Ferramentas de suporte à construção de aplicações de segmentação de imagens digitais

Gabriel Lühers Graça
Carla Maria Dal Sasso Freitas

Instituto de Informática
Universidade Federal do Rio Grande do Sul
{graca,carla}@inf.ufrgs.br

Resumo

O desenvolvimento de aplicações e técnicas de processamento de imagens envolve experimentar com algoritmos e parâmetros de execução. Então, os atuais ambientes de desenvolvimento oferecem ferramentas para programador e usuário em diferentes níveis de abstração. No entanto, esses são, na sua maioria, orientados ao desenvolvimento de algoritmos. Métodos automáticos não são práticos para muitas aplicações, por exemplo para a segmentação de imagens, e assim técnicas interativas são necessárias e estão sendo desenvolvidas. Este trabalho descreve um *framework* orientado a objetos para procedimentos interativos que encapsula funções comuns a aplicações de segmentação e também suporta processamento de imagens algorítmico em geral.
PALAVRAS-CHAVE: *Framework* Orientado a Objetos, Sistemas de Processamento de Imagens, Segmentação.

Abstract

The development of image processing techniques and applications involves experimenting with algorithms and execution parameters. Thus, current development environments provide tools for programmer and user at different abstraction levels. However, most software tools allow only for the development of algorithms. Automatic methods, such as for image segmentation, are currently not practical for many applications and in order to provide acceptable solutions interactive procedures are being developed. This work describes an object-oriented framework for interactive procedures that encapsulates common features in segmentation applications and also supports (algorithmic) image processing in general.
KEYWORDS: Object-Oriented Framework, Image Processing Systems, Segmentation.

1 Introdução

A pesquisa em processamento de imagens, segundo Gonzalez e Woods [GON92], tem sido estimulada por interesse em duas áreas de aplicação: o processamento de informação pictórica para a interpretação humana e a análise de imagens para a visão de sistemas autônomos. A área de processamento de imagens está fortemente associada a algoritmos de transformação para a restauração de imagens com defeitos de aquisição ou transmissão e para o realce de imagens, a fim de facilitar a interpretação humana.

O desenvolvimento de técnicas para o processamento de imagens é uma tarefa essencialmente experimental. Diferentes algoritmos com diferentes parâmetros devem ser testados até que se encontrem resultados satisfatórios no universo de imagens em estudo. Por isso, pacotes de processamento de imagens comumente oferecem a possibilidade de execução de operações sobre imagens com entrada de parâmetros através de *interfaces* gráficas ou linguagens de comandos.

Técnicas de segmentação de imagens são aplicadas em muitas áreas de pesquisa e da indústria como análise de imagens médicas, cartografia, sensoriamento remoto, automação industrial e robótica. A segmentação de uma imagem se dá pela identificação de características que permitam a sua decomposição nos seus elementos constituintes, conforme Jain [JAI89]. A visualização de imagens digitais, em especial imagens médicas, deve ser precedida de uma segmentação adequada já que a derivação de dados como formas e dimensões sobre estruturas presentes na imagem depende do sucesso dessa técnica.

Os algoritmos de segmentação costumam ser computacionalmente exigentes, podendo ser manuais ou automáticos. Para muitas aplicações, a segmentação automática é (atualmente) inviável. Em especial, em imagens médicas, o baixo contraste e a alta taxa de ruído significam que processos automáticos dificilmente geram resultados satisfatórios. A segmentação manual, por outro lado, exige mão-de-obra altamente especializada e consome muito tempo, tornando-a igualmente inviável. A solução é aliar em processos interativos o alto poder computacional da atual geração de computadores à grande capacidade de reconhecimento de padrões dos humanos. Na segmentação interativa, assim como definem Olabarriaga e Smeulders [OLA97a], o usuário pode dinamicamente interferir no processo ajustando os parâmetros da técnica de segmentação e fornecendo ou modificando informação pictórica. Um valor de *threshold* ou pontos sobre um contorno são exemplos de parâmetros para um algoritmo de segmentação.

As ferramentas interativas de segmentação devem ser construídas de forma a maximizar a produtividade do usuário e minimizar a ocorrência de erros. Esses dois fatores estão ligados tanto ao algoritmo per se como à realimentação visual fornecida pelo sistema. No entanto, nos sistemas de processamento de imagens atualmente em uso, os recursos de interação são mínimos. Usualmente, o usuário escolhe uma operação, especifica seus parâmetros e o sistema a executa. Se o resultado não o satisfaz, seu único recurso é aplicar novamente a operação com outros parâmetros ou escolher outra.

Com isso, fica evidente que ferramentas de suporte à construção de aplicações de segmentação de imagens digitais, tanto manuais como interativas, devem oferecer recursos mais sofisticados que aqueles atualmente encontrados em pacotes de processamento de imagens.

Dada a importância do processo de segmentação para a qualidade da imagem resultante (qualidade, aqui em termos de fidelidade das estruturas na imagem em relação às estruturas dos objetos ali representados), essa é uma área de pesquisa com muitos problemas ainda sem solução satisfatória. Uma deficiência é a falta de ferramentas de suporte que permitam a construção de novas técnicas, interativas ou não, de forma estruturada e consistente.

Então, ferramentas de suporte à construção de aplicações de segmentação devem fornecer recursos básicos, como as funções de processamento de imagens, e funções de mais alto nível para gerenciamento de arquivos e de *interface*. Além disso, devem oferecer alguns recursos para suprir as necessidades específicas de aplicações de segmentação interativa. Algumas dessas necessidades foram identificadas por Olabarriaga et alli [OLA97b] como o suporte para um mecanismo de apontamento (*cursor*) de alto nível e ferramentas de visualização de imagens com facilidades tais como o uso de cores falsas e a combinação de imagens pelo uso de transparências. Essas ferramentas devem ser projetadas de maneira a permitir um alto grau de reutilização para facilitar a tarefa de prototipação de novas técnicas.

A partir da constatação das deficiências dos atuais sistemas de processamento de imagens perante as necessidades de técnicas de segmentação interativa, foi projetado e implementado um *framework* orientado a objetos que suporta tanto aplicações de segmentação como técnicas de processamento de imagens em geral.

O presente trabalho descreve o desenvolvimento desse *framework*, denominado Blue, apresentando a metodologia de projeto e a utilização de *design patterns*.

A próxima seção descreve as deficiências dos sistemas de processamento de imagens. As seções 3 e 4 apresentam a metodologia de projeto e a descrição do *framework* Blue, respectivamente. A última seção discute a validação do *framework* e delinea alternativas de continuidade do trabalho.

2 Modos de utilização e deficiências de sistemas de processamento de imagens

Sistemas interativos têm sempre ocupado um lugar de destaque em processamento de imagens, segundo Balen et alli [BAL94], porque são raras as soluções de aplicação geral e porque, como ressaltam Pope e Lowe [POP94], a área de análise de imagens, sendo uma disciplina que abrange a criação de sistemas artificiais, deve enfatizar a comparação e a avaliação de diferentes métodos. Assim, a tarefa de criação de novos sistemas ou a validação de novas técnicas a partir de componentes construídos por instituições ou pesquisadores distintos deveria ser simples.

Bengtsson et alli [BEN81] afirmam, ainda, que o desenvolvimento de algoritmos de processamento de imagens é uma tarefa altamente experimental. Diferentes algoritmos são aplicados sobre uma imagem, o resultado é verificado, os algoritmos são modificados e aplicados novamente até que um resultado aceitável seja encontrado. A grande variabilidade dos dados, como apontam Balen et alli [BAL94], implica a necessidade de avaliação visual dos resultados e a resposta imediata de sistemas interativos reduz o tempo de desenvolvimento de novos métodos.

Assim que um algoritmo adequado é encontrado, ele deve ser aplicado sobre uma grande variedade de imagens a fim de se obter estatísticas sobre seu desempenho. O pesquisador deve, portanto, freqüentemente alternar entre um modo de programação e um modo usuário. Por isso, pacotes de desenvolvimento comumente oferecem a possibilidade de execução de operações sobre imagens com entrada de parâmetros através de *interfaces* gráficas ou linguagens de comando. Sistemas com facilidades desse tipo são chamados de sistemas interativos. Alguns pacotes oferecem recursos mais sofisticados, como ferramentas gráficas de programação, mas se prestam apenas para o desenvolvimento de algoritmos que, por definição, não são interativos. Com isso, fica claro que o ambiente de desenvolvimento deve ser não somente interativo mas deve permitir diferentes graus de interação.

Balen et alli [BAL94] apontam que ambientes de processamento de imagens variam desde bibliotecas até sistemas dirigidos a comandos, a *menus* ou de programação visual e Vámos et alli [VAM81] argumentam que para um ambiente suportar eficientemente uma grande variedade de tarefas dentro de um domínio de aplicação deve haver um ponto de equilíbrio entre o suporte para tarefas específicas e o suporte à solução de problemas em geral. Aparentemente, a solução está na elaboração de uma metodologia geral e um conjunto de ferramentas para uma grande variedade de problemas, combinadas a um método simples de composição para a execução de tarefas específicas. Tanto o sistema proposto por Vámos et alli [VAM81] como o sistema Khoros [KON94] possuem quatro níveis de interação (usuário, programador visual, programador de ferramentas e programador de infraestrutura) que correspondem, de maneira geral, a esses modos de utilização.

Por outro lado, o desenvolvimento de aplicações de processamento de imagens muitas vezes se dá pela utilização de uma biblioteca (de classes ou de funções) dentro de um ambiente de desenvolvimento de uso geral. A integração entre sistema operacional, linguagem de programação e sistema de janelas (GUI-*Graphical User Interface*) já é característica comum a

inúmeros ambientes de desenvolvimento de aplicações. A maioria, no entanto, não se limita a um domínio específico de aplicação.

Na computação gráfica, estruturas de dados são utilizadas para descrever objetos que são exibidos de acordo com as ações do usuário sem, no entanto, serem diretamente manipuladas. Na análise de imagens, por outro lado, elas servem como descrições de alto nível para objetos ou estruturas em imagens. Nesse aspecto, a análise de imagens pode ser vista como redução de dados, ao transformar informação pictórica em informação vetorial. As descrições vetoriais devem, muitas vezes, ser exibidas em sobreposição às respectivas imagens e estar sujeitas à edição pelo usuário. Dessas estruturas pode-se mais facilmente extrair informações de geometria como formas e dimensões. No entanto, não existem pacotes com suporte para essas funções, exceção feita ao sistema Vista [POP94].

Um problema comum a todos os ambientes de desenvolvimento é o problema do controle do fluxo de execução, ou simplesmente "controle". O problema está em determinar se o controle é da *interface*, da aplicação ou do usuário e usualmente se resolve por funções de *callback*. Laffra e van den Bos [LAF91] apontam que o problema de controle está intimamente ligado à questão de execução seqüencial versus execução *multi-threaded*. De fato, Olabarriaga et alli [OLA97b] identificaram uma deficiência importante dos atuais sistemas de processamento de imagens: a impossibilidade de se interromper operações que geram resultados parciais significativos. Um exemplo disso é o crescimento de regiões (*region growing*): se a borda de um objeto não é fechada, o algoritmo "transborda", invadindo outras regiões da imagem. Assim, o usuário não pode interromper a operação no momento em que a área de seu interesse foi preenchida. Além disso, nenhum sistema prevê a interrupção de uma operação e sua retomada posterior.

O problema mais sério surge quando interações do tipo *point-and-click* são necessárias, como é o caso, por exemplo, do tipo de interação implementada em pacotes de desenho vetorial ou de pintura. A manipulação direta de objetos gráficos (retângulos, círculos, etc) e a indicação de regiões de interesse são duas tarefas que são difíceis de implementar com o modelo de *callback*. O programador acaba por implementar complicadas e longas cadeias de comandos condicionais aninhados para determinar o estado da aplicação e a ação adequada. Pode-se também utilizar bibliotecas de sistema, que possuem rotinas de baixo nível para operações de entrada e saída que, no entanto, resultam em código tão ou mais complexo.

Olabarriaga et alli [OLA97b] apontam ainda que facilidades de alto nível para a implementação de interação, como um mecanismo de apontamento, existem em pacotes de desenho vetorial mas não em ambientes de processamento de imagens. A integração do tratamento dos eventos de *mouse* e a realimentação visual deve ser explicitamente programada.

O suporte a processamento de imagens muitas vezes tem a forma de uma biblioteca que, obviamente, não implementa qualquer forma de controle de fluxo de execução. O programador tem o poder de escolha do ambiente de desenvolvimento. Além disso, a forma tradicional, não interativa, de programar operações sobre imagens nunca exigiu a implementação de rotinas de interação.

Portanto, a principal deficiência atualmente encontrada em sistemas de processamento de imagens é a falta de suporte para a manipulação e a exibição de dados vetoriais e para a implementação e reutilização de rotinas de interação.

3 Metodologia de projeto

A área de processamento de imagens, sendo uma área em desenvolvimento, requer ambientes de desenvolvimento extensíveis, com múltiplos modos de utilização e vários níveis de abstração. Essas características levam à construção de sistemas complexos devido,

essencialmente, a dependências entre algoritmos e dados através de diferentes módulos, segundo a análise de Booch [BOO91] e que, segundo Wegner [WEG96], não podem ser facilmente resolvidas por técnicas de decomposição como a análise estruturada.

Gamma et alli [GAM93][GAM95] notaram que bons projetistas freqüentemente utilizam soluções baseadas em experiências passadas. Um projetista experiente tende a aplicar recorrentemente soluções comprovadamente eficazes que acabam por gerar sistemas flexíveis, elegantes e extensíveis. Essas soluções são chamadas *design patterns*, microarquiteturas ou, simplesmente, padrões.

As ferramentas mais populares de projeto utilizam algum tipo de decomposição, de forma que os componentes de um sistema possam ser analisados isoladamente. Padrões descrevem componentes e também relações entre componentes. Por essa razão, padrões são realmente uma técnica arquitetônica e não uma técnica de divisão-e-conquista como a análise estruturada.

Padrões são independentes de domínio ou granularidade de aplicação e devem ser refinados ou especializados quando de sua implementação. Eles não diminuem a importância de outros métodos de análise, como UML (Unified Modeling Language) [FOW97], mas os complementam, documentando suas melhores formas de aplicação. Por isso são descritos através de textos, diagramas, exemplos e não puramente em função de construções específicas a uma ou outra linguagem de programação.

Um *framework* é um conjunto de classes que provê estrutura e comportamento genéricos para um domínio de aplicações. Uma aplicação é desenvolvida pela criação de subclasses e/ou pela instanciação das classes do próprio *framework*. Os pontos de um *framework* que podem ser especializados ou estendidos são chamados de *hot-spots*. Nesses pontos o programador insere classes específicas à sua aplicação. Ao fazer isso ele está efetivamente adicionando um subsistema que tipicamente contém [SCH97]: uma classe que implementa uma *interface* comum para todo o subsistema; classes derivadas (de classes do *framework*) representando variações do *hot-spot*; classes adicionais criadas pelo programador.

O programador utiliza uma referência polimórfica para ligar seu subsistema ao *framework* que, por sua vez, tem acesso à funcionalidade do subsistema através de métodos *hook*. Como *frameworks* implementam fluxo invertido de controle, eles podem ser comparados a clientes de servidores que são os subsistemas.

Padrões podem ser usados na estrutura de um *framework* e também no projeto de subsistemas. Se um *framework* foi projetado sobre padrões, a escolha da estrutura dos subsistemas é mais fácil, até natural. Por isso, padrões são importantes também para a documentação de *frameworks*.. De fato, Beck e Johnson [BEC94] apontam que padrões facilitam o projeto de arquiteturas e Johnson [JOH92] argumenta que a documentação de padrões é um auxílio importante na utilização correta de *frameworks*. Isso é importante para o processamento de imagens porque sistemas como o Khoros compreendem centenas de rotinas em diferentes níveis de abstração e são, por conseqüência, difíceis de dominar.

Brugali et alli [BRU97] apontam que, com o auxílio de padrões, novas aplicações desenvolvidas a partir de um *framework* produzem novos componentes de suporte. Isso significa que, na sua concepção, um *framework* é predominantemente caixa-branca e, com o tempo, o desenvolvimento de componentes (subsistemas) gera soluções tipo caixa-preta que estendem sua funcionalidade.

Isso é exatamente o que se espera de um ambiente de desenvolvimento de aplicações de processamento de imagens, segundo os requisitos identificados na seção 2. O programador deve ter todas as facilidades para o desenvolvimento de componentes específicos à sua aplicação e, além disso, deve ter a oportunidade de naturalmente compor a partir de

componentes seus e de outras fontes. A fim de suportar aplicações de segmentação interativa, deve-se ter a possibilidade de, em tempo de execução, adicionar e retirar componentes que interajam diretamente com a GUI. Tradicionalmente, *frameworks* se ocupam do tratamento de eventos e delegam as funções de processamento para os subsistemas. O que se deseja para segmentação interativa, por outro lado, é que o *framework* também distribua eventos para os subsistemas. Dessa forma, as funções de interação podem ser escolhidas pelo usuário, em tempo de execução.

O *framework* Blue foi projetado a partir da identificação de requisitos de aplicações de segmentação interativa identificados na seção 2. Considerando as questões de reuso e composição, adotou-se o paradigma de orientação a objetos e padrões bem difundidos [GAM95], os quais encontram-se descritos na próxima seção. A linguagem escolhida para a implementação foi Java, não somente porque é simples e orientada a objetos mas também porque é portável e suporta concorrência. Java não possui ponteiros e a coleta de lixo é automática, características que diminuem consideravelmente o esforço de desenvolvimento. A existência de compiladores, compiladores JIT (*Just In Time*) e de uma *interface* para código nativo (JNI-*Java Native Interface*) solucionam a questão de velocidade já que Java foi projetada para ser executada sobre uma máquina virtual e tendo em vista que técnicas interativas exigem resposta imediata.

4 Arquitetura e aspectos de implementação do *framework* Blue

O *framework* Blue faz uso extensivo das facilidades da API padrão, na versão 1.1 do JDK (*Java Development Kit*). Ele é constituído por cinco pacotes de classes que complementam a funcionalidade do JDK e, em especial, do AWT (*Abstract Windowing Toolkit*).:

- **Blue**, que consiste basicamente de classes abstratas que definem os relacionamentos entre todos os objetos do *framework*;
- **Geometry**, que implementa alguns objetos básicos para entes geométricos, como ponto, linha e *polyline*.
- **RAZ**, que possui objetos para a manipulação do formato de arquivos homônimo criado para utilização com o Blue (descrito adiante);
- **Sky**, utilizado para a construção de interpretadores, a partir de classes para análise léxica e sintática (utilizado na construção do decodificador RAZ);
- **Indigo**, que, a partir dos pacotes Blue e RAZ, implementa as funções de manipulação de arquivos (abrir, salvar, fechar, etc), edição (copiar, cortar, colar, etc) e visualização (*zoom* e composição de imagens).

O pacote Blue está organizado de acordo com o padrão MVC. Suas classes definem os relacionamentos que determinam a forma de composição de aplicações e, por isso, ele dá nome ao *framework*.. Os demais pacotes implementam funções de suporte ou classes concretas, derivadas de classes do pacote Blue, que podem ser diretamente utilizadas na codificação de aplicações.

As classes Model, View e Controller são utilizadas para a construção de *interfaces* em Smalltalk. Objetos desses tipos representam, respectivamente, a aplicação (Model), a *interface* (View), e o mapeamento das funções da *interface* para as funções da aplicação (Controller). Essa decomposição, além de promover a reutilização dos componentes, imprime maior flexibilidade à aplicação. Cada modelo se ocupa de notificar mudanças no seu estado às visões associadas que, por sua vez, têm a oportunidade de atualizar sua aparência em função do estado do modelo. Dessa forma, cada modelo pode ser representado por múltiplas visões sem

que a modificação ou criação de novas visões implique modificações no modelo. O controle executa métodos do modelo em função de eventos da *interface*.

As classes CWindow, CCanvas e CImage fazem, respectivamente, os papéis de controle, visão e modelo, conforme a figura 1.

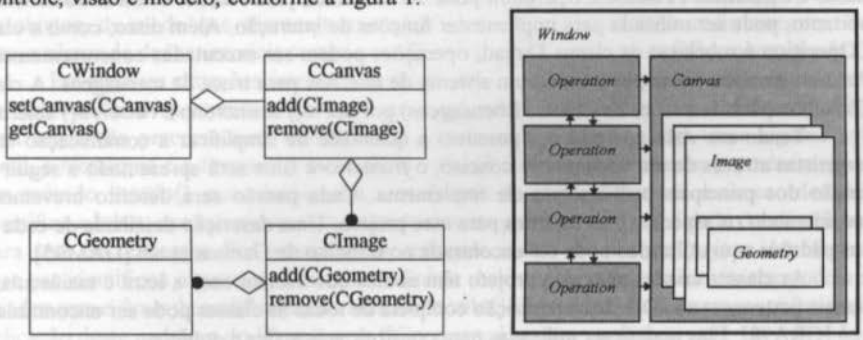

FIGURA 1 - Modelo básico do *framework* Blue

O ponto central do *framework* é a classe CWindow que contém um *canvas* (CCanvas) para a exibição e composição de imagens, gerência de operações sobre imagens e faz as vezes de contêiner para componentes de *interface* (fig.2). A classe CCanvas implementa *zoom*, *scrolling*, composição de imagens por transparências e cores falsas além de processar seus próprios eventos de *mouse*, transformando suas coordenadas para o espaço imagem antes que sejam capturados pela aplicação.

A classe CImage é a superclasse para imagens que obriga todas as subclasses a implementar uma *interface* comum, independente do formato dos *pixels*, além de implementar suporte para a gerência de coleções de entes geométricos (linhas, polígonos e até texto). Essa superclasse prevê apenas imagens com duas dimensões e *pixels* com valor inteiro de, no máximo, 32 *bits*. Entes geométricos são representados por instâncias de CGeometry e são exibidos sobre a imagem a qual estão associados. Entes geométricos podem ser utilizados como informação pictórica, representação vetorial de elementos de uma imagem ou para anotação de imagens. Atualmente existem duas implementações de CImage: CBitImage e CByteImage. Essas suportam, respectivamente, imagens binárias e imagens com 8 *bits* por *pixel*.

O formato de arquivos (RAZ=*RAw+Zipped*) criado para este projeto permite que imagens e dados vetoriais sejam armazenados conjuntamente. Cada arquivo possui um cabeçalho com uma assinatura ("raz"), tipo de imagem, número de *bits* por *pixel*, as dimensões da imagem e os comprimentos das seções de dados. O cabeçalho ocupa apenas 21 *bytes*. Ele suporta três tipos de codificação de cor: por paleta (IndexColor); alpha-RGB (ARGB encoded); por máscara de *bits* (DirectColor). Os dados vetoriais são representados por *strings*, segundo uma gramática recursiva simples: cada ente geométrico é representado pelo nome do seu tipo e uma lista (delimitada por { e }) de seus atributos. A gramática suporta também comentários dos tipos encontrados em C e C++ (// e /* */). Assim como o *string* contendo as informações vetoriais, os *bits* da imagem são compactados com o algoritmo LZW [LEL87]. As classes de codificação/decodificação utilizam o padrão Abstract Factory (descrito adiante) de tal forma que a criação de novos tipos de dados vetoriais não implica modificá-las.

Esse formato de arquivos foi desenvolvido porque nenhum outro pesquisado possuía ao mesmo tempo suporte ao armazenamento de dados vetoriais e as características de

simplicidade e extensibilidade. No entanto, o *framework* Blue pode ser utilizado com qualquer outro formato de arquivo.

Operações sobre imagens ou entes geométricos são implementadas como objetos da classe COperation. A classe COperation pode ser estendida para receber eventos do *canvas* e, portanto, pode ser utilizada para implementar funções de interação. Além disso, como a classe COperation é subclasse da classe Thread, operações podem ser executadas concorrentemente. Por isso, também foi implementado um sistema de *mailbox* para troca de mensagens. A classe CMailbox permite a troca de objetos (mensagens) por send() assíncrono e receive() síncrono.

Tendo em vista que padrões possuem a qualidade de simplificar a comunicação entre projetistas através de um vocabulário conciso, o *framework* Blue será apresentado a seguir em função dos principais padrões que ele implementa. Cada padrão será descrito brevemente, considerando os aspectos interessantes para este projeto. Uma descrição detalhada de cada um dos padrões aqui utilizados pode ser encontrada no trabalho de Gamma et alli [GAM95].

As classes criadas para este projeto têm nomes que iniciam com a letra c maiúscula, as demais pertencem ao JDK. Uma descrição completa de todas as classes pode ser encontrada na *web* [GRA98]. Elas podem ser utilizadas para construir aplicações e *applets*.

O padrão Observer ocorre quando existe uma relação de dependência um-para-muitos tal que, quando um objeto modifica seu estado, todos os seus dependentes são notificados e podem se ajustar às mudanças no tipo de interação chamada *publish-subscribe* (fig.2).

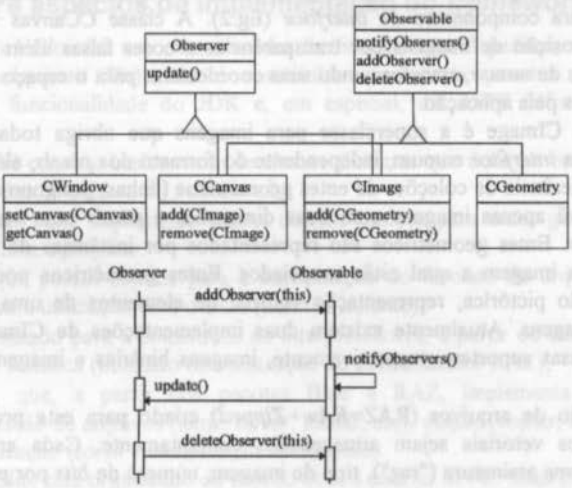

FIGURA 2 - Uso do padrão Observer

Blue utiliza este padrão para manter o *canvas* atualizado. A classe CImage é observada por CCanvas e, ainda, os entes geométricos (instâncias de CGeometry) de uma imagem são observados pela imagem. A princípio, a relação um-para-muitos parece estar invertida: um *canvas* observa muitas imagens e uma imagem observa muitos entes geométricos. Isso é sempre verdadeiro para o segundo caso. Para o primeiro, no entanto, podem existir situações em que várias janelas exibam a mesma imagem, talvez com diferentes modos de visualização.

Os entes geométricos não são observados diretamente pelo *canvas* por dois motivos: para manter a relação conceitual entre a imagem e seus entes geométricos e para otimizar as atualizações do *canvas*. Modificações em vários entes geométricos não implicam múltiplas

atualizações do *canvas*. Como conseqüência da aplicação deste padrão, a dependência entre as classes envolvidas é mínima.

No projeto da classe CGeometry foram usados três padrões: Wrapper, Chain of Responsibility e Prototype. O propósito do padrão Wrapper é permitir a composição recursiva de objetos, de modo que clientes tratem objetos simples e compostos uniformemente. Para isso, a classe de dados vetoriais compostos (CComposite) é derivada de CGeometry (fig.3).

Em algumas situações é necessário desvincular um cliente de seu servidor, dando a mais de um servidor a oportunidade de tratar uma requisição. No padrão Chain of Responsibility, uma requisição percorre uma hierarquia de servidores até que um deles tome alguma ação. Aqui, a hierarquia é a formada por um dado vetorial composto. Cada ente geométrico possui um método locate() que recebe uma coordenada e devolve true se essa coordenada o identifica e false, caso contrário. Um ente composto passa a responsabilidade de identificação para seus descendentes. Por exemplo, um retângulo indica que um par (x,y) o identificou se esse par identificou alguma de suas quatro linhas. Assim, novos entes geométricos podem ser criados sem o ônus da codificação do método locate(). É interessante notar que a diferença da aplicação deste padrão e do Wrapper em CGeometry é apenas semântica.

O padrão Prototype é implementado para permitir a clonagem de instâncias de CGeometry a fim de facilitar a implementação de operações do tipo *cut-and-paste*.. Ele se confunde com Wrapper e Chain of Responsibility porque cada ente geométrico composto monta um clone seu a partir de clones de seus componentes. O JDK permite a clonagem de objetos através do método clone() da classe Object, superclasse de todas as classes. No entanto, ao clonar um objeto, os valores dos campos do objeto original são copiados para seu clone. Então, referências do clone apontam para os mesmos objetos que as referências do original. Isso é inútil para entes geométricos que devem possuir instâncias únicas de seus componentes. Por isso, CGeometry sobreescreve clone() para devolver clones com todos os membros replicados (fig.3).

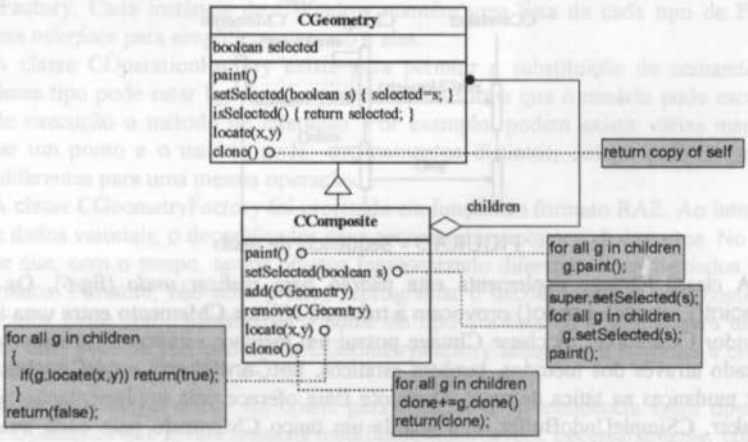

FIGURA 3 - A classe CGeometry

Aplica-se o padrão Facade para prover uma *interface* única a um conjunto de *interfaces* de um subsistema. Uma *interface* única para um subsistema complexo permite criar clientes mais simples desse subsistema. Ele é utilizado na classe CView, conforme pode ser visto na figura 4. Cada objeto CView é associado, em tempo de execução, a uma instância de

CWindow e é responsável pela criação dos componentes de *interface* para um determinado pacote de funções. Por exemplo, o pacote Indigo possui várias classes relacionadas à manipulação de arquivos (abrir, salvar, etc); CIndigoView implementa os *menus* associados a essas funções. Com isso, uma aplicação pode ser montada pela aglutinação de instâncias CView de diferentes pacotes. Essa solução difere do padrão Facade apenas em que cada CView carrega consigo os susbsistemas aos quais confere uma *interface* única. Instâncias de CView podem receber eventos das janelas associadas (minimizar, fechar, abrir, etc) para que cada uma possa ser implementada como uma aplicação em si.

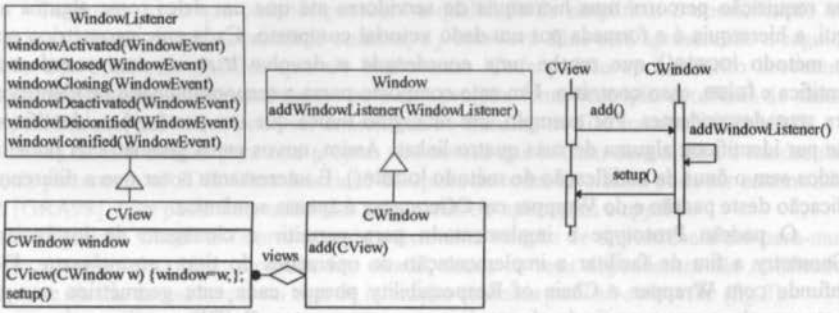

FIGURA 4 - Uso do padrão Facade

O padrão Memento é usado quando, sem violar o encapsulamento, deseja-se exportar o estado de um objeto para poder restaurá-lo mais tarde. A aplicação óbvia deste padrão é na construção de mecanismos de *undo* (fig.5). Ele resolve as relações entre uma fonte (neste caso uma imagem) que deseja exportar seu estado, os marcadores que encapsulam esses estados (Memento) e o servidor (Caretaker) que gerencia os marcadores.

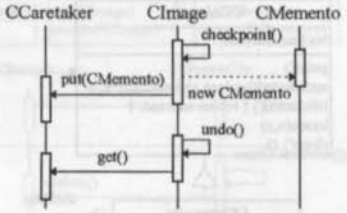

FIGURA 5 - Seqüência de *undo*

A classe CImage implementa este padrão para realizar *undo* (fig.6). Os métodos checkpoint(), undo() e redo() provocam a troca de objetos CMemento entre uma imagem e um servidor CCaretaker. A classe CImage possui um membro estático CCaretaker que pode ser trocado através dos métodos, também estáticos, setCaretaker() e getCaretaker() para permitir mudanças na tática de *undo*.. O pacote Blue oferece uma implementação simples de CCaretaker, CSimpleUndoBuffer, que guarda um único CMemento para cada imagem, em memória. Implementações alternativas de CCaretaker poderiam utilizar arquivos e/ou permitir múltiplos níveis de *undo/redo*. Uma chamada ao método checkpoint() provoca a criação de um CMemento contendo uma cópia da imagem e dos entes geométricos. Cada subclasse de CImage guarda informações diferentes nos marcadores mas o servidor conhece somente a classe abstrata CMemento e, portanto, não pode modificar os dados neles contidos. Mudanças na estratégia de *undo* dizem respeito somente a CCaretaker e mudanças no tipo de informação

de *undo* implicam modificações apenas nas subclasses de CImage. Um marcador otimizado poderia conter apenas as diferenças na imagem entre a atual e a última chamada a checkpoint().

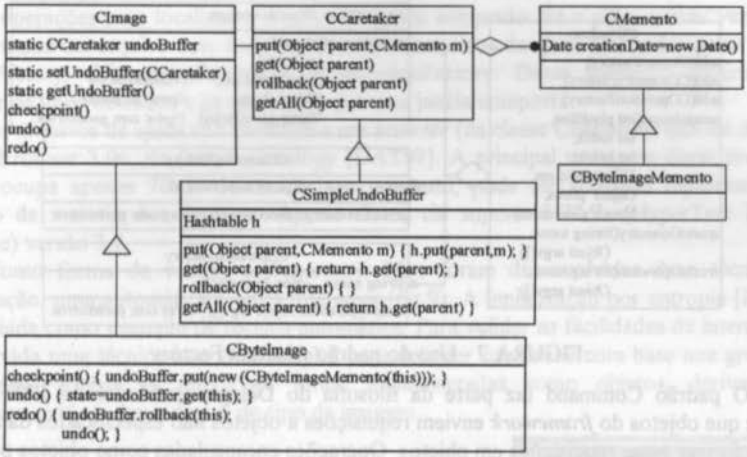

FIGURA 6 - Uso do padrão Memento

A intenção do padrão Abstract Factory é permitir a criação de objetos através de uma *interface* abstrata (Factory), retardando a escolha do tipo de objeto a ser instanciado para o tempo de execução.

A figura 7 mostra o uso de padrão Abstract Factory. A classe CWindow implementa suporte para três tipos distintos de Factory: COperationFactory, CGeometryFactory e CImageFactory. Cada instância de CWindow mantém uma lista de cada tipo de Factory e provê uma *interface* para simplificar o acesso a elas.

A classe COperationFactory existe para permitir a substituição de comandos. Um objeto desse tipo pode estar ligado a um *menu*, de tal forma que o usuário pode escolher em tempo de execução o método de interação. Por exemplo, podem existir várias maneiras de identificar um ponto e o usuário pode, em momentos distintos, desejar utilizar funções de entrada diferentes para uma mesma operação.

A classe CGeometryFactory foi projetada em função do formato RAZ. Ao interpretar a seção de dados vetoriais, o decodificador deve reconstruir os objetos ali descritos. No entanto, espera-se que, com o tempo, novos objetos representando diferentes tipos de dados vetoriais sejam criados. Portanto, não seria prático reprogramar o decodificador a cada novo tipo. O decodificador passa como parâmetros o nome do tipo e a lista de atributos para um objeto CWindow que, por sua vez, procura o CGeometryFactory adequado e requisita a criação do objeto.

A classe CImageFactory foi criada para reforçar a independência entre operações e imagens. Operações que geram imagens modificadas não devem, necessariamente, conhecer o tipo da imagem sobre a qual estão sendo executadas. A partir dos dados de uma imagem (número de *bits* por *pixel*, modelo de cor, etc) pode-se requisitar a criação de uma nova imagem do mesmo tipo, se o CImageFactory adequado existir.

Os três tipos de Factory podem, também, servir de suporte ao desenvolvimento de uma linguagem de comandos. O interpretador, assim como o decodificador de arquivos RAZ, não

necessitaria conhecer os tipos de imagens, dados vetoriais ou operações para poder interpretar os comandos do usuário e seria imune ao desenvolvimento de novos tipos.

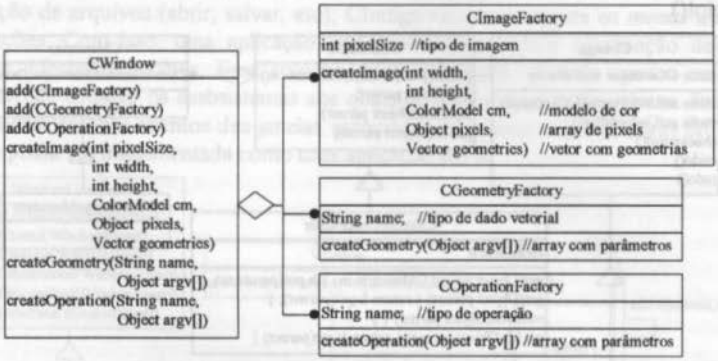

FIGURA 7 - Uso do padrão Abstract Factory

O padrão Command faz parte da filosofia do Delegation Model do JDK 1.1. Ele permite que objetos do *framework* enviem requisições a objetos não especificados da aplicação ao transformar essas requisições em objetos. Operações encapsuladas como objetos podem ser enfileiradas, armazenadas para registro de seqüências de execução (*log*) e facilitam a implementação da função de desfazer (*undo*).

Este padrão é aplicado pela necessidade de execução concorrente de múltiplas operações interativas. Além disso, ele facilita a distribuição e o tratamento de eventos, que são fatores importantes na codificação e reutilização de técnicas interativas (fig.8).

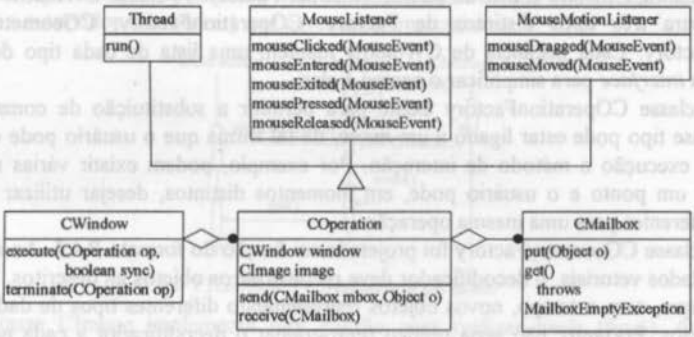

FIGURA 8 - A classe COperation

Cada objeto CWindow gerencia uma pilha de operações, garantindo que apenas a operação mais recentemente invocada receba os eventos do *mouse*. Executa também coleta de lixo ao destruir automaticamente as operações que já concluíram sua execução. As operações podem ser executadas síncrona ou assincronamente. No primeiro caso, o método execute(), da classe CWindow, aguarda o término da operação e, no segundo, retorna imediatamente.

A implementação de operações como objetos obviamente facilita a criação de um mecanismo de ajuda sensível a contexto. Cada objeto do tipo COperation, CImageFactory, CGeometryFactory e COperationFactory possui uma referência que deve apontar para um arquivo contendo as informações daquele tipo de operação, imagem ou dado vetorial. Essa

referência pode identificar um arquivo local ou pode ser um URL (Universal Resource Locator).

Quando executado, o método help() da classe CWindow procura o objeto no topo da pilha de operações para localizar o arquivo de ajuda adequado. Se a pilha estiver vazia, esse método monta um arquivo com *links* para todos os arquivos de ajuda apontados pelos objetos CImageFactory, CGeometryFactory e COperationFactory. Dessa forma, o usuário pode pesquisar os tipos de dados e as operações que uma janela comporta.

Os arquivos de ajuda são enviados a um *browser* (da classe CBrowser) que foi derivado do ICE Browser 3.06, da DataTechnology [DAT97]. A principal vantagem desse *browser* é que ele ocupa apenas 70KB de memória e, portanto, pode ser acessado rapidamente, ao contrário da maioria dos *browers*. Apesar disso, ele suporta HTML (HyperText Markup Language) versão 3.2.

Como forma de validar o *framework* Blue foram desenvolvidas duas técnicas de segmentação, uma automática e outra interativa (fig.9). A limiarização por entropia [PUN81] foi escolhida como exemplo de técnica automática. Para validar as facilidades de interação foi desenvolvida uma técnica interativa simples para delinear contornos com base nos gradientes da imagem. Ambas as operações foram implementadas como objetos, derivados de COperation, e são independentes do tipo de imagem.

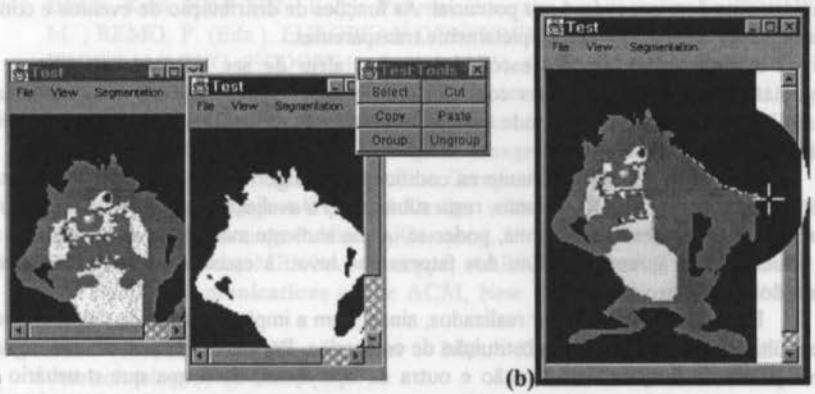

FIGURA 9 - Exemplos de uso do *framework* Blue: (a) limiarização por entropia (b) técnica interativa (detalhe incluído com manipulação da imagem)

5 Conclusões e extensões

A avaliação das necessidades de suporte ao desenvolvimento de técnicas interativas de segmentação de imagens permitiu a identificação das deficiências dos atuais sistemas de desenvolvimento de aplicações de processamento de imagens. Em especial, foram identificadas características que inibem a rápida prototipação de novas técnicas interativas.

O principal obstáculo à codificação de técnicas interativas está no fato de que a atual tecnologia de processamento de imagens é orientada a algoritmos. Dessa forma, os ambientes de desenvolvimento raramente possuem facilidades de interação mais sofisticadas que a simples entrada de parâmetros e a execução iterativa de diferentes algoritmos.

A segmentação interativa exige a cooperação entre usuário e aplicação durante o processamento. Isso significa que a avaliação de novas técnicas implica não somente a codificação de operações sobre imagens como também o controle do fluxo de eventos e a

visualização imediata de resultados parciais. Essas duas funções são muitas vezes responsáveis por código complexo e de difícil manutenção. Dessa forma, a reutilização de código é dificultada e, conseqüentemente, a tarefa de prototipação tem um custo maior.

Além disso, a pesquisa em processamento de imagens é essencialmente experimental e, portanto, deve-se freqüentemente alternar entre um modo usuário e um modo de programação dentro do ambiente de desenvolvimento. Já que é incomum o suporte à interação, geralmente os sistemas de processamento de imagens podem ser utilizados somente no nível mais baixo de abstração, tipicamente bibliotecas de rotinas, para a implementação de técnicas interativas.

Identificou-se, então, que um ambiente de desenvolvimento deve possuir ferramentas de uso geral aliadas a alguns componentes prontos e uma metodologia simples de composição. Uma metodologia de composição projetada com suporte à interação levaria naturalmente ao desenvolvimento de um sistema interativo em todos os níveis de abstração.

A solução encontrada foi a de projetar um *framework* orientado a objetos, de forma a promover a reutilização de componentes e de facilitar a composição de novas aplicações. O modo de utilização de *frameworks* foi analisado, levando à escolha de *design patterns* como metodologia de projeto a fim de conferir às ferramentas de desenvolvimento as características de simplicidade e extensibilidade. Apesar de conter mais de uma centena de classes, Blue possui uma metodologia simples de composição que deve auxiliar um usuário novato a rapidamente dominar todo o seu potencial. As funções de distribuição de eventos e controle da visualização dos dados são completamente transparentes.

A linguagem Java foi escolhida porque, além de ser orientada a objetos, oferece algumas facilidades importantes como coleta de lixo e independência de plataforma. Além disso, o JDK oferece uma grande variedade de classes de uso geral que permitiu acelerar o desenvolvimento deste projeto.

Blue pode ser utilizado tanto na codificação de algoritmos como de técnicas interativas com igual simplicidade. No entanto, resta submetê-lo à avaliação de pesquisadores experientes em outros ambientes. Dessa forma, poder-se-ia não somente avaliar sua eficácia como também a velocidade de aprendizado, um dos fatores que levou à escolha de *design patterns* como metodologia de projeto.

Experimentos devem ser realizados, ainda, com a implementação de técnicas compostas de múltiplos objetos ou com substituição de comandos. Por exemplo, poder-se-ia implementar uma paleta de funções de interação e outra de operações, de forma que o usuário poderia associar a operação ao modo de interação em tempo de execução. As paletas seriam implementadas como instâncias de COperationFactory, além de terem uma representação gráfica. Além disso, a implementação de uma linguagem de comandos realizaria o potencial do padrão Abstract Factory, na forma como foi utilizado neste projeto.

A tecnologia de JavaBeans permite a programação visual de aplicações através da composição de componentes em uma ferramenta gráfica. Essa possibilidade deve ser analisada já que permitiria a programação visual de técnicas de processamento de imagens com componentes do Blue.

Assim, através de uma linguagem de comandos e um ambiente de programação visual, Blue pode ser estendido até um ambiente de múltiplos modos de programação, como o sistema Khoros.

A classe abstrata de imagens deve ser estendida para comportar novos tipos de imagens e, possivelmente, imagens com mais de duas dimensões. Nesse caso, a classe CCanvas seria estendida para comportar operações de projeção e novas operações de visualização. Deve-se ainda, identificar um conjunto de operações básicas a ser implementado como métodos das

classes de imagem de forma a diminuir a necessidade de modificá-las já que isso comprometeria a independência de dados das operações implementadas como objetos.

Apesar do direcionamento ter sido aplicações de segmentação, Blue é abrangente e pode ser adotado no desenvolvimento de outras classes de aplicações. Senão diretamente, ao menos as soluções de projeto podem ser aplicadas além das áreas de processamento de imagens e computação gráfica. Em especial, Blue resolve de uma maneira simples as relações entre dados, visualização dos dados, *interface*, e múltiplas operações interativas e concorrentes.

Agradecimentos

Agradecemos a Silvia Delgado Olabarriaga pelas sugestões e críticas quando do início do projeto e ao CNPq pelo apoio financeiro.

Referências

[BAL 94] BALEN, Richard van et al. ScilImage: A Multi-layered Environment for Use and Development of Image Processing Software. In: **Experimental Environments for Computer Vision and Image Processing**.. Singapore: World Scientific Press, 1994, p.107-126.

[BEC 94] BECK, Kent; JOHNSON, Ralph. Patterns Generate Architectures. In: TOKORO, M. ; REMO, P. (Eds.). EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP'94), 8., July 1994, Bologna. **Proceedings...** Berlin: Springer Verlag, 1994, p.139-149. (Lecture Notes in Computer Science 821).

[BEN 81] BENGTSSON, E. et al. An Interactive System for Image Analysis. In: BOLC, Leonard; KULP, Zenon (Eds.). **Digital Image Processing Systems**. Berlin: Springer Verlag, 1981. (Lecture Notes in Computer Science 109).

[BOO 91] BOOCH, Grady. **Object-Oriented Design with Applications**.. Redwood City: The Benjamin/Cummings Publishing Company, 1991.

[BRU 97] BRUGALI, Davide; MENGA, Giuseppe; AARSTEN, Amund. The Framework Life Span. **Communications of the ACM**, New York, v.40, n.10, p.65-68, Oct. 1997.

[DAT 97] DataTechnology. **ICE Browser 3.06**.. Disponível por e-mail em datatechnology@bgnett.no (1997).

[FOW 97] FOWLER, Martin. **UML Distilled - Applying the Standard Object Modeling Language**..Reading: Addison Wesley, 1997.

[GAM 93] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In: NIERSTRASZ, O. M. (Ed.). EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP'93), 7., July 26-30, 1993, Kaiserslautern. **Proceedings...** Berlin: Springer-Verlag, 1993, p.406-431. (Lecture Notes in Computer Science 707).

[GAM 95] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns - Elements of Reusable Object-Oriented Software**..Reading: Addison Wesley, 1995.

[GON 92] GONZALEZ, Rafael; WOODS Richard. **Digital Image Processing**. Reading: Addisson Wesley, 1992.

[GRA 98] GRAÇA, Gabriel Lühers. **Documentação do *framework* Blue**.. Disponível por WWW em http://www.inf.ufrgs.br/~graca/Blue/. (1998).

[JAI 89] JAIN, Anil. **Fundamentals of Digital Image Processing**.. Englewood Cliffs: Prentice-Hall, 1989.

[JOH 92] JOHNSON, Ralph E. Documenting Frameworks using Patterns. In: ANNUAL ACM CONFERENCE ON OBJECT-ORIENTED SYSTEMS, LANGUAGES AND APPLICATIONS (OOPSLA'92), 7., 1992, Vancouver. **Proceedings...** New York: ACM Press, 1992.

[JOH 97] JOHNSON, Ralph E. Frameworks=(Components+Patterns). **Communications of the ACM**, New York, v.40, n.10, p.39-42, Oct. 1997.

[KON 94] KONSTANTINIDES, K.; RASURE, J. R. The Khoros Software Development Environment for Image and Signal Processing. **IEEE Transactions on Image Processing**, New York, v.3, n.3, May 1994.

[LAF 91] LAFFRA, C.; BOS, J. van den. A Layered Object-Oriented Model for Interaction. In: BLAKE, E. H.; WISSKIRCHEN, P. (Eds.). **Advances in Object-Oriented Graphics I**..Berlin: Springer Verlag, 1991. (Eurographics Seminars).

[LEL 87] LELEWER, D. A.; HIRSCHBERG, D. S. Data Compression. **ACM Computing Surveys**, New York, v.19, n.3, p.216-296, 1987.

[OLA 97a] OLABARRIAGA, S. D.; SMEULDERS, A. W. M. Setting the mind for Intelligent Interactive Segmentation: Overview, Requirements, and Framework. In: DUNCAN, J.; GINDI, G. (Eds.). INTERNATIONAL CONFERENCE ON INFORMATION PROCESSING IN MEDICAL IMAGING (IPMI'97), 15., 1997, Poultney VT. Proceedings... Berlin: Springer Verlag, 1997, p.417-422. (Lecture Notes in Computer Science 1230). Disponível por WWW em http://carol.winds.uva.nl/~silvia (1997).

[OLA 97b] OLABARRIAGA, S. D.; KOELMA, D.; SMEULDERS, A. W. M. A Simple Application Framework for Interactive Segmentation Systems. In: Annual Conference of the Advances School for Computing and Imaging (ASCI'97), 3, 1997. **Proceedings...** [S.l.], p.151-156, 1997. Disponível por WWW em http://carol.wins.uva.nl/~silvia (1997).

[POP 94] POPE, Arthur R.; LOWE, David G. Vista: A Software Environment for Computer Vision Research. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR'94), 1994. **Proceedings...** New York: IEEE Computer Society Press, 1994, p.768-772.

[PUN 81] PUN, T. Entropic thresholding: A new approach. **Computer Vision Graphics and Image Processing,** [S.l.], n.16, p.210-239, 1981.

[SCH 97] SCHMID, Hans Albrecht. Systematic Framework Design. **Communications of the ACM**, New York, v.40, n.10, p.48-51, Oct. 1997.

[VAM 81] VÁMOS, T. et al. A Knowledge-based Interactive Robot-vision System. In: BOLC, Leonard; KULP, Zenon (Eds.). **Digital Image Processing Systems**. Berlin: Springer Verlag, 1981. (Lecture Notes in Computer Science 109).

[WEG 96] WEGNER, Peter. Interactive Software Technology. In: TUCKER, A. (Ed.). **Handbook of Computer Science and Engeneering**..Boca Raton FL: CRC Press, 1996. Disponível por WWW em http://www.cs.brown.edu/people/pw (1996).