DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
INSTITUTO DE CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DE MINAS GERAIS

PHOTOPIX: Uma plataforma para sistemas de processamento digital de imagens orientada para objetos

Dissertação apresentada ao Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Alisson Augusto Souza Sol

Orientador: Arnaldo de Albuquerque Araújo

05 de Março de 1993
Belo Horizonte - MG
Brasil

Resumo

Este trabalho descreve como o uso das técnicas de programação orientada para objetos pode ajudar a isolar a implementação de algoritmos em sistemas de processamento digital de imagens da codificação de funcionalidade "não essencial". Com esta diretriz foi concebido e implementado um sistema, denominado **PHOTOPIX**, que foi codificado na linguagem C++, usando a interface para programação de aplicativos do ambiente *MS-Windows*.

Adicionalmente, foram investigadas técnicas para conversão da informação espectral entre imagens com diferentes resoluções espectrais.

[**Palavras-chave**]: Processamento digital de imagens, programação orientada para objetos, ambiente *MS-Windows.*

Abstract

This work describes how the use of object-oriented programming techniques can help to isolate the implementation of algorithms in digital image processing systems from the coding of "non-essential" functionality. With that directive it was conceived and implemented a system, named **PHOTOPIX**, which was coded in the C++ language, using the application programming interface of the *MS-Windows* environment.

Methods for converting the spectral information between images of different spectral resolution were also investigated.

[**Keywords**]: Digital image processing, object-oriented programming, *MS-Windows* environment

Agradecimentos

Este trabalho é dedicado aos meus pais, aos quais agradeço pelo constante apoio e pelo exemplo de vida.

Agradeço a todos os meus familiares e amigos pela convivência, em especial à minha avó Alice e aos meus tios Leopoldino e Bernadete, que forneceram um suporte fundamental ao início da minha educação formal.

Ao Professor Arnaldo, pela orientação e confiança.

A Denise, pela compreensão e paciência.

A Maurício Bussab, da Microsoft Corporation, pelo incetivo e suporte.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq, pelo auxílio financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1
Introdução

A contínua evolução das tecnologias de fabricação de circuitos integrados torna cada vez maior a capacidade de processamento dos computadores. Como resultado disto, é crescente o número de plataformas de *hardware*[1] capazes de efetuar de maneira eficiente o processamento digital de imagens (PDI) .

A finalidade dos sistemas de PDI é permitir a melhoria na qualidade de imagens digitais e automatizar procedimentos de extração de informação da imagem, como a segmentação e o reconhecimento de padrões [Jain89, Nibl86, Pratt78].

Entretanto, dois grandes problemas ainda impedem que programadores de sistemas de PDI concentrem-se unicamente na implementação da numerosa quantidade de algoritmos para tratamento de imagens sendo pesquisados na atualidade:

- a falta de padronização para o formato de arquivos armazenando imagens digitais;

- a incompatibilidade entre as diversas plataformas de *hardware* e APIs (*Application Programming Interface*, isto é, interface para programação de aplicativos) disponíveis, dificultando a portabilidade de código-fonte, em especial o relativo à interface aplicativo-usuário.

A impossibilidade de implementação de um sistema fornecendo suporte (leitura/escrita) a todos os tipos de arquivos de imagens existentes, assim como a todas as plataformas de *hardware* e APIs disponíveis, obriga o desenvolvedor[2] a concentrar-se em um número limitado de escolhas.

[1] Termos como *hardware*, *software*, *bits*, *bytes*, *pixels*, e outros neologismos comuns na área da Informática não serão definidos. Definições podem ser encontradas em [Frag86].

[2] Neste trabalho, a palavra "desenvolvedor" designa todos os envolvidos na implementação de um sistema: analistas, programadores, desenhistas de telas, gerente de projeto, etc.

Infelizmente, não apenas méritos técnicos podem ser utilizados para decidir dentre as diversas possíveis opções. Pressões comerciais têm freqüentemente levado ao completo desuso tanto formatos de arquivo que já foram muito utilizados quanto plataformas de *hardware* e APIs que poderiam até mesmo ser consideradas avançadas demais para sua época de lançamento.

1.1 Definição do sistema PHOTOPIX

O objetivo do sistema **PHOTOPIX** é fornecer uma base, tanto conceitual quanto prática, para sistemas de processamento de imagens. Tal sistema deve poder migrar não apenas entre diversas plataformas de *hardware* e APIs da atualidade, mas também ao longo do tempo, implementando plenamente o conceito de reusabilidade de *software* [Cox86].

Muitos esforços têm sido duplicados na codificação de algoritmos de PDI, simplesmente porque não se consegue reaproveitar a maior parte do código já existente. Isto ocorre porque, na maioria das implementações, é difícil separar o que seja código relativo ao algoritmo do que vem a ser código para manutenção da interface aplicativo-usuário, leitura/escrita de arquivos ou tratamento de restrições da API e do *hardware* utilizados.

A principal diretriz do sistema **PHOTOPIX** é isolar a implementação de algoritmos de PDI da codificação de outras funcionalidades "não essenciais".

A contínua evolução das tecnologias usadas na Informática, aproxima as plataformas de *hardware* atuais cada vez mais das concebidas teoricamente segundo o princípio de tecnologia perfeita da Análise Essencial [McPa84]. Portanto, um sistema que pretenda portabilidade ao longo do tempo não deve conter, ou deve restringir ao mínimo possível, a sua dependência de configurações de *hardware* específicas.

Diversas APIs disponíveis atualmente permitem a codificação completa de um sistema sem qualquer (ou com um mínimo de) referência à plataforma de *hardware* em uso. Dentre estas, a API do ambiente operacional *MS-Windows* foi escolhida como base para a primeira implementação prática do sistema **PHOTOPIX**.

O uso de técnicas de programação orientada para objetos é a melhor opção atualmente disponível para implementação de sistemas onde se deseja "encapsular" detalhes e "isolar" diversos módulos. Para codificação do sistema **PHOTOPIX**, foi escolhida a mais popular das linguagens orientadas para objetos: a linguagem **C++**.

A solução mais difundida para lidar com o problema da multiplicidade de formatos para arquivos contendo imagens digitais consiste em padronizar apenas um formato para os dados sobre os quais atuam os algoritmos de PDI implementados em um sistema. São codificados então diversos conversores entre este "formato interno" das imagens em um sistema de PDI e os inúmeros formatos existentes para arquivos contendo imagens. O sistema **PHOTOPIX** adota um formato interno para as imagens que é denominado *Packed-DIB* [Petz91a]. Tal formato foi escolhido por ser compatível com algumas primitivas gráficas do ambiente *MS-Windows* [MSDK92].

Atualmente, considera-se que uma resolução espectral de 24 *bits*/*pixel* é suficiente para a maior parte das aplicações de processamento de imagens. É comum utilizar o sistema RGB[3] [FVFH90] para especificação das cores em uma imagem digital, designando 8 *bits*/*pixel* para cada componente. Contudo, diversos formatos populares para arquivos contendo imagens não possibilitam o armazenamento de 24 *bits* de informação espectral por *pixel*. Além disto, é comum desejar obter-se imagens em escala de tons de cinza [HeBa86] ou com baixa resolução espectral, para impressão, transmissão ou armazenamento em dispositivos de baixa capacidade (como disquetes).

É consenso que uma plataforma para sistemas de PDI não deve obrigar o programador de um algoritmo a implementar versões específicas para cada uma das resoluções espectrais aceitas pelo sistema, ou mesmo optar por um código genérico e ineficiente. A maioria dos sistemas de PDI permite que determinados algoritmos sejam implementados apenas para imagens com uma resolução espectral específica, geralmente a mais alta disponível. Desabilita-se a seleção destes algoritmos quando a imagem "ativa" não tiver a resolução espectral adequada, sendo implementados algoritmos para converter uma imagem entre as diversas resoluções espectrais possíveis. Também esta "solução de consenso" foi adotada no sistema **PHOTOPIX**.

Diversos algoritmos para conversão da informação espectral entre imagens com diferentes resoluções espectrais foram pesquisados. Um problema particularmente difícil é a redução do número de *bits* utilizados para armazenar a informação espectral sobre cada *pixel*. São implementados no sistema **PHOTOPIX** os algoritmos de redução da informação espectral para uma palheta uniforme, o algoritmo da popularidade e o algoritmo do corte da mediana [Heck82], sendo consideradas as possibilidades de conversão para escala de tons de cinza e espalhamento do erro local.

[3] *Red, green and blue*. Em português: vermelho, verde e azul, respectivamente.

1.2 Organização do trabalho

Inicialmente, o Capítulo 2 apresenta os motivos que levaram à escolha do ambiente operacional *MS-Windows* como API utilizada pelo sistema **PHOTOPIX**. Em seguida, são apresentados conceitos necessários à compreensão da API do ambiente *MS-Windows*, sendo avaliadas algumas funcionalidades gráficas relevantes para um aplicativo de processamento de imagens.

No Capítulo 3, são apresentadas definições relativas à programação orientada para objetos. Discutem-se brevemente a sintaxe e semântica da linguagem C++, unicamente nos aspectos relativos à definição de "classes" e uso de "objetos".

O Capítulo 4 descreve a estrutura interna do sistema **PHOTOPIX**, seguida de uma apresentação de sua interface aplicativo-usuário, na versão para o ambiente *MS-Windows*.

No Capítulo 5 são apresentados os algoritmos implementados no sistema **PHOTOPIX** para converter a informação espectral entre imagens com as diversas resoluções espectrais permitidas pelo sistema.

As conclusões do trabalho são apresentadas no Capítulo 6, seguindo-se os Apêndices e a Bibliografia.

Capítulo 2

O ambiente operacional *MS-Windows*

O ambiente operacional Microsoft® Windows™ (*MS-Windows*), na sua versão 3.1 para o MS-DOS® [Dunc86], pode ser classificado como um sistema gerenciador de janelas [HDF+86, Myer88].

Os sistemas gerenciadores de janelas não têm, geralmente, a pretensão, nem a necessidade, de constituir um sistema operacional completo. Originalmente, sua única função era direcionar corretamente os fluxos de comunicação entre um usuário e diversos processos sendo executados concorrentemente por uma ou mais CPUs, locais ou remotas [Nune90, ScGe86, Tane87]. Para isto, é necessário controlar o acesso de cada processo aos dispositivos de vídeo, áudio, *mouse*, teclado, impressora, caneta óptica, etc.

O método mais usado para controle do acesso ao vídeo resulta, visualmente, na subdivisão da área do dispositivo de apresentação em uma série de "janelas" (Figura 2.1), dando origem ao nome "gerenciadores de janelas". O controle do acesso a outros dispositivos será discutido adiante.

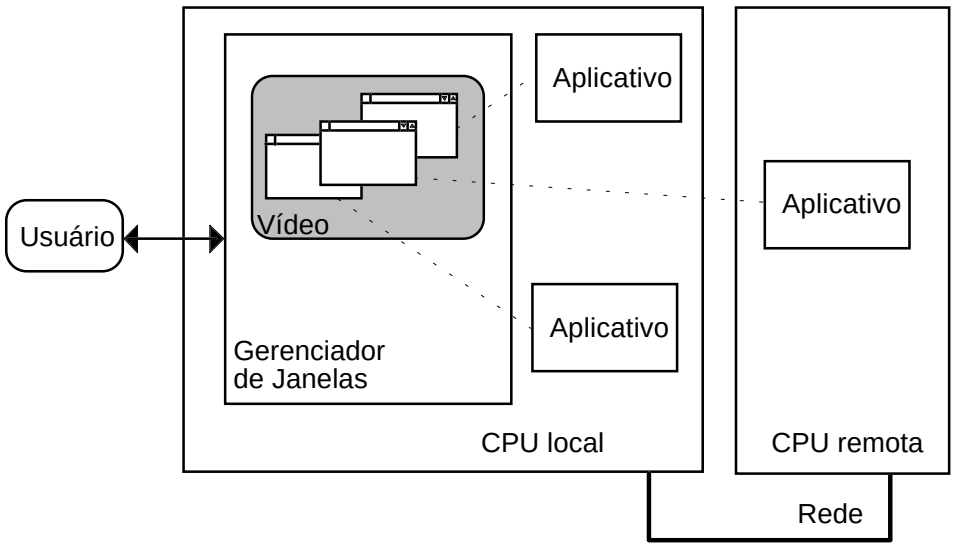

Figura 2.1 Gerenciador de janelas

Diversas classificações são possíveis para os sistemas gerenciadores de janelas. Contudo, é comum que cada taxonomia considere apenas uma das duas interfaces que todos estes sistemas devem possuir:

- a interface com o usuário, que consiste no conjunto de escolhas relativas à aparência e ao *modus operandi* do sistema e seus aplicativos [Amar92, Msft92a, Msft92b, Myer88];

- a interface com o programador, onde são detalhadas as primitivas oferecidas pelo sistema aos desenvolvedores de programas e o conjunto de regras que devem ser seguidas na codificação de um aplicativo, para garantir o funcionamente correto de todo o sistema [Chri92, CRTW88, NoYa90, Petz90, Sun90].

A comparação de gerenciadores de janelas, considerando-se principalmente a interface com o usuário, baseia-se em critérios subjetivos, não podendo nortear a escolha de uma plataforma para desenvolvimento de aplicativos. Deve-se considerar ainda o fato de que há uma evolução contínua na interface com o usuário dos gerenciadores de janelas mais populares.

A escolha do *MS-Windows* como ambiente para o sistema **PHOTOPIX** deveu-se, principalmente, aos seguinte fatores:

- interface com o programador estável e amplamente documentada;

- funcionalidades gráficas oferecidas pelo sistema;

- diversidade e disponibilidade de plataformas de *hardware* com as quais o *MS-Windows* é compatível, considerando também as possibilidade futuras [Cust92, Yao91, Yao92];

- suporte internacional embutido no ambiente operacional;

- disponibilidade e diversidade de ferramentas de desenvolvimento.

Considerou-se também a evolução da aceitação do ambiente *MS-Windows*, que é o gerenciador de janelas que conquista o maior número de novos adeptos na atualidade.

A interface com o usuário do ambiente *MS-Windows* é plenamente apresentada em [Msft92a, Msft92b]. Será apresentado aqui um conjunto de conceitos necessários à compreensão da API do sistema *MS-Windows* (*Application Programming Interface*, isto é, Interface para Programação de Aplicativos). Serão avaliadas também algumas funcionalidades gráficas da API relevantes para um aplicativo de processamento de imagens.

2.1 Processamento dirigido por mensagens

Duas abordagens são comumente adotadas para implementar o já citado controle do acesso a dispositivos de entrada e saída, que deve ser exercido pelo gerenciador de janelas em relação a seus aplicativos.

Supondo-se que o gerenciador de janelas e seus aplicativos são, individualmente, processos de um sistema operacional multitarefa, pode-se:

- deixar que cada aplicativo, no seu período de execução, atribuído pelo sistema operacional, faça chamadas a funções do gerenciador de janelas, verificando se existem entradas para aquele aplicativo ou solicitando a apresentação de informações para o usuário. O gerenciador de janelas, durante seu período de execução, comunica-se, via sistema operacional ou diretamente, com os dispositivos de entrada e saída, capturando e direcionando ao aplicativo correto as entradas do usuário e executando as operações de saída, solicitadas anteriormente pelos aplicativos. Este método de controle do acesso a periféricos é tecnicamente denominado *pull-model* [NoYa90, Sun90];

- deixar os processos relativos aos aplicativos em estado de "espera". O gerenciador de janelas é executado periodicamente, em intervalos atribuídos pelo sistema operacional, e, quando verifica que existe uma entrada para determinado aplicativo, "acorda" aquele processo, chamando uma **função de resposta a mensagens** (*callback function* [Sun90]). O aplicativo executa então uma ação adequada, que depende dos parâmetros ("mensagem") enviados à sua função de resposta, voltando depois ao estado de espera. Se, para responder a alguma mensagem, o aplicativo precisa de uma operação de entrada/saída, é chamada uma função do gerenciador de janelas, que é executada durante o período de processamento do aplicativo. Apesar disto, tal função acessa variáveis do processo gerenciador de janelas, e executa de maneira adequada para garantir a integridade do sistema, direcionando caracteres para a posição correta no vídeo, direcionando saídas para impressora a uma fila de impressão, postergando acessos a periféricos em uso, etc. Este método de controle do acesso a periféricos é tecnicamente denominado *push-model* [NoYa90, Sun90]

Resumindo, num ambiente *pull-model* os aplicativos solicitam continuamente por mensagens (em inglês *pull* significa "puxar"), enquanto num ambiente *push-model* é o gerenciador que "empurra" (em inglês *push*) as mensagens para a função de resposta do aplicativo.

O ambiente *MS-Windows*, em sua versão para o MS-DOS, que não é um sistema operacional multitarefa, deverá, obrigatoriamente, optar por uma solução do segundo tipo (*push model*), isto é, onde apenas o gerenciador de janelas executa continuamente, chamando os aplicativos quando houver necessidade. Contudo, durante a implementação das primeiras versões do ambiente *MS-Windows*, ocorreram problemas para a implementação de um sistema puramente *push-model*. Periféricos que geravam mensagens assincronamente, como o *mouse* ou o teclado, poderiam ocasionar situações de perda de "eventos", caso a função de resposta a mensagens do aplicativo em execução não fosse suficientemente rápida, algo bastante provável nas plataformas de *hardware* disponíveis à época.

Para evitar perda de eventos em plataformas de *hardware* com performance baixa, o ambiente *MS-Windows* implementa uma mistura dos dois modelos de distribuição de mensagens [NoYa90]. Há um acúmulo de mensagens geradas por dispositivos assíncronos em uma fila armazenada em memória temporária (*buffer*), sendo que, mediante solicitação, estas mensagens são entregues ao aplicativo correto, como num sistema *pull-model*. As mensagens que exigem processamento imediato são enviadas diretamente à função de resposta do aplicativo, como num sistema *push-model*.

O modelo de programação do ambiente *MS-Windows*, onde cada aplicativo responde mensagens que chegam, aparentemente, em uma ordem arbitrária, é denominado de *event-driven*, isto é, dirigido por eventos. A este modelo contrapõe-se a processamento seqüencial (*sequence-driven*), encontrado na maioria dos programas comuns, que têm seu ínicio, sequência de operações e fim rigidamente delineados no código-fonte [NoYa90].

Um aplicativo para o ambiente *MS-Windows* deverá portanto:

- possuir uma função principal, chamada no início da execução do aplicativo. A função principal "registra" uma função de resposta a mensagens junto ao gerenciador de janelas, permanecendo depois num laço de solicitação de mensagens (uma vez obtida, a mensagem é "despachada" para a função de resposta);

- possuir uma função de resposta a mensagens, que execute ação conveniente, de acordo com os parâmetros que lhe são passados (um destes parâmetros recebe o nome de "mensagem", fazendo jus à terminologia aqui empregada).

2.2 Multitarefa não-preemptiva

Ao tornar os aplicativos dependentes de mensagens para continuar sua execução, o ambiente *MS-Windows* gera, implicitamente, um chaveador de tarefas. Deverá haver um critério qualquer para, a partir de diversos "eventos", direcionados a diversos aplicativos, determinar qual aplicativo deve ser o próximo a receber uma mensagem.

O algoritmo implementado no ambiente *MS-Windows* para escalonamento das mensagens é denominado *round-robin* [KoRa88, Nune90, Tane87]. Basicamente, forma-se uma fila circular com os aplicativos, sendo que o primeiro da fila será o próximo a receber uma mensagem, se houver alguma a ele destinado, tornando-se em seguida o último da fila (Figura 2.2).

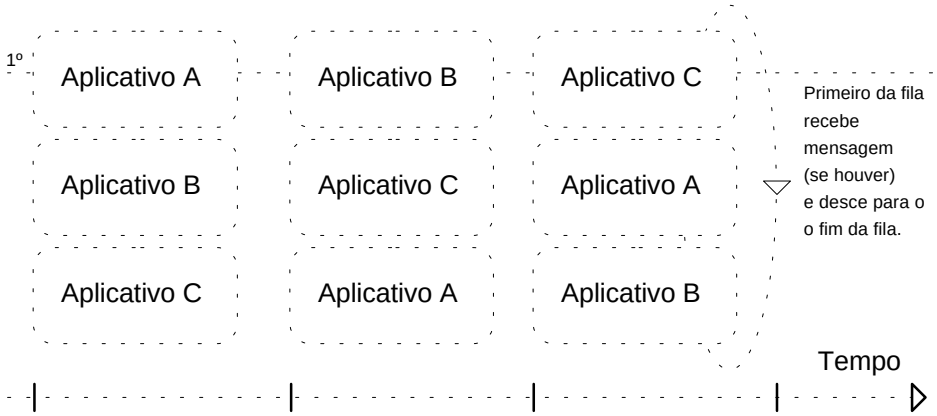

Figura 2.2 Escalonamento de mensagens pelo algoritmo *round-robin*

No ambiente *MS-Windows*, o aplicativo que recebe uma mensagem terá total controle da CPU durante todo o tempo necessário para respondê-la. Ao terminar o processamento, o aplicativo devolve o controle da CPU ao *MS-Windows*, que continua então sua tarefa de distribuição de mensagens. Este mecanismo de chaveamento de tarefas é denominado **não-preemptivo**. Em oposição a este modelo, um sistema operacional **preemptivo** interrompe um aplicativo que esteja controlando a CPU por demasiado tempo, tentando com isto preservar uma igualdade nos tempos de controle da CPU entre os diversos aplicativos em execução.

O uso de um mecanismo não-preemptivo para implementar o chaveamento de tarefas tem, como vantagens:

- a facilidade de implementação, liberando o *MS-Windows* da manutenção de diversas tabelas para guardar o "estado" atual de execução de cada aplicativo;

- o aumento do tempo útil gasto pela CPU em processamento solicitado pelo usuário. Isto ocorre pela simples inexistência do período de "troca de contexto" [Nune90, Tane87], denominação dada ao tempo gasto para interromper um processo em execução, guardar seu estado atual, restaurar o estado do próximo processo a executar e passar o controle da CPU a este processo;

- a simplificação do modelo de programação, pois desenvolvedores não terão de preocupar-se com a "disputa" pelo controle de periféricos, memória ou qualquer outro "recurso", a não ser durante situações especiais, como nos casos de comunicação entre aplicativos.

Algumas das desvantagens dos ambientes multitarefa não-preemptivos são:

- dependência em relação aos programadores para manter a aparente concorrência na execução de diversos aplicativos, pois, se um processo toma uma CPU por demasiado tempo, torna-se visível para o usuário a inatividade dos aplicativos que estão esperando para ali continuarem o seu processamento;

- inadequação de um ambiente não-preemptivo para dar suporte a aplicativos que tenham de coletar ou fornecer dados em "tempo real".

2.3 Interface para programação de aplicativos

A função de resposta a mensagens de um aplicativo desenvolvido para o ambiente *MS-Windows* pode receber centenas de tipos diferentes de mensagens. Entretanto, não se faz necessário "responder" a todos os tipos de mensagens existentes para codificar um aplicativo. Um dos maiores motivos para a crescente aceitação dos sistemas gerenciadores de janelas é a existência de um *modus operandi* comum entre os diversos aplicativos, no que diz respeito, principalmente, a procedimentos de manutenção da interface com o usuário, tais como:

- posicionamento, em cada janela, de tópicos como título do aplicativo, menus, ícones, etc.;

- redimensionamento e posicionamento das janelas destinadas aos aplicativos;

- operação dos menus e controles utilizados em cada aplicativo.

Para implementar a "resposta" a uma mensagem fornecida pelo *MS-Windows*, um aplicativo pode ter a necessidade de realizar operações que são implementadas por funções da API. Uma listagem completa das funções disponíveis para programas codificados na linguagem **C** ou **C++**, pode ser encontrada em [MSDK92].

Se determinado aplicativo não deseja modificar a resposta padronizada, atribuída pelo gerenciador de janelas a determinada mensagem, existe uma função na API que, quando chamada com os mesmos parâmetros da função de resposta a mensagens de um aplicativo, implementa a resposta padronizada para cada uma das mensagens possíveis (Figura 2.3).

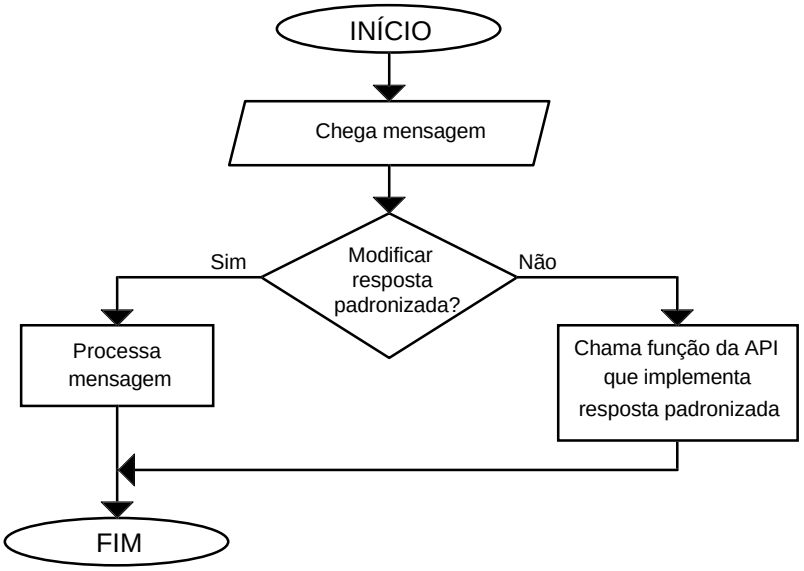

Figura 2.3 Função de resposta a mensagens de um aplicativo para o ambiente *MS-Windows*

As funções que um aplicativo para o ambiente *MS-Windows* pretende invocar em "tempo de execução" não são acopladas ao seu código-objeto diretamente. Ao invés disto, forma-se, em "tempo de compilação", uma tabela contendo as referências às funções oferecidas pela API, espalhadas pelo código-objeto. Durante o "carregamento" de um programa pelo ambiente *MS-Windows*, são "deferidas" as solicitações da tabela, transformando cada referência a um identificador de função da API numa chamada real à função. Este mecanismo, conhecido como *dynamic link* [NoYa90, Petz90], isto é, ligação dinâmica, permite que o código das funções da API seja único para os diversos aplicativos em execução no ambiente *MS-Windows*. Além disto, são possíveis atualizações no ambiente operacional sem necessidade de recompilação dos aplicativos.

2.3.1 Arquivo de recursos

Além de codificar a função de resposta a mensagens, o programador de um aplicativo para o ambiente *MS-Windows* tem, geralmente, a tarefa de elaborar um **arquivo de recursos** utilizados pelo programa. Neste arquivo, estarão devidamente relacionados os controles presentes em cada uma das diversas janelas e quadros de diálogo que compõe o aplicativo, os menus destas janelas, os *bitmaps* a serem utilizados pelo programa, etc.

A linguagem utilizada para codificação do arquivo de recursos é descrita em [MSDK92]. Usualmente, um conjunto de programas que acompanha os ambientes de desenvolvimento para o *MS-Windows* é utilizado para facilitar a etapa de elaboração do arquivo de recursos, que deve ser convertido para um formato binário pelo **compilador de recursos**, sendo em seguida ligado ao "executável" que contém a lógica do aplicativo (Figura 2.4).

Pode-se dizer que um programa para o ambiente *MS-Windows* é formado a partir de um ou mais arquivos de código-fonte, que especificam o "comportamento" do aplicativo, e por nenhum ou mais arquivos de recursos, que especificam a "aparência" do aplicativo, quando são utilizados os controles padronizados oferecidos pela interface.

Para a programação em linguagem **C/C++,** a ligação entre a descrição visual do aplicativo (arquivo de recursos) e a comportamental (código-fonte do programa) é feita através do uso de arquivos que definem identificadores, incluídos tanto no código-fonte quanto no arquivo de recursos. Um mesmo símbolo é usado então, por exemplo, para identificar um item de menu, no arquivo de recursos, e para especificar, no código-fonte, o comportamento do programa, quando receber aquele identificador como parâmetro em uma mensagem gerada pelo menu.

Outra abordagem possível para a ligação entre os aspectos visuais de um aplicativo e sua lógica funcional seria a geração automática de código. Algumas ferramentas fazem isto explicitamente [CASE92, Sun90], gerando código-fonte a partir de uma descrição da interface do aplicativo com o usuário, enquanto outras o fazem implicitamente [Msft91b, Perk92, Will87], ligando código-objeto gerado a partir de lógica implementada pelo desenvolvedor com código-objeto automaticamente gerado para lidar com os aspectos relativos à interface aplicativo-usuário.

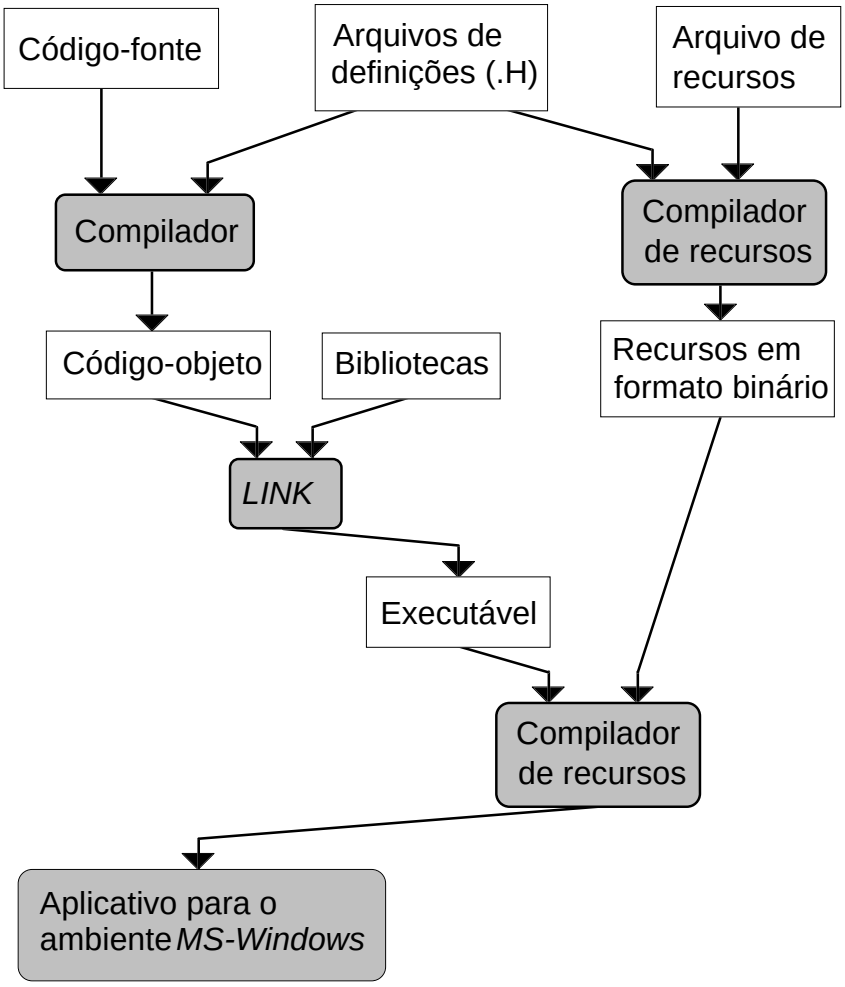

Figura 2.4 Desenvolvimento de aplicativos para o ambiente *MS-Windows*

O uso de ferramentas com geração automática de código pode acelerar em muito o desenvolvimento de um aplicativo para o ambiente *MS-Windows*. Entretanto, para implementar o sistema **PHOTOPIX** foi seguido o ciclo convencional de desenvolvimento, pois uma das funcionalidades que se pretendia oferecer ao usuário seria a troca, em "tempo de execução", da língua (Português/Inglês) utilizada nos meios de comunicação aplicativo-usuário, como menus e mensagens. Tal funcionalidade é incompatível com a maioria dos ambientes de desenvolvimento com geração automática de código.

2.3.2 Elementos da interface

O *MS-Windows*, como a maioria dos gerenciadores de janelas da atualidade, segue os princípios de projeto de interface com o usuário estabelecidos no guia CUA (*Common User Access: Advanced Interface Design Guide*) [IBM89]. A API do ambiente *MS-Windows* implementa o comportamento padronizado para diversos controles definidos na CUA, facilitando o trabalho do desenvolvedor de aplicativos.

Os controles disponíveis no ambiente *MS-Windows* diferem em poucos pontos da definição da CUA. Entretanto, alguns controles definidos na CUA não possuem suporte nativo no *MS-Windows* [Msft92b], podendo porém ser simulados pela união de controles nativos ou pela redefinição do seu comportamento, através de um mecanismo denominado *subclassing* [Petz90].

Os controles com suporte nativo no ambiente *MS-Windows*, em sua versão 3.1 para o MS-DOS, na ausência de extensões, têm a aparência e o comportamento descritos a seguir.

2.3.2.1 Janelas (*Windows*)

A janela de um aplicativo, no seu estilo mais comum, é subdividida em diversas regiões. A janela recebe mensagens geradas pelo *mouse*, ou outro dispositivo apontador, durante todo o tempo em que ele se encontra sobre a sua área, exceto no caso em que outra janela tenha capturado o *mouse*. Outras mensagens, como as geradas pelo teclado, são recebidas apenas caso a janela tenha o "foco" das entradas do sistema. Neste caso a janela é dita "ativa", sendo que a mais visível diferença entre a janela ativa e as inativas está na cor das bordas e da barra de título (Figura 2.5).

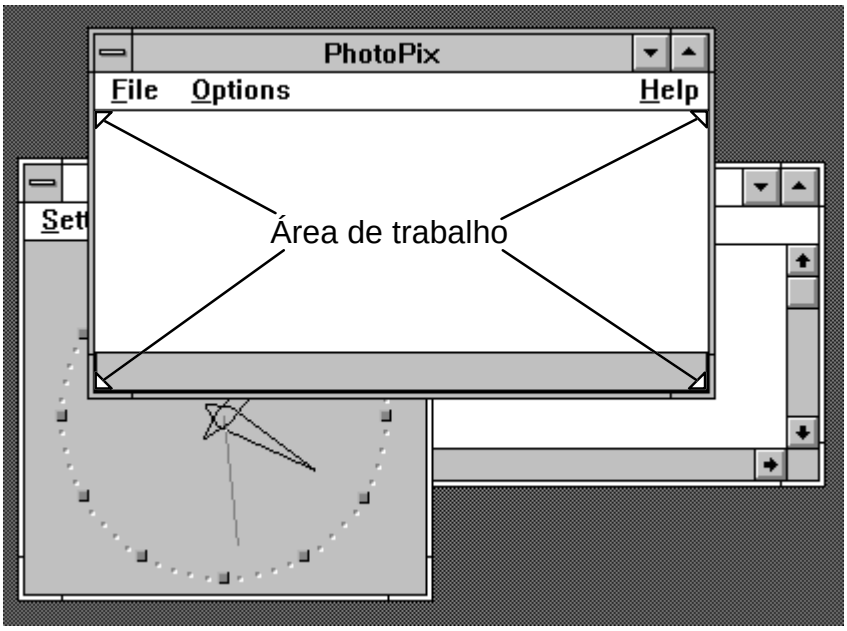

Figura 2.5 Janelas (*Windows*) no ambiente *MS-Windows*

A função da API do ambiente *MS-Windows* que implementa as respostas padronizadas para mensagens, implementa as seguintes funcionalidades para uma janela [NoYa90, Petz90, MSDK92]:

- mantém a aparência e as funcionalidades relativas ao *mouse*, como a forma do *cursor*, o teste da área sobre a qual o *cursor* encontra-se, etc.;

- gera mensagens de movimentação da janela quando o usuário pressiona o *mouse* sobre a barra de título;

- gera mensagens de redimensionamento da janela, quando o usuário pressiona o *mouse* sobre a área das bordas da janela;

- gera mensagens adequadas para operação dos ícones opcionais, embora comuns, da área da janela: o de menu do sistema, o ícone de minimização da janela e o de maximização;

- gera mensagens para manutenção e operação dos menus opcionalmente acoplados a uma janela (são comuns um menu do aplicativo e o menu do sistema);

- implementa as funcionalidades do menu do sistema, muitas das quais visam permitir a utilização do ambiente sem *mouse*;

- converte adequadamente seqüências de mensagens, correspondentes a teclas de atalho dos menus (*shortcuts* ou *accelerators* [Msft92b]) ou atalhos de *mouse* (como o duplo pressionamento em um curto intervalo de tempo sobre o menu do sistema, que gera uma mensagem para terminação do aplicativo);

- mantém a aparência da janela, redesenhando todas as áreas externas à "área de trabalho", de forma a refletir o estado da janela (ativa, inativa, maximizada, iconizada, etc.).

A terminologia "área do cliente" (*client area*) é também empregada para designar a "área de trabalho" de um aplicativo. Tal nomenclatura resulta do fato de que a comunicação dos aplicativos com o ambiente *MS-Windows* é baseada na filosofia *cliente-servidor* [ScGe86, Sun90]. Cada aplicativo é considerado como um "cliente" do gerenciador de janelas, pois solicita serviços, como o desenho de uma linha, que são efetuados na realidade pelo "servidor" *MS-Windows*.

2.3.2.2 Menus

A manutenção dos menus, especialmente em seu caso mais comum, onde todos os itens são compostos de textos, é feita automaticamente pelo *MS-Windows*. A área sobre a qual irá aparecer um menu é armazenada, para restauração posterior sem necessidade de processamento por parte do aplicativo (Figura 2.6).

As mensagens relativas ao suporte do menu, como o destaque do item selecionado, são geradas e, a menos que o aplicativo as intercepte, respondidas pela própria função da API que implementa respostas padronizadas.

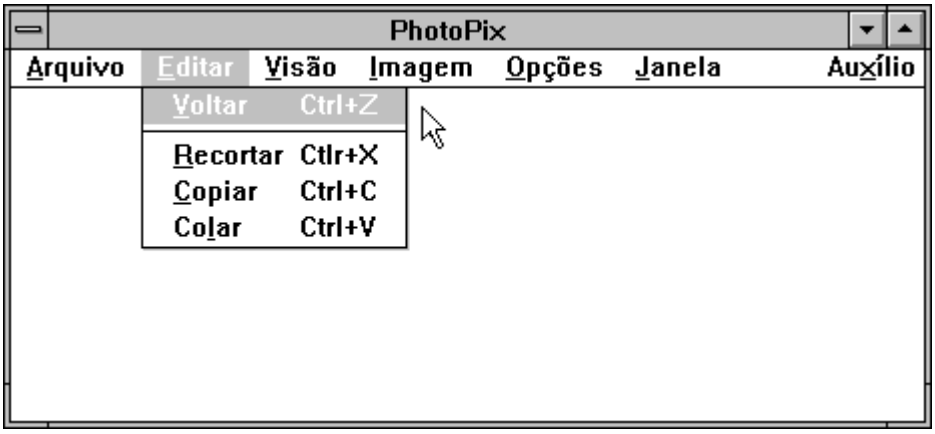

Figura 2.6 Menus no ambiente *MS-Windows*

Uma janela poderá ter um menu com itens que não são um texto. Neste caso, o menu deverá ser criado usando-se funções, em "tempo de execução", ao invés de ser definido no arquivo de recursos. Uma das funções para anexar itens a um menu possui um parâmetro que pode especificar o item como *owner-drawn*, isto é, desenhado pelo aplicativo. A função da API que implementa as respostas padronizadas enviará mensagens solicitando informações sobre o tamanho que terá tal item de menu, permitindo o dimensionamento da área a ser guardada para posterior restauração. Em seguida, será solicitado que o aplicativo desenhe o item, no estilo adequado para refletir as ações do usuário [NoYa90].

Quando o usuário finalmente escolher um item de menu, será enviada ao aplicativo uma mensagem com este significado, tendo como um dos parâmetros o identificador do item selecionado. É comum que alguns itens de menu possuam teclas de atalho (*shortcuts* ou *accelerators*) que, quando pressionadas, irão gerar a mesma mensagem que seria obtida selecionando o item de menu com o *mouse*.

2.3.2.3 *Cursor* vs. *Caret*

Deve-se distinguir a figura do *cursor*, que é controlado pelo *mouse* ou outro dispositivo apontador (*trackball*, caneta óptica, etc.), do *caret*, que é um marco indicativo do ponto que irá receber a próxima entrada do teclado (Figura 2.7).

O *caret* permanece sempre na janela que tem o foco das entradas do teclado (janela ativa). O *cursor* é justamente um dos meios de indicar qual janela o usuário deseja ver ativada.

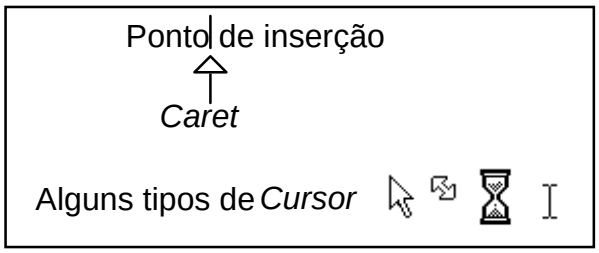

Figura 2.7 *Cursor* vs. *Caret* no ambiente *MS-Windows*

O programador pode modificar o formato tanto do *caret* quanto do *cursor*, além de poder solicitar alguns recursos pré-definidos no ambiente *MS-Windows*, como o *cursor* de espera, os de redimensionamento de janelas, etc.

Todo *cursor* possuirá um ponto, denominado *hotspot*, que será marcado, em editores especiais para este tipo de recurso, como o ponto a ser enviado junto com mensagens do *mouse*, como as de pressionamento, movimentação, etc. Este ponto é importante, principalmente, em programas de desenho, onde o usuário pode precisar de controle sobre pontos individuais de um objeto gráfico.

2.3.2.4 Quadros de diálogo (*Dialog boxes*)

Os quadros de diálogo são um meio utilizado para comunicação entre o aplicativo e o usuário de uma forma estruturada. Geralmente, cada quadro de diálogo trata de seleções relativas a uma única ação, auxiliando na manutenção de uma interface aplicativo-usuário objetiva e com boa estética.

Um quadro de diálogo usualmente contém **etiquetas** (*Static Controls*), que identificam cada um dos outros controles. Enquanto as etiquetas são elementos estáticos, os outros tipos de controles são dinâmicamente preenchidos, pressionados, selecionados ou modificados pelo usuário (Figura 2.8).

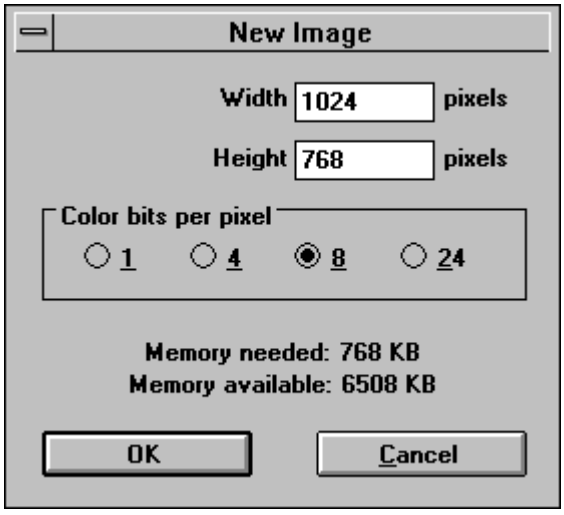

Figura 2.8 Quadro de diálogo (*Dialog box*) no ambiente *MS-Windows*

Geralmente, a única ação necessária em relação a uma etiqueta é a manutenção de valores ali apresentados ao usuário, de forma a refletir seleções feitas usando os outros controles do quadro de diálogo. Para realizar tal manutenção, tanto em etiquetas quanto em outros controles, **cada quadro de diálogo possui uma função de resposta a mensagens individual**.

Deve-se “registrar” a função de resposta a mensagens de um quadro de diálogo, da mesma forma que uma função de resposta é associada à janela principal de um aplicativo. Especificamente para os quadros de diálogo, está disponível na API uma função que fornece a resposta padronizada para mensagens que não se deseja interceptar. Teoricamente, pode-se considerar cada quadro de diálogo de um aplicativo como um programa completo. Isto permite, por exemplo, que equipes trabalhem simultaneamente em partes diferentes de um mesmo aplicativo.

Um quadro de diálogo pode ser utilizado em um aplicativo para receber informações do usuário que são vitais para continuação da execução. Um exemplo seria o quadro de diálogo de abertura de arquivo, que deve ser preenchido antes que se possa continuar e abrir o arquivo (ou talvez abandonar esta ação). Neste caso, o quadro de diálogo é denominado *modal*.

Quando um aplicativo aciona um quadro de diálogo *modal*, “todas” as mensagens recebidas são enviadas à função de resposta do quadro de diálogo. Portanto, torna-se impossível continuar trabalhando com o aplicativo. A função da API que implementa as respostas padronizadas para um quadro de diálogo emite um alerta sonoro quando detecta ações do *mouse* fora da área do quadro de diálogo ativo, mas ainda na área do aplicativo. O usuário poderá porém ativar outros aplicativos, a menos que o quadro de diálogo seja marcado como *modal* em relação a todo o sistema (*system modal dialog box*) [MSDK92].

Em sua outra forma, um quadro de diálogo é denominado *modeless*. Neste caso, o quadro de diálogo pode ser utilizado simultaneamente à janela principal ou a outros quadros de diálogo do aplicativo. Contudo, o aplicativo deve ser modificado para lidar com esta situação, pois as mensagens do usuário devem ser enviadas para a função de resposta correta. Isto é feito modificando-se o laço de solicitação de mensagens da função principal do aplicativo [NoYa90, Petz90].

Os quadros de diálogo oferecem ainda uma importante funcionalidade ao programador: o redimensionamento automático. Isto ocorre porque os valores para posicionamento e tamanho dos controles em um quadro de diálogo não são especificados em *pixels* e sim em um sistema de coordenadas especial, válido apenas para os quadros de diálogo.

Coordenadas de controles em um quadro de diálogo têm seus valores ajustados de acordo com o tamanho dos caracteres do tipo (*font*) utilizado pelo sistema *MS-Windows*: coordenadas horizontais são expressas em unidades múltiplas de 1/4 da largura média dos caracteres e coordenadas verticais são expressas em unidades de 1/8 da altura média dos caracteres. As ferramentas de auxílio ao desenvolvimento de aplicativos convertem para o sistema de coordenadas dos quadros de diálogo, automaticamente, o posicionamento e tamanho de controles colocados pelo programador nas *templates*, nome dado à especificação visual do quadro de diálogo no arquivo de recursos [MSDK92].

2.3.2.5 Botões & Caixas (*Buttons & Boxes*)

Diversos tipos de botões e caixas podem ser utilizados como controles em quadros de diálogo ou mesmo na janela principal de um aplicativo para o ambiente *MS-Windows* (Figura 2.9).

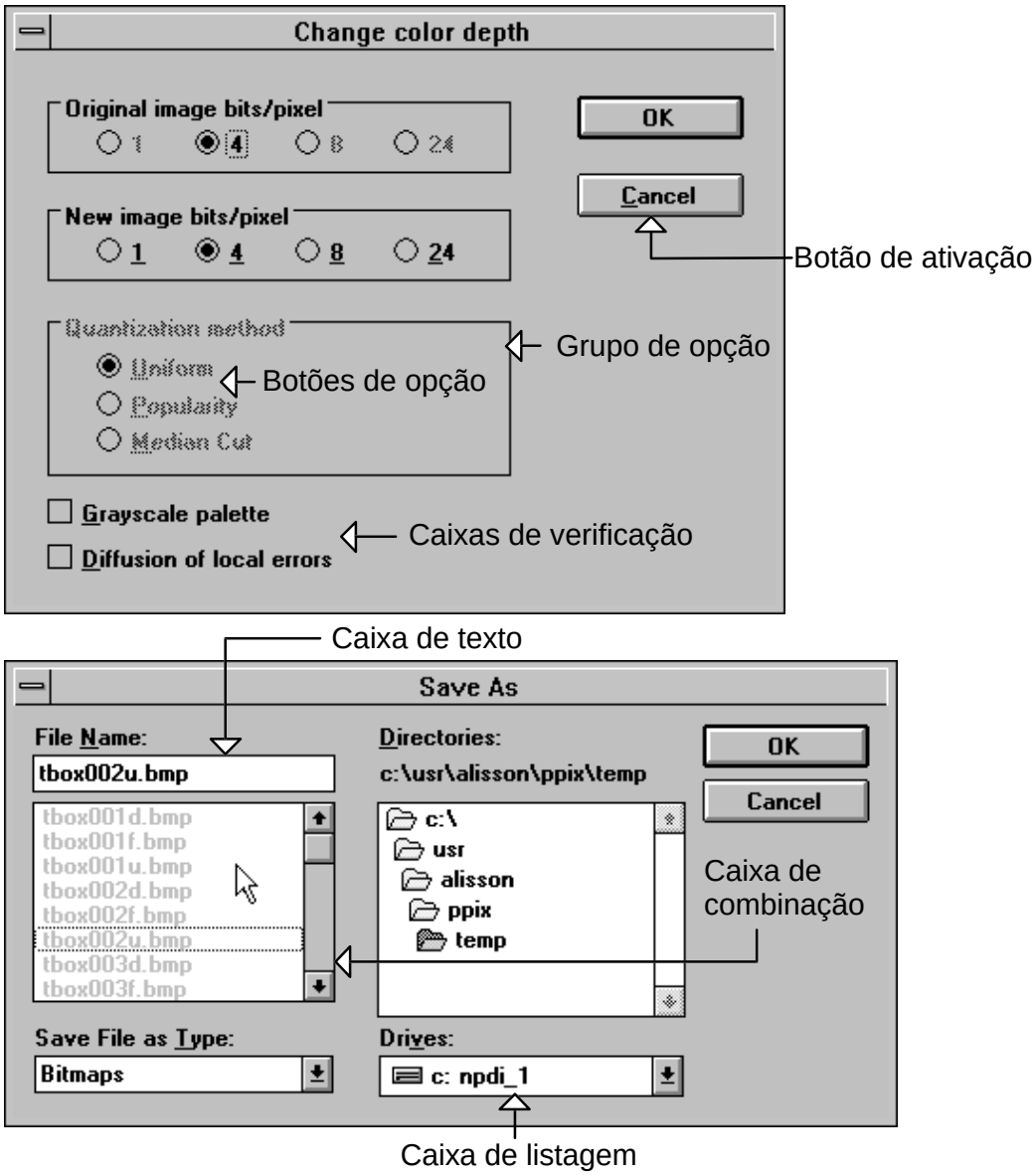

Figura 2.9 Botões & Caixas (*Buttons & Boxes*) no ambiente *MS-Windows*

Botões de ativação (*Push Buttons*) são utilizados para enviar à função de resposta um comando, do mesmo modo que um menu. Usualmente, o programador deve apenas preencher o texto e o identificador do botão em um "editor de recursos". A manutenção da aparência do botão é realizada automaticamente pelo *MS-Windows*, a menos que o programador deseje ter botões *owner-drawn* em seu aplicativo [NoYa90, Petz90].

Botões de opção (*Radio Buttons*) são utilizados para selecionar uma dentre diversas opcões exclusivas entre si. Geralmente aparecem envolvidos por um **grupo de opção** (*Group Box*), que é simplemente uma etiqueta com um quadro ao redor dos botões a serem visualmente agrupados. O programador tem um pouco mais de trabalho com os botões de opção, pois o sistema *MS-Windows* apenas envia a mensagem de comando com o identificador do botão selecionado à função de resposta. Cabe ao programador a codificação da chamada a funções da API, para passar todos os outros botões de opção do grupo ao estado de não selecionado, conforme as regras de operação da CUA.

Caixas de texto (*Edit Controls*) oferecem ao programador uma grande funcionalidade embutida no sistema gerenciador de janelas. Podem ser utilizadas em um mero campo para coleta de texto ou até para compor um completo editor para pequenos arquivos. O programador deve apenas identificar e dimensionar corretamente a caixa de texto na *template* do quadro de diálogo. Pode-se ignorar a maioria das mensagens enviadas por uma caixa de texto à função de resposta. Funções da API permitem especificar ou obter o valor atual de uma caixa de texto, transferindo-o, por exemplo, para uma váriavel do programa.

Caixas de verificação (*Check Boxes*) são utilizadas para que o usuário do aplicativo especifique se uma opção deve ou não ser ativada em um procedimento. Caso esta opção seja exclusiva em relação a qualquer uma das outras, deve-se preferir o uso de botões de opção, devidamente agrupados. A manutenção da aparência das caixas de verificação é feita pelo ambiente *MS-Windows*, automaticamente, cabendo ao programador simplesmente responder corretamente à mensagem de comando com o identificador de uma caixa de verificação. É comum que, a cada caixa de verificação de um quadro de diálogo, corresponda uma variável lógica no programa. Portanto, a "resposta" à mensagem de comando consiste apenas em ajustar o valor da variável lógica de acordo com o "estado" da caixa de verificação associada.

Caixas de listagem (*Listboxes*) são usadas para apresentar ao usuário diversas escolhas possíveis em relação a determinada opção. As escolhas podem ser representadas por textos, cores ou objetos gráficos. Uma caixa de listagem apresenta ao usuário todas as possíveis opções de escolha. Não é possível escolher um opção não constante na lista. O programador "preenche" a caixa de listagem com informações sobre cada um dos itens, durante o processamento inicial do quadro de diálogo, recebendo mensagens quando o usuário seleciona um novo item ou quando é necessário desenhar itens *owner-drawn* [NoYa90, Petz90]. Funções da API permitem obter o identificador do item selecionado atualmente, ou modificar seu valor.

Caixas de combinação (*Comboboxes*) são a união da funcionalidade de uma caixa de listagem com uma caixa de texto ou etiqueta. Sua utilidade é uma extensão daquela das caixas de listagem, permitindo que o usuário selecione um item que não aparece na lista. Um exemplo de uso de caixa de combinação seria um quadro de diálogo para obter o nome com o qual será armazenado um arquivo. Neste caso, além dos nomes dos arquivos já existentes, o usuário pode querer criar um novo arquivo, algo que seria impossível com uma mera caixa de listagem. Usa-se então a caixa de texto para digitar o nome do novo arquivo. É importante verificar que uma caixa de combinação é mais útil para o programador que a mera adição de uma caixa de texto sobre uma caixa de listagem. Diversas funcionalidades definidas na CUA, como a de que uma seleção na caixa de listagem preenche a caixa de texto, são automaticamente disponíveis para uma caixa de combinação.

2.3.2.6 Barras de rolagem (*Scroll bars*)

As barras de rolagem oferecem uma interface padronizada para o mecanismo de selecão da parte de um documento longo a ser apresentada na área de trabalho de um aplicativo. O documento pode ser um texto ou um objeto gráfico (Figura 2.10).

Cabe ao aplicativo determinar se a área de trabalho é suficiente ou não para apresentar todo o documento. No caso em que o documento pode ser apresentado na área de trabalho, não devem aparecer as barras de rolagem. Caso o documento seja longo demais em apenas uma dimensão, apenas a barra de rolagem daquela dimensão deve aparecer.

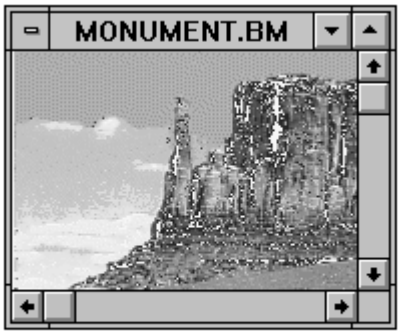

Figura 2.10 Barras de rolagem (*scroll bars*) no ambiente *MS-Windows*

Para cada uma das diversas possíveis ações do usuário sobre a área de uma barra de rolagem [Mstf92a, Msft92b], a função de resposta adequada recebe uma mensagem indicando o tipo de ação do usuário e a barra de rolagem que originou a mensagem. O programador deve interceptar as mensagens que deseja responder, codificando a ação adequada a cada situação.

É comum que os usuários desejem a existência de comandos de teclado que permitam simular operações disponíveis nas barras de rolagem. Não há suporte automático para esta rolagem de dados via teclado no ambiente *MS-Windows*, cabendo ao desenvolvedor de cada aplicativo prover este tipo de funcionalidade, de acordo com os princípios da CUA [IBM89, Msft92b].

2.3.2.7 MDI

A interface para múltiplos documentos, MDI (do inglês *Multiple Document Interface*), aumenta a funcionalidade de um aplicativo, permitindo apresentar e manipular diversos documentos em uma única instância de um aplicativo (uma instância corresponde a uma execução de um programa, que pode se dar múltiplas vezes em ambiente multitarefa, equivalendo a um processo [Nune90, Tane87]).

Um programa MDI terá pelo menos duas "classes" de janelas distintas, cada uma tendo sua função de resposta a mensagens individual. A janela principal do aplicativo, denominada *MDI frame*, cria e liga à sua área de trabalho um tipo especial de janela, denominada *MDI client window*. As janelas onde serão apresentados os documentos, *MDI child windows*, são então criadas como filhas da janela *MDI client* (Figura 2.11).

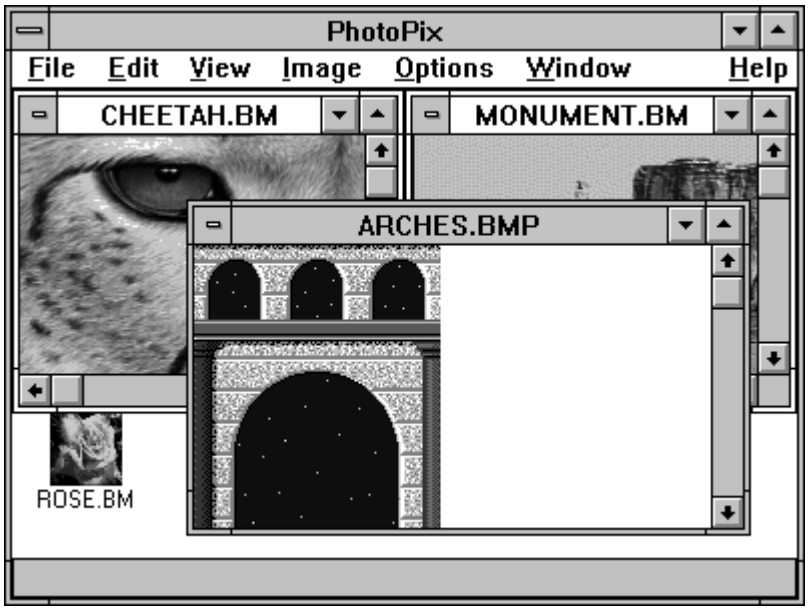

Figura 2.11 MDI (*Multiple Document Interface*)

A janela *MDI client* encarrega-se de enviar às suas filhas mensagens de dimensionamento, ativação, desativação, etc. A janela *MDI client* também anexa a um submenu (*Window*/Janela) da janela *MDI frame* uma lista das filhas. A mesma alteração necessária para o funcionamento de quadros de diálogo *modeless* deve ser feita em um programa MDI, isto é, deve-se modificar o laço de solicitação de mensagens na função principal do aplicativo, de forma que mensagens para as janelas-filhas sejam direcionadas à função de resposta correta [NoYa90, Petz90].

2.3.2.8 Funcionalidades sem aparência

Diversas funcionalidades oferecidas por um sistema gerenciador de janelas não têm aparência definida. Dizem respeito, principalmente, ao suporte para comunicação aplicativo-usuário ou ao gerenciamento da comunicação entre aplicativos.

Algumas das funcionalidades sem aparência do ambiente *MS-Windows* são [MsC+91, Msft91a, Msft92b, MSDK92]:

- recorte automático (*clipping*), permitindo que o aplicativo desenhe sua área de trabalho usando um procedimento que não se preocupa com os limites da janela, deixando ao gerenciador a tarefa de verificar se alguma ação altera a área que está realmente visível. O uso desta funcionalidade deve ser evitado, pois torna a "pintura" da área de trabalho muito ineficiente;

- suporte internacional, mediante apropriada configuração do sistema pelo usuário [Msft92a], fazendo com que seqüências de teclas resultem em apenas um código de caracter, já convertido para um alfabeto específico;

- gerenciamento da comunicação entre aplicativos para troca de dados, via protocolos que permitem apenas a transferência de dados estáticos (*clipboard*) ou protocolos para troca dinâmica de dados: DDE, OLE (*Dynamic Data Exchange, Object Linking and Embedding*);

- gerenciamento de memória independente do *hardware*, via funções da API que implementam memória "virtual" de maneira transparente para o aplicativo;

- gerenciamento da fila de impressão, que opera de maneira transparente para os aplicativos (*spool*);

- geração de tipos de caracteres (*fonts*) de diversos tamanhos em "tempo de execução" (*TrueType*), em qualquer dos dispositivos gráficos de saída;

- mecanismo de auxílio ao usuário embutido no sistema gerenciador de janelas (*help engine*), que permite a geração de auxílio *on-line* sensível ao contexto, com um mínimo de codificação. O processo de elaboração do arquivo de auxílio é separado do ciclo de desenvolvimento do aplicativo, podendo ser realizado por equipe independente;

- biblioteca de quadros de diálogo para ações comuns (*Common Dialog Boxes*), como seleção de arquivo, cor, configuração de impressão, etc.;

- possibilidade de instalação pelo usuário de extensões que aumentam a funcionalidade do sistema (como as já disponíveis *PenWindows* e *Multimedia Windows*).

2.3.3 Taxonomia das mensagens

Diversos agrupamentos são possíveis para simplificar a apresentação das mensagens existentes no ambiente *MS-Windows*. A classificação aqui apresentada, devida a [NoYa90], agrupa as mensagens em oito categorias.

2.3.3.1 *Hardware*

Tais mensagens são geradas por entradas do usuário feitas via teclado ou *mouse*. Um exemplo seria a mensagem gerada pelo pressionamento de uma tecla pelo usuário. Além do código específico para a mensagem, a função de resposta do aplicativo receberá, em um parâmetro, o código da tecla pressionada.

2.3.3.2 Manutenção da janela

Mensagens deste tipo notificam aos aplicativos mudanças de estado (ativação ou desativação do aplicativo são exemplos), requisitam ações (como a pintura da área de trabalho) ou solicitam informações (como o tamanho mínimo e máximo que a janela do aplicativo deve possuir).

2.3.3.3 Manutenção da interface com o usuário

Tais mensagens são geradas por menus (do aplicativo ou do sistema), por barras de rolagem ou outros controles colocados na janela do aplicativo (como botões, caixas, etc.) e por janelas filhas de um aplicativo MDI. São as mensagens usualmente interceptadas pela função de resposta de um aplicativo, pois correspondem à solicitação de uma ação por parte do usuário.

2.3.3.4 Terminação

Num dos dois tipos de mensagens de terminação solicita-se ao aplicativo que termine sua execução. No outro, o *MS-Windows* pergunta ao aplicativo se pode dar início a um procedimento de desativação do gerenciador de janelas (dando assim oportunidade para que sejam salvos arquivos de trabalho).

2.3.3.5 Privadas

Mensagens privadas são empregadas pelo ambiente *MS-Windows* em seus controles pré-definidos (botões, listas, caixas, etc.) ou podem ser definidas e usadas pelo desenvolvedor de aplicativos.

2.3.3.6 Notificação de recursos do sistema

Comunicam ao aplicativo mudanças ocorridas em recursos do sistema, como a cor de fundo das janelas, o horário, a configuração dos dispositivos periféricos, etc.. Geralmente, as mensagens deste tipo são enviadas diretamente à função de resposta do aplicativo (*push-model*).

2.3.3.7 Compartilhamento de dados

A troca de dados entre aplicativos pode ser realizada de diversas maneiras diferentes no ambiente *MS-Windows*, sendo que existem diversas mensagens padronizadas destinadas a iniciar e manter “conversações” entre aplicativos.

2.3.3.8 Internas do sistema

O ambiente *MS-Windows* possui, além das mensagens privadas documentadas, que podem ter sua resposta modificada pelo desenvolvedor de aplicativos, uma classe de mensagens internas, que seriam equivalentes a "mensagens privadas não-documentadas". Tais mensagens não devem ser interceptadas pelo aplicativo, pois podem ter sua funcionalidade alterada em futuras versões do ambiente operacional.

2.3.4 Interface com dispositivos gráficos

O ambiente *MS-Windows* diferencia-se pouco de outros gerenciadores de janelas na forma e nomenclatura dos mecanismos utilizados para comunicação dos aplicativos com dispositivos gráficos [NyOR90, ScGe86, Sun90].

Todo gerenciador de janelas tenta fornecer aos desenvolvedores de aplicativos um dispositivo gráfico virtual ideal. Em particular, é bastante útil não levar ao conhecimento do aplicativo as limitações de resolução espacial e espectral dos dispositivos físicos de apresentação, fazendo com que o gerenciador execute, automaticamente, algoritmos de conversão da informação dentre as diversas possíveis resoluções.

2.3.4.1 Contexto de dispositivo

As funções gráficas disponíveis na API do ambiente *MS-Windows* têm sempre como primeiro parâmetro o identificador de um tipo de dado denominado **contexto de dispositivo** (*Device Context*). Um contexto de dispositivo é uma estrutura que especifica não apenas o dispositivo físico onde deverá ser executada a primitiva gráfica mas também diversos valores para atributos que afetam as primitivas gráficas. Funções da API permitem modificar, de acordo com a necessidade, o valor de qualquer atributo contido em um contexto de dispositivo.

Sem o uso de contextos de dispositivo, uma função da API que apresentasse um texto na tela teria um número enorme de parâmetros, especificando o tipo (*font*) a usar, o tamanho dos caracteres, a cor de frente, a cor de fundo, etc. Com o uso dos contextos de dispositivo, tal função precisa de apenas cinco parâmetros: o contexto do dispositivo, a posição horizontal do texto na área de trabalho, a posição vertical, a *string* contendo o texto e um valor indicando quantos caracteres da *string* se deve apresentar.

2.3.4.2 Modos de Mapeamento

Para lidar com o problema de conversão da resolução espacial, o ambiente *MS-Windows* implementa diversos **modos de mapeamento** das coordenadas especificadas para primitivas da API. Na ausência de mudanças por parte do programador, um contexto de dispositivo tem suas unidades especificadas em *pixels*, medidos a partir do canto superior esquerdo da área de trabalho no dispositivo de apresentação. Como este sistema de coordenadas não é adequado para fornecer uma independência dos aplicativos em relação a diversos dispositivos de apresentação, existem vários outros modos de mapeamento nos quais pode ser configurado um contexto de dispositivo [NoYa90, Petz90].

Colocando-se, por exemplo, um contexto de dispositivo no modo de mapeamento `MM_LOMETRIC` (constante definida adequadamente nos arquivos de definições necessários para programas em **C**/**C++**), cada par (x,y) especificado em funções da API, para aquele contexto de dispositivo, deixa de valer um *pixel* e passa a valer 1/10.000 do metro, em qualquer dispositivo físico de apresentação. Deve-se observar que não é possível mudar o dispositivo físico associado a um contexto de dispositivo. Porém, pode-se obter um contexto de dispositivo para qualquer dispositivo físico devidamente configurado no ambiente *MS-Windows*, trocando-se em seguida o modo de mapeamento.

2.3.4.3 Conversão da informação espectral

Para converter a informação espectral especificada em uma função gráfica da API, o ambiente *MS-Windows* não implementa algoritmos que permitem obter o melhor resultado visual possível em qualquer dispositivo físico [Othm92]. Para a maioria das primitivas gráficas, a cor especificada é simplesmente aproximada para o cor mais próxima disponível no dispositivo físico, segundo a distância euclidiana no sistema RGB (ver item 5.2). Para primitivas onde será preenchida uma área, é possível solicitar o uso de *dithering* [Harr87, HeBa86].

Devido à metodologia de conversão da informação espectral não otimizada, o uso indiscriminado da abstração de utilizar 24 *bits*/*pixel*, especificando cores diretamente no sistema RGB, pode resultar em aplicativos visualmente distantes do desejado, principalmente em placas de vídeo com baixa resolução espectral e em impressoras.

2.3.4.4 *Metafiles*

Durante a execução de um aplicativo, pode ser necessário “repintar” diversas vezes a área de trabalho. Para isto, o aplicativo deve ter armazenado de alguma forma as ações necessárias para restaurar a aparência de sua área de trabalho. Uma maneira simples e prática de implementar esta funcionalidade são os *metafiles* [MSDK92, Petz90]. Um *metafile* é simplemente um arquivo contendo a descrição de primitivas gráficas solicitadas por um aplicativo.

Inicialmente, no processo de criação de um *metafile*, o aplicativo recebe um contexto de dispositivo. Na realidade, o contexto de dispositivo retornado pela função de abertura de um *metafile* é falso, não estando ligado a nenhum dispositivo físico real. Contudo, este contexto de dispositivo pode ser utilizado como parâmetro de qualquer primitiva gráfica da API, resultando no armazenamento do código da função e do valor dos parâmetros em um arquivo, num "formato interno" do ambiente *MS-Windows*. Ao fechar o *metafile*, o aplicativo recebe como retorno um *handle*, nome dado no ambiente *MS-Windows* a identificadores de recursos do sistema.

Quando for necessário que o aplicativo execute a seqüência de ações armazenadas no *metafile*, há uma função na API (de nome `PlayMetaFile`, para programas em **C**/**C++**) que tem como parâmetros um contexto de dispositivo, onde serão executados os comandos armazenados no *metafile*, e um *handle*, identificando qual *metafile* executar.

No *metafile*, os comandos da API são codificados de uma maneira que torna a execução do *metafile* idêntica à execução da seqüência de comandos armazenados, tanto em resultado visual quanto em "tempo de execução". Isto permite criar uma real independência de dispositivos, ao custo de uma baixa performance.

2.3.4.5 *Bitmaps*

Uma maneira de acelerar a performance do processo de pintura da área de trabalho de um aplicativo é guardar a área de trabalho em um *bitmap* [MSDK92, Petz90]. Um *bitmap* é simplesmente uma área em memória armazenando os mesmos *bytes* que conteria a memória do dispositivo físico associado, como resultado das primitivas gráficas solicitadas pelo aplicativo.

Para usar esta funcionalidade, o aplicativo deve criar um contexto de dispositivo associado à memória que seja "compatível" com o dispositivo de vídeo (isto é, que armazene a informação sobre cada *pixel* da mesma forma, tanto em número de *bits*/*pixel* quanto em relação à organização dos *bytes* de dados na memória; existem funções específicas para isto na API). Em seguida, cria-se um *bitmap* compatível com este contexto de dispositivo, tendo dimensões adequadas para guardar a área de trabalho do aplicativo. Através de uma função da API, "seleciona-se" o *bitmap* no contexto de dispositivo. Esta ação equivale a dizer que toda primitiva da API, solicitada naquele contexto de dispositivo, "escreve" na memória do *bitmap*.

Quando for necessário repintar sua área de trabalho, o aplicativo simplesmente usa uma função da API que transfere informação sobre *pixels* do contexto de dispositivo associado ao *bitmap* para o contexto de dispositivo do vídeo. Como os dois contextos de dispositivo são compatíveis, isto equivale a uma transferência de *bits* entre a memória onde está armazenado o *bitmap* e a memória da placa de vídeo (tal transferência de blocos de *bits* é denominada *bit blit*, do inglês *bit-block transfer)*.

As operações de *bit blit* são bastante rápidas, porém o uso de *bitmaps* como meio de armazenamento da área de trabalho resulta na perda da independência de dispositivos. Se a placa de vídeo possui apenas 1 *bit*/*pixel* de informação espectral, o *bitmap*, compatível com tal contexto de dispositivo, conterá também apenas 1 *bit*/*pixel*. A informação espectral é perdida no momento em que é executada uma primitiva gráfica da API no contexto de dispositivo associado ao *bitmap*.

2.3.4.6 DIBs

A partir da versão 3.0 do ambiente *MS-Windows*, foi colocado à disposição dos desenvolvedores de aplicativos um novo tipo de funcionalidade: os DIBs [MSDK92, Petz90]. DIBs, do inglês *Device Independent Bitmap*, são *bitmaps* que têm resolução espectral especificada pelo aplicativo, e não pelo contexto de dispositivo a eles associado. Apenas um contexto de dispositivo especial pode ser associado aos DIBs, permitindo a execução de primitivas gráficas da API sobre sua área.

A transferência da informação de um DIB para o vídeo durante a "repintura" da área de trabalho sofre, atualmente, de graves problemas de performance. Um DIB que tenha, por exemplo, 24 *bits*/*pixel* e deva ser transferido para uma placa de 8 *bits*/*pixel* irá gerar, para cada *pixel* a ser transferido, uma busca em todas as 256 posições da palheta-destino pela cor mais próxima da cor original do *pixel* no DIB. Em algumas máquinas, a transferência de um DIB de tamanho médio (512*512), da memória para a placa de vídeo, pode levar minutos.

Para manter uma performance aceitável da interface com o usuário, o aplicativo que usa DIBs deverá também manter um *bitmap* dependente do dispositivo, espelhando o conteúdo do DIB. Este *bitmap* é que será transferido ao vídeo durante a repintura da área de trabalho. Com o contínuo aumento da velocidade de processamente das CPUs e o aparecimento de placas de vídeo com coprocessadores dedicados, espera-se que, brevemente, tal duplicação da informação não seja mais necessária [MDTG92a, MDTG92b, MDTG92c].

2.4 Conclusões

O ambiente operacional *MS-Windows* é um gerenciador de janelas, que comunica-se com seus aplicativos através de mensagens. Um aplicativo para o ambiente *MS-Windows* tem sua lógica implementada em uma função de resposta a mensagens, que é acionada sincronamente (*pull-model*) ou, em situações especiais, assincronamente (*push-model*).

A distribuição das mensagens é feita pelo algoritmo *round-robin*, sendo que, após receber uma mensagem, o aplicativo não tem limite de tempo para fornecer sua resposta. Devido a isto, o ambiente *MS-Windows* é classificado como não-preemptivo.

Usualmente, durante o desenvolvimento de um aplicativo, é elaborado um arquivo de recursos, que relaciona controles, menus e outros elementos que irão compor as janelas e quadros de diálogo do programa. Diversas funcionalidades sem aparência também podem ser utilizadas pelos desenvolvedores de aplicativos, como o suporte internacional, gerenciamento de memória "virtual", gerenciamento da fila de impressão, etc.

Os aplicativos não se comunicam diretamente com os dispositivos gráficos conectados à plataforma de *hardware* em que o ambiente *MS-Windows* executa. As primitivas gráficas são solicitadas pelo aplicativo e executadas pelo *MS-Windows*, seguindo-se os princípios da arquitetura cliente-servidor.

Toda primitiva gráfica da API do ambiente *MS-Windows* tem, como primeiro parâmetro, um contexto de dispositivo, que é uma estrutura de dados especificando não apenas o dispositivo físico onde deve ser executada a função, mas também atributos que podem alterar sua execução, como o modo de mapeamento, origem da área de trabalho, cor corrente, etc. Transformações na resolução espacial e espectral dos parâmetros fornecidos a primitivas gráficas da API são efetuadas adequadamente, porém nem sempre por algoritmos que podem produzir o melhor resultado visual.

Como a pintura da área de trabalho na janela de um aplicativo pode ser solicitada assincronamente, são oferecidas pela API algumas funcionalidades para facilitar o armazenamento adequado do contéudo daquela área. *Metafiles* armazenam seqüências de chamadas a primitivas gráficas, *bitmaps* armazenam uma cópia do conteúdo da memória do dispositivo físico com o qual são compatíveis, enquanto DIBs são *bitmaps* com resolução espectral especificada pelo aplicativo.

O ambiente operacional *MS-Windows* fornece uma interface para programação de aplicativos que permite a codificação de programas gráficos sem qualquer referência à configuração específica da plataforma de *hardware* em uso, sendo que as primitivas gráficas oferecidas pela API do ambiente *MS-Windows* são bastante similares às disponíveis em outros gerenciadores de janelas [Sun90].

Capítulo 3

Programação orientada para objetos

O paradigma da programação orientada para objetos é a adição à linguagem em uso de novos tipos de dados, pelo programador de aplicativos. Numa linguagem orientada para objetos, o desenvolvedor de um sistema de processamento de imagens deve ter possibilidade de definir um tipo de dados "Imagem", declarar duas variáveis daquele tipo e usar funções como a adição ou a subtração de imagens numa sintaxe idêntica à utilizada para os tipos de dados pré-definidos.

Embora a idéia básica seja bastante simples, diversas questões são problemáticas em relação à implementação tanto de uma linguagem como de sistemas orientados para objetos:

- como definir os novos tipos de dados básicos?

- em que linguagem implementar o sistema?

- quais as limitações e problemas decorrentes da linguagem escolhida?

- quais tipos básicos de dados podem ser aproveitados de "bibliotecas" desenvolvidas por empresas especializadas?

- como o processamento dirigido por mensagens (*event-driven*) de ambientes como o *MS-Windows* é tratado na linguagem de programação escolhida?

Neste capítulo, são apresentadas definições relativas à programação orientada para objetos, necessárias para a correta compreensão da estrutura interna do sistema **PHOTOPIX**, a ser detalhada no capítulo seguinte.

3.1 Princípios da programação orientada para objetos

São três as principais diferenças de uma linguagem orientada para objetos em relação às outras linguagens de programação: a maior capacidade de "encapsulamento" de detalhes no código-fonte, o "polimorfismo" das operações e a possibilidade de "herança" de funcionalidades de tipos pré-definidos [AnAn91, Chri92, DeSt89].

Na terminologia da programação orientada para objetos, os tipos de dados, tanto básicos como definidos pelo programador de aplicativos, constituem "classes". A declaração de uma variável de determinada classe gera o aparecimento de um "objeto" (na linguagem Pascal orientada para objetos, o que aqui é denomidado classe recebe o nome de objeto, sendo uma variável de determinado tipo denominada "instância").

Por **encapsulamento** entende-se a possibilidade de adicionar novas classes a uma linguagem sem tornar públicos detalhes da implementação. De certa forma, o encapsulamento já está presente em quase todas as linguagens de programação da atualidade. Diversos tipos básicos de dados, como números inteiros e reais, são disponíveis em quase todas as linguagens, sem que o programador saiba como são armazenados seus valores internamente, ou como são implementadas as operações disponíveis nestas classes pré-definidas.

O **polimorfismo**, que significa literalmente "muitas formas", é uma característica que expressa a capacidade de uma mesma operação ter seu comportamento modificado, de acordo com a(s) classe(s) do(s) objeto(s) sobre o(s) qual(is) irá atuar. Um certo grau de polimorfismo também já está presente na maioria das linguagens da atualidade. É comum que a adição seja sempre representada por um único símbolo, usualmente o "**+**", independente da classe dos números a serem adicionados. Assim, a expressão "**x + y**" pode estar representando a soma de dois números inteiros, ou dois números reais e, em algumas linguagens, a soma de um inteiro a um real. É mais claro ainda o polimorfismo no caso do operador de atribuição, na maioria das linguagens denotado pelo símbolo "**=**". Na expressão "**a = b**", pode-se estar atribuindo um valor a uma variável inteira, real, lógica, uma *string*, etc.

Neste caso de polimorfismo, a escolha dentre todas as diferentes formas de um operador é feita em "tempo de compilação". A declaração das variáveis permite ao compilador determinar a classe correta que implementa o operador que atua sobre os objetos em uma expressão.

Através do mecanismo de **herança**, torna-se possível criar novos tipos de objetos aproveitando as classes já existentes, pela adição de dados ou funções, ou a modificação do comportamento das funções já definidas. A classe original, denominada "classe base", é especializada na "classe derivada". A relação entre as classes deve ser estabelecida de forma que objetos da classe derivada sejam casos particulares da classe base, e não partes dela.

Numa hierarquia em que a classe base é um polígono, seria correto ter triângulos e retângulos como classes derivadas. Contudo, não seria correto "derivar" uma classe reta, pois, embora seja parte do polígono, uma reta não é um polígono. Sabendo-se que um objeto é um polígono, seria possível definir uma função para "preencher" os objetos daquela classe. Uma maneira simples de verificar que a classe reta estaria erradamente derivada da classe polígono é o fato de que não é possível preencher uma reta.

O polimorfismo, no contexto da programação orientada para objetos, deve ser válido também para hierarquias de classes, que formam uma chamada "família de classes". No exemplo citado, deve ser possível definir funções `Preencher()` especializadas para cada uma das classes que tem por base a classe dos polígonos. Dado um objeto da família dos polígonos, deve ser possível invocar a sua função `Preencher()` específica em tempo de execução, mesmo não se sabendo durante a compilação qual será o caso particular de polígono a ser tratado.

Claramente, a implementação de uma linguagem orientada para objetos exige a definição de diversas regras de sintaxe e semântica. Uma complicação adicional aparece quando se deseja que a tecnologia de orientação para objetos seja adicionada a uma linguagem já existente. Apesar disto, as mais populares linguagens orientadas para objetos da atualidade tiveram como "base" linguagens já existentes.

Uma linguagem de programação deve auxiliar no projeto, implementação, extensão, correção e otimização de um sistema [Joyn92].

Uma linguagem orientada para objetos auxilia no projeto de um sistema, pois direciona o projetista para a identificação de classes, que podem então ser implementadas reproduzindo fielmente o especificado. A extensão da funcionalidade das classes de um sistema também é facilitada, pelo encapsulamento dos detalhes e pela possibilidade de herança de funcionalidades. Em um sistema orientado para objetos, corretamente definido, espera-se que a correção de erros seja bem localizada, não perturbando todo o código-fonte. Por fim, uma linguagem orientada para objetos auxilia na optimização do sistema, facilitando a troca de segmentos de código ineficientes por outros optimizados e permitindo a especialização de funções, com o uso do polimorfismo.

3.2 A linguagem C++

A linguagem **C++** é, atualmente, a mais utilizada na codificação de sistemas orientados para objetos. Em grande parte, sua aceitação decorre da popularidade da sua linguagem base: a linguagem **C**. Parafraseando-se a terminologia da programação orientada para objetos, pode-se dizer que a linguagem **C++** "herda" todas as características da linguagem **C**, adicionando mais algumas.

É longa uma discussão dos detalhes de sintaxe e semântica da linguagem **C++** [AnAn91, Chri92, DeSt89]. É grande também o número de críticas possíveis à forma como a linguagem **C++** implementa os princípios da programação orientada para objetos [Joyn92]. Apesar disto, a diversidade e a qualidade das ferramentas de programação atualmente disponíveis para a implementação de programas em **C++** supera o existente para todas as outras linguagens orientadas para objetos.

A correta compreensão da sintaxe e semântica da linguagem **C++** depende muito do conhecimento prévio destes temas em relação à linguagem **C**. Uma extensa análise da linguagem **C**, objetivando principalmente a portabilidade de código-fonte, pode ser encontrada em [RaSc90].

Assume-se o conhecimento da linguagem **C** na discussão que se segue, onde serão apresentadas as regras de sintaxe e semântica relativas à implementação de "classes" e uso de "objetos" na linguagem **C++**. Uma análise mais extensa da linguagem **C++** pode ser encontrada em [AnAn91, Chri92, DeSt89, Joyn92, MsC+91, Stev90].

3.2.1 Classes e o encapsulamento

Uma possível implementação para uma classe de polígonos, na linguagem C++, é apresentada na Figura 3.1.

```
// POLÍGONO.H: Declaração da classe POLÍGONO

class POLÍGONO
{
private:
         UINT  NúmeroDeLados;   // Unsigned integer
         POINT Vértices[];      // Aloca dinamicamente
public:
         POLÍGONO(UINT nLados); // Construtor
         {
             NúmeroDeLados = nLados;
             // Aloca memória para Vértices[]
         }

         ~POLÍGONO();           // Destrutor
         {
             // Libera memória usada para Vértices[]
         }

         void Desenhe();
         void FaçaVértice(UINT índice, POINT point);
};

// POLÍGONO.CPP: Implementação das funcionalidades

#include "POLÍGONO.H"

void POLÍGONO::Desenhe()
{
    // Código para desenhar o POLÍGONO
}

void POLÍGONO::FaçaVértice(UINT índice, POINT point)
{
    if (índice < NúmeroDeLados)
       Vértices[índice] = point;
}
```

Figura 3.1 Declaração e implementação de uma classe na linguagem C++

A palavra reservada `class` inicia a declaração de uma classe, cujo nome vem a seguir. As funções e variáveis pertencentes a uma classe são então declaradas, em seções rotuladas pelos atributos `public` ou `private`.

Um membro (função ou variável) de uma classe especificado sob uma seção `public` pode ser acessado em qualquer ponto de um programa. Um membro especificado como `private` só pode ser acessado dentro de funções da própria classe (ou em classes e funções marcadas com o atributo `friend` [DeSt89]). Existe um terceito atributo de acesso, denominado `protected`, que segue as mesmas regras de um membro `private`, exceto pelo fato de que membros `protected` podem ser acessados em funções pertencentes a classes derivadas (ver item 3.2.2).

O operador de "escopo", para o qual utiliza-se o símbolo "`::`", é de vital importância na linguagem C++. É ele que permite separar a declaração de uma classe do código onde estão implementadas suas funcionalidades, permitindo assim o encapsulamento de detalhes (embora, por outro lado, esta separação gere mais trabalho para o programador, que tem de manter a consistência entre os dois arquivos [Joyn92]). A sintaxe `POLÍGONO::Desenhe()` identifica a implementação da função `Desenhe()` especificamente para a classe `POLÍGONO`.

Uma função definida com o mesmo nome da classe a qual pertence é denominada um "construtor" para aquela classe. Usualmente, esta função implementa procedimentos iniciais necessários à subseqüente utilização dos objetos da classe. Um construtor não pode retornar nenhum valor, assim como não o pode fazer o "destrutor" de uma classe, declarado em uma função com o símbolo "**~**" antes do nome da classe. Ao contrário dos construtores, que podem ser especializados variando-se os parâmetros da função (ver item 3.2.3), só pode existir um destrutor para cada classe, o qual não tem argumentos. O destrutor é geralmente utilizado para "liberar" memória alocada dinâmicamente no construtor de uma classe [Chri92].

A implementação de uma função pode também ser feita imediatamente após sua declaração (assim são implementados o construtor `POLÍGONO()` e o destrutor `~POLÍGONO()` da classe `POLÍGONO`, na Figura 3.1).

A sintaxe para acesso aos membros de uma classe na linguagem **C++** é semelhante à usada na linguagem **C** (e, conseqüentemente, também em **C++**) para acesso aos membros de uma estrutura [RaSc90, DeSt89]. Um exemplo de código que utiliza a classe `POLÍGONO` é apresentado na Figura 3.2.

```
// Uso de objetos da classe POLÍGONO

#include "POLÍGONO.H"

POLÍGONO Poli3lados(3);          // Variável comum
POLÍGONO * pPoli4lados = new POLÍGONO(4); // No "heap"
POLÍGONO * pPoli;                         // Ponteiro
POINT    point;

point.x = 5.0; point.y = 5.0;
Poli3lados.FaçaVértice( 0, point );
pPoli4lados->FaçaVértice( 0, point );
// ... Coloca valores válidos nos outros vértices

// Tenta colocar valor em índice inválido
Poli4lados.FaçaVértice( 99, point );  // OK: ignora

pPoli = pPoli4lados;
pPoli->Desenhe();    // Desenha QUADRADO

pPoli = &Poli3lados; // pPoli = endereço de Poli3lados
pPoli->Desenhe();    // Desenha TRIÂNGULO

delete pPoli4lados;  // Libera variável no "heap"

Poli3lados.NúmeroDeLados = 3; // ERRO de compilação:
                              //   membro "private"
```

Figura 3.2 Declaração e uso de objetos na linguagem C++

A inclusão do arquivo `POLÍGONO.H` em qualquer arquivo de código-fonte torna possível a definição de objetos da classe `POLÍGONO`. O compilador se encarrega automaticamente de verificar se os acessos a membros da classe são feitos corretamente.

Os operadores `new` e `delete` são definidos na linguagem C++ para criar e destruir objetos dinamicamente. O operador `new` "aloca" memória para receber os dados de um objeto, chamando em seguida o construtor apropriado. O operador `delete` chama o destrutor da classe, liberando em seguida a memória utilizada por um objeto [Chri92].

Devido à separação entre a definição da classe e sua implementação, o programador deve "ligar" (*Link*) o arquivo-objeto gerado a partir da compilação do código-fonte que usa a classe `POLÍGONO` com o código-objeto do arquivo que implementa a classe `POLÍGONO` (arquivo `POLÍGONO.OBJ`, gerado a partir da compilação do arquivo `POLÍGONO.CPP`). Na maioria dos compiladores da atualidade, este tipo de tarefa é automatizada com o uso de "projetos" [MsC+91].

3.2.2 O mecanismo de herança na linguagem C++

O programador de aplicativos pode detectar que casos particulares de determinada classe, usados muito frequentemente, poderiam merecer uma implementação mais abstrata ou otimizada.

No caso da já definida classe de polígonos, um programador de aplicativo pode detectar a ocorrência da muitos objetos que são polígonos de apenas três lados. Neste caso, ao invés de repetir continuamente a declaração de polígonos de três lados, o programador pode decidir-se por derivar da classe de polígonos uma classe específica para os triângulos.

A sintaxe da linguagem **C++** para o mecanismo de "derivação" é apresentada na Figura 3.3.

```
// TRIANG.H: Definição da classe TRIÂNGULO

#include "POLÍGONO.H"

class TRIÂNGULO : public POLÍGONO
{
public:  TRIÂNGULO()   // Construtor
         : POLÍGONO(3)
         {
         }
};

// Uso da classe TRIÂNGULO

#include "TRIANG.H"

TRIÂNGULO triang;

triang.FaçaVértice(0, point1);
triang.FaçaVértice(1, point2);
triang.FaçaVértice(2, point3);

triang.Desenhe();
```

Figura 3.3 Mecanismo de herança na linguagem C++

Após o nome da nova classe, o programador deve colocar o sinal "**:**" seguido de uma ou mais classes que servirão como base para a nova definição, cada qual rotulada pelos atributos `public` ou `private` (o mecanismo de derivação tendo por base mais de uma classe é denominado "herança múltipla" [DeSt89]).

Na classe derivada permanecem disponíveis todos os membros (funções e variáveis) públicos de uma classe base rotulada como `public`. Os membros `private` da classe base não são acessíveis para a classe derivada. É exatamente para resolver o problema que aparece quando se deseja que um membro da classe base não seja `public`, porém seja acessível para uma classe derivada, que existe o atributo `protected`. Portanto, membros `protected` de uma classe base são como membros `public` para as classes descendentes (e `friends` [DeSt89]) e como membros `private` para o resto do programa.

O uso de classes base rotuladas como `protected` e `private`, embora possível e plenamente especificado na linguagem **C++** [Chri92, DeSt89], é incomum.

Quando é necessário que o construtor de uma classe derivada passe argumentos a um dos contrutores da classe base, coloca-se o sinal "`:`" após a definição do contrutor da classe derivada, vindo em seguida o construtor da classe base, com os parâmetros adequados.

3.2.3 Sobrecarga de funções e o polimorfismo

Um dos grandes problemas para a programação de grandes sistemas é manter atualizada e organizada a documentação de funções que estão sendo desenvolvidas por equipes independentes.

Tomando como exemplo a API do ambiente *MS-Windows*, existem atualmente mais de mil funções disponíveis para o programador de aplicativos. Embora um índice resolva o problema de localizar na documentação os parâmetros para cada uma das funções da API, não resolve o problema de localizar a função da API que implementa uma funcionalidade específica desejada pelo programador.

Se o programador usa uma linguagem não orientada para objetos (como a linguagem **C**), a metodologia adotada para dar nomes às funções da API do sistema *MS-Windows* auxilia na localização de funcionalidades específicas, ao custo do uso de nomes prolixos para algumas funções. Na API do *MS-Windows* para a linguagem **C**, existem dezenas de funções para criação de "objetos" da interface gráfica[4] (`CreateCaret`, `CreateCursor`, `CreateIcon`, `CreateDialog`, `CreateDialogIndirect`, etc.).

[4] Deve-se distinguir os dois casos onde é usada a palavra "objetos": os "objetos" da interface gráfica são armazenados pelo *MS-Windows* internamente, de forma independente da linguagem que os criou, enquanto os "objetos" da programação orientada para objetos são variáveis de determinada classe.

Na linguagem **C++**, a "assinatura" de uma função é composta não apenas pelo seu nome, mas também pelo tipo de seus parâmetros. Assim, duas funções com mesmo nome e parâmetros de tipos diferentes podem ser implementadas em uma classe, para lidar com casos particulares de determinada funcionalidade. Esta forma de polimorfismo na linguagem C++ é denominada sobrecarga de funções (*function overloading* [MsC+91]).

Para escolher entre as diversas funções de mesmo nome definidas para uma classe, o compilador observa o tipo dos parâmetros usados pelo programador em uma chamada específica (*argument matching* [MsC+91]). Caso ocorra uma perfeita identidade entre tipos das variáveis da chamada e os tipos de uma das versões da função, aquela versão é utilizada. Senão, tenta-se "promover" os parâmetros, segundo as conversões automáticas possíveis (`int` para `long`, `float` para `double`, etc.), até que se obtenha uma possível igualdade no tipo de todos os parâmetros. O compilador exibe uma mensagem de erro apenas no caso onde o uso de conversões não permite obter o tipo correto dos parâmetros, para nenhuma das versões de uma função, ou quando o número mínimo de conversões necessárias é igual para mais de uma das versões.

Um exemplo dos benefícios trazidos pelo polimorfismo pode ser dado pela função de carregamento de menus no ambiente *MS-Windows*. No "arquivo de recursos" (ver item 2.3.1) de um aplicativo, um menu pode ser identificado por um nome (*string*) ou por um número (inteiro). Para programadores na linguagem **C**, só existe na API uma função para carregamento de menus, que aceita como parâmetro uma *string* de caracteres. Logo, o programador deve converter um identificador numérico de um menu para uma *string*, usando uma macro pré-definida que, por ter múltiplas utilidades, leva um nome não muito intuitivo (`MAKEINTRESOURCE`).

O polimorfismo evita este tipo de complicação na linguagem **C++**. Usando a biblioteca MFC (ver item 3.3.1), duas funções estão disponíveis para o carregamento de menus:

- `BOOL LoadMenu(char * stringName)`
- `BOOL LoadMenu(UINT UnsignedIntegerID)`

As duas funções retornam um valor lógico indicando o sucesso ou não de sua execução, sendo que uma tem como parâmetro uma *string* e a outra um inteiro sem sinal. Assim, o programador precisa lembrar-se de apenas um nome (bastante intuitivo) para a função que implementa a operação de carregamento de menu.

Outra forma de polimorfismo disponível na linguagem **C++** é a implementação das "funções virtuais". Funções que, na classe base, foram rotuladas com o atributo `virtual` podem ser redefinidas com mesmo número e tipo dos parâmetros em classes derivadas[5]. Obviamente, o projetista do sistema deve prever a possibilidade de redefinição de uma função durante o projeto da classe base.

A funções virtuais redefinidas em classes derivadas serão invocadas adequadamente via ponteiros para a classe base, sem necessidade de "cast" [DeSt90]. Por um mecanismo da implementação denominado "tabelas virtuais" (*virtual tables* [MsC+91]), o compilador será capaz de decidir, em "tempo de execução", qual versão de uma função virtual está sendo invocada, observando a verdadeira classe do objeto identificado por um ponteiro.

O polimorfismo utilizando funções virtuais tem sido muito utilizado no ramo da Computação Gráfica [Stev90]. Um meio simples para que um programador implemente de maneira bastante modular um aplicativo que desenhe diversos objetos em uma "cena" é delineado na Figura 3.4.

Define-se uma classe básica onde está disponível uma função virtual para desenho do objeto. A função `Desenhe()` da classe `OBJETO` da Figura 3.4 usa uma sintaxe especial para definição das chamadas "funções virtuais puras" [Chri92, DeSt90].

As funções virtuais puras não são implementadas na classe base, apenas nas classes derivadas. Por este motivo, o compilador aponta como erro a definição de um objeto da classe base. Apenas objetos das classes derivadas podem ser criados. É possível porém definir "ponteiros" para objetos da classe base, como o que é feito na Figura 3.4. Tais ponteiros podem também apontar para objetos criados usando como protótipo classes derivadas da classe base.

[5] O mesmo pode ser feito para uma função não rotulada como `virtual` na classe base. Neste caso, a função da classe base só pode ser invocada com seu nome completo (formado usando o operador de escopo "::", precedido do nome da classe base e seguido do nome da função redefinida).

```
// OBJETOS.H: Declaração das classes

class OBJETO
{
private:
        // Variáveis internas
public:
        virtual void Desenhe() = 0;
};

class ESFERA : public OBJETO
{
public:  void Desenhe();
};

class CUBO : public OBJETO
{
public:  void Desenhe();
};

// OBJETOS.CPP: Implementação das funcionalidades

#include "OBJETOS.H"

void ESFERA::Desenhe()
{
        // Código que implementa a função
}

void CUBO::Desenhe()
{
        // Código que implementa a função
}

// PROGRAMA.CPP: Uso de objetos

#include "OBJETOS.H"

const int N = 255;
OBJETO*  Lista[N];

// Chama construtores
Lista[0]   = new ESFERA(x,y,z,...);
...
Lista[N-1] = new CUBO(x,y,z,...);

for (int i = 0; i < N; i++) {
      Lista[i]->Desenhe(); // Polimorfismo
      delete Lista[i];     // Destrói objeto
}
```

Figura 3.4 Aplicativo de Computação Gráfica demonstrando o uso do polimorfismo com funções virtuais na linguagem C++

3.3 Bibliotecas de classes

Um conjunto de novos tipos definidos em uma linguagem orientada para objetos pode ser desenvolvido especialmente comercialização, sendo este pacote denominado uma **biblioteca de classes**.

Para programadores especialistas em linguagens orientadas para objetos, esta possibilidade abre um novo campo de trabalho. Enquanto isto, programadores que ainda não dominam os conceitos da programação orientada para objetos podem usar as classes de uma biblioteca como se fossem tipos pré-definidos da linguagem.

A linguagem **C++** permite duas modalidades de distribuição de uma biblioteca de classes:

- a distribuição dos arquivos de cabeçalho (*headers*), que contém a definição das classes, junto com os arquivos de código-objeto (`.OBJ`), gerados a partir da compilação do código-fonte que implementa as funcionalidades das classes;

- a distribuição do código-fonte completo da biblioteca de classes.

Na primeira modalidade de distribuição, o desenvolvedor de aplicativos não tem possibilidade de alterar as classes pré-definidas. Além disto, o distribuidor da biblioteca de classes mantém sigilo sobre a implementação das funcionalidades. Devido a isto, as bibliotecas distribuídas apenas com o código-objeto e os arquivos de cabeçalho tem usualmente preço menor do que aquelas onde é fornecido o código-fonte completo. Apesar disto, em ambas modalidades é possível "derivar" novas classes tendo como base classes na biblioteca [MsC+91].

3.3.1 *Microsoft Foundation Class Library*

Para implementação do sistema **PHOTOPIX**, diversas classes básicas teriam de ser codificadas, para atender a requisições que não são parte da "finalidade essencial" do programa [McPa84]. Optou-se pelo uso de uma biblioteca de classes comercial, tendo sido escolhida a biblioteca MFC (*Microsoft Foundation Class Library*).

Em comparação a outras bibliotecas de classes, a biblioteca MFC apresenta os seguintes pontos positivos [Chri92, MsC+91]:

- além das classes básicas para a maioria das funcionalidades geralmente utilizadas em aplicativos comuns, há um subconjunto de classes específicas para suporte ao ambiente *MS-Windows*;

- portabilidade do código-fonte, que não depende de nenhuma extensão à sintaxe e semântica da linguagem C++;

- suporte à verificação dinâmica do tipo de um objeto (*dynamic type checking*), que permite obter o tipo de um objeto em "tempo de execução";

- suporte à depuração de aplicativos;

- suporte à persistência de objetos, isto é, o armazenamento e recuperação do "estado" de um objeto em um meio de armazenamento "não-volátil", como discos rígidos;

- disponibilidade do código-fonte da biblioteca, permitindo a modificação de classes pré-definidas.

A filosofia de projeto da biblioteca MFC privilegia a definição de classes possuindo poucas funções. Isto aumenta o número de classes básicas necessárias para "cobrir" determinado número de funcionalidades. Contudo, torna o uso das classes pré-definidas mais eficiente, especialmente em compiladores onde não é possível a ligação ao código-objeto do aplicativo apenas das funções realmente utilizadas em uma classe base.

3.3.1.1 Biblioteca MFC para o ambiente *MS-Windows*

Diversas classes da biblioteca MFC são definidas exclusivamente para "encapsular" funcionalidades da API do ambiente *MS-Windows*. A hierarquia do subconjunto da biblioteca MFC para o ambiente *MS-Windows* é apresentada na Figura 3.5. A funcionalidade oferecida em cada uma das classes, assim como a documentação dos nomes e parâmetros de suas funções, pode ser encontrada em [Chri92, MsC+91].

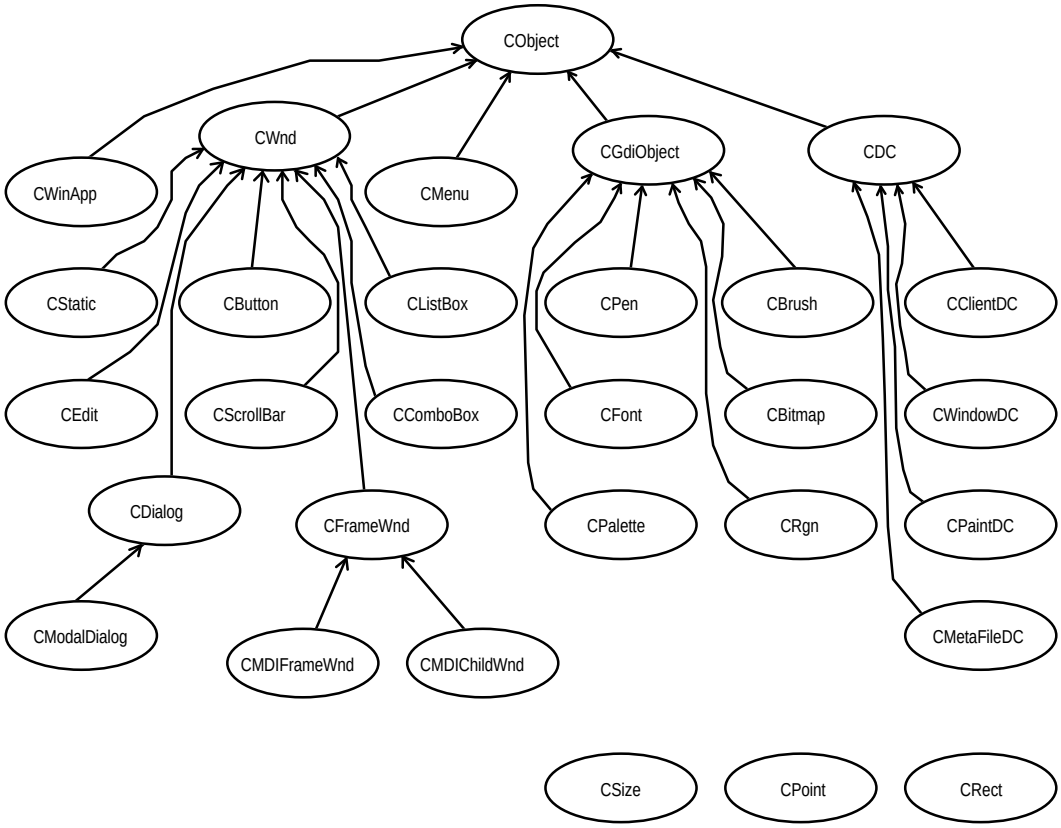

Figura 3.5 Biblioteca MFC para o ambiente *MS-Windows*

A classe `CObject`, da qual é derivada a maior parte das classes da biblioteca MFC, implementa funcionalidades básicas, como a verificação dinâmica do tipo de um objeto e o suporte à persistência, que tornam-se disponíveis em todas as classes derivadas.

Duas classes imprescindíveis à elaboração de um aplicativo para o ambiente *MS-Windows* são a classe `CWinApp` e a classe `CFrameWnd` (ou `CMDIFrameWnd` para aplicativos MDI).

A classe `CWinApp` implementa as funcionalidades básicas necessárias para um aplicativo no ambiente *MS-Windows*. O programador de aplicativo tem apenas de codificar uma classe derivada de `CWinApp`, redefinindo a sua função `InitInstance()` (invocada no início da execução de cada "instância" do aplicativo), onde deve ser criado um objeto que corresponde à janela principal do aplicativo. Após "carregar" o aplicativo para a memória, o *MS-Windows* invoca uma das funções do objeto `CWinApp`, que executa todos os procedimentos necessários ao início do funcionamento de um aplicativo.

Para implementar as funcionalidades necessárias na janela principal de um aplicativo, pode-se utilizar a classe `CFrameWnd` (ou a classe `CMDIFrameWnd`, em aplicativos MDI). As funções pertencentes à classe `CFrameWnd` são, em sua maioria, virtuais, para facilitar o uso do polimorfismo. Um aplicativo geralmente deriva uma classe para a sua janela principal tendo por base a classe `CFrameWnd`, adiciona algumas rotinas específicas, definindo em seguida um mapa de mensagens.

3.3.1.2 Mapa de mensagens

Para direcionar as mensagens que um aplicativo recebe ao procedimento que implementa sua resposta, a biblioteca de classes MFC implementa um mecanismo denominado mapa de mensagens (*message-map* [Chri92, MsC+91]). Sua sintaxe aparece na Figura 3.6, que mostra trechos do código-fonte do sistema **PHOTOPIX**.

Para usar o mapa de mensagens da biblioteca MFC, o programador de um aplicativo acrescenta à definição da classe derivada para sua janela a macro `DECLARE_MESSAGE_MAP()`. No arquivo que contém o código-fonte para as funcionalidades da classe, a expressão `BEGIN_MESSAGE_MAP` inicia a definição do mapa de mensagens, seguindo-se o nome da própria classe para a qual se está definindo o mapa, e o nome da classe base para a qual serão direcionadas as mensagens não respondidas "via mapa". Segue-se uma série de comandos, de acordo com regras de sintaxe estabelecidas em [MsC+91]. O mapa de mensagens termina na macro `END_MESSAGE_MAP()`.

```
// PPIX.H: Declaração da classe CPPixMainWnd

class CPPixMainWnd : public CMDIFrameWnd
{
private: // Variáveis internas

public:
         afx_msg void OnSize(UINT, int, int);
         afx_msg void OnAbout();
         DECLARE_MESSAGE_MAP()
};

// PPIX.CPP: Implementação das funcionalidades

BEGIN_MESSAGE_MAP(CPPixMainWnd, CMDIFrameWnd)

    ON_WM_SIZE()

    ON_COMMAND(IDM_HELPABOUT, OnAbout)

END_MESSAGE_MAP()

void CPPixMainWnd::OnAbout()
{
  // Código que implementa a função
}

void CPPixMainWnd::OnSize(UINT, int, int)
{
  // Código que implementa a função
}
```

Figura 3.6 Mapa de mensagens da biblioteca MFC

A implementação do mapa de mensagens depende exclusivamente do uso de macros, substituídas pelo pré-processador antes da compilação do código-fonte [RaSc90]. Seu uso "encapsula" diversos detalhes da programação para o ambiente *MS-Windows* [MSDK92, Nort90, Petz90], permitindo ao programador concentrar-se na codificação da "finalidade essencial" de seu aplicativo.

Cada comando do mapa de mensagens corresponde ao direcionamento de uma mensagem específica para uma das funções pertencentes à classe para a qual o mapa de mensagens é definido. Para a maioria das mensagens existentes há uma correspondente função virtual definida em uma das classes base para janelas da biblioteca MFC. O programador de aplicativo deve redefinir a função, com mesmo nome e parâmetros, na classe para sua janela. Acrescenta-se então ao mapa de mensagens uma macro, indicando que a mensagem será respondida pela classe derivada, e não direcionada à classe base.

Deve-se observar que o nome de cada uma das funções, que implementam respostas para mensagens direcionadas via mapa de mensagens, deve estar definido no escopo da classe à qual pertence o mapa. Um aplicativo pode ter mais de um mapa de mensagens, pois podem ser definidas diversas classes diferentes para janelas e quadros de diálogo que irão aparecer durante sua execução. Contudo, mensagens que chegam para uma janela não podem ser respondidas por uma função implementada em outra classe, que não a da própria janela. Portanto, funções de classes que não são correspondentes a nenhum objeto da interface gráfica não podem receber mensagens, podendo porém ser invocadas durante a "resposta" à mensagem, implementada por uma função pertencente à classe ligada a um objeto da interface[1] (nota de rodapé na página 36).

3.4 Conclusões

A programação orientada a objetos dá ao programador de aplicativos o poder de ampliar a linguagem que está sendo utilizada, adicionando novos tipos de dados.

A linguagem **C++** permite a utilização prática de todos os princípios da programação orientada para objetos, ainda que numa sintaxe e semântica que podem requerer um tempo de estudo considerável.

O desenvolvimento e teste de um novo tipo de dados pode ser realizado por programadores especializados, que dedicam-se à elaboração de bibliotecas de classes para comercialização.

A biblioteca de classe MFC possui diversas classes básicas que dão suporte à programação para o ambiente *MS-Windows*, liberando o programador de ter que aprender detalhes da API. Particularmente positivo é o fato de que tal biblioteca implementa um mecanismo de direcionamento de mensagens, baseado no uso de macros, que libera o programador da codificação da função principal de um aplicativo para o ambiente *MS-Windows*. Tal função deve estar presente em todo aplicativo do ambiente *MS-Windows*, para obter e distribuir mensagens (Capítulo 2).

Capítulo 4

O sistema PHOTOPIX

A finalidade da Engenharia de *Software* é o desenvolvimento e a aplicação de metodologias e ferramentas para a implementação e manutenção econômica de *software* de qualidade preditível e controlável, operando de modo econômico em máquinas e ambientes reais [Staa87]. Para atingir estes objetivos, é usual dividir a implementação de um sistema complexo em uma série de etapas.

Inicialmente, produz-se uma especificação do sistema, que define os objetivos a serem alcançados pelo(s) programa(s), sua interface com o ambiente, as restrições técnicas a serem satisfeitas e os critérios para avaliação da qualidade do resultado final obtido.

Conseguida uma especificação, detalhada ao nível adequado, procede-se a etapa de codificação do(s) programa(s), em versões para uma ou mais plataformas de *hardware*. É usual dividir um programa em módulos, que possam ser desenvolvidos e testados isoladamente, sendo depois devidamente acoplados, formando um conjunto integrado que cumpra os objetivos da especificação.

Após a etapa de codificação, segue-se a depuração e aperfeiçoamento do sistema, ou a extensão de suas capacidades previstas inicialmente. Infelizmente, para a maioria dos sistemas implementados com as tecnologias da Informática atualmente disponíveis, é difícil ou impossível o "teste sistemático" dos programas desenvolvidos [Staa87]. Devido a isto, o máximo de cuidado deve ser empregado nas etapas de projeto e codificação, para que delas resultem programas "corretos por construção", o que não é sinônimo de programas perfeitos, mas sim de programas de boa qualidade.

Este capítulo descreve as metodologias empregadas na especificação e codificação do sistema **PHOTOPIX**, detalhando sua estrutura interna. Ao final, apresenta-se a interface do sistema com o usuário, na versão implementada para o ambiente operacional *MS-Windows*.

4.1 Especificação do sistema PHOTOPIX

Para especificar o sistema **PHOTOPIX**, foi utilizada uma das mais modernas metodologias concebidas para especificação de sistemas informatizados: a "Análise Essencial" [McPa84,Your90]. Numa apresentação resumida, a Análise Essencial introduz poucos conceitos e regras a serem seguidas para obter a especificação de um sistema.

4.1.1 Análise Essencial

No seu atual estágio de desenvolvimento, as plataformas de *hardware* utilizadas para implementação de sistemas informatizados contém dois componentes básicos: os "processadores", que executam atividades, e a "memória", onde são armazenados dados para uso dos processadores. A Análise Essencial define conceitos e regras que auxiliam o projetista de um sistema a evitar uma especificação repleta de referências às limitações das plataformas de *hardware* atualmente disponíveis.

4.1.1.1 Tecnologia perfeita

O conceito básico definido pela Análise Essencial é o de "tecnologia perfeita". Uma plataforma de *hardware* implementada com tecnologia perfeita teria um processador perfeito e uma memória perfeita.

Um processador perfeito daria à plataforma de *hardware* que o utilizasse a habilidade de realizar qualquer atividade, instantaneamente e sem nenhum gasto. Não haveria consumo de energia, erros, falhas, quebras, demoras ou qualquer outra restrição imposta devido à imperfeição das atuais tecnologias de construção de computadores.

As mesmas virtudes seriam válidas para uma memória perfeita, que não teria custo e daria à plataforma de *hardware* onde estivesse instalada a capacidade de armazenar uma quantidade infinita de dados, que poderiam ser acessados pelo processador randomicamente, sem qualquer atraso [McPa84].

Deve-se atentar para o fato de que, numa plataforma de *hardware* implementada com tecnologia perfeita, poderiam ser executados algoritmos que ainda não possuem larga utilização prática justamente devido à sua ineficiência, ou necessidade de enormes bases de dados. Exemplos seriam algoritmos para análise de imagens, reconhecimento de comandos de voz, inteligência artificial, etc. Assim, o uso de tecnologia perfeita poderia alterar profundamente a interface homem-máquina [Amar92] a que estão acostumados os atuais usuários da Informática.

4.1.1.2 Essência de um sistema

A diretriz a ser seguida na etapa de especificação, segundo os princípios da Análise Essencial, é a de encontrar a "essência" do sistema. A essência de um sistema que se utilize das tecnologias da Informática consiste no conjunto das suas "atividades essenciais" e na sua "memória essencial" [McPa84].

As atividades essenciais são os procedimentos que o sistema teria de realizar, mesmo que fosse possível implementá-lo utilizando tecnologia perfeita. A memória essencial consiste nos dados que teriam de ser obrigatoriamente armazenados, mesmo que o sistema executasse apenas as suas atividades essenciais.

Não se impõe limite à complexidade de cada atividade essencial de um sistema, visto que, com tecnologia perfeita, sua execução seria sempre instantânea. Também não há limite para o volume de dados que pode ser necessário armazenar na memória essencial, pois a plataforma de *hardware* que executaria o "modelo essencial" do sistema possuiria uma memória perfeita.

É possível distinguir dois tipos de atividades essenciais em um sistema: as atividades fundamentais e as custodiais.

Uma atividade essencial é dita fundamental quando ajuda a justificar a existência do sistema. Já as atividades custodiais são necessárias para estabelecer e manter a memória essencial.

Atividades custodiais usualmente dizem respeito à coleta de dados a serem armazenados na memória essencial. Contudo, deve-se observar que procedimentos como "ler arquivo para memória principal" não são atividades custodiais, visto que não são sequer atividades necessárias em um sistema implementado com tecnologia perfeita.

4.1.1.3 Fronteira do sistema

É importante que, na busca pela essência de um sistema, o conceito de tecnologia perfeita não ultrapasse as fronteiras da plataforma de *hardware*. Conseqüência óbvia do fato de supor que os usuários de um sistema são perfeitos seria a conclusão de que estes usuários não precisariam de computadores e programas.

Durante a especificação da essência de um sistema devem ser mantidas todas as atividades que digam respeito ao tratamento das limitações do ambiente externo, o que compreende não apenas os usuários do sistema mas também outros sistemas com os quais ocorra troca de dados, limitações temporais, etc. [McPa84].

4.1.2 Essência do sistema PHOTOPIX

A memória essencial do sistema **PHOTOPIX** é constituída unicamente por "imagens". A sua única atividade essencial fundamental é executar algoritmos de PDI sobre estas imagens, gerando novas imagens que são também armazenadas na memória essencial. A Figura 4.1 apresenta o modelo essencial do sistema **PHOTOPIX**.

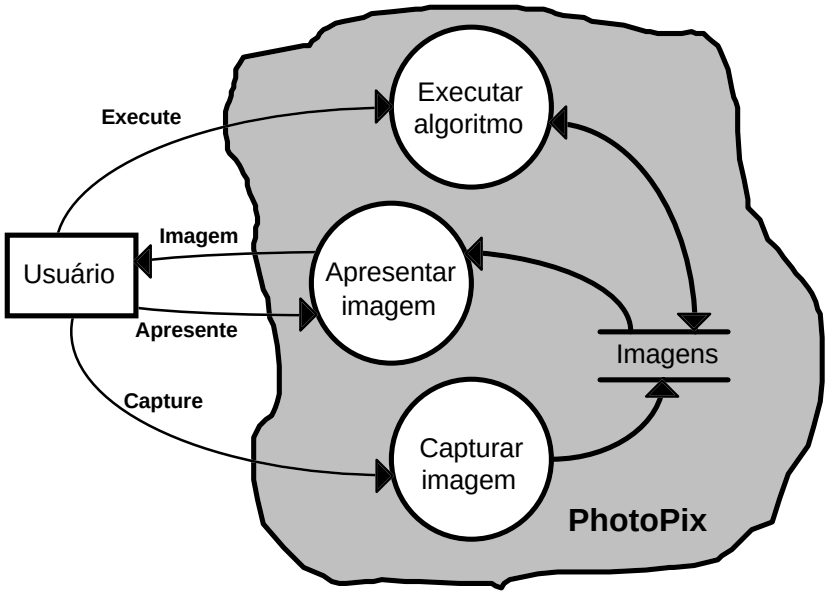

Figura 4.1 Modelo essencial do sistema PHOTOPIX

Duas atividades custodiais aparecem no modelo essencial. A atividade de "captura de imagens" é necessária para alimentar a memória essencial com imagens digitais. Idealmente, o sistema **PHOTOPIX** obteria imagens diretamente de *scanners*, câmeras e outros aparelhos para aquisição de imagens digitais[6]. Uma vez adquirida, uma imagem estaria indefinidamente disponível na memória essencial.

A outra atividade custodial necessária para o sistema é a que possibilita levar as imagens da memória essencial ao mundo externo. Atendendo comandos do usuário, o sistema deve apresentar uma ou mais imagens em qualquer dos dispositivos de apresentação disponíveis na configuração da plataforma de *hardware* em uso.

[6] Alguns programas comerciais já implementam esta funcionalidade.

4.2 Codificação do sistema PHOTOPIX

O modelo essencial de um sistema permite um enorme número de implementações diferentes. Contudo, o uso de linguagens de programação inadequadas e o impacto da tecnologia imperfeita disponível nas atuais plataformas de *hardware*, pode fazer desaparecer no código-fonte do sistema a clareza das relações estabelecidas no modelo essencial.

Sistemas para PDI são particularmente susceptíveis aos impactos causados pela tecnologia imperfeita. Referências a configurações de *hardware* específicas, impossibilitam o reaproveitamento da maior parte do código-fonte já desenvolvido.

Apesar do inevitável aparecimento de requisições não-essenciais no processo de codificação de um sistema, diversas APIs disponíveis comercialmente permitem minimizar as referências à plataforma de *hardware* em uso. Dentre estas, a API do ambiente operacional *MS-Windows* (Capítulo 2) foi escolhida como base para a primeira implementação prática do sistema **PHOTOPIX**.

Atualmente, as técnicas de programação orientada para objetos oferecem a melhor opção para codificação e teste de módulos isoladamente, permitindo depois a sua correta integração. Para codificação do sistema **PHOTOPIX**, foi escolhida a mais popular das linguagens orientadas para objetos: a linguagem C++ (Capítulo 3).

4.2.1 Restrições tecnológicas

Obtido o modelo essencial de um sistema, devem ser determinadas então as restrições tecnológicas que permitam obter um resultado útil em prazo pré-determinado.

As principais restrições tecnológicas presentes na implementação atual do sistema **PHOTOPIX** são:

- suporte apenas a imagens digitais codificadas no modelo *raster* [HeBa86]. Embora imagens digitais possam ser codificadas como uma série de vetores, este tipo de representação não é suportado na classe `CImage`, que armazena a imagem em um DIB (item 2.3.4.6);

- impossibilidade de aquisição de imagens diretamente do ambiente externo, assim como inviabilidade do armazenamento por tempo indefinido na memória essencial. O sistema obterá imagens digitais em arquivos, em uma limitada gama de formatos;

- redundância da informação sobre a imagem. A possível incompatibilidade entre a resolução espectral da imagem representada por um DIB e a resolução espectral do vídeo, poderia ocasionar problemas de performance na interface aplicativo-usuário (item 2.3.4.6). A solução empregada para garantir uma performance aceitável é manter um *bitmap* (item 2.3.4.5) espelhando o conteúdo do DIB [MDTG92a, MDTG92b, MDTG92c].

4.2.2 Identificando classes

A implementação de um sistema orientado para objetos inicia-se com a identificação dos novos tipos de dados (classes) a serem definidos, codificados isoladamente e depois utilizados no(s) programa(s).

A partir do modelo essencial do sistema **PHOTOPIX** (Figura 4.1) é possível imediatamente identificar uma classe: a das imagens. As imagens são a matéria-prima e também o produto final do processamento digital de imagens. Portanto, constituem uma classe básica para qualquer sistema de PDI.

Também a partir do modelo essencial, é possível identificar que os algoritmos devem constituir uma classe. Os diferentes algoritmos seriam implementados como "derivações" da classe básica, sendo que, em "tempo de execução", seria usado o polimorfismo via funções virtuais para garantir a chamada ao código correto que implementa um algoritmo específico (item 3.2.3).

O uso da interface para múltiplos documentos (MDI, item 2.3.2.7) é considerado vital para o sistema **PHOTOPIX**, pois a capacidade de armazenamento e processamento de um sistema de PDI é mal aproveitada quando o usuário não pode ver e comparar mais de uma imagem simultaneamente. Com o uso da biblioteca de classes MFC (item 3.3.1), um mínimo de codificação é necessário para implementar tal funcionalidade no sistema.

Na Figura 4.2 é apresentada a hierarquia das classes definidas para o sistema **PHOTOPIX**. Segue-se uma convenção de nomes semelhante à da biblioteca de classes MFC [Chri92, MsC+91]. As classes específicas para lidar com a interface aplicativo-usuário no ambiente *MS-Windows* são derivadas de suas equivalentes funcionais na biblioteca MFC (Figura 3.5).

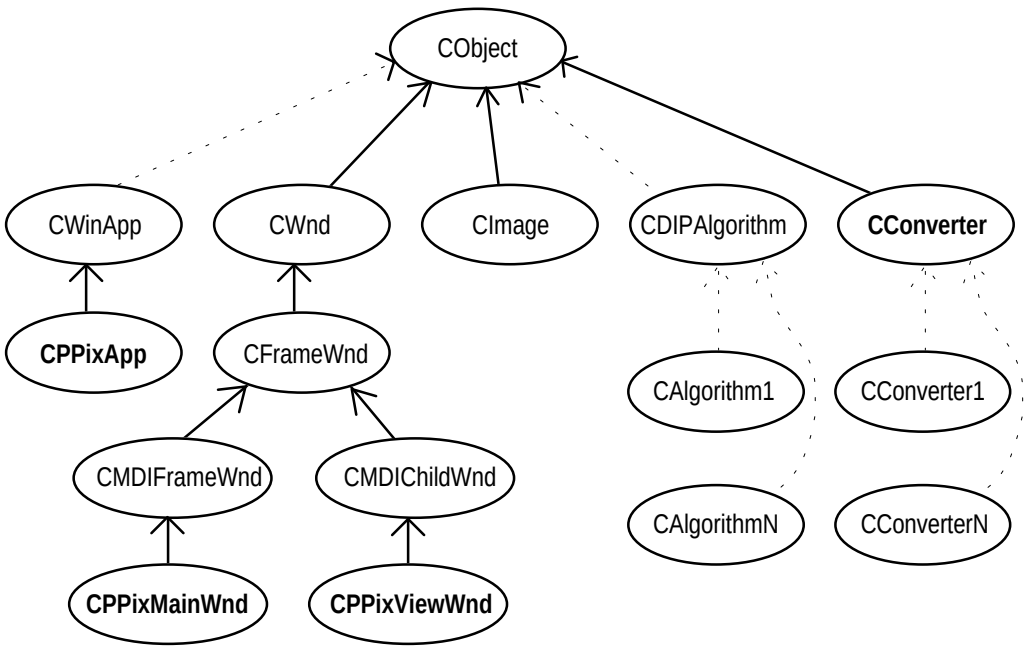

Figura 4.2 Hierarquia das classes do sistema PHOTOPIX

A classe `CPPixApp` controla a execução do aplicativo. A classe `CPPixMainWnd` fornece as funcionalidades da janela principal de um aplicativo MDI. A classe `CPPixViewWnd` implementa as funcionalidades das janelas-filhas de um aplicativo MDI, atuando como um visor através do qual é possível visualizar as imagens armazenadas na memória essencial, que são da classe `CImage`. A classe `CDIPAlgorithm` serve como base para derivação de classes que implementam algoritmos de PDI. A classe `CConverter` serve como base para derivação de classes que implementam conversores entre a representação interna das imagens e os inúmeros formatos existentes para arquivos contendo imagens.

A estrutura do código-fonte do sistema **PHOTOPIX** é apresentada no Apêndice A. São listados também os arquivos de cabeçalho (*headers*), que definem os dados e funcionalidades implementadas em cada classe definida para o sistema. É delineada também a metodologia de trabalho a ser seguida por programadores de algoritmos de PDI que irão extender as funcionalidades do sistema.

4.3 Interface do sistema PHOTOPIX com o usuário

Uma interface com o usuário de boa qualidade minimiza a ocorrência de erros, sejam eles devidos a modos de operação, representação, inconsistência, captura ou ativação [Amar92].

O sistema **PHOTOPIX** segue as regras de projeto para a interface com o usuário de aplicativos desenvolvidos para o ambiente *MS-Windows*, definidas em [Msft92b].

4.3.1 Janela principal

A janela principal do sistema **PHOTOPIX** possui todos os elementos comuns nos aplicativos que seguem o padrão MDI (Figura 4.3).

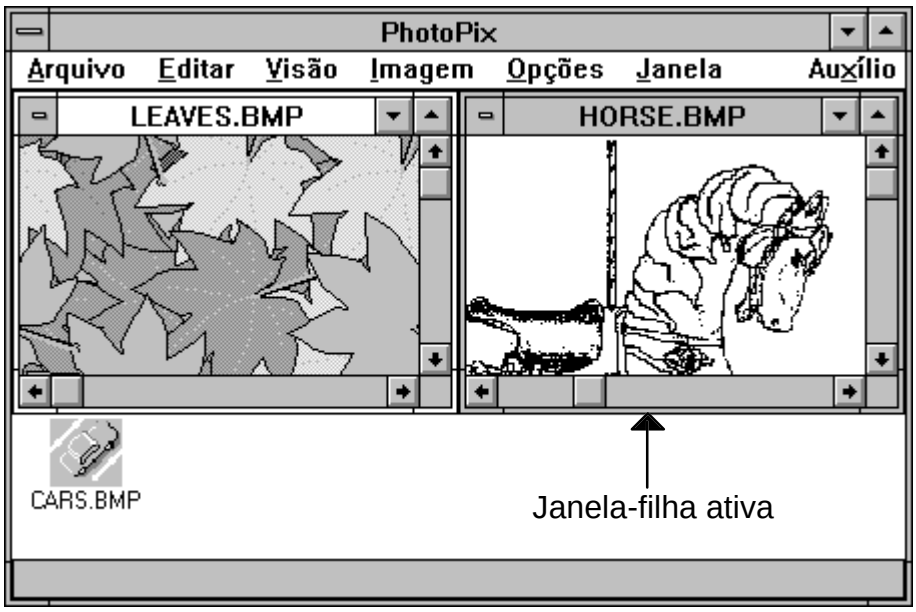

Figura 4.3 Janela principal do sistema PHOTOPIX

A janela principal pertence à classe `CPPixMainWnd`, enquanto as janelas-filhas pertencem à classe `CPPixViewWnd` (Figura 4.2). As janelas-filhas são recortadas (*clipping*) nos limites da janela principal.

A operação de menus, bordas, ícones e outros controles presentes na janela principal em nada difere dos procedimentos padronizados, especificados em [Msft92a, Msft92b].

A hierarquia de menus acessados a partir da janela principal é apresentada na Figura 4.4. Para alguns itens de menu foram definidas teclas de atalho (*shortcuts* ou *accelerators*).

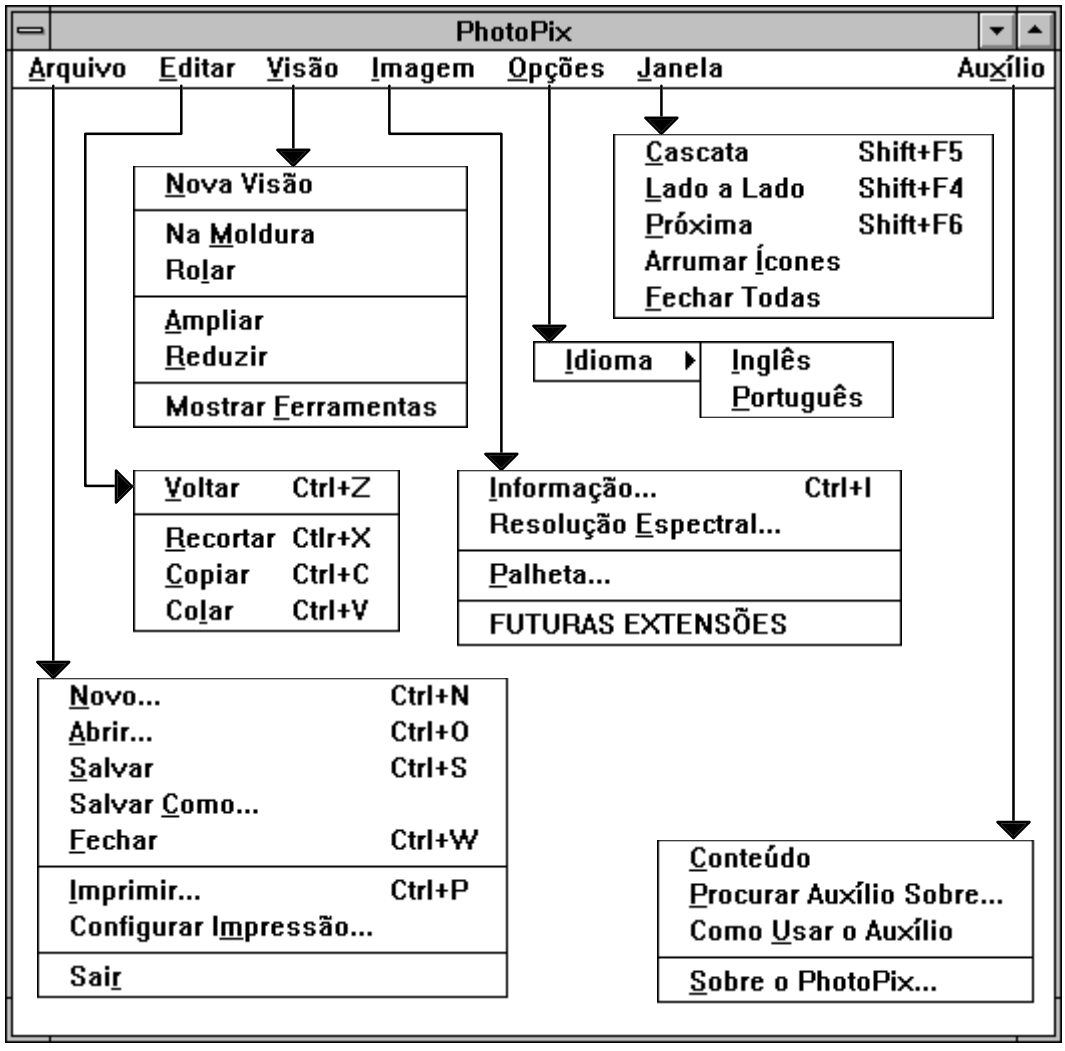

Figura 4.4 Hierarquia de menus do sistema PHOTOPIX

4.3.1.1 Menu Arquivo (*File*)

O menu **Arquivo** (*File*) permite ao usuário comandar ações para transferência de imagens entre a memória essencial do sistema e o ambiente externo. As opções disponíveis neste menu atuam sempre sobre a "imagem ativa", que é aquela mostrada na "janela-filha ativa" (Figura 4.3).

Novo... (*New...*): Aciona quadro de diálogo que, devidamente preenchido, permite a criação de uma nova imagem. É criada também uma janela-filha, que passa a apresentar a imagem, que torna-se a imagem ativa.

Abrir... (*Open...*): Aciona quadro de diálogo onde o usuário seleciona um arquivo contendo uma imagem, que será lida para a memória do sistema. É criada também uma janela-filha, que passa a apresentar a imagem, que torna-se a imagem ativa.

Salvar (*Save*): Transfere a imagem ativa da memória do sistema para um arquivo externo. Caso a imagem tenha sido lida de um arquivo ou já tenha sido salva anteriormente, os dados sobrescrevem o antigo arquivo.

Salvar Como... (*Save As...*): Transfere a imagem ativa da memória do sistema para um arquivo externo. Nesta opção o usuário deverá sempre especificar o nome do arquivo onde será armazenada a imagem.

Fechar (*Close*): Apaga a imagem ativa da memória essencial, destruindo também todas as janelas-filhas que estejam apresentando a imagem ou dados sobre ela. Caso a imagem tenha sido criada ou alterada mas ainda não transferida para o ambiente externo, é exibido um quadro de diálogo, onde o usuário autoriza ou não o "salvamento" da imagem.

Imprimir... (*Print...*): Transfere a imagem ativa para um contexto de dispositivo (item 2.3.4.1) associado a uma impressora ou outro dispositivo de apresentação que produz cópias não-voláteis da imagem.

Configurar Impressão... (*Print Setup...*): Exibe quadro de diálogo que permite configurar os dispositivos que podem servir como destino das operações de impressão.

Sair (*Exit*): Encerra a execução do sistema. Para cada imagem na memória do sistema que foi criada ou alterada mas ainda não foi transferida para o ambiente externo, é exibido um quadro de diálogo, onde o usuário autoriza ou não o "salvamento" da imagem.

4.3.1.2 Menu Editar (*Edit*)

Posicionado segundo os padrões da interface *MS-Windows* [Msft92b], o menu **Editar** (*Edit*) permite a troca de dados entre o sistema **PHOTOPIX** e outros aplicativos do ambiente *MS-Windows*, ou a desistência de uma ação com resultado indesejado. As opções disponíveis neste menu são as seguintes:

Voltar (*Undo*): Esta opção estará habilitada apenas quando a imagem ativa tiver sido processada, sendo possível retornar ao estado anterior.

Recortar (*Cut*): Esta opção transfere uma área selecionada da imagem ativa para o *clipboard* [Msft92b, MSDK92]. A área selecionada deve ser apagada na imagem ativa.

Copiar (*Copy*): Esta opção transfere uma área selecionada da imagem ativa para o *clipboard*, porém preserva o estado da imagem.

Colar (*Paste*): Esta opção transfere o conteúdo do *clipboard* para a imagem ativa, em posição a ser determinada pelo usuário, de preferência interativamente.

4.3.1.3 Menu Visão (*View*)

Este menu destina-se, principalmente, a apresentar comandos que atuam sobre as janelas-filhas do sistema **PHOTOPIX**, modificando a forma como as imagens são apresentadas. É utilizado também para fazer com que apareçam ou não quadros de diálogo *modeless*, como a caixa da ferramentas. As opções disponíveis neste menu são as seguintes:

Nova Visão (*New View*): Cria nova janela-filha, que passa a mostrar a imagem ativa. Portanto, mais de uma janela-filha pode estar mostrando a mesma imagem da memória essencial, em posições ou com ampliações diferentes.

Na Moldura (*Stretch*): Ajusta a visão da imagem ao tamanho da área de trabalho disponível na janela-filha. Não é preservada a razão de aspecto [HeBa86] da imagem.

Rolar (*Scroll*): Por meio de barras de rolagem (item 2.3.2.6), o usuário pode selecionar a área da imagem que será apresentada na área de trabalho da janela-filha. Será visível a porção da imagem que for possível apresentar, após feita a ampliação ou redução especificada pelo usuário.

Ampliar (*Zoom In*): Aumenta a ampliação corrente da imagem. Os aumentos possíveis multiplicam sequencialmente o tamanho de cada *pixel* por um número inteiro. Assim, começando de uma ampliação inicial de 1:1, que leva um *pixel* da imagem a cada *pixel* da tela, são possíveis ampliações de 1:2, levando um *pixel* da imagem para quatro (2*2) *pixels* na tela, 1:3, 1:4, etc.

Reduzir (*Zoom Out*): Diminui a ampliação corrente da imagem. As reduções possíveis correspondem a subamostragens da imagem original, desprezando um número inteiro de *pixels*. Assim, começando com uma amostragem inicial de 1:1, onde um *pixel* da tela é obtido consultando um *pixel* da imagem original, são possíveis reduções de 2:1, onde quatro (2*2) *pixels* da imagem geram apenas um *pixel* na tela, 3:1, 4:1, etc.

Mostrar Ferramentas (*Show ToolBox*): Mostra ou esconde a caixa de ferramentas (*Toolbox*) do sistema (item 4.3.3).

4.3.1.4 Menu Imagem (*Image*)

Neste menu são acionados quadros de diálogo que mostram dados sobre a imagem ativa ou permitem selecionar opções para execução de algoritmos. É o local onde deverão ser inseridos itens que acionem os algoritmos futuramente implementados. As opções disponíveis neste menu são as seguintes:

Informação... (*Info...*): Aciona quadro de diálogo que mostra informações sobre a imagem ativa (dimensões, resolução espectral, nome do arquivo, etc.).

Resolução Espectral... (*Color Depth...*): Aciona quadro de diálogo onde são selecionados algoritmos e outras opções para conversão da informação espectral da imagem ativa. É gerada nova imagem e uma nova janela-filha, onde tal imagem é apresentada, tornando-se a imagem ativa. A nova imagem possui a resolução espectral selecionada pelo usuário no quadro de diálogo e a mesma resolução espacial da imagem anteriormente ativa. A informação espectral da nova imagem é obtida a partir da conversão da informação espectral da imagem anteriormente ativa, por um dos algoritmos implementados no sistema (Capítulo 5).

Palheta... (*Palette...*): Mostra palheta da imagem corrente, permitindo edições em cada entrada individualmente.

4.3.1.5 Menu Opções (*Options*)

Atualmente, o menu de **Opções** (*Options*) possui apenas um item:

Idioma (*Language*): Permite a seleção da língua (Português/Inglês) utilizada para mensagens, menus e textos presentes nos quadros de diálogo do sistema **PHOTOPIX**.

4.3.1.6 Menu Janela (*Window*)

O menu **Janela** (*Window*) é utilizado para acionar os procedimentos de redimensionamento e posicionamento automático das janelas-filhas de um aplicativo MDI. As opções disponíveis neste menu são as seguintes:

Cascata (*Cascade*): Este comando faz com que a janela *MDI Client* (item 2.3.2.7) redimensione e posicione as janelas-filhas de forma que, iniciando no canto superior esquerdo da área de trabalho na janela principal, cada janela-filha seguinte seja deslocada, para a direita e para baixo, de forma que seu canto superior esquerdo inicie no canto inferior direito do ícone de menu do sistema da janela-filha anteriormente posicionada.

Lado a lado (*Tile*): Este comando faz com que a janela *MDI Client* redimensione e posicione cada janela-filha de forma que não ocorra sobreposição de suas áreas.

Próxima (*Next*): Torna ativa a próxima janela-filha, numa fila circular baseada na ordem de criação.

Arranjar Ícones (*Arrange Icons*): Posiciona organizadamente janelas-filhas que estejam iconizadas.

Fechar Todas (*Close All*): Envia mensagem de destruição para cada uma das janelas-filhas.

Lista de janelas: Permite escolher, em uma lista com o título de cada uma das janelas-filhas, qual deve ser ativada.

4.3.1.7 Menu Auxílio (*Help*)

Destina-se à prestação de auxílio *on-line* ao usuário do sistema. As opções disponíveis neste menu são as seguintes:

Conteúdo (*Contents*): Mostra o conteúdo do arquivo de auxílio.

Procurar Auxílio Sobre... (*Search for Help on...*): Procura por determinado tópico no arquivo de auxílio.

Como Usar o Auxílio (*How to Use Help*): Exibe arquivo de auxílio que mostra como utilizar o mecanismo de auxílio *on-line*.

Sobre o PhotoPix... (*About PhotoPix...*): Mostra informações sobre a versão do aplicativo e sobre o ambiente operacional.

4.3.1.8 Barra de *Status* (*Status bar*)

As barras de *status* são o mais visível exemplo de restrição tecnológica imposta ao desenvolvedor de aplicativos.

Originalmente, a utilidade da barra de *status* era, simplesmente, prover o usuário com informações sobre o andamento da execução de algoritmos complexos. Contudo, numa plataforma de *hardware* implementada com tecnologia perfeita, não há intervalo de tempo mensurável entre o instante em que o usuário determina a execução de um algoritmo e o instante em que a execução termina. Logo, para sua finalidade original, a barra de *status* não tem função no modelo essencial de um sistema.

Atualmente, é comum que aplicativos reservem parte de sua área de trabalho para apresentar mensagens sobre dados que o usuário pode não perceber precisamente, como a posição do cursor, ou lembrar-se, como o estado das teclas de controle do teclado (*shift*, *control*, etc.), o modo de operação do programa, etc. Estas "barras de mensagem" são também denominadas de barras de *status* por alguns autores [Msft92b]. O sistema **PHOTOPIX** reserva espaço na parte inferior da sua área de trabalho para uma barra de *status* (Figura 4.5).

4.3.2 Quadros de diálogo

Os quadros de diálogo do sistema **PHOTOPIX** foram elaborados segundo as regras de projeto de aplicativos para o sistema *MS-Windows*, especificadas em [Msft92b]. Um usuário proficiente do ambiente operacional *MS-Windows* está apto a utilizar qualquer dos quadros de diálogo do sistema **PHOTOPIX**. O idioma utilizado nas mensagens pode ser alterado, através de um item no menu do aplicativo.

4.3.3 Caixa de Ferramentas (*ToolBox*)

Para facilitar a futura extensão das capacidades do sistema **PHOTOPIX**, foi implementada uma caixa de ferramentas (*ToolBox* [Msft92b]), controle não pré-definido no ambiente *MS-Windows*, mas que tem recebido cada vez maior aprovação por parte de usuários de programas de "pintura" e PDI (Figura 4.5).

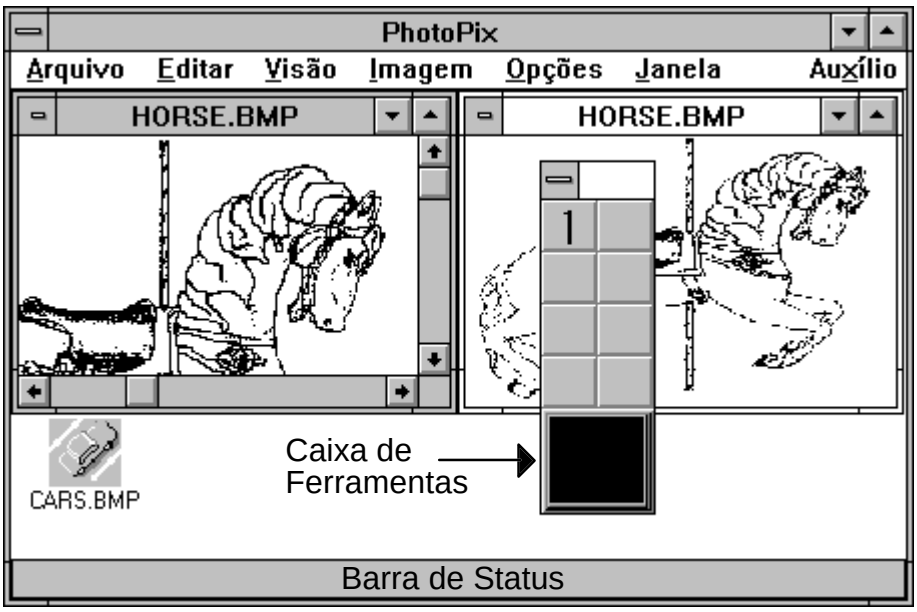

Figura 4.5 Caixa de ferramentas (*ToolBox*) do sistema PHOTOPIX

Grande parte da dificuldade para implementação de uma caixa de ferramentas deriva da obscuridade na documentação relativa à implementação de itens *owner-drawn* [MSDK92, Nort90, Petz90]. Estando a parte relativa à interface aplicativo-usuário já implementada, os programadores que pretendam ampliar as funcionalidades disponíveis no sistema **PHOTOPIX** têm apenas de "interceptar" mensagens geradas pela caixa de ferramentas, "respondendo" de maneira adequada.

4.4 Conclusões

O sistema **PHOTOPIX**, especificado segundo os princípios da Análise Essencial, possui como atividade essencial fundamental o processamento de imagens. Além desta, duas atividades custodiais são necessárias: uma que permite entrada de imagens para a memória essencial do sistema e outra que permite levar uma imagem da memória essencial para um dispositivo de apresentação.

A codificação do sistema **PHOTOPIX**, realizada na linguagem **C++**, seguiu o paradigma da programação orientada para objetos. Foram identificadas seis classes necessárias para implementação do sistema.

A classe `CPPixApp` controla a execução do aplicativo. A classe `CPPixMainWnd` fornece as funcionalidades da janela principal de um aplicativo MDI. A classe `CPPixViewWnd` implementa as funcionalidades das janelas-filhas de um aplicativo MDI, atuando como um visor através do qual é possível visualizar as imagens armazenadas na memória essencial, que são da classe `CImage`. A classe `CDIPAlgorithm` serve como base para derivação de classes que implementam algoritmos de PDI. A classe `CConverter` serve como base para derivação de classes que implementam conversores entre a representação interna das imagens e os inúmeros formatos existentes para arquivos contendo imagens.

O sistema **PHOTOPIX** segue as regras de projeto de interface aplicativo-usuário padronizadas para o ambiente *MS-Windows*. Destaca-se apenas o fato de que é reservada uma porção inferior da área de trabalho para um barra de status, tendo sido implementada também uma caixa de ferramentas, controle popular em programa de pintura e PDI no ambiente *MS-Windows*.

Capítulo 5

Conversão da informação espectral

O sistema **PHOTOPIX** lida com imagens contendo 1, 4, 8 ou 24 *bits* de informação espectral por *pixel*. Tal informação é organizada, quando na memória principal, num formato conveniente ao ambiente *MS-Windows*, conhecido como *packed-DIB* (*Device Independente Bitmap*, ou seja Mapa de *bits* Independente do Dispositivo) [Petz91a]. Neste formato, detalhado no Apêndice B**,** o valor armazenado para cada *pixel* representa um índice na palheta, para imagens com 1, 4 ou 8 *bits*/*pixel*, ou cada componente no sistema de cor RGB, para imagens com 24 *bits*/*pixel*.

Ao converter a informação espectral entre imagens com diferentes resoluções espectrais deve-se lidar com as seguintes possibilidades:

- expansão ou redução do número de *bits*/*pixel* usados para armazenar a informação espectral;

- conversão ou não da informação espectral, considerada *a priori* como especificando uma cor no espaço RGB, para uma escala de tons de cinza (`RGB(X,X,X)`, para `0 <= X <= 255`);

- distribuição ou não do erro local, devido à mudança da resolução espectral, para os *pixels* adjacentes.

São apresentados a seguir os algoritmos implementados no sistema **PHOTOPIX** para lidar com as possibilidades enumeradas acima.

Considera-se sempre o problema de converter a informação espectral de uma imagem original, no formato *packed-DIB*, para uma imagem-destino, cujo construtor já foi invocado (Capítulo 3), também no formato *packed-DIB*. Ambas imagens encontram-se na memória principal, que pode ser virtual.

5.1 Acesso à informação espectral de uma imagem

Para ler ou modificar o DIB de uma imagem foram implementadas duas subrotinas na classe `CImage` (Capítulo 4). Outras funções públicas da classe `CImage` são apresentadas no apêndice A.2.4.

DWORD CImage::GetPixel(DWORD x, DWORD y)

Esta subrotina retorna "sempre" o valor de cada componente, no sistema de cor RGB, da informação sobre o *pixel* na posição (x,y). Logo, para imagens com 1, 4 ou 8 *bits*/*pixel*, esta subrotina realiza a busca na palheta da imagem dos valores RGB correspondentes ao índice armazenado no DIB.

O dado retornado, que é do tipo DWORD, contém campos de um byte para cada componente do sistema RGB. Logo, são possíveis valores de 0 a 255 para a intensidade de cada componente.

DWORD CImage::SetPixel(DWORD x, DWORD y, DWORD dwColor)

Esta subrotina atualiza no DIB de uma imagem a informação sobre o *pixel* na posição (x,y). Se a imagem possui 24 *bits*/*pixel*, o valor de cada componente no sistema RGB é extraído de `dwColor` e armazenado convenientemente no DIB.

Para imagens com 1, 4, ou 8 *bits*/*pixel* o parâmetro `dwColor` deve especificar um índice na palheta da imagem. Isto melhora o desempenho da subrotina `SetPixel` nestes casos, além de permitir que a busca na palheta pela cor adequada para representar cada *pixel* seja realizada de modo diferente, de acordo com os diferentes métodos de conversão da informação espectral descritos a seguir.

O procedimento utilizado no sistema **PHOTOPIX** para conversão da informação espectral entre imagens é apresentado na Figura 5.1.

```
DWORD dwColor;       // Variáveis auxiliares
BYTE  btGray;

for (DWORD y = 0; y < ImgOriginal.GetHeight(); y++) {
   for (DWORD x = 0; x < ImgOriginal.GetWidth(); x++) {
      dwColor = ImgOriginal.GetPixel(x,y);
      if (ConvertToGrayScale) {
         btGray = (BYTE) ( (30*GetRValue(dwColor) +
                             59*GetGValue(dwColor) +
                            11*GetBValue(dwColor))/100 );
         dwColor = RGB(btGray, btGray, btGray);
      }
      if (ImgDestino.IsPaletted())
         dwColor =
            ImgDestino.GetNearestPaletteIndex(dwColor);
      ImgDestino.SetPixel(x,y,dwColor);
   }
}
```

Figura 5.1 Conversão da informação espectral entre imagens

Tal procedimento "percorre" toda a imagem original, obtendo, em cada *pixel*, o valor de cada componente no sistema RGB. Para imagens-destino que não serão convertidas para tons de cinza e nem usam palheta, é simplesmente invocada a função `SetPixel`, para que a imagem destino receba a mesma informação espectral armazenada sobre o *pixel* na imagem original.

No caso de conversão para escala de tons de cinza, é calculado o nível de cinza associado à cor do *pixel* na imagem original, de acordo com a expressão clássica que determina a luminosidade apresentada em televisores preto-e-branco [Harr87]:

NívelDeCinza = 0.3*Vermelho + 0.59*Verde + 0.11*Azul.

Para imagens destino que usam palheta, a informação espectral é aproximada, através da função `GetNearestPaletteIndex()`. Esta função devolve o índice na palheta-destino da cor que mais aproxima-se da especificada pelo parâmetro de entrada. Este índice indica a cor que minimiza a fórmula que mede a distância euclidiana entre cores no sistema RGB. Para duas cores **cor1** e **cor2**, a fórmula é:

$$distância^2 = (cor1.R - cor2.R)^2 + (cor1.G - cor2.G)^2 + (cor1.B - cor2.B)^2$$

Tal avaliação da distância entre a cor do *pixel* na imagem original e as cores da palheta-destino tem de ser feita para todos os índices possíveis na palheta destino. Este procedimento deve ser repetido para cada *pixel* da imagem. Portanto, a conversão da informação espectral é uma tarefa computacionalmente onerosa quando a imagem-destino usa palheta.

5.1.1 Palheta-invertida

Uma abordagem possível para reduzir o tempo gasto na avaliação de distâncias entre cores seria o uso de uma tabela onde estaria armazenada, para cada uma das cores da palheta original, a cor mais próxima na palheta-destino. Na literatura, esta tabela é chamada de palheta-invertida [Heck82, Othm92], pois relaciona cores puras a índices de uma palheta, fazendo o papel contrário de uma palheta comum, onde os índices "apontam" para cores puras.

Nesta abordagem, ao invés de invocar repetidamente a função `GetNearestPaletteIndex()`, deve-se apenas consultar a palheta-invertida, o que aumenta consideravelmente a performance da rotina de conversão da informação espectral para uma imagem-destino com palheta. O esforço para formação da palheta-invertida é usualmente desprezível em relação ao esforço para conversão da informação espectral sem o uso da palheta-invertida, a não ser para imagens de baixa resolução espectral.

Exemplificando, o número necessário de avaliações da distância entre cores, formando-se uma palheta-invertida, para conversão da informação espectral de uma imagem original de 8 *bits*/*pixel* para uma imagem-destino com a mesma resolução espectral, é de 256*256, sendo este valor independente da resolução espacial das imagens. Tal número corresponde à mesma quantidade de avaliações da distância entre cores necessária, sem o uso da palheta-invertida, para converter a informação espectral de apenas 16*16 *pixels*, de qualquer resolução espectral original, para uma imagem-destino de 8 *bits*/*pixel*.

5.2 Expansão da resolução espectral

O procedimento para expansão da resolução espectral é executado quando o número de *bits*/*pixel* da imagem-destino é maior "ou igual" ao da imagem original (no caso em que a resolução espectral da imagem-destino é igual à da imagem original não ocorre propriamente uma expansão da resolução espectral; contudo, observou-se que seria redundante a existência de um procedimento para tratar conversões entre imagens com mesma resolução espectral).

O procedimento utilizado para conversão da informação espectral entre imagens é o apresentado na Figura 5.1. Será preciso apenas estabelecer critérios para formação da palheta da imagem-destino, quanto isto for necessário.

5.2.1 Formação da palheta da imagem-destino

A formação da palheta da imagem-destino é um fator essencial na qualidade visual da imagem gerada. Nos casos de expansão da resolução espectral, o sistema **PHOTOPIX** usa um procedimento diferente para gerar a palheta-destino, para cada uma das possíveis resoluções espectrais da imagem-destino.

5.2.1.1 Imagem-destino de 1 *bit*/*pixel*

A imagem-destino possuirá apenas duas posições na sua palheta. Portanto, a única hipótese em que é chamado o procedimento de expansão da resolução espectral, com uma imagem-destino de 1 *bit*/*pixel*, é a de uma imagem original também de 1 *bit*/*pixel*.

Se o usuário escolheu a opção de converter a imagem para tons de cinza, a palheta-destino é preenchida com a cor preto, `RGB(0,0,0)`, em sua primeira posição, sendo o branco, `RGB(255,255,255)`, colocado na segunda posição.

Caso o usuário não tenha decidido por uma imagem-destino em tons de cinza, a palheta-destino é uma cópia da palheta da imagem original.

5.2.1.2 Imagem-destino de 4 *bits*/*pixel*

Se o usuário decidiu-se pela conversão da imagem para tons de cinza, a palheta-destino é preenchida com uma subdivisão uniforme dos níveis de 0 a 255.

Para uma imagem-destino em tons de cor, a palheta-destino é preenchida com a palheta-padrão da placa VGA [Thom88] no ambiente *MS-Windows*. Os motivos para esta escolha são:

- aumento da performance do procedimento de apresentação da imagem quando a placa de vídeo do sistema é uma placa VGA;

- boa distribuição no espaço de cor RGB da palheta-padrão para placas VGA;

- o fato de que a palheta-padrão VGA é assumida para imagens de 4 *bits*/*pixel* em alguns programas de pintura e processamento de imagem populares.

Portanto, o valor de cada componente das cores na palheta-destino, nas duas opções possíveis para imagens-destino de 4 *bits*/*pixel*, é o especificado na Tabela 5.1.

	Tons de cinza				Tons de cor			
Índice	R	G	B		R	G	B	Cor
0	0	0	0		0	0	0	preto
1	17	17	17		0	0	128	azul-escuro
2	34	34	34		0	128	0	verde-escuro
3	51	51	51		0	128	128	ciano-escuro
4	68	68	68		128	0	0	vermelho-escuro
5	85	85	85		128	0	128	magenta-escuro
6	102	102	102		128	128	0	marrom
7	119	119	119		128	128	128	cinza-escuro
8	136	136	136		192	192	192	cinza-claro
9	153	153	153		0	0	255	azul-claro
0	170	170	170		0	255	0	verde-claro
11	187	187	187		0	255	255	ciano-claro
12	204	204	204		255	0	0	vermelho-claro
13	221	221	221		255	0	255	magenta-claro
14	238	238	238		255	255	0	amarelo
15	255	255	255		255	255	255	branco

Tabela 5.1 Valores da palheta para imagem-destino de 4 *bits*/*pixel*

A escassez de níveis usando-se apenas 4 *bits*/*pixel* pode acentuar os erros devidos ao efeito gama, um efeito referente à sensibilidade logarítmica da visão humana a variações na intensidade de uma fonte luminosa [FVFH90, Harr87, Roge85].

5.2.1.3 Imagem-destino de 8 *bits*/*pixel*

Para uma imagem-destino em tons de cinza, a palheta-destino é preenchida com uma subdivisão uniforme dos níveis de 0 a 255 (Figura 5.2).

```
// Imagem-destino em tons de cinza

PALETTEENTRY pePal;
for (int i = 0; i < 256; i++) {
    pePal.peRed   = (BYTE) i;
    pePal.peGreen = (BYTE) i;
    pePal.peBlue  = (BYTE) i;
    pePal.peFlags = 0;
    ImgDestino.SetPaletteEntries(i, 1, &pePal);
}
```

Figura 5.2 Formação de palheta-destino uniforme de 8 *bits*/*pixel* em tons de cinza

Para uma imagem-destino em tons de cor, a palheta-destino é preenchida com um subconjunto de cores uniformemente distribuídas no espaço RGB, particionando-se os 8 *bits*/*pixel* do índice na palheta entre as primárias R, G e B. São usados 3 *bits* para o componente verde (G), 3 *bits* para o vermelho (R) e 2 *bits* para o azul (B). Tal particionamento deve-se ao fato de que alterações no componente azul tem menor efeito proporcional na luminosidade de uma cor especificada no sistema RGB [FVFH90, Harr87] (Figura 5.3).

```
// Imagem-destino em tons de cor

PALETTEENTRY pePal;
BYTE r[8] = { 0, 36, 73, 109, 146, 182, 219, 255 };
BYTE g[8] = { 0, 36, 73, 109, 146, 182, 219, 255 };
BYTE b[4] = { 0, 73, 182, 255 };

for (int i = 0; i < 256; i++) {
    pePal.peRed   = r[ (i & 0x1C) >> 2 ];
    pePal.peGreen = g[ (i & 0xE0) >> 5 ];
    pePal.peBlue  = b[ (i & 0x03) ];
    pePal.peFlags = 0;
    ImgDestino.SetPaletteEntries(i, 1, &pePal);
}
```

Figura 5.3 Formação de palheta-destino uniforme de 8 *bits*/*pixel* em tons de cor

A função `SetPaletteEntries`, do objeto `ImagemDestino`, é chamada repetidamente para colocar uma cor na palheta, na posição indicada por "i". A cor estará na posição de memória `&pePal` (endereço da variável `pePal`), sendo que os campos desta variável são preenchidos de acordo com a metodologia adequada a cada situação.

5.2.2 Tratamento do erro local

Num processo de conversão da informação espectral, o erro local em determinado *pixel* é dado pela diferença entre a cor do *pixel* na imagem original e a cor mais próxima disponível na palheta-destino (logo, no caso de cores especificadas no sistema RGB, o erro é um vetor tridimensional). Erros locais de quantização aparecerão sempre que a palheta-destino não for um superconjunto da usada na imagem original.

No procedimento de expansão da resolução espectral isto pode ocorrer, por exemplo, na conversão de uma imagem com palheta de tons de cinza e resolução espectral de 4 *bits*/*pixel* para uma imagem com palheta em tons de cor de 4 *bits*/*pixel* ou mesmo 8 *bits*/*pixel* (devido ao fato de que a palheta de tons de cor, em imagens com 8 *bits*/*pixel*, usa apenas 2 *bits* para o componente azul, existirão apenas quatro níveis de cinza "puros"). Nestes casos, alguns dos 16 níveis de cinza da imagem original podem vir a ser mapeados para "cores" na imagem-destino.

No sistema **PHOTOPIX**, atualmente, o erro local é simplesmente desprezado em todos os casos de expansão da resolução espectral. Outras abordagens possíveis seriam:

- o espalhamento do erro local, usando métodos como o de Floyd-Steinberg, Jarvis, Judice e Ninke ou Stucki [Ulic88];

- uso de *dithering* com aumento da resolução espacial da imagem-destino [Harr87, HeBa86];

- otimização da palheta-destino, por métodos análogos aos utilizados nos casos de redução da resolução espectral.

5.3 Redução da resolução espectral

O procedimento para redução da resolução espectral é executado quando o número de *bits*/*pixel* da imagem a ser gerada é menor que o da imagem original. Neste caso, devido às resoluções espectrais possíveis no sistema **PHOTOPIX**, a imagem-destino sempre possuirá uma palheta, com 1, 4 ou 8 *bits* significativos para o índice.

O procedimento utilizado na conversão da informação espectral é o já apresentado na Figura 5.1. Portanto, para lidar com os casos de redução da resolução espectral deve-se apenas acrescentar ao sistema métodos que permitam uma escolha adequada das cores que formarão a palheta-destino.

Quando o usuário decide pelo uso de uma palheta com tons de cinza para a imagem-destino, a formação da palheta-destino é efetuada usando-se a mesma metodologia anteriormente definida para o procedimento de expansão da resolução espectral (itens 5.2.1.1 a 5.2.1.3).

Resta atender aos casos em que o usuário determina a redução da informação espectral para uma palheta-destino com tons de cor, considerando também a possibilidade de tratamento dos erros locais. Três métodos são implementados no sistema **PHOTOPIX** para formar a palheta-destino com tons de cor, a partir de um imagem original com maior resolução espectral que a imagem-destino.

5.3.1 Palheta-destino uniforme

Quando a imagem-destino tem 4 ou 8 *bits*/*pixel* em tons de cor, a palheta-destino é formada do mesmo modo definido para o procedimento de expansão da resolução espectral (itens 5.2.1.2 e 5.2.1.3).

No caso de imagem-destino com 1 *bit*/*pixel*, a palheta é preenchida com o preto, RGB(0,0,0), em sua primeira posição, e o branco, RGB(255,255,255), na segunda posição. Esta é a distribuição uniforme de maior abrangência no espaço RGB, para apenas 1 *bit* de resolução espectral.

5.3.2 Algoritmo da popularidade

Para uma imagem-destino com uma palheta de K posições (K = 2, 16 ou 256, no sistema **PHOTOPIX**) o algoritmo da popularidade [Heck82] escolhe as K cores mais frequentes no histograma da imagem original para compor a palheta-destino.

A única particularidade a ser considerada neste algoritmo é a memória necessária para formar o histograma de imagens com 24 *bits*/*pixel*. Como, mesmo usando técnicas de memória virtual, ainda não é comum que se consiga disponibilidade de memória e um desempenho suficientemente bom para o cálculo de um histograma de 2^{24} valores; e considerando resultados experimentais que mostram ser desnecessário e, às vezes, até prejudicial considerar todos os *bits* dos componentes RGB de uma imagem que sofrerá redução da resolução espectral [Petz91b], são considerados, para um imagem original de 24 *bits*/*pixel*, apenas 15 *bits*, correspondentes aos 5 *bits* mais significativos de cada componente de cor no sistema RGB.

5.3.3 Algoritmo do corte da mediana

O algoritmo do corte da mediana [Hand91, Heck82], tenta compor uma palheta-destino onde cada cor representa igual número de *pixels* da imagem original. Para isto, procura-se inicialmente, por uma caixa no espaço espectral que seja de tamanho mínimo e suficiente para incluir todas as cores que ocorrem na imagem original. Prossegue-se com a subdivisão desta caixa em duas menores que contenham conjuntos de cores com, aproximadamente, a mesma popularidade na imagem original (corte da mediana).

Repete-se o procedimento de subdivisão do espaço espectral até que sejam obtidas K caixas, em número igual ao de posições disponíveis na palheta-destino (K = 2, 16 ou 256, no sistema **PHOTOPIX**). Cada posição da palheta-destino é ocupada pela média das cores em cada caixa. Todas a cores de uma caixa são mapeadas, na palheta-invertida, para a cor representante da caixa na palheta-destino.

5.3.4 Tratamento do erro local

Nos casos de redução da resolução espectral, o usuário tem a opção de escolher pelo espalhamento ou não do erro local para os *pixel* adjacentes, de acordo com o algoritmo de Floyd-Steinberg [NeSp81, Roge85, Ulic88].

O algoritmo de Floyd-Steinberg, em sua forma mais simples, distribui o erro local de acordo com pesos de 3/8, para o *pixel* à direita e para o na mesma coluna da linha seguinte, e 1/4, para o *pixel* à direita na linha seguinte (Figura 5.4). Com isto, tenta-se manter uma luminosidade global da imagem-destino próxima à da imagem original.

Tal método foi escolhido devido à facilidade de implementação e eficiência, pois possui as seguintes características:

- não demanda aumento da resolução espacial da imagem-destino;
- distribui o erro local em apenas uma "varredura" da imagem.

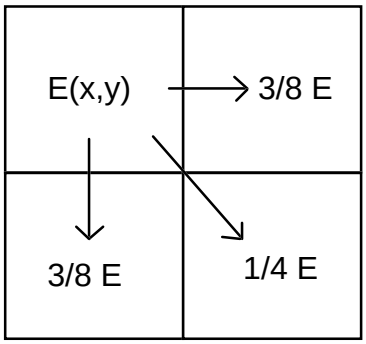

Figura 5.4 Distribuição do erro local pelo algoritmo de Floyd-Steinberg

Outros algoritmos para distribuição do erro local, como os de Jarvis, Judice e Ninke e o de Stucki [Ulic88], deverão ser implementados futuramente.

5.4 Conclusões

Para aumentar a portabilidade dos algoritmos de PDI implementados no sistema **PHOTOPIX**, foram implementadas subrotinas na classe das imagens (`CImage`) que permitem acesso à informação espectral de maneira padronizada, independente da resolução espectral da imagem a ser processada e do formato utilizado para armazenamento dos dados.

Dois agrupamentos para os algoritmos de conversão da informação espectral entre imagens digitais foram identificados:

- o dos procedimentos para redução da resolução espectral;
- o dos procedimentos para expansão da resolução espectral ou conversão da informação espectral entre imagens de mesma resolução.

Em ambos os casos, existem ainda as possibilidades de conversão para escala de tons de cinza e tratamento do erro local.

A expansão da resolução espectral ou conversão da informação espectral entre imagens de mesma resolução espectral é tratada por procedimentos que sempre convertem a informação espectral original para uma palheta destino uniformemente distribuída no espaço de cores RGB. Como há menor probabilidade de erro neste caso, em relação à situação de redução da resolução espectral, atualmente não está implementado no sistema **PHOTOPIX** nenhum tratamento para o erro local.

Para os casos de redução da resolução espectral, foram identificados três métodos de formação da palheta-destino:

- o uso de uma palheta-destino uniformemente distribuída no espaço RGB;

- o algoritmo da popularidade, que forma a palheta-destino com as cores mais populares na imagem original;

- o algoritmo do corte da mediana, que atua como um algoritmo da popularidade para pacotes de cores, repetidamente subdividindo o conjunto das cores presentes na imagem original em dois pedaços que representam, aproximadamente, o mesmo número de *pixels*.

Foi pesquisado um método para acelerar a busca da cor presente na palheta-destino que mais aproxima-se de uma cor desejada. O mecanismo encontrado é a formação de uma tabela que relaciona cores puras a índices na palheta-destino, sendo tal tabela denominada palheta-invertida.

Capítulo 6
Conclusão

Devido à imperfeição das plataformas de *hardware* da atualidade, um grande número de obstáculos ainda é imposto ao desenvolvedor de aplicativos, dispersando esforços que poderiam concentrar-se no aumento das funcionalidades presentes no produto oferecido ao usuário final.

O sistema **PHOTOPIX** foi desenvolvido para servir como base, conceitual e prática, para sistemas de processamento digital de imagens (PDI). Sua especificação segue os princípios da Análise Essencial e sua arquitetura interna baseia-se no paradigma da programação orientada para objetos.

Para a codificação do sistema **PHOTOPIX**, mesmo com o uso de uma API (*MS-Windows*) que minimiza as referências ao *hardware*, diversos obstáculos tiveram de ser superados, alguns devido a restrições específicas da plataforma de *hardware* utilizada, como a memória segmentada [Nels88, Petz92, Pros91] e a limitação da resolução espectral dos dispositivos de apresentação disponíveis [Durr87, MSDK92, Thom88].

Superar tais obstáculos exigiu um enorme volume de conhecimento, necessário para permitir uma codificação eficiente, auto-documentada e fiel aos objetivos do projeto. Esta barreira é geralmente citada como a curva de aprendizado (*learning curve* [Petz90]).

Diversas curvas de aprendizado tiveram de ser percorridas para a implementação prática do sistema **PHOTOPIX**:

- o conhecimento das funcionalidades e do mecanismo de operação e desenvolvimento de programas para o ambiente *MS-Windows*;

- o uso da programação orientada para objetos;

- a sintaxe e semântica da linguagem C++;

- o uso da biblioteca de classes MFC.

Considera-se que o resultado final obtido cumpre os objetivos do trabalho, pois nenhuma das curvas de aprendizado investigadas durante a implementação do sistema deverá ser obrigatoriamente percorrida por aqueles que irão ampliar suas funcionalidades. A princípio, qualquer código-fonte de um algoritmo de PDI, implementado na linguagem C, pode ser facilmente adaptado para o sistema **PHOTOPIX**.

O próprio sistema **PHOTOPIX** já pode se beneficiar do uso de tecnologias que facilitam a portabilidade de aplicativos. Uma nova versão da API utilizada está sendo desenvolvida, pelo seu fornecedor, para operar como um sistema operacional completo, tanto em microcomputadores como em *workstations*. O sistema **PHOTOPIX** compila e executa sem qualquer modificação nesta nova versão da API, denominada *Windows NT* (*New Technology*) [Cust92, Yao91, Yao92].

Durante o desenvolvimento do sistema **PHOTOPIX** foi testado e comprovado o isolamento dos algoritmos de PDI do código relativo ao tratamento da interface aplicativo-usuário. A execução dos algoritmos já implementados foi testada com uma interface de linha de comando, codificada para o sistema operacional MS-DOS [Dunc86], sendo criados e executados objetos das classes que implementam algoritmos de PDI.

O isolamento entre as classes desenvolvidas permite uma rápida detecção e correção de erros, ou a mudança nos métodos utilizados para implementar determinada funcionalidade. Futuramente, pode ser necessário mudar o formato internamente utilizado para representar a informação sobre as imagens. Contudo, nenhuma alteração será necessária no código dos algoritmos de PDI, sendo preciso apenas a adaptação de subrotinas na classe das imagens, que acessam sua informação espectral.

O sistema **PHOTOPIX** define uma API específica para a codificação de algoritmos de processamento digital de imagens. Tal API é constituída pelas funções públicas da classe `CImage` (Capítulo 4, Apêndice A.2.4), sendo denominada DIPAPI (*Digital Image Processing Algorithms Programming Interface*).

Apêndices

Apêndice A

Codificação do sistema PHOTOPIX na linguagem C++

A.1 Estrutura do código-fonte do sistema PHOTOPIX

A.1.1 Arquivos de cabeçalho (*Headers*)

`PPix.H`: Este arquivo contém a declaração das classes `CPPixApp` e `CPPixMainWnd`. A classe `CPPixApp` implementa as funcionalidades necessárias à operação do sistema **PHOTOPIX** no ambiente *MS-Windows*, enquanto a classe `CPPixMainWnd` implementa as funcionalidades da janela principal de um aplicativo MDI.

`PPixDefs.H`: Este arquivo contém definições de constantes necessárias para estabelecer a conexão entre a lógica e a aparência do sistema **PHOTOPIX** (Figura 2.4).

`PPixDlgs.H`: Este arquivo contém a declaração das classes para os quadros de diálogo do sistema **PHOTOPIX**.

`PPixGlob.H`: Este arquivo contém a declaração de variáveis, constantes e macros globais do sistema **PHOTOPIX**.

`PPixImg.H`: Este arquivo contém a declaração da classe `CImage`, a classe de imagens da memória essencial do sistema **PHOTOPIX**.

`PPixView.H`: Este arquivo contém a declaração da classe `CPPixViewWnd`, que implementa a funcionalidade de uma janela-filha de aplicativos MDI. Na área de trabalho de janelas da classe `CPPixViewWnd` são visualizadas as imagens da memória essencial do sistema **PHOTOPIX**.

`PPix_DIP.H`: Este arquivo contém a declaração da classe `CDIPAlgorithm` e outras classes derivadas desta, onde são implementados algoritmos de PDI.

`PPix_IO.H`: Este arquivo contém a declaração da classe `CConverter` e outras classes derivadas desta, onde são implementados conversores entre a representação interna das imagens e formatos para arquivos contendo imagens.

A.1.2 Arquivos de implementação das funcionalidades

PPix.CPP: Neste arquivo são implementadas as funcionalidades das classes `CPPixApp` e `CPPixMainWnd`.

PPixDlgs.CPP: Neste arquivo são implementadas as funcionalidades de todos os quadros de diálogo do sistema **PHOTOPIX**.

PPixImg.CPP: Neste arquivo são implementadas as funcionalidades da classe `CImage`.

PPixView.CPP: Neste arquivo são implementadas as funcionalidades da classe `CPPixViewWnd`.

PPix_DIP.CPP: Neste arquivo são implementados os algoritmos de PDI, nas classes derivadas de `CDIPAlgorithm`.

PPix_IO.CPP: Neste arquivo são implementados os procedimentos para tranferência de imagens de arquivos externos para a memória essencial, e vice-versa.

A.1.3 Arquivos auxiliares

PPixHead.H e **Dummy.CPP**: Tais arquivos são utilizados para acelerar o processo de compilação, através do uso de arquivos de cabeçalho pré-compilados (*precompiled headers* [MsC+91]).

PPix.DEF: Este arquivo contém definições necessárias para o processo de ligação do código-objeto do sistema **PHOTOPIX** às bibliotecas de código-objeto (*link* [MsC+91]).

PPix.DLG: Neste arquivo estão definidas as *templates* dos quadros de diálogo do sistema **PHOTOPIX**.

PPix.MAK: Este arquivo contém a definição dos passos necessários para transformar o código-fonte do sistema **PHOTOPIX** em um programa executável. Este arquivo foi gerado automaticamente, sendo mantido pelo ambiente de desenvolvimento utilizado para implementação do sistema **PHOTOPIX** (*Programmer's WorkBench* [MsC+91]).

PPix.RC: Este arquivo contém a definição de menus, tabelas de strings, tabelas de teclas de atalho (*shortcuts* ou *accelerators*), *bitmaps* e outros recursos a serem utilizados pelo sistema **PHOTOPIX**.

PPixHelp.RTF e **PPix.HPJ**: Este arquivos destinam-se à elaboração do arquivo de auxílio do sistema **PHOTOPIX** (*Help Compiler* [MsC+91]).

A.2 Declaração das classes do sistema PHOTOPIX

A apresentação que é feita neste tópico das classes definidas no sistema **PHOTOPIX** tem a finalidade de ilustrar a metodologia de definicão de nomes para os membros de cada classe.

A documentação do significado e tipo dos parâmetros de cada função deve ser obtida no único lugar onde estará sempre atualizada: no código-fonte.

A.2.1 Classe CPPixApp

```
class CPPixApp : public CWinApp
{
public:
    CPPixApp( const char* pszAppName = NULL )
    : CWinApp( pszAppName )
    {
    }

    virtual BOOL InitInstance();

}; // class CPPixApp
```

A.2.2 Classe `CPPixMainWnd`

```
class CPPixMainWnd : public CMDIFrameWnd
{
  private:    // Omitido

  public:

    CPPixMainWnd();    // Constructor

    // message handlers
    afx_msg void OnGetMinMaxInfo( LPPOINT );
    afx_msg void OnSize(UINT, int, int);
    afx_msg BOOL OnQueryNewPalette();
    afx_msg void OnPaletteChanged(CWnd *);

    // File Menu
    afx_msg void OnImgNew();
    afx_msg void OnImgOpen();
    afx_msg void OnImgSave();
    afx_msg void OnImgSaveAs();
    afx_msg void OnImgPrint();
    afx_msg void OnPrinterSetup();
    afx_msg void OnImgClose();

    afx_msg void OnClose();
    afx_msg BOOL OnQueryEndSession();
    afx_msg void OnEndSession( BOOL );

    // Edit Menu
    afx_msg void OnImgUndo();
    afx_msg void OnImgCut();
    afx_msg void OnImgCopy();
    afx_msg void OnImgPaste();

    // View Menu
    afx_msg void OnViewNew();
    afx_msg void OnViewStretch();
    afx_msg void OnViewScroll();
    afx_msg void OnViewZoomIn();
    afx_msg void OnViewZoomOut();
    afx_msg void OnViewToolbox();

    // Image Menu
    afx_msg void OnImgInfo();
    afx_msg void OnImgConvert();
    afx_msg void OnImgPalette();
    afx_msg void OnDIPAlgorithm();

    // Options Menu
    afx_msg void OnLanguageChange();

    // Window Menu
    afx_msg void OnCloseAll();

    // Help Menu
    afx_msg void OnAbout();
    afx_msg void OnHelp();
```

```
    // custom message handlers
    afx_msg LONG OnImgDestroy(UINT, LONG);
    afx_msg LONG OnViewActivate(UINT, LONG);
    afx_msg LONG OnToolBoxClosed(UINT, LONG);

    // Overloaded functions
    afx_msg int OnCreate(LPCREATESTRUCT);

    // Member functions
    void ChangeMenu(UINT, UINT);
    void MaintMenu();

    DECLARE_MESSAGE_MAP()

}; // class CPPixMainWnd
```

A.2.3 Classe `CPPixViewWnd`

```
class CPPixViewWnd : public CMDIChildWnd
{
  private:    // Omitido

  public:

    CPPixViewWnd();    // Constructor

    // message handlers
    afx_msg void OnClose();
    afx_msg int  OnCreate(LPCREATESTRUCT );
    afx_msg void OnMDIActivate(BOOL, CWnd*, CWnd*);
    afx_msg void OnPaint();

    // custom message handlers
    afx_msg LONG OnViewQueryNewPalette(UINT, LONG);
    afx_msg LONG OnViewPaletteChanged(UINT, LONG);

    // Overloaded functions
    BOOL Create(LPCSTR, LONG, RECT& CMDIFrameWnd*);
    afx_msg void OnSize(UINT, int, int);
    afx_msg void OnHScroll( UINT, UINT, CScrollBar* );
    afx_msg void OnVScroll( UINT, UINT, CScrollBar* );
    afx_msg void OnKeyDown( UINT, UINT, UINT );
    afx_msg void OnMouseMove(UINT, CPoint);

    // Member functions
    void UpdateZoom();
    void ViewStretch();
    void ViewScroll();
    void ViewZoomIn();
    void ViewZoomOut();
    void Info();  // Show Image Info in Modal Dialog

    DECLARE_MESSAGE_MAP()

}; // class CPPixViewWnd
```

A.2.4 Classe CImage

```
class CImage : public CObject
{
  private:    // Omitido

  public:

    CImage();    // Constructor
    ~CImage();    // Destructor

    // Member functions

    WORD  GetBitsPerPixel() const;
    DWORD GetHeight() const;
    WORD  GetNearestPaletteIndex(DWORD dwColor) const;
    WORD  GetPaletteEntries(WORD wStartIndex,
                            WORD wNumEntries,
                  PALETTEENTRY* pPaletteColors) const;
    WORD  GetPaletteSize() const;
    DWORD GetPixel(DWORD x, DWORD y) const;
    DWORD GetWidth() const;
    BOOL  IsPaletted() const;
    WORD  SetPaletteEntries(WORD wStartIndex,
                            WORD wNumEntries,
                  PALETTEENTRY* pPaletteColors);
    DWORD SetPixel(DWORD x, DWORD y, DWORD dwColor);

}; // class CImage
```

A.2.5 Classe CDIPAlgorithm

```
class CDIPAlgorithm : public CObject
{
public:

   // Pure virtual member functions

   virtual BOOL GetName(CString& DIPAlgorithmName) = 0;
   virtual BOOL GetParameters() = 0;
   virtual BOOL Execute() = 0;


   // The derived class must also implement
   // static BOOL CanProcessBitsPerPixel(WORD wBits);

   // The derived class constructor must receive a
   // point to an CImage object as parameter

}; // class CDIPAlgorithm
```

A.2.6 Classe CConverter

```
class CConverter : public CObject
{
public:

   // Pure virtual member functions

   virtual BOOL InputImage() = 0;
   virtual BOOL OutputImage(CImage* pImg) = 0;

}; // class CConverter
```

A.3 Metodologia para extensão do sistema PHOTOPIX

O sistema **PHOTOPIX** pode ter suas funcionalidades ampliadas por qualquer programador com proficiência na linguagem **C**. Deve-se observar porém que apenas um razoável conhecimento da API do ambiente *MS-Windows* e uma análise do código-fonte do sistema **PHOTOPIX** permitirá manter a interface aplicativo-usuário consistente. A implementação de novos quadros de diálogo para o sistema exige também o conhecimento das ferramentas de desenvolvimento [MsC+91].

Os seguintes passos deverão ser seguidos para implementação de um algoritmo que atue sobre a imagem ativa:

- declarar uma classe derivada de `CDIPAlgorithm` no arquivo `PPix_DIP.H`. Esta classe deverá ter uma implementação própria das funções `CanProcessBitsPerPixel()`, `GetName()`, `GetParameters()` e `Execute()`. O construtor deve receber como parâmetro um ponteiro para a imagem ativa;

- codificar o algoritmo no arquivo `PPix_DIP.CPP`, na função `Execute()` da classe declarada no passo anterior. Devem ser utilizadas para acesso à imagem apenas funções públicas do objeto `CImage`;

- definir um novo identificador de algoritmo no arquivo `PPixDefs.H`;

- definir itens de menu que retornem este identificador, no arquivo `PPix.RC`. Deve-se atentar para o fato de que o novo item deve ser adicionado nos menus para todos os idiomas suportados pelo sistema;

- direcionar a mensagem de menu gerada pelo item com o identificador do novo algoritmo à subrotina `OnDIPAlgorithm()` da classe `CPPixMainWnd`. Tal alteração deve ser feita no mapa de mensagens do arquivo `PPix.CPP`;

- adicionar um novo `case` ao `switch` da subrotina `MaintMenu()`, no arquivo `PPix.CPP`. Neste novo `case` deve ser invocada a função `CanProcessBitsPerPixel()` da classe do novo algoritmo;

- adicionar um novo `case` ao `switch` da subrotina `OnDIPAlgorithm()`, no arquivo `PPix.CPP`. Neste novo `case` deve ser criado um objeto da classe definida para o novo algoritmo.

Após a recompilação do sistema o novo algoritmo já estará disponível. Quando o usuário aciona o item do novo algoritmo no menu, é enviada à janela principal do aplicativo uma mensagem de comando de menu. Nesta mensagem, está identificado o item de menu, que terá o identificador do novo algoritmo. Pelo mapa de mensagens, será acionada a subrotina `OnDIPAlgorithm()` da classe `CPPixMainWnd`. Nesta subrotina, é criado um objeto da classe do novo algoritmo, sendo em seguida chamadas suas funções virtuais `GetParameters()` e `Execute()`.

Futuramente, pretende-se separar o código-objeto dos algoritmos de PDI do código-objeto que trata da interface aplicativo-usuário, usando o mecanismo de DLL (*dynamic link library*) [NoYa90, Petz90].

Apêndice B

Formato interno das imagens

As imagens são armazenadas na classe `CImage` em um formato denominado *Packed-DIB* [MSDK92, Petz91a].

Este formato foi escolhido por ser um dos formatos padronizados para troca de dados no ambiente *MS-Windows*, via *clipboard*. Além disto, é possível associar um contexto de dispositivo especial a um *Packed-DIB*, permitindo a execução de qualquer primitiva gráfica da API sobre sua área.

No formato *packed-DIB* são armazenadas em posições de memória contíguas:

- um cabeçalho contendo dados necessários para interpretar corretamente os *bits* de informação espectral sobre cada *pixel* da imagem. Tal cabeçalho é uma estrutura do tipo `BITMAPINFOHEADER`, sendo de tamanho variável, pois alguns campos podem não estar presentes;

- a palheta da imagem, se houver, armazenada em estruturas contíguas do tipo `RGBQUAD`;

- os *bits* de informação espectral sobre cada *pixel*, armazenados a partir da linha inferior da imagem, da esquerda para a direita. Cada linha deve conter um número de *bytes* múltiplo de 4. Caso os número de *bits*/*pixel* multiplicado pelo número de *pixels* em uma linha não resulte em um número de *bytes* múltiplo de 4, os *bits* e *bytes* restantes até o número de *bytes* múltiplo de 4 mais próximo devem ser ignorados.

Imagens com resolução espectral de 24 *bits*/*pixel* não contém o campo destinado à palheta. Neste caso, a cor de cada *pixel* é armazenada em três *bytes* adjacentes, contendo respectivamente a componente azul, verde e vermelha.

Estrutura BITMAPINFOHEADER

```
typedef struct
{
DWORD biSize;          // Tamanho da estrutura
DWORD biWidth;         // Largura da imagem em pixels
DWORD biHeight;        // Altura da imagem em pixels
WORD  biPlanes;        // Colocar 1
WORD  biBitCount;      // Bits de cor por pixel
DWORD biCompression;   // Compressão utilizada
DWORD biSizeImage;     // Número de bytes no bitmap
DWORD biXPelsPerMeter; // Resolução horizontal
DWORD biYPelsPerMeter; // Resolução vertical
DWORD biClrUsed;       // Número de cores utilizadas
DWORD biClrImportant;  // Cores importantes
}
BITMAPINFOHEADER;
```

Estrutura RGBQUAD

```
typedef struct
{
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved; // Colocar 0
}
RGBQUAD;
```

Bibliografia

AnAn91 ANDERSON, G. & ANDERSON, P., Moving on to C++, *UnixWorld*, Jan. 1991

Amar92 AMARAL, L.M., *Interface Homem-Máquina do Ambiente de Definição de Semântica LDS*, Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, 1992

CASE92 CASEWORKS, INC., CASE:W User's manual, *CASEWORKS, Inc.*, 1992

Chri92 CHRISTIAN, K., *The Microsoft Guide to C++ Programming*, Microsoft Press, 1992

CRTW88 COOK, R.L., RAWSON III, F.L., TUNKEL, J.A. & WILLIAMS, R.L., Writing an Operating System/2 application, *IBM Systems Journal*, Vol. 27, No. 2, 1988

Cox86 COX, B.J., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Inc., 1986

Cust92 CUSTER, H., A grand tour of Windows NT, *Microsoft Systems Journal*, Jul-Aug 1992

DeSt89 DEWHURST, S.C. & STARK, K.T., *Programming in C++*, Prentice-Hall, Inc., 1989

Dunc86 DUNCAN, R., *Advanced MS-DOS*, Microsoft Press, 1986

Durr87 DURRETT, H.J., *Color and the Computer*, Academic Press, Inc., 1987

Frag86 FRAGOMENI, A.H., *Dicionário Enciclopédico de Informática*, Editora Campus, 1986

FVFH90 FOLEY, J.D., VAN DAM, A., FEINER, S.K. & HUGHES, J.F., *Computer Graphics: Principles and Practice, 2nd. Ed.*, Addison-Wesley, Inc., 1990

Hand91 HANDMADE SOFTWARE, INC., *Image Alchemy version 1.4 - User's Manual*, 1991

Harr87 HARRINGTON, S., *Computer Graphics - A Programming Approach, 2nd. Ed.*, McGraw-Hill, Inc., 1987

HDF+86 HOPGOOD, F.R.A., DUCE, D.A., FIELDING, E.V.C., ROBINSON, K. & WILLIAMS, A.S., *Methodology of Window Management*, Springer-Verlag, 1986

HeBa86 HEARN, D. & BAKER, M.P., *Computer Graphics*, Prentice-Hall, Inc., 1986

Heck82 HECKBERT, P., Color image quantization for frame buffer display, *Computer Graphics*, Vol. 16, No. 3, Jul. 1982

IBM89 IBM CORPORATION, *Common User Access: Advanced Interface Design Guide*, IBM Co. (SC26-4582-0), 1989

Jain89 JAIN, A.K., *Fundamentals of Digital Image Processing*, Prentice-Hall, Inc., 1989

Joyn92 JOYNER, I., *A C++ Critique*, e-mail correspondence, 1992

KoRa88 KOGAN, M.S. & RAWSON III, F.L., The design of Operating System/2, *IBM Systems Journal*, Vol. 27, No. 2, 1988

McPa84 MCMENAMIN, S.M. & PALMER, J.F., *Essential Systems Analysis*, Prentice-Hall, Inc., 1984

MDTG92a MICROSOFT DEVELOPER TECHNOLOGY GROUP, DIBs and their use, *Microsoft Developer's Network CD-ROM*, Mar. 1992

MDTG92b MICROSOFT DEVELOPER TECHNOLOGY GROUP, Using True Color Devices, *Microsoft Developer's Network CD-ROM*, Mar. 1992

MDTG92c MICROSOFT DEVELOPER TECHNOLOGY GROUP, Using DIBs with palettes, *Microsoft Developer's Network CD-ROM*, Mar. 1992

MSDK92 MICROSOFT CO., *MS-Windows 3.1 Software Development Kit*, Microsoft Press, 1992

Msft91a MICROSOFT CO., *Microsoft Windows Multimedia Programmer's Reference*, Microsoft Press, 1991

Msft91b MICROSOFT CO., *Visual Basic Programmer's Guide*, Microsoft Co., 1991

MsC+91 MICROSOFT CO., *Microsoft C/C++ 7.0 compiler manuals*, Microsoft Co., 1991

Msft92a MICROSOFT CO., *Microsoft Windows 3.1 - User's Guide*, Microsoft Co., 1992

Msft92b MICROSOFT CO., *The Windows Interface - An Application Design Guide*, Microsoft Press, 1992

Myer88 MYERS, B.A., A Taxonomy of Window Manager User Interfaces, *IEEE Computer Graphics & Applications*, Sep. 1988

Nels88 NELSON, R.P., *The 80386/80468 Programming Guide*, Microsoft Press, 1988

NeSp81 NEWMAN, W.M. & SPROULL, R.F., *Principles of Interactive Computer Graphics, 2nd. Ed.*, McGraw-Hill, Inc., 1981

Nibl86 NIBLACK, W., *An Introduction to Digital Image Processing*, Prentice-Hall, Inc., 1986

NoYa90 NORTON, P. & YAO, P., *Windows 3.0 Power Programming Techniques*, Bantam Books, 1990

Nune90 NUNES, J.R.S., *Introdução aos sistemas operacionais*, Livros Técnicos e Científicos Editora, 1990

NyOR90 NYE, A. & O'REILLY, T., *X Toolkit Intrinsics Programming Manual - OSF/Motif Edition*, O'Reilly & Associates, 1990

Othm92 OTHMER, K., Inside QuickDraw, *BYTE*, Vol. 17, No. 9, Sep. 1992

Perk92 PERKINS, C.L., Getting with the program, *Nextworld*, Summer 1992

Petz90 PETZOLD, C., *Programming Windows, 2nd. Ed.*, Microsoft Press, 1990

Petz91a PETZOLD, C., Preserving a Device-Independent Bitmap: The Packed-DIB Format, *PC Magazine*, Vol. 10, No. 13, Jul. 1991

Petz91b PETZOLD, C., Working with 24-bit Color Bitmaps For Windows, *PC Magazine*, Vol. 11, No. 15, Sep. 1991

Petz92 PETZOLD, C., The case for 32 bits, *Microsoft Systems Journal*, Jul-Aug. 1992

Prat78 PRATT, W., *Digital Image Processing*, John Wiley & Sons, 1978

Pros91 PROSISE, J., Segmented Memory, *PC Magazine*, Vol. 10, No. 6, Mar. 1991

RaSc90 RABINOWITZ, H. & SCHAAP, C., *Portable C*, Prentice-Hall, Inc., 1990

Roge85 ROGERS, D.F., *Procedural Elements for Computer Graphics*, McGraw-Hill, Inc., 1985

ScGe86 SCHEIFLER, R.W. & GETTYS, J., The X Window system, *ACM Trans. on Graphics*, Vol. 5, No. 2, Apr. 1986

Staa87 STAA, A.V., *Engenharia de programas*, Livros Técnicos e Científicos Editora, Ltda, 1987

Stev90 STEVENS, R.T., *Fractal Programming and Ray Tracing with C++*, M&T Books, 1990

Sun90 SUN MICROSYSTEMS, INC., *Writing Applications for Sun Systems - A Guide for Macintosh Programmers*, Addison-Wesley, Inc., 1990

Tane87 TANENBAUM, A.S., *Operating Systems Design and Implementation*, Prentice-Hall, Inc., 1987

Thom88 THOMPSON, S., VGA - Design choices for a new video subsystem, *IBM Systems Journal*, Vol. 27, No. 2, 1988

Ulic88 ULICHNEY, R., *Digital Halftoning*, MIT Press, 1988

Will87 WILLIAMS, G., HyperCard, *BYTE*, Vol. 12, No. 14, Dec. 1987

Yao91 YAO, P., Windows 32-bit API gives developers advanced operating system capabilities, *Microsoft Systems Journal*, Nov-Dec. 1991

Yao92 YAO, P., An introduction to Windows NT memory management fundamentals, *Microsoft Systems Journal*, Jul-Aug. 1992

Your90 YOURDON, E., *Análise Estruturada Moderna*, Editora Campus, 1990