

               direcao = RAD_PARA_GRAUS(coord_robo[2]) + ang_inic;

               fprintf(ctrllog,"t=%.3f:   (%.2f,%.2f)  ang=%.2f
\n",treal,y[0],y[1],direcao);

              cvector xlin=x;         // state for linearization

              cvector v(2);
```

```
                cvector dym=g(ym,p);

                v[0]=dym[0]+ALPHA0*(ym[0]-y[0]);
                v[1]=dym[1]+ALPHA1*(ym[1]-y[1]);

                cvector w=invbeta(xlin)*v;

                double re=w[0]-rvel;
                ur=rpid.out(re);

                double le=w[1]-lvel;
                ul=lpid.out(le);

                rov.rmotor=ur;
                rov.lmotor=ul;

// Reference Model Simulation

                ym=rk(ym,p,ST,g);


                ODOMETRY *odo=new ODOMETRY;
                odo->npr=npr;
                odo->npl=npl;
                odo->ur=ur;
                odo->ul=ul;
                odo->x=y[0];
                odo->y=y[1];
                odo->phi=x[2];

                odolist.append(odo);

        }


        rov.off();

#ifdef REAL_TIME
        close(timer);
#endif

        if(!error)
        {
                cout << "Saving odometry data to disk...";
                ofstream odofile("./twlckin.dat");
                if (!odofile)
                {
                        cout << "Can't open output file.\n\n";
                        return ;
                }
                ODOMETRY *odo;
                int t=0;
                while((odo=odolist.head()))
                {
                        odofile << t++ << "\t";
                        odofile << odo->npr << "\t";
                        odofile << odo->npl << "\t";
                        odofile << odo->ur << "\t";
                        odofile << odo->ul << "\t";
                        odofile << odo->x << "\t";
                        odofile << odo->y << "\t";
                        odofile << odo->phi << "\n";
                        delete odo;
                }

                cout << "OK.\n";
        }

         fprintf(ctrllog,"Vou enviar UP pro principal\n");
         sem_post(&sem_princ);

         fprintf(ctrllog,"Terminando...\n");
         fclose(ctrllog);

}
```

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ASAMI, S. *Robots in Japan: Present and Future*, IEEE Robotics & Automation Magazine, Vol. 1, No. 2, June, 1994.

[2] SCHRAFT, R. D. *Mechatronics and Robotics for Service Applications*, IEEE Robotis & Automation Magazine, Vol. 1, No. 4, December, 1994.

[3] LATOMBE, J. C. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[4] FOUX, G.; HEYMANN, M. and BRUCKSTEIN, A. *Two-Dimensional Robot Navigation Among Unknown Stationary Polygonal Obstacles*. IEEE Transactions on Robotics and Automation, vol. 9, nº 1. Fevereiro 1993.

[5] DIJKSTRA, E. W. *A note on two problems in connexion with graphs*. Numerische Mathematik, 1959.

[6] SCHWATZ, J. T., SHARIR, M. *On the Piano Mover's Problem: The Case if a Two-Dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers*. Communications on Pure and Applied Mathematics, 36. 1983.

[7] LINGAS, A. *The Power of Non-Rectilinear Holes*. Proceedings of the 9th Colloquium on Automata, Languages and Programming. LNCS Springer-Verlag, 1982.

[8] RIBEIRO, C. C., ARAGÃO, M. V. P. *Metaheurísticas*. 11ª Escola de Computação, 1998.

[9] WELZL, E. *Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time*. Information Processing Letter, 20, 167-171. 1985.

[10] BRESENHAM, J. E. *Algorithm for Computer Control of a Digital Plotter*. IBM Systems Journal, 4(1), 25-30. 1965.

[11] LEROY, X. *The LinuxThreads library*. 1996. http://pauillac.inria.fr/~xleroy/linuxthreads/

[12] ANOUSAKI, G.C., KYRIAKOPOULOS, K.J. *Simultaneous Location and Map Building for Mobile Robot Navigation*. IEEE Robotics & Automation Magazine. Setembro, 1999.

[13] LAGES, W.F., HEMERLY, E.M., PEREIRA, L.F.A. *Controle Linearizante de uma Plataforma Móvel Empregando Realimentação Visual.* XI Congresso Brasileiro Automática. São Paulo, 1998.

[14] RENON, F.J., LAGES, W.F. *Navegação de robôs móveis utilizando sonares*. Anais do VIII Congresso de Iniciação Científica FURG/UFPel/UCPel. Rio Grande, 1999.

[15] LAGES, W.F., HEMERLY, E.M. *Controle em Coordenadas Polares de Robôs Móveis com Rodas*. XII Congresso Brasileiro de Automática. Uberlândia, 1998.

[16] LAGES, W.F., HEMERLY, E.M. *Smooth Time-invariant Control of Wheeled Mobile Robots*. XIII International Conference on Systems Science. Wroclaw, Polônia, 1998.

[17] LAGES, W.F., HEMERLY, E.M. *Adaptive Linearizing Control of Mobile Robots*. 5th IFAC Workshop on Intelligent Manufacturing Systems. Gramado, 1998.

[18] RENON, F.J.R., LAGES, W.F. *Navegação de Robôs Móveis Utilizando Sonares de Ultra-som*. Anais do VII Congresso de Iniciação Científica FURG/UFPel/UCPel. Pelotas, 1998.