O conteúdo do presente relatório é de única responsabilidade do(s) autor(es).
(The contents of this report are the sole responsibility of the author(s).)

Compreensão de Algoritmos através de Ambientes Dedicados a Animação

Rackel Valadares Amorim
Pedro Jussieu de Rezende

Relatório Técnico DCC–25/93

Setembro de 1993

Compreensão de Algoritmos através de Ambientes Dedicados a Animação

Rackel Valadares Amorim* Pedro Jussieu de Rezende†

Departamento de Ciência da Computação
Universidade Estadual de Campinas
13081-970 Campinas, SP

Sumário

Descrevemos diferentes formas de animação de algoritmos que já foram estudadas, técnicas de apresentação e ambientes de programação desenvolvidos para este fim nos últimos anos. Apresentamos uma descrição do ambiente **AnimA** que projetamos e implementamos com o objetivo de auxiliar no processo de produção de animações de algoritmos, provendo os usuários de facilidades para produção de animações de qualquer algoritmo de maneira sistemática através da congregação num mesmo ambiente de ferramentas de suporte a animação como: interface gráfica uniforme, sistema de pré-processamento, manutenção automática de bibliotecas, reusabilidade de componentes através de programação orientada a objetos e utilização de bibliotecas compartilhadas e ligação dinâmica.

Abstract

We describe various forms of algorithm animation which have been studied, presentation techniques and programming environments developed for this purpose in recent years. We present a

*Pesquisa desenvolvida com suporte financeiro parcial da CAPES, e FAPESP — Fundação de Amparo à Pesquisa do Estado de São Paulo, através do auxílio 91/4167-6.

†Pesquisa desenvolvida com suporte financeiro parcial do CNPq — Conselho Nacional de Desenvolvimento Científico e Tecnológico, através dos auxílios 300157/90-8 e 500787/91-3

description of the environment **AnimA** which we designed and implemented with the goal of aiding in the process of production of algorithm animation providing users with facilities for the production of animations of any algorithm in a systematic way, by means of congregating in a single environment tools for animation support such as: uniform graphical interface, preprocessing system, automatic maintenance of libraries, reusability of shared libraries and dynamic linking.

1 Introdução

Em meados da década de oitenta, houve um grande avanço na tecnologia de *hardware*, que resultou no surgimento de estações de trabalho com monitores de alta resolução, grande capacidade de memória real e virtual e processadores poderosos, interligadas em redes locais, compondo ricos ambientes gráficos de recursos compartilhados.

Com o maior potencial destas arquiteturas, várias novas áreas se desenvolveram a partir da exploração dos recursos gráficos, em particular para sua aplicação em animações visando a pesquisa e mesmo o ensino de diversas áreas de Ciência da Computação.

Monitores de alta resolução gráfica podem prover representações gráficas detalhadas de programas, estruturas de dados e de outros conceitos estudados. Poderosos sistemas de computação que suportam sofisticados *displays* podem ser amplamente utilizados para expor características dinâmicas essenciais de vários conceitos.

Uma forma de aplicação de recursos gráficos à programação, classificada como **programação visual,** [RE 86] [RS 86] [RS 87] utiliza diagramas bidimensionais que descrevem fluxos de controle de algoritmos, e inclui sistemas que usam objetos gráficos como entidades computacionais fundamentais.

A **visualização de programas,** embora relacionada, é distinta da programação visual. Ela destaca o uso da tecnologia de recursos gráficos e de interação, da tipografia e animação para facilitar a apresentação e o entendimento de algoritmos [CA 85].

A visualização de programas pode ser classificada conforme o conteúdo apresentado (código ou dados) e o modo das visualizações (estáticas ou dinâmicas). Os fluxogramas, diagramas de escopo e gráficos mostrando a conexão entre módulos são exemplos de visualizações de código estáticas. Visualizações estáticas de dados podem ser produzidas de forma bastante diversificada, devido às diferentes representações possíveis para cada estrutura de dados e às várias maneiras possíveis de implementação.

Visualizações estáticas de programas podem ser facilmente tornadas dinâmicas, através de *highlights* nas partes apropriadas, colocados à medida que o código vai sendo executado. Esta dinamização permite uma observação limitada do comportamento do algoritmo.

O presente trabalho descreve uma forma sofisticada de visualização de programas, que se concentra na representação dinâmica de estruturas de dados e do fluxo de controle envolvidos na execução de um algoritmo.

2 Animação de Algoritmos

A animação de um algoritmo é uma forma de visualização de seu comportamento, onde se utiliza a tecnologia de computação gráfica e técnicas de animação para permitir uma melhor apresentação e entendimento da lógica envolvida em suas operações.

Imagine, por exemplo, um algoritmo que envolve recursão, como o *Quicksort*. Mapeando-se os valores a serem ordenados para coordenadas na tela, de modo que a coordenada x represente a posição do elemento no vetor, e a coordenada y seja o valor do elemento, teremos um conjunto de pontos na tela (inicialmente desordenados). Se cada operação de troca de elementos for refletida na tela trocando-se os pontos correspondentes, à medida que o conjunto de pontos for sendo recursivamente subdividido, pode-se ver os pontos sendo subdivididos em grupos, e ordenados por partes. Veja a figura 1.

Através da animação pode-se observar certas características do algoritmo que poderiam passar despercebidas, além de ser possível analisar o seu comportamento diante de diferentes conjuntos de dados de entrada.

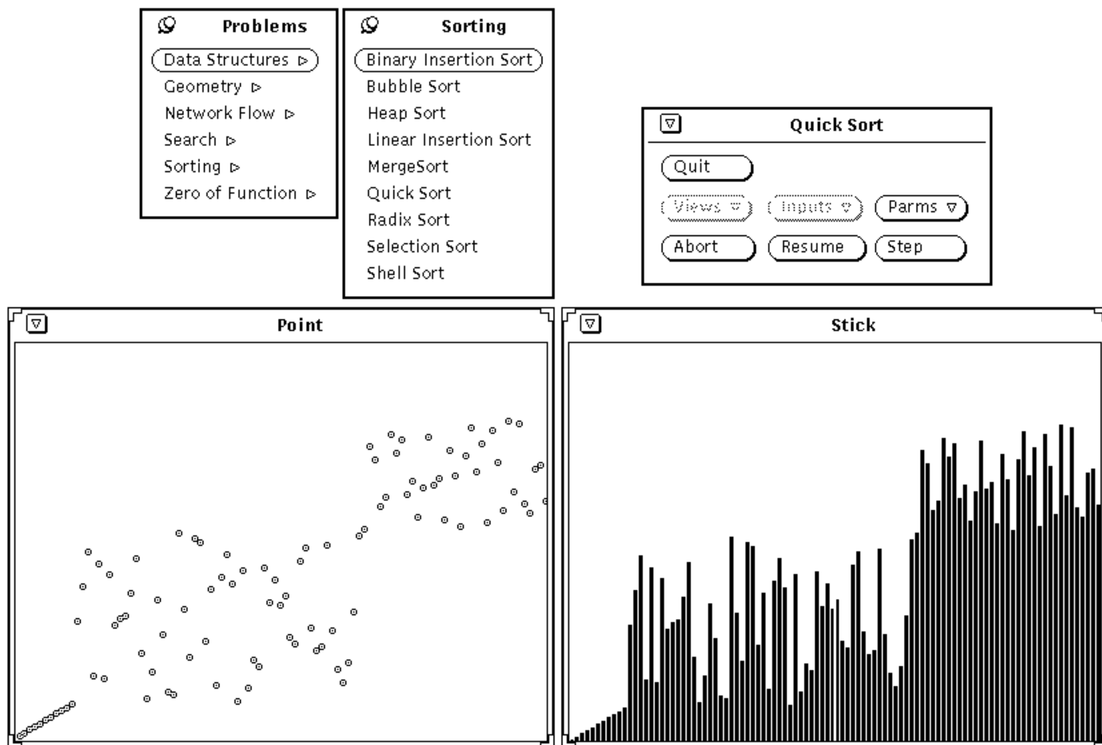


Figura 1: Animação de QuickSort em ação.

As visualizações envolvidas em animações de algoritmos podem ser classificadas com base em três aspectos [BR 87]:

- Conteúdo: o conteúdo das visualizações pode variar entre representações diretas dos dados ou código do algoritmo, até imagens abstratas não necessariamente presentes no programa (figura 1).
- Persistência: este aspecto diz respeito ao intervalo mostrado pela visualização, que pode se concentrar apenas no estado corrente ou mostrar um histórico completo das mudanças durante a execução.
- Transformação: diz respeito à maneira como as mudanças no estado do algoritmo são retratadas, podendo refletir mudanças discretas nas gravuras ou mudanças incrementais e contínuas.

2.1 Visualização passiva vs. Visualização ativa

Visualizações dinâmicas podem ser passivas ou ativas, conforme o grau de interação com o espectador. Nas visualizações passivas, o espectador assiste à animação, sem influenciar o seu andamento. Como exemplo de visualização passiva pode-se citar filmes vistos em *videotapes*. Nas visualizações ativas, o espectador interage com a animação, influenciando diretamente o seu comportamento.

A visualização ativa de um programa permite que o espectador crie situações especiais que reflitam, por exemplo, o comportamento do algoritmo para conjuntos de dados críticos, ou casos de exceção.

3 Ambientes Desenvolvidos na Área

Um ambiente de animação de algoritmos tem como principal objetivo fornecer as ferramentas necessárias para a animação de qualquer algoritmo de maneira sistemática, além de prover o usuário final de facilidades para executar e interagir com as animações.

Alguns ambientes que objetivam congregar estas características já foram desenvolvidos.

3.1 BALSA

BALSA (Brown ALgorithm Simulator and Animator) [BR 84] foi desenvolvido no início da década de oitenta, entrando em uso regular em 1983. O sistema foi projetado com o intuito de servir como um laboratório de experimentação com representações dinâmicas de algoritmos em tempo real.

BALSA aplica-se a quatro classes de usuários: o usuário final, que assiste e interage com as animações; o produtor de *scripts* (descritos a seguir); o projetista de algoritmos, que escreve o algoritmo e define quais operações são interessantes para a produção de uma boa animação; e o animador, que define e implementa as representações visuais que melhor refletem o real funcionamento do algoritmo.

BALSA oferece uma maneira sistemática para a produção de animações. O projetista do algoritmo escreve uma versão do programa a ser animado, e módulos que fornecem vários tipos de dados de entrada. Então, juntamente com o animador, ele identifica os eventos interessantes que devem resultar em atualizações nas imagens que serão mostradas durante a execução. O animador escreve os programas responsáveis por atualizar as visualizações, e o projetista marca os eventos interessantes no programa fonte que já foi escrito. Feito isto, o ambiente possui o algoritmo e os visualizadores necessários para a produção da animação. Apesar de apresentar uma interface amigável, os recursos gráficos utilizados nas visualizações são limitados.

BALSA apresenta ainda *scripts*, um recurso através do qual tudo que é feito durante uma sessão de animações é guardado em um arquivo para ser executado posteriormente.

O sistema foi desenvolvido para rodar em um ambiente de estações de trabalho ligadas em rede, e foi utilizado em disciplinas do curso de Ciência da Computação e em pesquisas na área de projeto e análise de algoritmos. Posteriormente, seu uso foi substituído por BALSA-II, onde houve o acréscimo e o aperfeiçoamento de algumas características, com base na experiência de uso do primeiro sistema.

BALSA-II provê recursos para animações de alta qualidade além

de uma interface amigável que oferece uma maneira homogênea para a execução e interação com animações. No entanto, a manutenção das bibliotecas do ambiente ainda é realizada manualmente pelo programador. BALSA-II apresenta ainda uma melhor versão dos *scripts* existentes em BALSA.

BALSA-II foi produto da tese de Doutorado de Marc H. Brown, e entrou em uso regular em 1987. O sistema foi posteriormente portado para a plataforma Macintosh e presupõe códigos escritos na linguagem Pascal para algoritmos a serem animados.

Apesar de seu grande potencial, BALSA-II apresenta algumas limitações pois a nível do programador que escreve os componentes, não é oferecida nenhuma interface gráfica para a manutenção das bibliotecas do ambiente. Além disso, o sistema ainda não permite os recursos de cores e sons em suas animações.

3.2 ANIM

ANIM é um sistema de animação de algoritmos cuja intenção é prover o usuário de grande facilidade de uso e rapidez na produção de animações, produzindo visualizações bastante simples.

O primeiro passo para se animar um algoritmo usando-se o ANIM é escrever o programa fonte, trocando-se os comandos de saída por um conjunto de comandos que seguem especificações próprias do sistema. A partir deste programa é produzido um *script* que consiste de chamadas a primitivas do sistema para a produção das visualizações, que podem ser vistas através de filmes ou gravuras.

Na produção de filmes a animação é mostrada do início ao fim com a possibilidade de se examinar a mesma com mais detalhes através das opções de *stop-go* e controle de velocidade. Além disso, pode-se ver a animação no sentido normal (do início para o fim) ou no sentido inverso.

Este sistema apresenta algumas limitações como nao permitir que o usuário interaja com as animações. Uma vez tendo sido criado um *script*, qualquer modificação na animação exige que um novo *script* seja gerado. Além disso, não há flexibilidade na geração dos dados de entrada, e os

recursos de visualização são bastante limitados. Poucos objetos visuais podem ser usados nas animações, com poucos recursos visuais, e não há o uso de objetos compostos. ANIM foi escrito em 1987 em linguagem C para executar no Teletype 5620.

3.3 Zeus

O ambiente de animação de algoritmos Zeus [BR 89] foi desenvolvido por Marc H. Brown, depois das experiências adquiridas com BALSA e BALSA-II, e entrou em uso regular em 1990. Zeus foi posteriormente portado por Marc Brown e John Hershberger para Modula 3. Os autores se concentraram em algumas características que consideraram de maior importância tendo como base os sistemas anteriores.

Zeus provê o usuário de facilidades a nível de configuração e a nível de execução. As facilidades de configuração permitem ao usuário selecionar o algoritmo e as visualizações, e informar os dados de entrada. Existem ainda funções adicionais que gravam registros descrevendo o estado do sistema (*snapshots)* em um arquivo (informações sobre as visualizações, valores de dados de algoritmos), e que recuperam o estado do sistema a partir de registros guardados anteriormente.

O sistema não apresenta o recurso de *scripts* apresentado em BALSA e BALSA-II, e não há mais um componente específico para a entrada de dados (geradores de entrada). O ambiente se destaca entre os demais sistemas de animação por seguir o paradigma de orientação a objetos e utilizar tipos abstratos de dados e paralelismo, além de explorar mais recursos de visualização. Zeus suporta a construção de programas de grande porte, integrados, orientados a objetos e concorrentes. Os componentes também devem ser escritos em Modula 3.

4 AnimA — Um Ambiente em Desenvolvimento

Estamos atualmente em fase final de implementação do ambiente **AnimA** que se relaciona com os mencionados acima no sentido de que seus objetivos são de auxiliar no processo de produção de animações de algoritmos

provendo os usuários de facilidades para produção de animações de qualquer algoritmo de maneira sistemática.

AnımA elimina várias dificuldades presentes na tarefa de se animar um algoritmo isoladamente, e apresenta novidades com relação aos ambientes similares desenvolvidos até o momento. A reusabilidade de componentes permite que módulos sofisticados de visualização sejam implementados apenas uma vez, e fiquem disponíveis em bibliotecas do ambiente, para serem recombinados com outros componentes criando-se novas animações. Além disso, através deste ambiente pode-se executar várias animações concorrentemente, sendo possível a comparação entre algoritmos distintos para um mesmo problema. Uma interface gráfica amigável permite efetiva interação com as animações, provendo o usuário de facilidades para se comunicar com o sistema durante as animações, bem como de flexibilidade na configuração de suas telas. Há também grande facilidade na manutenção das bibliotecas do ambiente, que permite que o usuário programador inclua, exclua ou altere facilmente os componentes do **AnımA** através de uma interface gráfica que provê um mecanismo automático de pré-processamento e compilação.

A principal característica presente no ambiente é a possibilidade de crescimento incremental através do uso de *link* dinâmico e bibliotecas compartilhadas. Cada componente é uma biblioteca independente a ser ligada em tempo de execução, conforme a seleção do usuário final. Este esquema permite que vários usuários trabalhem no ambiente mantendo o seu próprio contexto de desenvolvimento independente. O arquivo executável é mantido estável, não precisando ser recompilado a cada inclusão de componente.

AnımA é escrito em C++ e faz extensivo uso de programação orientada a objetos para modelagem hierárquica de algoritmos, visualizadores e geradores de entrada. O ambiente executa em estações de trabalho SparcStation sob Sun/OS usando a biblioteca gráfica XView, seguindo a filosofia de interface gráfica OpenLook.

4.1 Componentes do **AnımA**

O **AnımA** destina-se a dois níveis de usuários. Em um primeiro nível está o usuário programador, responsável por programar as animações e incluí-las no ambiente. Em um segundo nível encontra-se o usuário final, que através de uma interface gráfica amigável configura, executa e interage com as animações, segundo suas necessidades.

Para ser animado pelo **AnımA**, um programa implementado pelo usuário programador deve ser subdividido em componentes, de modo que o ambiente possa manipulá-los sistematicamente. Cada programa deve ser composto por três módulos: o algoritmo em si, que consiste no código contendo as operações a serem realizadas; o gerador de entradas, que produz os dados que serão tratados pelo algoritmo; e o(s) visualizador(es), que produz(em) as apresentações visuais dinâmicas durante a animação.

Esta seção descreve alguns conceitos adotados no **AnımA**, desenvolvidos a partir de idéias introduzidas em [BR 87] [BR 88] [BS 84].

4.1.1 Algoritmos

Com o objetivo de prover o usuário final (vide seção 4.2.2) de facilidades de interação com as animações, o **AnımA** utiliza o conceito de algoritmo estruturado.

> **Algoritmo Estruturado:** é um algoritmo subdividido em métodos com funções específicas (inicialização, término, tratamento de parâmetros e o código do algoritmo propriamente dito), que são chamadas pelo sistema de animação de algoritmos. Com exceção do código do algoritmo, os demais são opcionais.

O usuário final pode interagir com as animações através de parâmetros. O código responsável pelo tratamento dos parâmetros para o algoritmo deve ser separado do código do algoritmo em si, de modo que o usuário final possa chamá-lo para a verificação dos valores correntes a

qualquer momento, sem interromper a animação em execução.

A implementação das rotinas de inicialização e término em métodos, separados do código do algoritmo, não resulta em ganhos consideráveis na interação com a animação, mas favorece o desempenho. No caso de um algoritmo ser executado repetidamente, algumas operações podem ser efetuadas apenas uma vez, reduzindo-se o *overhead*.

Durante uma animação, o mecanismo de atualização de visualizações (veja seção 4.1.3) é dirigido por eventos, definidos a seguir.

Evento: é uma operação do algoritmo que indica alguma forma de entrada de dados ou que reflete algum aspecto interessante de ser visualizado a respeito do funcionamento do algoritmo. Cada evento tem um nome e uma lista de parâmetros para a operação a que ele se refere.

Evento de Entrada: indica que o algoritmo deve receber dados, e que um gerador de entradas deve ser chamado para produzí-los. Os dados gerados são retornados em parâmetros passados por endereço.

Evento de Saída: indica cada operação que reflete o comportamento do algoritmo, e resulta na atualização das visualizações durante a animação.

Repertório de Eventos: é o conjunto de eventos de entrada e eventos de saída utilizados por um ou mais algoritmos.

Algoritmo Anotado: é um algoritmo estruturado acrescido de marcas adicionais indicando os eventos.

A cada evento está associado um método de tratamento existente em um gerador de entradas (caso o evento seja de entrada) ou em um visualizador (caso o evento seja de de saída). A figura 2 mostra um algoritmo de ordenação *(Selection Sort)* a ser animado pelo ambiente. O código está devidamente anotado conforme as especificações do **AnimA**, escrito em linguagem C++. Observe que os nomes dos métodos e as marcas que indicam os eventos seguem uma convenção específica.

No exemplo da figura 2, os eventos de entrada são denotados por marcas formadas pelo prefixo *"InputEvent."* seguido do nome do evento. Analogamente, os eventos de saída são denotados por marcas formadas pelo prefixo *"OutputEvent."* seguido do nome do evento. Observe que o algoritmo é modelado como uma classe (veja[LP 91]), em cujo construtor há duas marcas, indicando eventos. A primeira corresponde à geração dos dados de entrada, e a segunda à criação da primeira ilustração mostrando os dados gerados. Por estarem localizadas no construtor do algoritmo, estas chamadas são realizadas antes do início da animação (veja seção 4.2.2). A execução do algoritmo em si corresponde à chamada do método *alg_code()*.

4.1.2 Geradores de Entradas

A função básica de um gerador de entradas é fornecer os dados para o algoritmo. Ao ser detectado um evento de entrada durante a execução, o ambiente chama o método de tratamento correspondente no gerador de entradas selecionado, que pode tanto produzir o dado quanto solicitá-lo do usuário final.

Assim como os algoritmos, os geradores de entradas devem ser estruturados em métodos. Cada gerador de entradas é caracterizado pelo repertório de eventos de entrada por ele reconhecidos. Um algoritmo pode ser executado com diferentes geradores de entradas que respondam ao mesmo repertório.

```
#define length 500

typedef int type;

class Alg.selection
{
public:
     int k;
     int x [length];

     selection();
     void alg_code();
};

selection::selection()
{
     k = length;
     InputEvent.getlength (&k);
     InputEvent.fillvector (x, k);
     OutputEvent.showelements (x, k);
}

void selection::alg_code()
{
     int i, j, m;
     type temp;

     for (i = 0; i < k; i++)
       {
            m = i;
            for (j = i+1; j ≤ k; j++)
              if (x[j] < x[m])
                 m = j;
            OutputEvent.exchange(i, m, wait_time);
            temp = x[i];
            x[i] = x[m];
            x[m] = temp;
       }
}
```

Figura 2: Exemplo de Algoritmo Estruturado e Anotado

Gerador de Entradas Estruturado: é um módulo subdividido em métodos com as funções de inicialização (construtor), término (destrutor), tratamento de parâmetros para o gerador de entradas e geração de dados (métodos que tratam os eventos de entrada).

A comunicação entre o algoritmo e o gerador de entradas é feita pelo sistema de animação através dos eventos de entrada. Analogamente, o sistema faz a comunicação entre os geradores de entradas e os visualizadores (no caso do gerador solicitar dados de entrada ao usuário) através de mensagens que chamamos mensagens de entrada (veja seção 4.1.3).

A figura 3 mostra um gerador de entradas que responde aos eventos de entrada marcados no algoritmo do exemplo 2. Este gerador de entradas pode ser usado com o algoritmo de ordenação por seleção apresentado na figura 2, ou com qualquer outro algoritmo que use o mesmo repertório de eventos.

Observe a semelhança entre o formato deste módulo e o do algoritmo mostrado no exemplo 2. Isto ocorre porque o **AnımA** sugere ao usuário programador uma maneira homogênea para a implementação dos componentes.

Em um gerador de entradas estruturado, os métodos obrigatórios são os que geram dados em resposta a eventos de entrada. Na convenção utilizada no exemplo, os nomes destes métodos são precedidos de *"InputEvent."*. As demais rotinas são opcionais.

4.1.3 Visualizadores

O usuário programador (veja seção 4.2.1) também é responsável pela implementação dos visualizadores. Inicialmente, ele projeta o resultado visual desejado, seguindo algumas diretrizes de estética. Esta fase é de grande importância, pois para que uma animação mostre efetivamente o funcionamento do algoritmo, é crucial a produção de um resultado visual claro, preciso e agradável [BR 85] [BR 89].

```
#include <stdlib.h>
#include <iostream.h>

class Input.decreasing
{
  public:
    void InputEvent.getlength (int *);
    void InputEvent.fillvector(int *, int);
};

void decreasing::InputEvent.getlength (int *length);
{
    cerr << "Digite o numero de elementos do vetor:";
    cin >> *length;
}  /* getlength */

void decreasing::InputEvent.fillvector(int *vector, int max_index);
{
    int i;
    for (i = 0; i < max_index; i++)
      vector[i] = max_index - i;
}  /* fillvector */
```

Figura 3: Exemplo de Gerador de Entradas Estruturado

Concluída esta fase, o usuário programador implementa os visualizadores, que produzem os efeitos visuais com base nos eventos de saída do algoritmo. O sistema de animação se encarrega do gerenciamento de janelas, *menus*, e outros componentes de interface.

Cada visualizador é responsável por atualizar os *displays* de modo a refletir os eventos de saída do algoritmo durante a execução, e também a mapear coordenadas da tela para informações consistentes, caso o gerador de entradas solicite um dado ao usuário final.

Durante a fase de projeto do **AnımA**, foram analisadas diversas possibilidades para a implementação de visualizadores. Inicialmente, estudou-se o paradigma clássico de computação gráfica [FO 82], segundo o qual um visualizador é composto de um modelador e um desenhador. O modelador é responsável por manter as estruturas de dados que constituem o modelo da visualização.

Modelo: é uma representação abstrata das informações mostradas na visualização.

O desenhador é um módulo separado que produz as ilustrações baseado no modelo mantido pelo modelador. A cada evento de saída o modelador é acionado para realizar uma atualização do modelo, e o desenhador é chamado para refletir a mudança na visualização correspondente. Podem haver vários desenhadores para um único modelo, mostrando simultaneamente diferentes visualizações.

Esta abordagem foi utilizada em [BR 87]. Neste caso, o modelador e o desenhador respondem a mensagens de entrada e a mensagens de atualização.

Mensagem de Entrada: é uma mensagem emitida quando ocorre uma entrada de dados feita pelo usuário final, através de visualizadores.

Mensagem de Atualização: é uma mensagem emitida a cada vez que uma visualização deve ser atualizada.

Repertório de mensagens: é o conjunto de mensagens de entrada e mensagens de atualização utilizadas por um visualizador e/ou um gerador de entradas.

Em [BR 87] foi adotada ainda uma extensão do paradigma de [FO 82], introduzindo-se a figura do adaptador, um módulo que tem como funções: converter os eventos de saída do algoritmo para mensagens de atualização, através das quais o modelador atualiza o modelo e o desenhador atualiza a ilustração; transmitir as mensagens de entrada do gerador de entradas para o desenhador, quando um dado é solicitado ao usuário final; e retornar esse dado. Com a existência deste módulo, há maior possibilidade de reutilização dos componentes, podendo-se utilizar um mesmo modelador e/ou desenhador para se responder a diferentes eventos de saída. Portanto, em alguns casos, para produzir um novo visualizador basta que o usuário programador implemente um novo adaptador, o que geralmente é trivial, consistindo apenas em receber os eventos de saída e repassá-los ao modelador/desenhador na forma de mensagens.

O **AnımA** se utiliza de uma biblioteca de objetos gráficos elementares denominada **EGOLib** (Elementary Graphic Objects Library) implementada em linguagem C++, com chamadas a rotinas *XView* e *Xlib*. **EGOLib** [PA 93] implementa um conjunto de objetos gráficos com atributos e métodos próprios que facilitam consideravelmente o trabalho do usuário programador na implementação do desenhador. A figura 4 mostra os níveis de abstração do *Xlib*, *XView* e **EGOLib**.

Enquanto a biblioteca *Xlib* dispõe de um conjunto extenso e complexo de funções que manipulam atributos de janelas, desenhos gráficos, textos, cores, eventos de *mouse*, teclado, e outros, a **EGOLib** destina-se basicamente a desenhos de gráficos e textos e à manipulação dos atribu-

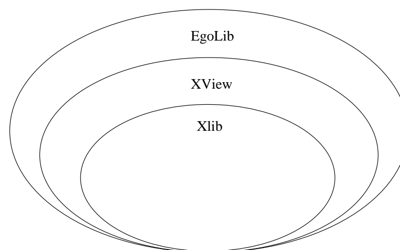


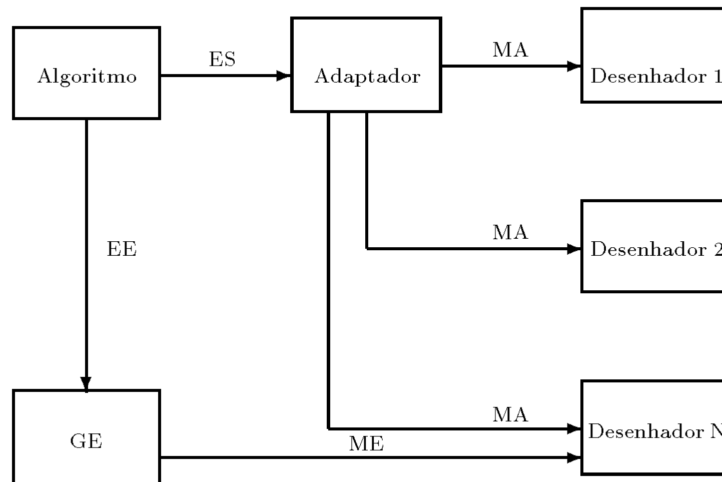
Figura 4: Níveis de abstração das rotinas gráficas

tos relacionados.

A implementação de um visualizador utilizando-se **EGOLib** envolve a criação de um contexto gráfico, sobre o qual uma lista de objetos gráficos é mantida internamente. Esta lista constitui o modelo do visualizador. Logo, o modelo está encoberto no nível do **EGOLib**, cabendo ao usuário programador implementar apenas o adaptador e os desenhadores.

Este esquema apresenta as vantagens de facilitar consideravelmente a tarefa do usuário programador, e prover o ambiente de maior segurança, uma vez que as estruturas de dados do modelo não são mantidas explicitamente. No caso de se ter vários visualizadores para uma mesma animação, poderia haver uma redundância de informações dando possibilidade de inconsistência. Porém, o **AnımA** garante que os modelos estarão sempre consistentes entre si, pois qualquer atualização é obrigatoriamente propagada para todos os desenhadores ativos. A figura 5 mostra a relação entre os componentes de uma animação no **AnımA**.

Os adaptadores e desenhadores também devem ser estruturados em métodos com funções específicas, como os algoritmos e geradores de entradas.



GE : Gerador de Entradas
EE : Evento de Entrada
ES : Evento de Saída
MA : Mensagem de Atualização
ME : Mensagem de Entrada

Figura 5: Componentes de uma animação

Adaptador Estruturado: é um adaptador subdividido em métodos com funções específicas (inicialização, término, tratamento de parâmetros e o tratamento de eventos de saída), que são chamadas pelo sistema de animação de algoritmos. Somente os métodos de tratamento dos eventos de saída são obrigatórios.

Desenhador Estruturado: é um desenhador subdividido em métodos com funções específicas (inicialização, término, tratamento de parâmetros e atualização), que são chamadas pelo sistema de animação de algoritmos. Somente os métodos de atualização são obrigatórios.

A implementação de desenhadores exige conhecimentos básicos em computação gráfica [FO 82], e pode envolver algoritmos relativamente complexos [LI 85] [RE 81] [WE 79]. No entanto, uma vez tendo sido implementados, tanto os adaptadores quanto os desenhadores ficam disponíveis em bibliotecas do ambiente, permitindo dois níveis de reutilização. Em um primeiro nível, cada um deles pode ser recombinado com outros componentes produzindo novos visualizadores. Em um segundo nível, cada visualizador pode ser reutilizado com quaisquer algoritmos e geradores de entradas que utilizarem os mesmos repertórios de eventos e de mensagens.

4.2 Interface

4.2.1 Manutenção das bibliotecas do ambiente

O usuário programador é responsável por implementar os módulos componentes da animação, e incluí-los no ambiente. A inclusão, exclusão ou alteração de um componentes das bibliotecas não afeta os demais já existentes.

Nos demais ambientes para animação de algoritmos, dedicou-se

grande parte do trabalho à interface com o usuário final (veja seção 4.2.2), com o intuito de facilitar a execução e a interação com as animações. O **AnimA** provê como novidade, além de uma poderosa interface com o usuário final, uma interface gráfica amigável com o usuário programador, para a manutenção automática das bibliotecas do ambiente. O usuário programador tem disponíveis as opções de inclusão, alteração e exclusão de problemas, repertórios (de eventos e de mensagens), algoritmos, geradores de entradas e visualizadores. As três últimas classes de componentes são programas fonte escritos na linguagem C++ aumentada segundo a especificação do **AnimA**, e após a sua inclusão o ambiente dispara uma fase de pré-processamento, gerando o código em C++ puro, que é compilado em seguida, guardando-se o programa objeto em uma biblioteca específica do ambiente, onde ele fica disponível para ser combinado com outros componentes. A figura 6 mostra exemplos de janelas para a inclusão de alguns componentes (neste caso, algoritmos, geradores de entradas e visualizadores).

Tendo em vista a possibilidade de reutilização de visualizadores e geradores de entradas, a tarefa do usuário programador para animar um algoritmo pode consistir apenas em estruturá-lo, e depois anotá-lo com os eventos interessantes.

Para permitir a independência de componentes é preciso apenas que o usuário programador se adeqüe a algumas especificações de interface, ou seja, o código do componentes deve seguir algumas regras de nomenclatura de métodos e algumas convenções para marcar os eventos.

4.2.2 Execução de animações

No nível de utilização (em oposição a programação) do sistema, temos o usuário final, que assiste e interage com as animações. Neste nível, o ambiente tem como objetivo principal permitir uma interação dinâmica com as animações, quaisquer que sejam os algoritmos observados. Através de uma interface homogênea, o usuário final seleciona os componentes, configura a janela de animação conforme as suas necessidades e controla a execução.

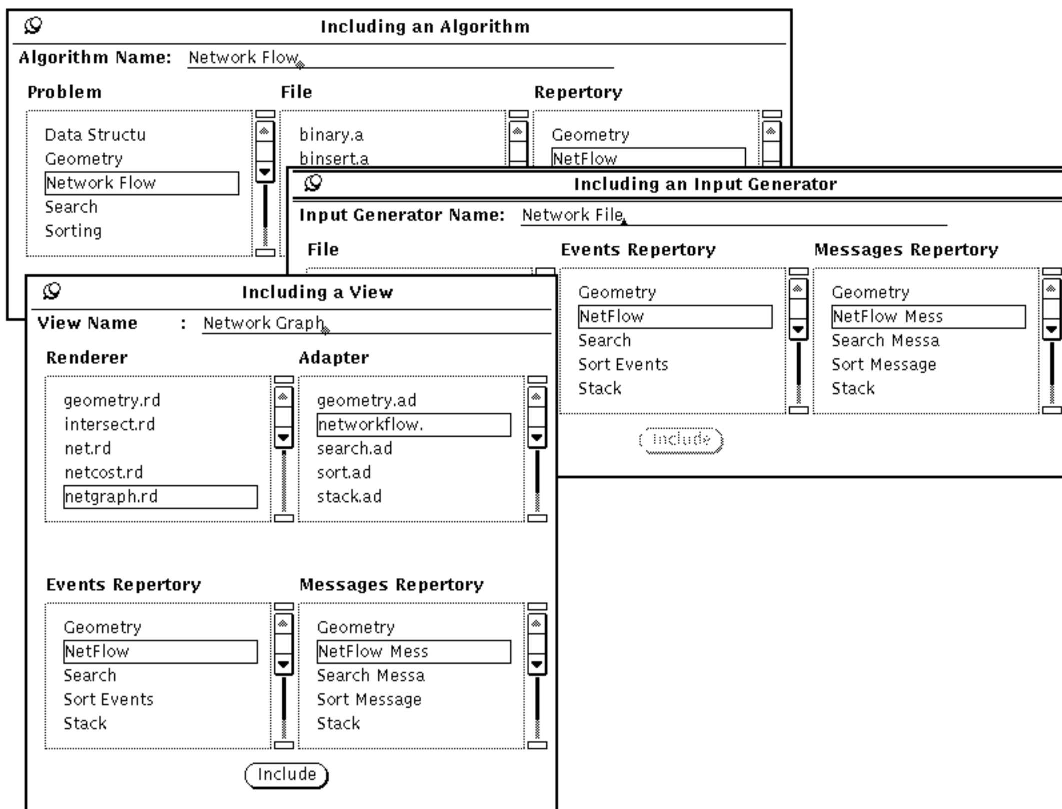


Figura 6: Janelas de inclusão de componentes

A uniformidade de interface constitui uma forte característica do **AnımA**, e permite que, uma vez tendo usado o ambiente para executar e interagir com uma animação, o usuário final possa controlar animações que utilizem quaisquer algoritmos, geradores de entradas e visualizadores existentes nas bibliotecas do ambiente.

Em uma sessão de animação, o usuário final pode estar em fase de *configuração* ou em fase de *execução*. Em fase de *configuração* ele adapta a tela conforme suas necessidades ou preferências, escolhe o problema, o algoritmo, o gerador de entradas e o(s) visualizador(es) desejados, e informa os valores de parâmetros, se desejado. Em fase de *execução*, ele dispara o algoritmo, podendo interrompê-lo temporariamente (através do botão *Pause/Resume*), executá-lo passo a passo (através do botão *Step*) ou encerrá-lo (botão *Abort*).

O usuário final interage com as animações através de parâmetros para o algoritmo, gerador de entradas e visualizadores. Ele pode, por exemplo, informar qual entre duas possíveis estruturas de dados deve ser utilizada em um algoritmo, ou escolher o modo de geração de um conjunto de números, ou ainda, especificar se as informações contidas em um nó de uma árvore devem ser expandidas ou se o sistema deve usar uma representação compacta para o mesmo durante a animação.

4.3 O Ambiente

Os componentes são tratados pelo ambiente em duas fases: o pré-processamento e a aplicação.

4.3.1 Pré-processamento

O pré-processamento é realizado antes da compilação do componente incluído pelo usuário programador. Ao final desta fase, tem-se o programa fonte sem as marcas iniciais, podendo-se compilá-lo com um compilador padrão e guardar o resultado em uma biblioteca.

Uma das grandes inovações introduzidas pelo **AnımA** é a utilização do recurso de ligação dinâmica de módulos. Cada módulo é compilado pelo ambiente, gerando uma biblioteca compartilhável *(shared library)* e

a ligação é feita no momento em que o usuário seleciona os componentes para uma animação.

A utilização do recurso de ligação dinâmica permite que o **AnımA** tenha a vantagem de, apesar de ser um ambiente cuja tendência é ser alimentado com cada vez mais componentes, possibilitando uma gama cada vez mais ampla de animações, não ser necessária a recompilação do ambiente a cada nova inclusão, e o seu código executável não aumentar de tamanho, proporcionando economia de memória e de armazenagem. Tipicamente, uma nova animação completa pode ser incorporada ao ambiente utilizando-se cerca de 100K bytes de novas bibliotecas compartilhadas, em oposição ao uso de 800K a 1M bytes para cada nova animação nos outros ambientes existentes. Isso certamente resulta em grande economia de recursos, considerando que usualmente temos várias dezenas de animações disponíveis.

4.3.2 Aplicação

Interface Gráfica A interface gráfica do **AnımA** constitui um ponto forte do ambiente. Está dividida em duas partes distintas, segundo os dois níveis de usuários. A primeira parte trata da interação com o usuário programador. Ela provê facilidades de inclusão, alteração e exclusão de componentes nas bibliotecas do ambiente, mantendo a consistência global. Uma vez tendo incluído um componente através da interface gráfica, o usuário programador não precisa se preocupar com a geração do código objeto. O ambiente dispara o pré-processamento e posteriormente a compilação do componente, deixando-o pronto para ser utilizado na produção de animações. A outra parte da interface destina-se ao usuário final, apresentando um conjunto de janelas e menus construídos dinamicamente com base nos componentes que já existem no **AnımA**, permitindo ao usuário final combiná-los adequadamente para produzir animações. Esta parte oferece várias opções para a configuração das janelas, além da possibilidade de resultados esteticamente agradáveis nas animações pelo uso de cores variadas e efeitos sonoros.

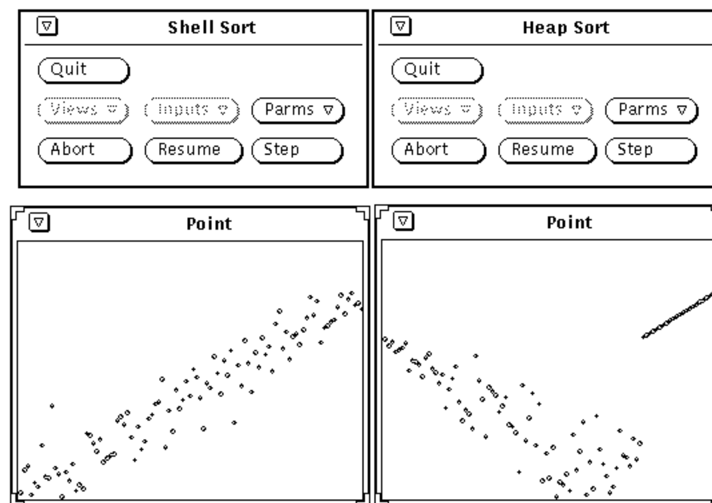


Figura 7: Animações de ShellSort e HeapSort com o mesmo gerador de entradas.

Execução Uma animação resulta da comunicação entre um algoritmo, um gerador de entradas e um ou mais visualizadores, realizada por um módulo controlador do **AnımA**. O **AnımA** permite que sejam executadas várias animações simultaneamente, através da utilização de *tasks*, um recurso oferecido pelo compilador de C++ da AT&T para a implementação de co-rotinas [CO 89].

5 Aplicações

O principal objetivo de um ambiente para animação de algoritmos é facilitar as etapas envolvidas na criação de animações para servirem de auxílio à pesquisa e ao ensino de ciência da computação, em especial no que se refere a projeto e análise de algoritmos e estruturas de dados [HO 74] [BS 84]. A figura 7 mostra a animação de dois algoritmos de ordenação utilizando duas formas de visualização.

Na pesquisa em análise e projeto de algoritmos, a observação visual do funcionamento de um algoritmo pode levar a novas descobertas, bem como a melhorias do mesmo, especialmente na determinação de *bottlenecks*, conjuntos de dados críticos, etc. Por exemplo, uma experiência com a animação de árvores de Huffman dinâmicas [KN 85] [BR 88] revelou um comportamento inesperado para uma determinada entrada, dando origem a uma melhoria no algoritmo [VI 85].

Em uma aula de computação auxiliada por animações, o instrutor pode preparar os algoritmos, geradores de entradas e visualizadores antecipadamente, ou deixar a cargo dos alunos escolherem dentre os componentes existentes nas bibliotecas do sistema.

Outra aplicação do ambiente de animação é a produção de ilustrações de estruturas complexas que aparecem em diversas áreas fundamentais da Ciência da Computação como geometria computacional, teoria dos grafos, compressão de dados, criptografia, buscas em árvores, ordenação, entre outras.

Conclusões

As aplicações de animações de algoritmos são inúmeras e a tecnologia para se explorar esta área já está disponível. Porém sem um ambiente de suporte adequado para o desenvolvimento de animações, o trabalho se torna demasiadamente complexo para que se objetive a produção de grande número de animações.

O desenvolvimento do ambiente **AnimA** vem mostrando que a produtividade cresce consideravelmente quando se congrega num mesmo ambiente ferramentas de suporte a animação como: interface gráfica uniforme, sistema de pré-processamento, manutenção automática de bibliotecas, reusabilidade de componentes através de programação orientada a objetos e utilização de bibliotecas compartilhadas e ligação dinâmica. Acreditamos que o **AnimA** cobre uma lacuna fundamental na área de animação de algoritmos.

Referências

[BR 84] Brown, Mark H.and Sedgewick, Robert, *Progress Report: Brown University Instructional Computing Laboratory*, ACM SIGCSE Bulletin, 16, 1, February 1984, 91-101.

[BS 84] Brown, Mark H.and Sedgewick, Robert, *A System for Algorithm Animation*, Computer Graphics, 18, 3, July 1984, 177-186.

[BR 85] Brown, Mark H.and Sedgewick, Robert, *Techniques for Algorithm Animation*, IEEE Software, 2, 1, January 1985, 28-39.

[BR 87] Brown, Mark H., *Algorithm Animation*, The MIT Press, Massachusetts, 1987.

[BR 88] Brown, Mark H., *Exploring Algorithms using BALSA-II*, IEEE Computer, May 1988, 14-36.

[BR 89] Brown, Mark H., *Zeus: A system for algorithm animation and multiview editing*, Proc. IEEE Workshop on Visual Languages, pages 4-9, 1991.

[BR 89] Brown, Mark H., *Color and sound in algorithm animation*, Proc. IEEE Workshop on Visual Languages, pages 10-17, 1991.

[CA 85] Carling, Richard T., Brown, Gretchen P., Heron, Christopher F., Kramlich, David A. and Souza, Paul, *Program Visualization: Graphical Suport for Software Development*, IEEE Computer 18, 8, August 1985, 27-35.

[CO 89] *Co-routine Style Programming*, in At&T C++ Language System Release 2.0, Library Manual, 1989.

[FO 82] Foley, James D. and Dam, Andries van, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

[HO 74] Hopgood, F. R. A., *Computer Animation Used as a Tool in Teaching Computer Science*, Proc. 1974 IFIP Congress, 1974, 889-892.

[KN 85] Knuth, Donald E., *Dynamic Huffman Coding*, Journal of Algorithms, 6, 2, June 1985, 163-180.

[LI 85] Lipton, Richard J., North, Steven C. and Sendberg, J. S., *How to Draw a Graph*, Proc. of the Symposium on Computational Geometry, June 1985, 153-160.

[LP 91] Lippman, S., *C++ Primer*, Addison-Wesley, Reading, MA, 2nd edition, 1991.

[PA 93] Patrocínio, Eduardo A., de Rezende, Pedro J., *EGOLib — Uma Biblioteca Orientada a Objetos Gráficos*, Anais do XX Semish, 1993.

[RE 81] Reingold, Edward M. and Tilford, J. S., *Tidier Drawings of Trees*, IEEE Transactions on Software Engineering, SE-7, 2, March 1981, 223-228.

[RE 86] Reiss, Steven P., *An Object-Oriented Framework for Graphical Programming*, Technical Report CS-86-17, Brown University, March 1986.

[RS 86] Reiss, Steven P., *Displaying Program and Data Structures*, Technical Report CS-86-19, Brown University, April 1986.

[RS 87] Reiss, Steven P., *GARDEN: An Environment for Graphical Programming - Reference and Programmers Manual*, Brown University, October 1987.

[VI 85] Vitter, Jeffrey S., *Design and Analysis of Dynamic Huffman Coding*, Proc. 26th Annual Symposium on the Foundations of Computer Science, October 1985, 293-302.

[WE 79] Wetherell, Charles and Shannon, Alfred, *Tidy Drawings of Trees*, IEEE Transactions on Software Engineering, SE-5, 5, September 1979, 514-520.

Relatórios Técnicos – 1992

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

07/92 **New Experimental Results For Bipartite Matching,** *J. C. Setubal*

08/92 **Maintaining Integrity Constraints across Versions in a Database,** *C. B. Medeiros, G. Jomier, W. Cellary*

09/92 **On Clique-Complete Graphs,** *C. L. Lucchesi, C. P. Mello, J. L. Szwarcfiter*

10/92 **Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms,** *T. Kowaltowski*

11/92 **Debugging Aids for Statechart-Based Systems,** *V. G. S. Elias, H. Liesenberg*

12/92 **Browsing and Querying in Object-Oriented Databases,** *J. L. de Oliveira, R. de O. Anido*

Relatórios Técnicos – 1993

01/93 **Transforming Statecharts into Reactive Systems,** *Antonio G. Figueiredo Filho, Hans K. E. Liesenberg*

02/93 **The Hierarchical Ring Protocol: An Efficient Scheme for Reading Replicated Data,** *Nabor das C. Mendonça, Ricardo de O. Anido*

03/93 **Matching Algorithms for Bipartite Graphs,** *Herbert A. Baier Saip, Cláudio L. Lucchesi*

04/93 **A lexBFS Algorithm for Proper Interval Graph Recognition,** *Celina M. H. de Figueiredo, João Meidanis, Célia P. de Mello*

05/93 **Sistema Gerenciador de Processamento Cooperativo,** *Ivonne. M. Carrazana, Nelson. C. Machado, Célio. C. Guimarães*

06/93 **Implementação de um Banco de Dados Relacional Dotado de uma Interface Cooperativa,** *Nascif A. Abousalh Neto, Ariadne M. B. R. Carvalho*

07/93 **Estadogramas no Desenvolvimento de Interfaces,** *Fábio N. de Lucena, Hans K. E. Liesenberg*

08/93 **Introspection and Projection in Reasoning about Other Agents,** *Jacques Wainer*

09/93 **Codificação de Seqüências de Imagens com Quantização Vetorial,** *Carlos Antonio Reinaldo Costa, Paulo Lício de Geus*

10/93 **Minimização do Consumo de Energia em um Sistema para Aquisição de Dados Controlado por Microcomputador,** *Paulo Cesar Centoducatte, Nelson Castro Machado*

11/93 **An Implementation Structure for RM-OSI/ISO Transaction Processing Application Contexts,** *Flávio Morais de Assis Silva, Edmundo Roberto Mauro Madeira*

12/93 **Boole's conditions of possible experience and reasoning under uncertainty,** *Pierre Hansen, Brigitte Jaumard, Marcus Poggi de Aragão*

13/93 **Modelling Geographic Information Systems using an Object Oriented Framework,** *Fatima Pires, Claudia Bauzer Medeiros, Ardemiris Barros Silva*

14/93 **Managing Time in Object-Oriented Databases,** *Lincoln M. Oliveira, Claudia Bauzer Medeiros*

15/93 **Using Extended Hierarchical Quorum Consensus to Control Replicated Data: from Traditional Voting to Logical Structures,** *Nabor das Chagas Mendonça, Ricardo de Oliveira Anido*

16/93 **$\mathcal{LL}$ – An Object Oriented Library Language Reference Manual,** *Tomasz Kowaltowski, Evandro Bacarin*

17/93 **Metodologias para Conversão de Esquemas em Sistemas de Bancos de Dados Heterogêneos,** *Ronaldo Lopes de Oliveira, Geovane Cayres Magalhães*

18/93 **Rule Application in GIS – a Case Study,** *Claudia Bauzer Medeiros, Geovane Cayres Magalhães*

19/93 **Modelamento, Simulação e Síntese com VHDL,** *Carlos Geraldo Krüger e Mário Lúcio Côrtes*

20/93 **Reflections on Using Statecharts to Capture Human-Computer Interface Behaviour,** *Fábio Nogueira de Lucena e Hans Liesenberg*

21/93 **Applications of Finite Automata in Debugging Natural Language Vocabularies,** *Tomasz Kowaltowski, Cláudio Leonardo Lucchesi e Jorge Stolfi*

22/93 **Minimization of Binary Automata,** *Tomasz Kowaltowski, Cláudio Leonardo Lucchesi e Jorge Stolfi*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

reltec@dcc.unicamp.br