Faculdade de Engenharia da Universidade do Porto

Departamento de Engenharia Mecânica e Gestão Industrial

Simulação Visual de Armazéns Automáticos

Dissertação de Mestrado de:

José Manuel Feliz Teixeira

Licenciado em Física Aplicada pela

Faculdade de Ciências da Universidade do Porto

—

Realizado sob a orientação de:

Prof. Doutor António Carvalho Brito

Mestrado de Engenharia Mecânica 1995/97

Secção de Gestão e Engenharia Industrial (GEIN)

Departamento de Engenharia Mecânica

OUTUBRO 1997

ÍNDICE

Lista de Figuras

1. Introdução

1.1 Apresentação

Os sistemas automáticos de armazenamento são hoje em dia equipamentos fundamentais na modernização dos centros de produção e distribuição como forma de aumentar tanto a sua flexibilidade como a capacidade de resposta às exigências actuais de processamento. Desde a década de oitenta que o património industrial tem vindo a enriquecer com a instalação de sistemas deste tipo, tendo sido acumulada uma larga experiência neste sector por parte de algumas empresas portuguesas, principalmente no que respeita a soluções de instalação e ao controle dos equipamentos envolvidos.

Uma das empresas nacionais que maior volume de negócios possui nesta área é, como se sabe, a EFACEC S.A. Devido a esse facto, e ainda pelo interesse que os seus responsáveis demonstraram em acompanhar este projecto, foi decidido desenvolver-se este trabalho com base nas soluções mais utilizadas por esta empresa na construção de armazéns automáticos. Foram realizadas algumas reuniões onde se definiram os principais contornos do projecto, assim como nos inteirámos dos verdadeiros processos envolvidos, tanto no que respeita aos métodos de armazenagem como aos mecanismos de transporte de material. Foi ainda apreciada a estrutura dos seus sistemas informáticos de controle e de gestão com vista a planear-se este software segundo uma filosofia de responsabilidades idêntica à usada por aquela empresa. Com esta tentativa de modelar o problema de uma forma o mais próximo possível com a realidade, espera-se conseguir uma aplicação informática que permita tanto uma rápida e eficiente configuração pela simples manipulação dos recursos existentes, como a obtenção de resultados concretos mais fiáveis relativamente à eficiência da futura instalação.

A importância de sistemas de simulação neste campo da indústria é reconhecida por todos, no entanto, dado que os actuais sistemas disponíveis no mercado se revelam ou demasiado simples, o que inibe a construção de um modelo suficientemente representativo da realidade, ou demasiado complexos, tornando-se estes de difícil manuseamento e fastidiosa configuração, raras empresas se munem deste tipo de ferramentas. A escolha da alternativa mais adequada, em cada caso, depende

essencialmente da experiência do analista, das características da instalação a simular, do tempo disponível para a realização do projecto e, como é óbvio, do seu orçamento. No entanto, com a evolução dos meios informáticos no sentido das linguagens orientadas por objectos, pela sua acentuada baixa de preço e pela flexibilidade introduzida pelas novas interfaces com o utilizador, a simulação começa a ser praticável mesmo nos casos que implicam uma solução por medida.

1.2 Objectivos

O trabalho aqui descrito representa uma parte de um projecto mais vasto no âmbito da simulação de armazéns automáticos, parte esta que diz respeito à modelação dos vários elementos envolvidos (veículos, tapetes, etc.), assim como ao desenvolvimento da respectiva interface do utilizador. Não se poderá, por isso, considerar este trabalho um trabalho acabado, pois há ainda que conceber e implementar muitos conceitos que aqui não são descritos. No entanto, como utilidade futura, pretende-se que este software possa vir a ser usado na avaliação de performances, quer para apoio a acções de 'marketing' quer na ajuda ao planeamento de recursos e configuração de instalações.

Tratar um armazém de um ponto de vista hierárquico, de forma a separar as responsabilidades dos acontecimentos pelos vários elementos que dele fazem parte, é a concepção que aqui se propõe. Trata-se de uma abordagem modular, onde cada entidade é responsável pela sua própria integridade e pela sua própria funcionalidade. Tal como acontece na realidade, um tapete, por exemplo, é visto como um elemento que possui a capacidade de transportar um objecto que se lhe coloque em cima, desde que se encontre em funcionamento. É de esperar, então, que o mesmo aconteça logo que o operador decida criar um tapete no programa de simulação. Nesta perspectiva, os restantes componentes da simulação são métodos de transferência de material entre os vários elementos e a implementação de alguns critérios de decisão em situações mais ou menos particulares.

Para além do aspecto funcional dos vários elementos, ou entidades, da simulação, foi ainda desenvolvido um módulo de desenho que permite ao utilizador representar todo o armazém à escala, proporcionando facilidades de ZOOM e de movimentação dos vários objectos através do simples arrastar do ponteiro do "rato", tal como é comum nos programas desenvolvidos para os sistemas operativos *Windows95/NT*.

Os objectivos da parte aqui apresentada referem-se à restruturação dos conceitos de simulação de armazéns, tendo em conta a futura implementação de uma solução prática que se baseie numa filosofia orientada para objectos.

Por se tratar de uma ferramenta de programação orientada para objectos, pela grande semelhança de conceitos entre esta linguagem e os problemas que aqui se pretendem simular, e pela facilidade com que se implementa a interface com o utilizador, decidiu-se desenvolver este trabalho em *Visual C++ da Microsoft*. Desta forma, espera-se que o leitor se encontre minimamente familiarizado com os conceitos inerentes às linguagens deste tipo, pois poderá encontrar neste trabalho ideias concretas de implementação que lhe poderão ser úteis na prática. Pareceu também de maior utilidade substituir alguns fluxogramas por uma apresentação comentada do respectivo código desta linguagem.

1.3 Temática dos próximos capítulos

No próximo capítulo (*capítulo 2*) faz-se a uma breve descrição do actual "estado das coisas" no campo da simulação em computador e algumas referências a outros sistemas normalmente usados para esse efeito. Introduzem-se depois os princípios e conceitos básicos da simulação por eventos, de forma a que seja mais fácil perceber os processos de modelação dos diversos elementos do armazém, e de forma a familiarizar a nomenclatura aqui usada.
No *capítulo 3* pretende dar-se uma perspectiva mais concreta do que são e como funcionam os elementos físicos mais comuns num armazém automático, igualmente visando introduzir o leitor à nomenclatura usada e aos mecanismos envolvidos em cada um desses elementos. Trata-se de elementos físicos, isto é, susceptíveis de ocupar espaço dentro da área do armazém
No *capítulo 4* é apresentada uma breve introdução à estrutura básica de uma aplicação informática criada através do *Microsoft Visual C++*, e no *capítulo 5* fala-se da estrutura que sustenta o programa de simulação ARMAZÉM ao qual se refere este trabalho.
Ao longo do *capítulo 6* são descritas e explicadas as estruturas das classes de objectos que pretendem modelar os elementos do armazém. Esses elementos foram distinguidos em *elementos físicos*, que ocupam espaço no armazém (veículos, estantes, áreas, etc.), e *elementos conceptuais*, tais como ordens, tarefas, pedidos, etc.
O *capítulo 7* é reservado aos processos envolvidos na simulação do armazém. Aí se fala sobre os diversos "mecanismos" do simulador e de como eles funcionam, para além de se descreverem os eventos tanto dos *elementos físicos* como dos *elementos conceptuais*. É neste capítulo que se trata de interligar estes elementos uns com os outros.
No *capítulo 8* é mostrado o aspecto do programa ARMAZÉM e do tipo de interface do utilizador, ao passo que se descreve o modo de criar um modelo simples com tapetes e veículos.
Finalmente, no *capítulo 9* são apresentadas as conclusões.

2. Conceitos básicos de Simulação

2.1 Considerações gerais

O objectivo da simulação é a criação de um *modelo* capaz de representar um determinado *sistema* real sobre o qual o utilizador possa executar acções que o levem a dele extrair conclusões o mais fiáveis possível. Segundo esta perspectiva, a própria ciência mais não é do que o grande *modelo* da realidade. Através de determinadas equações matemáticas é possível "prever" a posição de um planeta na sua órbita, o instante em que um pêndulo atinge o máximo de deslocamento, a temperatura de uma caldeira, as horas em Nova York, etc. Modelar é, por isso, uma necessidade inerente à nossa espécie.
No entanto, perante a elevada complexidade de muitos *sistemas* reais, a abordagem analítica torna-se, por vezes, "impraticável", não permitindo criar uma "perspectiva" eficaz do sistema, quer pelo tempo que é necessário despender na criação do modelo, quer pela impossibilidade prática de lidar com um grande número de variáveis, quer pelos conhecimentos matemáticos exigidos ao analista. Por isso, ao longo destas últimas décadas, e acompanhando a evolução dos sistemas digitais e das linguagens de programação, outras técnicas têm vindo a ser estudadas e desenvolvidas para "contornar" esta questão. Surge a *Simulação por Computador*, que alia a capacidade de cálculo à possibilidade de manuseamento de um grande número de dados, mas a construção do modelo continua o cerne do problema: o computador só ditará resultados se nele for introduzido o modelo.
Aparecem as técnicas de simulação *discreta*, com as quais algumas tentativas de representar certos *sistemas* de um outro ponto de vista, nomeadamente com base em relações funcionais, deram origem a abordagens mais simples que, aos poucos, evoluíram para as actuais técnicas de simulação. Desenvolveram-se métodos baseados em ferramentas de software "*data-driven*"[1], como o caso do XCell+ e do HOCUS, onde o analista não necessita de escrever qualquer código de programa, mas somente interagir com o software de forma a criar o modelo. Neste tipo de aplicações, o analista define as entidades do sistema e os estados pelos quais elas passam (chamando-lhes actividades e filas), assim como a duração dessas actividades e o modo como permanecem nessas filas. No entanto, este tipo de ferramentas destinam-se a simular *sistemas* simples, que muitas vezes não correspondem aos *sistemas* reais. Nestes, a existência de um grande número de entidades e de interacções complexas entre elas exige outro tipo de programas, quer baseados em linguagens de simulação (como os SIMSCRIPT, SIMULA, SLAM II), quer desenvolvidos para o efeito em linguagens '*multi-propose*' como o C, o FORTRAN, ou o Pascal. Aplicações deste tipo podem demorar bastante mais tempo a serem desenvolvidas, mas são a melhor maneira de simular determinados sistemas de forma satisfatória, uma vez que permitem implementar um *modelo* específico para o *sistema* que se deseja estudar.
A.C.Brito[2], nos finais da década de 80, desenvolve em FORTRAN um *modelo Visual e Interactivo* de armazéns automáticos usando simulação *discreta*, introduzindo técnicas de CAD (*Computer Aided Design*) e definindo grande parte dos processos envolvidos neste tipo de *sistemas*. Nos últimos anos, com o aparecimento das linguagens de programação *Orientadas a Objectos*, o trabalho de modelação tem

vindo a ser facilitado, uma vez que estas linguagens permitem usar conceitos de hierarquia de classes, encapsulamento de características, polimorfismo, etc., tornando mais fácil estruturar e conceber o modelo, e em 1987, Balci e Nance[3] lançam as bases de um gerador de modelos designado por "*Simulation Model Development Environment*" (SMDE) que, em 1997, veio dar origem ao "*Visual Simulation Environment®*" (VSE) cuja estrutura assenta neste paradigma de programação.
Hoje em dia assiste-se à proliferação de sistemas de simulação discreta (SIMFACTORY, SLAMII, MODSIM), uns mais elaborados do que outros, desenvolvidos com intuitos generalistas, no entanto, continua a ser preferível, pelo menos enquanto a fiabilidade e a precisão dos resultados assim o exigirem, desenvolver um modelo específico tendo em conta um problema específico.

Basicamente, quanto ao tipo de sistema a modelar, distinguem-se hoje dois tipos de técnicas de simulação[4]: a simulação *contínua* e a simulação *discreta*. Na primeira trata-se de sistemas onde os estados das entidades variam continuamente. É o caso do movimento de um projéctil, por exemplo, considerando-se este a entidade e a velocidade o seu estado. Na simulação *discreta* representam-se sistemas cujas entidades possuem estados *discretos*, ou melhor, um número finito de estados. Num tapete transportador de material podemos, por exemplo, definir o estado PARADO e o estado EM-MOVIMENTO, e, a partir daí, conseguir modelar minimamente o seu funcionamento.
Por razões óbvias, neste trabalho segue-se a técnica da simulação discreta.

2.2 Simulação por eventos

Como breve introdução ao método de simulação aqui usado, importa referir que se optou pela simulação discreta, isto é, pela representação dos estados do sistema somente em instantes discretos do tempo. Este tipo de abordagem, ao contrário do que se passa no caso da simulação contínua no tempo, conduz normalmente a uma formulação mais simples do problema. A simulação contínua exige, em geral, modelos mais complexos, baseados em equações diferenciais. Por outro lado, a simulação discreta, observando o sistema somente em determinados instantes do tempo, revela-se mais flexível para utilizar em sistemas informáticos, assim como exige um menor esforço matemático no que respeita à construção e ao processamento do modelo. Existem, no entanto, dois métodos de avanço do tempo em simulação discreta (Pidd[1]):

- avanço por intervalos regulares (*Time Slicing*), em que se opta por uma amostragem dos vários estados do sistema (neste caso do modelo) com uma determinada frequência fixa.

- técnica do próximo evento (*Next Event*) onde o tempo avança para o próximo instante em que acontece uma mudança de estado.

Normalmente a técnica do próximo evento é mais eficiente, uma vez que a frequência de amostragem se adapta à sucessão natural dos acontecimentos, permitindo uma actualização do sistema somente nos momentos relevantes. Como parece evidente, a quantidade de cálculos tende também a ser menor neste tipo de aproximação, implicando uma maior precisão de resultados, pois é menos susceptível à acumulação de erros.
A escolha desta técnica de simulação deveu-se, por um lado, à sua eficiência na modelação de sistemas de armazenagem, e, por outro, ao elevado nível de conhecimentos existentes neste laboratório a esse respeito, onde esta técnica tem vindo a ser usada para modelar diversos tipos de sistemas desde há cerca de uma década.

2.3 Conceitos de entidade e de evento

A evolução temporal do estado de um qualquer sistema real pode ser considerada como o somatório de comportamentos parciais de determinados elementos desse sistema, elementos esses que normalmente se denominam por *entidades*. Entidades serão, por exemplo, os operários e as máquinas numa fábrica, os clientes e os barbeiros numa barbearia, e em geral todos os elementos que sejam susceptíveis de mudar de estado.
Por outro lado, considera-se que existe um *evento* no instante em que um sistema muda de estado. O aparecimento de um *evento* despoleta então uma determinada sequência de acções no sistema, podendo ou não dar origem a novos eventos no futuro. A essa sequência de acções chamaremos o *método* do evento.
Existem vários paradigmas de simulação com objectivos didácticos, entre eles, o caso dos filósofos e o caso da barbearia, através dos quais se costuma fazer a introdução a estes conceitos assim como à utilização de filas de espera. No entanto, optou-se aqui por utilizar um exemplo mais simples que, apesar de original, se revela bastante eficiente nessa tarefa, tendo sido baptizado como "*o caso do autocarro*". Este autocarro é responsável por recolher, no início do dia, os operários de um armazém e de os transportar até ao local de trabalho.

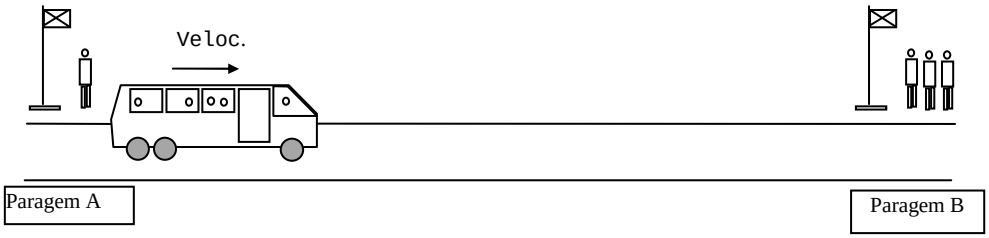

Fig. 2.1 O caso do autocarro

Suponhamos então, como única *entidade* do sistema, o autocarro. Sem nos preocuparmos com questões de lotação, considere-se o objectivo do autocarro dirigir-se da paragem **A** para a paragem **B**. Neste exemplo simples, o autocarro poderá encontrar-se em dois estados diferentes: em carreira (a que chamaremos estado `CARREIRA`) enquanto se encontra a realizar o percurso, ou parado (estado

PARADO) no momento em que chega a uma paragem e aguarda pela entrada e saída dos clientes. A existência destes dois estados implica que se devem associar dois *eventos* à *entidade* autocarro: um evento que representa o instante da mudança do estado PARADO para o estado CARREIRA (chamemo-lo EVENTO_INICIO) e outro que assinala a mudança de CARREIRA para PARADO (chamemo-lo EVENTO_FIM).

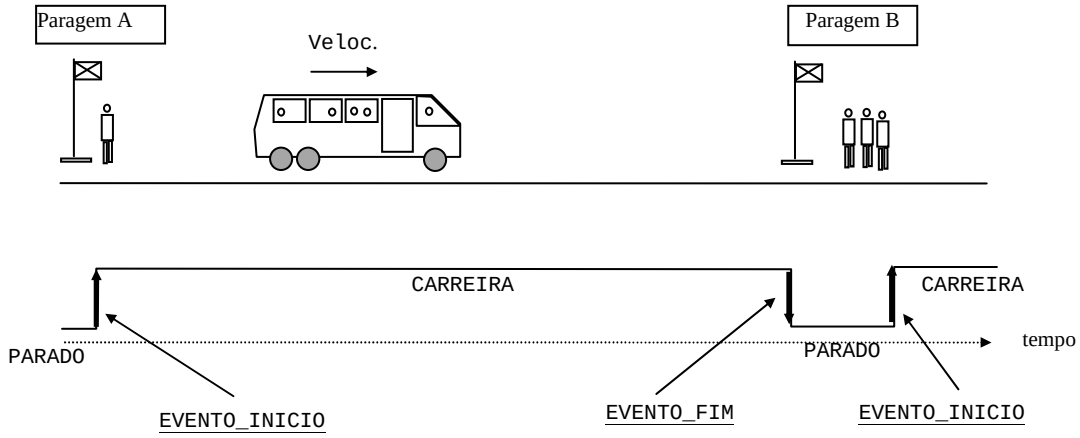

Fig. 2.2 Caso do autocarro e respectivos estados e eventos

Ao acontecer um *evento* é despoletado um determinado conjunto de acções no sistema que definirá a actividade que será mantida até ao aparecimento do próximo evento. Essa actividade é activada pela execução do *método* (ou função) associado ao evento em causa. Para cada *evento* é então necessário definir-se o respectivo conjunto de acções, isto é, o seu *método*. No caso aqui considerado, a simulação resume-se à implementação prática dos seguintes métodos:

- EVENTO_INICIO (*método*)
    - *Coloca o estado do autocarro em* CARREIRA.
    - *Dada a velocidade do autocarro e a distância até à próxima paragem, calcula o instante em que será atingida a próxima paragem e marca um evento* EVENTO_FIM *para esse instante do tempo.*
- EVENTO_FIM (*método*)
    - *Coloca o estado do autocarro em* PARADO.
    - *Calcula, de alguma forma, o tempo que o autocarro vai estar parado para a entrada e saída de clientes (dt). Se for para continuar a viagem, marca um novo* EVENTO_INICIO *para o tempo actual + dt.*

Neste caso, a actividade do sistema resume-se à alteração do estado de uma única entidade. Vejamos agora um caso em que existem duas entidades diferentes a interferir uma com a outra. Para tal, consideremos a existência de uma passadeira para peões entre a paragem **A** e a paragem **B** (fig.2.3). A maneira mais simples de simular esta situação seria a de considerar a passadeira como um obstáculo e não

como uma entidade independente. Nesse caso, o estado da passadeira (`LIVRE` ou `OCUPADA`) e o tempo médio de passagem dos peões (*dT*) seriam determinados por um método estocástico durante o acontecimento relativo ao `EVENTO_INICIO`. Se a passadeira viesse a estar ocupada, então marcar-se-ia um `EVENTO_FIM` para o instante em que o autocarro atingisse a passadeira (*to*) e outro `EVENTO_INICIO` para o instante em que os peões já tivessem atravessado (*to* + *dT*). Se não se esperasse a passadeira `OCUPADA`, o autocarro percorreria o seu trajecto normal até atingir a paragem **B**.

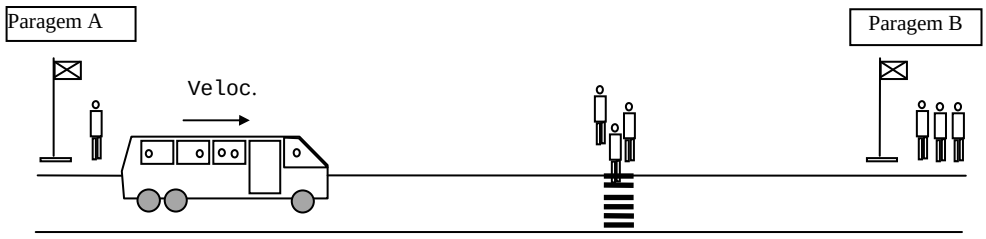

Fig. 2.3 Caso do autocarro com passadeira para peões

Com um procedimento deste tipo poder-se-ia simular a situação, no entanto, a passadeira não seria considerada como uma entidade independente do autocarro, isto é, o seu estado era determinado no instante em que o autocarro iniciava o percurso. Suponhamos agora uma situação mais representativa da realidade, em que o estado da passadeira só deverá ser conhecido no instante em que o autocarro aí se encontra. Digamos que o estado da passadeira poderá ser alterado por outros acontecimentos. Neste caso é necessário modelar-se a passadeira tendo em conta os eventos associados a ela própria. Chamando `PASS_OCUPADA` ao evento que representa o instante de tempo em que ela passa de `LIVRE` a `OCUPADA`, e `PASS_LIVRE` ao evento correspondente à transição de `OCUPADA` para `LIVRE`, teremos dois métodos associados à passadeira que representam os respectivos acontecimentos:

- `PASS_OCUPADA` (*método*)
    - *Coloca o estado da passadeira em* `OCUPADA`.
    - *Calcula, de alguma forma, o tempo que a passadeira vai estar ocupada com a travessia de peões (dt). Marca um novo* `PASS_LIVRE` *para o tempo actual + dt.*
- `PASS_LIVRE` (*método*)
    - *Coloca o estado da passadeira em* `LIVRE`.
    - *Verifica se existe algum autocarro parado à espera da passadeira. Se sim, marca um novo* `EVENTO_INICIO` *dirigido a esse autocarro para o tempo actual.*

Como é evidente, os métodos do autocarro deverão ser alterados também, pois agora existe um novo estado referente a essa entidade: o estado de espera pela passadeira (`ESPERA`). Existem vários modos de incluir este estado no processo da simulação, podendo de novo considerar-se a existência de eventos nos instantes de transição

para o estado `ESPERA` ou, simplesmente, colocando essa entidade numa lista de autocarros à espera da passadeira. Por motivos didácticos utilizaremos esta última abordagem. Assim, os novos métodos relacionados com o autocarro serão:

- `EVENTO_INICIO` (*método*)
  - *Se existir uma passadeira entre a posição actual e a próxima paragem, e o autocarro não se encontra à espera na passadeira (o estado do autocarro é diferente de* ESPERA*) então calcula o instante em que será atingida a passadeira e marca um evento* EVENTO_ESPERA *para esse instante do tempo.*
  - *Se não for válida a anterior consideração, dada a velocidade do autocarro e a distância até à próxima paragem, calcula o instante em que será atingida a próxima paragem e marca um evento* EVENTO_FIM *para esse instante do tempo.*
  - *Coloca o estado do autocarro em* CARREIRA.
- `EVENTO_ESPERA` (*método*)
  - *Coloca o estado do autocarro em* ESPERA.
  - *Se a passadeira se encontrar ocupada, coloca o estado do autocarro em* ESPERA *e coloca a entidade autocarro na lista de autocarros à espera na passadeira.*
  - *Se a passadeira estiver livre, marca um evento* EVENTO_INICIO *para o autocarro e para este instante do tempo.*
- `EVENTO_FIM` (*método*)
  - *Coloca o estado do autocarro em* PARADO.
  - *Calcula, de alguma forma, o tempo que o autocarro vai estar parado para a entrada e saída de clientes (dt). Se for para continuar a viagem, marca um novo* EVENTO_INICIO *para o tempo actual + dt.*

2.4 A lista de eventos da simulação (organização do tempo)

Como se pode deduzir da secção anterior, a marcação de eventos é muitas vezes feita num tempo futuro, sendo por isso necessário guardar esses eventos até que o tempo da simulação tenha atingido esse instante. É, por isso, vulgar usar-se uma lista na simulação, a que se costuma chamar *Lista de Eventos*, que é responsável por manter a informação sobre todos os eventos que ainda não foram executados. Qualquer evento a ser executado é, antes de mais, enviado para essa lista através da chamada de um método da simulação que aqui designaremos por *Schedule()*. A utilização desta palavra inglesa deve-se, por um lado, ao facto de ela ser usada em grande parte da literatura que aborda este assunto e, por outro, porque significa, precisamente, a marcação de acontecimentos no tempo, como acontece num horário. Dado que um evento na simulação é identificado não só pelo instante de tempo em que deverá ocorrer, mas também por um número de ordem e por uma referência à entidade a que está associado, a *Lista de Eventos* deverá ser constituída por elementos que possuam toda essa informação. Concebeu-se para isso um objecto

elementar a que se chamou *célula de tempo*, e que representa a unidade onde é guardado um evento na *Lista de Eventos*.

Fig. 2.4 Lista de eventos como um conjunto de células de tempo.

A marcação de eventos na *Lista de Eventos* é feita, como já se referiu, através do método *Schedule()* que é responsável por reorganizar esta lista por ordem de tempos e, tratando-se de eventos a marcar para o mesmo instante de tempo, por prioridade dos eventos. Considerou-se que a prioridade de um evento é definida pelo número de ordem desse evento, sendo tanto maior quanto menor for o valor desse número. A prioridade máxima estabeleceu-se ser *zero* (0), limitando a identificação de eventos ao domínio dos números inteiros.
Segundo os critérios aqui adoptados, qualquer entidade que necessite de marcar um evento deverá fazê-lo através de uma chamada ao método *Schedule()*, passando-lhe como parâmetros os parâmetros que constituem uma *célula de tempo*, por exemplo:

```
Schedule( t = 10, PASS_OCUPADA, pPassadeira);
```

será a forma de marcar para o tempo 10 o evento `PASS_OCUPADA` respeitante à passadeira referenciada por `pPassadeira`.

2.5 O controle da execução de eventos (relógio)

Uma vez marcado um evento na *Lista de Eventos*, a sua execução é despoletada por um outro método da simulação a que se costuma chamar *Relógio()*. Esta designação deriva do facto de ser este método o responsável pela actualização do tempo na simulação. Para além disso, é também a este método que cabe a tarefa de retirar o primeiro evento da cabeça da *Lista de Eventos* e de o "enviar" à respectiva entidade para execução. Trata-se de um ciclo no programa de simulação cuja actividade cessará quando deixarem de existir, nessa lista, eventos para processar.

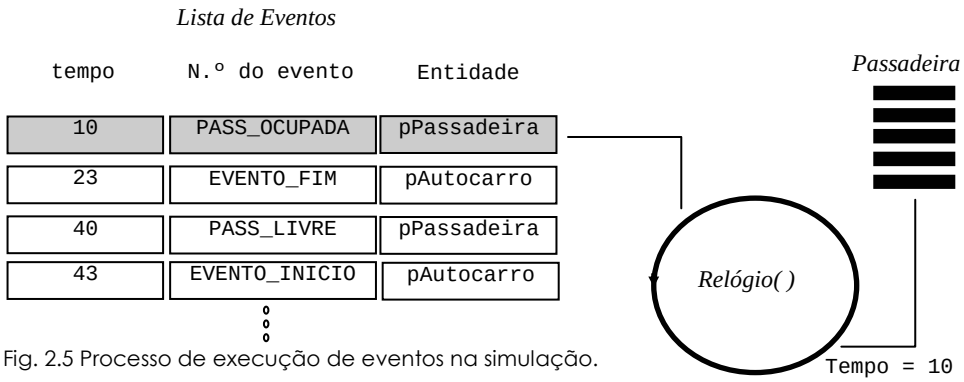

Fig. 2.5 Processo de execução de eventos na simulação.

Na figura 2.5 está esquematizado, como exemplo, o processo de execução do evento `PASS_OCUPADA` que se encontra à cabeça da *Lista de Eventos*. A primeira tarefa do método *Relógio()* é retirar esse evento da lista (que corresponde a retirar a respectiva *célula de tempo*), depois actualizar o tempo da simulação colocando-o igual ao tempo desse evento, e, por último, dar ordem à respectiva entidade (`pPassadeira`) para executar o método relativo a esse evento.
Existem variadas formas de implementar em software as ideias aqui referidas, no entanto, o objectivo desta secção limita-se a uma breve introdução ao problema, deixando para mais tarde as considerações de ordem prática.

3. Elementos físicos de um armazém automático

Os elementos existentes num armazém são múltiplos, considerando os diversos métodos de armazenagem e as variadas filosofias de projecto e implementação, no entanto, aqui se referem somente aqueles que usualmente fazem parte das soluções adoptadas pela empresa EFACEC. Nesta secção pretende-se dar uma perspectiva mais concreta do que são e como funcionam os elementos mais comuns de um armazém automático, com vista à introdução da nomenclatura usada e dos mecanismos envolvidos em cada um desses elementos.
Trata-se aqui de elementos físicos, isto é, susceptíveis de ocupar espaço dentro da área do armazém. Para além deste tipo de elementos considerar-se-ão, mais tarde, outros que não possuem características físicas mas que de igual forma são importantes para os vários processos envolvidos num armazém, tal como, por exemplo, a *Ordem de Saída de Material*.
No geral, podem dividir-se nos dois seguintes grupos os elementos físicos existentes dentro de um armazém automático:

- **Elementos estáticos**
- **Elementos móveis**

No primeiro destes grupos podem incluir-se, por exemplo, as zonas de recepção e de expedição de material, os locais de parque de veículos, o complexo de vias por onde os veículos se movem (carris, por exemplo), as estantes, as células das estantes, etc.
Pertencentes ao grupo dos *elementos móveis* poder-se-ão considerar os tapetes, as mesas rotativas, um operário cujo objectivo é executar uma movimentação de caixas, assim como os vários tipos de veículos que promovem a transferência de material dentro da área de armazenamento.
Em geral, ao grupo dos *elementos estáticos* cabem responsabilidades de armazenamento ou de espera, enquanto que aos *elementos móveis* se atribuem tarefas de transferência de material. Esta definição não é, contudo, exacta, uma vez que, por exemplo, para uma zona de parque de um veículo não é esperado transferir-se material, apesar de um parque ser considerado um *elemento estático*. Da mesma forma, uma via (outro *elemento estático*) só tem como objectivo apoiar o movimento de determinados *elementos móveis*. De qualquer maneira, a ideia de dividir os elementos físicos nestes dois grupos parece ser suficientemente representativa das responsabilidades atribuídas dentro de um armazém automático aos diversos tipos de elementos que o constituem.
De um ponto de vista de "caixa negra", um armazém pode ser encarado como uma entidade que recebe ordens de entrada e de saída de material, assim como ordens internas de reorganização, e as processa através de mecanismos mais ou menos complexos, envolvendo nessa actividade tanto os seus *elementos móveis* como os *estáticos*.

3.1 Elementos físicos estáticos

3.1.1 Áreas e cais de recepção e de expedição

Uma *área de recepção* representa o espaço físico do armazém reservado para a recepção de material proveniente do exterior. No entanto, apesar da sua designação, uma *área de recepção* não tem que ser exclusivamente usada para esse fim, podendo albergar outro tipo de zonas, tal como, por exemplo, zonas de parque de veículos. De qualquer forma, a localização de um *cais de recepção* será sempre dentro de uma *área de recepção* (fig.3.1).

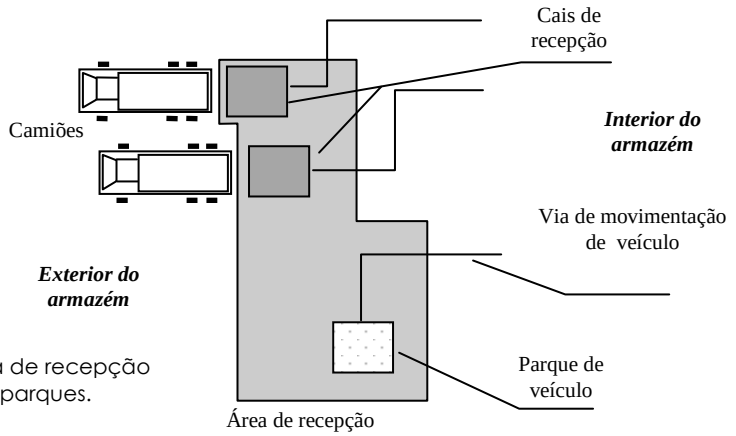

Fig. 3.1 Exemplo de Área de recepção contendo cais e parques.

Chama-se *cais de recepção* a um local específico da área de recepção que tem por finalidade receber o material proveniente do exterior (geralmente de um camião) de onde depois será transportado por um *elemento móvel* (ou vários) até à zona de armazenamento. A um cais, tal como a qualquer outro *elemento estático*, estará sempre associado, pelo menos, um *elemento móvel*.
O material proveniente do camião é descarregado para o *cais de recepção* e aí aguardará pela sua transferência para o interior do armazém. Dado que estes locais são propícios à acumulação de material, é importante assegurar-se uma eficaz gestão dos mecanismos de transporte associados, dependendo deles a eficiência do processo de transferência de material.
As características de uma *área de expedição* são idênticas ás de uma *área de recepção*, diferindo somente nos objectivos a que se destina. É na *área de expedição* que se encontram localizados os *cais de expedição*, locais destinados a servir os pedidos de material vindos do exterior. Existe, no entanto, uma diferença fundamental entre a zona de *recepção* e a de *expedição*: Na primeira é possível determinar como o material chega ao armazém, através das encomendas aos fornecedores ou à produção de uma fábrica, o que permite controlar com maior exactidão os momentos de chegada de material, enquanto que à *expedição* chegam as encomendas dos clientes que, como se sabe, exigem um processamento de maior responsabilidade, razão pela qual se considera, normalmente, prioritária a actividade nesta zona do armazém.

3.1.2 Estantes

As *estantes* são, por excelência, os locais onde o material é armazenado. Estas podem apresentar diversos tipos de configuração e possuir diversas formas de acesso ao seu conteúdo. Sendo, em geral, constituídas por um aglomerado de *células* (ou alvéolos) e sustentadas por uma estrutura que lhes confere rigidez, as *estantes* permitem organizar o espaço do armazém no que respeita a uma disposição mais racional do material. Para além disso, a escolha adequada do tipo de *estante* pode diminuir consideravelmente o tempo de resposta dos processos de entrada e de saída de material do armazém.
Antes de se falar dos diversos tipos de estantes, convém definir melhor os conceitos de *célula* e de *alvéolo*. Chama-se *alvéolo* à unidade mínima de armazenagem de uma estante. No exemplo ilustrado na figura 3.2, um *alvéolo* coincide com uma *célula*, no entanto, nem sempre assim é. De facto, o conceito de *célula* é abstracto, pois trata-se mais de um suporte para modelar este tipo de objectos do que propriamente

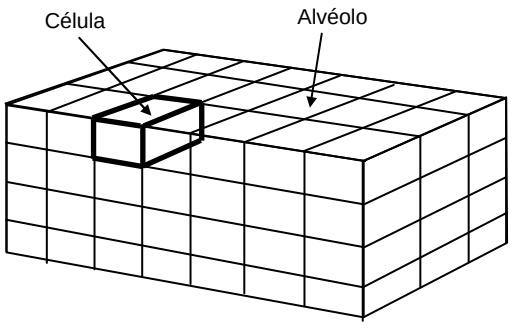

Fig. 3.2 Estante como um aglomerado de células.

de uma entidade física. Digamos que uma *célula* se poderá considerar como o conjunto mínimo de *alvéolos* contíguos que admite o mesmo tipo de lógica de acesso ao material. Do ponto de vista da simulação, somente interessa considerar-se o conceito de *célula*, sendo praticamente esquecido o conceito de *alvéolo*. Por isso aqui falaremos de *estante* como se tratando de um aglomerado de *células*.
Neste trabalho consideraram-se seis tipos diferentes de estantes, designados por:

- **APR Single Deep**

Estantes com um alvéolo de profundidade em cada lado de acesso. A célula corresponde ao alvéolo e só permite espaço para uma palete de material. Permite um acesso aleatório às paletes aí armazenadas.

- **APR Double Deep**

Estantes com um alvéolo de profundidade em cada lado de acesso. A célula corresponde ao alvéolo e permite espaço para duas palete de material. Permite um acesso aleatório a cada célula, mas não às paletes na mesma célula.

- **Block Stacking**

Estantes com vários alvéolos de profundidade em cada lado de acesso. Não possui estrutura de suporte das paletes, sendo estas colocadas umas em cima das outras. O acesso às paletes é feito, em cada célula, segundo uma lógica de *Last In / First Out* (última a entrar é a primeira a sair).

- **Drive-in**

Estantes com vários alvéolos de profundidade em cada lado de acesso. Em termos de lógica comporta-se como a Block Staking, só que existe uma estrutura responsável por suportar as paletes. O elemento móvel encarregado da movimentação do material pode penetrar dentro da estrutura da estante para aceder às paletes do seu interior.

- **Powered Mobile**

Este tipo de estante permite um acesso aleatório ao material, como no caso das APR. Distingue-se destas porque possui um mecanismo de afastamento de certas colunas da estrutura de forma a que o elemento móvel se possa posicionar em qualquer célula do interior.

- **Live Storage**

Estantes deste tipo permitem uma lógica de acesso do tipo *Fist In / First Out* (primeira a entrar é a primeira a sair) apoiadas por um mecanismo que automaticamente desloca a palete para a uma fila de saída. Por vezes esse processo de movimentação interna é assegurado pela força da gravidade, dando origem a estantes inclinadas.

Cada um destes tipos será descrito com mais pormenor numa próxima secção deste trabalho, aquando da apresentação do software do programa de simulação. Entretanto, interessa somente dar uma ideia ao leitor do que é considerado uma *célula* nos diversos tipos de estantes. A próxima figura pretende resumir esta

ideia, mostrando as três formas possíveis de *célula* correspondentes aos tipos de estante atrás considerados:

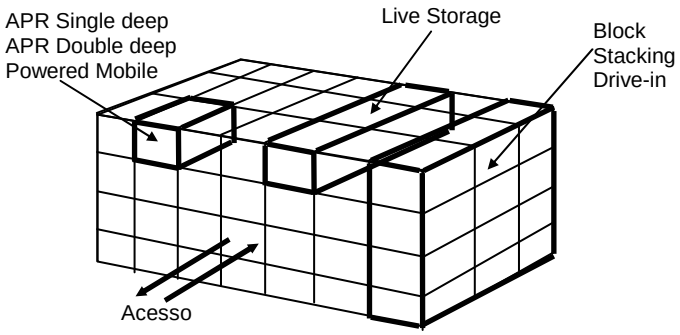

Fig. 3.3 Conceito de célula para os vários tipos de estantes.

3.1.3 Vias de movimentação

Associados a certos tipos de elementos móveis, principalmente aos veículos, é comum definirem-se determinados percursos físicos dentro de um armazém automático, através dos quais esses elementos se poderão deslocar sempre que for necessário. A esses percursos deu-se o nome de *vias de movimentação*. Num armazém real, são geralmente usados para esse efeito troços de carris, calhas servindo de guias, fitas ou riscas de tinta magnética para apoio ao movimento de veículos guiados automaticamente, etc., podendo esses percursos apresentar características diversas, conforme o tipo de veículos a que se destinam. No entanto, o conceito de *via de movimentação* servirá aqui para englobar todos esses casos de forma a permitir construir um modelo de simulação apoiado numa só *entidade* orientadora do movimento. Não se farão distinções entre carris, fitas, riscas, ou qualquer outro mecanismo de orientação do movimento dos veículos, designando-os em geral por *vias de movimentação*.

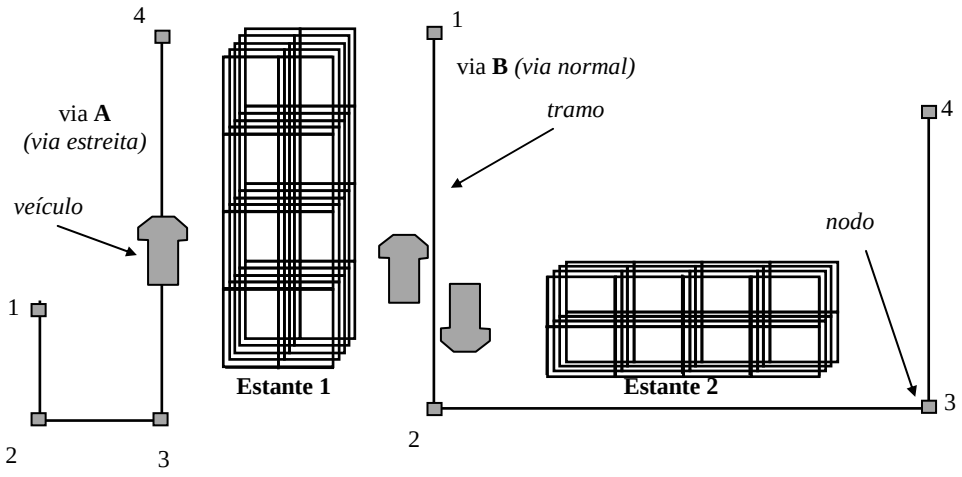

Fig. 3.4 Exemplo de vias de veículos e respectivos tramos e nodos

As *vias* formam, no seu conjunto, uma espécie de rede por onde se efectuam determinadas operações de transporte de material, ou mesmo movimentos de outra espécie, como a recolha de um veículo ao seu parque de estacionamento.
Na construção à escala de um modelo de armazém, considera-se uma *via* um conjunto de segmentos de recta consecutivos definidos pelo projectista de uma só vez, pretendendo representar um possível percurso para determinados elementos móveis (fig.3.4).
Assim, cada *via* é constituída por uma sucessão de *tramos* ligados entre si por pontos de contacto a que se chamam *nodos*.
Para que fosse possível modelar uma grande parte dos casos reais, principalmente no que respeita à movimentação de veículos guiados automaticamente, considerou-se a existência de dois tipos fundamentais de *vias*:

- a *via estreita*, que somente pode ser ocupada por um veículo de cada vez, tendo os outros possíveis interessados que esperar pela libertação da via,

- e a *via normal*, que permite o acesso a vários veículos ao mesmo tempo movimentando-se estes lado a lado.

3.2 Elementos físicos móveis

Os *elementos móveis* são os responsáveis pela transferência de material entre as diversas zonas do armazém, sendo, por isso, elementos de grande importância no que respeita à velocidade de resposta do sistema como um todo. Bons desempenhos, tais como taxas de saída elevadas, dependem não só do traçado das vias de movimentação e das soluções de armazenamento, mas também, e em grande parte, do tipo de *elementos móveis* escolhidos.
A EFACEC, como detentora de uma vasta experiência neste campo, classifica os vários *elementos móveis* segundo os seguinte grupos:

- **Veículos**
- **Transportadores periféricos**
- **Diversos**

Esta classificação resulta das diferenças dos vários mecanismos de movimentação e também das funções a que esses elementos se destinam. Enquanto que o grupo dos *transportadores periféricos* se dedica, em geral, ao transporte de material dentro de um percurso fixo, não tendo acesso directo às estantes, os *veículos* são elementos mais versáteis, uma vez que possuem a faculdade de se movimentar numa malha de percursos e escolher, para um caso particular, o melhor percurso a seguir. Por outro lado, grande parte dos veículos possui acesso directo às estantes, o que os torna elementos essenciais num armazém automático.
O grupo dos *diversos* engloba certos elementos cuja actividade é mais específica, saindo fora dos dois restantes grupos.

3.2.1 **Veículos**

Neste grupo de elementos (fig.3.5) consideram-se os *Stocadores*, que são muitas vezes elementos de grande porte deslocando-se em calhas ou carris ao longo das zonas de armazenamento, os 'Automated Guided Vehicles' (*AGVs*), que se guiam por eles próprios seguindo determinados traçados com a ajuda de um sistema de detecção automático do percurso, e os *Transfers*, que são estruturas responsáveis por fazer deslocar outro tipo de veículos de um sitio para outro. Os *Transfers* são muitas vezes usados para mudar os *Stocadores* de um carril para outro, permitindo-lhes aceder a diferentes zonas de armazenagem.
Apesar de certos tipos de *AGVs* (os *AGVs Stocadores*) poderem cumprir tarefas de colocar e retirar material das estantes, é mais vulgar reservar esse trabalho para os *Stocadores*, pois tratam-se de estruturas mais precisas no que respeita ao posicionamento e de maior robustez, podendo ser usados tanto em instalações de pequeno porte como de grande dimensão. Para além disso, os *Stocadores* apresentam maiores velocidades de movimentação, o que os torna preferíveis aos *AGVs* em grande parte dos sistemas de armazenamento reais. A principal desvantagem deste tipo de veículos reside no facto de necessitarem do apoio de um maior número de periféricos para fazerem transportar o material de um ponto ao outro do armazém. Os *AGVs* são usados, ou em pequenas instalações onde o espaço livre é escasso, ou nos casos em que a estrutura do armazém não permita a instalação de *Stocadores*.

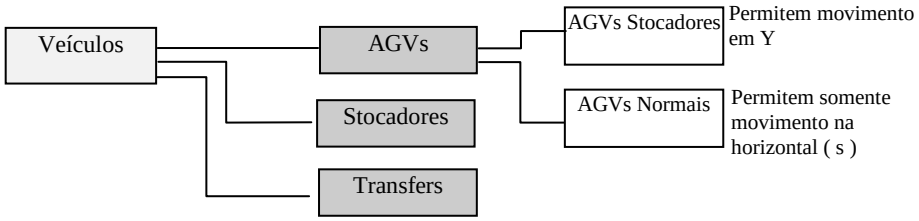

Fig. 3.5 Classificação dos *veículos.*

A este tipo de elementos é costume associar-se um sistema de eixos através do qual se definem as suas possibilidades de movimentação (fig.3.6). Assim, qualquer *Stocador* permitirá movimentos tanto no plano horizontal (coordenada s), como segundo a altura (coordenada y), assim como segundo a profundidade das células das estantes (movimento em z). Apesar de o mesmo acontecer no caso do *AGV Stocador*, no caso do *AGV* normal só serão possíveis movimentações de material no plano horizontal.

No que respeita ao modo de transferência de carga, isto é, ao tipo de mecanismo que esse elemento usa para colocar ou retirar material de uma célula, podem-se referir os mecanismos baseados em garfos telescópicos e os por extracção, no entanto, na simulação usar-se-ão somente os respectivos *tempo de carga* e *tempo de descarga*, tornando estes processos transparentes do nosso ponto de vista.

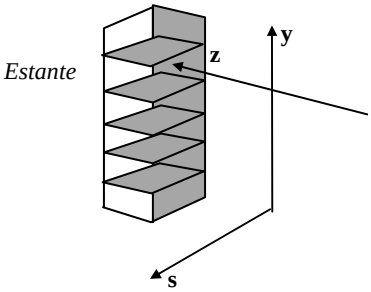

Fig. 3.6 Direcções de movimentação de material.

Deve referir-se ainda que as características de arranque e de paragem destas unidades móveis, definidas por uma aceleração de arranque, uma velocidade de funcionamento normal, uma aceleração de travagem e uma velocidade lenta de aproximação, serão nesta fase do trabalho, e em quaisquer elementos móveis, substituídas por uma *velocidade média* de movimentação.

3.2.2 Transportadores periféricos

Incluem-se neste grupo de elementos o *Vaivém*, espécie de veículo para transporte horizontal que possui uma *via de movimentação* dedicada, os *Transportadores contínuos*, dos quais se podem referir o tapete de correntes, o tapete de rolos, as mesas rotativas e as mesas de transferência ortogonal, e na secção dos *Diversos* outros elementos menos comuns. Designam-se transportadores periféricos porque são elementos que não têm acesso directo às estantes onde o material é armazenado, funcionando somente como interface entre os veículos o certas zonas da "periferia" do armazém.

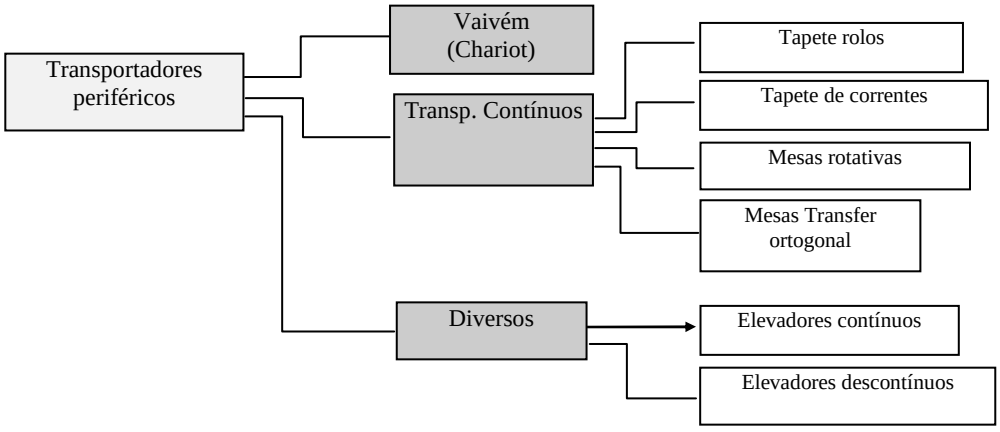

Fig. 3.7 Transportadores periféricos.

Na maior parte dos casos, o *Vaivém* funciona como se fosse uma mesa de transferência móvel, isto é, admitindo material num determinado posto e deslocando-o dentro de um percurso fixo para depois o depositar num posto

diferente. A direcção de transferência de material é, em noventa por cento dos casos, perpendicular à direcção de movimentação. Na próxima figura pretende-se exemplificar uma movimentação de material típica de um *Vaivém*. O material é transferido do *Tapete1* para o *Vaivém* segundo a perpendicular à direcção de deslocamento deste. O *Vaivém* transporta o material até ao *Tapete3* e aí o descarrega, novamente segundo a perpendicular ao seu percurso.

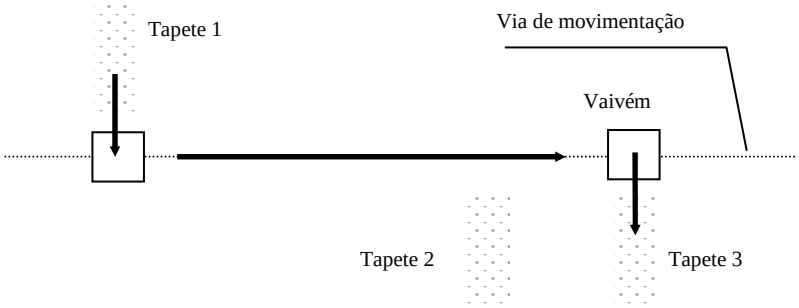

Fig. 3.8 Exemplo de percurso de material entre dois tapetes utilizando um *Vaivém*.

No que respeita aos tapetes, há que considerar a diferença entre *tapetes de rolos* e *tapetes de correntes*. Os *tapetes de rolos* são constituídos por rolos móveis por cima dos quais as paletes de material deslizam, enquanto que nos *tapetes de correntes* as paletes se fazem deslocar assentes em correntes móveis de comprimento igual ao comprimento do tapete. Por este facto, um *tapete de rolos* poderá também funcionar como um tapete de acumulação[(1)] , dependendo do número de rolos com movimento independente, enquanto que no *tapete de correntes* é impossível este tipo de comportamento. Apesar de qualquer tapete limitar a movimentação de material a uma única direcção, em alguns casos é permitida a circulação nos dois sentidos. No entanto, funcionando como tapete de acumulação, só um sentido é utilizado para esse efeito.

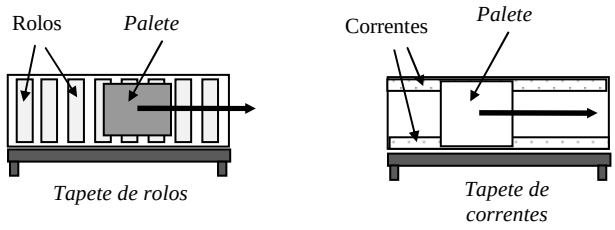

Fig. 3.9 *Tapete de rolos e tapete de correntes*.

As *mesas* são elementos usados ou para alterar a direcção da movimentação do material, para depois o transferir para outra entidade, ou para rodar a palete de forma a posicioná-la correctamente antes de ser transferida. A *mesa de*

(1) Enquanto o tapete não puder descarregar, as paletes vão-se acumulando encostando-se umas às outras no fim do tapete.

transferência ortogonal é normalmente usada para levar a cabo o primeiro tipo de tarefas, enquanto que para rodar a palete se costumam usar *mesas rotativas*.
As *mesas de transferência ortogonal* assemelham-se a um pequeno tapete que, para além do movimento horizontal, possui a capacidade de se mover na vertical dentro de um curso de alguns milímetros. Encontram-se muitas vezes associadas a *tapetes de correntes*, instaladas no seu interior. Quando a palete chega a esta mesa, ela eleva-se um pouco (segundo **z**, fig.3.6) deslocando a palete do *tapete de correntes*, o que permite alterar o sentido de circulação do material assim que for activado o seu mecanismo de transferência (fig.3.10).

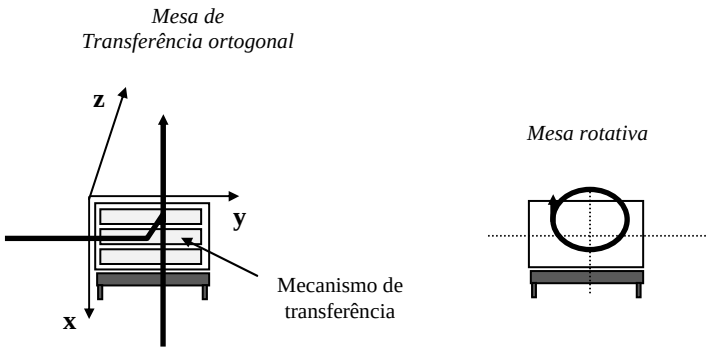

Fig. 3.10 Circulação do material. *Mesa de transferência ortogonal* e *Mesa rotativa*

Para além dos elementos já referidos nesta secção, podem considerar-se pertencentes ao grupo dos **diversos** os *elevadores contínuos*, cujo mecanismo de funcionamento se assemelha ao descrito para a *mesa de transferência ortogonal* (apresentando, no entanto, um curso vertical da ordem dos metros), e os *elevadores descontínuos*, muitas vezes em forma de carrossel, possuindo um mecanismo semelhante ao de uma nora. No entanto, tratando-se este último de um elemento pouco usado em instalações reais, não nos preocuparemos, para já, com a sua modelização.

3.2.3 Diversos

Neste grupo incluem-se os *elementos móveis* de utilidade mais específica. Podem-se referir o *ponto de identificação*, local onde o material poderá parar para se proceder à sua identificação (leitura de códigos de barras, colocação de etiquetas, etc.), e outros elementos relacionados com a movimentação, distribuição e aglomeração de paletes vazias, elementos esses que nesta fase do trabalho não serão considerados importantes do ponto de vista da simulação.

4. Concepção de aplicações usando o *Microsoft Visual C++*

4.1 Considerações gerais

Perante o crescente aproximar das actuais ferramentas de programação aos conceitos de funcionamento da realidade, o que torna possível a análise de um sistema do ponto de vista de objectos, pretende-se nesta secção introduzir o leitor ao modo como é estruturada uma aplicação em *Visual C++*, assim como apresentar as principais facilidades incluídas na *Microsoft Foundation Class* (*MFC*), biblioteca que permite desenvolver uma rápida e versátil interface com o utilizador e, ao mesmo tempo, usar uma larga variedade de objectos predefinidos.

O conceito de objecto, tal como o temos da realidade à nossa volta, permite estruturar de uma forma mais ordenada as várias *entidades* de um sistema, assim como desenvolver uma estrutura de programa onde as responsabilidades passam a ser distribuídas, apesar de se manterem hierarquizadas. Cada objecto é responsável pelos seus próprios métodos e pelas suas próprias variáveis, mantendo coesa a sua estrutura de dados e permitindo aos outros a mesma individualidade. Para além disso, os objectos podem ser criados e aniquilados durante a execução do programa, fornecendo ao projectista uma forma poderosa de análise de sistemas.

O exemplo mais distinto de distribuição de responsabilidades é o conceito de *Serialização*, introduzido pelo *Microsoft Visual C++*. A determinado tipo de objectos é permitida a *Serialização*, isto é, que sejam eles próprios a guardar os seus dados em disco ou a autoconfigurarem-se aquando da leitura dos dados de um ficheiro. Não existe necessidade de escrever qualquer tipo de código para tornar activo este sistema, mas somente indicar quais as variáveis desse objecto que se devem *Serializar*.

Tal como acontece na realidade, não é necessário interferir-se nos processos da responsabilidade desse objecto, pois ele está preparado para levar a cabo as suas tarefas, tal como para conduzir um automóvel não necessitamos de nos preocupar com controlar a quantidade de combustível injectado ou de medir continuamente a temperatura do motor.

A gestão da comunicação entre os vários objectos passa a ser a tarefa mais importante neste tipo de abordagem, permitindo aproximar a estrutura de um modelo de simulação ao funcionamento de um caso real.

A *Microsoft Foundation Class* (*MFC*) é uma ferramenta paralela ao *C++* vocacionada para o desenvolvimento de aplicações para Windows95/NT e outras plataformas que suportem *Win32 API* (grupo de rotinas usadas pelas aplicações para comunicarem com um sistema operativo de 32 bits). Trata-se de um grupo de classes implementadas em *C++*, algumas das quais representam objectos conhecidos, tais como *Janelas*, *Janelas de diálogo*, '*toolbars*', etc.

Por de trás de todas as classes (ou objectos) da *MFC* existe uma classe designada por *CObject* de onde as outras derivam, e a partir da qual se podem construir novos objectos. A grande importância desta classe reside no facto de ela implementar mecanismos de extrema utilidade, como o suporte de *Serialização*, informação

sobre a classe e diagnóstico do objecto. Daí que a grande parte dos objectos definidos neste trabalho possuam como base esta classe.

4.2 Estrutura básica de uma aplicação

O esqueleto de uma aplicação para *Windows* em *Visual C++* é constituído por três tipos de objectos diferentes: a janela de suporte (‘*Frame’*), o Documento (*Documento*) e a vista sobre o documento (‘*View’*). A ‘*Frame’* principal é o suporte do menu principal da aplicação, da ‘*ToolBar’* e da ‘*StatusBar’*, sendo também a mãe da janela da *‘View’*. O *Documento* guarda a informação sobre os dados associados à aplicação, e a *‘View’* apresenta os dados referentes a uma determinada secção do *Documento* de uma forma visual, respondendo também às acções do utilizador.

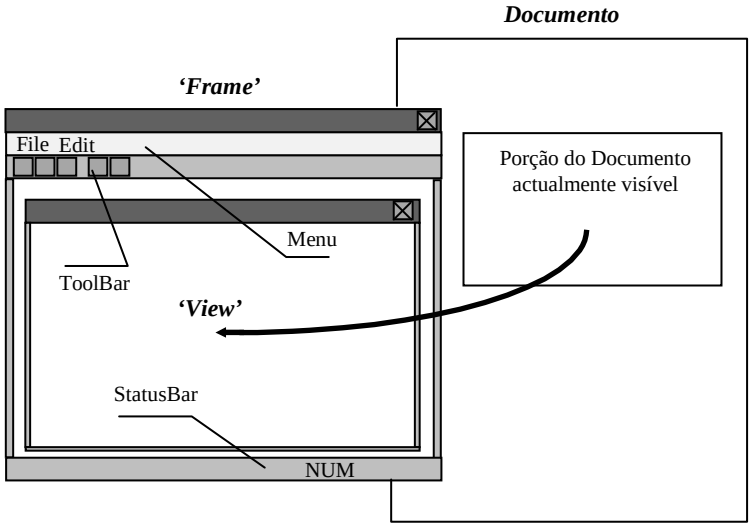

Fig.4.1 Esqueleto de uma aplicação em Visual C++.

Apesar de ser esta a estrutura básica de uma aplicação, na maior parte dos casos este esquema torna-se mais complexo, uma vez que se podem criar várias ‘*Views’* para o mesmo *Documento* e incluir outros tipos de janelas, menus e objectos. Grande parte da interface do utilizador é conseguida através da criação de *Janelas de diálogo*, através das quais determinados dados podem ser introduzidos na estrutura do *Documento* ficando disponíveis a toda a aplicação.
Existem duas filosofias básicas de implementação de aplicações: a designada por *Multi-Documento*, que permite numa mesma aplicação manejar documentos diferentes, como é o caso do *Microsoft Word*, por exemplo, e a *Documento Único*, em que a aplicação é inteiramente dedicada a um só documento. Na aplicação de simulação que se está a desenvolver optou-se por seguir a filosofia de *Documento Único*.

5. Estrutura da aplicação ARMAZÉM

5.1 Esquema geral

A aplicação que está em desenvolvimento foi baptizada com o nome ARMAZEM, e tem como objectivos, em primeiro lugar, permitir ao utilizador desenhar o seu armazém à escala, incluindo a definição de áreas, o posicionamento das *estantes*, o traçado de *vias de movimentação*, e em geral, a criação de todos os objectos existentes dentro de um armazém automático. Em segundo lugar, esta aplicação pretende vir a ser usada como uma ferramenta de simulação, permitindo uma visualização do funcionamento do modelo do armazém previamente criado pelo utilizador.
Por não ter sido considerado útil possuir dois armazéns abertos ao mesmo tempo na mesma aplicação, e ainda por uma questão de simplicidade de processamento na simulação, optou-se por só permitir simular um modelo de armazém de cada vez, o que levou a implementar o programa segundo uma filosofia de *Documento Único*.
Para além do esqueleto normal de uma aplicação genérica de *Visual C++*, o ARMAZEM engloba um conjunto de objectos que pretende representar os diversos elementos de um armazém, assim como *Janelas de Diálogo* através das quais se permite o acesso aos dados desses objectos e à configuração do sistema. Para além disso, foi incluído um novo objecto responsável pelo processo de simulação, designado por *Simulador*, ou *ArmSimulation* (fig.5.1).

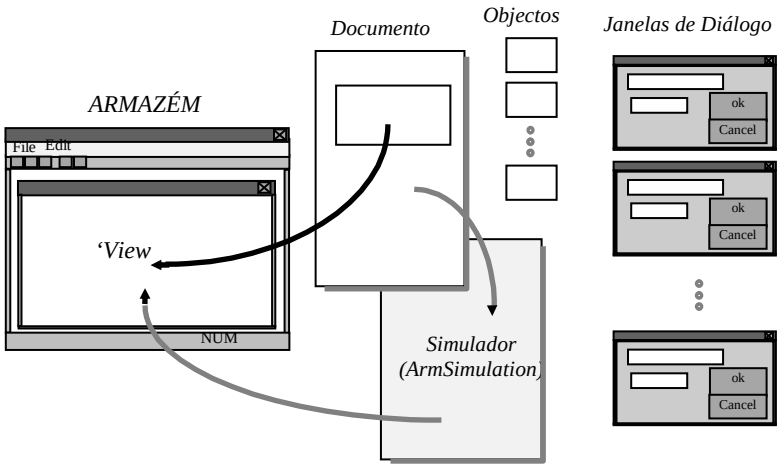

Fig. 5.1 Estrutura da aplicação ARMAZÉM em Visual C++.

Durante a tarefa de representação do armazém à escala, todo o processo é controlado pela *'View'* e pelo *Documento*, encontrando-se inibida a actividade do simulador (*ArmSimulation*). O operador escolhe os diversos objectos através de opções do menu, especificando as suas características nas *Caixas de diálogo* relacionadas com esses objectos. É, por exemplo, o caso da criação de um veículo. Quando se tratam de elementos cujas dimensões ou forma não podem ser predeterminadas pelo programa, o operador desenha-os directamente sobre a *'View'*, com a manipulação do "rato", sendo estes depois transformados em objectos e guardados no *Documento* para futura utilização. Assim, à medida que o operador vai criando as várias

entidades do armazém, o conteúdo do *Documento* cresce, pois é este o responsável por guardar toda essa informação. O *Documento* mantém o conhecimento desses objectos guardando os respectivos apontadores numa *listas de apontadores para objectos* (fig.5.2). Este tipo de listas fazem parte de uma das classes predefinidas na *MFC*. As *estantes* são inscritas *na Lista de Estantes*, os *veículos* na *Lista de Veículos*, os *tapetes* na *Lista de Tapetes*, etc. Ao mesmo tempo, e porque a *'View'* necessita dessa informação para se auto-redesenhar, o *Documento* mantém ainda uma lista de apontadores para todos os objectos susceptíveis de serem representados no ecrã. Quando a *'View'* necessita de refazer a informação visual, percorre os objectos dessa lista e desenha-os nos seus devidos lugares. Esta tarefa é levada a cabo por um método da *'View'* que se chama `Draw()`.

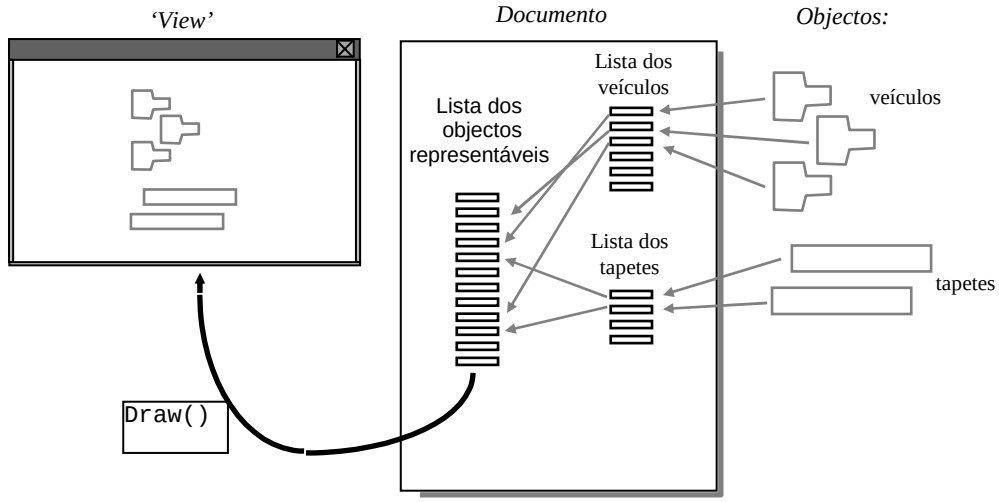

Fig. 5.2 Exemplos de listas de objectos e sua aplicação na *'View'*.

O *Documento* é também responsável por despoletar a *Serialização* dos dados da aplicação, entre eles os que dizem respeito aos diversos objectos. No processo de gravação desses dados no disco, o *Documento* dá ordem de *Serialização* às suas listas de objectos e estas, por sua vez, transmitem essa ordem a cada um dos seus objectos, fazendo com que eles se inscrevam no disco. Trata-se de um processo em cadeia, partindo do *Documento* e acabando no último elemento do objecto mais remoto. É desta forma que se grava num ficheiro o 'layout' do armazém previamente definido pelo operador. Ao abrir-se um documento guardado no disco, o mesmo processo é despoletado, só que agora são os próprios objectos que se criam, preenchendo a sua estrutura de dados com a informação recolhida do ficheiro.

Logo que o operador decide colocar a aplicação em modo de simulação, passa a ser o *Simulador* o responsável pelo processamento. As listas de objectos do *Documento* são passadas ao *Simulador*, podendo este manejá-las como entender para levar a cabo o processo de simulação. Desta forma, ao *Simulador* fica acessível a informação sobre as diversas entidades do armazém definido pelo utilizador, tornando possível movimentar veículos, deslocar material nos tapetes, retirar ou colocar paletes nas estantes, etc. Para representar visualmente o movimento de

certas entidades, e no geral para mostrar o estado do armazém que se está a simular, o *Simulador* tem também possibilidades de comunicar com a *'View'*, quer seja indirectamente através dos objectos, quer por intermédio de acções dirigidas directamente para a *'View'*.

5.2 Estrutura do Simulador

Sendo o *Simulador* o responsável máximo de todo o processo de simulação, decidiu-se estruturá-lo tentando seguir a filosofia de organização usada nos sistemas informáticos de controle e gestão da EFACEC. Não se trata de um esquema de organização geral, pois a própria EFACEC o altera conforme a especificidade da instalação a que se destina, no entanto, pretende-se com isso conseguir maximizar o paralelismo de conceitos entre as soluções informáticas utilizadas por esta empresa no controle de armazéns e a estrutura de funcionamento do *Simulador*. Uma organização que aproxime a realidade e a simulação permitirá um melhor entendimento do modelo, assim como facilitará a sua concepção e a sua adaptação a diferentes situações. Para além disso espera-se, mais tarde, poder substituir certos blocos do *Simulador* por os seus equivalentes do sistema informático real, o que permitirá misturar simulação com realidade e usar este software numa perspectiva mais abrangente.
Em termos de lógica, podem-se considerar num armazém automático os seguintes níveis (fig.5.3): lógica de *Objecto*, lógica de *Controle* e lógica de *Gestão.*

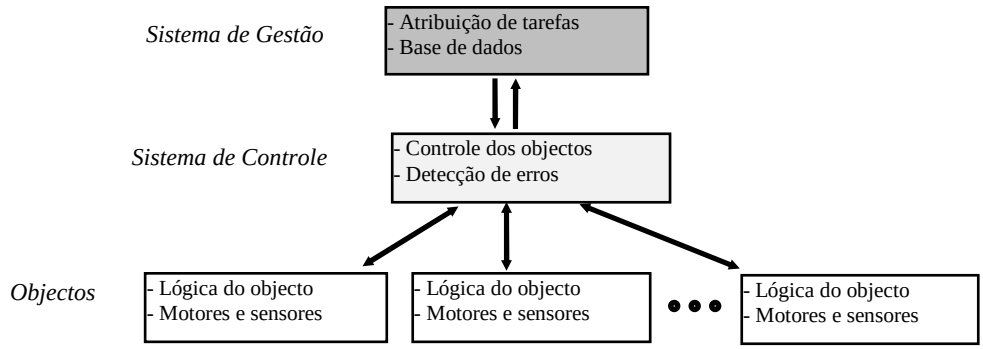

Fig. 5.3 Hierarquia de lógica dentro de um armazém automático.

O nível mais baixo corresponde à lógica encapsulada nos objectos, isto é, o seu modo de funcionamento como elementos isolados de um sistema. Como exemplo, considere-se um tapete. Quando se liga, ele anda, transportando automaticamente o que se encontrar sobre ele, e quando se desliga, ele pára. Esta é, em primeira análise, a lógica associada ao tapete.
O nível de lógica a seguir diz respeito à interligação entre os diversos objectos, assim como aos critérios sobre a transferência de informação entre eles. Representa, no fundo, a lógica de um controlador. É aqui que se resolvem as questões relacionadas com o mecanismo dos objectos como um todo, mandando parar um veículo porque outro se encontra no seu percurso, accionando o movimento de um

tapete porque está livre o tapete da frente, etc. Chamemos a este nível o *Sistema de Controle*.
No topo da hierarquia, existe a lógica de mais alto nível, relacionada com a comunicação com o exterior, com a atribuição de tarefas e com a organização geral da base de dados do sistema. Chamemos-lhe *Sistema de Gestão*.
No fundo esta acaba por ser a organização adoptada pela EFACEC, podendo, no entanto variar de caso para caso a sua implementação.
Para o desenvolvimento do *Simulador* optou-se por manter este tipo de estrutura, considerando para isso três módulos básicos (fig.5.4):

- ***SimWindow***, que deverá corresponder ao espaço físico do armazém e que contém, por isso, a informação referente às entidades aí existentes (tapetes, estantes, vias, veículos, etc.),

- ***SimControl***, que é o módulo ao qual se pode fazer corresponder o *Sistema de Controle* do armazém e cuja função é conter a lógica dos diversos processos envolvidos com a finalidade de gerar os comandos apropriados conforme os estados das diversas entidades,

- ***SimMaster***, que corresponderá ao *Sistema de Gestão* do armazém e que comunica com o *SimControl* através de comandos de alto nível. Este bloco é responsável por manter a informação sobre os produtos, ordens de entrada e de saída, e outras e, para além disso, gerar tarefas para serem administradas pelo *SimControl* e executadas pelos diversos elementos móveis do armazém.

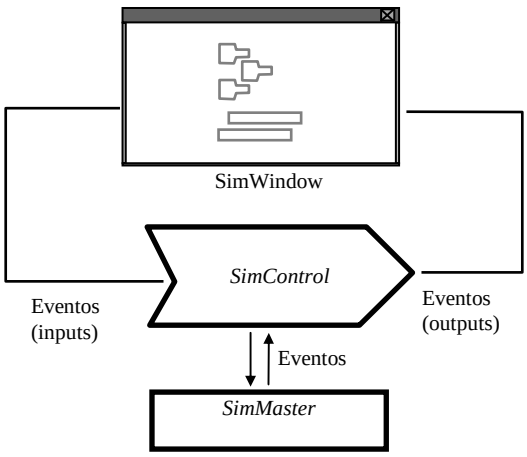

Fig. 5.4 Estrutura do Simulador.

Tendo em conta esta estrutura, e desde que se assegure um eficiente mapeamento entre as entradas e comandos do *Sistema de Controle* e os eventos de input e de output do módulo *SimControl*, assim como uma eficaz correspondência entre as mensagens de comunicação *Sistema de Gestão* / *Sistema de Controle* e os respectivos módulos *SimMaster* / *SimControl*, para além da função de simulação pura, espera-se ser possível ligarem-se os dois sistemas de forma a substituir-se a actividade do *Sistema de Gestão* pelo seu equivalente de software. Assim, tornar-se-á possível testar o comportamento de uma instalação real em que o *Sistema de*

Gestão é substituído pelo *SimMaster*, ou usar o *Sistema de Gestão* real para testar o 'layout' de um armazém "virtual" simulado através do *SimWindow*, permitindo, neste caso, fazer previsões sobre a futura performance da instalação real. Estes, no entanto, são objectivos futuros que não farão parte do trabalho aqui descrito.
Para além da lógica envolvida no *SimMaster* e no *SimControl,* é importante recordar que existe em cada entidade física pertencente ao armazém uma lógica inerente e exclusiva dessa entidade. Por exemplo, um tapete deverá ser suficientemente "inteligente" para transportar as paletes que se encontrem em cima dele sem a interferência dos outros níveis de lógica, assim como um AGV se deverá deslocar no seu percurso, alterando, se necessário, o sentido do seu movimento.
Se bem que a estrutura do *Simulador* se possa considerar bastante bem definida nesta fase do projecto, em termos de implementação o trabalho aqui descrito refere-se principalmente à modelização do comportamento de cada entidade, o que significa estabelecer as propriedades de cada objecto particular (atributos) e descrever o respectivo modo de funcionamento através de métodos que englobem a sua lógica específica. No entanto, para que o funcionamento desses elementos pudesse ser testado pelo programa, teve que ser assegurada uma funcionalidade mínima dos blocos *SimControl* e *SimMaster*, o que permitiu visualizar no ecrã alguns casos de movimentação de paletes e da sua transferência entre diferentes elementos do armazém.
A razão de se ter começado pela modelização dos objectos do armazém, situando-nos no nível mais baixo de lógica, é porque a este nível se podem considerar os objectos independentes uns dos outros, assim como independentes do tipo de armazém, e, sendo assim, o seu comportamento poderá ser generalizado para qualquer tipo de instalação. Trata-se de criar uma espécie de biblioteca dos objectos correntemente utilizados num armazém, através da qual o utilizador escolhe uma solução para o seu caso particular.
Já aos níveis do *Sistema de Controle* e do *Sistema de Gestão* as coisas tornam-se um pouco diferentes, uma vez que a lógica aí envolvida depende das características da instalação em questão, o que nos leva a ter de meditar com maior profundidade no assunto. Desenvolver um aplicação que simule um 'layout' específico não apresenta dificuldades de maior, no entanto, o problema torna-se mais complexo quando se pretende dar a facilidade ao operador de representar e simular um grande número de 'layouts' diferentes. As dificuldades residem, essencialmente, na modelação do *Sistema de Controle* e do *Sistema de Gestão* de forma a garantir a versatilidade suficiente para que o próprio utilizador possa decidir sobre quais os critérios de decisão usados nesses níveis de lógica. Apesar de ainda não ter sido desenvolvido qualquer trabalho neste campo, apontam-se para o futuro soluções baseadas numa linguagem de interface entre o sistema real e o software de simulação, assim como a implementação de uma "*base de dados*" de critérios que possam ser escolhidos pelo utilizador.

6. Modelação dos elementos de um Armazém

Nesta secção explica-se o modo como estão a ser modelados os elementos de um armazém automático usando uma filosofia de programação por objectos. A separação que aqui se faz entre *elementos físicos* e *elementos conceptuais* deve-se ao facto de os primeiros serem elementos representáveis no ecrã, o que implica estarem relacionados com a parte gráfica do programa, enquanto que os segundos, tratando-se de elementos que representam ideias ou ordens, não necessitam de ser tratados graficamente. Decidiu-se começar pelos primeiros, pois a sua concepção interfere com a própria concepção da interface gráfica com o utilizador.

6.1 Elementos físicos

A primeira tarefa que este programa deve permitir ao utilizador é a representação à escala do 'layout' do armazém a simular. Tratando-se de um programa de simulação visual, é importante que essa representação seja o mais próxima possível da realidade, pois o utilizador poderá assim conceber com maior precisão o seu 'layout', gerindo os espaços e experimentando as mais variadas soluções de armazenagem e transporte de material.
Do ponto de vista da sua representação à escala, os *elementos físicos* de um armazém são aqui vistos como elementos que ocupam uma determinada área (ou espaço) dentro do próprio armazém. Por isso se decidiu iniciar a modelação dos diversos objectos tendo como base as características representativas de uma área. Assim, a área é entendida como um objecto (ou classe) básico, a partir do qual os outros poderão ser representados.
Durante a execução do programa, todos os *elementos físicos* serão representados em coordenadas relativas a um sistema de eixos incluído no espaço reservado na *'View'* para a representação do armazém. Esse espaço é definido inicialmente pelo operador no momento em que decide dar início à criação de um novo armazém (fig.6.1).

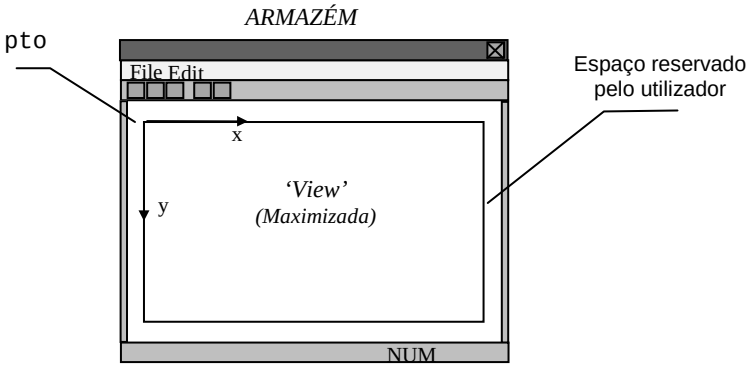

Fig. 6.1 Espaço da *'View'* reservado à representação do armazém.

É ainda de referir que todos os objectos serão definidos com base numa hierarquia de propriedades, típica das linguagens orientadas para objectos, o que, na maior parte dos casos, implicará a herança de características de outros mais simples. O

mesmo será dizer que a funcionalidade de um determinado objecto será dividida em diversos níveis, correspondendo cada nível à funcionalidade de uma determinada classe. Consideremos, como exemplo, o objecto *Estante* (fig.6.2). Neste caso, poder-se-ão considerar dois níveis de funcionalidade (ou de lógica, se quisermos): ao nível mais baixo encontram-se as características que representam a estante como uma área ocupada do armazém, área essa que poderá ser movimentada, eliminada, copiada, etc. Este nível corresponde então à funcionalidade da classe *Area*. No segundo nível englobam-se as características de armazenamento, o tipo de estrutura, a lógica de acesso ao material, etc. O objecto *Estante* será então definido pela reunião de estes dois níveis de informação.

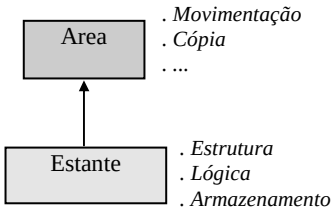

Fig. 6.2 Objecto Estante como a reunião de dois níveis de informação .

Não se pretende neste trabalho fazer uma descrição pormenorizada do código do programa com que cada classe foi implementada, pois isso resultaria num volume de informação demasiado extenso, pretendendo-se unicamente descrever os conceitos existentes por de trás dessa implementação. No entanto, em casos de grande interesse, poder-se-á recorrer a esse meio para melhor expor os assuntos.
A criação, por parte do utilizador, da grande maioria dos *elementos físicos*, é levada a cabo através das facilidades de desenho implementadas no objecto *'View'*. Nestas ferramentas incluem-se o controle dos movimentos e acções do "rato", o efeito de ZOOM, o posicionamento do desenho por 'scroll' da janela, e, em geral, todas as funções básicas normalmente disponíveis num programa de desenho vectorizado. Para que o operador não cometa determinado tipo de erros, tais como tentar desenhar um objecto numa zona já ocupada por outro, foi ainda incluído na *'View'* um processo de filtragem[(1)] de certos movimentos do "rato", activado quando o programa se encontra em modo de desenho.
Comecemos então por esquematizar o conteúdo de um armazém, de tal forma que essa informação possa ser transferida para uma linguagem orientada para objectos.

6.1.1 A classe *Area*

O intuito de criar esta classe foi incluir num só tipo de objecto os atributos e os métodos necessários à manipulação de uma área. Do ponto de vista do programa, *Area* mais não é do que um objecto definido por um conjunto de linhas consecutivas, variáveis que representam os atributos desse conjunto de linhas e funções específicas (métodos) que permitem a sua manipulação. Esta classe pretende englobar diversos tipos de áreas dentro do armazém, desde as

(1) Algoritmo baseado na intercepção de segmentos de recta utilizado por *A.C.Brito* no seu trabalho de Doutoramento.

ocupadas pelas *estantes* até às definidas pelos *veículos*, diferenciando-as umas das outras através de um parâmetro interno designado por `m_tipo`. Este parâmetro deve ser interpretado pelo sistema como representando o tipo do objecto *Area*, e é definido no momento em que o utilizador escolhe a opção de desenho do respectivo objecto. Para além deste atributo, esta classe inclui parâmetros de posicionamento do objecto relativamente à origem das coordenadas do armazém, a escala com que as suas dimensões foram criadas (em metros/pixel), a sua cor, as coordenadas do seu centro, e outros. Ainda, para permitir usar as facilidades da *Serialização*, esta classe foi derivada da classe *CObject* definida na *MFC* do *Microsoft Visual C++*.

Passe-se à apresentação dos membros da classe *Area*.

ATRIBUTOS:

`m_tipo;`	Tipo de área que o objecto representa.
`m_nPenWidth;`	Espessura da linha do objecto.
`m_color;`	Cor do objecto.
`m_selectON;`	*Flag* que indica se o objecto está (ou não) seleccionado.
`m_pointArray;`	Array de pontos que define a área do objecto.
`m_escala;`	Escala a que o objecto foi criado (metros / pixel).
`m_ptCentro;`	Ponto do centro do objecto.
`m_orient;`	Orientação da área (Vertical ou Horizontal).
`m_rectBound;`	Rectângulo que contém o objecto.

MÉTODOS (ou funções membro):

`Area();`	Construtor do objecto
`End();`	Termina a construção do objecto.
`Bound();`	Cria o rectângulo que contêm o objecto.
`Set();`	Coloca o objecto numa dada posição do armazém.
`Move();`	Move o objecto, sem o reposicionar.
`MoveTo();`	Move o objecto e para um determinado ponto do armazém.
`Rotate();`	Roda o objecto para uma determinada direcção.
`Clean();`	Limpa o objecto de pontos supérfluos.
`IsTherePoint();`	Testa se um determinado ponto pertence ao objecto.
`Draw();`	Desenha o objecto.
`ShowNode();`	Mostra os nodos do objecto.
`CopyTo();`	Copia uma área para outra área.
`Serialize();`	Serializa o objecto.

6.1.2 Áreas elementares

Decidiu-se designarem-se por áreas elementares as áreas cuja funcionalidade é representável por um objecto da classe *Area*. Desta forma, uma *estante*, por exemplo, apesar de possuir certas características de área, não é considerada uma área elementar. Normalmente podem existir dentro de um armazém automático quatro espécies de áreas elementares, a saber:

. Área limítrofe do armazém
. Áreas de recepção

. Áreas de Expedição
. Áreas Interditas

A *área limítrofe do armazém* representa os contornos do armazém que se quer representar, dentro da qual as outras áreas se irão posicionar. As *áreas de recepção* definem o espaço reservado a locais de recepção de material, e as *áreas de expedição* representam as zonas onde se situarão os cais de expedição. Por fim, as *áreas interditas* são zonas que não poderão ser consideradas como espaço útil do armazém, representando pilares, espaços reservados a acessos, zonas de arrumos, etc.

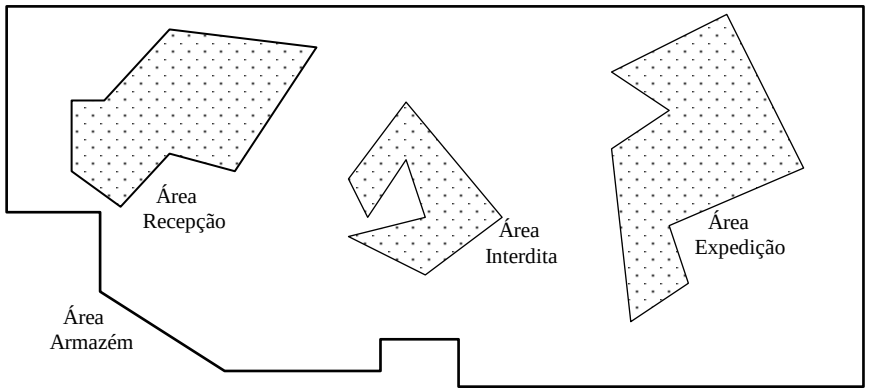

Fig.6.3 Os diversos tipos de áreas elementares .

Cada área deste tipo é criada directamente na *'View'* pelo utilizador através da manipulação do "rato", o que permite representar uma grande variedade de formas, desde que sejam fechadas e que os seus lados não se intersectem. A tarefa de filtrar os movimentos proibidos enquanto se está a desenhar a área é da responsabilidade da *'View'*, tendo sido incluída nos métodos de resposta aos eventos do sistema operativo relacionados com a movimentação do "rato". A área é terminada com um DUPLO-CLIC[(1)] do botão esquerdo do "rato".
Qualquer das áreas aqui referidas é guardada pelo sistema como um objecto simples da classe *Area*, podendo por isso ser manejada pelos métodos desta classe: movimentada, duplicada, eliminada, *serializada*, etc.

6.1.3 Conceito de Palete

Geralmente designa-se por *palete* a estrutura que serve de base de apoio ao transporte de material dentro de um armazém automático, no entanto, esse suporte é outras vezes chamado "palete vazia", o que acabou por inspirar o conceito de *palete* aqui usado. Assim, considera-se uma *palete* a unidade de movimentação de material, unidade essa que depositada dentro de uma *estante* ocupará o espaço de um *alvéolo*. Deste modo, a *palete* compreende tanto a sua base de suporte como o material que acondiciona.

(1) Termo técnico que significa premir duas vezes seguidas.

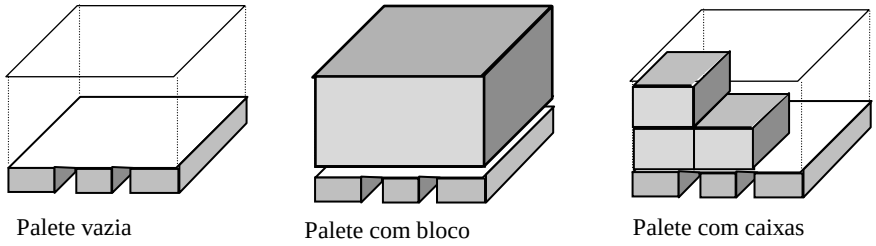

Fig. 6.4 Conceito de palete e seus níveis de ocupação .

Tal como se encontra ilustrado na figura 6.4 podem-se considerar três níveis de ocupação numa *palete*: vazia, com material em bloco e com material em caixas. Para além destes estados de ocupação, para caracterizar uma *palete* é importante referirem-se as suas dimensões, o tipo de material (ou produto) que contém, etc., o que levou à implementação de uma nova classe de objectos. Pelo facto de a *palete* poder ser movimentada através do armazém, faz com que apresente também características da classe *Area*, atrás definida, por isso se derivou o objecto *Palete* do objecto *Area*. A seguir apresenta-se a estrutura desta classe de objectos tal como foi definida em *Visual C++*.

```
class Palete: public Area              //Palete deriva de Area
{
 public:
                  // ATRIBUTOS:

    float         m_largura;           //largura da palete
    float         m_profund;           //profundidade da palete
    BOOL          m_END;               //palete posicionada
    UINT          m_tipoPal;           //VAZIA, BLOCO ou CAIXAS
    UINT          m_maxCaixas;         //máximo de caixas possíveis na palete
    UINT          m_nCaixas;           //n.º actual de caixas
    UINT          m_ref;               //referência da palete
    UINT          m_produto;           //índice do produto

                  // MÉTODOS:

    Palete();//construtor:
    Palete(float escala, float largura, float profund); //construtor:
    virtual void Serialize(CArchive& ar);  //Serialização:

 DECLARE_SERIAL(Palete)
};
```

Esta forma de apresentar os atributos e os métodos de uma classe equivale à anterior descrição feita para a classe *Area*, sendo, no entanto, preferível apresentar a informação desta forma, uma vez que permite ao leitor apreciar a sua implementação no próprio código da linguagem de programação, o que o informa também sobre o tipo das variáveis em jogo.
Repare-se que o método que representa o segundo *construtor*[(1)] recebe o parâmetro `escala` que é a escala do desenho na altura da criação da *palete*.

(1) Método responsável por inicializar o objecto, chamado no momento da criação do objecto.

Apesar de não existir qualquer variável na definição desta classe que receba este parâmetro, ele será passado à parte da *palete* que é inerente à classe *Area*, de onde o objecto deriva, mantendo-se assim o código consistente. A partir desse momento, os métodos da classe *Area* passam a estar acessíveis também para a *palete*, podendo esta ser desenhada, rodada, deslocada dentro do armazém, etc.

6.1.4 Conceito de Célula

Como atrás foi dito, uma *estante* é representada como um aglomerado de *células*, considerando-se uma *célula* o conjunto mínimo de *alvéolos* contíguos que admite o mesmo tipo de lógica de acesso ao material. Tratando-se mais de um elemento conceptual do que de uma realidade, este objecto não necessita de ser derivado da classe *Area*. Aliás, qualquer *célula* estará sempre relacionada com outro objecto real, o que dispensa a sua representação gráfica, reservando-se essa tarefa ao objecto que a contenha.
As verdadeiras funções de uma *célula* são conter um determinado número de *paletes*, permitir a sua manipulação e sustentar um tipo específico de lógica de acesso. As suas dimensões não interessam, uma vez que elas serão definidas tanto pelas dimensões dos *alvéolos* como pelas das *paletes* que contém. Interessa, no entanto, permitir a este objecto as facilidades da *serialização*, o que o obrigou a descender da classe *CObject*. Na sua estrutura foi também incluída uma *lista de paletes*, objecto a partir do qual se terá acesso ao interior da *célula* e onde outros elementos da simulação poderão colocar ou retirar *paletes* (fig.6.5).

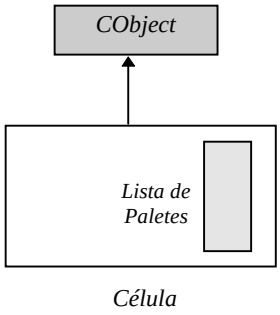

Fig.6.5 Estrutura do objecto *Célula*.

No que respeita à lógica de acesso ao material, foi de novo considerada a existência de três tipos de *células*, tal como atrás se havia considerado na secção em que se falou das *estantes*. Por isso se optou por caracterizar o tipo de acesso à célula mantendo a terminologia que foi usada no caso das estantes. Assim, uma *estante* do tipo 'Live Storage' é constituída por *células* do tipo 'Live Storage', uma 'APR single deep' por *células* 'APR single deep', e o mesmo se poderá dizer para os restantes tipos de *estantes* e de *células*. No entanto, para aqui podermos usar uma nomenclatura mais reduzida, optou-se por dividir as células em três grupos distintos: células do tipo I, do tipo II e do tipo III (fig.6.6). Nas do primeiro tipo, o material ocupa somente as dimensões de um alvéolo. As do tipo II estendem-se segundo a profundidade da estante, podendo ocupar vários alvéolos contíguos, mas somente um único nível (ou andar); e as do tipo III são células que tanto se estendem segundo a profundidade da estante como segundo a sua altura.

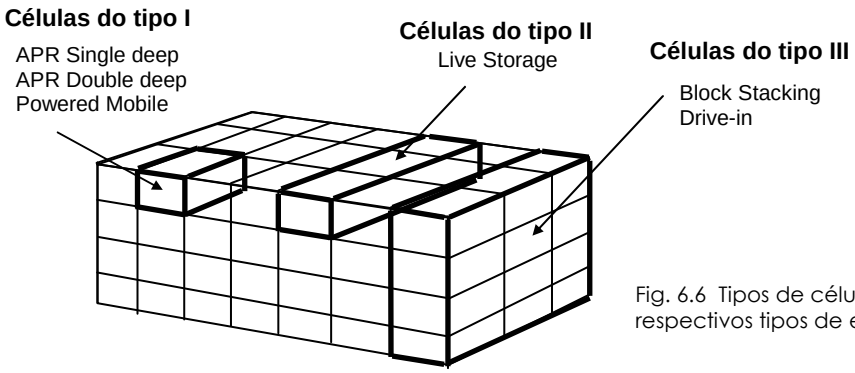

Fig. 6.6 Tipos de células e respectivos tipos de estantes.

É fácil de perceber que as *células* do tipo I implementam uma lógica de acesso aleatório às paletes do seu interior, as do tipo II promovem uma lógica de '*First In / First Out*', e as do tipo III uma lógica mais complicada se acedidas dos dois lados. Normalmente, no entanto, por questões de segurança no manuseamento do material, nas células do tipo III usa-se a lógica de '*Last In / First Out*'. Para além do modo como o material é acondicionado dentro da *célula*, é também importante estabelecer os seus pontos de acesso, pontos esses que serão utilizados pelos *elementos móveis* para se posicionarem antes de acederem à *célula*, e ainda do tipo de suporte da célula, isto é, do tipo de elemento físico que é responsável por sustentar essa célula. Falou-se aqui de células pertencentes a *estantes*, mas, como se verá mais adiante, também as *mesas* se consideram um suporte para células, por isso se englobam na estrutura deste objecto as variáveis `m_tipoSuporte` e `m_suporte`, que pretendem caracterizar o tipo de suporte específico de uma determinada célula. Para que o objecto *Célula* contenha toda esta informação, foi definida a seguinte classe de elementos:

```
class Celula : public CObject
{
 public:
    UINT      m_logica;         //Lógica da célula (APR sd, DRIVE IN, etc...)
    UINT      m_maxPaletes;     //n.º máximo de paletes possíveis na célula
    UINT      m_nPaletes;       //n.º actual de paletes na célula
    UINT      m_nPalReserv;     //n.º de paletes reservadas para serem retiradas
    UINT      m_nPalChegar;     //n.º de paletes a caminho da célula
    CTypedPtrList<CObList,Palete*> m_paleteList; //Lista de paletes
    Cpoint    m_ptInput;        //ponto de input (relacionado com a via de acesso)
    Cpoint    m_ptOutput;       //ponto de output (relacionado com a via de acesso)
    float     m_escala;         //escala dos pontos de Input e Output.
    UINT      m_tipoSuporte;    //tipo de suporte da célula: ESTANTE ou MESA
    Area*     m_suporte;        //apontador para o suporte da célula.
    Produto*  m_produto;        //apontador para o produto na célula.

 public:
    Celula();                           //construtor
    Celula(float escala);               //construtor
    Palete* GetPalete();                //retira uma palete da célula
    BOOL   PutPalete(Palete* pPal);     //coloca uma palete na célula
    UINT   GetCaixas(UINT n);   //retira  n caixas de uma célula
    BOOL   PutCaixas(UINT n);   //coloca  n caixas numa célula
    BOOL   IsEmpty();           //verifica se a célula está vazia (sem paletes)
    BOOL   IsFull();            //verifica se a célula está completa (máx. de paletes )
```

```
    virtual void Serialize(CArchive& ar); //Serializador
  DECLARE_SERIAL(Celula)
};
```

Os métodos `PutPalete()` e `GetPalete()` são os responsáveis por, respectivamente, colocar e retirar uma palete da *célula*, actuando sobre a sua lista de paletes `m_paleteList`. Dada a lógica da *célula*, registada na variável `m_logica`, e as dimensões da *estante* onde a *célula* se encontra, é fácil depois calcular a posição de uma determinada *palete* dentro dessa *célula*. Por outro lado, os métodos `PutCaixas()` e `GetCaixas()` são responsáveis por, respectivamente, colocar e retirar uma dada quantidade de caixas da *célula*, actuando sobre as paletes no seu interior.

6.1.5 Estantes

Tal como no caso das áreas, das *paletes* e das *células*, as *estantes* são elementos que possuem propriedades específicas, o que levou a considerá-las pertencendo a uma nova classe de objectos. Repare-se que, numa *estante* não só é importante conhecer a área que ocupa, mas também o número de alvéolos que contém, a lógica de acesso a esses alvéolos, as dimensões de cada alvéolo, e outras características que dependem do modo como tencionamos aceder-lhe. Uma vez que serão os *elementos móveis* os responsáveis por retirar ou colocar *paletes* nas estantes, é de esperar que este tipo de objectos possua também uma referência às vias de movimentação. Tratando-se de um aglomerado de *células*, à *estante* estará também associada uma *lista de células*. Essa lista conterá as *células* da *estante* segundo uma ordem estabelecida, começando pela *célula* mais próxima do ponto de início da *estante* (`pi`) e acabando na que contém o seu ponto de fim (`pf`). A próxima figura pretende representar o método aqui adoptado para numerar as *células* dentro de uma *estante*. Em cada caso é apresentada uma só coluna de armazenagem da estante, espaço ao qual se chamou um *Bloco*.

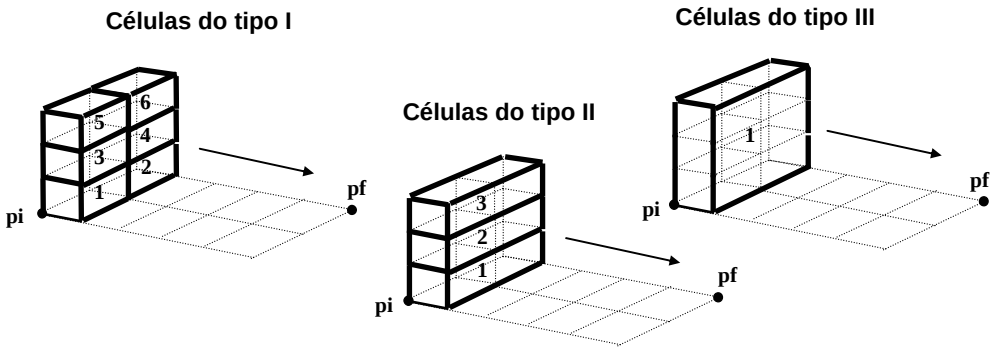

Fig. 6.7 Tipos de células e sua numeração na estante.

No caso da figura pode-se então dizer que cada estante possui cinco (5) blocos de células, sendo a primeira estante constituída por blocos de seis (6) células, a segunda por blocos de três (3) células e a última por blocos de uma (1) célula.

O objecto *Estante* deverá conter também a informação relativa às dimensões dos seus *alvéolos*, ao número de alvéolos segundo as três direcções do espaço, à orientação da estante no plano horizontal (xy), etc., informação essa que foi condensada na seguinte estrutura de classe:

```
class Estante : public Area
{
 public:
    UINT     m_tipoMod;          //tipo da estante (um dos predefinidos )
    CPoint   m_pi;               //ponto de origem da estante
    CPoint   m_pf;               //ponto de fim da estante
    UINT     m_nModX;            //N.º de alvéolos em X
    UINT     m_nModY;            //N.º de alvéolos em Y
    UINT     m_nModZ;            //N.º de alvéolos em Z ( N.º de andares)
    UINT     m_nBlocos;          //Número de Blocos na estante
    UINT     m_nCelulas;         //Número de Células na estante
    float    m_altura;           //Altura de cada alvéolo ( em metros )
    float    m_largura;          //Largura de cada alvéolo ( em metros )
    float    m_profund;          //Profundidade de cada alvéolo ( em metros )
    UINT     m_alfa;             //orientação da estante (Vertical / Horizontal)
    Cstring  m_nome;             //Texto para designação da estante
    CTypedPtrList<CObList,Celula*> m_celulaList; //Lista de células
    BOOL     m_logicDOWN;        //Lógica associada ao lado 1.
    BOOL     m_logicUP;          //Lógica associada ao lado 2.
    int      m_via1POS;          //Via de movimentação associada ao lado 1.
    int      m_via1Node;         //Nodo da via de inicio do tramo do lado 1.
    int      m_via2POS;          //Via de movimentação associada ao lado 2.
    int      m_via2Node;         //Nodo da via de inicio do tramo do lado 2.


    Estante(CString nome,int tipo,UINT alfa);          //construtor público
    void   DrawPalete(CDC* pdc,int xi,int yi,int xf,int yf,int esp); //desenha palete
    virtual BOOL Draw(CDC* pdc);                       //desenha toda a estante
    virtual void CopyTo(Estante* peTo);                //faz cópia da estante
    virtual void Serialize(CArchive& ar);              //Serializador
    void   InOutPoints(); //calcula pontos input/Output das células da estante

protected:
    Estante();           //construtor privado
    //desenho das estruturas para simular vista aérea (perspectiva tridimensional):
    void DrawEstrutura1(CDC* pdc,int xi,int yi,int xf,int yf,UINT nz,float escala);
    void DrawEstrutura2(CDC* pdc,int xi,int yi,int xf,int yf,UINT nz,float escala);

 DECLARE_SERIAL(Estante)
};
```

Se bem que a *estante* deva ser entendida como um aglomerado de células, o que a coloca a um nível de complexidade superior à do objecto *célula*, a sua tarefa resume-se, na simulação, a servir de referência para a localização espacial das suas células. A entidade *célula* é a entidade mais importante no processo, pois os mecanismos de movimentação dos veículos, assim como os métodos de atribuição de tarefas, realizam os seus cálculos olhando para o espaço de armazenagem como se, de facto, se tratasse de um "mar" de células independentes (EFACEC[5]). No entanto, optou-se por deixar à responsabilidade de cada estante a o suporte das suas células, sendo estes objectos criados sem qualquer interferência do utilizador. Ao nível da interface com o utilizador

somente se exige a informação respeitante ao objecto *estante*, criando depois o sistema as células respectivas.
Por esse motivo esta classe inclui no seu rol de métodos a função `InOutPoints()` responsável por calcular os pontos de entrada e de saída do material de cada célula, tendo em conta as dimensões atribuídas à estante e a lógica de acesso definida pelo utilizador. A criação das células, como objectos do sistema, é da responsabilidade da *'View'*, uma vez que esta possui acesso directo ao *Documento* onde as mesmas células serão guardadas numa lista de objectos. No início do processo de simulação é esta lista de células que é passada ao *Simulador*. Desta forma, permite-se ao *Simulador* olhar para um "mar" de células independentes.

A classe *Estante* está preparada para assegurar a funcionalidade dos vários tipos de estantes já referidos neste trabalho, e, para guardar essa informação, esta classe possui a variável membro designada por `m_tipoMod` que pode tomar os seguintes valores:

```
m_tipoMod = MOD_APR_SD          //Estante do tipo APR Single deep.
m_tipoMod = MOD_APR_DD          //Estante do tipo APR Double deep.
m_tipoMod = MOD_BLOCKSTACKING   //Estante do tipo Block Stacking.
m_tipoMod = MOD_DRIVEIN         //Estante do tipo Drive-in.
m_tipoMod = MOD_POWERMOBIL      //Estante do tipo Powered Mobile.
m_tipoMod = MOD_LIVESTORAGE     //Estante do tipo Live Storage.
```

Com excepção dos dois tipos **APR**, não é feita qualquer restrição ao número de alvéolos que uma estante pode conter, quer na direcção do seu comprimento quer segundo a sua largura ou altura. No caso das **APR**, a sua extensão é limitada a dois (2) alvéolos segundo a direcção de acesso, uma vez que, neste caso, um veículo não pode ter acesso a alvéolos internos. Segundo as direcções perpendiculares à direcção de acesso, uma estante **APR** pode conter um qualquer número de alvéolos, tal como nos restantes casos, desde que haja espaço para isso.

Para englobar estes seis tipos básicos de estantes, mais uma vez se estabeleceram três tipos diferentes de acessos, intimamente relacionados com os três tipos de lógica atrás considerados para o caso das células: acessos do tipo I, do tipo II e do tipo III. É importante não confundir, no entanto, a lógica de acondicionamento de material no interior da célula com a lógica de acesso dos veículos às estantes, uma vez que se tratam de coisas diferentes. No primeiro caso, trata-se de estabelecer qual o alvéolo onde será colocada a palete que entra na célula ou de que posição ela vai ser retirada, mantendo assim um permanente conhecimento da distribuição do material dentro da célula. No segundo caso, o que aqui se trata, refere-se ao modo como os veículos podem aceder às estantes, ou seja, se devem retirar o material de um dos lados e colocá-lo do outro, se podem aceder do mesmo lado à estante quer para 'input' quer para 'output', etc. Esta lógica de acesso é definida pelo utilizador do programa, enquanto que a lógica que se refere à distribuição de material dentro das células é inerente à própria célula.

6.1.5.1 *Acessos do tipo I*

Como atrás foi referido, este tipo de acessos está relacionado com as estantes constituídas por células do tipo I, células estas que se destinam a permitir um acesso aleatório às suas paletes, como é o caso das estantes **APR** e das **Powered Mobile**.

Nestes tipos de estantes, são possíveis os três (3) esquemas de acesso indicados na figura 6.8. Aí se representa, segundo uma perspectiva aérea, uma estante **APR** com profundidade máxima de dois (2) alvéolos e quatro (4) alvéolos de comprimento.

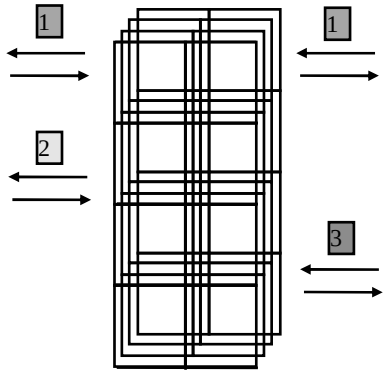

Fig. 6.8 Acessos a Estantes do tipo **APR**

Durante a fase de configuração da estante, o programa permite que seja o utilizador a escolher o modo como ela é acedida, validando somente os acessos possíveis e não permitindo aqueles que caiam fora da lógica prevista para esse modelo de estante. Por outro lado, depois de definidos os acessos, a sua lógica é guardada no próprio objecto *Estante* para futura utilização. Para isso, a classe *Estante* utiliza as variáveis `m_logicDOWN`, que diz respeito à lógica do lado esquerdo da estante da figura, e `m_logicUP`, respeitante ao acesso do seu lado direito.

Considerando que a cada lado acessível de uma estante se pode associar um estado lógico **IN** se por aí existir entrada de material, e um **OUT** existindo saída de material, a lógica dessa estante poderá então ser guardada nas duas variáveis `m_logicDOWN` e `m_logicUP` da seguinte forma (fig.6.9):

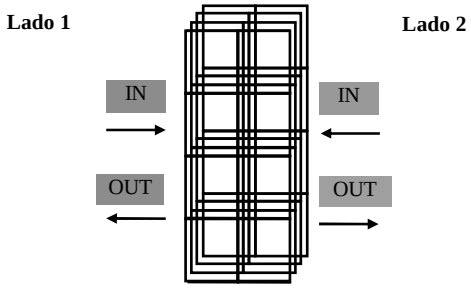

Fig. 6.9 Estados lógicos dos acessos a uma Estante

No esquema de acessos designado por 1 na figura 6.8 corresponde a ter-se, para a parte esquerda[(1)] da estante:

```
m_logicDOWN = IN + OUT
```

[(1)] Os lados da estante são definidos em relação ao seu ponto de início (pi).

e, para o seu lado direito:

```
m_logicUP = IN + OUT
```

No entanto, para a situação 2 tem-se:

```
m_logicDOWN = IN + OUT
m_logicUP   = 0
```

o que significa que nenhum material poderá ser acedido através do lado 2 da estante.

Seguindo o mesmo raciocínio, tem-se para o caso da situação 3:

```
m_logicDOWN = 0
m_logicUP   = IN + OUT
```

Como exemplo último, se a estante não fosse do tipo **APR** e se somente permitisse receber material do lado esquerdo e descarregá-lo pelo seu lado direito, ter-se-ia:

```
m_logicDOWN = IN
m_logicUP   = OUT
```

Este método permite uma representação inequívoca da lógica de acesso a qualquer tipo de estante, no entanto, é necessário verificar, antes da atribuição de valores a estas variáveis, se a lógica escolhida pelo utilizador é sustentada ou não pelo modelo de estante em questão. Para cada tipo de estante é feita uma validação da lógica atribuída, sendo o utilizador alertado em caso de conflito. Esta validação é feita com base no mapa de *Karnaugh* dos estados possíveis das variáveis temporárias **IN** e **OUT** para cada lado da estante. De salientar que será permitida uma lógica nula em qualquer tipo de estante, significando que essa estante se encontra "desactivada".
Continuando com o caso das estantes do tipo **APR** e **Powered Mobile**, é fácil verificar-se que o mapa de *Karnaugh* para a validação dos seus acessos é o seguinte:

IN - OUT (lado 1) \ IN - OUT (lado 2)	C D = 0 0	0 1	1 0	1 1
A B = 0 0	**1**	0	0	**1**
0 1	0	0	0	0
1 0	0	0	0	0
1 1	**1**	0	0	**1**

Fig. 6.10 Mapa de *Karnaugh* para os acessos às estantes *APR* e *Powered Mobile*

Donde facilmente se deduz que a lógica definida pelo utilizador é válida quando for *VERDADEIRA* a seguinte relação (por conveniência de escrita colocou-se A = **IN**(lado1), B = **OUT**(lado1), C = **IN**(lado2), D = **OUT**(lado2)):

$$OK = \overline{A} \,.\, \overline{B} \,.\, \overline{C} \,.\, \overline{D} \;+\; \overline{A} \,.\, \overline{B} \,.\, C \,.\, D \;+\; A \,.\, B \,.\, \overline{C} \,.\, \overline{D} \;+\; A \,.\, B \,.\, C \,.\, D$$

ou seja, de uma forma mais reduzida:

$$OK = (\overline{A} . \overline{B} + A . B .).(\overline{C} . \overline{D} + C . D)$$

6.1.5.2 *Acessos do tipo II*

Nas estantes do tipo **Live Storage** o material é colocado de um dos lados e retirado pelo seu lado oposto, implementando-se uma lógica do tipo '*First In / First Out*' (FIFO), dando origem ao esquema de acessos representado na figura 6.11.
Em termos das variáveis lógicas **IN** e **OUT**, as situações de acesso podem ser representadas da seguinte forma:

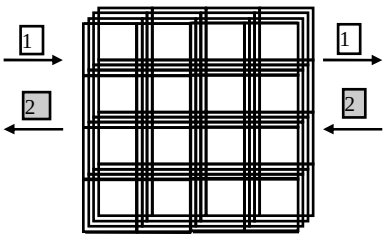

Fig. 6.11 Acessos possíveis para uma Estante **Live Storage**.

situação 1 :

```
m_logicDOWN = IN
m_logicUP   = OUT
```

situação 2 :

```
m_logicDOWN = OUT
m_logicUP   = IN
```

Tal como foi feito para o caso anterior, construa-se o mapa de *Karnaugh* para este caso, a fim de determinar uma lógica de validação para os acessos:

IN - OUT (lado 1) \ C D — IN - OUT (lado 2)	0 0	0 1	1 0	1 1
0 0	**1**	0	0	0
0 1	0	0	**1**	0
1 0	0	**1**	0	0
1 1	0	0	0	0

Fig. 6.12 Mapa de *Karnaugh* para os acessos a estantes *Live Storage*.

De onde se pode deduzir a seguinte expressão lógica de validação:

$$OK = \overline{A} . \overline{B} . \overline{C} . \overline{D} + \overline{A} . B . C . \overline{D} + A . \overline{B} . \overline{C} . D$$

6.1.5.3 *Acessos do tipo III*

O próximo esquema de acessos é reservado às estantes do tipo **Block Stacking** e **Drive-in**. Apesar de estes dois tipos de estantes diferirem substancialmente em termos de estrutura, a lógica de acesso é precisamente a mesma. O facto de os veículos poderem ter acesso aos alvéolos interiores da estante faz com que não existam quaisquer restrições ao número de alvéolos segundo a direcção do acesso. Isto significa que a lógica de acesso a estes tipos de estantes se torna um pouco mais complexa do que nos casos anteriores, se considerarmos as possibilidades de acesso de ambos os lados da estante. No entanto, por questões de segurança, nem todos os esquemas de acesso representados na figura 6.13 são usados na prática, reduzindo-se às situações 2 e 3 dessa figura. Daí que normalmente estas estantes impliquem uma lógica '*Last In / First Out*' (LIFO).

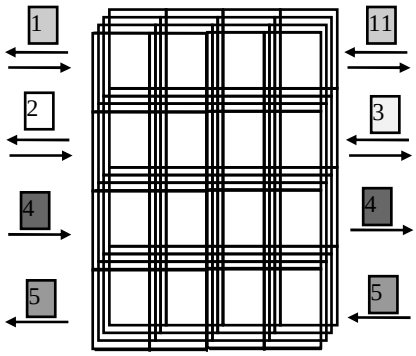

Fig. 6.13 Acessos possíveis a uma Estante **Block Staking**.

Sendo assim, as variáveis que guardam a lógica de acesso à estante poderão apresentar os seguintes valores:

situação 2 :

```
m_logicDOWN = IN + OUT
m_logicUP   = 0
```

situação 3 :

```
m_logicDOWN = 0
m_logicUP   = IN + OUT
```

Preenchendo o mapa de *Karnaugh* somente para estes dois casos, tem-se:

IN - OUT (lado 1) \ C D — IN - OUT (lado 2)	0 0	0 1	1 0	1 1
0 0	**1**	0	0	**1**
0 1	0	0	0	0
1 0	0	0	0	0
1 1	**1**	0	0	0

Fig. 6.14 Mapa de *Karnaugh* para os acessos do tipo III.

A partir do qual se pode chegar à seguinte expressão de validação:

$$OK = \overline{A} \,.\, \overline{B} \,.\, \overline{C} \,.\, \overline{D} \;+\; A \,.\, B \,.\, \overline{C} \,.\, \overline{D} \;+\; \overline{A} \,.\, \overline{B} \,.\, C \,.\, D$$

6.1.5.4 *Orientação de uma Estante*

A orientação da estante, tal como o seu número de alvéolos, o seu tipo, as dimensões dos alvéolos, etc., é um parâmetro definido pelo utilizador antes de iniciar a construção do objecto. Note-se que nem sempre é necessário dar novos valores a estes parâmetros, uma vez que na criação das estantes, como também acontece nos outros objectos, se usa o conceito de *"Tipo de Estante Activo"*, permitindo ao utilizador continuar a criar novas estantes por "clonagem" da estante activa.

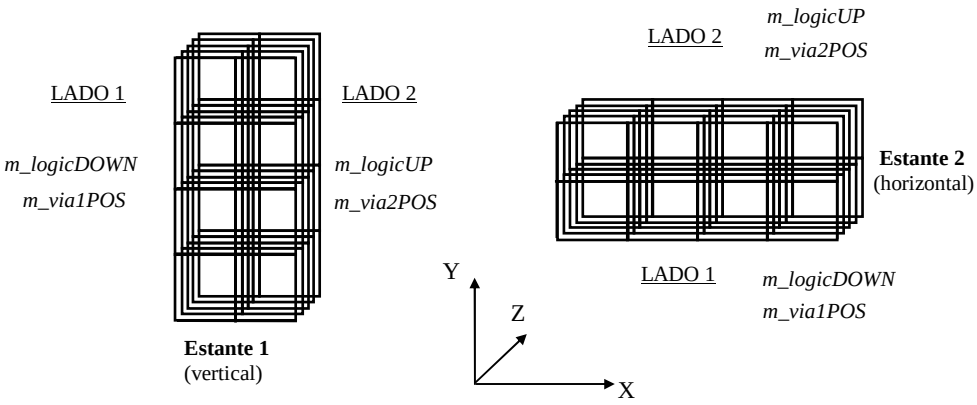

Fig. 6.15 Estante vertical e estante horizontal e algumas variáveis associadas.

Para manter a informação sobre a orientação da estante, a classe *Estante* utiliza a sua variável membro `m_alfa`. Esta variável é susceptível de tomar dois valores distintos: `MODULO_VERT` (estante vertical) e `MODULO_HORZ` (estante horizontal). Repare-se que estas direcções são no plano da área do armazém, e nada têm a ver com o aspecto tridimensional do objecto. A direcção vertical coincide, assim, com a direcção do eixo Y, e a horizontal com a direcção X do sistema de eixos representado na figura 6.15.
Optou-se por considerar somente estas duas orientações da estante porque na prática são as mais comuns, no entanto, está previsto, para uma futura versão, permitir orientar a estante segundo um qualquer ângulo no plano XY.
O acesso aos alvéolos de uma estante `VERTICAL` é feito na direcção horizontal, enquanto que, numa estante `HORIZONTAL` o acesso é feito na direcção vertical. Esta norma permite definir dois lados diferentes da estante (representados a tracejado na figura 6.15), aos quais se podem associar, conforme o tipo de estante em questão, tanto vias de movimentação como determinado tipo de lógica de acesso. Ao LADO 1, dizem respeito as variáveis `m_logicDOWN`, que contém a respectiva lógica de acesso, `m_via1POS`, que identifica a via de movimentação que serve esse lado da estante, e `m_via1Node`, que identifica o nodo dessa via onde se inicia o tramo de acesso à estante. No caso do LADO 2, são usadas as variáveis `m_logicUP` para a respectiva lógica de acesso, `m_via2POS` para a respectiva via, e `m_via2Node` para o respectivo nodo.

6.1.5.5 *Associação de vias de movimentação a Estantes*

A escolha das vias de movimentação que servem uma determinada estante é feita automaticamente através de um algoritmo implementado no programa, de modo a ser satisfeita a lógica atribuída a essa estante. O utilizador poderá identificar visualmente as referidas vias, logo que seleccione a estante em causa e esteja em modo de "*Ver vias associadas às Estantes*". Repare-se que é permitido alterar a lógica de uma estante mesmo depois de ela ter sido criada, bastando para isso seleccionar a estante com um DUPLO-CLIC do "rato" e escolher a nova lógica na *Janela de diálogo* que em seguida aparece.
Para que o sistema associe uma determinada via a uma determinada estante, é necessário verificarem-se as seguintes condições:

a) Tenham sido criadas, e por isso se encontrem disponíveis, *vias de movimentação*.

b) Que a lógica de acessos atribuída à estante exija a escolha de uma via.

c) Que a via em questão possua um ramo paralelo ao lado da estante em causa.

d) Não existam objectos entre esse ramo e a estante, e que o comprimento do ramo permita servir aquele lado da estante em toda a sua extensão.

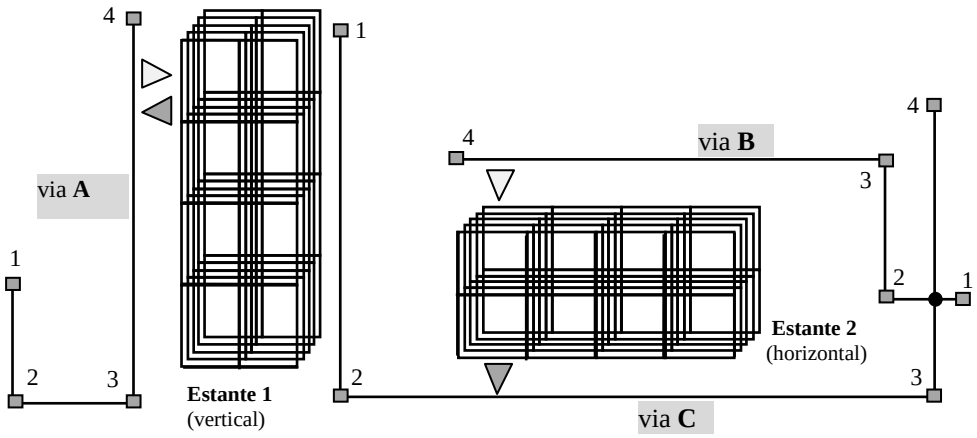

Fig. 6.16 Associação de vias de movimentação a estantes.

Como exemplo ilustrativo, analisemos o caso da figura 6.16. Aí estão representadas três (3) vias de movimentação (via **A**, via **B** e via **C**) e duas estantes. Para a estante 1 existem dois ramos de via com potencial acesso à estante: o ramo que se inicia no nodo 3 da via **A**, e o ramo que tem início no nodo 1 da via **C**. No entanto, a atribuição das vias depende da lógica de acesso predefinida pelo utilizador. Suponhamos que a estante 1 somente deverá permitir aceder ao material segundo as setas indicadas na figura. Sendo assim, a lógica desta estante é definida por `m_logicDOWN` = IN + OUT e `m_logicUP` = 0, o que desde logo fará com que o algoritmo de associação de vias ignore a via **C**,

enquanto que a via **A** é escolhida como a via de acesso à estante. Para reter essa informação, e referindo-se esta via ao LADO 1 da estante, o objecto *Estante* guarda na variável `m_via1POS` um índice que identifica esta via, e na variável `m_via1Node` o nodo onde se inicia o ramo dessa via que serve aquele lado da estante. Para o caso particular da associação da via **A** à estante 1 ter-se-ia o seguinte conjunto de valores nos parâmetros de lógica e de via associada:

Lógica de acesso:
```
m_logicDOWN = IN + OUT
m_logicUP   = 0
```

via do lado 1:
```
m_via1POS   = INDEX(via A)
m_via1Node  = 3
```

Repare-se que o índice da via é uma forma de identificar a via na *lista de vias* do *Documento*, e só será levado em conta se a lógica de acesso desse mesmo lado não for nula.

No que se refere à estante 2, a lógica associada exige que lhe sejam dedicados dois ramos de via de transportador. Um do LADO 1 (em baixo), responsável pela descarga de material, e outro no LADO 2 (em cima), utilizado para a sua carga. Como as vias **B** e **C** possuem ramos que cumprem todas as condições exigidas, automaticamente o sistema associa estas vias a esta estante. Neste caso, as referidas variáveis assumem os valores:

Lógica de acesso:
```
m_logicDOWN = OUT
m_logicUP   = IN
```

via do lado 1:
```
m_via1POS   = INDEX(via C)
m_via1Node  = 3
```

via do lado 2:
```
m_via2POS   = INDEX(via B)
m_via2Node  = 3
```

Sendo o algoritmo de associação de vias automático, cada vez que se alteram as posições quer das vias quer das estantes, é necessário de novo executá-lo. Essa opção é disponibilizada através de um comando do menu da aplicação, no entanto, sempre que se inicie o processo de simulação este algoritmo é chamado automaticamente, como teste final à consistência da informação gráfica.

6.1.6 Cais

Os *cais* são zonas específicas de recepção ou de expedição de material. Por isso mesmo, são os locais de ligação entre o armazém e o seu exterior. Os cais de recepção são responsáveis por processar *ordens de entrada de material* enquanto nos de expedição se processam as *ordens de saída de material*.

Antes de mais, um cais ocupa um determinado espaço físico do armazém, o que levou a considerá-lo com as características de uma área. Para além disso, deve possuir uma *lista de paletes* associada, uma vez que o material aí terá de esperar até que acabe de ser processado. Por esse facto, incluiu-se também no objecto *cais* um objecto *célula* que, como se sabe, contém já uma *lista de paletes*. O motivo porque se optou por incluir uma *célula* neste objecto em vez de uma simples *lista de paletes* deve-se a que assim se podem atribuir diversos tipos de lógica ao modo de processamento do material chegado ao *cais*, tantos quanto os tipos de lógica de acesso ao material de uma *célula*. Apesar disso, na maior parte dos casos considera-se esta célula do tipo **Live Storage**, o que implica uma lógica de processamento do tipo FIFO. Outra razão para incluir uma *célula* no objecto *cais* é que as tarefas de movimentação de material, criadas pelo *SimMaster* (*Sistema de Gestão* real), tratam como *células* tanto o local de origem como o local de destino do material. Sendo assim, um cais pode ser visto pelo sistema como uma célula isolada de uma estante fictícia, reduzindo-se a movimentação de material à transferência de produtos entre células.
Existe, no entanto, uma diferença fundamental entre o acesso aos cais e o acesso às estantes: as estantes são, na grande maioria dos casos, acedidas por veículos, o que levou a associar ao objecto *estante* determinadas vias de movimentação. Contudo, no caso do cais, esse acesso tanto pode ser feito por veículos como por meio de tapetes, o que o torna num caso bastante mais complicado de modelar. Se a um cais se associar uma via de movimentação, ele só poderá ser acedido por veículos, se lhe associar uma entidade do tipo tapete ele só poderá ser servido por tapetes.
Esta é uma das questões à qual ainda não se deu uma resposta e que terá de ser melhor debatida com as partes envolvidas neste projecto. No entanto, e para já, optou-se por considerar afecta ao cais uma *via de movimentação* (fig.6.17).

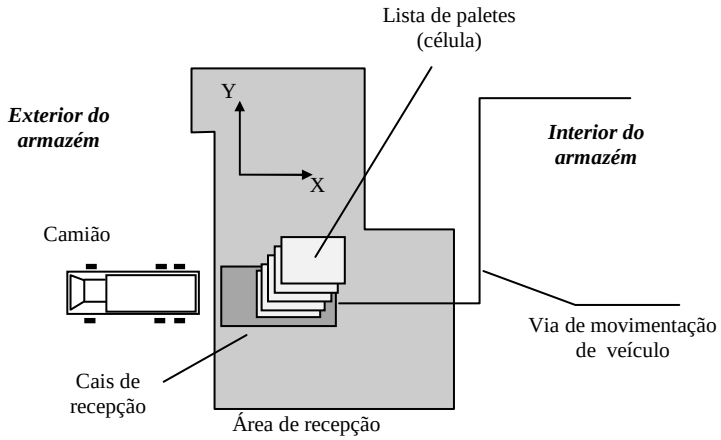

Fig. 6.17 Estrutura de um cais de recepção.

Dada a semelhança de estruturas entre o cais de recepção e o cais de expedição, tomemos como exemplo o cais de recepção representado na figura anterior. Assim que é gerada uma *ordem de entrada de material* (que corresponde, por exemplo, ao aparecimento de um camião com material para o armazém), as

respectivas paletes são transferidas para a célula do cais onde ficarão a aguardar o processamento. O cais é marcado como ocupado, não permitindo ao sistema de gestão (*SimMaster*) escolhê-lo de novo para processar novas ordens de entrada. Em seguida, o *SimMaster* verifica se existe algum *veículo* disponível com acesso a este cais e capaz de transportar uma palete da célula do cais para a zona de armazenagem. Se for possível, então atribui essa tarefa ao veículo, se não, as paletes esperarão por nova oportunidade, que acontecerá quando um *veículo* ficar livre.
Esta é uma forma simplista de apresentar o processo de recepção, no entanto, mais adiante, na secção designada por "**Eventos e métodos da *InOrders***", se voltará a este assunto com outro detalhe. No caso de um cais de expedição o processo é semelhante, diferindo somente no sentido do fluxo do material.

Por agora, apresente-se a estrutura da classe *Cais*:

```
class Cais : public Area
{
 public:
    UINT      m_tipoCais;    //RECEPTION-DESPATCH(área = AREA_CAIS)
    float     m_dX;          //Dimensão segundo X (metros)
    float     m_dY;          //Dimensão segundo Y (metros)
    BOOL      m_LIVRE;       //TRUE se o cais estiver LIVRE
    Via*      m_pVia;        //Via de acesso
    int       m_viaPos;      //Índice do extremo da via de acesso (0,1)
    Celula    m_celula;       //Célula para conter as paletes

    Cais();                              //construtor...
    Cais(UINT tipo, float dX, float dY);     //construtor...
    virtual BOOL Draw(CDC* pdc);   //Desenha o cais
    virtual void Serialize(CArchive& ar); //Serializador

 DECLARE_SERIAL(Cais)
};
```

O processo de criar um cais poderia ser implementado de várias maneiras, dependendo essencialmente do tipo pretendido de interface com o utilizador. Nesta fase, optou-se por permitir a criação deste tipo de objectos somente depois de criadas as vias de movimentação. Na verdade, isto deve-se ao facto de ter sido imposto que um cais se deva localizar num dos pontos extremos de uma via. Assim, no momento em que o operador escolhe o extremo da via onde deseja inserir o cais, automaticamente fica definida a sua *via de acesso* (`m_pVia`) e o respectivo *ponto extremo* (`m_viaPos` = 0 se for o ponto inicial da via, 1 se for o final). Se, por um lado, este automatismo fornece ao utilizador uma maneira bastante expedita de criar cais a partir de vias, por outro, exige um maior tratamento a nível de software, uma vez que as alterações gráficas realizadas na via implicam um novo reposicionamento dos cais. Da mesma forma, se uma via for eliminada, todos os cais associados a essa via devem ser eliminados também.
Depois de introduzidas as dimensões do cais, o seu tipo e a sua capacidade máxima em número de paletes, o sistema encarrega-se de criar a respectiva *célula*, que automaticamente fica associada a esse *cais*. É importante referir ainda que o *cais* assim criado é depois adicionado à *lista de cais* do *Documento*,

assim como a sua célula à *lista de células* do *Documento*, ficando por isso disponíveis aos vários módulos da aplicação.

6.1.7 Parques de veículos

Os *parques* são locais de estacionamento de veículos, onde estes geralmente aguardam até que lhes seja atribuída uma tarefa. É ainda em *parques* de determinado tipo que os *AGVs* poderão carregar baterias. Aqueles que permitem operações de carga de baterias foram designados por *parques* BATERIA, enquanto que os outros se denominaram *parques* NORMAIS.
Um *parque* fica definido através do conhecimento da *via de movimentação* que o contém e do respectivo ponto extremo dessa *via* ao qual foi associado. Sendo entendido um *parque* como uma espécie de "atributo" do extremo de uma *via*, não carece de representação física em termos de área ocupada do armazém, o que levou a derivar este objecto da classe *CObject* e não da classe *Area*. De qualquer forma, existindo um *parque* associado a uma *via* ele será fisicamente assinalado com um pequeno rectângulo no extremo dessa *via*, tarefa levada a cabo pelo método responsável pelo desenho da *via* (método `Draw()`).
Considera-se ainda que um parque pode estar OCUPADO ou LIVRE, conforme se encontre, ou não, um veículo aí estacionado.

A seguinte estrutura do objecto *Parque* contempla todas estas considerações.

```
class Parque : public CObject
{
 public:
    UINT      m_tipoParque;    //NORMAL - BATERIA
    BOOL      m_LIVRE;         //TRUE se o cais estiver LIVRE
    Via*      m_pVia;          //Via de acesso
    int       m_viaPos;        //Índice do extremo da via de acesso (0,1)

    Parque();           //construtor...
    Parque(UINT tipo);  //construtor...
    virtual void Serialize(CArchive& ar); //Serializador

 DECLARE_SERIAL(Parque)
};
```

Do ponto de vista do operador um *parque* é criado com a mesma facilidade de um *cais*, sendo somente necessário definir qual o seu tipo e especificar o extremo de uma *via* que ainda se encontre disponível. Depois o sistema encarrega-se de criar o objecto e de automaticamente lhe associar aquela *via* particular. O *parque* assim criado é depois adicionado à *lista de parques* do *Documento*, ficando por isso disponível aos restantes módulos da aplicação.

6.1.8 Vias de movimentação

As vias de movimentação são elementos auxiliares que permitem simplificar o processo de modelação do movimento de veículos. São usadas como um conjunto de eixos de movimentação e definem determinados percursos elementares aos quais os veículos se deverão cingir enquanto se movimentam no armazém. Se na maior parte dos casos estas vias representam percursos físicos concretos, como carris, fitas magnética, calhas, etc., por vezes são usadas para definir outros tipos de percursos como, por exemplo, o percurso de um operário que deve transportar material através do armazém.
Pelas suas características de encaminhamento das entidades móveis, as vias jogam um importante papel no processo de simulação, sendo através da análise do conjunto das diversas vias que se escolhe tanto o melhor percurso entre dois pontos do armazém, como o veículo ao qual se deve atribuir determinada tarefa.

Uma via é directamente desenhada pelo utilizador na *'View'* da aplicação como um conjuntos de pontos sucessivos ligados entre si por *tramos* (ou segmentos de recta), o que assemelha a sua representação à representação de uma área. A diferença fundamental é que a área é uma figura fechada, enquanto que a via é aberta.

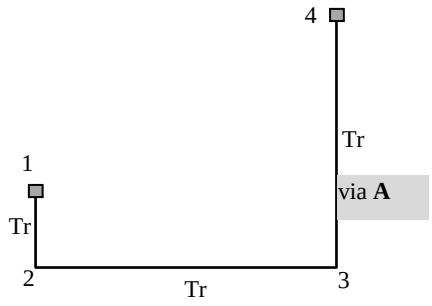

Fig. 6.18 Via de movimentação e respectivos tramos (Tr).

Na figura 6.18 representa-se esquematicamente uma via como um conjunto de pontos (ou *nodos*) ligados entre si por *tramos*, o que, só por si, justifica ter-se derivado o objecto *Via* da classe *Area*. De facto, o operador poderá movimentar a *via* dentro do armazém e colocá-la onde desejar, e essa funcionalidade é automaticamente adquirida devido à sua derivação da classe *Area*. Na mesma figura são visíveis os pontos extremos da via (*nodo* 1 e *nodo* 4), locais onde serão permitidos instalar *cais* de recepção ou de expedição, ou ainda, zonas de *parque* de veículos. Sendo assim, do ponto de vista formal, o que distingue uma *Area* de uma *Via* é precisamente a possibilidade de a esta serem associados *cais* e *parques*, o que não é previsto na classe *Area*.
É ainda de referir que foram considerados dois tipos de vias: a *via* ESTREITA, que só permite ser utilizada por um veículo de cada vez, obrigando os restantes veículos que a pretendem aceder a esperarem pela sua vez, e a *via* NORMAL, que permite que dois veículos utilizem a mesma *via* sem entrarem em conflito.

Vejamos a estrutura de implementação desta classe:

```
class Via : public Area
{
 public:   //ATRIBUTOS:
    UINT          m_viaTipo;    //Tipo de via (ESTREITA ou NORMAL)
    Cais*         m_Cais0;      //Cais do início da via (se diferente de NULL)
    Cais*         m_Cais1;      //Cais do fim da via (se diferente de NULL)
    Parque*       m_Parque0;    //Parque do início da via (se diferente de NULL)
```

```
    Parque*       m_Parque1;    //Parque do fim da via (se diferente de NULL
    int           m_actPos;     //para uso genérico (não serializada !)

  public: //MÉTODOS:
    Via(UINT viaTipo, float escala);          //Construtor público
    virtual BOOL Draw(CDC* pdc);              //Desenha a via com os cais e parques
    virtual void CopyTo(Via* pVia);           //Copiador de vias
    virtual void Serialize(CArchive& ar); //Serializador

  protected:
    Via(); //construtor privado

  DECLARE_SERIAL(Via)
};
```

Refira-se que as variáveis que dizem respeito aos *cais* e aos *parques* somente serão preenchidas no momento de criação dessas entidades, sendo iniciadas em `NULL` no momento de criação da *via*, o que significa inexistência de *cais* ou de *parques*. Assim que uma dessas entidades é criada nesta via, a parte da *'View'* da aplicação encarrega-se de preencher estas variáveis com os endereços válidos dessas entidades, tornando-as assim disponíveis para uma futura utilização. Só é permitido associar um *parque* ou um *cais* a um extremo da *via* se este ainda não se encontrar ocupado, verificação que é feita pelo software no momento em que o operador tenta criar um destes elementos.
Dado que a *via* somente mantém apontadores para os seus *parques* ou *cais*, e não inclui os objectos propriamente ditos, levanta-se aqui uma pequena questão relacionada com o processo de *Serialização*, pois o *MFC* não permite *serializar* apontadores. Isto quer dizer que não poderá ser a *via* a guardar no disco a informação respeitante aos seus *cais* ou aos seus *parques*, sendo necessário delegar essa responsabilidade ao *Documento*, que mantém as listas desses objectos. Sendo assim, quando se carregam os dados do disco para a aplicação, operação que se inicia no método de *Serialização* do *Documento*, é necessário de novo "procurar" os *cais* e os *parques* desta *via*, de forma a os associar de novo a este objecto.
Esta questão poderia ser resolvida de outra forma, permitindo à *via serializar* directamente esses elementos, criando-os automaticamente quando se criasse o próprio objecto *via*. De qualquer forma, isso implicaria considerarem-se o *cais* e o *parque* como o mesmo tipo de objecto, para além de ter de se arranjar um esquema para a sua validação, pois nem sempre a uma determinada *via* o operador associa *cais* ou *parques*.
Para já, mantém-se a actual estrutura do objecto *via*, sendo, no entanto provável a sua futura modificação, pois o objectivo é tentar criar objectos o mais compactos possível.

6.1.8.1 *Intercepção de vias e Malha de tramos*

Falou-se já do conceito de *via* como entidade isolada, mas a verdade é que no modelo de um armazém podem existir diversas *vias de movimentação*, interceptando-se ou não, conforme os casos. Na figura 6.19 representam-se três *vias de movimentação* diferentes, desenhadas pelo operador, e interceptando-se

em dois pontos. Se bem que uma *via* isolada defina um conjunto de *tramos*, quando as *vias* se interceptam o número de *tramos* disponíveis na aplicação aumenta. Repare-se que da intercepção de dois *tramos* surge um número total de *tramos* igual a quatro. Por isso, o número de *tramos* disponível para o processo de movimentação dos veículos aumenta com o aumento do número de intercepções das *vias*. Por essa razão, muitas vezes se fala em *malha de tramos* do armazém em vez de conjunto de *vias*.

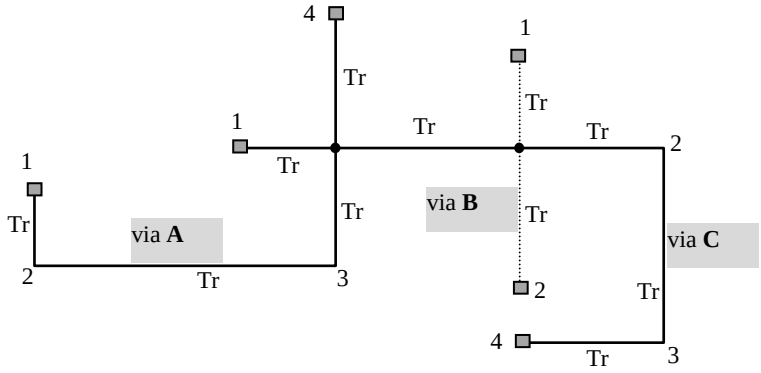

Fig. 6.19 Intercepção de vias e respectiva malha de tramos.

A *via*, em si, representa mais uma referência para a localização de determinados *tramos* do que propriamente um elemento de condução do movimento dos veículos, sendo esta tarefa levada a cabo com base no conceito de *tramo* e no de conjunto *de tramos* (*percurso*). Este conjunto de tramos é automaticamente calculado pelo sistema (por intermédio de um método da *'View'*) a partir do conjunto de vias definido pelo operador. Este cálculo pode ser despoletado pelo próprio operador, através de uma opção do menu, ou realizado automaticamente em determinados momentos da configuração do 'layout' do armazém. O conjunto global dos tramos é guardado numa lista geral de tramos do *Documento*, lista essa acessível aos diversos módulos da aplicação.
Na próxima secção falar-se-á do objecto *Tramo* com mais pormenor, fazendo a distinção entre tramos pertencentes a *vias* ESTREITAS e os pertencentes a *vias* NORMAIS.

6.1.9 O objecto Tramo

O *tramo* é considerado o objecto mais simples através do qual, no processo de simulação, se desenvolve a movimentação dos veículos. Apesar da sua óbvia relação com as vias de movimentação, o *tramo* é aqui considerado como um elemento que se situa entre o grupo dos *elementos físicos* e o dos *elementos conceptuais*. Por um lado, possui algumas características de *elemento físico*, pois ocupa um espaço no armazém, por outro esse espaço não necessita de ser representado no modelo, pois está já incluído nas *vias de movimentação*, o que o faz aproximar a um *elemento conceptual*. Assim, para evitar derivar o *Tramo* da classe *Area*, o que o transformava num objecto mais extenso e com

informação supérflua, optou-se por o derivar de um objecto mais simples herdando-o de *CObject*.
Na sua estrutura de objecto, um *tramo* é definido por dois pontos e um segmento de recta a uni-los, no entanto é necessário distinguir entre três tipos de tramos: os que fazem parte de *vias* ESTREITAS, que ficam ocupados quando utilizados por determinados veículos, o que implica associar-lhes uma fila de espera para os veículos que pretendem aceder-lhe, e os que pertencem a *vias* NORMAIS, tramos mais simples que não precisam de usar esse tipo de filas e os *tramos* que definem o percurso dentro dos limites de um *tapete* (tramo TAPETE).
Para contemplar a informação requerida por estes três tipos de tramo, foi implementado um objecto básico (*Tramo*) que satisfaz as exigências dos tramos NORMAIS e dos tramos TAPETE, e um outro objecto derivado deste, designado por *TramoEstreito* e que pretende modelar o conteúdo de um tramo de via ESTREITA. Assim, um *TramoEstreito* contém a informação de um *tramo* normal acrescida de uma nova variável que sinaliza se o *tramo* se encontra ou não livre e ainda uma *lista de veículos*, onde os veículos que pretendem acesso deverão esperar enquanto o *tramo* estiver ocupado.

A definição da classe do *tramo* básico é a seguinte:

```
class Tramo : public CObject
{
 public:
    UINT     m_tipoVia;         //NORMAL - ESTREITA- TAPETE
    CPoint   m_pi;              //Ponto de início
    CPoint   m_pf;              //Ponto de fim
    float    m_comp;            //Comprimento
    float    m_escala;          //Escala

    Tramo(); //construtor default
    Tramo(Cpoint pi, Cpoint pf, float escala);//construtor...
    virtual void Serialize(CArchive& ar); //Serializador

 DECLARE_SERIAL(Tramo)
};
```

Repare-se que é importante incluir a variável `m_escala` para representar a escala do desenho do armazém pois os pontos de início e de fim do *tramo* são registados em 'pixels', tanto para facilitar determinado tipo de cálculos como para reduzir o espaço de memória ocupado por este objecto.
Outro pormenor que merece referência é o uso de uma variável interna para o comprimento do tramo (`m_comp`). Esta variável é preenchida no momento em que o objecto é criado, passando a ter disponível o comprimento total do tramo (em metros), o que permitirá reduzir a quantidade de cálculos em determinados processos.
Relativamente ao objecto *TramoEstreito*, ele é constituído por uma variável que indica se o tramo se encontra livre (`m_LIVRE`) e por uma lista de objectos do tipo *Veículo*, onde os veículos que deverão esperar enquanto o *tramo* estiver ocupado. A sua implementação é a seguinte:

```
class TramoEstreito : public Tramo
{
 public:
    BOOL      m_LIVRE;          //TRUE se o tramo estiver LIVRE
    CTypedPtrList<CObList, Veiculo*> m_waitList; //Lista de espera

    TramoEstreito();            //construtor default
    TramoEstreito(Cpoint pi, Cpoint pf, float escala);//construtor...
    virtual void Serialize(CArchive& ar); //Serializador

 DECLARE_SERIAL(TramoEstreito)
};
```

Repare-se que não existe nesta classe referência ao tipo de tramo, aos pontos de início e de fim, ao comprimento e à escala, pois essa informação reside na sua "faceta" de *Tramo*.

Quando o sistema decide criar a malha de tramos do armazém vai verificar, tramo a tramo, o seu tipo. Se o tramo pertence a uma via ESTREITA é criado um objecto *TramoEstreito*, se não, é criado um *Tramo* normal. Se o tramo for de TAPETE é posteriormente alterada a variável `m_tipoVia` para TAPETE. Em ambos os casos os respectivos construtores(1) inicializam o objecto dados os parâmetros que representam o ponto de início, o ponto de fim, e a escala do desenho, calculando ainda o comprimento do tramo e guardando-o na variável `m_comp`.

Vejamos o construtor que é chamado para criar um tramo de via NORMAL:

```
Tramo:Tramo(Cpoint pi, Cpoint pf, float escala)
{
    m_tipoVia = NORMAL; //por default...
    m_pi = pi;
    m_pf = pf;
    m_escala = escala;
    m_comp = escala * Sqrt((pf.x - pi.x)^2 + (pf.y - pi.y)^2);
}
```

O construtor para os tramos de via ESTREITA executa instruções semelhantes, colocando, no entanto, `m_tipoVia` = ESTREITA e a sua variável m_LIVRE = TRUE.

Pela semelhança entre os construtores destes dois objectos é fácil perguntar porque razão não se considerou um só objecto que englobasse estes dois casos. A verdade é que se trata de economia de memória. Se um *tramo* NORMAL fosse entendido como um caso particular de um *TramoEstreito* iria possuir na sua estrutura uma lista de objectos que não necessitaria de ser usada, o que implicaria o uso supérfluo de memória.

O facto de estes objectos serem diferentes não limita a possibilidade de se ter uma só lista de tramos no *Documento*, pois ambos os objectos possuem uma base de construção comum: ambos possuem características da classe *Tramo*. Por isso, a lista geral de tramos guardada no *Documento* pode incluir tanto *Tramos* NORMAIS como *TramosEstreitos*.

(1) Método executado no momento da criação de um objecto.

6.1.10 Conceito de Percurso

Um *Percurso* é aqui considerado como um conjunto de *Tramos* sucessivos através do qual o material se pode movimentar, seja pela acção dos veículos ou por conta de um conjunto de tapetes. Apesar de não se poder considerar um elemento físico, ele é aqui referido por estar intimamente relacionado com certos elementos móveis, sendo preferível referirmo-nos a ele antes de entrarmos nas próximas secções deste trabalho. Posteriormente, numa próxima secção reservada para o efeito, voltar-se-á a falar deste elemento com mais detalhe.
A definição da sua classe, que a seguir se apresenta, diz simplesmente que o *Percurso* é uma lista de apontadores para objectos do tipo *Tramo*, e que no momento da criação deste objecto essa lista é forçada a estar vazia.

```
class Percurso : public CTypedPtrList<CObList, Tramo*>
{
    Percurso(){RemoveAll();} //construtor (limpa o percurso)
};
```

Note-se que a limpeza inicial desta lista é assegurada pelo construtor do objecto que, neste caso, é directamente implementado na definição da classe. Repare-se ainda que se mantém a possibilidade de *Serialização*, pois esse método é acessível à classe de base.

6.1.11 Veículos e Vaivém

Conforme já foi referido, dos veículos fazem parte os *AGVs*, os *Stocadores*, os *AGVs Stocadores* e os *Transfers*. Este último, no entanto, como veículo de transferência de outro veículo, não será aqui considerado como um objecto da classe dos veículos, uma vez que ainda se encontra em fase de estudo o método a usar para a sua modelação. Provavelmente irá ser considerado como uma *via de movimentação especial*, de forma a poder fazer parte de um percurso como outra via qualquer. Poderá também considerar-se uma classe derivada da classe dos veículos possuindo uma variável que contém o endereço do veículo que se encontra a movimentar, mas tais implicações ainda não foram completamente analisadas, o que ainda não permitiu uma escolha definitiva do tipo de elemento a considerar.
No caso dos outros, apesar de apresentarem estruturas e métodos de funcionamento um pouco diversos, foi possível modelá-los com base numa única classe de objectos: a class *Transp* (diminutivo de Transporte). Por isso mesmo esta classe possui a informação necessária para sustentar um modelo geral de veículo, fazendo com que alguns casos particulares só usem parte dessa informação. Optou-se por este tipo de formulação, em vez de criar uma classe

diferente para cada um destes elementos, uma vez que a grande parte das características é comum a todos os tipos de veículos.
Para além disso é de referir que também o *Vaivém*, classificado como um *Transportador periférico* e não como um *veículo*, é modelado com base nesta classe, pois a sua funcionalidade assemelha-se muito à incluída neste tipo de elementos.

6.1.11.1 *Movimentação*

A primeira faceta característica de um veículo é a sua capacidade de se movimentar ao longo de um determinado percurso. Consideremos uma malha genérica de tramos, representada pelo conjunto dos tramos *Tr.j* na figura 6.20, e suponhamos que esse percurso corresponde à linha a cheio entre os pontos **A** e **B**.

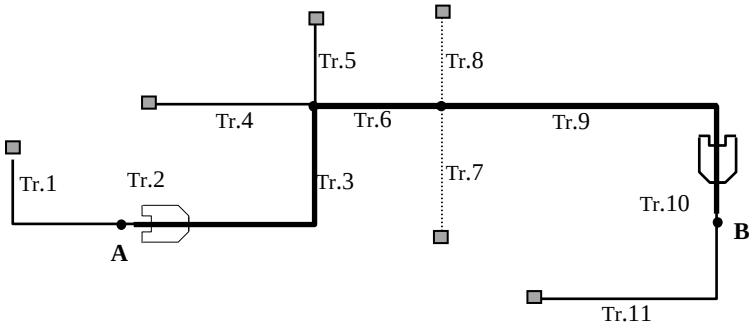

Fig. 6.20 Exemplo de percurso numa malha de Tramos.

Este percurso inicia-se no tramo *Tr.2* e acaba no tramo *Tr.10*, tendo o sentido de **A** para **B**. Para introduzir esta informação no veículo usou-se um objecto *Percurso* no qual previamente se colocam os tramos pela ordem que vão ser utilizados ao longo daquele caminho. A este percurso deu-se o nome de *Percurso Actual* e corresponde à variável da classe *Transp* designada por `m_actPercurso`. No caso representado na figura anterior este percurso será organizado da seguinte forma:

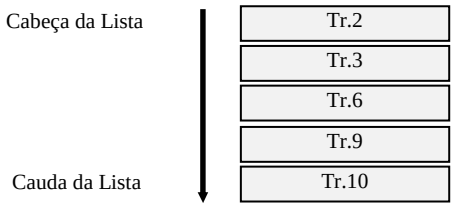

Fig. 6.21 Lista que representa o *Percurso Actual* do veículo.

Este *percurso* é preenchido antes de ser iniciado o movimento do veículo. Desta forma, assim que o movimento se inicia, o veículo desliza dentro do tramo *Tr.2* até encontrar o início do tramo seguinte (*Tr.3*). Nesse momento muda de tramo,

e de novo desliza nesse tramo até encontrar o próximo (*Tr.6*), e por aí adiante até atingir o fim do *percurso*.
No entanto, tanto o fim do percurso como o seu início podem não coincidir com um extremo de tramo, podendo tratar-se de pontos intermédios, como no caso da figura 6.20. Por isso, não só é importante conhecer-se, num determinado instante, o tramo em que o veículo se encontra, mas também a sua posição dentro desse tramo.
Segundo o método que aqui foi adoptado, a posição do veículo dentro do armazém é definida por duas coordenadas gerais: o tramo em que se encontra e a percentagem do comprimento do tramo já percorrido (λ). Como é óbvio, esta última coordenada está definida no intervalo de valores entre 0 e 1. Assim, o centro do tramo corresponde a $\lambda = 0.5$, e os extremos a 0 ou a 1.
Recordando a estrutura do objecto *Tramo*, existe um ponto de início `pi` e um ponto de fim `pf`, pontos esses que estão directamente associados ao modo como o operador desenhou a respectiva *via*. De qualquer forma, o sentido original de um tramo é o de `pi` para `pf`, e, sendo assim, $\lambda = 0.25$ representa o ponto que fica a ¼ do inicio do tramo, $\lambda = 0.75$ o ponto a ¾ do inicio, até que $\lambda = 1$ indica o ponto final do tramo. Esta ideia corresponde, no fundo, a representar o tramo com base nas sua equações paramétricas, sendo λ o parâmetro que varia. Na sua forma vectorial essas equações podem escrever-se da seguinte maneira:

$$\mathbf{p} = \mathbf{p}i + \lambda\,(\mathbf{p}f - \mathbf{p}i)$$

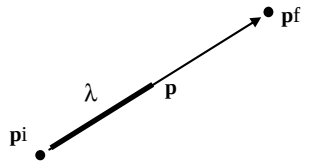

Fig. 6.22 Representação de um tramo.

Que estabelecem que qualquer ponto `p` do tramo pode ser acedido fazendo variar λ desde 0 até 1. Portanto, conhecendo o tramo e o valor de λ fica-se a conhecer exactamente o ponto em que veículo se encontra. Se o sentido do movimento do veículo dentro do tramo coincide com o sentido original do tramo, λ crescerá em direcção ao valor 1. No caso contrário, λ tenderá a diminuir para o valor 0.
Para registar o sentido do movimento no tramo actual, o veículo usa uma variável interna, designada por `m_sentido`, que toma o valor +1 na primeira situação e -1 na situação inversa.
Sendo assim, para fazer movimentar um veículo ao longo de um conjunto de tramos com diversos sentidos, basta ir-se determinando, ao longo desse percurso, o sentido do movimento no tramo actual e calcular depois o respectivo valor de λ.
O algoritmo que aqui se usa para cumprir esta tarefa é extremamente simples, o que torna este processo de rápida execução, e baseia-se na determinação do ponto do tramo actual no qual encaixa o próximo tramo. Tomemos como exemplo o percurso representado na próxima figura, onde são também apresentados os sentidos dos diversos tramos. O veículo deve deslocar-se desde o ponto **A** até ao ponto **B**.

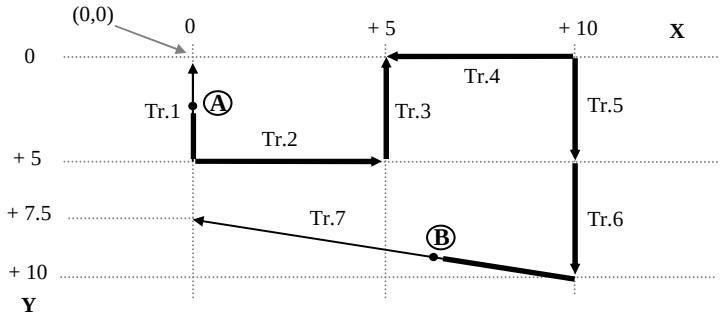

Fig. 6.23 Percurso.

O *Percurso Actual* é constituído por uma lista com os tramos ordenados de *Tr.1* a *Tr.7*. Relativamente aos tramos em que se encontram, suponhamos que o ponto **A** possui $\lambda = \lambda_a$, e o ponto **B** $\lambda = \lambda_b$. Consideremos ainda que o veículo se movimenta com uma velocidade constante *V* e analisemos então o decorrer do tempo.

Quando o veículo arranca deve verificar qual o sentido que deverá tomar dentro do tramo, por isso evoca o próximo tramo na lista do percurso e verifica em qual dos extremos do tramo actual ele se encaixa. No caso da figura verifica-se que o *Tr.2* encaixa em *Tr.1* no seu ponto inicial `pi`, o que significa que o sentido a percorrer em *Tr.1* é o sentido inverso. Sendo assim, o veículo atribui os seguintes valores às suas variáveis internas:

`m_sentido` = -1	pois o sentido é inverso...
$\lambda_{act} = \lambda_a$	
$Tr._{act} = Tr.1$	

Em que λ_{act} representa o λ actual do veículo e $Tr._{act}$ o seu tramo actual.

Ao fim do incremento de tempo *dt* o veículo andou um espaço $V \cdot dt$, o que quer dizer que se encontrará numa posição do tramo correspondente a:

$\lambda = \lambda_{act} + V \cdot dt / \Delta L$	se `m_sentido` = +1 (DIRECT)
$\lambda = \lambda_{act} - V \cdot dt / \Delta L$	se `m_sentido` = -1 (INVERSE)

Sendo ΔL o comprimento total do tramo em questão. Estas duas equações podem reduzir-se a uma só fazendo intervir a variável `m_sentido`:

$$\lambda = \lambda_{act} + \texttt{m\_sentido} \cdot (V \cdot dt / \Delta L)$$

Repare-se que desta forma poder-se-iam obter valor de λ fora do intervalo previsto [0-1], bastando para isso que *dt* fosse suficientemente elevado. No entanto, por o próprio veículo possuir na sua estrutura de dados uma variável (`m_time`) que recorda o último instante da simulação em que foi evocado, o valor de *dt* deixa de ser arbitrário, pois corresponderá à diferença entre o tempo actual da simulação (t_{sim}) e esse instante registado no veículo. Ainda, como a variável `m_time` é actualizada, pelo menos, no início e no fim de cada tramo, não se corre o risco de se obterem valores inconvenientes de *dt* durante os

períodos de movimento do veículo. A expressão mais precisa para o cálculo do valor de λ é a seguinte:

$$\lambda = \lambda_{act} + \texttt{m\_sentido} \cdot [\, V \cdot (t_{sim} - \texttt{m\_time}) / \Delta L \,]$$

Conhecido este valor de λ, calculam-se as coordenadas *(x, y)* do respectivo ponto do armazém através de:

$$x = xi + \lambda\,(xf - xi)$$
$$y = yi + \lambda\,(yf - yi)$$

E aí se representa o veículo.
Quando o veículo atingir o fim do tramo *Tr.1* e começar a entrar no tramo *Tr.2*, de novo será actualizada a sua variável `m_time` e recalculado o sentido a seguir dentro desse novo tramo. O valor desse sentido, mais uma vez registado na variável `m_sentido`, definirá o actual valor de λ_{act}:

$\lambda_{act} = 0$ se `m_sentido` = +1 (DIRECT)
$\lambda_{act} = 1$ se `m_sentido` = -1 (INVERSE)

Este processo é repetido para todos os tramos do percurso actual do veículo até que se atinja o último tramo.
Há que considerar dois casos particulares para a determinação do sentido em que o tramo vai ser percorrido: o caso do último tramo do percurso e o caso de um percurso de um só tramo. Em ambos os casos não existe um próximo tramo, por isso exige-se que este sentido seja calculado através de λ_{act} e de λ_b da seguinte forma:

$\lambda_{act} < \lambda_b$ => `m_sentido` = DIRECT
$\lambda_{act} > \lambda_b$ => `m_sentido` = INVERSE

Em quaisquer destes casos, sempre que $\lambda_{act} = \lambda_b$, o que significa que o veículo se encontra parado no fim do percurso, o sentido é reposto no valor DIRECT (sentido original do tramo).
O cálculo do sentido do movimento do tramo é realizado por um método da classe *Transp* designado por `CalcSentido()`, chamado pelos métodos `Start()` e `End()` de resposta a eventos do veículo. Estes métodos serão apresentados no capítulo 7 reservado à simulação.
Como resultado das considerações anteriores, apresenta-se um resumo do algoritmo responsável por assegurar o movimento do veículo ao longo do seu percurso actual:

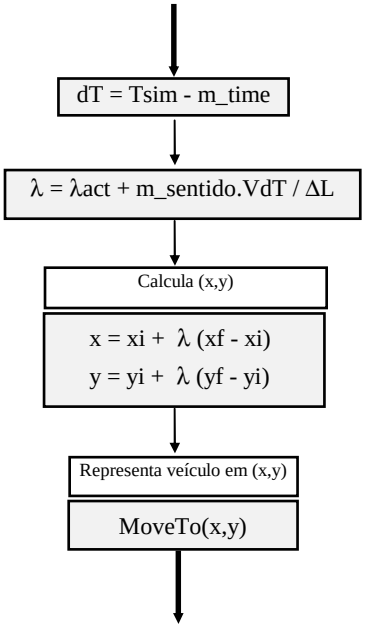

Fig. 6.24 Algoritmo de movimentação de um veículo.

Este algoritmo encontra-se incluído no método *SimShow()* da classe *Transp*, método este que é utilizado para assegurar a representação visual do veículo em movimento na janela da aplicação. Repare-se que nenhuma variável do veículo é alterada por este algoritmo, a não serem as coordenadas do ponto do armazém onde ele deve ser representado. Por isso, a execução deste conjunto de cálculos e instruções não interfere com a simulação de "fundo", o que permite desactivar o processo de representação visual sem alterar o funcionamento do processo de simulação "não visual".

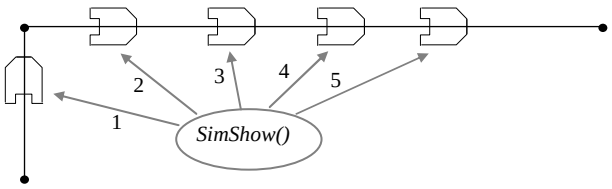

Fig. 6.25 Exemplo de actuação do *SimShow()* num veículo.

Com a introdução deste algoritmo na classe *Transp* a tarefa de movimentar um veículo reduz-se a fixar os tramos inicial e final e os respectivos λ_a e λ_b. A partir daí, o veículo encarrega-se do resto, deslocando-se através do seu percurso até ao ponto final escolhido.

A escolha do percurso a utilizar numa dada movimentação é feita com base numa lista de percursos predefinidos e associados ao veículo em causa. Por essa razão foi incluída neste tipo de objectos uma *lista de percursos* (`m_percursoList`) que é preenchida pelo operador e não poderá ser alterada durante o decorrer da simulação. O seu preenchimento é simples, visto o

operador só necessitar de indicar com o "rato" a sequência de tramos que lhe diz respeito. Sendo assim, pressupõe-se que um determinado veículo só possa movimentar-se numa certa região do armazém, que corresponde precisamente à região servida pelos seus percursos pessoais.
Se bem que este pressuposto é uma aproximação ao problema real, outros pressupostos poderiam ter sido usados, por exemplo, deixar o módulo *SimMaster* (Sistema de Gestão) decidir durante a simulação sobre qual o percurso específico a atribuir a esse veículo. Nesse caso esta classe não necessitaria de uma *lista de percursos*, mas somente de um *percurso global* onde estariam registados todos os tramos passíveis de serem acedidos pelo veículo, e um *percurso actual* que seria preenchido pelo *SimMaster* antes de mandar movimentar o veículo. Apesar de se ter optado pela anterior solução, esta questão continua em discussão com os responsáveis da EFACEC, podendo vir a ser alterada a estrutura da classe *Transp* como resultado das próximas decisões tomadas.

6.1.11.2 *Associação de Cais e Parques*

Considerou-se também que a um determinado veículo estão associados determinados *cais* e *parques*. Através dos primeiros o veículo pode aceder às zonas de entrada e de saída de material do armazém e, através dos segundos, aos seus parques de estacionamento. Tratando-se de AGVs, alguns parques podem ser também utilizados como zonas de carga de baterias.
A inclusão desta informação na classe *Transp* é feita através de duas listas de objectos: a *lista de Cais* (`m_caisList`) e a *lista de Parques* (`m_parqueList`). Estas listas são preenchidas pelo operador, como no caso da *lista de percursos*, antes de se dar inicio ao processo de simulação. A partir desse momento estabelecem-se as possibilidades de acesso do veículo ao conjunto de *cais* e *parques* previamente definidos na aplicação.
Refira-se que a um *cais* podem estar associados vários veículos, no entanto, a um *parque* somente se considerou poder associar um veículo. Na verdade, apesar de a classe *Transp* possuir uma *lista de parques*, esta lista pode-se considerar de um só elemento, pois cada veículo tem reservado para si um só *parque*.
Em todos os casos de associação de objectos uns aos outros, é necessário ter-se em conta as acções a realizar no momento em que um desses objectos é eliminado. Em grande parte dos casos, é necessário que a aplicação 'refresque' a informação de todos os objectos envolvidos, resultando num acréscimo do código do programa. Neste caso, quando um *parque* ou um *cais* é eliminado, todos os veículos são inspeccionados para ver se contêm referências a esses objectos, e, caso sim, essas referências serão eliminadas também. Isto faz parte da interface com o utilizador do programa, e, nesta fase, somente alguns aspectos se encontram implementados, pois o mais premente é o processo de simulação.

6.1.11.3 *Tarefas e subTarefas*

No contexto deste trabalho uma *tarefa* é considerada uma sequência de objectivos básicos atribuída a certas unidades móveis de forma a permitir executar um ordem de movimentação de material entre dois pontos do armazém. Já atrás foi referido que qualquer movimentação de material se faria entre duas *células* preestabelecidas, por isso a *tarefa* corresponde, no fundo, a uma forma de registar a sucessão de acontecimentos entre uma *célula* de início e uma *célula* de fim, podendo, como é óbvio, existir *células* de entremeio. Consideremos como exemplo genérico o apresentado na figura 6.25 em que a *tarefa* corresponde a transportar material desde a *célula* de início `Cel1` até à *célula* de fim `Cel3`. Dado que este percurso atravessa duas zonas diferentes de movimentação, uma de *veículos* e outra de *tapetes* - separadas por uma célula intermédia `Cel2` - a tarefa completa é dividida em duas partes: a primeira correspondendo a levar o material de `Cel1` para `Cel2`, e a segunda a transportá-lo de `Cel2` para `Cel3`.

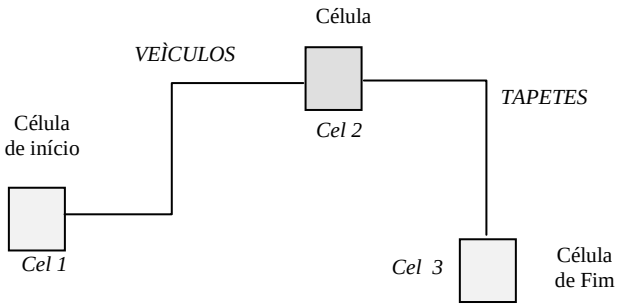

Fig. 6.26 Percurso de material e conceito de *Tarefa*.

O mesmo será dizer que um veículo deverá ser encarregado de carregar (LOAD) material da célula `Cel1` e de o transportar e descarregar (UNLOAD) na célula `Cel2`, sendo de seguida um tapete encarregado de o carregar (LOAD) de `Cel2` e de o transportar e descarregar (UNLOAD) em `Cel3`. Este processo engloba então as quatro *subtarefas* resumidas no seguinte quadro:

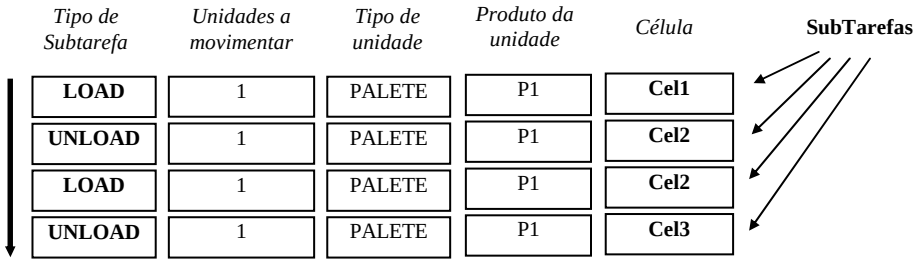

Fig. 6.27 *Tarefa* como uma sequência de *SubTarefas*.

Desta forma uma *Tarefa* pode ser representada por uma lista de *SubTarefas*. Este assunto será tratado com mais detalhe na secção reservada aos *elementos conceptuais*, por isso basta para já referir que um veículo possui na sua

estrutura um espaço reservado a uma *Tarefa*. Esse espaço é ocupado pela variável `m_tarefa` e a informação que contém corresponde a uma espécie de tabela de objectivos, onde se encontra registado o tipo de acção, o tipo de material e a quantidade a movimentar, o produto, e a respectiva célula de destino.

6.1.11.4 *Estrutura da classe Transp*

A classe *Transp* fez-se derivar da classe *Area*. Possui na sua estrutura um vasta secção de atributos, onde se registam as características gerais do veículo, e uma secção que inclui os métodos responsáveis pelo processamento da sua actividade. Na secção de atributos figuram desde o nome, o modelo, as velocidades características, até às listas de parques, de cais e de percursos. Para que melhor se entenda o significado de cada variável foi acrescentado um pequeno comentário junto à sua definição dentro da classe.
Na secção dos métodos há que distinguir duas partes: uma constituída por métodos que são uma "expansão" de alguns da classe *Area* que chamam depois os métodos com o mesmo nome desta classe de base, e uma outra constituída por métodos específicos dos veículos. Esta última inclui métodos de acesso geral e métodos relacionados com o processo de simulação, de onde se destacam o "*encaminhador de eventos*" `Executa()`, o "*visualizador de movimento*" `SimShow()`, e alguns métodos específicos que respondem aos eventos específicos do veículo. Sobre estes métodos relacionados com a simulação falar-se-á mais adiante no capítulo 7 reservado à **Simulação**, ficando-nos por agora pela apresentação de algumas definições seguidas da estrutura da classe *Transp*:

```
//para os modelos dos veículos
#define MODELO_AGV_NORMAL     0
#define MODELO_AGV_STOCADOR   1
#define MODELO_STOCADOR       2
#define MODELO_VAIVEM         3

//estados da variável <m_orient>:
#define TR_UP           1
#define TR_LEFT         2
#define TR_DOWN         3
#define TR_RIGHT        4

//estados da variável <m_sentido>: (sentido do deslocamento no tramo actual)
#define DIRECT          +1
#define INVERSE         -1

class Transp : public Area
{
 public:   //ATRIBUTOS: ****************************************

    CString      m_nome;             //nome do veículo
    UINT         m_orient;           //orientação do veículo
    int          m_sentido;          //sentido do movimento no tramo actual
    UINT         m_tipoTransp;       //via ESTREITA ou NORMAL
    UINT         m_tipo2Transp;      //pega CAIXAS ou NORMAL
    UINT         m_modelo;           //modelo do veículo
    float        m_timeCarregar;     //tempo de carregar
```

```
    float        m_timeDescarregar;  //tempo de descarga
    UINT         m_caixasMinut;      //caixas por minuto
    float        m_comp;             //comprimento do veículo (m)
    float        m_larg;             //largura do veículo (m)
    float        m_velDescCarg;      //velocidade na descida (c/carga)
    float        m_velDescVaz;       //velocidade na descida (vazio)
    float        m_velHorzCarg;      //velocidade na horizontal (c/carga)
    float        m_velHorzVaz;       //velocidade na horizontal (vazio)
    float        m_velSubCarg;       //velocidade na subida (c/carga)
    float        m_velSubVaz;        //velocidade na subida (vazio)
    Palete       m_palete;           //palete (vazio se palete vazia)
    Percurso     m_actPercurso;      //Percurso actual do veículo
    Tramo*       m_actTramo;         //Tramo actual
    UINT         m_actTramoIndex;    //Índice do Tramo actual
    float        m_actLambda;        //λ no tramo actual
    Tramo*       m_TramoDest;        //Tramo de destino
    UINT         m_tramoDestIndex;   //Índice do Tramo de destino
    float        m_lambdaDest;       //λ no tramo de destino
    Tarefa*      m_tarefa;           //Tarefa a processar pelo veículo
    SubTarefa*   m_subTarefa;        //Actual subtarefa
    UINT         m_tipoTarefa;       //Tipo da tarefa
    float        m_batery;           //Carga da bateria
    float        m_timeBatery;       //Tempo de carga da bateria


    //Listas de Parques, Cais e Percursos (fixas)
    CTypedPtrList<CObList, Parque*>        m_parqueList;
    CTypedPtrList<CObList, Cais*>          m_caisList;
    CTypedPtrList<CObList, Percurso*>      m_percursoList;

    //MÉTODOS: ************************************************************

    Transp(UINT tipo, float escala); //Construtor público

    //Métodos de expansão da classe Area:
    virtual UINT Rotate(UINT dir);
    virtual void CopyTo(Transp* pTransp);
    virtual BOOL Draw(CDC* pdc);
    virtual void MoveTo(ArmView* pView, CPoint ptf);

    void   Modelo(UINT modelo, float larg, float comp); //Cria o modelo do veículo
    int    CalcSentido(Tramo* tramo);//Calcula o sentido com que percorrer o tramo
    float  CalcDestino();//Calcula o destino e coloca o valor em m_lambdaDest
    Percurso* CalcPercurso();//Calcula e preenche o percurso actual
    float  CalcTimeToStop();//Calcula o tempo para chegar ao destino
    float  CalcTimeToEnd();//Calcula o tempo para atingir o próximo tramo

protected:
    Transp();           //construtor default
    virtual void Serialize(CArchive& ar);//Serializador

    //MÉTODOS RELACIONADOS COM A SIMULAÇÃO: **************

public:
    virtual BOOL Executa(UINT event); //Encaminhador dos eventos
    virtual void SimShow();           //Visualizador do movimento

 private: //resposta aos eventos :
    BOOL  Start();
    BOOL  End();
    BOOL  Stop();
    BOOL  Carga();
    BOOL  Descarga();
    BOOL  Free();
```

```
  DECLARE_SERIAL(Transp)
};
```

É ainda de referir que no momento da criação de um veículo o objecto *Transp* é adicionado à lista de veículos do *Documento*, ficando assim acessível aos restantes módulos da aplicação.

6.1.12 Tapetes

Um tapete é, na generalidade, considerado como um objecto que possui uma velocidade média de movimentação `m_vel`, uma lista de elementos do tipo *Palete* (`m_paleteList`), um ponto de início `m_si`, um ponto de fim `m_sf` e duas referências para os objectos que têm a ver com a sua descarga. O objecto referenciado por `m_pEntity1` representa a entidade para onde é feita a descarga da palete quando o tapete se encontra em movimento invertido, enquanto que `m_pEntity2` referencia a entidade para a qual o tapete descarrega a palete quando se movimenta no sentido normal. O sentido normal é o sentido com que o utilizador da aplicação criou este objecto (fig. 6.28). Quando se cria um tapete é automaticamente criado também um objecto *Tramo* do tipo TAPETE que depois o sistema se deve encarregar de fazer incluir na *lista de Tramos do Documento*. Associam-se ainda ao tapete dois estados possíveis: `STOPPED` se está parado e `MOVING` encontrando-se em movimento.

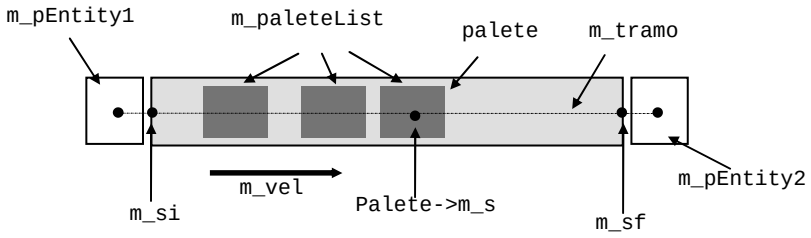

Fig. 6.28 Representação do modelo de um *Tapete*.

Para que o objecto *tapete* possa inverter o sentido do movimento foi englobada na sua secção de atributos a variável `m_dir`, que toma o valor +1 no caso de o sentido de movimentação coincidir com o sentido original da criação do tapete, e o valor -1 tratando-se do sentido inverso. A alteração do sentido influencia tanto a escolha do ponto onde se deve considerar o fim do tapete, como implica uma reavaliação da ordem das paletes na sua *lista de Paletes*. Existindo duas entidades nas quais o tapete pode descarregar o material, foi necessário introduzir uma nova variável, `m_pEntDescarg`, que referencia a actual entidade para onde essa descarga é feita, em face do valor de `m_dir`.

```
m_dir = +1   =>    m_pEntDescarga = m_pEntity2
m_dir = -1   =>    m_pEntDescarga = m_pEntity1
```

Foi também incluída na classe *Tapete* a variável `m_pFim` que representa o ponto de fim do tapete conforme o sentido do movimento, podendo tomar os seguintes valores:

```
m_dir = +1   =>    m_pFim = m_sf
m_dir = -1   =>    m_pFim = m_si
```

Há que distinguir entre dois tipos básicos de tapetes: os *tapetes de correntes*, que geralmente transportam a palete apoiada em duas correntes, uma de cada lado do tapete, e os *tapetes de rolos* que a deslocam sobre rolos mantendo-a guiada como numa espécie de calha. Os primeiros permitem a instalação de mesas no seu interior, pois são tapetes abertos, e são geralmente usados nas zonas de cais e de interface com os veículos. Os segundos são mais compactos e preferencialmente usados nos deslocamentos que envolvem maiores distâncias.
Do ponto de vista de simulação, no entanto, estes dois tipos apresentam características semelhantes, diferindo somente na possibilidade que é dada ao operador de instalar mesas no interior de *tapetes de correntes*.
Quanto ao seu modo de funcionamento, mais precisamente ao processo de paragem, certos tapetes de rolos podem ser de acumulação, isto é, quando param, as paletes tendem a aglomerar-se no fim do tapete encostando-se umas às outras. Um tapete de gravidade é um caso bem representativo de um tapete de acumulação.
Para assinalar este tipo de tapetes foi introduzida na classe *Tapete* a variável Booleana `m_ACUMULA` que identifica um tapete de acumulação no caso de possuir o valor TRUE. Esta imposição é feita pelo operador através de um diálogo onde se definem as características do tapete.
Também no tapete é usada uma variável do tipo *Tarefa* que permitirá ao objecto conhecer a tarefa que se encontra a realizar.

Tal como no caso dos veículos, os métodos da classe *Tapete* dividem-se em dois géneros: o que representa a "expansão" da classe *Area*, e outro constituído por métodos específicos desta classe. Nestes se incluem métodos de acesso geral e métodos relacionados com o processo de simulação. Neste últimos, novamente o "*encaminhador de eventos*" `Executa()`, o "*visualizador de movimento*" `SimShow()`, e os métodos específicos que respondem aos eventos específicos do tapete. Sobre estes métodos falar-se-á mais adiante no capítulo reservado à **Simulação**.

A seguir apresenta-se a estrutura da classe dos objectos do tipo *Tapete*.

```
class Tapete : public Area
{
 public:   //ATRIBUTOS: ****************************************

    float        m_larg;            //largura do tapete (m)
    float        m_comp;            //comprimento do tapete (m)
    UINT         m_moving;          //direcção de instalação em XY
    SimEntity*   m_pEntity1;        //Entidade de descarga 1
    SimEntity*   m_pEntity2;        //Entidade de descarga 2
    SimEntity*   m_pEntDescarg;     //Entidade actual de descarga
```

```
    float         m_tDescarga;        //tempo de descarga (s)
    float         m_vel;              //velocidade com carga (m/s)
    CPoint        m_si;               //ponto de início do tapete
    CPoint        m_sf;               //ponto de fim do tapete
    CPoint        m_pFim;             //ponto actual de fim de tapete
    int           m_dir;              //direcção do movimento
    UINT          m_nPaletes;         //n.º de paletes actualmente no tapete
    BOOL          m_ROLOS;            //TRUE se for tapete de rolos
    BOOL          m_ACUMULA;          //TRUE se for tapete de acumulação
    Palete*       m_endPalete;        //Palete que se encontra no fim
    Tarefa*       m_tarefa;           //Tarefa a processar pelo tapete
    SubTarefa*    m_subTarefa;        //SubTarefa a processar pelo tapete
    CArray<int,int> m_dirArray;       //Array de direcções dos tapetes na tarefa.
    Tramo         m_tramo;            //Tramo associado ao tapete

    //Lista de paletes:
    CTypedPtrList<CObList, Palete*>  m_paleteList;

    //MÉTODOS: ************************************************************
                  Tapete(float escala);   //construtor público
    virtual void  CopyTo(Tapete* tapTo);//extensão da Area
    virtual BOOL  Draw(CDC* pdc);         //extensão da Area

protected:
                  Tapete();                     //construtor default
    void          DrawMoving();                 //efeito de movimento
    void          LoadPalete(Celula* pCel);     //Carrega palete
    virtual void  Serialize(CArchive& ar);      //Serializador

    //MÉTODOS RELACIONADOS COM A SIMULAÇÃO: **********************

public:
    virtual BOOL Executa(UINT event); //Encaminhador dos eventos
    virtual void SimShow();           //Visualizador do movimento

private: //resposta aos eventos:

    BOOL   StartJob();
    BOOL   Start();
    BOOL   End();
    BOOL   Descarga();
    BOOL   Stop();
    BOOL   StopAcumula();
    BOOL   EndAcumula();
    BOOL   Invert();

DECLARE_SERIAL(Tapete)
};
```

Tal como no caso dos restantes objectos, depois de criado um *Tapete* ele é adicionado à *lista de Tapetes* do *Documento*.

6.1.13 Mesas e Pontos de Identificação (IP)

As *mesas* não se consideram elementos de transporte de material mas sim de interface entre percursos por onde o material se movimenta, quer se tratem de mesas *rotativas* ou de mesas de *transferência ortogonal*. Na prática, uma *mesa* é um posto que funciona como uma *célula* onde se podem depositar e de onde

se podem retirar *paletes*. São, por isso, entidades com o mesmo tipo de funções que as *células* de uma estante, apresentando-se, no entanto, isoladas. Do ponto de vista de um percurso de material, as mesas ocupam sempre determinados nodos, pois são elementos normalmente usados para alterar a direcção do movimento ou para reposicionar as paletes de forma a serem depois manejadas por outras entidades do armazém.
Considerar-se uma *mesa* como uma espécie de *célula* parece, de facto, um ponto de vista interessante, quer pela simplicidade que isso permite na concepção dos algoritmos de cálculo de percursos de escoamento do material, como por se poderem usar diversos métodos de acomodação das paletes dentro da mesma mesa. Por exemplo, se se optar por associar à mesa uma célula do tipo **APR single-deep**, então somente uma palete poderá ocupar essa mesa de cada vez, no entanto, se a célula for do tipo, por exemplo, **Live Storage**, já aí se poderão encontrar várias paletes ao mesmo tempo, obedecendo depois o seu acesso à lógica de acesso das células deste tipo (secção 6.1.4).

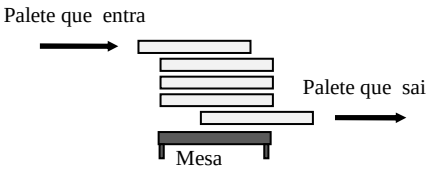

Fig.6.29 Exemplo de *Mesa* com várias paletes.

Na figura 6.29 está esquematizado um exemplo que ilustra um caso de uma mesa com uma célula do tipo **Live Storage**: a primeira palete a entrar será a primeira a sair.
Este tipo de concepção permite cobrir um vasto leque de comportamentos relacionados com mesas, pois amplia a sua utilização a casos mais complexos do que o das mesas normalmente usadas por uma palete, como, por exemplo, às mesas onde pode existir acumulação de material.
Para além da dimensão lateral da mesas (considera-se aqui, por uma questão de simplicidade, que uma mesa é sempre um objecto quadrado), tanto no caso das mesas *rotativas* como nas de *transferência ortogonal* foi considerado um tempo característico que deverá ser definido pelo operador. Este tempo refere-se ao tempo de actuação do seu mecanismo para o posicionamento da palete e é guardado na variável `m_actionTime`. Nas mesas de *transferência ortogonal*, este tempo corresponde ao tempo de subida ou de descida da mesa, conforme se trate da transferência num sentido ou noutro, e nas mesas *rotativas*, ao tempo que demora uma rotação da palete. Neste último caso, considera-se uma rotação simples a rotação de um ângulo de 90º, por isso, querendo rodar a palete de 180º terão de se considerar dois "impulsos" de rotação.
Pelas semelhanças encontradas entre uma *mesa* e um *Ponto de identificação (IP)*, decidiu-se considerar este último objecto um caso particular de *mesa*, ficando por isso englobado nesta mesma classe de objectos. Nesse caso a mesa será do tipo "IP" e o tempo de demora no processo de identificação dado por `m_actionTime`.

Por todos estes elementos se tratarem de objectos que necessitam de representação no ecrã derivou-se a classe *Mesa* da classe *Area*:

```
class Mesa : public Area
{
 public:   //ATRIBUTOS:
    UINT          m_tipoMesa;   //Tipo de mesa (ROTATION, TRANSFER ou IP)
    float         m_largura;    //Largura da mesa em metros
    float         m_actionTime;//Tempo de posicionamento da palete
    Celula        m_celula;     //Célula da mesa

 public:   //MÉTODOS:
    Mesa(UINT tipo, float larg, Cpoint pt, float escala); //Construtor público
    virtual void Serialize(CArchive& ar);      //Serializador

 protected:
    Mesa(); //construtor privado

 DECLARE_SERIAL(Mesa)
};
```

Como para os restantes objectos já considerados, no momento da criação de uma mesa o sistema deverá chamar o construtor público desse objecto, passando-lhe neste caso o tipo, a largura, o ponto central e a escala da mesa. O construtor preenche as respectivas variáveis da classe, obrigando em seguida a sua *Area* a construir um objecto quadrado com as dimensões correctas centrado no ponto referido, tornando a mesa representável como qualquer outro objecto do tipo *Area*. Essa tarefa é levada a cabo pelo seguinte método construtor:

```
Mesa:Mesa(UINT tipo, float larg, CPoint pt, float escala)
{
    m_tipoMesa = tipo; //ROTATION, TRANSFER ou IP
    m_largura = larg;

    Area::m_tipo = AREA_MESA;
    Area::m_ptCentro = pt;
    Area::m_escala = escala;
    Area::m_pointArray[0].x= pt.x - larg/(2*escala);
    Area::m_pointArray[0].y= pt.y - larg/(2*escala);
    Area::m_pointArray[1].x= pt.x - larg/(2*escala);
    Area::m_pointArray[1].y= pt.y + larg/(2*escala);
    Area::m_pointArray[2].x= pt.x + larg/(2*escala);
    Area::m_pointArray[2].y= pt.y + larg/(2*escala);
    Area::m_pointArray[3].x= pt.x + larg/(2*escala);
    Area::m_pointArray[3].y= pt.y - larg/(2*escala);
    Area::End();
};
```

Assim que é criada uma *mesa*, automaticamente é também criada a sua *célula*. O sistema deve depois encarregar-se de preencher o tempo de actuação da mesa e o tipo de célula requerido, responsabilidades que nesta aplicação foram atribuídas à respectiva *'View'* por ser a responsável pela interface com o utilizador. Repare-se que no preenchimento das variáveis da célula se incluem também as referentes ao seu suporte físico, que neste caso será uma MESA em vez de uma ESTANTE.

Por fim, a *mesa* é adicionada à *lista de mesas do Documento* e a nova *célula* à *lista de células da aplicação*.

6.2 Elementos conceptuais

Se aos *elementos físicos* se fizeram corresponder os componentes do armazém que ocupavam um determinado espaço, ao grupo dos *elementos conceptuais* pertencem as entidades que traduzem uma ideia, um procedimento, ou mesmo a estrutura de uma ordem. Enquanto que aos primeiros se associam objectos concretos, tais como veículos e mesas, a estes últimos podem associar-se documentos, procedimentos, esquemas de organização, etc. É fácil de entender que, por exemplo, um *pedido de material*, como elemento conceptual, seja uma entidade importante na simulação, pois despoleta no armazém todo um processo de escolha de células e de movimentação de material que pode activar os mais diversos mecanismos. A consequente actividade no armazém dependerá então do conteúdo desse pedido.
Um outro exemplo é o caso de um *produto*. Um *produto* existe no espaço do armazém, no entanto encontra-se dentro de *paletes* ou de *caixas*, que são os elementos físicos que o contêm, passando então a representar uma espécie de "atributo" desses elementos físicos, o que levou a considerá-lo também um *elemento conceptual*.
Existem, no entanto, casos em que a fronteira entre elemento físico e conceptual não é tão nítida. É, por exemplo, o caso de um *percurso*, que pode ser entendido de uma maneira ou de outra conforme a abordagem que se decida fazer. Neste caso, optou-se por incluí-lo nos elementos conceptuais, uma vez que se entende como uma sucessão de *tramos*, e estes estão relacionados com *vias de movimentação* que são os elementos físicos que os suportam. De qualquer forma, o importante é que se mantenha uma determinada coerência na concepção dos diversos elementos, e esse foi o propósito da estrutura de objectos aqui descrita.

6.2.1 Percurso

Um *Percurso* é entendido como um conjunto de *tramos* sucessivos através do qual o material se poderá movimentar. Como se sabe, a responsabilidade de movimentação do material cabe aos veículos e aos tapetes, o que faz com que estes elementos estejam intimamente relacionados com o conceito de *percurso*. Apesar de, ao contrário dos veículos, os tapetes não incluírem na sua estrutura de objecto nenhuma variável deste tipo, mas somente do tipo *tramo*, eles continuam intimamente relacionados com a ideia de *percurso*, pois o cálculo do trajecto a ser realizado por uma determinada palete, levado a cabo pelo *SimMaster*, passa também por considerar o tapete como fazendo parte de um determinado *percurso* possível. Consideremos como exemplo a situação representada na próxima figura.

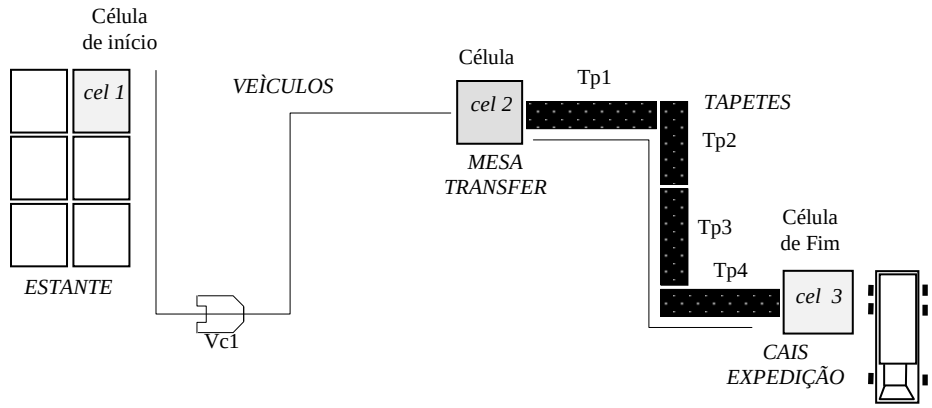

Fig.6.30 Exemplo de um *percurso* geral englobando veículos e tapetes.

Neste caso considera-se que a tarefa corresponde a transportar material desde a célula de início (`cel1`) até à célula de fim (`cel3`) e o percurso calculado envolve *tramos* afectos a veículos e *tramos* afectos a tapetes. O material é retirado da célula da estante (`cel1`) por intermédio de um veículo (`vc1`) e transportado por este para uma mesa de transferência (`cel2`). Em seguida, o tapete `tp1` encarrega-se de retirar o material dessa mesa e de o transportar até ao próximo tapete (`tp2`) que, por sua vez o transporta até ao tapete `tp3`, e este até ao `tp4`. Este último tapete encarrega-se de fazer deslizar a palete até ao cais de expedição e depois de a descarregar para a respectiva célula (`cel3`). Assim, um *percurso* pode ser representado por uma lista ordenada de *tramos*, neste caso iniciando no tramo afecto ao veículo que serve a estante e finalizando no tramo do tapete que serve o cais de expedição (`tp4`).
Pela semelhança entre um *percurso* e uma lista de apontadores para objectos, definiu-se esta classe da seguinte forma:

```
class Percurso : public CTypedPtrList<CObList, Tramo*>
{
    Percurso(){RemoveAll();} //construtor (limpa o percurso)
};
```

Indicando que o objecto *percurso* é uma lista de apontadores para *tramos*, e que no momento da criação deste objecto essa lista é forçada a estar vazia. Note-se que a limpeza inicial desta lista é assegurada pelo construtor do objecto que, neste caso, foi directamente implementado na definição da classe. Refira-se ainda que se mantém a possibilidade de *Serialização*, pois esse método continua acessível à classe de base e o percurso não possui qualquer secção de atributos que necessite de ser *serializada*.

6.2.2 Tarefas e SubTarefas

Como já foi referido na secção 6.1.11.3, uma *tarefa* é considerada uma sequência de objectivos básicos atribuída a veículos ou a tapetes de forma a permitir executar uma ordem de movimentação de material entre duas células do armazém, e a esses objectivos básicos chamaram-se *SubTarefas* (fig.6.31).

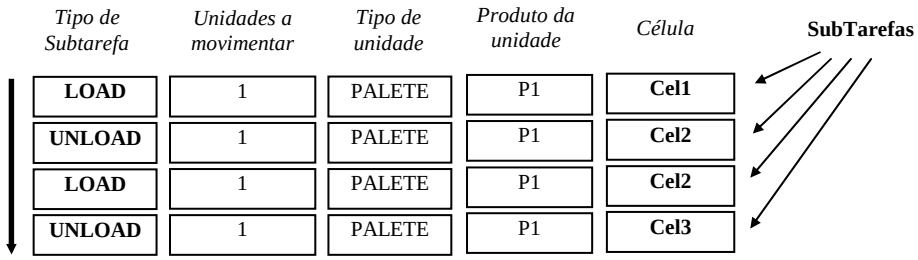

Fig. 6.31 *Tarefa* como uma sequência de *SubTarefas*.

A *tarefa* corresponde, portanto, a uma forma de registar a sucessão de *subtarefas* entre uma *célula* de início e uma *célula* de fim, existindo ou não *células* de entremeio. Sendo assim, a *tarefa* é uma entidade que deverá pertencer ao "gestor" do armazém, podendo ser dirigida a vários ou a um único elemento móvel. Se bem que A.C.Brito(1) distinga quatro tipos de tarefas dentro de um armazém automático, todas elas da responsabilidade dos veículos, essa proposta não parece adaptar-se à grande maioria dos casos práticos das instalações da EFACEC, uma vez que esta empresa apresenta amiúde soluções mistas de movimentação, incluindo tapetes e veículos. Para além disso, pelos contactos mantidos com esta empresa e através de alguns debates conjuntos, foi sabido que também ela atribui a responsabilidade da tarefa ao "gestor" da instalação, isto é, a lista de tarefas a serem executadas é propriedade do "gestor" e este encarrega-se de atribuir aos veículos ou aos tapetes uma só tarefa de cada vez. Estes, por seu turno, são responsáveis por cumprir as suas *subtarefas*, devolvendo depois o resto da *tarefa* ao próprio "gestor" que se encarregará de nomear nova entidade para a execução das restantes *subtarefas*. Assim, não é forçoso que uma determinada tarefa tenha de ser realizada pelo mesmo elemento móvel, como acontecia na abordagem proposta por A.C. Brito.
Em face destas considerações, debrucemo-nos sobre o conceito de subtarefa. Como facilmente se poderá deduzir da anterior figura, a subtarefa poderá ser de dois tipos: subtarefa de Carga (LOAD) e subtarefa de Descarga (UNLOAD). Na primeira é dito ao elemento móvel que execute uma operação de carregar material de uma determinada célula, enquanto que na segunda se lhe pede que aí execute uma descarga. O tipo de unidades a manejar nestes processos poderão ser *paletes* ou *caixas*, conforme o atributo "tipo de unidades" da *subtarefa* em questão. É a seguinte a estrutura do objecto *SubTarefa*:

```
class SubTarefa : public CObject
{
 public:   //ATRIBUTOS:
    UINT          m_tipoSubTarefa;     //Tipo de subtarefa:
                                        //LOAD
                                        //UNLOAD

    UINT          m_nUnidades;         //N.º de unidades a movimentar
    UINT          m_tipoUnidade;       //Tipo de unidades (PALETE, CAIXAS)
```

(1) No seu trabalho de doutoramento.

```
    UINT          m_produto;          //Índice do  Produto das unidades
    Celula*       m_celula;           //Célula envolvida

 public: //MÉTODOS:
    SubTarefa(UINT tipo, UINT nUnid, UINT tipoUnid); //Construtor público
    virtual void Serialize(CArchive& ar); //Serializador

 protected:
    SubTarefa(); //construtor privado

 DECLARE_SERIAL(SubTarefa)
};
```

Uma vez que os elementos de transporte somente lidam com *subtarefas*, consideram-se "cegos" relativamente ao processo que está a ser levado a cabo no armazém, pois não sabem se esse processo corresponde a uma operação de entrada de material, ou a uma saída, ou mesmo a uma tarefa de 'Picking'. Para eles o que interessa é cumprirem a sua sequência de *subtarefas* dentro da *tarefa* que lhes foi atribuída pelo *SimMaster*.
É de referir ainda que no fim da execução de qualquer *subtarefa* ela é eliminada da lista, e que depois de executada uma subtarefa de UNLOAD o elemento transportador deve devolver a respectiva tarefa ao *SimMaster*, de forma a que este possa nomear novo elemento de transporte para dar seguimento à execução do resto da tarefa. Isto pressupõe que qualquer processo de transporte acabe no momento da descarga, passando-se depois a responsabilidade ao "gestor" do sistema. Este verifica se a tarefa está concluída. Se sim, o processo finaliza, se não, atribui o resto da tarefa ao mesmo ou a um novo elemento de transporte. Assim, tudo se passa como se cada entidade transportadora fosse responsável por ir diminuindo a lista de *subtarefas* que constitui a *tarefa* total, até que esta lista se encontre vazia, significando que a *tarefa* se encontra finalizada.
O caso da tarefa representada na figura anterior correspondia ao transporte de uma palete de uma célula para outra. Suponhamos agora que se tratava de uma tarefa de recolha de caixas de várias células e consequente descarga numa determinada célula de destino. Essa tarefa poderá ser representada pela sequência:

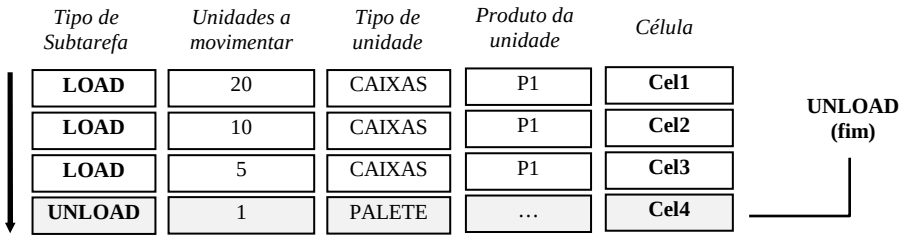

Fig. 6.32 Exemplo de *Tarefa* de 'Picking'.

Aqui, o elemento de transporte dirigir-se-ia para a célula `Cel1` e aí carregaria 20 caixas do produto `P1`, eliminando em seguida esta subtarefa da lista. Depois iria dirigir-se para a célula `Cel2` e aí carregar 10 caixas do produto `P1`, depois para `Cel3` e carregar mais 5 caixas de `P1`, até que, por fim, iria descarregar o conteúdo da sua palete na célula de destino `Cel4`. Depois disso esta tarefa seria enviada ao *SimMaster* (uma vez executada a subtarefa UNLOAD), e este, verificando que a tarefa estava cumprida, daria como finalizado o processo.

Costumam designar-se este tipo de tarefas como tarefas de ‘Picking’, e são tarefas específicas dos veículos.
Repare-se que este método de modelar a *tarefa* permite uma grande variedade de possibilidades no manejo do material, principalmente quando se trata de tarefas de ‘Picking’. Por exemplo, é possível mandar recolher de uma determinada célula um determinado número de caixas de um produto, de outra célula outro número de caixas de outro produto diferente, e descarregá-los a ambos depois numa terceira célula. No entanto, como é obvio, permitir ou não tal tipo de tarefas depende dos critérios usados no “gestor”, uma vez que é dele tanto a responsabilidade da definição da tarefa como a verificação da sua coerência. De facto, diferentes filosofias de projecto podem dar origem a diferentes ideias ou restrições sobre o que deve ser considerada uma tarefa válida. Contudo, no que se refere a este trabalho, nenhuma restrição será considerada, mantendo-se possível qualquer tipo de *tarefa* desde que ela possa ser entendida como uma sequência de *subtarefas*.
É importante ter presente, no entanto, que uma tarefa atribuída a uma entidade transportadora será executada por essa entidade somente até à próxima *subtarefa* de UNLOAD, finda a qual o objecto *tarefa* de novo é devolvido ao *SimMaster* para este decidir sobre o seu próximo destino. Este método permite a circulação de uma tarefa entre elementos transportadores diferentes, o que viabilizará a utilização de veículos e de tapetes dentro do mesmo percurso de material.
Tendo em conta que uma *tarefa* é vista como uma lista de *subtarefas*, a classe que representa este objecto foi implementada da seguinte forma:

```
class Tarefa : public CTypedPtrList<CObList, SubTarefa*>
{
    Tarefa(){RemoveAll();} //construtor (limpa a tarefa)
};
```

6.2.3 Produto e zona do produto

O material manejado dentro de um armazém é diferenciado por *produtos* sendo cada *produto* caracterizado por um conjunto de atributos. Dentro desses atributos distinguem-se aqueles que servem para a identificação do produto, os que representam o estado do produto, e ainda certos parâmetros que caracterizam aquele produto particular.
Para além da sua caracterização por intermédio destes atributos, cada produto ocupará dentro do armazém uma determinada zona de armazenagem previamente definida, o que faz com que a ele esteja associada essa região particular do armazém. No caso mais geral, essa zona pode não corresponder a uma estante ou a um conjunto definido de estantes, mas sim a diferentes células espalhadas pelo espaço de armazenagem (fig.6.33).

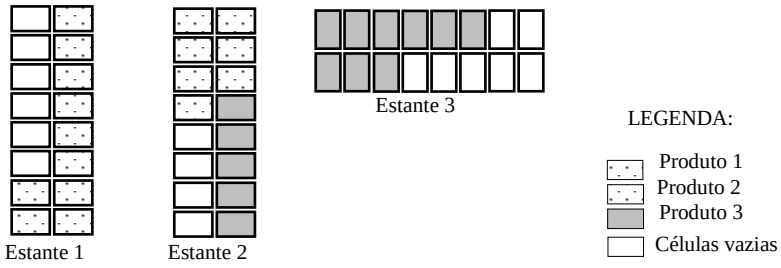

Fig. 6.33 Zonas de armazenagem de produtos.

Poder-se-á entender, então, uma *zona* de produto como um conjunto de células reservado a esse produto. Por isso, a atribuição de uma zona a um produto permite que o "gestor" do armazém possa facilmente dirigir o material que vai chegando para aquele local específico do armazém ou daí retirar uma determinada palete para depois a fazer transportar para a zona de saída. No entanto, se a cada produto se associar somente uma zona, os critérios possíveis para a escolha de uma célula particular dentro dessa zona resultam em escassas opções que não contemplam a gestão desse espaço. Pode-se escolher aquela célula particular por ser a que se encontra mais acessível, por representar o menor percurso a ser efectuado, mas não se tem em conta qualquer outro tipo de considerações. Uma maneira de permitir um maior leque de critérios nesse processo de escolher a célula dentro da zona é considerar essa zona dividida em quatro zonas distintas: zona I, zona II, zona III e zona PIC.
As três primeiras representam diferentes prioridades de acesso por ordem decrescente, isto é, só se poderá escolher uma célula da zona II depois de estar completa a zona I, e da zona III se completa a zona II. Por outro lado, a zona PIC corresponde à região da *zona do produto* onde se permitem tarefas de 'Picking'.

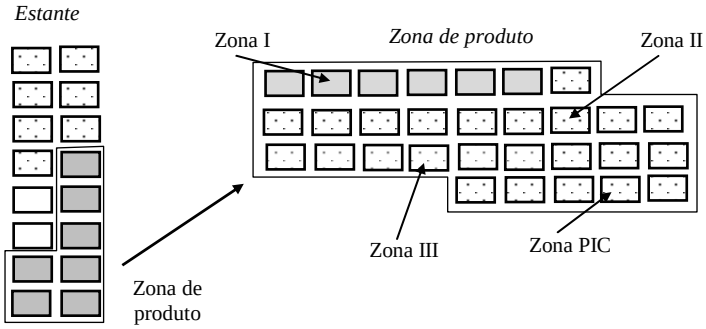

Fig. 6.34 Sub-zonas dentro de uma Zona de produto.

Tendo em conta este princípio, uma *Zona de Produto* poderá ser representada por um conjunto de quatro listas de células, uma para cada sub-zona considerada, o que levou à implementação da seguinte classe de objectos:

```
class ZonaProduto : public CObject
{
 public:  //Listas de células:
          CTypedPtrList<CObList, Celula*> m_zonaI;
          CTypedPtrList<CObList, Celula*> m_zonaII;
```

```
        CTypedPtrList<CObList, Celula*> m_zonaIII;
        CTypedPtrList<CObList, Celula*> m_zonaPIC;

public:
        ZonaProduto(); //Construtor público
        virtual void Serialize(CArchive& ar);//Serializador
protected:
        ZonaProduto(); //construtor privado
DECLARE_SERIAL(ZonaProduto)
};
```

Desta forma, primeiro deve ser criada uma *zona de produto*, depois preenchidas as suas sub-zonas, e depois atribuída essa *zona de produto* a um determinado *produto*. Assim, produtos diferentes poderão partilhar a mesma zona de armazenagem, o que muitas vezes acontece em casos de produtos semelhantes. Refira-se ainda que depois de criada uma *zona de produto* ela é adicionada à *lista de zonas de produto do Documento*, ficando assim disponível à aplicação em geral.
Incluindo a informação relativa à respectiva *zona de produto*, a classe *produto* foi implementada tendo em conta a seguinte classe de objectos:

```
class Produto : public SymEntity
{
 public:   //ATRIBUTOS:
    CString       m_nome;            //Nome do produto
    CString       m_ref;             //Referência do produto
    COLORREF      m_cor;             //Cor para representar o produto
    float         m_peso;            //Peso da palete (Kg)
    UINT          m_nCaixasPalete;   //N.º de caixas por palete
    UINT          m_nPaletesInOrder; //N.º de paletes para 'InOrder'
    UINT          m_percentPicking;  //% de caixas para reabastecer zonaPIC
    float         m_InOrderTime;     //Tempo de espera pela 'InOrder'
    BOOL          m_InORDERING;      //TRUE estando à espera da 'InOrder'
    BOOL          m_reabPICKING;     //TRUE se a reabastecer zonaPIC
    UINT          m_maxPaletes;      //N.º máximo de paletes deste produto
    UINT          m_nPaletes;        //N.º actual de paletes
    UINT          m_rPaletes;        //N.º actual de paletes reservadas
    UINT          m_maxCaixasPIC;    //N.º máximo de caixas p/ picking
    UINT          m_nCaixasPIC;      //N.º actual de caixas p/ picking
    UINT          m_rCaixasPIC;      //N.º caixas reservadas p/ picking
    ZonaProduto*  m_zonaProduto;     //zona de armazenagem

 public: //MÉTODOS:
    Produto(CString nome, CString ref); //Construtor público
    virtual void Serialize(CArchive& ar); //Serializador

 protected:
    Produto(); //construtor privado

 DECLARE_SERIAL(Produto)
};
```

Repare-se que a respectiva zona de armazenagem é referenciada pela variável `m_zonaProduto`, um apontador para um objecto do tipo *ZonaProduto*. O número de paletes que despoleta uma encomenda de material (`m_nPaletesInOrder`) e o tempo médio previsto para a chegada dessa encomenda (`m_InOrderTime`) são parâmetros que dizem respeito à gestão do produto, e são aqui usados como uma forma simples de fazer depender as

Ordens de Entrada de Material da quantidade de produto em 'stock'. Na realidade a gestão do produto pode ser bastante complexa, saindo esse assunto do âmbito deste trabalho, no entanto, esta consideração simples permitirá ligar as existências de um determinado produto às respectivas encomendas para reposição do 'stock', o que contribui para aproximar o modelo de um caso real. A quantidade a encomendar do produto ficará ao critério do "gestor" do sistema, tendo em conta a número máximo de paletes de reserva desse produto (`m_maxPaletes`). Para despoletar o mecanismo de encomendar material, o *SimMaster* encarrega-se de gerar um evento IN_ORDER_ARRIVE, sempre que o número de paletes atinge o valor crítico `m_nPaletesInOrder`.
No que respeita à zona de 'Picking', considerou-se o parâmetro `m_percentPicking` que representa a percentagem de caixas relativamente ao máximo de caixas para 'picking' abaixo da qual se deve gerar uma ordem de *Reabastecimento da Zona de Picking*. Esse pedido é enviado por um veículo ao *SimMaster* através de um outro evento, o PRD_FILL_PICKING, no momento em que é retirado de uma célula o número de caixas suficiente para o produto entrar nesse estado. Para que ao *SimMaster* pudesse ser enviado este evento, de forma a manter uma referência ao produto em causa, houve que derivar o produto da classe *SimEntity*.
Os produtos, depois de criados pelo utilizador da aplicação, serão guardados no *Documento* numa *lista de produtos*.

6.2.4 Pedido de material

Com o objectivo de se generalizar a estrutura tanto de uma *Ordem de Entrada de Material*, que corresponde a uma encomenda feita ao fornecedor do armazém, como de uma *Ordem de Saída de Material*, encomenda feita pelo cliente ao armazém, optou-se por definir o conceito de *Pedido de Material* como a unidade básica constituinte dessas ordens. Assim, um *Pedido de Material* corresponde, pura e simplesmente, a especificar-se o número de unidades requeridas, o tipo de unidades, e o respectivo produto. A próxima figura exemplifica dois desses pedidos. No primeiro pretendem-se 20 caixas do produto P1 e no segundo 10 paletes do produto P5.

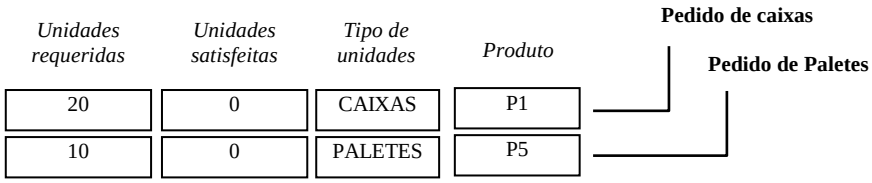

Fig. 6.35 Dois exemplos de *Pedidos de Material*.

Repare-se que se reservou também um campo para o número de *unidades satisfeitas*, que permitirá verificar se o *pedido de material* se encontra satisfeito

ou não. Como é óbvio, o pedido estará satisfeito quando este número for igual ao número de *unidades requeridas*.
Como facilmente se perceberá, quer *a Ordem de Saída* quer a *Ordem de Entrada* poderão ser entendidas como um conjunto de pedidos deste tipo, simplificando-se assim a concepção das suas estruturas. Para representar um *Pedido de Material* foi definida a seguinte classe de objectos:

```
class PedidoMaterial : public CObject
{
 public:   //ATRIBUTOS:
    UINT          m_nUnidades;        //quantidade requerida
    UINT          m_nUnidadesOk;      //quantidade satisfeita
    UINT          m_tipoUnidade;      //tipo de unidades: PALETES-CAIXAS
    UINT          m_produto;          //produto

 public: //MÉTODOS:
    PedidoMaterial(); //Construtor público
    virtual void Serialize(CArchive& ar); //Serializador

 protected:
    PedidoMaterial(); //construtor privado

 DECLARE_SERIAL(PedidoMaterial)
};
```

6.2.5 Ordem de Entrada de Material (InOrder)

Esta ordem pode ser despoletada por diversos mecanismos e está intimamente ligada às encomendas ao fornecedor. Por isso é extremamente dependente tanto da rede de fornecedores que abastece o armazém como da política de gestão interna. Se no armazém só existisse um produto seria simples determinar o momento em que deveria ser lançada uma encomenda, no entanto, na realidade a situação é bastante diversa. Não só poderão existir vários produtos como também os fornecedores podem ser diferentes conforme esses produtos. Existem ainda fornecedores de um só produto e outros que asseguram o fornecimento de vários. Por isso, perante situações tão diversas, é muitas vezes difícil generalizar os critérios envolvidos a este nível de decisão.
A principal questão que aqui se coloca é a seguinte: de quem deve ser a responsabilidade de despoletar uma *Ordem de Entrada de Material*? Do *produto*, ou do "gestor" do armazém (*SimMaster*)?
Sendo do "gestor", que critérios seguir se ao mesmo fornecedor se puderem encomendar vários produtos? Uma das hipóteses de contornar esta questão seria considerar-se um objecto *Fornecedor* afecto a cada produto para onde se iriam enviando os sucessivos *Pedidos de Material*, e ser este objecto o responsável por gerar a *Ordem de Entrada* depois de atingido um determinado número de paletes na sua *lista de encomendas*. Por um lado permitir-se-ia associar esse *Fornecedor* a vários produtos, por outro, poder-se-iam considerar vários tempos de encomenda conforma o fornecedor, o que parece uma situação bastante mais próxima da realidade.

De qualquer forma, o processo de *Entrada de Material* é normalmente menos importante do ponto de vista da performance do armazém do que o processo de *Saída de Material*, por isso, optou-se por implementar a solução mais simples: ser do *produto* a responsabilidade de pedir uma *Ordem de Entrada de Material*. Assim, sempre que o 'stock' desse produto cai abaixo de um determinado nível, automaticamente o produto pede o seu reabastecimento, enviando ao *SimMaster* o evento PRD_IN_ORDER. Por sua vez, o *SimMaster* será responsável por gerar a ordem de reabastecimento dirigida a esse produto. Como é óbvio, isto equivale a considerar que cada produto possui um fornecedor dedicado e que não existe no armazém uma gestão global das encomendas. É claro que desta forma a *Ordem de Entrada* estará relacionada com um único *Pedido de Material*, no entanto, optou-se por generalizar a estrutura dessa ordem incluindo na sua classe uma lista de *Pedidos de Material* que futuramente poderá vir a ser usada.
Com base nestes princípios foi definida a seguinte classe de objectos para representar uma *Ordem de Entrada de Material* (*InOrder*):

```
class InOrder : public SimEntity
{
 public:   //ATRIBUTOS:*******************************************************
    UINT          m_number;          //numero de ordem
    float         m_timeArrive;      //Tempo de chegada
    float         m_timeStart;       //Tempo de início de processamento
    float         m_timeEnd;         //Tempo de fim de processamento
    Cais*         m_cais;            //Apontador para o cais de entrada
    //Lista dos pedidos de material:
    CTypedPtrList<CObList, PedidoMaterial*> m_listaMaterial;

 public: //MÉTODOS: ******************************************************
    InOrder(ArmSimulation* pSim); //Construtor público
    virtual void Serialize(CArchive& ar); //Serializador
 protected:
    InOrder(); //construtor privado

 public://MÉTODOS RELACIONADOS COM A SIMULAÇÂO:*********************
    virtual BOOL Executa(UINT event);//Encaminhador dos eventos

 private: //resposta aos eventos:
    BOOL   Arrive();
    BOOL   End();

DECLARE_SERIAL(InOrder)
};
```

O facto de se ter derivado esta classe da classe *SimEntity*, da qual se falará mais adiante no capítulo reservado à simulação, deve-se a que a *InOrder* é entendida como uma entidade activa na simulação, isto é, existem eventos a ela associados, tal como no caso dos veículos e dos tapetes. Também por essa razão existem nesta classe métodos relacionados com a simulação, como o `Executa()`, que mais uma vez é o responsável pelo encaminhamento dos eventos, e os `Arrive()` e `End()` de resposta aos dois eventos processados pela *InOrder*.
Quando o sistema cria uma *InOrder* ela é adicionada à *lista de InOrders* do *Documento*, lista sobre a qual o *Simulador* (*ArmSimulation*) poderá depois actuar. Note-se ainda que as *InOrders* são objectos criados em 'runtime', não

necessitando da interferência do utilizador da aplicação, e que cada *InOrder* recebe da aplicação um apontador para o *cais* responsável por receber o material proveniente do exterior.

6.2.6 Ordem de Saída de Material (OutOrder)

Se a *Ordem de Entrada de Material* se encontra intimamente ligada ao processo de encomenda de material ao fornecedor do armazém, por outro lado, a *Ordem de Saída de Material* (*OutOrder*) diz respeito às encomendas de material feitas pelo cliente ao armazém. Por isso, em termos de estrutura, elas são muito semelhantes. Tal como as *InOrders*, as *OutOrders* são entendidas como um conjunto de *Pedidos de Material*. No entanto, nas *OutOders* não se colocam as questões de gestão levantadas pelas *InOrders*, pois a encomenda é organizada pelo cliente, exteriormente aos critérios de gestão do armazém.
A *OutOrder* é então uma lista de *Pedidos de Material*, podendo englobar vários produtos e diversas quantidades, quer acondicionados em paletes quer em de caixas, conforme o desejado pelo cliente. É comparável a uma lista de compras a efectuar no supermercado.

É a seguinte a estrutura da classe *OutOrder*:

```
class OutOrder : public SimEntity
{
 public:   //ATRIBUTOS:************************************************************
    UINT          m_number;          //numero de ordem
    float         m_timeArrive;      //Tempo de chegada
    float         m_timeStart;       //Tempo de início de processamento
    float         m_timeEnd;         //Tempo de fim de processamento
    Cais*         m_cais;            //Apontador para o cais de saída
    //Lista dos pedidos de material:
    CTypedPtrList<CObList, PedidoMaterial*> m_listaMaterial;

 public: //MÉTODOS: ********************************************************************
    OutOrder(ArmSimulation* pSim); //Construtor público
    virtual void Serialize(CArchive& ar); //Serializador
 protected:
    OutOrder(); //construtor privado

 public://MÉTODOS RELACIONADOS COM A SIMULAÇÂO:**********************
    virtual BOOL Executa(UINT event);//Encaminhador dos eventos

 private: //resposta aos eventos:
    BOOL   Arrive();
    BOOL   End();

DECLARE_SERIAL(OutOrder)
};
```

Repare-se que esta estrutura é precisamente igual à estrutura da *InOrder*, o que faria pensar na possibilidade de incluir as duas na mesma classe de objectos. De facto tal poderia ter sido feito, no entanto, devido a ser preferível separarem-se as entradas de material das saídas de material e visto que se pretende futuramente incluir em cada um destes objectos uma zona dedicada ao controle

estatístico, com diferentes parâmetros de interesse para cada tipo de ordem, optou-se por separá-las desde já em dois objectos distintos.
Pelas mesmas razões apresentadas no caso da *InOrder*, esta classe fez-se derivar da classe *SimEntity*. Existem por isso nesta classe os métodos que a tornam uma classe activa na simulação: o `Executa()` como encaminhador de eventos, e os `Arrive()` e `End()` de resposta aos dois eventos processados pela *OutOrder*.
Quando o sistema cria uma *OutOrder* ela é adicionada à *lista de OutOrders* do *Documento* a qual o *Simulador* poderá depois manejar. Refira-se ainda que as *OutOrders* são objectos criados em 'runtime', não necessitando por isso da interferência do utilizador, e que cada *OutOrder* recebe da aplicação um apontador para o *cais* de expedição para o qual se deverá fazer escoar o material.

6.2.7 Ordem de reabastecimento de zona de 'Picking'

Quando o número de caixas na zona de 'Picking' de um determinado produto cai abaixo de um certo valor crítico é gerada uma *Ordem de Reabastecimento da Zona de 'Picking'* dirigida a esse produto. O processo de reabastecimento é despoletado pelo veículo que se encontra em operação de 'picking', enviando ao *SimMaster* o evento PRD_FILL_PICKING. A este evento o "gestor" responde criando uma determinada *tarefa* para ser executada logo que possível. Repare-se que este caso é diferente dos anteriores, pois trata-se de uma ordem interna que somente exige a transferência de material entre células na zona de armazenagem. Por isso esta ordem não despoleta uma *InOrder*, mas somente introduz uma nova *tarefa* no sistema.

7. Simulação

Neste capítulo fala-se do processo de simulação propriamente dito considerando os objectos anteriormente descritos como entidades da simulação. Se os anteriores capítulos serviram para definir e apresentar esses objectos, aqui se falará de como eles interagem uns com os outros e da evolução dos seus estados com o avanço do tempo da simulação. Para levar a cabo tal tarefa houve que distinguir esses objectos pela forma como participam neste processo. Assim, foram consideradas entidades passivas, isto é, que não são responsáveis por quaisquer tipo de eventos, e as entidades activas, às quais determinados eventos foram associados. Uma mesa, por exemplo, foi considerada uma entidade passiva, uma vez que é entendida como uma espécie de "depósito" de paletes tal como acontece com qualquer outra célula. Por outro lado, um tapete é uma entidade activa, pois a ele se encontram associados eventos que promovem o desenrolar de determinadas acções na simulação.
Antes de se entrar no domínio dos processos da simulação, convém primeiro referirmo-nos ao modo como as entidades activas processam os seus eventos, questão que será abordada já de seguida com a apresentação de uma nova classe de objectos: a classe *SimEntity*.

7.1 Execução de *eventos* e Classe *SimEntity*

No capítulo 2 foi descrita a forma como na simulação se marcam e guardam os eventos e ainda o processo de retirar esses eventos da *lista de eventos* para serem executados. No entanto, nada foi dito sobre o mecanismo que está subjacente à execução de um determinado evento. Depois de definidos os vários objectos que constituem o interior de um armazém automático, falta agora explicar esse mecanismo, que se encontra intimamente ligado às entidades activas da simulação.
Na abordagem aqui proposta, cada elemento activo da simulação é responsável pelo processamento dos seus próprios eventos, o que, só por si, permite separar a responsabilidade de execução dos diversos acontecimentos. Nas linguagens de programação procedimental esta tarefa era levada a cabo por um único bloco de software, responsável por processar tanto os eventos relacionados com os tapetes, por exemplo, como os que diziam respeito às outras entidades. A um evento estava também associado um método, mas, ao contrário daquilo que aqui se propõe, esse método era uma função global da aplicação e não um método específico de um determinado objecto.
Para melhor se entender o mecanismo de execução dos eventos aqui proposto façamos uma pequena revisão de alguns conceitos referidos no capítulo 2 deste trabalho. Um determinado acontecimento no tempo é representado por um conjunto de três parâmetros, conjunto esse a que se chamou *célula de tempo* e que constitui a unidade de informação da *lista de eventos* da simulação (fig.7.1).

Fig. 7.1 Lista de eventos como um conjunto de *células de tempo*.

O primeiro parâmetro representa o tempo em que esse acontecimento aparece na simulação, o segundo o número de ordem do evento e o terceiro a entidade relacionada com esse evento. Para manter coesa esta estrutura foi criado o objecto *SimTimeCell* cuja definição de classe é a seguinte:

```
class SimTimeCell : public CObject
{
 public:
        float         m_time;       //tempo do evento
        UINT          m_event;      //número do evento
        SimEntity*    m_pEntity;    //entidade associada ao evento
};
```

A *lista de eventos* da simulação não é mais do que uma lista contendo objectos deste tipo. Assim, quando o método *Relógio()*[(1)] retira dessa lista um destes objectos para que seja executado o respectivo evento, o que é feito é chamar um método específico da respectiva entidade envolvida, que se decidiu baptizar *Executa()* e que faz parte da estrutura dos objectos da classe *SimEntity*. Portanto todos os elementos activos na simulação deverão derivar, de alguma forma, desta classe básica. Para que o processo de execução do evento seja despoletado basta então executar-se a seguinte instrução:

```
m_pEntity->Executa(m_event);
```

o que corresponde a "enviar" o evento `m_event` à entidade `m_pEntity`. Uma vez recebido o seu evento, a entidade encarrega-se de executar o seu método específico de resposta a esse evento particular, método esse que deverá pertencer à estrutura dessa mesma entidade.
Como exemplo, consideremos de novo o "caso do autocarro" e, em particular, a entidade AUTOCARRO. Como atrás foi referido, existem três eventos associados a esta entidade: EVENTO_INICIO, EVENTO_ESPERA e EVENTO_FIM. Logo, deverão existir também três métodos de resposta a estes eventos dentro da própria estrutura do objecto "autocarro". Designemo-los respectivamente por *Inicio(), Espera()* e *Fim()*. Suponhamos agora que o *Simulador* retira da *lista de eventos* a seguinte *célula de tempo* para depois mandar executar o respectivo evento:

t = 10	EVENTO_INICIO	pAutocarro

(1) Referido no capítulo 2 deste trabalho

Para accionar o método associado a este evento basta então executar a instrução:

```
pAutocarro->Executa(EVENTO_INICIO);
```

passando ao método *Executa()* a responsabilidade de encaminhar a informação para o método de resposta a este evento que, neste caso, seria o método *Inicio()*. Por isso se tem vindo a chamar "encaminhador de eventos" ao método *Executa()* de certos elementos derivados da classe *SimEntity* (fig.7.2).

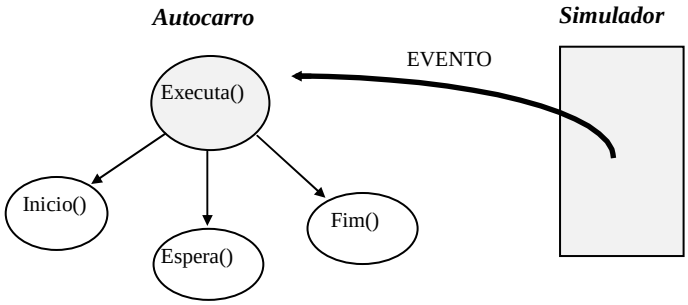

Fig. 7.2 Mecanismo de execução de eventos.

Este mecanismo é proposto por Dirk Bolier [6] num trabalho sobre bibliotecas de simulação desenvolvidas em C++, tendo sido aqui adoptado por se revelar simples, prático e eficiente. Note-se que para isso é necessário que *Executa()* seja definido como um método público da classe da respectiva entidade, de forma a poder ser acedido pelo objecto *Simulador*, enquanto que os métodos de resposta aos eventos podem ser privados dessa classe.
Como se começa a perceber, a classe *SimEntity* tem como objectivo definir a funcionalidade básica de uma entidade activa da simulação, o que leva a concluir que tanto a classe *Tapete* como a classe *Veículo* deverão também derivar de *SimEntity*. No entanto existem duas formas de conseguir isso. A primeira, mais elegante, consiste em usar o que em C++ se designa por "herança múltipla", isto é, fazer derivar um objecto de dois objectos básicos diferentes (fig.7.3).

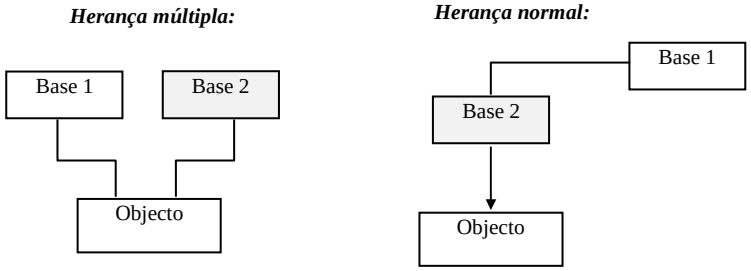

Fig. 7.3 Métodos de herança permitidos em C++.

A outra, considerada "herança normal", é fazer derivar a classe de base mais próxima do objecto (*Base2*) de uma segunda classe de base (*Base1*), o que, como é evidente, transforma a classe de onde o objecto directamente deriva (fig.7.3).

Se bem que o *Visual C++* funcione bem com "herança múltipla", ela não é permitida se ambas as classes de base derivarem de *CObject*. Por isso, como tanto a classe *Area* como a *SimEntity* derivam, de facto, de *CObject*, foi impossível utilizar este processo quer para os *Tapetes* quer para os *Veículos*, tendo-se optado pela "herança normal". Para tal fez-se derivar a classe *Area* da classe *SimEntity*, resolvendo-se assim a situação. Desta forma, todos os objectos derivados da classe *Area* contêm também a informação da classe *SimEntity*, sendo esta usada ou não conforme se trate ou não de um objecto que represente uma entidade activa da simulação.
Para além deste mecanismo de processamento de eventos, foi incluída na classe *SimEntity* uma secção de atributos comuns a todas as entidade activas, assim como um método responsável pela representação gráfica dessas entidades durante a simulação, método esse designado por *SimShow()*. A seguir apresenta-se a estrutura desta classe.

```
class SimEntity : public CObject
{
 public:
        BOOL                m_livre;      //variável lógica para uso genérico
        CString             m_nome;       //nome da entidade
        float               m_time;       //tempo da entidade na simulação
        UINT                m_estado;     //estado da entidade
        ArmSimulation*      m_pSim;       //apontador para o Simulador
        UINT                m_tipo;       //tipo da entidade
        float               m_x;          //posição x da entidade
        float               m_y;          //posição y da entidade


        SimEntity(CString nome); //Construtor público
        virtual void Serialize(CArchive& ar); //Serializador

 protected:
        SimEntity(); //Construtor privado

 public:        //MÉTODOS RELACIONADOS COM A SIMULAÇÂO:
        virtual BOOL        Executa(UINT event);//Encaminhador de eventos
        virtual void SimShow(); //Visualizador do movimento

 DECLARE_SERIAL(SimEntity)
};
```

7.2 Mecanismos do *Simulador*

O *Simulador* da aplicação é também um objecto, tal como a *'View'* ou o *Documento*, e tem como finalidade assegurar a consistência da simulação, mantendo a coerência do tempo e fazendo executar todos eventos que vão aparecendo no processo de simulação. É ele que possui referências para todas as listas de objectos criados pelo utilizador e guardados no *Documento*, e que engloba a *lista de eventos* da simulação e os métodos dedicados à sua manipulação: o *Schedule()* para marcação de eventos nessa lista e o *Relógio()* como ciclo principal da simulação, onde os eventos são retirados da lista e mandados executar. Digamos que o *Simulador* possui uma imagem do armazém previamente definido, sobre a qual

executa acções referentes aos eventos que vão surgindo, provocando mudanças de estado nas diversas entidades, gerando ordens e tarefas, fazendo deslocar paletes, veículos, etc.
Para além dos métodos *Schedule()* e *Relógio()* este objecto sustenta ainda os métodos *SimControl()* e *SimMaster()* que estão conotados com os sistemas de controle e de gestão do armazém, respectivamente, aos quais mais adiante se fará referência ainda nesta secção.
Como utilitários de uso geral foram também incluídos no *Simulador* os métodos *Unschedule()*, que permite remover da *lista de eventos* um determinado evento pertencente a uma dada entidade, o *GetRandom()* que retorna um número inteiro pseudo-aleatório dado um valor máximo, e o método *Run()* que inicializa a simulação antes de arrancar o ciclo principal de processamento dos eventos.

7.2.1 O *SimMovie* (efeito de movimento)

Certas entidades da simulação usam o efeito de movimento para dar ao utilizador uma ideia mais "real" de como certos processos se desenvolvem na prática. É o caso dos tapetes e dos veículos que aparecem a mover-se no ecrã. Este processo, no entanto, nem sempre é requerido pelo utilizador da aplicação, principalmente quando se trata de simular a passagem do tempo a uma grande "velocidade". Tal levou a conceber um novo tipo de objecto dedicado somente à movimentação, o *SimMovie*, reservando-lhe um evento responsável por despoletar esse mecanismo de representação do movimento (SIM_SHOW). Se esse objecto for criado na simulação o processo de movimento é activado, caso contrário, não poderão apreciar-se no ecrã os objectos a moverem-se. A estrutura deste objecto é a seguinte:

```
class SimMovie : public SimEntity
{
 public:

    SimMovie(ArmSimulation* pSim); //Construtor

    virtual BOOL Executa(UINT event); //Encaminhador de eventos

    BOOL ShowON(); //Resposta ao evento SIM_SHOW
};
```

Como se pode verificar, este objecto possui as características de uma entidade activa da simulação, uma vez que deliberadamente se fez derivar da classe *SimEntity*, e é responsável por processar o evento SIM_SHOW através do seu método *ShowON()*. Na criação deste objecto é-lhe passado um apontador para o *Simulador*, através do qual o *SimMovie* pode aceder à lista de objectos em movimento, propriedade do próprio *Simulador*.
Na próxima figura está representado o mecanismo do *Simulador* subjacente ao processo de movimentação dessas entidades no ecrã, mecanismo só activado se no início da simulação foi criado um objecto *SimMovie*.

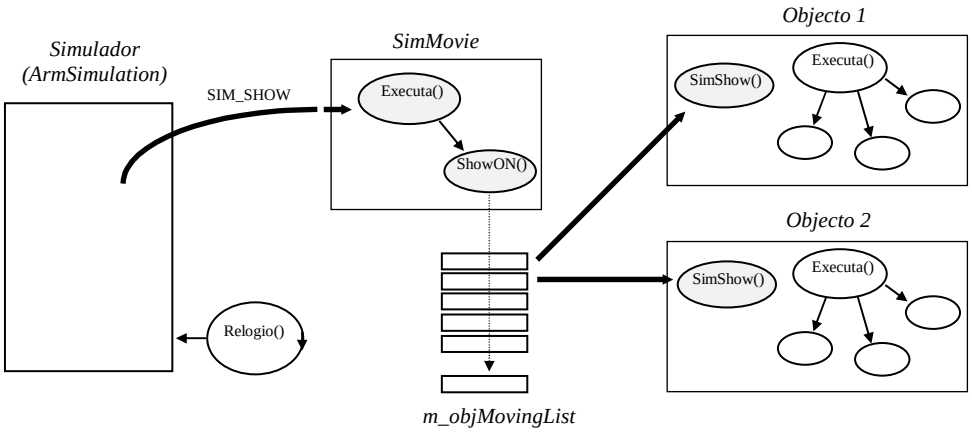

Fig. 7.4 Mecanismo do Simulador para movimentação das entidades no ecrã.

Segundo este mecanismo, quando no "Relógio" da simulação surge um evento SIM_SHOW, o *Simulador* passa-o ao objecto *SimMovie* chamando o respectivo método *Executa()*. Este, por sua vez, manda executar o método *ShowON()* de resposta a este evento, cuja actividade se resume a, em primeiro lugar, percorrer a *lista de objectos que se encontram em movimento* na simulação (`m_objMovingList`) e mandar executar a cada um desses objectos o respectivo método *SimShow()*, que trata de actualizar a posição do objecto e de o fazer representar graficamente. Em segundo lugar, o *ShowON()* é responsável por marcar o próximo evento SIM_SHOW na *lista de eventos* da simulação, o que transforma este objecto no "motor" dele próprio. Este método é apresentado a seguir.

```
BOOL SimMovie::ShowON()
{
    SimEntity* pObject;

    ////////////////////////////////////////////////
    //  Percorre a lista de objectos em movimento e fá-los representar no ecrã.
    ////////////////////////////////////////////////
    POSITION pos = m_pSim->m_objMovingList.GetHeadPosition();
    while(pos != NULL)
           {
            pObject = m_pSim->m_objMovingList.GetNext(pos);
            pObject->SimShow();
           }

    ////////////////////////////////////
    //  Executa um som para dar a sensação de trabalho
    ////////////////////////////////////
    if(m_pSim->m_objMovingList.GetCount() > 0)
           sndPlaySound("som.wav", SND_ASYNC);

    ////////////////////////////////////////////////
    //  Marca o próximo evento SIM_SHOW
    ////////////////////////////////////////////////
    float DT = m_pSim->m_DTime;
    m_pSim->Schedule( m_pSim->m_time + DT, SIM_SHOW, this);

    return TRUE;
}
```

7.2.2 O *SimControl* (Sistema de Controle)

Um outro método importante do *Simulador* é o *SimControl()*. Este método é usado por determinados objectos como órgão decisor em circunstâncias específicas, tal como, por exemplo, por um tapete no momento em que uma palete chega a um dos seus extremos. Por isso, o *SimControl()* pode ser entendido como um bloco de controle onde se localizam os critérios de decisão geralmente incluídos nos PLCs (autómatos) de uma instalação real. Claro que essa lógica poderia ser embutida nos métodos dos próprios objectos, mas dessa forma esses critérios estariam dispersos, e não concentrados num só bloco, o que dificultaria a sua possível alteração.

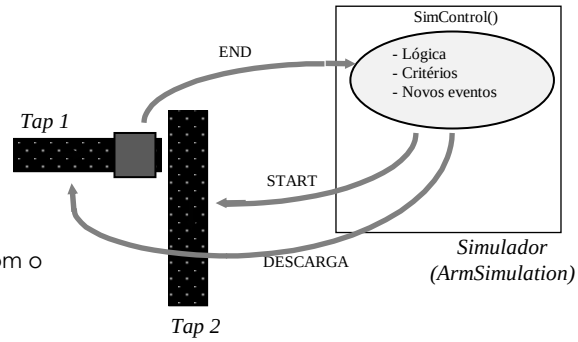

Fig. 7.5 Diálogo de eventos com o *SimControl* da simulação.

Na figura 7.5 é apresentado um exemplo do processo despoletado quando uma palete atinge o fim do tapete `tap1`. Este tapete envia ao *SimControl* o evento END e o *SimControl* decide então o que fazer, conforme o caso. Neste caso manda arrancar o tapete `Tap 2`, enviando-lhe o evento START, e dá ordem ao `Tap 1` para executar a descarga da palete, dirigindo-lhe o evento DESCARGA. Por isso, o *SimControl* deverá manter aglomerada toda a lógica de controle, de maneira a que possa decidir as acções a tomar numa determinada situação particular. De qualquer maneira, este método é actualmente dirigido principalmente aos tapetes, uma vez que os veículos integram o seu próprio sistema de controle, tal como na realidade incluem um PLC dedicado embutido na sua estrutura mecânica.
Segundo esta abordagem, se se pretenderem alterar os critérios envolvidos neste nível de lógica, basta simplesmente alterar-se o método *SimControl()* do *Simulador* sem que seja necessário mexer-se no resto da aplicação.
A seguir apresenta-se o método *SimControl()* implementando o critério de decisão relativamente à chegada de uma palete ao fim de um tapete.

```
void ArmSimulation::SimControl(float time,UINT event,SimEntity* pEnt)
{

  if(pEnt->m_tipo == AREA_TAPETE) // Se a entidade que envia o evento é um tapete:
    {
      Tapete* pTap = (Tapete*)pEnt;
```

```
        switch(event)
        {
         case TAP_END:
         ////////////////////////////////////////////////////////
         // Critério a seguir quando uma palete atinge o fim do tapete:
         // Se existe entidade para onde este tapete possa descarregar e essa entidade estiver
         // pronta para  receber a palete, então manda efectuar a Descarga() , caso
         // contrário, coloca este tapete em estado WAITING de espera por descarga.
         // Note-se que a descarga se poderá fazer para dois tipos de entidades: para um outro
         // tapete ou para uma mesa. Para se poder descarregar para outro tapete basta que
         // este se encontre no estado STOPPED, isto é, parado e sem paletes. Para o caso de
         // uma mesa considera-se que é sempre possível executar a descarga, enviando a palete
         // para a lista de paletes dessa mesa.
         // Se este tapete não puder executar a descarga, é colocado numa lista de objectos à
         // espera de descarga onde aguardará por uma futura ordem.
         ////////////////////////////////////////////////////////

         float dt = pTap->m_tDescarga;

         if(pTap->m_pEntDescarg != NULL)
           {
            //Caso de descarga para TAPETE:
            if(pTap->m_pEntDescarg->m_tipo == AREA_TAPETE)
              {
                if(pTap->m_pEntDescarg->m_estado == STOPPED) //descarrega
                   m_pSim->Schedule(time+dt, TAP_DESCARGA, pTap);

                else //Espera
                    {
                     pTap->m_estado = WAITING;
                     m_objWaitList.AddTail(pTap);
                      POSITION pos = m_objMovingList.Find(pTap);
                     if(pos!=NULL) m_objMovingList.RemoveAt(pos);
                    }
              }

            //Caso de descarga para MESA:
            else  m_pSim->Schedule(time+dt, TAP_DESCARGA, pTap);

           }

        //não existe entidade para descarga: descarrega para o chão.
        else  m_pSim->Schedule(time+dt, TAP_DESCARGA, pTap);


        break;
        default:break;
       }
      }//Entidade que enviou o evento foi um tapete…
}
```

7.2.3 O *SimMaster* (Sistema de Gestão)

Um outro método importante, senão o principal do *Simulador*, é o *SimMaster()*. Este método pretende fazer-se corresponder ao sistema de gestão real e é responsável por manter a informação sobre os produtos, assegurar a distribuição de tarefas e zelar pela boa comunicação com o exterior do armazém. É ele que trata de "desenrolar" as ordens de saída e de entrada de material, de escolher as células envolvidas nos processos de transferência desse material, de calcular os melhores percursos e de atribuir as tarefas aos elementos de transporte. Por isso

é no *SimMaster* que se deverão localizar os critérios que se relacionam com a gestão do armazém.
Claro que estes critérios não são únicos, dependendo da filosofia subjacente à concepção do armazém, mas o que é importante é que eles possam encontrar-se concentrados num só bloco da aplicação, pois caso necessitem de ser alterados, ou mesmo substituídos, facilmente se poderá levar a cabo essa tarefa. Inclusive, pensa-se deixar para um futuro trabalho o desenvolvimento de um processo que permita modificar esses critérios sem que seja necessário voltar a compilar o código da aplicação…
Como "gestor" do armazém, o *SimMaster* é também o principal responsável pela utilização e manutenção das listas de objectos que participam na simulação, usando-as para manter uma "imagem" global do estado do sistema e para, com base nesse estado, tomar certas decisões.

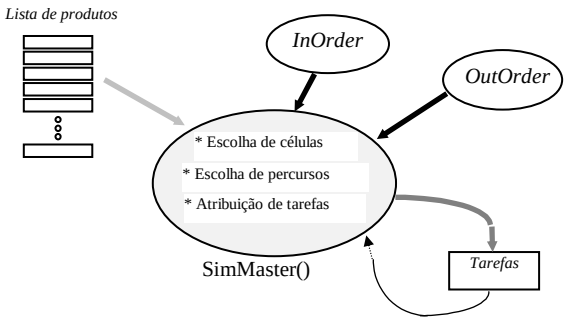

Fig. 7.6 Critérios básicos envolvidos no *SimMaster*.

De qualquer forma, as funções mais importantes do *SimMaster* referem-se ao processamento das *Ordens de Entrada* e *de Saída de material*, processamento que envolve o manejo de informação sobre os produtos, com vista a determinarem-se as células para onde o material deve ser transferido e de forma a criarem-se as respectivas tarefas a serem atribuídas aos elementos de transporte (fig. 7.6). Por este motivo, o *SimMaster* usa com grande frequência a *lista de produtos* da simulação.
Também porque os critérios de gestão se encontram neste bloco, certos mecanismos de outros objectos redireccionam, por vezes, os eventos para o *SimMaster*, esperando que este depois se encarregue do resto do processo. É o caso da circulação de uma *tarefa* de movimentação de material entre vários elementos de transporte: cada elemento reserva-se a execução da parte da *tarefa* que lhe compete e depois devolve-a ao *SimMaster* para que este se responsabilize pelo passo seguinte.
Para se ficar com uma ideia mais concreta da participação do *SimMaster* num destes processos de transferência de material, consideremos de novo o caso do transporte de uma palete desde uma célula de início até uma célula de fim representado na próxima figura.

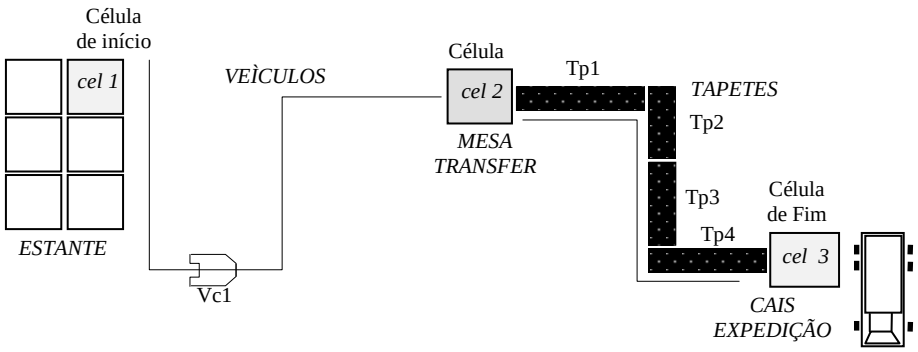

Fig. 7.7 Exemplo de um processo genérico de transferência de material.

No início, o *SimMaster* cria a seguinte *tarefa* que atribui ao veículo `vc1` por verificar ser ele que pode aceder à célula `cel1`.

Tipo de Subtarefa	*Unidades a movimentar*	*Tipo de unidade*	*Produto da unidade*	*Célula*	
LOAD	1	PALETE	P1	**Cel1**	
UNLOAD	1	PALETE	P1	**Cel2**	**FIM**
LOAD	1	PALETE	P1	**Cel2**	
UNLOAD	1	PALETE	P1	**Cel3**	

Sendo assim, este veículo começa por se dirigir para o ponto de acesso à célula `cel1` onde depois carrega (LOAD) um palete do produto `P1`. Em seguida, conforme lhe é indicado pela tarefa, dirige-se para a célula `cel2` e aí descarrega (UNLOAD) essa palete, eliminando então as linhas de LOAD já executadas da *tarefa* e passando-a de novo ao *SimMaster* na seguinte forma:

Tipo de Subtarefa	*Unidades a movimentar*	*Tipo de unidade*	*Produto da unidade*	*Célula*	**UNLOAD Já executado**
UNLOAD	1	PALETE	P1	**Cel2**	
LOAD	1	PALETE	P1	**Cel2**	**UNLOAD (fim)**
UNLOAD	1	PALETE	P1	**Cel3**	

O *SimMaster*, ao receber de volta esta *tarefa* do veículo, fica a saber que já foi depositada a palete na célula `cel2` e, por outro lado, verifica que a tarefa ainda não está completa. Por isso, procura a entidade que poderá cumprir a próxima parte da tarefa, isto é, que seja capaz de carregar o material da célula `cel2` e de o transportar até à célula `cel3`. Estudando a malha de percursos e baseando-se em determinados critérios, o *SimMaster* decide atribuir esta tarefa ao tapete `Tp1`, depois de eliminar a linha de UNLOAD inicial que dizia respeito ao passo anterior:

Tipo de Subtarefa	*Unidades a movimentar*	*Tipo de unidade*	*Produto da unidade*	*Célula*	
LOAD	1	PALETE	P1	**Cel2**	**UNLOAD (fim)**
UNLOAD	1	PALETE	P1	**Cel3**	

Assim, este tapete carrega a palete da `cel2` e transporta-a até ao tapete `Tp2`, no qual a descarrega e ao qual passa a responsabilidade da tarefa em causa. Este, por sua vez, transporta a palete até `Tap3`, onde de novo ela é descarregada e a tarefa passada como um testemunho. Depois o `Tap3` descarregará o material em `Tap4` e este, por fim, visto que a sua entidade de descarga coincide com a célula de destino, depositará a referida palete na célula `cel3`, terminando assim a tarefa e avisando o *SimMaster* de que a missão foi cumprida.
Adiante veremos como o *SimMaster* interfere no processamento das *InOrders* e das *OutOrders*. Para já fique-se com o tipo de estrutura usada no *SimMaster*, e com o exemplo do caso particular da interpretação de um evento FIM_TAREFA proveniente de uma entidade da classe *Transp*.

```
void ArmSimulation::SimMaster(float time,UINT event,SimEntity* pEnt)
{

 /*****************************************************************
   Se a entidade que envia o evento é da classe Transp:
 *****************************************************************/

 if(pEnt->m_tipo == AREA_TRANSP)
   {
     Transp* pTranp = (Transp*)pEnt;
     switch(event)
     {
      case FIM_TAREFA:
            ////////////////////////////////////////
            // Critério a seguir quando um veículo fica LIVRE:
            ////////////////////////////////////////

            ////////////////////////////////////////
            // Aqui se implementa o critério a seguir neste caso. Para já,
            // deve-se verificar se existe outra tarefa que se possa atribuir
            // a esse veículo. Se não existir essa tarefa, o veículo é mandado
            // recolher ao seu parque de estacionamento.
            ////////////////////////////////////////
       break;

       default:break;
      }
   }//Foi um Transp


 /*****************************************************************
   Se a entidade que envia o evento é da classe Tapete:
 *****************************************************************/

 else if(pEnt->m_tipo == AREA_TAPETE)
   {
     Tapete* pTap = (Tapete*)pEnt;
     switch(event)
     {
      case FIM_TAREFA:
            ////////////////////////////////////////
            // Critério a seguir quando um tapete finda uma tarefa:
            ////////////////////////////////////////

            ////////////////////////////////////////
            // Aqui se implementa o critério a seguir neste caso.
            ////////////////////////////////////////
       break;

       default:break;
      }
```

```
    }//Foi um Tapete

  /*****************************************************************
    Se a entidade que envia o evento é da classe ...:  (Código neste momento não apresentado)
  *****************************************************************/
                                        ...
}
```

7.3 Actividade dos elementos de simulação

Aos elementos activos da simulação cabe a responsabilidade da actividade do sistema, uma vez que os seus eventos despoletam acções e essas acções alteram, na maior parte das vezes, o estado de uma ou várias entidades. Se, por qualquer razão, não existissem eventos, o sistema manter-se-ia para sempre no mesmo estado, o que significa que as entidades passivas da simulação (estantes, células, mesas, etc.) não promovem qualquer tipo de actividade, apesar de poderem encontrar-se em estados diferentes em instantes diferentes do tempo.
Por isso, dedicou-se esta secção à apresentação dos eventos e dos respectivos métodos dos elementos activos, pois neles se encontram encapsulados os mecanismos que promovem o desenrolar da simulação.
Antes de mais, no entanto, convém recordar que os eventos passíveis de acontecerem na simulação são predefinidos no sistema, isto é, o sistema conhece de antemão todos os seus eventos. Por isso eles foram definidos numa secção de `#define` incluída pelo código do simulador, e o valor numérico associado a cada um corresponde à sua prioridade de execução. Desta forma, quando dois eventos são para acontecer no mesmo instante de tempo na simulação, ao serem marcados na *lista de eventos* através do método *Schedule()* automaticamente este método se encarrega de os organizar segundo a respectiva prioridade.
Por outro lado, a cada evento estará directamente associado um só método, podendo este utilizar chamadas a outros indirectamente, seja do próprio objecto, seja do *Simulador* da aplicação. Ainda, cada objecto possui no seu interior uma variável para registo do último instante de tempo em que foi chamado a intervir na simulação (`m_time`), o que lhe permitirá realizar determinados cálculos importantes.

7.3.1 Eventos e métodos dos veículos

Depois de identificadas as funções de um veículo, decidiu-se atribuir a este tipo de objectos um conjunto de eventos que representasse os acontecimentos possíveis de se lhes afectar durante a simulação. Daí resultou considerarem-se seis eventos, através dos quais foi possível modelar o seu comportamento tanto nos processos de execução de tarefas como nos de circulação e acesso em vias de movimentação, estreitas ou normais. Desta forma, um veículo é visto pelo *Simulador* como uma entidade com a seguinte estrutura de métodos:

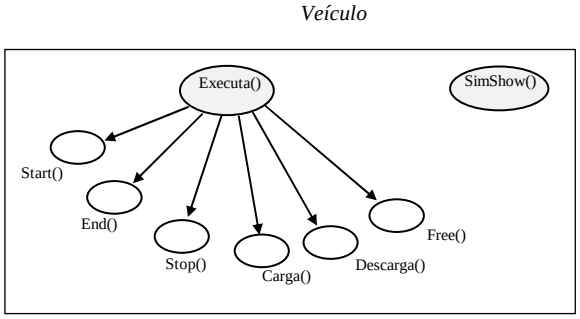

Fig. 7.8 Estrutura da entidade de simulação *Veículo*.

Como em qualquer entidade activa exigindo representação gráfica, o veículo faz uso do método de acesso público *SimShow()*, característico da classe *SimEntity*, a partir do qual o *Simulador* sincroniza a sua representação. Refira-se que é neste método que é implementado o algoritmo de movimentação descrito anteriormente na secção 6.1.11 reservada aos veículos.
Para além disso, e para além do "encaminhador de eventos" típico de qualquer entidade activa na simulação, ao veículo associaram-se os seis métodos de resposta a eventos assinalados na figura 7.8. Os eventos foram baptizados com o nome do respectivo método precedido de VEIC_:

- VEIC_START (evento que identifica o instante em que o veículo rearranca.)
- VEIC_END (instante em que o veículo chega ao fim de um determinado tramo)
- VEIC_STOP (instante em que é o veículo chega ao fim do percurso e pára)
- VEIC_CARGA (instante em que é iniciado a carga de material)
- VEIC_DESCARGA (instante em que é iniciado a descarga do material)
- VEIC_FREE (instante em que é o veículo se considera LIVRE para novas tarefas)

Como já foi referido, a modelação dos veículos baseia-se na utilização da classe *Transp*, por isso, os métodos que processam estes eventos fazem parte da estrutura desta classe.

Convém recordar neste momento que um veículo possui um "percurso actual", representando o caminho por onde se deverá movimentar para atingir o seu destino, e uma "tarefa" que é constituída por um conjunto de objectivos a serem cumpridos por uma certa ordem. O percurso é guardado na variável `m_actPercurso` e a tarefa na variável `m_tarefa`. A tarefa é previamente elaborada pelo *SimMaster* e em seguida atribuída ao veículo, imediatamente antes de a este ser enviado um evento VEIC_START. Note-se que o evento VEIC_START é usado pelo *Simulador* para avisar o veículo de que deve iniciar a executar a tarefa que lhe foi distribuída, no entanto, poderá também ser gerado por outro tipo de objectos, inclusive pelo próprio veículo, em certas circunstâncias de re-interpretação da tarefa ou de reavaliação do percurso.
As acções relacionadas com os eventos dos veículos poderão ser melhor entendidas com o apoio do esquema apresentado na próxima figura – note-se que o tramo `Tr.5`, representado a tracejado, é um tramo do tipo ESTREITO.

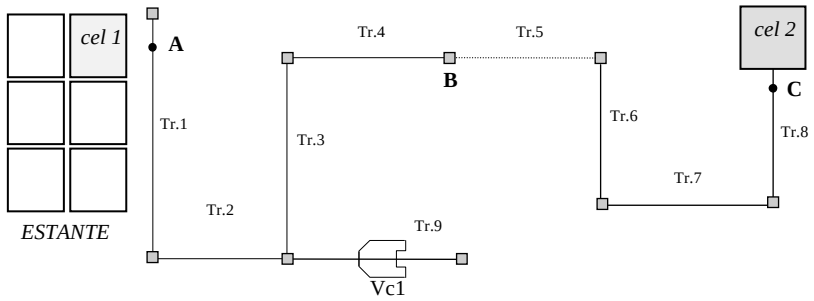

Fig. 7.9 Exemplo de uma tarefa de um *Veículo*.

Suponhamos então que o veículo `Vc1` se encontra inicialmente num tramo qualquer (`Tr.9`) e que é incumbido de realizar a tarefa de carregar uma palete da célula `cel1` e de a transportar até à célula `cel2` onde depois a deve depositar. Quando o *SimMaster* atribui esta tarefa ao veículo já sabe que este tem acesso a ambas as células, por isso o próprio veículo não necessita de se incomodar com de novo realizar esses cálculos.
Recebido o evento VEIC_START, em primeiro lugar o veículo deve preparar-se para executar a subtarefa de carregar a palete de `cel1`. Para isso, começa por determinar qual dos seus tramos dá acesso a esta célula e deduz ser o tramo `Tr.1`. Determina depois o valor de λ que corresponde ao ponto de acesso à célula (ponto **A**), a que designaremos λ_A. Assim, ficam estabelecidos os valores do tramo e do λ de destino:

```
m_TramoDest = Tr.1  //Tramo de destino
m_lambdaDest = λA   //λ no tramo de destino
```

Depois constrói um "percurso actual", que guardará na variável `m_actPercurso`, de forma a mover-se até ao ponto **A**. Esse percurso será constituído pela sucessão de tramos: `Tr.9, Tr.2, Tr.1`.
Uma vez identificado o seu destino, o veículo está pronto a arrancar.
Uma vez que ainda não se encontra no tramo de destino, o veículo calcula o tempo que vai demorar a abandonar o tramo actual, e marca então para esse instante um evento VEIC_END que, como já se referiu, será responsável pela próxima avaliação da situação. Digamos que este evento está associado ao instante em que o veículo muda de tramo, altura em que de novo é verificado se já se encontra no tramo de destino. No caso da figura isso não acontece, portanto, de novo é executado, imediatamente, um evento VEIC_START que permitirá ao móvel continuar a movimentar-se percorrendo o tramo Tr.2 até ao fim, altura em que um novo evento VEIC_END será processado. Nessa altura o veiculo prepara-se para entrar no tramo de destino (referente à primeira subtarefa), e então será marcado um evento VEIC_STOP para assinalar o momento em que o veículo atinge o ponto **A** de destino.
Chegado esse instante, o veículo é levado a interpretar a subtarefa em questão, donde deduz tratar-se de uma acção de LOAD (carga). Por isso, marca para esse instante um evento VEIC_CARGA. Este evento, é responsável por retirar uma

palete da célula `cel1` e de a colocar no veículo, eliminando depois a subtarefa que acaba de ser cumprida. Por fim, conhecido o tempo de carga registado na variável `m_timeCarregar`, é calculado o instante de tempo em que se considerará terminada essa operação e marcado para esse instante um novo evento VEIC_START.
A tarefa é agora constituída por uma única linha de UNLOAD (descarga), e, quando de novo for reavaliada pelo veículo, já a situação será diferente da anterior. O veículo determina qual dos seus tramos dá acesso à célula `cel2` e deduz ser o tramo `Tr.8`. Determina depois o valor de λ que corresponde ao ponto de acesso à célula (ponto **C**), a que designaremos λ_C. Assim, estabelece os novos valores do tramo e do λ de destino:

```
m_TramoDest = Tr.8  //Tramo de destino
m_lambdaDest = λC   //λ no tramo de destino
```

Depois constrói um "percurso actual", que guardará na variável `m_actPercurso`, de forma a mover-se até ao ponto **C**. Esse percurso será constituído pela sucessão de tramos: `Tr.1, Tr.2, Tr.3, Tr.4, **Tr.5**, Tr.6, Tr.7, Tr.8`. Repare-se que o tramo **`Tr.5`** é do tipo ESTREITO.
O veículo continua a executar uma sucessão de eventos VEIC_END e VEIC_START até atingir o tramo **`Tr.5`**, tramo que poderá estar ocupado por outro veículo no momento em que este o desejar aceder. Por essa razão, o evento VEIC_END marcado para o instante de chegada ao ponto **B** é responsável por verificar o estado desse tramo, podendo resultar em dois procedimentos diferentes: se o referido tramo estiver ocupado então coloca este veículo na lista de veículos desse tramo, alterando o estado do veículo para WAITING. Se o tramo estiver livre, o veículo entra no tramo e executa imediatamente um novo evento VEIC_START que irá reavaliar a situação. Repare-se que será da responsabilidade do veículo que sai do tramo ESTREITO alertar o veículo que espera entrar nesse tramo. Esse aviso é feito através de um evento VEIC_START dirigido ao primeiro veículo na fila de espera desse tramo.
Quando for atingido o ponto **C** um novo evento VEIC_STOP será processado, despoletando agora um evento de descarga VEIC_DESCARGA. Findo o tempo de descarga ocorrerá o evento VEIC_FREE, através do qual o veículo se encarrega de avisar o *SimMaster* restituindo-lhe o resto da tarefa.
A partir desse instante, e tratando-se de um *AGV*, o *SimMaster* verifica se é necessário mandar o veículo carregar as baterias. Se for esse o caso, então a sua variável `m_tipoTarefa` é colocada em BATERY e encarregado de executar um novo evento VEIC_START para que se dirija ao parque e carregue as baterias. No fim desse processo é executado um evento VEIC_FREE.
Se não necessitar de carregar baterias, ou não se tratando de um *AGV*, o *SimMaster* verifica se existe alguma nova tarefa que lhe possa atribuir. Se essa tarefa existir, então atribui-a ao veículo e volta a dirigir-lhe um evento VEIC_START para que de novo todo o processo recomece. No caso de não existir tarefa para o veículo, este é enviado para o seu parque através dos procedimentos:

```
veiculo::m_tipoTarefa = PARK;
veiculo::Executa(VEIC_START);
```

Repare-se que a um veículo se podem atribuir três tipos de missões distintas, e que o evento VEIC_START é interpretado de maneira um pouco diferente conforme a função atribuída ao veículo. Essas missões podem ser

JOB	- tarefa de transferência de material
PARK	- retorno ao parque de estacionamento
BATERY	- retorno ao parque para carregar baterias

Se a missão que lhe foi distribuída é do tipo JOB, então ele fica a saber que deverá cumprir uma tarefa de transferência de material. Mas se a missão for do tipo PARK ou BATERY, simplesmente o veículo deverá dirigir-se para o seu parque de estacionamento, aguardando por novas ordens no primeiro caso, ou carregando as baterias e avisando o *SimMaster* através de um evento VEIC_FREE no segundo. Em quaisquer destas duas últimas missões, o veículo ignora o conteúdo da sua variável `m_tarefa`.
Apresentados os eventos associados aos veículos, passemos então à descrição dos respectivos métodos:

7.3.1.1 *Evento VEIC_START*

Este evento é responsável por assegurar a movimentação do veículo tramo a tramo ao longo do "percurso actual" previamente registado na variável `m_actPercurso`. Esse registo é feito por este mesmo método sempre que o veículo se encontre parado (STOPPED) e seja mandado iniciar o movimento. Por isso este evento destina-se também a promover o arranque do veículo, calculando-se o "percurso" a ser seguido até que seja atingido o destino. Esse cálculo depende do tipo de missão a que o veículo foi destinado. Tratando-se de uma missão de PARK ou de BATERY, o percurso é calculado com base na localização do seu parque de estacionamento. Se, por outro lado, a missão for do tipo JOB, o cálculo inicial deste "percurso" baseia-se nas coordenadas da célula que está envolvida na subtarefa a processar naquele momento.

Fique-se então com o código comentado deste método:

```
BOOL Transp::Start()
{
    m_time = m_pSim->m_time; //tempo do veículo é o tempo actual do Simulador

    ////////////////////////////////////////////////////////
    // Se o veículo estiver parado, inicia o arranque, determinando o tramo de destino e o
    // λ de destino. Se a missão é do tipo JOB, a determinação destes parâmetros faz-se
    // através da detecção do tramo que dá acesso à célula envolvida na Subtarefa actual.
    // Se a missão for de outro tipo, estes parâmetros são determinados pelo tramo de
    // acesso ao parque do veículo. Depois calcula o percurso até ao tramo final e guarda-o
    // na variável m_actPercurso.
    ////////////////////////////////////////////////////////

    if(m_estado == STOPPED)
```

```
 {
  CalculaDestino(); //actualiza m_lambdaDest
  CalculaPercurso();//actualiza m_percurso

  ////////////////////////////////////////
  //  Actualiza o sentido com que deverá percorrer este tramo.
  ////////////////////////////////////////

  m_sentido = CalcSentido(m_actTramo);
 }

////////////////////////////////////////////////////////////
// Se o tramo actual for diferente do tramo de destino, calcula o tempo (dT) que demora
// a sair do tramo actual e marca um evento VEIC_END para o instante m_time+dT.
////////////////////////////////////////////////////////////

if(m_actTramo != m_tramoDestino)
  {
       //tempo para sair do tramo actual:
       dT = CalcTimeToEnd(m_actTramo);

       //Marca evento VEIC_END:
       m_pSim->Schedule(m_time+dT, VEIC_END, this);
  }

////////////////////////////////////////////////////////
// Se o tramo actual for igual ao tramo de destino, calcula o tempo (dT) que demora
// a atingir o destino neste tramo e marca um evento VEIC_STOP para o instante
// em que tal acontecerá (m_time+dT).
////////////////////////////////////////////////////////

else
    {
       //tempo para sair do tramo actual:
       dT = CalcTimeToStop(m_actTramo);

       //Marca evento VEIC_STOP:
       m_pSim->Schedule(m_time+dT, VEIC_STOP, this);
    }


m_estado = MOVING; //veículo em movimento

//coloca o veículo na lista de movimento da simulação.
m_pSim->m_objMovingList.AddTail(this);

return TRUE;
}
```

7.3.1.2 *Evento VEIC_END*

Este evento surge sempre que o veículo se encontra a abandonar um tramo e a dirigir-se para outro. Trata-se de um evento de mudança de tramo, e destina-se principalmente, a lidar com as situações que advêm da existência de tramos ESTREITOS. É a este método que cabe a responsabilidade de verificar o tipo do tramo que abandona, alertando o próximo veículo à espera caso o tramo seja ESTREITO. É ainda aqui que se verifica o tipo do tramo para onde o veículo está prestes a dirigir-se e, com base nisso, se decidem as acções a executar.

```
BOOL Transp::End()
{
```

```
m_time = m_pSim->m_time; //tempo do veículo é o tempo actual do Simulador

////////////////////////////////////////////////////////
// Se o tramo actual for do tipo estreito, então é necessário avisar o próximo veículo à
// espera deste tramo que pode entrar. É-lhe por isso enviado um evento VEIC_START
// depois de ser retirado da lista de espera deste tramo.
////////////////////////////////////////////////////////

if(m_actTramo->m_tipoVia == ESTREITA)
  {
    if(!m_actTramo->m_waitList.IsEmpty()) // se existe veículo à espera
      {
       Transp* pVeiculo = m_actTramo->m_waitList.RemoveHead();
       pVeiculo->Executa(VEIC_START);
       }
  }

////////////////////////////////////////////////
// Coloca o tramo actual igual ao próximo tramo do percurso actual,
////////////////////////////////////////////////

POSITION pos = m_actPercurso.Find(m_actTramo);
m_actPercurso.GetNext(pos); //tramo actual
m_actTramo = m_actPercurso.GetNext(pos); //próximo tramo

////////////////////////////////////////////////
// Actualiza o sentido com que deverá percorrer este tramo e actualiza
// o valor da varável m_actLambda.
////////////////////////////////////////////////

m_sentido = CalcSentido(m_actTramo);
m_actLambda = (m_sentido == DIRECT)? 0: 1;

////////////////////////////////////////////////////////
// Se este novo tramo actual for do tipo estreito, então é necessário verificar se ele já se
// encontra ocupado. Se for esse o caso, então coloca este veículo na lista de espera
// desse tramo, retira-o da lista de objectos em movimento na simulação e coloca o seu
// estado em WAITING. Se o tramo estiver livre, então envia imediatamente a este
// veículo uma ordem de Start().
////////////////////////////////////////////////////////

if(m_actTramo->m_tipoVia == ESTREITA)
  {
    if(!m_actTramo->m_LIVRE) // se o tramo está ocupado
      {
        //retira o veículo na lista de movimento da simulação, se ele lá estiver.
        pos = m_pSim->m_objMovingList.Find(this);
        if(pos != NULL) m_pSim->m_objMovingList.RemoveAt(pos);

        //coloca o veículo no fim da lista de espera do tramo.
        m_actTramo->m_waitList.AddTail(this);

        m_estado = WAITING; //veículo em espera.
      }
    else //tramo está livre
        {
          //executa imediatamente um VEIC_START
          this->Start();
        }
  }

////////////////////////////////////////////////
// Se o tramo actual for NORMAL, então envia imediatamente a este
// veículo uma ordem de Start().
////////////////////////////////////////////////
```

```
    else  //tramo actual é um tramo normal
        {
           //executa imediatamente um VEIC_START
           this->Start();
        }

    return TRUE;
}
```

7.3.1.3 *Evento VEIC_STOP*

Este evento surge no instante em que é atingida a posição de destino no percurso do veículo. O seu método encarrega-se de fazer interpretar a missão atribuída ao móvel e, se for um caso de JOB, de mandar iniciar a operação de carga ou de descarga de material, conforme o indicado na actual subtarefa em execução. Este evento é marcado pelo método que processa o evento VEIC_START quando o veículo se encontra no último tramo do percurso.

```
BOOL Transp::Stop()
{
    m_time = m_pSim->m_time;   //tempo do veículo é o tempo actual do Simulador

    m_estado = STOPPED;        //veículo parado.

    ////////////////////////////////////////////////
    //  Repõe sentido directo de percorrer este tramo e actualiza
    //  o valor da variável m_actLambda.
    ////////////////////////////////////////////////

    m_sentido = DIRECT;
    m_actLambda = m_lambdaDest; //Chegou ao destino

    //retira o veículo na lista de movimento da simulação, se ele lá estiver.

    POSITION pos = m_pSim->m_objMovingList.Find(this);
    if(pos != NULL) m_pSim->m_objMovingList.RemoveAt(pos);

    ////////////////////////////////////////////////////////////////
    //  Se a tarefa do veículo é JOB, então retira da tarefa a SubTarefa actual  e interpreta-a.
    //  se essa SubTarefa for de LOAD marca um evento de VEIC_CARGA para ser executado
    //  imediatamente, se for de UNLOAD marca o evento de VEIC_DESCARGA.
    ////////////////////////////////////////////////////////////////

    if(m_tipoTarefa == JOB)
      {
           if(!m_tarefa->IsEmpty())
           {
             m_subTarefa = m_tarefa->RemoveHead();

            if(m_subTarefa->m_tipoSubTarefa == LOAD)
                 this->Carga();

            else if(m_subTarefa->m_tipoSubTarefa == UNLOAD)
                        this->Descarga();

           else //erro..
                 m_pSim->MessageBox("Tarefa desconhecida!");
           }
      }

    ////////////////////////////////////////////////////////////////
```

```
    // Se a tarefa do veículo for PARK, então o veículo não usa a informação da tarefa. Antes,
    // estaciona no parque e transfere um evento VEIC_FREE ao SimMaster para este saber que
    // pode contar com ele para novas tarefas.
    ////////////////////////////////////////////////////////////

    else if(m_tipoTarefa == PARK)
           m_pSim->SimMaster(m_time, VEIC_FREE, this);

    ////////////////////////////////////////////////////////////
    // Se a tarefa do veículo for BATERY, então o veículo também não usa a sua tarefa.
    // Estaciona no parque e carrega as baterias durante o tempo m_timeBatery, marcando
    // para o fim desse tempo um evento VEIC_FREE.
    ////////////////////////////////////////////////////////////

    else if(m_tipoTarefa == BATERY)
           m_pSim->Schedule(m_time+m_timeBatery, VEIC_FREE, this);


    return TRUE;
}
```

7.3.1.4 *Evento VEIC_CARGA*

Este evento marca o inicio do processo de carga de material da célula de destino para a palete do veículo. Relembre-se que o veículo possui na sua estrutura de objecto uma palete para esse efeito. Uma vez chegado à célula referenciada na subtarefa em execução, o veículo retira dessa célula o material e deposita-o na sua palete.
Através da interpretação dos vários campos da subtarefa é decidido que tipo de unidades se devem carregar: caixas ou palete. No primeiro caso, são retiradas da célula o número requerido de caixas e adicionadas às que já existem na palete do veículo. No caso de se tratar de uma palete, toda a informação da palete retirada da célula é transferida para a palete do veículo, eliminando em seguida o objecto palete que foi retirado da célula.
Finda a operação de carga é marcado um novo evento VEIC_START para que seja interpretada a próxima subtarefa da *tarefa* distribuída ao veículo.

```
BOOL Transp::Carga()
{
    m_time = m_pSim->m_time;   //tempo do veículo é o tempo actual do Simulador

    ////////////////////////////////////////////////////////////
    // Se a SubTarefa actual for do tipo PALETE, então retira uma palete da respectiva célula
    // e coloca-a na palete do veículo.
    // Se a SubTarefa actual for do tipo CAIXAS, então retira da célula respectiva o número de
    // caixas exigidas na subtarefa e adiciona-as às existentes na palete do veículo.
    ////////////////////////////////////////////////////////////

    Palete* pPal;

    Celula* pCel = m_subTarefa->m_celula;

    if(m_subTarefa->m_tipoUnidades == PALETE) //PALETE
       {
         pPal = pCel->GetPalete();
         m_palete.m_largura = pPal->m_largura;
         m_palete.m_profund = pPal->m_profund;
```

```
            m_palete.m_END = FALSE;
            m_palete.m_tipoPal = pPal->m_tipoPal;
            m_palete.m_maxCaixas = pPal->m_maxCaixas;
            m_palete.m_nCaixas = pPal->m_nCaixas;
            m_palete.m_ref = pPal->m_ref;
            m_palete.m_produto = pPal->m_produto;
            m_palete.m_escala = pPal->m_escala;

            delete pPal;
           }
        else //CAIXAS
           {
            UINT nCaixas = pCel->GetCaixas(m_subTarefa->m_nUnidades);
            m_palete.m_tipoPal = CAIXAS;
            m_palete.m_nCaixas += nCaixas;

            ////////////////////////////////////////////////
            // O método GetCaixas() da célula é responsável por actualizar as variáveis
            // do produto m_nCaixasPIC e m_rCaixasPIC, o que permite ao veículo
            // saber a quantidade actual de caixas disponíveis para 'picking' nesse
            // produto, e assim resolver se deve ou não enviar ao SimMaster um pedido
            // de reabastecimento da zona de 'picking' desse produto. Isto acontece
            // somente se a célula pertencer a uma estante.
            ////////////////////////////////////////////////
            if(pCel->m_tipoSuporte == ESTANTE)
              {
               Produto* pProduto = pCel->m_produto;
               int existem = pProduto->m_nCaixasPIC;
               int reserva = pProduto->m_rCaixasPIC;
               int maximo  = pProduto->m_maxCaixasPIC;
               float ratio = pProduto->m_percentPicking;

               if(existem - reserva <= ratio*maximo)
                 m_pSim->SimMaster(m_time,PRD_FILL_PICKING,pProduto);
              }
           }

        ////////////////////////////////////////////////////////////
        // Conhecendo o tempo de Carga, dado pela variável m_timeCarregar, o veículo
        // marca para o fim desta operação um evento VEIC_START para assinalar que está pronto
        // para continuar. Note-se que depois de uma operação de LOAD o veículo terá sempre
        // mais alguma coisa que fazer, pois só transferirá o resto da tarefa ao SimMaster no final
        // de executar uma acção de UNLOAD.
        ////////////////////////////////////////////////////////////

        m_pSim->Schedule(m_time+m_timeCarregar, VEIC_START, this);


        return TRUE;
    }
```

7.3.1.5 *Evento VEIC_DESCARGA*

A estrutura do método de resposta a este evento é muito semelhante à do método anterior. A diferenças residem na operação de transferência que é evocada e no último evento que é marcado. Em vez de uma operação de carga, é executada uma de descarga. O material da palete do veículo é transferido para a célula de destino.

Mais uma vez, da interpretação dos campos da actual subtarefa é decidido que tipo de unidades se devem transferir: caixas ou palete. No primeiro caso, são retiradas da palete do veículo o número de caixas requerido e adicionadas às

que já existem na célula de destino. No caso de se tratar de uma palete, toda a informação da palete do veículo é transferida para uma nova palete criada e esta adicionada à célula em causa. Neste caso, a palete do veículo é depois sinalizada como VAZIA.
Depois de terminada a operação de descarga é marcado na simulação um evento VEIC_FREE para que seja avisado o *SimMaster* do fim do cumprimento daquela parte da tarefa.

```
BOOL Transp::Descarga()
{
    m_time = m_pSim->m_time;   //tempo do veículo é o tempo actual do Simulador

    ////////////////////////////////////////////////////////////
    //  Se a SubTarefa actual for do tipo PALETE, então cria uma palete nova, passa o conteúdo
    //  da do veículo para ela, e coloca-a depois na célula. A palete do veículo é mantida VAZIA.
    //  Se a SubTarefa actual for do tipo CAIXAS, então retira esse número de caixas da palete do
    //  veículo e coloca-as na célula de destino. Se a palete do veículo ficar sem quaisquer caixas
    //  é alterado o seu estado para palete VAZIA:
    ////////////////////////////////////////////////////////////

    Palete* pPal;
    Celula* pCel = m_subTarefa->m_celula;

    if(m_subTarefa->m_tipoUnidades == PALETE) //PALETE
       {
         float escala = m_palete.m_escala;
         float largura = m_palete.m_lagura;
         float profund = m_palete.m_profund;

         pPal = new Palete(escala,largura,profund);

         pPal->m_largura = m_palete.m_largura;
         pPal->m_profund = m_palete.m_profund;
         pPal->m_END = FALSE;
         pPal->m_tipoPal = m_palete.m_tipoPal;
         pPal->m_maxCaixas = m_palete.m_maxCaixas;
         pPal->m_nCaixas = m_palete.m_nCaixas;
         pPal->m_ref = m_palete.m_ref;
         pPal->m_produto = m_palete.m_produto;

         pCel->PutPalete(pPal);
         m_palete.m_tipoPal = VAZIA;
        }
    else //CAIXAS
       {
         UINT nCaixas = m_subTarefa->m_nUnidades;
         pCel->PutCaixas(nCaixas);
         m_palete.m_nCaixas -= nCaixas;
         if(m_palete.m_nCaixas == 0)
              m_palete.m_tipoPal = VAZIA;
       }

    ////////////////////////////////////////////////////////////
    //  Conhecendo o tempo de Descarga, dado pela variável m_timeDescarregar, o
    //  veículo marca para o fim desta operação um evento VEIC_FREE através do qual é
    //  alertado o SimMaster para a finalização deste processo de transferência de material.
    ////////////////////////////////////////////////////////////

    m_pSim->Schedule(m_time+m_timeDescarregar, VEIC_FREE, this);


    return TRUE;
}
```

7.3.1.6 *Evento VEIC_FREE*

Neste evento o veículo transfere a responsabilidade da acção para o *SimMaster*, deixando-o decidir seguindo os critérios aí implementados.

```
BOOL Transp::Free()
{
    m_time = m_pSim->m_time;   //tempo do veículo é o tempo actual do Simulador

    ////////////////////////////////////////////////////////////////
    //  Neste evento o veículo passa a responsabilidade da decisão para o SimMaster, podendo
    //  este analisar a última subtarefa executada e o tipo de acção que acabou de ser realizada
    //  pelo veículo. Estando FREE, o veículo poderá ser escolhido para uma nova tarefa.
    ////////////////////////////////////////////////////////////////

    m_pSim->SimMaster(m_time, FIM_TAREFA, this);


    return TRUE;
}
```

7.3.2 Eventos e métodos dos Tapetes

Com o objectivo de modelar o funcionamento do tapete foram definidos na classe deste objecto os seguintes oito eventos:

- `TAP_START_JOB` (momento de iniciar a tarefa)
- `TAP_START` (momento de arranque ou rearranque)
- `TAP_END` (instante de chegada de palete ao fim do tapete)
- `TAP_DESCARGA` (instante em que a palete abandona o tapete)
- `TAP_STOP` (momento de paragem do tapete normal)
- `TAP_STOP_ACUMULA` (momento de paragem do tapete de acumulação)
- `TAP_END_ACUMULA` (instante de encosto de palete)
- `TAP_INVERT` (momento de inversão do movimento do tapete)

Como de costume, a cada evento foi associado o método que é executado assim que o respectivo evento surge no decurso da simulação, método que foi incluído na estrutura da classe *Tapete*. O conjunto desses métodos define o tapete como uma entidade activa na simulação com a seguinte estrutura:

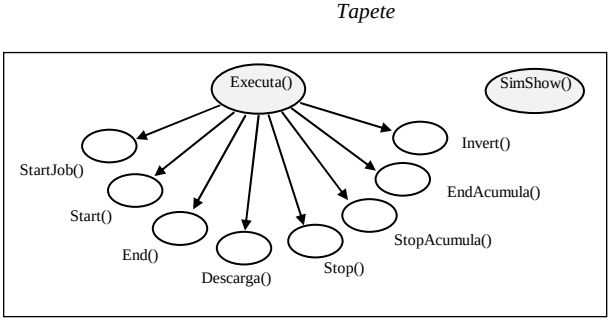

Fig. 7.10 Estrutura da entidade de simulação *Tapete*.

Repare-se que, para além dos métodos de resposta a eventos e do *Executa()*, o *Tapete* usa também o método *SimShow()*, herdado da classe *SimEntity*, uma vez que se trata de uma entidade activa que exige representação gráfica.
Os processos envolvidos nos tapetes, principalmente no que se refere à transferência de paletes entre uns e outros, merecem um pouco mais de atenção. Debrucemo-nos sobre o caso representado na próxima figura com o objectivo de esclarecer os mecanismos envolvidos nesse processo de transferência de material.

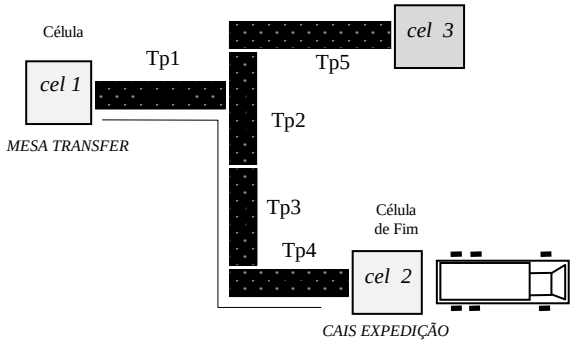

Fig. 7.11 Exemplo de um processo de transferência de material usando tapetes.

A tarefa a ser executada consiste em fazer transportar uma palete desde a célula `cel1` até à célula `cel2`. Como se torna evidente através da observação da figura, esta rede de tapetes permite não só transferir material segundo esse percurso, como também de `cel1` para `cel3`, conforme previamente for definido o sentido do movimento no tapete `Tp2`.
Por ser o "gestor" o responsável pela atribuição da tarefa ao primeiro tapete do percurso, e porque para atribuir essa tarefa ele necessita de calcular o percurso entre a célula inicial e a célula de fim, deixou-se também à responsabilidade do *SimMaster* a definição dos sentidos de movimentação dos tapetes que fazem parte desse percurso. Por isso, quando se trata de atribuir uma tarefa a um tapete, o *SimMaster* não só lhe passa um objecto do tipo *Tarefa* como também um 'Array' de inteiros com a sequência de sentidos dos tapetes envolvidos nessa tarefa. Como se pode verificar inspeccionando a estrutura da classe *Tapete*, existe nela um 'array' reservado a esse efeito: o `m_dirArray`.
Partamos então do princípio de que o *SimMaster* atribuiu ao tapete `Tp1` a tarefa de movimentar uma palete de `cel1` para `cel2`, passando-lhe também o respectivo 'array' de sentidos, e posteriormente lhe dirigiu o evento TAP_START_JOB para que fosse iniciada essa tarefa. Repare-se que serão considerados os seguintes três estados de um tapete:

- STOPPED (tapete parado e disponível para receber palete)
- MOVING (tapete executando movimentação de palete)
- WAITING (tapete esperando descarregar uma palete)

e só é possível atribuir uma tarefa a um tapete se ele se encontrar no estado STOPPED.

Todo o processo começa pela execução do método *StartJob()*, responsável por retirar da célula de início uma palete depois de definida a primeira subtarefa e de estabelecido o sentido do movimento do tapete. Este método chama depois o método *Start()* que irá calcular o instante em que a palete se encontrará no fim do tapete `Tp1`, marcando para esse instante um evento TAP_END.
O método *End()*, de resposta a este evento, encarrega-se de alertar o *SimControl* para o facto de a palete se encontrar no fim do tapete, esperando que este actue no sentido de resolver a situação. É o *SimControl* que verifica as condições de transferência da palete para o tapete `Tp2`. Se este tapete se encontrar parado (STOPPED), o que quer dizer que pode receber a palete, calcula o tempo que demorará a descarga e marca para esse instante um evento TAP_DESCARGA dirigido ao tapete `Tp1`. Se o tapete `Tp2` não puder receber a palete então é colocado o tapete `Tp1` numa lista de espera por descarga, lista essa que será inspeccionada quando `Tp2` retornar ao estado STOPPED.
Chegado o instante da descarga, o tapete `Tp1` executa o método *Descarga()* que se encarrega de transferir a palete para o tapete `Tp2` assim como de lhe passar a tarefa em execução e o respectivo 'array' de sentidos. O tapete `Tp1` pára em seguida.
Ao novo tapete (`Tp2`) é dada uma ordem de *Start()*, para que este possa reavaliar a situação, e de novo é despoletado um processo semelhante ao aqui descrito até que a palete seja transferida para o último tapete. Neste caso, quando a palete é colocada na célula final, o "gestor" é avisado desse facto através do envio ao método *SimMaster()* do evento FIM_TAREFA.
Apresentado este exemplo passemos a descrever com mais pormenor os métodos associados aos eventos dos tapetes:

7.3.2.1 *Evento TAP_START_JOB*

Este evento marca o início da execução de uma tarefa atribuída ao tapete. É enviado pelo *SimMaster* depois de este calcular a tarefa e o respectivo 'array' de sentidos dos tapetes envolvidos no percurso do material. O código do respectivo método é o seguinte:

```
BOOL Tapete::StartJob()
{
  BOOL ok;
  m_time = m_pSim->m_time; //Tempo do objecto é o tempo actual do Simulador

 ////////////////////////////////////////////////////////////
 //  Se existir uma tarefa atribuída ao tapete pelo SimMaster, então carrega a palete da célula
 //  correspondente indicada na subtarefa actual. Antes, porém, estabelece o sentido correcto
 //  do movimento do tapete impondo-o igual ao definido pelo SimMaster em m_dirArray.
 ////////////////////////////////////////////////////////////

 if(!m_tarefa->IsEmpty() && m_estado == STOPPED)
    {
     ok = TRUE;
     if(m_dir != m_dirArray[0]) Invert(); //altera m_dir
     m_dirArray.RemoveAt(0); //remove o primeiro elemento do array

     m_subTarefa = m_tarefa->RemoveHead();
```

```
      //retira a palete da célula e coloca-a na lista de paletes do tapete:
      LoadPalete(m_subTarefa->m_celula);
     }
 else ok = FALSE;

 if(ok == TRUE)
   {
    ////////////////////////////////////////////////////////////
    // Se não existirem paletes no tapete  manda o tapete parar. Isto é feito chamando o método
    // Stop() que coloca o tapete em STOPPED e avisa os outros tapetes que para ele executam
    // a descarga que já o podem usar. Se existirem inicia o processo com Start().
    ////////////////////////////////////////////////////////////

    m_nPaletes = m_paleteList.GetCount();
    if(!m_nPaletes) { Stop(); ok = FALSE; }
    else  this->Start();
   }

 return ok;
}
```

7.3.2.2 *Evento TAP_START*

Este evento coloca o tapete em movimento e é responsável por calcular o instante de tempo em que a palete atingirá a posição final nesse tapete. Para esse instante é marcado um evento TAP_END.

```
BOOL Tapete::Start()
{
 float dt;
 float ds;
 float dw;
 POSITION pos;

 m_time = m_pSim->m_time; //Tempo do objecto é o tempo actual do Simulador


 ////////////////////////////////////////////////////////////
 // Existem paletes no tapete, por isso coloca o seu estado em MOVING e o tapete na lista
 // de objectos em movimento na simulação. Esta lista é usada pela simulação para guardar
 // os apontadores dos objectos que deverão ser movimentados no ecrã. Um outro evento
 // da simulação (SIM_SHOW) é responsável pelo seu varrimento, mandando actualizando
 // depois as posições dos respectivos objectos:
 ////////////////////////////////////////////////////////////

 m_estado = MOVING; //Tapete passa a estar em movimento

 pos = m_pSim->m_objMovingList.Find(this);
 if(pos == NULL) m_pSim->m_objMovingList.AddTail(this);

 ////////////////////////////////////////////////////////////
 // PRÓXIMO EVENTO:
 // Se existem paletes neste tapete, calcula o instante em que a palete da frente da sua lista de
 // paletes se encontrará no fim do tapete. Marca para esse instante um evento de TAP_END
 // para assinalar que uma palete estará nesse momento no fim do tapete. Antes, porém,
 // retira de todas as paletes da sua lista de paletes a marca de END (marca que indica que
 // aquela palete é considerada na posição de FIM)
 ////////////////////////////////////////////////////////////

 Palete* palete;
 pos = m_paleteList.GetHeadPosition();
 while(pos!=NULL)
```

```
    {
     palete = m_paleteList.GetNext(pos);
     palete->m_END = FALSE;
    }

 palete = m_paleteList.GetHead(); //palete da frente
 ds = m_larg/2.0f;

 if(m_orient == AREA_HORIZONTAL)
    dw = (float)fabs(m_escala*m_pFim.x - palete->m_x);

 else if(m_orient == AREA_VERTICAL)
    dw = (float)fabs(m_escala*m_pFim.y - palete->m_y);

 dt = (dw-ds)/m_velCarg;

 m_pSim->Schedule(m_time+dt, TAP_END, this); //marca o evento END.


 return TRUE;
}
```

7.3.2.3 *Evento TAP_END*

Neste instante a palete encontra-se no fim do tapete. Assim, este evento passa ao *SimControl* a responsabilidade de decidir sobre o que fazer a seguir, uma vez que é o *SimControl* que possui os critérios de decisão que se referem à transferência de material entre tapetes. Como resultado, poderão surgir duas situações: ou o *SimControl* manda o tapete executar a descarga da palete, ou o coloca à espera dessa descarga.

```
BOOL Tapete::End()
{
 CPoint pt;
 float escala;
 float ds;
 float escalaView = m_pSim->m_view->m_escala;

 m_time = m_pSim->m_time; //Tempo do tapete é o tempo actual do Simulador

 ////////////////////////////////////////////////////////////
 // Desenha na posição final do tapete a primeira palete da lista de paletes. É também
 // actualizada a posição dessa palete relativamente às coordenadas do armazém.
 ////////////////////////////////////////////////////////////

 Palete* palete = m_paleteList.GetHead();

 escala = palete->m_escala;

 ds = m_dir*m_larg/2.0f;

 palete->m_x = m_escala*m_pFim.x;
 palete->m_y = m_escala*m_pFim.y;

 if(m_moving == MOVING_RIGHT)        palete->m_x -= ds;
 else if(m_moving == MOVING_LEFT)    palete->m_x += ds;
 else if(m_moving == MOVING_UP)      palete->m_y += ds;
 else if(m_moving == MOVING_DOWN)    palete->m_y -= ds;

 pt.x = (int)(0.5 + palete->m_x/escalaView);
 pt.y = (int)(0.5 + palete->m_y/escalaView);
```

```
palete->MoveTo( m_pSim->m_view, pt );
palete->m_END = TRUE;//esta palete está no fim do tapete

////////////////////////////////////////////////////////
// Retira o tapete na lista de objectos à espera de executar a operação de descarga de
// material, se for caso disso.
////////////////////////////////////////////////////////
 POSITION pos = m_pSim->m_objWaitList.Find(this);
 if(pos!=NULL) m_pSim->m_objWaitList.RemoveAt(pos);

////////////////////////////////////////////////////////////////
// CRITÉRIO DE DESCARGA:
// Neste momento há que decidir sobre o destino da palete que se encontra no fim do tapete.
// Note-se que a responsabilidade de decidir sobre este assunto é, segundo a estrutura do
// Simulador que se propõe neste trabalho, do SimControl(). Por isso mesmo, o tapete deverá
// enviar a este método o evento TAP_END de forma a que ele possa decidir que acções
// deverão ser realizadas a seguir. Portanto, este critério é implementado no SimControl() e
// não directamente no tapete.

m_pSim->SimControl(m_time, TAP_END, this);

// Apresenta-se, no entanto, também aqui esse código para que o leitor possa, com mais
// facilidade, seguir este processo na globalidade (código em itálico):
// Se existe entidade para onde este tapete possa descarregar e essa entidade estiver pronta
// para receber a palete, então manda efectuar a Descarga() , caso contrário coloca este
// em estado WAITING de espera por descarga. É também retirado da lista de objectos em
// movimento na simulação.
// Note-se que a descarga se poderá fazer para dois tipos de entidade: para um outro tapete
// ou para uma mesa. Para se poder descarregar para outro tapete basta que este se
// encontre no estado STOPPED, isto é, parado e sem paletes. Para o caso de uma mesa
// considera-se que é sempre possível executar a descarga, enviando a palete para a lista
// de paletes da mesa.
// Se este tapete não puder executar a descarga, é colocado numa lista de objectos à espera de
// descarga onde aguardará por uma futura ordem.
////////////////////////////////////////////////////////////////

float dt = m_tDescarga;

if(m_pEntDescarg != NULL)
   {
    //Caso de descarga para TAPETE:
    if(m_pEntDescarg->m_tipo == AREA_TAPETE)
      {
          if(m_pEntDescarg->m_estado == STOPPED) //descarrega
                 m_pSim->Schedule(m_time+dt,TAP_DESCARGA,this);

          else //Espera
              {
                 m_estado = WAITING;
                 m_pSim->m_objWaitList.AddTail(this);
                  pos = m_pSim->m_objMovingList.Find(this);
                 if(pos!=NULL)m_pSim->m_objMovingList.RemoveAt(pos);
              }
       }

      //Caso de descarga para MESA:
      else  m_pSim->Schedule(m_time+dt,TAP_DESCARGA,this);

    }

  //não existe entidade para descarga: descarrega para o chão.
 else m_pSim->Schedule(m_time+dt,TAP_DESCARGA,this);

 return TRUE;
}
```

7.3.2.4 *Evento TAP_DESCARGA*

Este evento representa o instante em que a palete acabou de ser descarregada para a entidade seguinte. Se esta entidade for um novo tapete, então essa palete é colocada na sua lista de paletes e para ele transferida a tarefa e o 'array' de sentidos respectivo. Se, por outro lado, essa entidade corresponder a uma mesa, presume-se tratar-se da mesa que contém a célula de fim e para aí é transferida a palete, avisando depois o *SimMaster* de que aquela parte da tarefa se encontra finalizada. Em qualquer dos casos a palete abandona o tapete actual passando este a executar o método *Stop()* que se encarrega de o parar e tornar disponível para receber outras paletes.

```
BOOL Tapete::Descarga()
{
 CPoint pt;
 float ds;
 float escalaView = m_pSim->m_view->m_escala;

 m_time = m_pSim->m_time; //Tempo do tapete é o tempo actual do Simulador

 ////////////////////////////////////////////////////////////
 // Retira a palete da frente da lista de paletes do tapete e calcula as coordenadas do ponto
 // para onde vai ser movimentada.
 ////////////////////////////////////////////////////////////

 Palete* palete = m_paleteList.RemoveHead();

 ds = m_dir*(m_velCarg*m_tDescarga + m_larg/2.0f);

 palete->m_x = m_escala*m_pFim.x;
 palete->m_y = m_escala*m_pFim.y;

 if(m_moving == MOVING_RIGHT)        palete->m_x += ds;
 else if(m_moving == MOVING_LEFT)    palete->m_x -= ds;
 else if(m_moving == MOVING_UP)      palete->m_y -= ds;
 else if(m_moving == MOVING_DOWN)    palete->m_y += ds;

 pt.x = (int)(0.5 + palete->m_x/escalaView);
 pt.y = (int)(0.5 + palete->m_y/escalaView);
 palete->MoveTo( m_pSim->m_view, pt );
 palete->m_END = FALSE;


 ////////////////////////////////////////////////////////////
 // TRANSFERÊNCIA DA PALETE:
 // Este evento foi marcado pelo Descarga() podendo ou não existir uma entidade válida
 // para onde transferir a palete. Se essa entidade for válida, (diferente de NULL), então
 // adiciona esta palete à lista de paletes dessa entidade. Por norma, esta palete é adicionada à
 // cauda da lista de paletes dessa entidade. A tarefa é também passada à próxima entidade se
 // ela for um tapete. Se a entidade para onde descarregar não existir, então a palete é
 // descarregada para o chão.
 // Por outro lado, no final da descarga é sempre executado o método Stop() deste tapete para
 // que sejam avisados aqueles que o utilizam como entidade de descarga de que ele passou a
 // estar livre.
 ////////////////////////////////////////////////////////////

 if(m_pEntDescarg->m_tipo == AREA_TAPETE) //entidade de descarrega é um tapete
    {
     Tapete* pTap = (Tapete*)m_pEntDescarg;
     pTap->m_paleteList.AddTail(palete);
```

```
      pTap->m_tarefa = m_tarefa;
      pTap->m_subTarefa = m_tarefa->GetHead();

      for(int i=0; i<m_dirArray.GetSize(); i++)
            pTap->m_dirArray.Add(m_dirArray[i]);

      if(pTap->m_dir != m_dirArray[0]) pTap->Invert(); //nova direcção
      pTap->m_dirArray.RemoveAt(0); //remove o primeiro elemento do array

      pTap->Start();
     }//se o destino for um tapete...

  else if(m_pEntDescarg->m_tipo == AREA_MESA) //entidade descarrega é uma mesa
     {
      Mesa* pMesa = (Mesa*)m_pEntDescarg;
      pMesa->m_celula.m_paleteList.AddTail(palete);
      m_pSim->SimMaster(m_time, FIM_TAREFA, this); //Avisa SimMaster
     }//se o destino for uma mesa...

  this->Stop(); //tapete livre para aceitar paletes.

  return TRUE;
}
```

7.3.2.5 *Evento TAP_STOP*

Este evento é gerado depois de o tapete ter executado uma operação de descarga ou quando se pretende colocá-lo disponível para receber novas paletes. O método de resposta a este evento foi designado por *Stop()*.
Se o tapete for de acumulação, este evento automaticamente se relaciona com o evento TAP_STOP_ACUMULA responsável pelo processo de paragem desse tipo de tapetes. Por isso, tratando-se de um tapete de acumulação, o método *Stop()* delega a responsabilidade ao método *StopAcumula()*, o qual será descrito a seguir.
Tratando-se de um tapete normal, o método *Stop()* encarrega-se de colocar em STOPPED o estado do tapete, de o retirar da lista de objectos em movimento na simulação e de enviar aos tapetes que para ele esperam descarregar material a respectiva ordem de descarga.

```
BOOL Tapete::Stop()
{
    ////////////////////////////////////////////////////////
    //  Se o tapete for de acumulação, transfere a responsabilidade para o método
    //  StopAcumula() .
    ////////////////////////////////////////////////////////

    if(m_ACUMULA) return this->StopAcumula();

    ////////////////////////////////////////////////////////
    //  Tratando-se de um tapete normal, coloca o tempo deste tapete igual ao tempo
    //  actual da simulação e o seu estado em STOPPED.
    ////////////////////////////////////////////////////////

    m_time = m_pSim->m_time;
    m_estado = STOPPED;

    ////////////////////////////////////////////////////////
```

```
    // Retira depois o tapete da lista de entidades que se encontram em movimento na
    // simulação. Isto permite que a simulação deixe de se preocupar com a actualização
    // de um objecto que está parado.
    ////////////////////////////////////////////////////////

    POSITION pos = m_pSim->m_objMovingList.Find(this);
    if(pos!=NULL) m_pSim->m_objMovingList.RemoveAt(pos);

    ////////////////////////////////////////////////////////
    // Retira o tapete na lista de objectos à espera de executar a operação de descarga de
    // material, se for caso disso. Nesta lista são guardadas as entidades que estão à
    // espera de descarregar. Assume-se assim que se foi dada uma ordem de STOP
    // ao tapete, é porque ele já executou a sua descarga.
    ////////////////////////////////////////////////////////

    pos = m_pSim->m_objWaitList.Find(this);
    if(pos!=NULL) m_pSim->m_objWaitList.RemoveAt(pos);

    ////////////////////////////////////////////////////////
    // Envia uma ordem de reavaliar a descarga aos possíveis tapetes que esperam
    // descarregar material para este tapete, pois ele já se encontra parado e sem paletes.
    // Isto é feito enviando a esses tapetes um evento TAP_END.
    ////////////////////////////////////////////////////////

    Tapete* pTap;
    float dt;
    pos = m_pSim->m_objWaitList.GetHeadPosition();
    while(pos!=NULL)
          {
           pTap = (Tapete*) m_pSim->m_objWaitList.GetNext(pos);
           if(pTap->m_pEntDescarg == this)
                {
                 dt = pTap->m_tDescarga;
                 m_pSim->Schedule(m_time, TAP_END, pTap);
                }
          }

 return TRUE;
}
```

Se o tapete for de acumulação, o evento anterior é desviado para o método *StopAcumula()*, o qual inicia o processo de acumulação calculando o instante de encosto da primeira palete da lista de paletes que ainda não está encostada e marcando para esse instante um evento TAP_END_ACUMULA. Este evento, por sua vez, volta a chamar o método *StopAcumula()*, mantendo-se este processo até que todas as paletes se encontrem encostadas.

```
BOOL Tapete::StopAcumula()
{
    float dt;
    float ds;
    float dw;

    ////////////////////////////////////////////////////////
    // Trata-se da paragem de um tapete de acumulação. Coloca o tempo deste tapete
    // igual ao tempo actual da simulação
    ////////////////////////////////////////////////////////

    m_time = m_pSim->m_time;

    ////////////////////////////////////////////////////////
    // Verifica se existe alguma palete no tapete que ainda não esteja encostada
    // (palete->m_END = FALSE) e, nesse caso, toma essa palete como a próxima a
```

```
// ser encostada.
////////////////////////////////////////////////////////////

POSITION pos = m_paleteList.GetHeadPosition();
Palete* palete;
m_endPalete = NULL;
int n = 0;
while( pos != NULL ) //existem paletes no tapete...
      {
       palete = m_paleteList.GetNext(pos); //palete a mover
       if(!palete->m_END) //ainda para mover...
            {
             m_endPalete = palete;
             break;
            }
       n++;
      }

////////////////////////////////////////////////////////////
// Existindo ainda uma palete para ser movimentada até ao encosto, calcula o espaço a
// ser percorrido por essa palete  até estar encostada e o tempo da simulação em que
// esse acontecimento se irá dar, marcando para esse instante o evento:
// TAP_END_ACUMULA.
////////////////////////////////////////////////////////////

if(m_endPalete!=NULL) //alguma palete para ser movimentada...
      {
       ds = (n + 0.5f)*m_larg;

       if(m_orient == AREA_HORIZONTAL)
       dw = (float)fabs(m_escala*m_sf.x - m_endPalete->m_x);
       else if(m_orient == AREA_VERTICAL)
        dw = (float)fabs(m_escala*m_sf.y - m_endPalete->m_y);

       dt = (dw-ds)/m_velCarg;

       m_pSim->Schedule(m_time+dt, TAP_END_ACUMULA, this);
      }


////////////////////////////////////////////////////////////
// Se já não existem paletes para mover, então pára o tapete retirando-o da lista de
// objectos em movimento e impondo o seu estado em STOPPED tornando-o acessível
// à outros que nele queiram descarregar material.
////////////////////////////////////////////////////////////

else
    {
      m_estado = STOPPED; // pára o tapete…
      pos = m_pSim->m_objMovingList.Find(this);
      if(pos!=NULL) m_pSim->m_objMovingList.RemoveAt(pos);

       ////////////////////////////////////////////////////
       // Retira o tapete na lista de objectos à espera de executar a operação de
       // descarga , se for caso disso. Nesta lista são guardadas as entidades  que
       // estão à espera de descarregar.
       ////////////////////////////////////////////////////

       pos = m_pSim->m_objWaitList.Find(this);
       if(pos!=NULL) m_pSim->m_objWaitList.RemoveAt(pos);

       ////////////////////////////////////////////////////
       // Envia uma ordem de reavaliação da descarga aos possíveis tapetes que
       // esperam descarregar para este tapete, pois ele já se encontra parado.
       ////////////////////////////////////////////////////

       Tapete* pTap;
       pos = m_pSim->m_objWaitList.GetHeadPosition();
```

```
            while(pos!=NULL)
            {
             pTap = (Tapete*) m_pSim->m_objWaitList.GetNext(pos);
             if(pTap->m_pEntDescarg == this)
               {
                 dt = pTap->m_tDescarga;
                m_pSim->Schedule(m_time, TAP_END, pTap);
               }
            }
         }

 return TRUE;
}
```

Em seguida apresenta-se o método *EndAcumula()* encarregado de encostar as paletes no tapete de acumulação. Realizada essa operação, este método volta a chamar o método *StopAcumula()* para que de novo a situação possa ser avaliada.

```
BOOL Tapete::EndAcumula()
{
    CPoint pt;
    float escala;
    float ds;
    float escalaView = m_pSim->m_view->m_escala;


    m_time = m_pSim->m_time; // Tempo do objecto é o tempo actual da simulação.

    ////////////////////////////////////////////////////////////
    // A próxima palete a ser encostada é a m_endPalete, e será encostada à n ésima
    // palete parada no tapete.
    ////////////////////////////////////////////////////////////

    POSITION pos = m_paleteList.GetHeadPosition();
    int n = 0;
    while(pos != NULL)
           {
            palete = m_paleteList.GetNext(pos);
            if(palete==m_endPalete) break;
            n++;
           }

    ////////////////////////////////////////////////////////////
    // Existindo ainda uma palete para ser movimentada até ao  encosto, calcula o espaço a
    // ser percorrido por essa palete até estar encostada e movimenta-a para o ponto de
    // encosto.
    ////////////////////////////////////////////////////////////

    if(m_endPalete!=NULL) //alguma palete...
           {
           escala = m_endPalete->m_escala;

           ds = (n + 0.5f)*m_larg;

           m_endPalete->m_x = m_escala*m_sf.x;
           m_endPalete->m_y = m_escala*m_sf.y;

           //coloca a palete encostada à última...
           if(m_moving == MOVING_RIGHT)     m_endPalete->m_x -= ds;
           else if(m_moving == MOVING_LEFT) m_endPalete->m_x += ds;
           else if(m_moving == MOVING_UP)   m_endPalete->m_y += ds;
           else if(m_moving == MOVING_DOWN) m_endPalete->m_y -= ds;

           pt.x = (int)(0.5 + m_endPalete->m_x/escalaView);
```

```
            pt.y = (int)(0.5 + m_endPalete->m_y/escalaView);
            palete->MoveTo( m_pSim->m_view, pt );
            m_endPalete->m_END = TRUE;
            }

 ////////////////////////////////////////////////
 //  Evoca de novo o método StopAcumula() para decidir se deve continuar o
 //  processo de acumulação ou se pode parar o tapete.
 ////////////////////////////////////////////////

  this->StopAcumula();

  return TRUE;
}
```

7.3.2.6 *Evento TAP_INVERT*

Passemos, por fim, ao último evento relacionado com a modelação dos tapetes. Ao contrário dos restantes eventos aqui descritos, o TAP_INVERT somente actua nos atributos do objecto, não sendo responsável pela marcação de outros eventos ou por chamadas a outros métodos reservados a eventos. De qualquer modo, as poucas alterações que aqui se promovem reflectir-se-ão na actividade dos restantes eventos, pois permitirão uma completa inversão do sentido de movimentação das paletes no tapete em questão assim como a mudança de entidade para onde se realizarão as descargas.

```
BOOL Tapete::Invert()
{

    ////////////////////////////////////////////////
    //  Inverte o movimento do tapete, alterando também  o seu ponto de fim
    //  e a entidade para onde se  efectuarão as descargas.
    ////////////////////////////////////////////////

    m_dir = - m_dir;

    if(m_dir == 1){m_pFim = m_sf; m_pEntDescarg = m_pEntity2;}
              else{m_pFim = m_si; m_pEntDescarg = m_pEntity1;}

    ////////////////////////////////////////////////
    //  Inverte também a lista de paletes, pois agora a palete que se encontrava
    //  em último lugar deverá ser considerada a primeira da lista. É para esta
    //  palete que irá ser executado o próximo evento TAP_END.
    ////////////////////////////////////////////////

    CTypedPtrList<CObList, Palete*>  newList;

    POSITION pos;
    Palete* palete;
    pos = m_paleteList.GetHeadPosition();
    while(pos!=NULL)
           {
           palete = m_paleteList.GetNext(pos);
           palete->m_END = FALSE;
           newList.AddTail(palete);
           }
    m_paleteList.RemoveAll();

    pos = newList.GetHeadPosition();
    while(pos!=NULL)
           {
```

```
            palete = newList.GetNext(pos);
            m_paleteList.AddHead(palete);
            }
    newList.RemoveAll();

 return TRUE;
}
```

7.3.3 Eventos e métodos das Mesas e IPs

Nesta versão do programa de simulação, as mesas e os pontos de identificação (*IPs*) não são tratados como verdadeiras entidades activas, ou seja, não respondem a eventos. Relembre-se que um *IP* foi considerado um caso específico de mesa, e que quaisquer destes elementos possuem associado um tempo de resposta para confirmar a sua mudança de estado. No caso de uma mesa de transferência ortogonal esse tempo corresponde ao tempo que medeia entre o momento em que uma palete nela é colocada e o instante em que essa palete fica acessível ao próximo elemento móvel. Na mesa rotativa acontece o mesmo, só que esse tempo engloba o tempo despendido na rotação da palete. Tratando-se de um *IP*, esse tempo corresponde ao tempo durante o qual decorre o processo de identificação.
Neste caso, para simplificar a actividade da simulação, fez-se depender do *SimMaster* o controle desses tempos, uma vez que a ele retorna a tarefa acabada de executar sempre que a palete é depositada num destes elementos. Sendo assim, quando o *SimMaster* recebe de volta essa tarefa começa por localizar a célula a que se refere a última operação de UNLOAD. Se essa célula pertencer a uma mesa (ou *IP*), então inspecciona o seu tempo de actividade, usando a variável da mesa designada por `m_actionTime`, e só dirige o próximo evento de início de tarefa (VEIC_START para um veículo ou TAP_START_JOB para um tapete) depois de decorrido esse tempo. É um caso desses que se exemplifica de seguida com a ajuda da próxima figura.

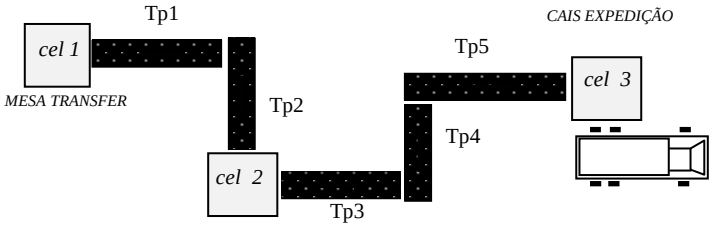

Fig. 7.12 Transferência de material entre várias células usando tapetes.

O *SimMaster* principia por atribuir ao tapete `Tp1` uma tarefa que consiste em carregar uma palete da célula `cel1`, e descarregá-la em `cel2`, voltar a carregá-la desta célula e finalmente descarregá-la em `cel3`. O tapete `Tp1` carrega a palete de `cel1` e transporta-a até ao tapete `Tp2` para o qual a transfere, assim como a tarefa corrente. Por sua vez, o tapete `Tp2` transporta-a até à célula `cel2` e aí a

deposita, devolvendo ao *SimMaster* a tarefa em execução dado que executou uma operação de UNLOAD. O *SimMaster* verifica que foi executada essa operação de descarga na célula `cel2` e "lê" o tempo que demora esse processo através da variável `m_actionTime` da mesa que contém essa célula. De novo transfere o resto da tarefa para o tapete `Tp3` depois de decorrido esse tempo. Assim, através deste mecanismo simula-se o tempo de espera que o operador associou a cada um dos elementos do tipo *Mesa* sem que seja necessário considerá-los elementos activos na simulação e, portanto, atribuir-lhes quaisquer eventos.

7.3.4 Eventos e métodos das *OutOrders*

A uma *OutOrder*, ou *Ordem de Saída* de material, associaram-se dois eventos. Um que representa o instante em que essa ordem dá entrada no sistema, e outro que assinala o fim do seu processamento. A *OutOrder* considerar-se-á processada, e, portanto, finalizada a sua execução, quando estiverem satisfeitos todos os *Pedidos de Material* que a constituem. São os seguintes os eventos definidos para a *OutOrder*:

- `OUT_ORD_ARRIVE` (momento de chegada da *OutOrder*)
- `OUT_ORD_END` (momento em que a *OutOrder* é satisfeita)

De novo, a cada um destes eventos foi associado um método específico deste objecto, transformando a *OutOrder* numa entidade activa na simulação com a seguinte estrutura:

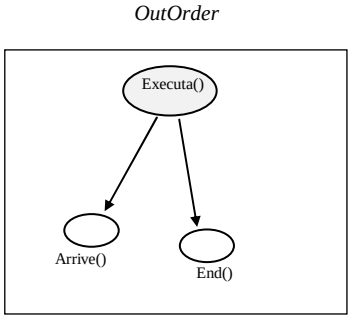

Fig. 7.13 Estrutura da entidade de simulação *OutOrder*.

Repare-se que, ao contrário do que acontecia com os tapetes e os veículo, este elemento não carece de representação gráfica para simular movimento, facto pelo qual ignora o método *SimShow()* herdado da classe *SimEntity*.
De facto, para melhor informar visualmente o utilizador de que decorre um processo de saída de material, convém representar um camião junto ao cais a que a *OutOrder* for associada, mas isso poderá ser feito através de uma função específica para o efeito, chamada no início e no fim desse processamento. No início do processo é desenhando um camião junto ao cais, no fim, o camião é retirado do ecrã.

7.3.4.1 *Evento OUT_ORD_ARRIVE*

Com já foi referido, quando na simulação aparece uma *OutOrder* ela é directamente "desviada" para o *SimMaster* a fim de ser analisada e de, se for caso disso, se estabelecer um processo para satisfazer o seu conjunto de *pedidos de material*. Por esse motivo, e dado que os critérios de decisão ao nível de manuseamento de produtos se encontram incluídos no "gestor", o próprio objecto *OutOrder* automaticamente dirige o evento OUT_ORD_ARRIVE para o *SimMaster* da simulação através do seu método *Arrive()*. Antes, porém, é criada pelo *Simulador* a próxima *OutOrder*, tendo em conta critérios predefinidos no seu método *NewOutOrder()*, e marcada a sua chegada para um instante do futuro. Esse instante é calculado com base no tempo médio entre chegadas de clientes (`m_OutOrder_dTime`), parâmetro pertencente ao próprio *Simulador*. Ainda no método *Arrive()* é dado um número de ordem à *OutOrder* e estabelecido o seu tempo de chegada.

```
BOOL OutOrder::Arrive()
{

    m_time = m_pSim->m_time;   //Tempo do objecto é o tempo actual do Simulador
    m_timeArrive = m_time;     //Tempo de chegada da OutOrder

    //incrementa o número de ordem que representa as OutOrders chegadas ao armazém.
    m_number = m_pSim->m_nOutOrders++;

    ////////////////////////////////////////////////////
    //  Cria a próxima OutOrder a aparecer na simulação e marca-a para o tempo
    //  previsto da chegada do próximo cliente.
    ////////////////////////////////////////////////////

    float dTime = m_pSim->m_OutOrder_dTime;

    OutOrder* pNew = m_pSim->NewOutOrder(m_time);
    m_pSim->Schedule(m_time+dTime, OUT_ORD_ARRIVE, pNew);

    ////////////////////////////////////////////////////
    //  Envia este evento (OUT_ORD_ARRIVE) ao SimMaster para dar inicio
    //  a  todo o processo de atendimento do actual cliente.
    ////////////////////////////////////////////////////

    m_pSim->SimMaster(m_time, OUT_ORD_ARRIVE, this);

 return TRUE;
}
```

Como se pode verificar, a grande responsabilidade no processamento de uma *OutOrder* caberá ao *SimMaster*, o que levou a apresentar de seguida a parte deste método do *Simulador* que diz respeito ao aparecimento de uma ordem deste tipo. Note-se que este código não se refere ao *SimMaster* completo, mas unicamente à parte relacionada com a chegada da *OutOrder*:

```
void ArmSimulation::SimMaster(float time,UINT event,SimEntity* pEnt)
{

 /*****************************************************************
   Se a entidade que envia o evento é da classe OutOrder:
 *****************************************************************/

 else if(pEnt->m_tipo == OUT_ORDER)
   {
     OutOrder* pOut = (OutOrder*)pEnt;
     switch(event)
     {
      case OUT_ORD_ARRIVE:
            ////////////////////////////////////////////////
            // Critérios a seguir quando aparece uma OutOrder:
            // NOTA: estes critérios são provisórios e deverão ser discutidos com a
            // EFACEC de forma a aproximarem-se, o mais possível, dos usados por
            // essa empresa em instalações reais.
            ////////////////////////////////////////////////

            ////////////////////////////////////////////////
            // 1º: Percorrendo a Lista de Material da OutOrder, e tendo em conta
            // tanto os produtos como as quantidades requeridas, verifica se é
            // possível atender o cliente. Se não for possível rejeita a OutOrder.
            ////////////////////////////////////////////////

            if(!IsPossible(pOut)) Reject(pOut);

            ////////////////////////////////////////////////
            // 2º: Sendo possível atender o pedido verifica se é possível associar à
            // OutOrder um cais de expedição. Se não for possível, por os cais se
            // encontrarem ocupados, coloca a OutOrder na Lista de OutOrders à
            // espera de cais (m_OutOrderWaitList). Esta lista será de novo
            // inspeccionada quando um cais ficar livre, depois de passado ao
            // SimMaster um evento de FIM_TAREFA e se não existirem tarefas à
            // espera de serem executadas.
            ////////////////////////////////////////////////

            else
                {
                  if(!IsThereCaisTo(pOut))
                     m_OutOrderWaitList.AddTail(pOut);

                  ////////////////////////////////////////
                  // Se foi possível encontrar um cais livre, automaticamente
                  // esse cais foi associado à OutOrder e logo de seguida criada
                  // a primeira tarefa de forma a satisfazer o pedido de material
                  // em causa. A OutOrder é colocada na lista m_OutOrderRun
                  // do Simulador, e essa tarefa atribuída a um elemento que
                  // se encarrega do transporte do material. Se a tarefa não pode
                  // ainda ser executada é enviada para a lista de tarefas à espera
                  // de execução m_JobWaitingList. De qualquer forma, a
                  // tarefa encontra-se já associada à OutOrder.
                  ////////////////////////////////////////
                  else
                     {
                       Tarefa* pTarefa = CalcTarefa(pOut);
                       StartExecute(pTarefa);
                      }
                }
        break;

    case OUT_ORD_END:

            // apresentado na próxima secção...

        break;
```

```
            default:break;
           }
      }//Foi uma OutOrder
}
```

Repare-se que no final de cada passo da execução da tarefa, isto é, quando o *SimMaster* recebe um evento FIM_TAREFA de um elemento transportador, deverão ser executados por ordem os seguintes procedimentos:

1º Verificar se essa tarefa se encontra realmente terminada. Isso acontece quando não existirem mais *subtarefas* na tarefa a não ser a última de UNLOAD. Se não estiver terminada, a tarefa em questão é dirigida pelo *SimMaster* ao próximo elemento transportador ou colocada na lista de tarefas à espera de serem executadas. Se essa tarefa estiver terminada, então passa ao seguinte procedimento:

2º Verificar se essa tarefa pertence a alguma *OutOrder* da lista de OutOrders em execução. Se pertencer e se a *OutOrder* não estiver finalizada, calcula a próxima tarefa para essa *OutOrder* e manda-a executar se for possível. Se não for possível executá-la envia-a para a lista de espera das tarefas a serem executadas. Se aquela tarefa não pertence a nenhuma *OutOrder* da lista em execução, então passa ao próximo procedimento:

3º Verificar se essa tarefa pertence a alguma *InOrder* da lista de InOrders em execução. Se pertencer e se a *InOrder* não estiver finalizada, calcula a próxima tarefa para essa *InOrder* e manda-a executar se for possível. Se não for possível executá-la envia-a para a lista de espera das tarefas a serem executadas. Se aquela tarefa não pertence a nenhuma *InOrder* da lista em execução, então passa ao próximo procedimento:

4º Verificar se existe alguma nova tarefa para executar na lista de tarefas em espera . Essa acção é realizada verificando o tamanho da lista `m_JobWaitingList` do *Simulador*. Se existir alguma tarefa que possa ser executada por aquele elemento, então dá essa tarefa a esse elemento. Se não existirem tarefas na lista de espera ou se nenhuma for possível de atribuir àquele elemento, então passa ao seguinte procedimento:

5º Verificar se a tarefa terminada libertou algum cais. Essa acção é realizada verificando se o suporte da célula de UNLOAD é um cais. Se for um cais de expedição, percorre a lista de *OutOrders* à espera de cais e tenta atribuí-lo. Se foi atribuído, manda iniciar o processamento da *OutOrder*, retirando-a da lista de espera e colocando-a na lista `m_OutOrderRun`. Se a célula de suporte era um cais de recepção, executa as mesmas operações mas usando a lista de *InOrders* à espera de cais.

6º Como último procedimento a ser sempre executado: Se o elemento transportador era um veículo e se não lhe foi atribuída nova tarefa, dá-lhe ordem de recolher ao seu parque de estacionamento.

Estes procedimentos fazem parte dos critérios implementados no *SimMaster* para lidar com a situação de FIM_TAREFA e interferem também com o processamento das *OutOrders*, assim como com as *InOrders*, com as *tarefas* e com os próprios *elementos de transporte* de material.

7.3.4.2 *Evento OUT_ORD_END*

Este evento é despoletado pelo *SimMaster* quando verifica que uma *OutOrder* se encontra completamente satisfeita. O método da *OutOrder* responsável por este evento, que a seguir se apresenta, simplesmente preenche algumas variáveis dessa *OutOrder*, retira-a da lista de processamento e acaba por o chamar o *SimMaster*.

```
BOOL OutOrder::End()
{

    m_time = m_pSim->m_time;    //Tempo do objecto é o tempo actual do Simulador
    m_timeEnd = m_time;         //Tempo de finalizada a OutOrder
    ////////////////////////////////////////////////////
    //  Retira a OutOrder da lista de OutOrders em processamento, se lá existir,
    //  pois já se encontra finalizada.
    ////////////////////////////////////////////////////

    POSITION pos = m_pSim->m_OutOrderRun.Find(this);
    if(pos!=NULL) m_pSim->m_OutOrderRun.RemoveAt(pos);

    ////////////////////////////////////////////////////
    //  Envia este evento (OUT_ORD_END) ao SimMaster para o avisar que
    //  chegou o fim do  processo de atendimento do actual cliente.
    ////////////////////////////////////////////////////

    m_pSim->SimMaster(m_time, OUT_ORD_END, this);

 return TRUE;
}
```

Como este evento representa o instante em que o material abandona o armazém, aproveitou-se para nele incluir o cálculo das actuais existências em 'stock', o que, em certos casos, poderá despoletar *Ordens de Entrada* de material (*InOrders*). Para cada produto cuja quantidade disponível se encontre abaixo do seu nível de reposição de 'stock' é despoletada um *InOrder*. A quantidade pedida nessa *InOrder* é a que falta para repor o máximo de paletes desse produto.
Estes cálculos e a criação das *InOrders* serão da responsabilidade do *SimMaster*, cujo código referente a este evento se apresenta a seguir.

```
void ArmSimulation::SimMaster(float time,UINT event,SimEntity* pEnt)
{

 /*****************************************************************
   Se a entidade que envia o evento é da classe OutOrder:
 *****************************************************************/

 else if(pEnt->m_tipo == OUT_ORDER)
   {
     OutOrder* pOut = (OutOrder*)pEnt;
     switch(event)
     {
      case OUT_ORD_ARRIVE:
                          //Apresentado na secção anterior…
            break;

      case OUT_ORD_END:
            ////////////////////////////////////////////////
            //  Critérios a seguir quando finaliza uma OutOrder:
```

```
                // NOTA: como para o caso da OUT_ORD_ARRIVE, estes critérios são
                // provisórios e esperam ser discutidos com a EFACEC.
                ////////////////////////////////////////////////

                ////////////////////////////////////////////////
                // 1º: Percorrendo a Lista de Produtos do armazém e tendo em conta
                // as respectivas quantidades disponíveis para despoletar uma InOrder,
                // verifica se algum dos produtos necessita de ser encomendado. Se sim,
                // então cria essa InOrder e marca a sua chegada para o instante de tempo
                // calculado com base no tempo previsto de espera pela chegada dessa
                // encomenda de material, que é um parâmetro do produto.
                ////////////////////////////////////////////////

                Produto* pProd;
                InOrder* pInOrder;
                POSITION pos = m_produtList.GetHeadPosition();
               while(pos!=NULL)
               {
                pProd = m_produtList.GetNext(pos);

                int existentes = pProd->m_nPaletes;
                int reservadas = pProd->m_rPaletes;
                int fasquia = pProd->m_nPaletesInOrder;

                if(existentes - reservada <= fasquia)//necessário encomendar
                  {
                      pInOder = CriaEncomenda(pProd);
                      float dT = pProd->m_InOrderTime;
                      Schedule(m_time+dT, IN_ORD_ARRIVE, pInOrder);
                  }
               }

                ////////////////////////////////////////////////
                // 2º: O que fazer à OutOrder que foi finalizada?
                // Uma solução seria guardá-la numa lista de OutOrders satisfeitas, mas
                // por agora optou-se pela sua aniquilação. Digamos que o Simulador
                // não possuirá um registo das encomendas satisfeitas ao cliente, mas
                // somente as trata como entidades em trânsito. O único registo é o
                // número de clientes atendidos, que é guardado em m_nOutOrders.
                ////////////////////////////////////////////////

                delete pOut;

            break;

            default:break;
           }
       }//Foi uma OutOrder
}
```

7.3.5 Eventos e métodos das *InOrders*

Os eventos associados à *InOrder*, ou *Ordem de Entrada* de material, são muito semelhantes aos anteriormente descritos para a *OutOrder*. Igualmente se referem aos instantes de chegada e de fim de processamento. A *InOrder* considerar-se-á processada, e, portanto, finalizada a sua execução, quando estiverem satisfeitos todos os *Pedidos de Material* que a constituem. Os seus dois eventos foram designados por:

- `IN_ORD_ARRIVE` (momento de chegada da *InOrder*)
- `IN_ORD_END` (momento em que a *InOrder* é satisfeita)

de onde resulta a seguinte estrutura de elemento activo na simulação:

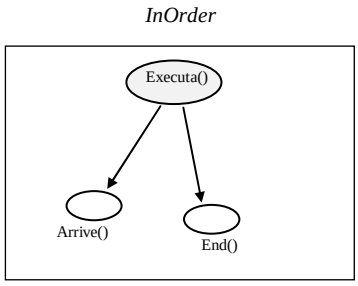

Fig. 7.14 Estrutura da entidade de simulação *InOrder*.

Também neste caso o elemento não carece de representação gráfica para simular movimento, facto pelo qual ignora o método *SimShow()* herdado da classe *SimEntity*. Apesar disso, convém representar um camião junto ao cais a que a *InOrder* for associada, o que poderá ser feito através de uma função específica para o efeito, chamada no início e no fim desse processamento. No início do processo é desenhando um camião junto ao cais, no fim, o camião é retirado do ecrã.

7.3.5.1 *Evento IN_ORD_ARRIVE*

Quando na simulação aparece uma *InOrder* ela é directamente "desviada" para o *SimMaster* a fim de ser analisada e despoletado o processo para satisfazer o conjunto dos seus *pedidos de material* (que neste caso se referem a pedidos do armazém). Por esse motivo, o próprio objecto *InOrder* dirige o evento IN_ORD_ARRIVE para o *SimMaster* ao finalizar o seu método *Arrive()*. Ainda no método *Arrive()* é dado um número de ordem à *InOrder* e estabelecido o seu tempo de chegada.

```
BOOL InOrder::Arrive()
{

    m_time = m_pSim->m_time;   //Tempo do objecto é o tempo actual do Simulador
    m_timeArrive = m_time;     //Tempo de chegada da InOrder

    //incrementa o número de ordem que representa as InOrders chegadas ao armazém.
    m_number = m_pSim->m_nInOrders++;

    ////////////////////////////////////////////////
    //  Envia este evento (IN_ORD_ARRIVE) ao SimMaster para dar inicio
    // a  todo o processo de atendimento da Ordem de Entrada de material.
    ////////////////////////////////////////////////

    m_pSim->SimMaster(m_time, IN_ORD_ARRIVE, this);

 return TRUE;
}
```

Repare-se que não cabe a este método marcar na simulação a chegada da próxima *InOrder*, uma vez que, como já vimos, essa decisão é tomada no fim do processamento de uma *OutOrder*. Mais uma vez, a grande responsabilidade de iniciar o processamento caberá ao *SimMaster*, pois é nele que residem os critérios de "gestão" do armazém. A parte do *SimMaster* relacionada com a chegada de uma *InOrder* é a seguinte:

```
void ArmSimulation::SimMaster(float time,UINT event,SimEntity* pEnt)
{

 /*************************************************************
   Se a entidade que envia o evento é da classe InOrder:
 *************************************************************/

 else if(pEnt->m_tipo == IN_ORDER)
   {
     InOrder* pIn = (InOrder*)pEnt;
     switch(event)
     {
      case IN_ORD_ARRIVE:
            ////////////////////////////////////////////////
            // Critérios a seguir quando aparece uma InOrder:
            // NOTA: estes critérios são provisórios e deverão ser discutidos com a
            // EFACEC de forma a aproximarem-se, o mais possível, dos usados por
            // essa empresa em instalações reais.
            ////////////////////////////////////////////////

            ////////////////////////////////////////////////
            // 1º: Percorrendo a Lista de Material da InOrder, e tendo em conta
            // tanto os produtos como as quantidades requeridas, verifica se é
            // possível atender o fornecedor. Se não for possível rejeita a InOrder.
            ////////////////////////////////////////////////

            if(!IsPossible(pIn)) Reject(pIn);

            ////////////////////////////////////////////////
            // 2º: Sendo possível atender o fornecedor verifica se é possível
            // associar à InOrder um cais de recepção. Se não for possível, por os
            // cais se encontrarem ocupados, coloca a InOrder na Lista de InOrders
            // à espera de cais (m_InOrderWaitList). Esta lista será de novo
            // inspeccionada quando um cais ficar livre, depois de passado ao
            // SimMaster um evento de FIM_TAREFA e se não existirem tarefas à
            // espera de serem executadas.
            ////////////////////////////////////////////////

            else
                {
                  if(!IsThereCaisTo(pIn))
                     m_InOrderWaitList.AddTail(pIn);

                  ////////////////////////////////////////
                  // Se foi possível encontrar um cais livre, automaticamente
                  // esse cais foi associado à InOrder e logo de seguida criada
                  // a primeira tarefa de forma a satisfazer o pedido de material
                  // em causa. A InOrder é colocada na lista m_InOrderRun
                  // do Simulador, e essa tarefa atribuída a um elemento que
                  // se encarrega do transporte do material. Se a tarefa não pode
                  // ainda ser executada é enviada para a lista de tarefas à espera
                  // de execução m_JobWaitingList. De qualquer forma, a
                  // tarefa encontra-se já associada à InOrder.
                  ////////////////////////////////////////
                  else
                     {
```

```
                        Tarefa* pTarefa = CalcTarefa(pIn);
                        StartExecute(pTarefa);
                      }
                }
        break;


    case IN_ORD_END:

            // apresentado na próxima secção...

        break;

        default:break;
       }
    }//Foi uma InOrder
}
```

A inspecção das listas do *Simulador* que contêm *InOrders* é de novo assegurada no final de cada passo da execução de uma tarefa, isto é, quando o *SimMaster* recebe um evento FIM_TAREFA, o que permite à simulação reavaliar a situação das várias *InOrders* já lançadas no sistema. Tal como já foi referido no caso das *OutOrders*, o *SimMaster* encarregar-se-á de executar os seguintes procedimentos (que de novo se apresentam por uma questão de comodidade):

1º Verificar se essa tarefa se encontra realmente terminada. Isso acontece quando não existirem mais *subtarefas* na tarefa a não ser a última de UNLOAD. Se não estiver terminada, a tarefa em questão é dirigida pelo *SimMaster* ao próximo elemento transportador ou colocada na lista de tarefas à espera de serem executadas. Se essa tarefa estiver terminada, então passa ao seguinte procedimento:

2º Verificar se essa tarefa pertence a alguma *OutOrder* da lista de OutOrders em execução. Se pertencer e se a *OutOrder* não estiver finalizada, calcula a próxima tarefa para essa *OutOrder* e manda-a executar se for possível. Se não for possível executá-la envia-a para a lista de espera das tarefas a serem executadas. Se aquela tarefa não pertence a nenhuma *OutOrder* da lista em execução, então passa ao próximo procedimento:

3º Verificar se essa tarefa pertence a alguma *InOrder* da lista de InOrders em execução. Se pertencer e se a *InOrder* não estiver finalizada, calcula a próxima tarefa para essa *InOrder* e manda-a executar se for possível. Se não for possível executá-la envia-a para a lista de espera das tarefas a serem executadas. Se aquela tarefa não pertence a nenhuma *InOrder* da lista em execução, então passa ao próximo procedimento:

4º Verificar se existe alguma nova tarefa para executar na lista de tarefas em espera . Essa acção é realizada verificando o tamanho da lista `m_JobWaitingList` do *Simulador*. Se existir alguma tarefa que possa ser executada por aquele elemento, então dá essa tarefa a esse elemento. Se não existirem tarefas na lista de espera ou se nenhuma for possível de atribuir àquele elemento, então passa ao seguinte procedimento:

5º Verificar se a tarefa terminada libertou algum cais. Essa acção é realizada verificando se o suporte da célula de UNLOAD é um cais. Se for um cais de expedição, percorre a lista de *OutOrders* à espera de cais e tenta atribuí-lo. Se foi atribuído, manda iniciar o processamento da *OutOrder*, retirando-a da lista de espera e colocando-a na lista `m_OutOrderRun`. Se a célula de suporte era um cais de recepção, executa as mesmas operações mas usando a lista de *InOrders* à espera de cais.

6º Como último procedimento a ser sempre executado: Se o elemento transportador era um veículo e se não lhe foi atribuída nova tarefa, dá-lhe ordem de recolher ao seu parque de estacionamento.

7.3.5.2 *Evento IN_ORD_END*

Este evento é despoletado pelo *SimMaster* quando verifica que o material de uma *InOrder* se encontra completamente acomodado. O método que corresponde a este evento é o seguinte:

```
BOOL InOrder::End()
{

    m_time = m_pSim->m_time;    //Tempo do objecto é o tempo actual do Simulador
    m_timeEnd = m_time;         //Tempo de finalizada a InOrder
    ////////////////////////////////////////////////
    //  Retira a InOrder da lista de InOrders em processamento, se lá existir,
    //  pois já se encontra finalizada.
    ////////////////////////////////////////////////

    POSITION pos = m_pSim->m_InOrderRun.Find(this);
    if(pos!=NULL) m_pSim->m_InOrderRun.RemoveAt(pos);

    ////////////////////////////////////////////////
    //  Envia este evento (IN_ORD_END) ao SimMaster para o avisar que
    //  chegou o fim do  processo de atendimento do actual fornecedor.
    ////////////////////////////////////////////////

    m_pSim->SimMaster(m_time, IN_ORD_END, this);

 return TRUE;
}
```

Como este evento representa o instante em que o material foi acomodado no armazém, nele se inclui a reposição do ‘stock’. Para cada produto da *InOrder* é adicionado o respectivo número de paletes que entraram. Essa tarefa é levada a cabo pelo *SimMaster*, cujo código referente a este evento se apresenta a seguir.

```
void ArmSimulation::SimMaster(float time,UINT event,SimEntity* pEnt)
{

 /*****************************************************************
   Se a entidade que envia o evento é da classe InOrder:
 *****************************************************************/

 else if(pEnt->m_tipo == IN_ORDER)
   {
     InOrder* pIn = (InOrder*)pEnt;
     switch(event)
     {
      case IN_ORD_ARRIVE:
                          //Apresentado na secção anterior…
            break;

      case IN_ORD_END:
            ////////////////////////////////////////////////
```

```
                // Critérios a seguir quando finaliza uma InOrder:
                // NOTA: como para o caso da IN_ORD_ARRIVE, estes critérios são
                // provisórios e esperam ser discutidos com a EFACEC.
                ////////////////////////////////////////////////

                ////////////////////////////////////////////////
                // 1º: Percorrendo a lista de Pedidos de Material da InOrder
                // as respectivas quantidades disponíveis para despoletar uma InOrder,
                // verifica se algum dos produtos necessita de ser encomendado. Se sim,
                // então cria essa InOrder e marca a sua chegada para o instante de tempo
                // calculado com base no tempo previsto de espera pela chegada dessa
                // encomenda de material, que é um parâmetro do produto.
                ////////////////////////////////////////////////

                PedidoMaterial* pMat;
                Produto* pProd;
                POSITION posProd;

                POSITION pos = pIn->m_listaMaterial.GetHeadPosition();

                while(pos!=NULL)
                {
                 pMat = pIn->m_listaMaterial.GetNext(pos);
                 posProd = m_produtList.FindIndex(pMat->m_produto);
                 if(posProd!=NULL)//existe o produto no armazém
                   {
                       pProd = m_produtList.GetAt(posProd);
                      pProd->m_nPaletes += pMat->m_nUnidades;
                   }
                }

                ////////////////////////////////////////////////
                // 2º: O que fazer à InOrder que foi finalizada?
                // Uma solução seria guardá-la numa lista de InOrders satisfeitas, mas
                // por agora optou-se pela sua aniquilação. Digamos que o Simulador
                // não possuirá um registo das encomendas feitas ao fornecedor, mas
                // somente as trata como entidades em trânsito. O único registo é o
                // número de fornecedores atendidos guardado em m_nInOrders.
                ////////////////////////////////////////////////

                delete pIn;

            break;

            default:break;
           }
       }//Foi uma InOrder
}
```

7.3.6 Ordem de reabastecimento de zona de 'Picking'

Como já foi referido, cada produto possui uma zona de armazenagem na qual é permitida efectuarem-se operações de 'picking'. À medida que daí vão sendo retiradas caixas de produto o número de caixas disponível diminui. Por isso, a determinada altura, torna-se necessário reabastecer essa zona, processo que será despoletado por uma *ordem de reabastecimento de zona de 'picking'* dirigida a esse produto. A esta ordem fez-se corresponder o evento PRD_FILL_PICKING gerado assim que o número de caixas disponível atinge um determinado nível mínimo. O veículo que se encontre a retirar caixas dessa zona é o responsável por despoletar essa ordem, e de a enviar de seguida ao *SimMaster* para que este

se encarregue do processo de reabastecimento. O *SimMaster* analisa a situação, cria uma tarefa que solucione o pedido e lança-a na simulação para que seja executada logo que possível. O correspondente código do "gestor" é o seguinte:

```
void ArmSimulation::SimMaster(float time,UINT event,SimEntity* pEnt)
{
 /*****************************************************************
    Se a entidade é da classe Produto:
 *****************************************************************/

 else if(pEnt->m_tipo == PRODUTO)
   {
     Produto* pProd = (Produto*)pEnt;
     switch(event)
     {
      case PRD_FILL_PICKING:
            ////////////////////////////////////////////////
            // Critério a seguir quando um Produto necessita de reabastecer a zona
            // de 'picking':
            // Calcula o número de paletes desse produto a transportar para a zona
            // de 'picking'.  Calcula depois a tarefa para resolver a questão e
            // coloca-a na lista de tarefas à espera de execução.
            ////////////////////////////////////////////////
             int existem = pProd->m_nCaixasPIC;
             int reserva = pProd->m_rCaixasPIC;
             int maximo  = pProd->m_maxCaixasPIC;
             float ratio = pProd->m_percentPicking;
             int caixasPalete = pProd->m_nCaixasPalete;

             int deslocar;
             int palDeslocar;
             Tarefa* pTarefa;
            deslocar = maximo - (existem - reserva);
            palDeslocar = (int)(0.5 + deslocar/caixasPalete);
             if(palDeslocar > 0)
               {
                pTarefa = CalcTarefaPick(pProd);
                m_JobWaitingList.AddTail(pTarefa);
               }
       break;

      default:break;
     }
   }//Foi uma ordem de reabastecimento de zona de 'picking'
}
```

8. Criação do modelo no computador

Tendo em conta o tipo de concepção utilizada no desenvolvimento deste programa, a criação do modelo de um armazém deverá passar por três fases distintas. Na primeira é desenhado o armazém à escala, definindo e posicionando os diversos tipos de áreas: a área limítrofe, as áreas de expedição e de recepção e as áreas interditas. Na Segunda fase são criados os diversos elementos físicos, especificando as suas características e o seu posicionamento dentro do espaço do armazém: definem-se as estantes, as vias de movimentação, os veículos, etc. Na terceira fase inicia-se a simulação propriamente dita, na qual se poderá apreciar a actividade dos vários elementos do armazém com o decorrer do tempo.

Ao iniciar a criação do modelo, o programa pede ao utilizador que introduza as dimensões do rectângulo onde irá ser inscrito o armazém. Essas dimensões são dadas em metros e o programa automaticamente desenha no ecrã um rectângulo a tracejado com a correspondente relação de comprimentos. É dentro desse espaço que é permitido representar os restantes elementos. A origem das coordenadas è o canto superior esquerdo desse rectângulo (fig.8.1).

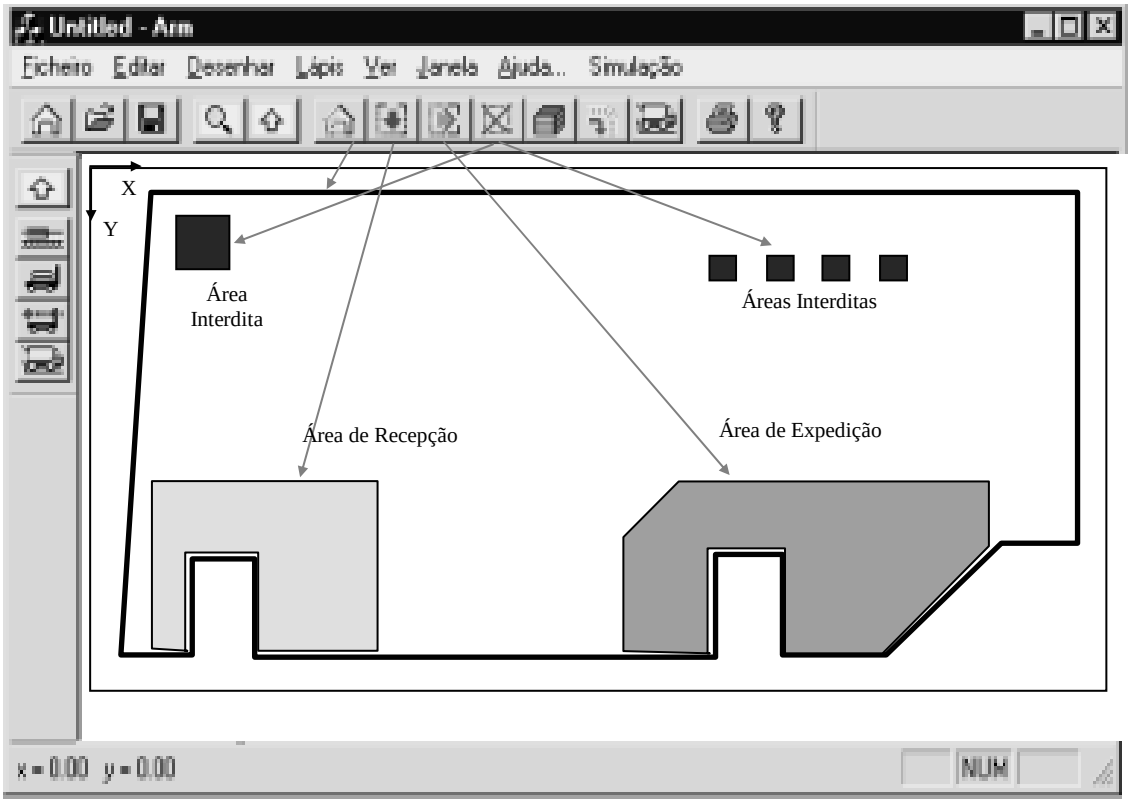

Fig. 8.1 Aspecto geral do programa *Armazém* no início da criação de um modelo.

Depois disso, o utilizador começa por definir, à escala, as diversas áreas existentes no seu armazém, usando o programa como quem usa uma ferramenta de desenho. O tipo

de área a desenhar é escolhido ou na '*ToolBar*' ou através da respectiva opção do menu *Desenhar*. Depois de traçada uma área ela poderá ser movimentada, duplicada, apagada, etc., como na maioria das aplicações dedicadas ao desenho vectorizado.
O posicionamento da área dentro do espaço do armazém é facilitado pela constante apresentação das coordenadas do "rato" na '*StatusBar*'. Essas coordenadas são expressas em metros e referidas ao sistema de eixos representado na figura anterior. Para assegurar um posicionamento mais fino o operador poderá fazer uso da opção de *ZOOM* por janela, também disponibilizada nesta aplicação.
Depois de definidas e posicionadas as áreas de interesse, o operador poderá começar a preocupar-se com os restantes elementos. Normalmente segue-se a representação das estantes, no entanto, não é forçoso que assim seja, podendo desde já definirem-se vias de movimentação, veículos, tapetes, etc.
Para um melhor entendimento das possibilidades de criação de objectos e das acções permitidas ao utilizador, apresenta-se na próxima figura a legenda dos comandos disponíveis nas '*ToolBars*' do programa.

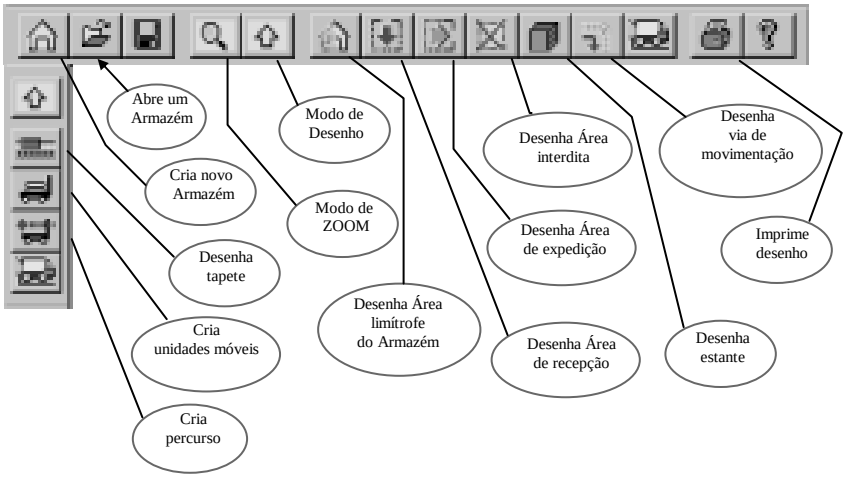

Fig. 8.2 Comandos acessíveis nas '*tolbars*' do programa.

Todos estes comandos são igualmente acessíveis através de opções do menu, o que facilita ao operador a escolha do método a seguir. É de referir ainda que o menu permite aceder a outros comandos que aqui não serão referidos, uma vez que não se pretende descrever o modo de utilizar a aplicação, mas somente apresentar os passos que levam à criação de um modelo de armazém.

Comecemos pela criação de estantes. Para isso o utilizador deverá usar a opção do menu Desenhar->Estantes ou premir o botão respectivo acessível na '*ToolBar*'. O programa passa automaticamente ao modo de desenho e a primeira vez que o botão do "rato" for premido é definido o ponto de origem da estante. À medida que o "rato" se movimenta é mostrada no ecrã uma previsão da área ocupada pela futura estante, assim como o número de alvéolos que possuirá ao longo de cada direcção, o que permite ao operador melhor decidir sobre as dimensões da estante a criar. A futura estante apresentará as características que previamente foram introduzidas na seguinte *Janela de diálogo*:

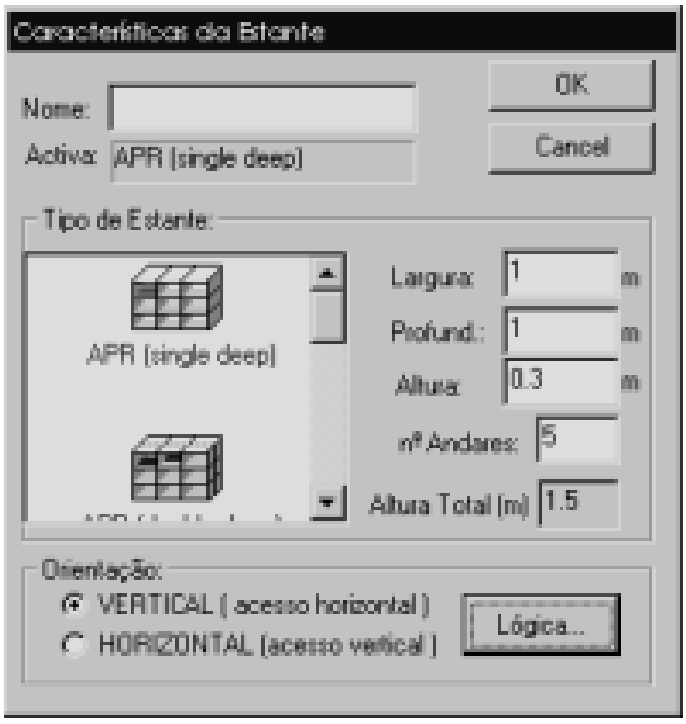

Fig. 8.3 Diálogo para caracterização da estante.

É através desta janela que o operador define o tipo de estante, as suas dimensões e a sua orientação. Se pretender poderá também estabelecer a lógica de acesso à estante premindo o botão 'Lógica', o que fará aparecer o seguinte diálogo:

Fig. 8.4 Diálogo da lógica de acesso à estante.

Aqui se representam os dois lados da estante e as direcções de acesso ao material em cada um desses lados. O lado ESQUERDA/BAIXO representa o lado ESQUERDO se a estante é VERTICAL e o lado de BAIXO se a estante é HORIZONTAL. O lado DIREITA/CIMO refere-se ao lado DIRETO numa estante VERTICAL e ao lado de cima

numa HORIZONTAL. Com base nesta representação o operador é levado a escolher a forma de acesso pretendida, no entanto, o programa só validará esses acessos se eles puderem ser suportados pelo tipo de estante em causa.

Depois de criadas e posicionadas as estantes, passemos à criação das vias de movimentação de material. Neste caso o utilizador deverá usar a opção do menu Desenhar->ViasMovimentação ou premir o botão respectivo da '*ToolBar*'. De novo o programa passa ao modo de desenho e a primeira vez que o botão do "rato" for premido é definido o ponto de origem da via. À medida que o "rato" se movimenta é mostrada no ecrã uma previsão da futura via, permitindo ao operador apurar com mais facilidade o aspecto final desse elemento. O tipo de via a ser desenhado é previamente escolhido através do seguinte diálogo:

Fig. 8.5 Diálogo para escolha do tipo de via.

Note-se que, a qualquer momento, é possível alterar os atributos da via, quer através de um menu de "propriedades", que aparecerá quando se utiliza o botão direito do "rato", quer quando se executa um DUPLO-CLIC sobre esse elemento de desenho.

Assim que as vias se encontrem criadas poderão ser associadas a estantes, isto é, para uma determinada estante poderão definir-se as vias através das quais os veículos a devem aceder. Esta associação é feita automaticamente pelo programa, como já foi referido numa das secções anteriores reservadas à concepção do objecto estante, e despoletada pela opção do menu Ver->MostrarViasEstantes. O operador somente necessita de afinar o posicionamento dessas vias e de decidir sobre a lógica de acesso dessa estante particular, de forma a que seja possível estabelecer-se tal associação. Se a via foi a associada é desenhada a uma cor diferente para melhor a destacar das outras.

O operador pode agora partir para a criação dos elementos móveis, escolhendo de uma lista o tipo de elemento pretendido. Essa lista é visualmente apresentada numa *Janela de diálogo* e pretende-se que venha a conter todos os tipos de elementos móveis usados pela EFACEC. Para já, permite escolher entre *Stocadores*, *AGVs* e *tapetes*:

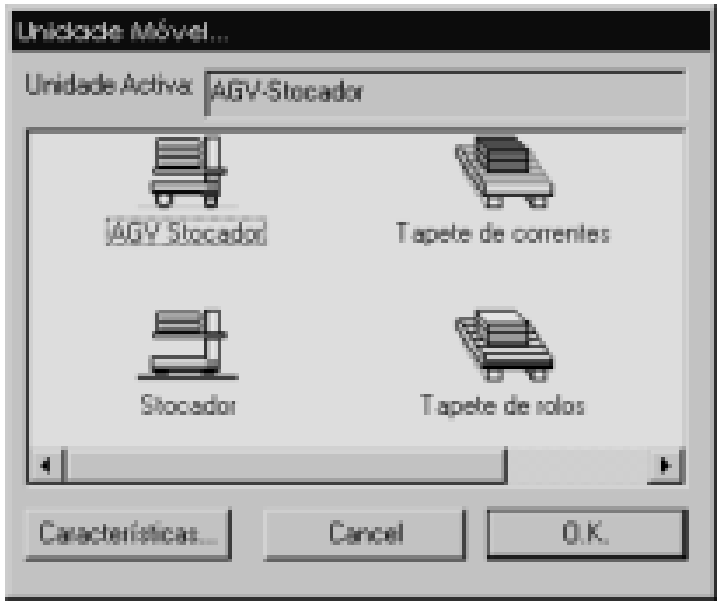

Fig. 8.6 Diálogo para escolha do elemento móvel.

Para cada elemento móvel escolhido é possível definirem-se as suas características, executando um DUPLO-CLIC no respectivo ícone ou premindo o botão "características" dessa *Janela de diálogo*. Repare-se que é neste diálogo que se deverá escolher tanto o tipo do elemento móvel como o modelo pretendido dentro desse tipo.
Quando se escolhe um veículo, poder-se-á de seguida definir as suas propriedade através do diálogo(1) :

Fig. 8.7 Diálogo das características do veículo.

(1) O diálogo que se apresenta faz parte da última versão disponível do programa, sendo por isso um pouco diferente daquele que se encontra em construção neste momento. Servirá, no entanto, para dar uma ideia de como são introduzidos os dados nos veículos.

Aqui se estabelecem as velocidades do veículo, os seus tempos de carga e de descarga e as suas dimensões. O programa automaticamente criará uma representação visual desse veículo que coloca na janela da simulação. Depois esse 'ícon' poderá ser posicionado onde o operador desejar dentro da área do armazém. No entanto, o veículo só se tornará acessível a partir do momento em que o operador define um cais para ele, o que automaticamente desloca esse veículo para o local do cais atribuído.

Tratando-se de uma escolha de tapetes, o diálogo onde se especificam as suas características é o seguinte:

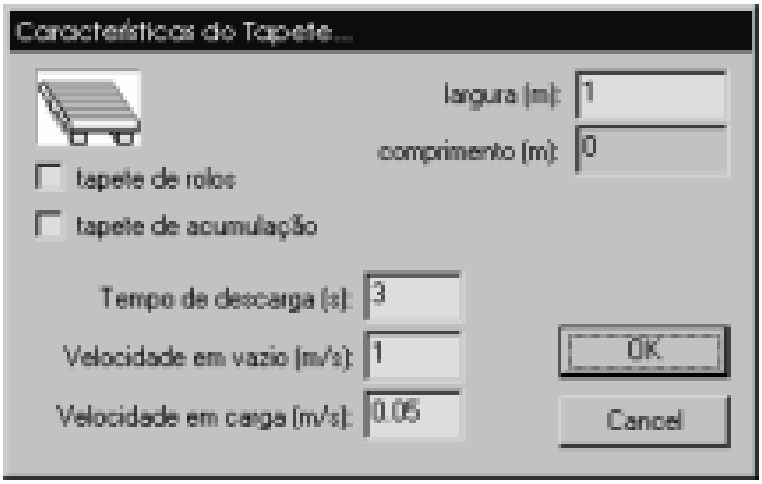

Fig. 8.8 Diálogo das características do tapete.

Neste diálogo o operador define o modelo de tapete, a sua largura, a sua velocidade e o tempo de descarga. A velocidade em vazio, isto é, quando não existe carga no tapete, não é para já considerada e, provavelmente não terá grande aplicação na versão do programa que se encontra em desenvolvimento. O comprimento do tapete só será conhecido depois de o operador acabar de o desenhar no ecrã com a ajuda do "rato". Por essa razão esse campo só permite leitura, e não introdução de dados.
Este diálogo é também usado para alterar as características de um tapete já definido, sendo nesse caso acedido através de um DUPLO-CLIC do "rato" no respectivo elemento de desenho. Nessa altura, aparecerá no campo "comprimento" o actual comprimento do tapete em questão.

Uma vez que a interface com o utilizador ainda não contempla os restantes elementos da simulação, tais como a introdução de produtos, mesas de transferência, etc., e como o principal objectivo desta primeira fase do trabalho era a modelação dos elementos móveis, fica-se pela apresentação da criação do modelo até este ponto, momento em que já será possível apreciar tanto a movimentação de material numa malha de tapetes como a deslocação de veículos ao longo de percursos predefinidos. Nesta fase, pretende dar-se uma ideia tanto do tipo de interface com o utilizador como da flexibilidade conseguida com um sistema deste tipo.

Construída esta parte do modelo, a simulação destes processos é despoletada através da opção do menu Simulação->Início, o que permitirá desde já visualizar o funcionamento dos tapetes e a movimentação dos veículos. A seguir apresenta-se uma imagem do

programa que representa um modelo a ser simulado, incluindo uma malha de tapetes e alguns percursos de veículo.

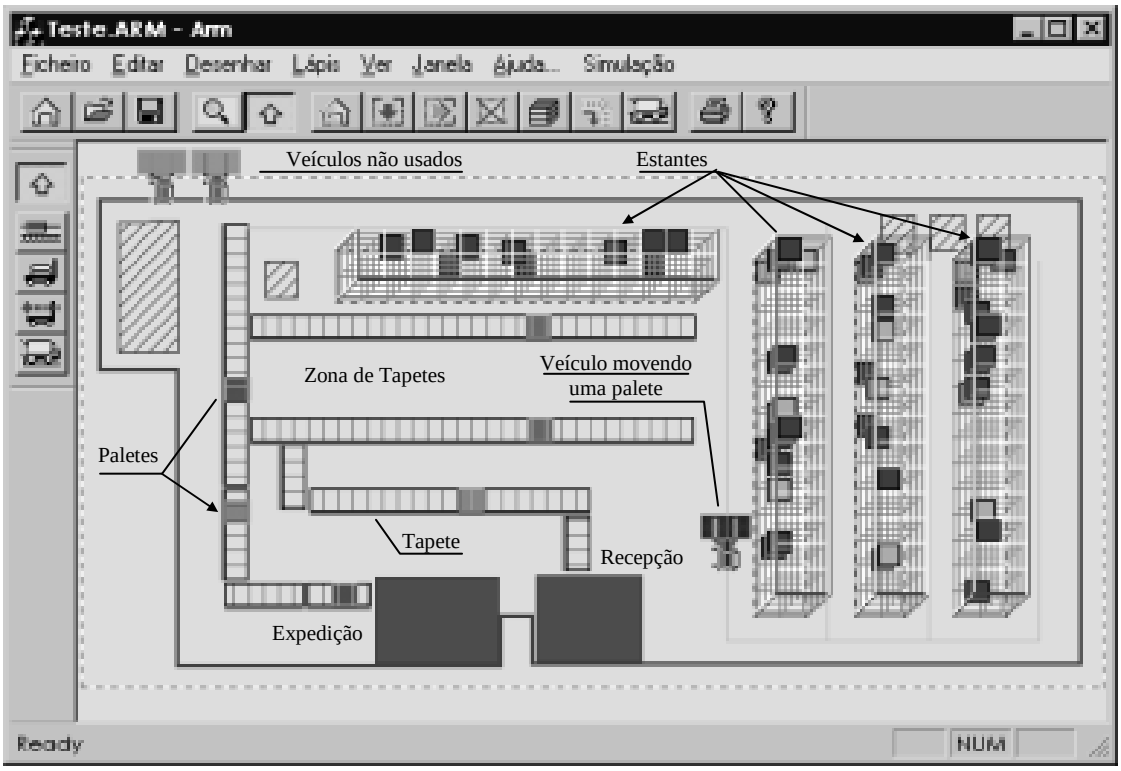

Fig. 8.9 Aspecto de uma simulação de tapetes e veículos.

A qualquer momento é permitido ao operador cancelar a simulação, podendo depois serem alteradas as posições dos tapetes ou dos restantes elementos para de novo dar inicio à simulação. Cada elemento pode ser modificado através da sua janela de "propriedades", acessível com um DUPLO-CLIC nesse elemento, o que permite ao utilizador estudar e intervir na performance final prevista pelo modelo. Podem-se mover as estantes e reconfigurar as suas dimensões, assim como introduzir novos veículos e alterar o 'layout' do armazém, sendo depois somente necessário usar a opção Simulação->Início para que a nova simulação tenha início.

9. Conclusões e trabalho futuro

Os resultados obtidos com a estrutura de simulação que aqui se propõe mostram a exequibilidade do modelo conceptual desenvolvido, pois foi possível pôr a funcionar na aplicação informática implementada os mecanismos de base mais importantes de um armazém automático.
Como resultados observáveis, distinguem-se a movimentação e transporte de material em malhas de tapetes e a movimentação de veículos em percursos predefinidos. Contudo, há que ter em conta todo o trabalho de base que tornou possível estes resultados, tais como a criação das várias classes de objectos, o desenvolvimento das ferramentas de desenho incluídas no programa, e, principalmente, a concepção da estrutura geral em que assenta a simulação.
Não se trata, por isso, de uma mera simulação particular de tapetes e veículos, mas sim de um resultado integrado numa estrutura mais geral de simulação de armazéns. Até agora, nenhumas tentativas foram feitas no sentido de quantificar a precisão e a fiabilidade destes resultados, tendo-se somente observado, de uma forma qualitativa, a coerência dos processos simulados.

A ideia de separar em diferentes níveis de lógica as responsabilidades dos processos envolvidos num armazém automático (objectos, controlador e "gestor"), permite olhar para este *Simulador* como um sistema constituído por diversos módulos independentes, e, para os elementos do armazém, como entidades estanques cujas características e funcionalidade se manterão para além do próprio *Simulador*. Tal abordagem confere a possibilidade de alterar uns sem a necessidade de modificar os outros, e, para além disso, a portabilidade. Se, por exemplo, todos os objectos definidos forem incluídos numa DLL (*Dynamic Link Library*) facilmente poderão ser utilizados por outras aplicações. Como é óbvio, a somar a estas vantagens há que assinalar a natural portabilidade do código da linguagem usada (fala-se de portabilidade entre aplicações e não entre diferentes sistemas operativos).

Em certa medida, esta abordagem aproxima, de facto, a estrutura do modelo à realidade, tanto porque os eventos passam a ser entendidos como "pertencentes" aos objectos, e não como propriedade exclusiva do sistema como um todo, como pela "hierarquia de decisão" conseguida com a estrutura do *Simulador* aqui apresentada. Este somente deverá encarregar-se de traduzir as suas determinações para uma "linguagem" de eventos que tenha em conta o tipo de objectos presentes. É o que se passa numa instalação real: se o sistema de controle envia uma ordem de arranque a um tapete é porque a existência desse tapete lhe permite gerar uma ordem desse tipo. Retire-se o tapete e deixarão de ter sentido semelhantes ordens.
Em particular no que respeita à modelação dos elementos físicos do armazém, isto é, tapetes, estantes, veículos, mesas, etc., foi definido um plano de base para a inclusão de futuros elementos, de forma a que o utilizador possa vir a contar com uma espécie de

biblioteca de equipamentos *standards* acessíveis no mercado. A maior ou menor precisão dos resultados obtidos na simulação dependerá, em grande parte, tanto da minúcia com que tais elementos forem modelados como dos critérios de controle e gestão permitidos.

Em termos de paradigma de simulação, esta abordagem não se poderá considerar usual, uma vez que, na maioria dos modelos, não são considerados diversos níveis de "responsabilidade" nas acções a tomar, sendo os próprios objectos a decidirem o que fazer em quaisquer circunstâncias. No entanto, trata-se aqui de simulação aplicada a uma área específica, armazéns automáticos, e não de um gerador de modelos com ambições generalistas. Apesar disso, e em certa medida, esta abordagem corresponde a uma das propostas por Bolier e Eliens[6] em que as entidades permanecem no sistema durante toda a simulação. No entanto, enquanto que noutro tipo de abordagens os critérios de decisão são incluídos dentro dos métodos das próprias entidades, aqui esses critérios são apartados delas e agrupados noutros módulos, permitindo assim uma maior portabilidade dessas entidades e uma mais fácil adaptação do sistema a novos critérios que se desejem implementar.
Em termos de velocidade de processamento, esta abordagem poderá parecer menos eficiente do que se os eventos tivessem sido directamente "atendidos" pelo próprio *Simulador* e se não se tivesse considerado o "desvio" de determinados eventos para o *SimMaster* ou para o *SimControl*, no entanto, dadas as actuais performances dos sistemas informáticos, tal questão não necessite de ser considerada relevante.
Da experiência adquirida durante este trabalho, e depois de apreciada a flexibilidade deste modo de formular o problema, talvez não seja de todo desinteressante tentar-se generalizar a outros casos de simulação a filosofia dos "três níveis de lógica", principalmente tratando-se de sistemas complexos onde existam decisões a tomar em face dos estados das diversas entidades do sistema.

No que respeita ao trabalho futuro, e em particular ao desenvolvimento da aplicação informática aqui referida, há ainda que aperfeiçoar e implementar o resto da interface com o utilizador, assim como melhorar as técnicas de desenho do armazém à escala e implementar os algoritmos de escolha de percursos e de criação de tarefas. Alguns elementos carecem ainda de modelação, entre eles o *Transfer*. Para dar aos veículos a possibilidade de utilizarem percursos "em diagonal" é necessário reestruturar a sua classe de objectos, assim como alguns dos seus métodos. Rotações, que neste momento somente são permitidas em inteiros de 90º, deverão ser tornadas possíveis para qualquer ângulo.
Deverão também ter-se em conta as restrições de determinadas associações de cais a veículos, de parques e de percursos, e, em geral, implementar um processo de validação global do modelo, a ser executado antes do início da simulação. Neste processo o programa deverá encarregar-se de analisar toda a informação contida no modelo e, se for caso disso, avisar o utilizador das incongruências que possam ser detectadas.
Pensa-se também "afinar" os critérios envolvidos no controle e na gestão, através de novas reuniões com a EFACEC, assim como aperfeiçoar certos conceitos depois de visitadas mais instalações reais.
Pensa-se incluir em cada objecto uma zona reservada ao controle estatístico, de forma a que, a qualquer momento, se possa apreciar a evolução dos parâmetros da sua

actividade. Na generalidade, tem-se em mente incluir no programa funções estatísticas, histogramas, relatórios, etc. A simulação de avarias é outra proposta que se perspectiva para o futuro.

Referências

[1] Michael Pidd, "*Computer Simulation in Management Science*", John Wiley & Sons Ltd, 1992

[2] A.E.S.C. Brito, "*Configuring Simulation models using CAD techniques: a new approach to warehouse design*", Cranfield University of Technology, School of Management, PhD thesis, 1992

[3] Osman Balci, et al., "*Introduction to the Visual Simulation Environment*", Arca Computer, Inc, Virginia Tech Corporate Research Center, Blacksburg, Virginia, U.S.A., Winter Simulation Conference 1997.

[4] A. Alan B. Pritsker, "*Introduction to Simulation and SLAM II*", John Wiley & Sons, New York, and Systems Corporation, Indiana, 1986.

[5] EFACEC, "*GESTRA2000 - Interface WMS-WCS - Concepts and message flow*", Doc nr. 1 - Rev. 1.4, 15-07-97

[6] Dirk Bolier, Anton Eliens, "*SIM : a C++ library for Discrete Event Simulation*", Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam. October 1995.

Bibliografia

- Allan Carrie, "*Simulation of Manufacturing Systems*", John Wiley & Sons, 1988.
- Averill M. Law, W. David Kelton, "*Simulation Modeling & Analysis*", McGraw-Hill International Editions, Industrial Engineering Series, 1991.
- Derek Merrill and James S. Colofello, "*Improving Software Managment Skills Using a Software Project Simulator*", Department of Computer Science and Engineering, Arizona State University, Tempe, U.S.A.
- Hugh J. Watson, John H. Blackstone Jr., "*Computer Simulation*", John Wiley & Sons, 1989.
- Manual of "*Simulation with SIMFACTORY II.5 and SIMPROCESS*", CACI Products Company.
- Michael Pidd, "*Computer Modelling for Discrete Simulation*", Michael Pidd, John Wiley & Sons, 1989.
- Osman Balci, et al., "*Developing a Library of Reusable Model Components by Using the Visual Simulation Environment*", Arca Computer, Inc, Virginia Tech Corporate Research Center, Blacksburg, Virginia, U.S.A., July 1997.
- Osman Balci, et al., "*Dinamic Object Decomposition in the Visual Simulation Environment*", Arca Computer, Inc, Virginia Tech Corporate Research Center, Blacksburg, Virginia, U.S.A., June 1997.
- Osman Balci, et al., "*The Visual Simulation Environment*", Arca Computer, Inc, Virginia Tech Corporate Research Center, Blacksburg, Virginia, U.S.A., June 1997.
- Osman Balci, et al., "*Introdution to the Visual Simulation Environment*", Orca Computer, Inc, Virginia Tech Corporate Research Center, Blacksburg, Virginia, U.S.A., 1997.
- Robert E. Shannon, "*Systems Simulation, the art and the science*", Prentice-Hall, New Jersey, 1975.
- Ruth Davies, Robert O'Keefe, "Sim*ulation Modelling with Pascal*", Prentice Hall, 1989.
- S. Narayanan, et al., "*Research in Object-Oriented Manufacturing Simulations: An Assessment of the State of the Art*", College of Engineering and Computer Science, Wright State University, Dayton U.S.A.
- "*Visual C++ Books Online, C++ Language Reference*", Microsoft Corporation, 1992-1995.