UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

APLICAÇÃO DE TÉCNICAS DE ANTIALIASING COM PROCESSAMENTO DISTRIBUÍDO

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

RAFAEL LEONHARDT

BLUMENAU, JUNHO/2001

2001/1-58

APLICAÇÃO DE TÉCNICAS DE ANTIALIASING COM PROCESSAMENTO DISTRIBUÍDO

RAFAEL LEONHARDT

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Antônio Carlos Tavares — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Antônio Carlos Tavares

_____

Prof. Paulo César Rodacki Gomes

_____

Prof. Dalton Solano dos Reis

DEDICATÓRIA

Dedico este trabalho a minha namorada Cintia,

aos meus familiares

e aos meus amigos pelo apoio

durante a elaboração deste trabalho.

AGRADECIMENTOS

Ao Professor Antônio Carlos Tavares, pela paciência e pelo interesse com o qual orientou este trabalho.

Ao Professor José Roque Voltolini da Silva, coordenador do Trabalho de Conclusão de Curso.

A todos os professores e funcionários do Departamento de Sistemas e Computação que auxiliaram para que este trabalho pudesse ser realizado.

Aos colegas, que tive contato no decorrer do curso, que de alguma forma contribuíram para concluir mais uma etapa de minha vida.

E a todos que de alguma forma contribuíram para a realização deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

PiPL - *Plug-in Property List*

PVM – *Parallel Virtual Machine*

WPVM – *Windows Parallel Virtual Machine*

RESUMO

Este trabalho descreve a implementação de um *plug-in* para o programa gráfico *Adobe Photoshop*, que aplica técnicas de *antialiasing* sobre imagens *bitmap* através de processamento distribuído. A imagem *bitmap* é particionada e distribuída entre vários processadores cooperantes para execução paralela do algoritmo. O ambiente paralelo foi implementado utilizando a biblioteca WPVM.

ABSTRACT

This job describes Adobe Photoshop graphical software plug-in implementing. It applies antialiasing techniques on bitmap images through distributed processing. After bitmap image fractionating, it is distributed among several join processors to algorithmic parallel execution. The parallel behavior was implemented using the WPVM library.

1 INTRODUÇÃO

Com a evolução da informática, as quantidades de informações a serem processadas pelos computadores são cada vez maiores. Conseqüentemente, o tempo necessário para processar estas informações é cada vez maior. E devido à dinâmica de mercado estas informações necessitam ser obtidas mais rapidamente.

O desenvolvimento de processadores mais poderosos tem contribuído para minimizar o tempo de processamento das informações. Contudo, os custos deste tipo de *hardware* continuam relativamente altos, o que intensifica a procura de soluções eficientes embutidas no próprio *software* ou mesmo utilizando-se de *hardwares* combinados entre si.

Observando a área gráfica, verifica-se que esta gera um grande número de informação a ser processada, como por exemplo, fotos de satélites digitalizadas, fotos aéreas, entre outras.

Porém o processo de digitalização de imagens leva a degradações que podem criar efeitos visuais indesejáveis não existentes na cena original, chamados em inglês de *aliasing*. Métodos para reduzir os efeitos do *aliasing* são chamados de algoritmos de *antialiasing*.

As abordagens mais comuns de *antialiasing* são a de Pós-Filtragem, Pré-Filtragem, Superamostragem e Monte Carlo (*Jittering*), porém sendo relevante ao trabalho proposto somente às últimas duas.

A Superamostragem foi a primeira técnica de *antialiasing* baseada na integração numérica, envolvendo uma amostragem uniforme da cena a uma resolução mais alta que a desejada para a saída. Estas superamostragens são então transformadas numa média em grupos para a saída.

As técnicas de Monte Carlo não utilizam amostragens periódicas e deste modo não podem produzir os padrões de *aliasing* periódicos. Através delas introduz-se um ruído aleatório na saída. A magnitude deste ruído pode ser reduzida incrementando-se o número de amostras. Este ruído aleatório causa menos distúrbios visuais do que padrões regulares.

Atualmente, existem bons editores gráficos que aplicam técnicas de *antialiasing*, como o *Corel PhotoPaint* e o *Adobe Photoshop*, mas a grande maioria não possui soluções eficientes para aplicar determinadas técnicas sobre imagens de grandes proporções em tempo hábil. Ou seja, o tempo de processamento é bastante longo, para as imagens que ocupam grande área de memória, como por exemplo, as imagens de satélites.

Efetuando-se o processamento das informações, com o conceito de processamento paralelo, seria possível minimizar este tempo, devido estas serem processadas simultaneamente em vários computadores.

Segundo Strack (1984) a idéia de incluir paralelismo nas aplicações é tão antiga quanto os computadores eletrônicos. Trabalhos desenvolvidos por von Neumann na década de 40 já discutiam a possibilidade de algoritmos paralelos para a solução de equações diferenciais. Nas Universidades e nos centros de pesquisa, computação paralela e distribuída tem merecido um papel de destaque dentre os projetos de pesquisa em desenvolvimento.

As técnicas de emprego de paralelismo estão sendo desenvolvidas e aplicadas paulatinamente ao longo dos anos. Um importante marco foi a introdução dos processadores de Entrada e Saída (Canais) nos computadores de Segunda Geração. Isso motivou o aparecimento dos conceitos de concorrência, comunicação e sincronização: uma vez que dois processadores estão operando simultaneamente, surge a necessidade de prover mecanismos para sincronizá-los para estabelecer um canal de comunicação entre eles.

Então em 1989 nos laboratórios da *Emory University* e *Oak Ridge National Laboratory*, surgiu o *Parallel Virtual Machine* (PVM), que nasceu com o objetivo de criar e executar aplicações paralelas em um hardware já existente. Atualmente oferece como principal característica a interoperabilidade entre máquinas com sistema operacional UNIX.

O PVM teve grande difusão e foi aceito facilmente, contando com milhares de usuários e tornando-se, assim, um padrão de fato devido a sua flexibilidade, pois habilita uma coleção de computadores heterogêneos a comportarem-se como se fosse um único recurso computacional expansível e concorrente. Assim, grandes problemas

computacionais podem ser resolvidos através da agregação e compartilhamento de processadores e memórias de outros computadores com um custo efetivo menor.

Surge então o WPVM, uma biblioteca derivada da própria PVM, porém sendo possível utilizar em computadores com o sistema operacional W*indows* mantendo total interoperabilidade com a biblioteca PVM utilizada em computadores com sistema operacional UNIX.

Considerando o fato, de que algumas ferramentas gráficas disponibilizam o recurso de *plug-in*, pode-se desenvolver um *plug-in* utilizando-se o conceito de processamento paralelo.

Segundo Alspach (1995) *plug-in* é uma ferramenta incorporada ao programa, visando à inclusão de novos métodos e técnicas para aprimorar, facilitar e incluir novos recursos ao respectivo *software.*

A ferramenta gráfica *Adobe Photoshop*, é um *software* amplamente utilizado pelos profissionais da área gráfica e possibilita a inclusão de *plug-ins* externos ao mesmo, sendo assim este trabalho propõe desenvolver um *plug-in* para esta ferramenta, que irá aplicar as técnicas de *antialiasing* Superamostragem e Monte Carlo com o conceito de processamento paralelo.

O *plug-in* a ser implementado permitirá a distribuição da informação entre computadores através da biblioteca WPVM, os quais irão processar a informação recebida em paralelo para então retornar o resultado ao *plug-in* para que o mesmo efetue a junção destas informações, chegando ao resultado desejado, com performance superior aquela obtida utilizando um único computador mono-processador.

1.1 OBJETIVO

O objetivo geral do trabalho proposto é desenvolver um *plug-in* para o programa gráfico *Adobe Photoshop* para aplicar técnicas de *antialiasing* sobre imagens *bitmap* com processamento distribuído, utilizando a biblioteca WPVM.

Os objetivos específicos são:

a) desenvolver um *plug-in* para o programa gráfico *Adobe Photoshop*;

b) desenvolver um programa para aplicar as técnicas de *antialiasing* Superamostragem e Monte Carlo sobre uma imagem;
c) desenvolver um programa utilizando processamento distribuído.

1.2 ORGANIZAÇÃO DO TEXTO

O primeiro capítulo fornece uma introdução ao trabalho desenvolvido, demonstrando qual o objetivo do trabalho e apresentando os principais tópicos deste trabalho.

O segundo capítulo apresenta uma visão dos conceitos gerais do processamento distribuído bem como as vantagens ou não de utilizar processamento distribuído numa aplicação.

O terceiro capítulo apresenta uma breve história do WPVM, seus fundamentos básicos, e alguns exemplos de aplicações que utilizam os recursos da WPVM.

O quarto capítulo fornece uma visão do que é o *aliasing* e como funcionam algumas técnicas que tratam este problema.

No quinto capítulo são apresentados os módulos de *plug-in*s, o que é, o que pode fazer, como e onde funciona. São apresentados ainda, alguns tipos básicos de *plug-in*s suportados pela ferramenta gráfica *Adobe Photoshop*.

No sexto capítulo são apresentadas as especificações do protótipo, englobando o seu funcionamento e aspectos de implementação.

O sétimo capítulo faz uma análise conclusiva sobre o trabalho, inclusive apontando limitações e sugestões de extensões para este trabalho.

2 PROCESSAMENTO DISTRIBUÍDO

Atualmente, observa-se uma demanda cada vez maior por computadores de alto desempenho, devido a crescente complexidade das aplicações bem como a quantidade de informações a serem processadas.

Pesquisas voltadas para o aprimoramento do hardware vem conseguindo melhorar a performance dos mesmos. Contudo, os custos do hardware continuam relativamente altos, o que intensifica a procura de soluções eficientes embutidas no próprio software ou mesmo utilizando-se de hardwares combinados entre si.

Um ambiente de processamento distribuído consiste de múltiplas unidades processadoras trabalhando em conjunto para a realização de uma determinada tarefa.

Sendo assim, o ambiente de processamento distribuído destaca-se dos ambientes convencionais, pelo fato de apresentar desempenho superior.

2.1 HISTÓRICO

Segundo Wijegunaratne (1998), de 1955 a 1970 o processamento de dados resumiam-se ao conceito de processamento centralizado, onde toda a informação era processada por um único centro computacional, o processamento distribuído não passava de um conceito.

Após 1970, surgiram os primeiros sinais de processamento distribuído, com os terminais remotos ou minicomputadores que efetuavam tarefas independentemente e quando necessário se conectavam com o *mainframe*.

Até meados de 1980, destacavam-se três tipos de processamento distribuído:

a) Estrutura em Estrela: terminais localizados remotamente da instalação central e conectavam-se com a central via modem;
b) Distribuição Hierárquica: conglomerados de terminais remotos conectavam-se a um minicomputador, e este por sua vez, quando necessário se conectava com o *mainframe*;
c) Rede Distribuída em Anel: computadores independentes interligados de modo ponto-a-ponto.

Após a década de 80, o processamento distribuído já despertava o interesse da indústria de tecnologia, conseqüentemente outros conceitos surgiram, os quais alguns são bastante utilizados hoje, por exemplo, o conceito de cliente/servidor.

2.2 ALGORITMOS PARA PROCESSAMENTO DISTRIBUÍDO

Segundo Dorow (1997), uma aplicação que utilize o conceito de processamento distribuído é mais difícil de projetar, pois, é necessário prever o comportamento de diversos processos a serem executados em arquiteturas diversas.

Há a necessidade do algoritmo primar pela simplicidade. Algoritmos demasiadamente extensos são de difícil compreensão. Por isso, a grande parte dos algoritmos compõem-se de pequenos programas.

Outro fator importante é a portabilidade do código, devido que, ao optar por uma linguagem com características de abstração, o problema de transferência para outros ambientes computacionais pode ser resolvido mais rapidamente sem a necessidade de reescrever o código.

2.3 VANTAGENS E DESVANTAGENS

Segundo Amaral (1999), as principais vantagens do processamento distribuído são:

a) Autonomia Local: A distribuição do sistema permite aos grupos individuais exercerem um controle local sobre os seus próprios dados, com contabilidade local e, de maneira mais geral, que se tornem menos dependentes de um centro de processamento de dados remoto e ao mesmo tempo permite aos grupos locais o acesso aos dados de outras localidades, quando necessário.
b) Crescimento incremental: Um sistema distribuído pode crescer mais facilmente que um sistema centralizado. Se é necessário expandir o sistema porque o volume de dados cresceu ou o volume de processamento aumentou, é mais fácil acrescentar um novo nó a rede de computadores, desde que os nós sejam autônomos, do que substituir um sistema centralizado já existente por outro maior.

c) Confiança e disponibilidade: Um sistema distribuído oferece maior confiança do que um sistema centralizado, visto que o mesmo não é uma proposição de tudo-ou-nada - o sistema continua funcionando (a um nível reduzido) em caso de avaria em localidade individual ou de ligação de comunicações individuais entre as localidades. No caso de existência da réplica de dados, a disponibilidade é aperfeiçoada, porque um determinado objeto de dados permanece disponível à medida que pelo menos uma cópia daquele objeto esteja disponível.
d) Eficiência e flexibilidade: Os dados podem ser armazenados no sistema distribuído próximo ao seu ponto normal de uso, reduzindo, desta forma, o tempo de resposta e os custos de comunicações (a maioria dos dados devem ter acesso local). O paralelismo inerente nas redes de localidades múltiplas pode fornecer uma passagem de dados aperfeiçoada e, possivelmente, melhorar os tempos de resposta em certas situações.

Principais desvantagens:

a) Tecnologia ainda não dominada;
b) Aplicações: complexidade das aplicações; linguagens não específicas para o processamento distribuído;
c) Baixa velocidade em redes de longa distância: em comparação com a velocidade de leitura dos discos;
d) Padronização mais rígida: protocolos; compatibilidade de sistemas não homogêneos.
e) Grande potencial de falhas: desde os nós que possibilitam o sistema distribuído de operar em paralelo, é mais difícil assegurar a correção dos algoritmos.
f) Custo do desenvolvimento de software: é mais difícil implementar um sistema de banco de dados distribuídos e, assim, mais custoso.
g) Aumento do *overhead* de processamento**:** a troca de mensagens e a computação adicional requerida para atingir a coordenação entre os nós são a forma de *overhead* que não aparecem em sistemas centralizados.

3 WINDOWS PARALLEL VIRTUAL MACHINE

3.1 INTRODUÇÃO

O ambiente distribuído é constituído de diversos tipos de arquiteturas e plataformas de processamento. A integração destas arquiteturas e plataformas em uma rede heterogênea para a execução de programas paralelos, defrontam-se com alguns obstáculos decorrentes destas diferenças.

Surge no verão de 1989, uma alternativa para o processamento paralelo, um protótipo, denominado PVM 1.0, criado por Vaidy Sunderan e Al Geist, no Oak Ridge National Laboratory, com o intuito de interligar plataformas heterogêneas com sistema operacional UNIX. Com problemas como a falta de portabilidade do código e interface precária, o protótipo ficou restrito ao próprio laboratório. Em março de 1991, é liberada a segunda versão, a qual foi escrita na Universidade do Tennessee, obtendo apenas algumas melhorias no código. A partir desta versão, diversas aplicações científicas a utilizaram. Em fevereiro de 1993, é liberada a terceira versão.

Para ampliar ainda mais a utilização do PVM, que até o momento era limitado a arquiteturas com sistema operacional UNIX, surge na Universidade de Coimbra, em Portugal, um protótipo derivado do PVM, o WPVM (*Windows Parallel Virtual Machine*), com o objetivo de interligar computadores pessoais com sistema operacional Windows versão 3.1 até a versão NT. A implementação WPVM mantém total compatibilidade com o PVM Versão 3, ou seja, é possível executar programas tanto no UNIX quanto no Windows, apenas portando o código para cada sistema operacional.

3.2 FUNDAMENTOS BÁSICOS

WPVM é uma ferramenta para criar e executar aplicações paralelas ou concorrentes. Ela permite que uma rede heterogênea de computadores seja usada como um computador paralelo com memória distribuída, ou seja, uma máquina virtual paralela (*Parallel Virtual Machine*). A comunicação é realizada através de um mecanismo de troca de mensagens (*Message Passing*). O WPVM possui dois componentes principais: o WPVM Daemon (wpvmd) e as bibliotecas WPVM.

O WPVM Daemon é um processo que vigia a operação de processos usuários dentro de uma aplicação WPVM, e coordena a comunicação entre máquinas. O WPVM Daemon é executado em cada máquina onde o WPVM está configurado. Outros usuários com suas próprias máquinas virtuais paralelas, terão seus próprios Daemons executando. Cada Daemon mantém uma tabela de configuração e informação sobre processos relativos a sua própria máquina virtual paralela. Os processos se comunicam entre si através dos Daemons. Para que um processo se comunique com outro, ele primeiro se comunica com o seu Daemon local através de rotinas de interface das bibliotecas WPVM. Depois, o Daemon local envia e recebe mensagens dos Daemons de máquinas remotas. Cada máquina deve possuir sua própria versão do Daemon, de acordo com a compilação para aquela arquitetura. O esquema de controle dos Daemons se baseia no modelo mestre-escravo

As bibliotecas WPVM possuem chamadas de sub-rotinas simples que o programador pode incluir no código de seus programas paralelos. Essas bibliotecas permitem criar e matar processos; empacotar, enviar e receber mensagens; sincronizar processos; e ainda consultar e mudar dinamicamente a configuração da máquina virtual paralela. Essas rotinas não se comunicam diretamente com outros processos. Em vez disso, elas se comunicam com o Daemon local e este realiza todo o trabalho, retornando informações de estado.

Existem algumas estruturas que são comuns a todos os programas WPVM escritos em C. Todo programa deve incluir o arquivo de cabeçalho do WPVM e suas bibliotecas. Os quais possuem informações necessárias sobre a interface de programação WPVM.

A primeira rotina geralmente chamada num programa, é a pvm_mytid(), que registra o processo no WPVM. Assim como todas as rotinas WPVM, pvm_mytid() retornará um número negativo se algum erro ocorrer. Os programas podem testar os códigos de erro retornados nas chamadas de funções e tomar as decisões apropriadas. A última linha de comando de um programa WPVM deve ser pvm_exit() o qual informa ao WPVM Daemon que terminou o processamento.

A comunicação do WPVM Daemon com a aplicação, e aplicação com aplicação, é conseguida através do mecanismo de troca de mensagens (*Message Passing*).

Para enviar uma mensagem de um processo A para um processo B, o processo A deve primeiro chamar a rotina *pvm_initsend()*. Isso limpa o buffer *default* de envio de dados. Depois, o processo transmissor (A) deve então empacotar todos os dados que ele deseja enviar para o processo receptor (B). O empacotamento é feito através da família de rotinas *pvm_pack()*.

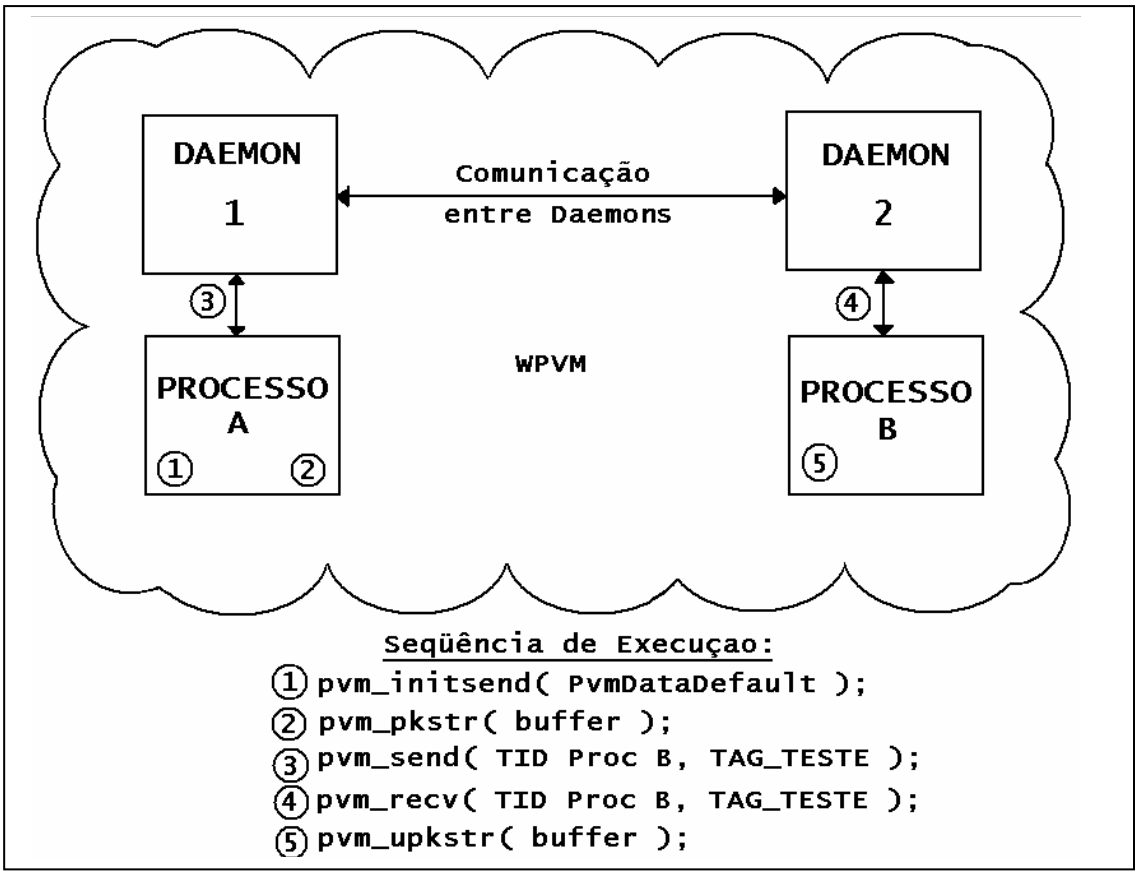
Depois que os dados foram empacotados no buffer de envio de dados, a mensagem está pronta para ser enviada. Para isso, basta chamar a rotina pvm_send(). A linha de comando info = pvm_send(tid, msgtag) enviará os dados contidos no buffer para o processo com identificação *tid*. O campo *msgtag* é usado para informar ao processo receptor qual tipo de dados ele está recebendo. A rotina *pvm_mcast* é similar à *pvm_send*. A diferença é que *pvm_mcast* passa um vetor de *tids* como parâmetro em vez de apenas um. Isso pode ser útil quando se quer enviar uma mensagem para um conjunto de processos (*multicasting*).

Para receber a mensagem, o processo receptor deve chamar a rotina *pvm_recv()*. A linha de comando info = pvm_recv(tid, msgtag) vai esperar por uma mensagem de um processo cuja identificação é *tid* com um *tag msgtag*. Valores -1 para ambos os parâmetros implica que o processo receberá mensagens de qualquer processo como qualquer *tag*.

Quando um processo recebe uma mensagem, ele deve desempacotar os dados contidos nela para o buffer de recebimento. Isso pode ser feito através da família de rotinas *pvm_unpack()*. Os dados devem ser desempacotados na mesma ordem em que foram empacotados.

A Figura 1 ilustra o procedimento de transmissão e recepção de uma mensagem entre dois processos conforme descrito acima.

Figura 1 – Troca de mensagem entre dois processos



A execução de tarefas em diferentes processadores é conseguido através da rotina pvm_spawn().

WPVM provê muitas funções úteis para colher informações. Alguns exemplos são *pvm_parent*, *pvm_config*, *pvm_tasks*, etc. Estas rotinas podem dar acesso a informações disponíveis no console do WPVM.

3.3 ALGORITMOS WPVM

Segundo Dorow (1997), a programação para WPVM consiste basicamente, na construção da máquina virtual, controle de processos e comunicação entre processos.

Na construção da máquina virtual, define-se quem irá fazer parte da máquina virtual ou quem será excluído da mesma. Para efetuar tais tarefas são utilizadas funções como pvm_addhost() para adicionar computadores (hosts) e pvm_delhost(), para remover.

Para o controle de processos, o modelo de comunicação do WPVM assume que uma tarefa pode enviar uma mensagem para outra tarefa qualquer, tendo como limite de tamanho da mensagem a memória física disponível em cada buffer.

A comunicação entre processos é feita através de mensagens, as quais são transmitidas/recebidas através de buffers, que constituem depósitos de dados. O processo de comunicação envolve as seguintes etapas:

a) inicialização do buffer a ser enviado, através da função pvm_initsend();
b) empacotamento da mensagem, através da função pvm_pk();
c) envio da mensagem a outro processo, através da função pvm_send();
d) recebimento da mensagem, através da função pvm_recv();
e) desempacotamento da mensagem, através da função pvm_upk().

A seguir os Quadros 1 e 2 apresentam o código fonte de dois exemplos de programas que utilizam a biblioteca WPVM. O programa “hello”, imprime o número de identificação da tarefa, e inicia o programa “hello_other”, o qual irá obter o nome do *host* e enviar para o programa “hello” este nome e mais uma mensagem. Então após o recebimento da mensagem, “hello” exibirá a mensagem “Olá mundo, de” e o número de identificação da tarefa, o TID.

Quadro 1 – Código fonte do programa de exemplo Hello.

```
/* PROGRAMA EXEMPLO : HELLO */
#include <stdio.h>
#include "wlpvm.h"          // Incluido arquivo de cabecalho das funcoes WPVM
#include "shell.h"          // Incluido arquivo de incializacao do WPVM

void main() {
        int cc, tid;
        char buf[100];
        char *args[]={"arg1",NULL};
        printf("Meu TID t%x\n", pvm_mytid()); // Exibe a identificacao do processo WPVM
        cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid); // Cria novo processo

        if (cc == 1) {
                // Verifica se recebeu mensagem do processo especificado pela variável tid
                cc = pvm_recv(-1, -1);
                // Ao receber msg, desempacota e exibe na tela.
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("de t%x: %s\n", tid, buf);
        } else printf("Não foi possivel acionar o programa hello_other\n");
        // Avisa o WPVM Daemon que o processo foi encerrado
        pvm_exit();
        exit(0);
}
```

Quadro 2 – Código fonte do programa de exemplo Hello_other.

```
/* PROGRAMA EXEMPLO: HELLO_OTHER
#include "wlpvm.h"          // Incluido arquivo de cabecalho das funcoes WPVM
#include "shell.h"          // Incluido arquivo de incializacao do WPVM

void main(int argc, char *argv[]) {
        int ptid,tid;
        char buf[500];

        tid = pvm_mytid(); // Obtem identificacao do processo
        ptid = pvm_parent(); // Obtem identificacao do processo que criou este processo

        strcpy(buf, "Ola mundo, de ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault); // Inicializa mensagem
        pvm_pkstr(buf); // empacota mensagem
        pvm_send(ptid, 1); // Envia mensagem ao processo especificado na variavel tid

        pvm_exit(); // Avisa WPVM Daemon que encerrou execução
        exit(0);
}
```

4 ANTIALIASING

4.1 INTRODUÇÃO

Segundo Sun (1991), o processo de geração de imagens de alta definição implica em se desenhar em dispositivos matriciais, o que envolve essencialmente a conexão dos *pixels* com linhas. Estas linhas devem adaptar-se à geometria do *grid* formado pelos *pixels*. Com exceção das linhas diretas que correm paralelas aos eixos x ou y, a maioria das linhas e curvas cruzam a grade matricial entre os *pixels*, ao invés de atravessá-los diretamente. Devido à orientação da linha, há uma ambigüidade com relação a quais *pixels* devem formar a linha, gerando um problema, já que o sistema gráfico necessita conhecer quais *pixels* deve iluminar.

Segundo Banon (1989), esse problema leva a degradações que podem criar efeitos visuais indesejáveis, isto é, não existentes na cena original, chamados em inglês de *aliasing*. Este é um problema potencial sempre que um sinal analógico é amostrado em pontos para digitalização.

Segundo Kaushik (1997), os erros causados pelo *aliasing* são denominados artefatos. Artefatos comuns de *aliasing* incluem o efeito serrilhado, o desaparecimento ou interpretação imprópria de detalhes e a desintegração de texturas.

Métodos para reduzir os efeitos do *aliasing* são chamados de algoritmos de *antialiasing*. Estes algoritmos têm vantagem em relação aos métodos de exibição matricial no que tange a capacidade de selecionar um conjunto de valores de tons de cor para cada *pixel*. Se os *pixels* que representam uma linha são ativados na sua iluminação máxima e todos os *pixels* vizinhos desativados, o contraste é alto, e assim o efeito denteado é visível. A idéia básica implícita nos algoritmos de *antialiasing* é que uma linha parece menos denteada se a linha estiver um pouco obscurecida. A solução é variar a intensidade dos *pixels* que são afetados no cálculo de uma linha ou curva.

4.2 CONCEITOS BÁSICOS

Todos os gráficos matriciais estão sujeitos ao fenômeno do *aliasing*. Principalmente no caso das curvas desenhadas em telas matriciais, porque a linha curva,

por natureza, não se adapta à malha retangular e precisa ser aproximada por um conjunto de pontos matriciais.

O grau de *aliasing* geralmente depende de uma série de fatores. Um é a resolução da tela matricial. Com um monitor de alta resolução, é difícil notar que a tela é composta por pequenos pontos. Em monitores de baixa resolução, o *aliasing* é ampliado, porque os *pixels* são maiores.

No processo de interpretação de imagens os pontos utilizados para estimar as intensidades de luz são infinitamente estreitos. Mesmo assim, cada *pixel* de uma imagem interpretada tem uma largura finita. A interpretação, em seu aspecto básico, supera esta incompatibilidade rastreando um único raio primário através do centro de cada *pixel* e utilizando a cor no *pixel* completo.

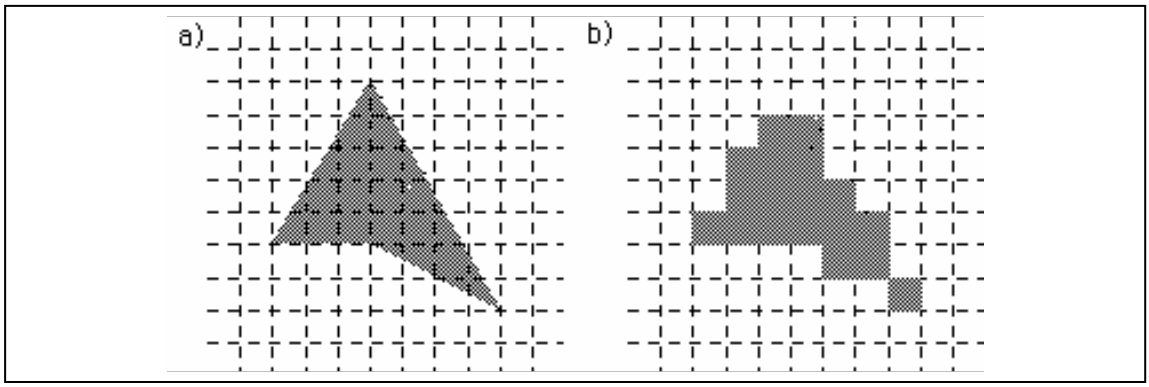
Tendo em vista que a cor resultante de cada *pixel* é baseada em uma amostragem pequena tomada do centro do *pixel* e porque os *pixels* estão dispostos em intervalos de freqüência regular muitas vezes surgem problemas de *aliasing*. O *aliasing* se refere à inclusão de características ou artefatos em uma imagem.

4.3 TIPOS COMUNS DE ALIASING

4.3.1 EFEITO SERRILHADO

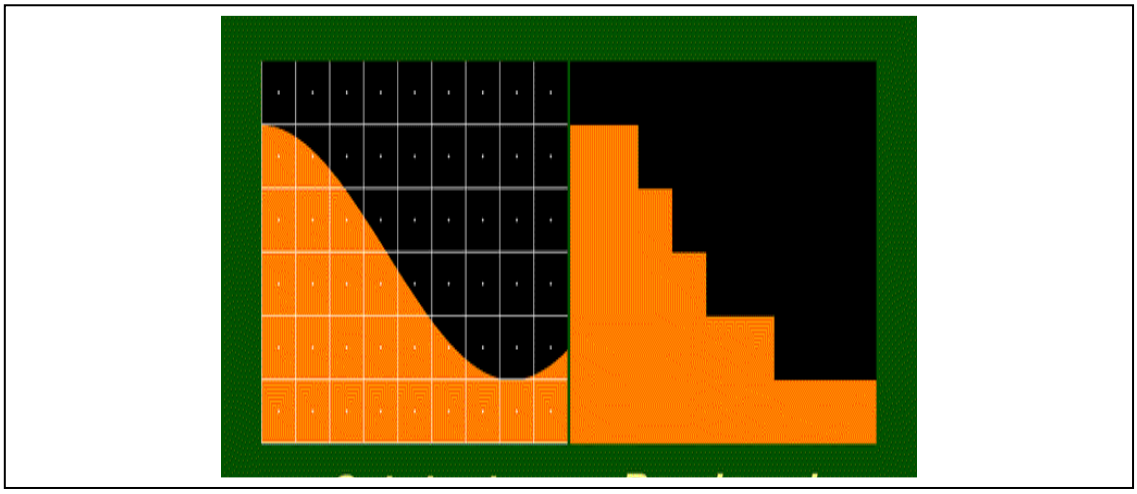
Também conhecidas como *jaggies*, silhuetas serrilhadas são provavelmente os efeitos mais comuns causado pelo *aliasing*. Esse efeito é a forma predominante de *aliasing* e ocorre em regiões onde há mudanças bruscas de intensidade. Exemplos destas regiões são limites de objetos, sombras ou limites de realces contrastantes. A Figura 2 ilustra o efeito serrilhado onde (a) mostra o exemplo que deveria aparecer e (b) mostra o resultado interpretado. As linhas traçadas são os limites dos *pixels*.

Figura 2 - Exemplo de efeito serrilhado.



A maioria das bordas dos objetos são sombreadas incorretamente, tanto em seu interior quanto em sua parte externa. A imagem a seguir (Figura 3) mostra uma matriz de amostragem superposta à cena original. Na imagem interpretada, à direita, o efeito serrilhado é evidente.

Figura 3 - *Jaggies*.

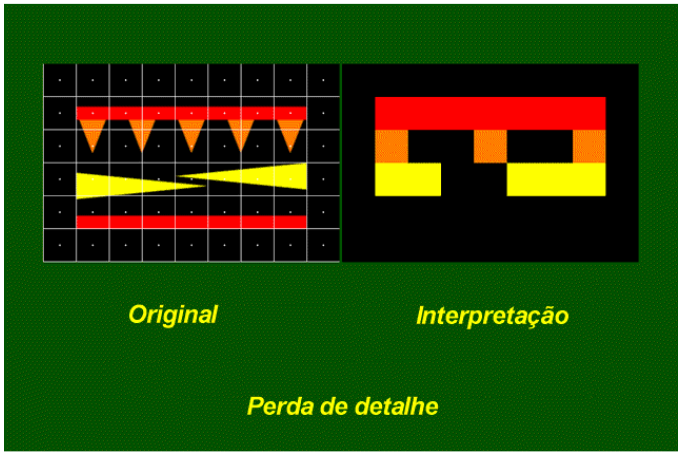


4.3.2 INTERPRETAÇÃO ERRÔNEA DE DETALHES

Segundo Kaushik (1997), outro efeito típico é a interpretação errônea de detalhes, como no caso da Figura 4 em que a cena original à esquerda mostra um grupo de polígonos pequenos.

Na cena interpretada, um dos dois retângulos vermelhos desaparece inteiramente, e o outro dobra de largura. Dois dos triângulos alaranjados desaparecem. Além disso, os dois triângulos amarelos são idênticos em tamanho, um é maior que o outro na imagem interpretada.

Figura 4 - Interpretação errônea de detalhes.

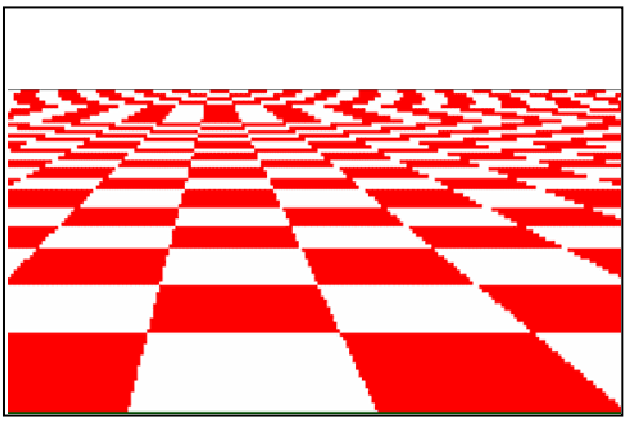


4.3.3 DESINTEGRAÇÃO DE TEXTURAS

Em outra situação, ilustrada na Figura 5, tem-se uma textura quadriculada sobre um plano. Os quadrados deveriam se tornar menores conforme a distância do observador aumenta.

Ao invés disso, os quadrados tornam-se maiores ou irregularmente formados quando sua distância do observador aumenta. Simplesmente aumentar a resolução não irá remover este artefato, mas apenas deslocará o artefato mais em direção ao horizonte. Este artefato é conhecido como desintegração de texturas.

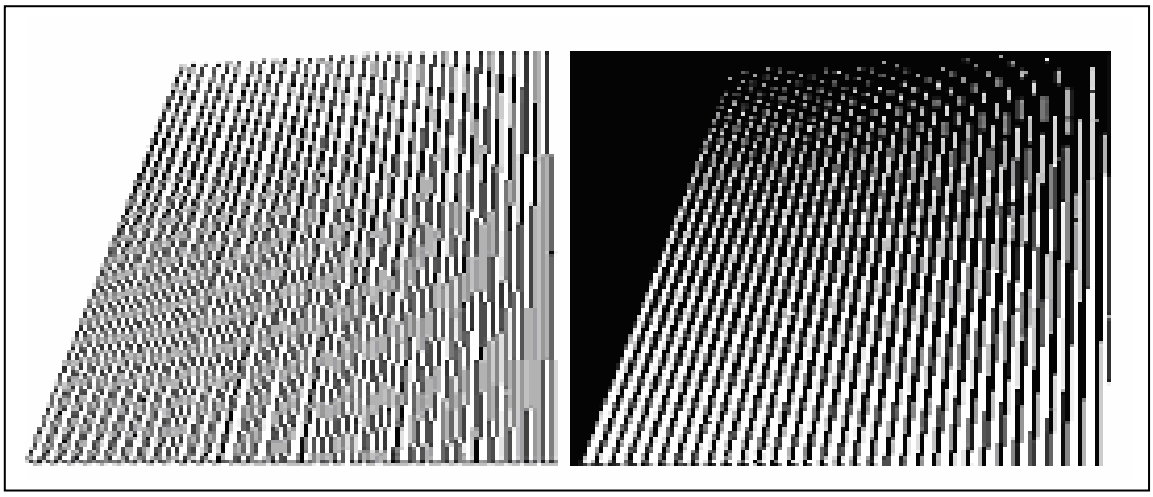
Figura 5 - Desintegração de texturas.



4.3.4 INTERFERÊNCIA

Segundo Hill (1990), outro importante problema gerado pelo *aliasing* ocorre transversalmente nas regiões de uma imagem cujos objetos componentes se alternam a uma freqüência alta. As amostras tomadas podem, algumas vezes, gerar imagens contendo um padrão de interferência denominado Moiré. A Figura 6 ilustra dois exemplos de imagens que contém padrão de interferência Moiré.

Figura 6 - Padrão de interferência *Moiré*.



4.4 ALGORITMOS DE *ANTIALIASING*

O *antialiasing* é absolutamente essencial na síntese de imagens. Apenas aumentar a resolução da amostragem não substitui adequadamente a filtragem adequada.

Segundo Kaushik (1997), para realizar um *antialiasing* perfeito, o sinal contínuo deve ser filtrado antes da amostragem eliminando qualquer detalhe que seja muito pequeno. As bordas em forma de degraus ou "*jaggies*" são um exemplo do *aliasing* causado pela amostragem sem filtragem.

Todas as técnicas de *antialiasing* utilizam alguma forma de obscurecimento ou suavização da imagem para reduzir o efeito do *aliasing* e todos os algoritmos de interpretação de imagem se classificam em duas categorias: técnicas de filtragem contínua e filtragem discreta.

Técnicas de filtragem contínua: quantifica-se diretamente o todo envolvido na filtragem. Isto requer um algoritmo de superfície continuamente visível, pois o todo é gerado antes da amostragem. Esta categoria atua pré-filtrando a imagem e extraindo suas freqüências altas antes da amostragem dos valores dos *pixels*. Todavia, as explanações disponíveis sobre o assunto são relativamente complexas, sendo de difícil compreensão e, conseqüentemente, dificultando sua implementação.

Técnicas de filtragem discreta: utiliza-se uma técnica de integração numérica que combine várias amostras do sinal de entrada (usualmente a uma taxa de amostragem mais alta) para aproximar-se do todo desejado. Têm-se alguns erros com esta técnica, mas este erro tende a se aproximar de zero conforme o número de amostras aumenta. Nesses algoritmos a geração e amostragem da imagem é interligada, sendo que há uma função contínua da imagem que é amostrada. Não se pode inserir um filtro de *antialiasing* que opere no domínio contínuo da informação.

Tendo relevância para este trabalho, apenas as técnicas de filtragem discreta, das quais duas técnicas serão analisadas, a técnica de Superamostragem e a de Monte Carlo.

4.4.1 TÉCNICA SUPERAMOSTRAGEM

A Superamostragem foi a primeira tentativa de *antialiasing* baseada em integração numérica e envolve uma amostragem uniforme da cena a uma resolução mais alta que a desejada para a saída. Estas superamostras são então transformadas numa média em grupos para a saída. O *aliasing* continua presente nas superamostras devido às freqüências maiores que metade da taxa de Superamostragem. Estes componentes de freqüência com *alias* usualmente tem magnitude menor conforme a freqüência aumenta, assim a Superamostragem usualmente reduz o *aliasing*.

As superamostras continuam contendo freqüências que são maiores que metade da freqüência de amostragem de saída. Estas freqüências devem ser filtradas quando o sinal é reamostrado para a taxa de amostragem menor.

A média de peso constante usualmente utilizada para esta função faz um bom trabalho. É mais veloz que outros métodos porque não há cálculos ponderados e o domínio do filtro para cada *pixel* não atinge *pixels* adjacentes e, deste modo, cada amostra contribui para apenas um *pixel*. Filtros mais elaborados farão um trabalho

melhor. Se uma função ponderada mais complexa for desejada, tabelas resumidas computadas previamente para os pesos das superamostras regularmente espaçadas irão gerar um aumento significativo de velocidade.

A Superamostragem não elimina o *aliasing*, mas meramente aumenta a freqüência na qual o *aliasing* começa.

A Superamostragem uniforme sobre a imagem inteira pode se tornar muito dispendiosa. O próximo avanço nesta técnica é a amostragem adaptativa. Áreas das imagens onde transições de intensidade são detectadas são amostradas a uma resolução maior que áreas com pouca variação. As superamostras são então transformadas em média ponderada de acordo com a área que cada superamostra representa. O sucesso da amostragem adaptativa se baseia na habilidade de detectar áreas que necessitem uma amostragem mais precisa. Erros nesta detecção podem ocorrer e causarão artefatos visíveis.

Segundo Hill (1990), utilizando-se uma função ponderada que é uma função do local da superamostra, bem como, da taxa de amostragem local permite-se uma melhor filtragem entre as amostras na taxa de Superamostragem e na taxa de amostragem de saída.

4.4.2 TÉCNICA MONTE CARLO

As distribuições de amostras consideradas (exceto a amostragem adaptativa) são todas periódicas. Como qualquer padrão de amostragem periódica, isto pode gerar um efeito de *aliasing* periódico sobre grandes segmentos da imagem. Estes padrões são muito perceptíveis e causam distúrbios visuais.

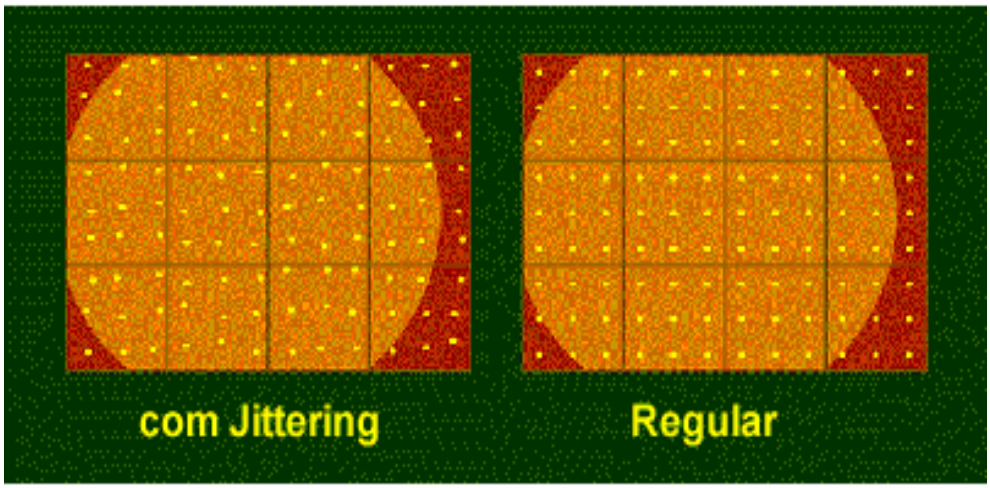
Segundo Kaushik (1997), as técnicas de integração de Monte Carlo não utilizam esta amostragem periódica e deste modo não podem produzir os padrões de *aliasing* periódico. Através delas introduz-se um ruído aleatório na saída. A magnitude deste ruído pode ser reduzida incrementando-se o número de amostras. Este ruído aleatório causa menos distúrbios visuais do que padrões regulares.

Demonstra-se, também, os resultados obtidos através do *jittering* visualmente através da Figura 7. Em ambos os casos, a resolução do monitor tem 4 *pixels* de largura

por 3 de altura. A grade superposta diminui o tamanho de um *pixel*. Ambas as figuras mostram a Superamostragem a três vezes a altura e três vezes a largura da resolução de monitor. Na figura da direita, as amostras são distribuídas regularmente. A figura da esquerda, as posições das amostras são distribuídas de forma aleatória. O montante aleatório é pequeno em relação ao tamanho do *pixel*.

A vantagem do *jittering* é que o olho humano tolera ruído com mais facilidade do que tolera os artefatos de *aliasing* e, como resultado disso, percebe-se uma maior qualidade na imagem interpretada.

Figura 7 - Diferença entre *jittering* e distribuição regular.



As técnicas de Monte Carlo tem a vantagem (sobre os algoritmos contínuos) de que é possível utilizar funções ponderadas multidimensionais complexas. Outra vantagem é que essas técnicas podem ser utilizadas em integrais complexas. Permite-se, assim, ao *antialiasing* e a outros efeitos que requerem integração utilizarem-se da mesma técnica.

Uma desvantagem (sobre os algoritmos contínuos) é que a convergência a baixos valores de ruído conforme aumenta o número de amostras é baixa, mas é mais rápida que a amostragem a intervalos uniformes para grandes números de dimensões.

5 *PLUG-IN* PARA *ADOBE PHOTOSHOP*

5.1 INTRODUÇÃO

Plug–ins não são exclusivos do *Adobe Photoshop*. Muitas aplicações para *Macintosh* e *Windows* suportam alguma forma de extensões de *plug–in*.

Uma das primeiras companhias a incorporar módulos de *plug–in* nos seus produtos é a *Silicon Beach*, em seu *Digital Darkroom* e produtos do SuperPaint.

No início, o *Adobe Photoshop* implementou módulos de *plug–in* semelhante a arquitetura utilizada pela *Silicon Beach*. Porém, a semelhança não durou muito tempo. Com a arquitetura de *plug–ins* evoluída, a interface detalhada para os módulos de *plug–in* do *Photoshop* tornou-se completamente diferente da utilizada pela *Silicon Beach*. As diferenças eram exigidas para suportar imagens coloridas e principalmente o apoio à memória virtual implementado no *Adobe Photoshop*.

5.2 MÓDULOS E SERVIDORES DE *PLUG-IN*

Módulos de *plug–in* são programas desenvolvidos pela *Adobe Systems* e vendedores especializados da *Adobe Systems* para estender uma aplicação. *Plug–ins* podem ser somados ou podem ser atualizados independentemente para personalizar os servidores de *plug-in* para as necessidades particulares.

Um servidor de *plug–in* é responsável por carregar módulos de *plug–in* na memória e os chamar.

Os programas da Adobe que funcionam como servidores de *plug-in* são: Adobe *After Effects*, *Adobe Premiere*, *Adobe Illustrator*, *Adobe PageMaker*, *Adobe PhotoDeluxe* e *Adobe Photoshop*. A maioria destes programas suporta alguns, mas não todos os *plug–in* do *Photoshop*.

A maioria dos servidores de *plug–in* é um programa, mas isto não é uma exigência. Um servidor de *plug–in* pode ser também um módulo de *plug–in*. Um bom exemplo disto é o "*Photoshop* Adapter", um *plug-in* que permite o Adobe Ilustrator 6.0 ser servidor de módulos de formato de arquivos e de filtro do *Photoshop*.

No *Photoshop*, os arquivos dos *plug-ins* estão localizados na pasta "*plug-ins*", dentro da pasta de instalação do *Photoshop*. Para adicionar novos *plug-ins* ao programa, basta adicionar o *plug-in* desejado na respectiva pasta.

5.3 TIPOS DE *PLUG-INS*

Os módulos de *plug-in* para *Adobe Photoshop* são arquivos externos que contêm código para estender o *Photoshop* sem modificar a aplicação básica.

O *Photoshop* suporta nove tipos de módulos de *plug–ins*, são eles: de Automatização, Paleta de Cores, Importação, Exportação, Extensão, Formato, Parser, Seleção e de Filtro. Tendo relevância para o presente trabalho o *plug-in* do tipo Filtro.

Este módulo de *plug-in* modifica uma área selecionada de uma imagem existente ou toda a imagem existente. Este módulo está presente no menu "Filter".

Alguns filtros fazem parte do aplicativo *Photoshop*. Outros são módulos externos que se localizam na pasta "Plug-ins" da instalação do *Photoshop*. Isso permite que se acrescente uma funcionalidade ao *Photoshop*, adquirindo filtros adicionais de coletâneas de outros fornecedores, por exemplo, *o PhotoTools (Extensis), Eye Candy (Alien Skin), Series (Andromeda), Paint Alchemy (Xaos Tools)* e *Kai's Power Tools (MetaCreations).*

Os arquivos de *plug–in* devem seguir as regras da Tabela 1 para serem identificados no Mac OS e no Windows.

Tabela 1 – Extensões dos *plug-in*s (Macintosh e Windows)

Tipo de Plug-in	Macintosh File Type	Windows File Extension
General (any type of plug–in)	8BPI	.8BP
Automation	8LIZ	.8LI
Color Picker	8BCM	.8BC
Import	8BAM	.8BA
Export	8BEM	.8BE
Extension	8BXM	.8BX
Filter	8BFM	.8BF
File Format	8BIF	.8BI
Parser	8BYM	.8BY
Selection	8BSM	.8BS

Através das extensões dos *plug-in*s, o *Photoshop* identifica que o arquivo é um *plug-in* e após a identificação é acionado o servidor de *plug-in* do *Photoshop* que então irá carregar o *plug-in* para a memória do computador.

6 DESENVOLVIMENTO

O protótipo foi desenvolvido no ambiente de programação Microsoft Visual C++ 6.0 da *Microsoft* com a incorporação das bibliotecas WPVM e *Adobe Photoshop 5.5 SDK*.

6.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Requere-se que o sistema possa carregar uma imagem *bitmap* no *Photoshop* e aplique as técnicas de *antialiasing* denominadas Superamostragem e Monte Carlo com processamento distribuído. Fragmentando a imagem e tratando cada fragmento em processos paralelos. E ao seu final exiba a imagem original alterada, ou seja, com o *aliasing* corrigido dentro do possível de cada técnica.

6.2 ESPECIFICAÇÃO

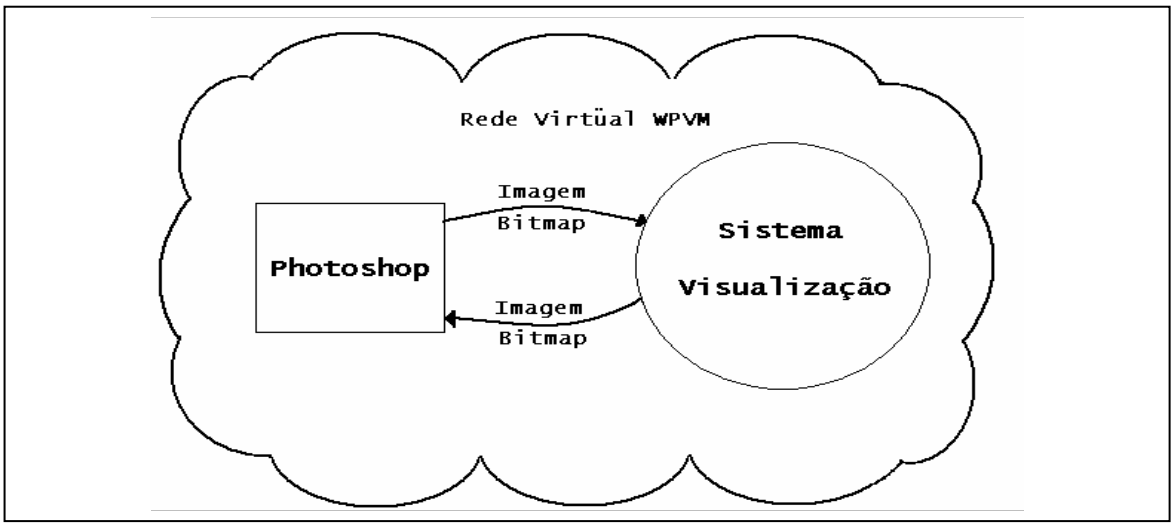
Apresenta-se na especificação diagramas que representam logicamente o funcionamento e estruturação dos três programas que compõem o sistema. São eles o *plug-in*, responsável pela interação do sistema com o usuário, o Master responsável pela fragmentação, desfragmentação e criação dos processos paralelos e o Slave responsável pelo tratamento do *aliasing* da imagem.

O desenvolvimento seguiu a metodologia de prototipação. A partir da documentação existente foram implementadas etapas do software e testadas, avaliando resultados para a próxima etapa.

6.2.1 DIAGRAMA DE CONTEXTO

O diagrama de contexto do protótipo é apresentado na Figura 8.

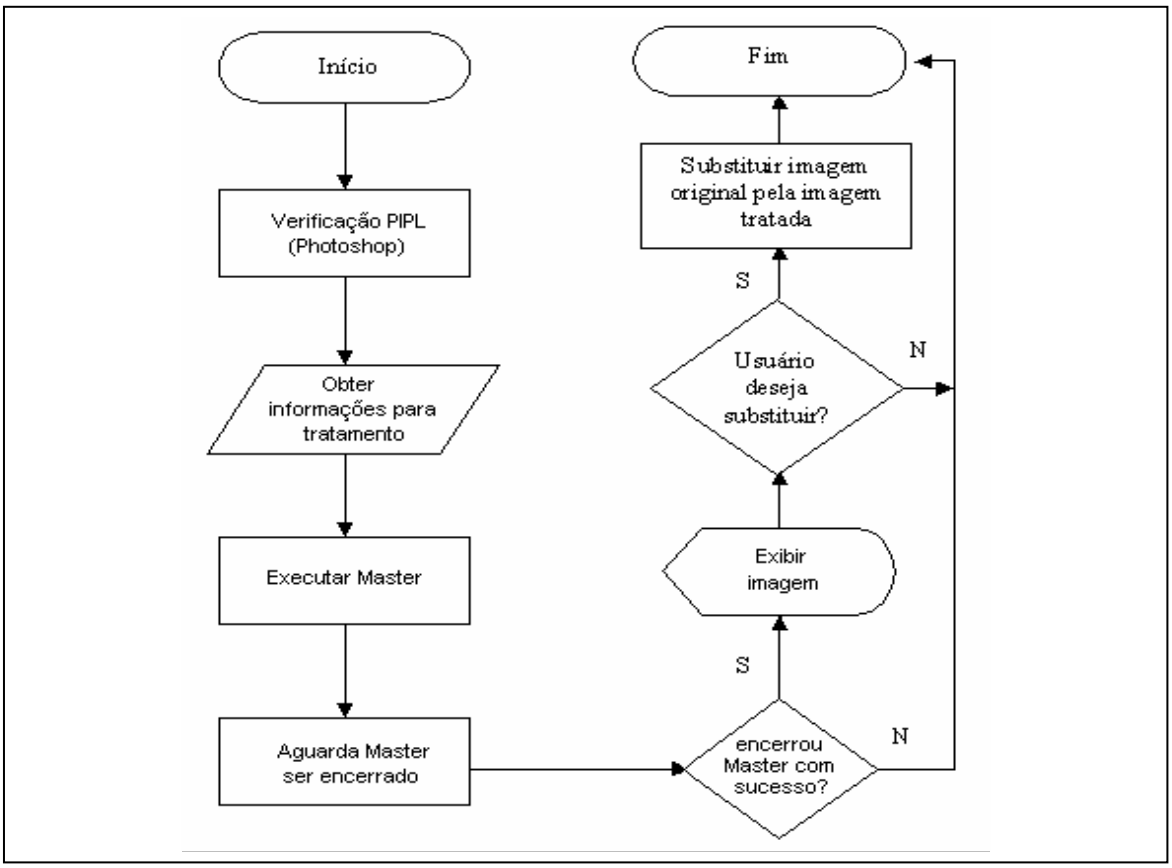
Figura 8 - Diagrama de contexto



6.2.2 FLUXOGRAMA DO PROTÓTIPO

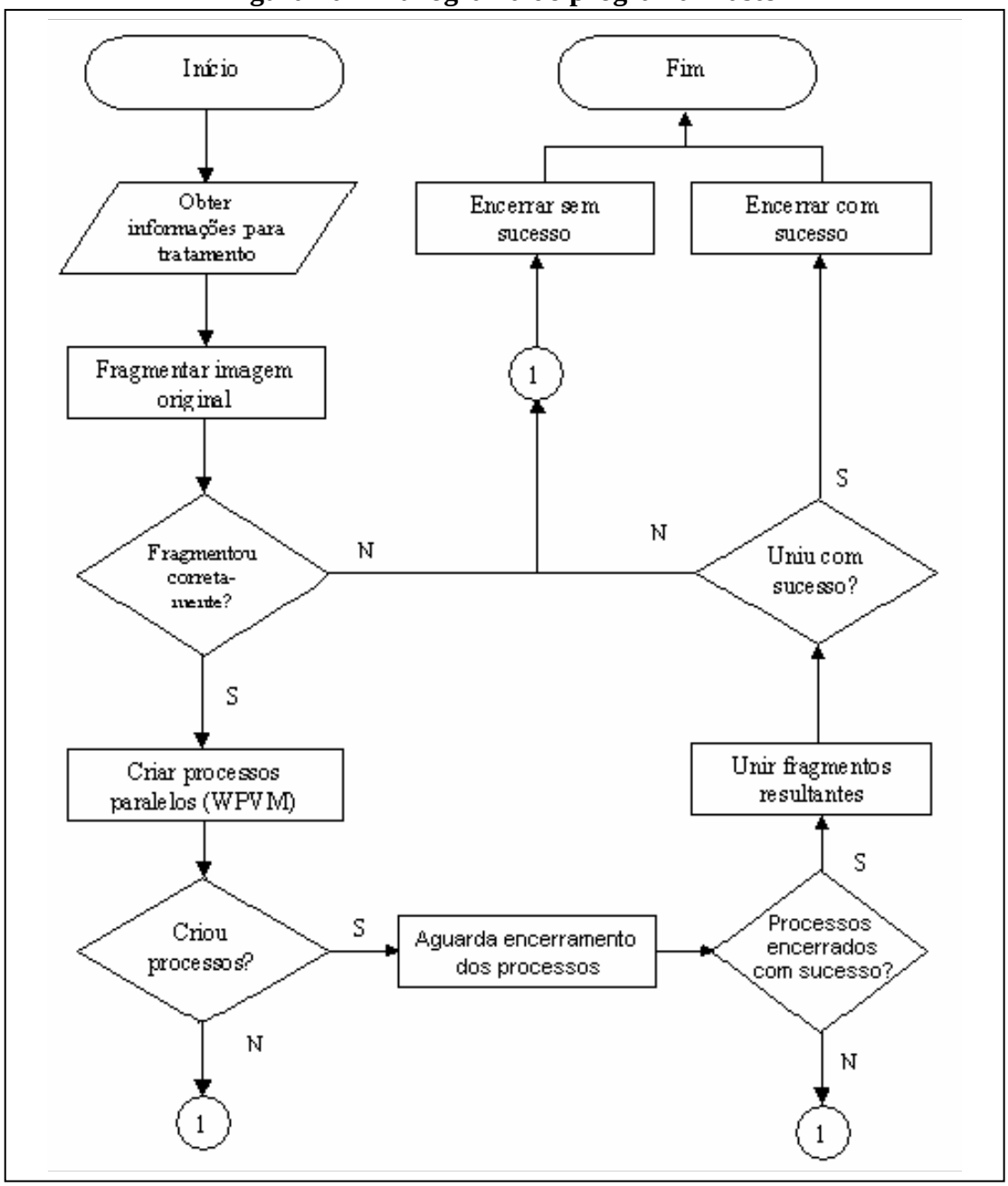
O funcionamento lógico do *plug-in* é apresentado na Figura 9.

Figura 9 – Fluxograma do *plug-in*



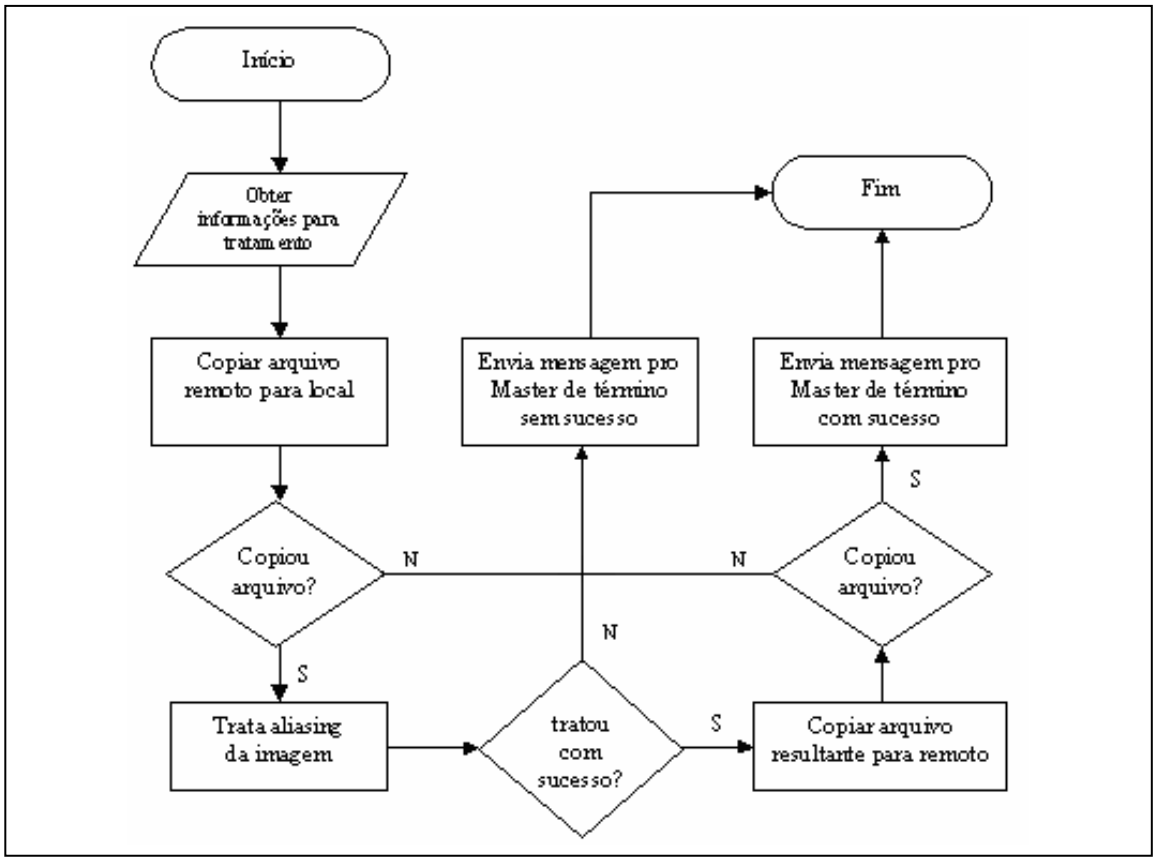
O funcionamento lógico do Master é apresentado na Figura 10.

Figura 10 – Fluxograma do programa Master



O funcionamento lógico do programa Slave é mostrado na Figura 11.

Figura 11 – Fluxograma do programa Slave



6.3 IMPLEMENTAÇÃO

6.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

Todo o trabalho de implementação foi desenvolvido utilizando o ambiente de programação Microsoft Visual C++ 6.0, em conjunto com o kit de desenvolvimento de software para *Adobe Photoshop* versão 5.5 para o sistema operacional Windows para a implementação do *plug*-in. A biblioteca para processamento distribuído *Windows Parallel Virtual Machine* (WPVM) versão 2.0 foi utilizada no programa “Master" e no “Slave” para efetuar a execução em paralelo de determinados processos conforme será visto adiante.

6.3.1.1 ADOBE PHOTOSHOP PIPL

Segundo a Adobe (2000), a PiPL *(Plug-in Property List*) é uma lista que contêm todas as informações necessárias para que o *Photoshop* possa identificar e carregar os

módulos de *plug-ins*, como também *flags* e outras propriedades que controlam a operação de cada *plug-in*.

As PiPLs são utilizadas por vários aplicativos da Adobe, entre eles o Adobe *After Effects*, o *Adobe Illustrator*, o *Adobe PageMaker*, o *Adobe Premiere*, o *Adobe PhotoDeluxe* e o *Adobe Photoshop*. Quando o *Photoshop* é inicializado, ele procura dentro do diretório dos *plug-ins* todos PiPLs válidos.

Cada propriedade contém um código do fornecedor (todos as propriedades dos *plug-ins* do *Photoshop* possuem o código do fornecedor igual a ’8BIM’), uma chave (ver Tabela 2), um identificador (reservado para uso futuro, o valor padrão é zero), uma variável que possui o tamanho do campo da propriedade e outro a propriedade em si (ver Quadro 3).

Quadro 3 – Propriedades do PiPL

```
typedef struct PIProperty
{
   OSType          vendorID;
   OSType          propertyKey;
   int32           propertyID;
   int32           propertyLength;
   char            propertyData[1];
} PIProperty;
```

A lista de propriedades do PiPL (ver Quadro 4) possui um número de versão, um contador que indica a quantidade de propriedades contidas na estrutura e uma variável que indica o comprimento da estrutura de dados das propriedades.

Quadro 4 – Lista de Propriedades do PiPL

```
typedef struct PIPropertyList
{
   int32           version; /* Versão Corrente = 0
   int32           count;   /* 0 = Nenhuma Propriedade
   PIProperty      properties[1];
} PIPropertyList;
```

A Tabela 2 apresenta uma relação de alguns tipos de propriedades aceitas pela estrutura do PiPL. O nome da propriedade é definida pela chave atribuída no PiPL.

Tabela 2 – Tipos de propriedades válidas na estrutura PiPL

Tipo	Nome	Chave	Descrição
OSType	PIKindProperty	'kind'	Especifica qual o tipo do *plug-in* criado. O protótipo proposto refere-se a um *plug-in* de filtro. '8BFM'=Filter module
int32	PIVersionProperty	'vers'	32 bits
int16	PIPriorityProperty	'prty'	Utilizado para definir a ordem de carga do *plug-in* no *Photoshop*.
Cstring	EnableInfo	'enbl'	Define quando o *plug-in* estará habilitado para ser utilizado, através do modo de imagem. Alguns tipos aceitos: GrayScaleMode, IndexedMode, RGBMode, CMYKMode entre outros.
Pstring	PICategoryProperty	'catg'	Especifica qual o nome do sub-menu que será colocado no menu **Filter**.
Pstring	PINameProperty	'name'	Nome do *plug-in* que será colocado no sub-menu especificado na propriedade PICategoryProperty.
Pstring	PIWin32x86CodeProperty	'wx86'	Especifica para qual estrutura irá ser colocado os valores da imagem do *Photoshop*

O Quadro 5 mostra parte do código implementado para configurar o *plug-i*n conforme as estruturas acima descritas.

Quadro 5 – Configuração (PIPL) do *plug-in*

```
// Arquivo Plugin3D.r
#define plugInName              "Plug-in WPVM - Antialiasing"
#define plugInCopyrightYear     "2001"
#define  plugInDescription      "Plug-in para o Photoshop com processamento distribuído."
#define vendorName              "TCC"
#define plugInAETEComment       "Plug-in para o Photoshop com processamento distribuído"
#define plugInSuiteID           'sdK1'
#define plugInClassID           plugInSuiteID
#define plugInEventID           plugInClassID

resource 'PiPL' (ResourceID, plugInName " PiPL", purgeable) {
        // Propriedades do Plug-in
        Kind { Filter },
        Name { plugInName "..." },
        Category { vendorName },
        Version { (latestFilterVersion << 16) | latestFilterSubVersion },
        #if defined(__PIWin__)
                CodeWin32X86 { "ENTRYPOINT" },
        #endif
        HasTerminology {
                plugInClassID,    // Class ID
                plugInEventID,   // Event ID
                ResourceID,              // AETE ID
                "98b5a608-46ce-11d3-bd6b-0060b0a13dc4"
        },
        SupportedModes { // Define quais os formatos de imagens permitidos ou  não pelo plug-in
                doesSupportBitmap, doesSupportGrayScale,
                doesSupportIndexedColor, doesSupportRGBColor,
                noCMYKColor, noHSLColor,
                noHSBColor, noMultichannel,
                noDuotone, noLABColor
        },
        EnableInfo {
                "in (PSHOP_ImageMode, BitmapMode, GrayScaleMode,"
                "IndexedColorMode, RGBMode)"
        }
...
```

Com o *Photoshop* iniciado, e tendo uma imagem *bitmap* carregada é acionado o *plug-in* através da opção de menu “Filter”, submenu “TCC”, “Plugin WPVM – Antialiasing”.

Na tela do *plug-in,* o usuário aciona o tratamento do *aliasing* através do botão “Processar”. O programa irá captar os dados informados pelo usuário e iniciará a execução do programa “Master", informando-o alguns parâmetros como, o caminho (*path)* do arquivo de imagem original a ser tratado, o caminho (*path*) da pasta compartilhada entre as máquinas que compõem a rede WPVM, a técnica de *antialiasing* a ser utilizada no tratamento, e caso a técnica selecionada seja a Monte Carlo, serão

passados também determinados pesos que formarão o ruído aleatório da técnica, conforme explicado na seção 4.4.2.

Após iniciado o programa “Master" através do comando “_spawnl”, é aguardado automaticamente o seu término devido que no comando foi passado o atributo “_P_WAIT” fazendo com que o processamento do código só continue após haver retorno do comando “_spawnl”. Caso o término ocorra com sucesso, ou seja, o retorno do programa “Master" seja *TRUE*, o processamento ocorreu com sucesso e é então exibida a imagem tratada. Porém caso retorne *FALSE*, significa que ocorreu algum problema durante todo o processamento que não foi possível tratar o *aliasing* da imagem, sendo assim não é possível exibi-la.

O Quadro 6 mostra o código parcial responsável pela execução do programa “Master”.

Quadro 6 – Código fonte parcial responsável pela execução do “Master”

```
...
// Monta em uma string os parâmetros a serem passados ao Master
for ( pesoX = 0; pesoX < 3; pesoX++ )
        for ( pesoY = 0; pesoY < 3; pesoY++ )
                // Matriz de pesos para a tecnica monte carlo
                strcat( Aux, _itoa( m_Pesos[ pesoX ][ pesoY ], Aux2, 10 ) );
strcat( Aux, "|" );
// Nome do Arquivo de imagem a ser tratado mais separador |
strcat( Aux, documentInfo->fileSpec->path );  strcat( Aux, "|" );
// Pasta compartilhada entre os computadores mais separador |
strcat( Aux, gPstComp);   strcat( Aux, "|" );
// Técnica de antialiasing a ser aplicada sobre a imagem
strcat( Aux, _itoa( gTecnica, Aux2, 10 ) );   strcat( Aux, "|" );
// Qtde de fragmentos que o master deve gerar a partir da imagem
strcat( Aux, _itoa( gQtdeFrag, Aux2, 10 ) );  args[0] = Aux;
// Obtem o diretorio do programa Photoshop, sendo que será o mesmo pro Master
GetModuleFileName( NULL, Aux2, 255 );
for ( int i = strlen( Aux2 ); i >= 0; i-- )
        if ( Aux2[ i ] == '\\' )
                Aux2[ i + 1 ] = 0;  break;

strcat( Aux2, "Master.exe" ); // Adiciona o nome do executavel do master
erro = _spawnl( _P_WAIT, Aux2, "|", args[0], NULL ); // Executa o Master e aguarda encerramento
if ( erro < 0 ){
        // Caso tenha ocorrido algum problema ao executar o Master
        MessageBox( hDlg, "O programa Master.exe não foi executado!", "ERRO", MB_OK );
        break;
}
...
```

O Quadro 7 mostra o código fonte parcial responsável pela exibição da imagem resultante.

Quadro 7 – Código fonte parcial responsável pela exibição da imagem resultante

```
...
HBITMAP hbit;
// Carrega a imagem na memoria
hbit = (HBITMAP) LoadImage(NULL, Aux, IMAGE_BITMAP, 0, 0,
          LR_LOADFROMFILE | LR_CREATEDIBSECTION | LR_DEFAULTSIZE);
// Obtem o controle onde sera exibido a imagem
HWND hWndImg = ::GetDlgItem(hDlg, IDC_IMAGEM_RESULT);
// Caso tenho conseguido carregar a imagem na memoria, envia mensagem
// STM_SETIMAGE ao controle para que seja exibida a imagem.
if (hWndImg != NULL)
 ::SendMessage(hWndImg, STM_SETIMAGE, IMAGE_BITMAP, (LPARAM) (HANDLE) hbit);
…
```

Então caso o usuário deseje aceitar o resultado obtido, a imagem original será substituída pela imagem resultante.

6.3.1.2 MASTER

Inicialmente, o programa recebe e armazena as informações passadas por parâmetro pelo *plug-in*.

O próximo passo é a divisão da imagem, onde a altura de cada fragmento será igual a altura da imagem original e a largura será igual a largura da imagem dividida pela quantidade de fragmentos a serem gerados.

Para que o tratamento da imagem ocorra corretamente, adiciona-se faixas extras de *pixels* aos fragmentos. Por exemplo, ao iniciar o tratamento no primeiro fragmento gerado, ou seja, a região mais à esquerda da imagem original, percorre-se todos os *pixels* da imagem analisando cada *pixel* e seus vizinhos.

Porém os *pixels* mais a direita da imagem não possuem vizinhos à sua direita. Mas na imagem original estes *pixels* possuem.

Se o tratamento da imagem fosse efetuado ignorando-se estes *pixels* que faltam, o resultado não seria o mesmo que se o tratamento ocorresse sem fragmentar a imagem.

Para resolver este problema, adiciona-se no primeiro fragmento uma faixa *pixels* à direita. Sendo esta faixa idêntica a faixa de *pixels* inicial do segundo fragmento gerado.

Desta forma os *pixels* vizinhos serão identificados com seus valores corretos, conseqüentemente o tratamento da imagem também ocorrerá corretamente.

Os demais fragmentos também recebem faixas de *pixels* adicionais. No último fragmento, correspondente a parte mais a direita da imagem original, adiciona-se uma faixa de *pixels* à esquerda. Os demais fragmentos, que formam a região intermediária da imagem, recebem uma faixa à esquerda e outra a direita.

No processo de divisão da imagem, é armazenado em um vetor informações sobre cada fragmento gerado, tais como, local onde será gravado o arquivo do fragmento, status de processamento, dimensões, entre outras, sendo que estas informações serão úteis na criação dos processos paralelos e na junção dos fragmentos após feito o tratamento.

No Quadro 8 pode ser visto o algoritmo básico de divisão da imagem.

Quadro 8 – Código fonte básico responsável pela divisão da imagem

```
...
for ( int i = 0; i < m_pMasterDlg->m_QtdeFragmentos; i++ ) {
        if ( i == 0 ) // Define a largura que o fragmento em questao tera,
                Largura = ( (DWORD)LarguraImagem / (DWORD)m_QtdeFragmentos ) + 1 ;
        else if ( i == m_QtdeFragmentos-1 ) // Pega os pixels restantes da imagem
                Largura = ( LarguraImagem - Coord_X_JaCopiado ) + 1;
        else Largura = ( LarguraImagem / m_QtdeFragmentos ) + 2;

        // Identifica a faixa de pixel da img original que sera o inicio  da img frag., ou seja, identifica
        // em que faixa de pixels da img original sera comecado a copiar os pixels para o fragmento.
        if ( Coord_X_JaCopiado > 0 ) { // se nao for o primeiro fragmento
                // Caso seja o primeiro ou o ultimo fragmento
                if ( ( i == ( m_QtdeFragmentos - 1 ) ) || ( i == 0 ) )
                        Coord_X_Ini = Coord_X_JaCopiado - 1;
                else  Coord_X_Ini = Coord_X_JaCopiado - 2;
        }
        Altura = AlturaImagem;
        // Copia area definida para o frag. da img original p/ depois ser gravado em um arq bitmap
        BitBlt(HDCFrag, 0,0, Largura, Altura,memDC, Coord_X_Ini,Coord_Y_Ini,SRCCOPY );
        HBITMAP hMeuBmp = ::CreateCompatibleBitmap( memDC, Largura, Altura );
        ::SelectObject( HDCFrag, hMeuBmp );
        cMeuBmp.Attach( hMeuBmp );
        BitBlt(HDCFrag, 0, 0, Largura,Altura,memDC,Coord_X_Ini,Coord_Y_Ini, SRCCOPY );
        ::SelectPalette( HDCFrag, Palette, FALSE );

        // Cria um DIB do Bitmap resultante para depois gravar em arquivo
        HANDLE hDIB = DDBToDIB( (CBitmap&)cMeuBmp, BI_RGB, &Palette );
        if ( hDIB == NULL )
            m_cLstEventos.InsertString( 0, _T("ERRO: Problema na conversao da img para DIB." ) );
...
        // Grava o fragmento da imagem em um arquivo temporario.
        if ( !WriteDIB( NomeFragOriginal, hDIB ) )
                m_cLstEventos.InsertString( 0, _T("ERRO: Problema ao gravar arquivo." ) );
...
```

Cada fragmento da imagem original é armazenado em uma pasta definida pelo usuário através do campo “Pasta compartilhada” disponível na tela do *plug-in*. O nome de cada fragmento é definido conforme efetua-se a divisão da imagem, tendo o primeiro fragmento o nome de “0.bmp”, o segundo “1.bmp” e assim por diante.

Concluída com sucesso a divisão da imagem, inicia-se a criação dos processos paralelos que tratarão o *aliasing* em cada fragmento.

Para cada fragmento criado, será solicitado ao programa WPVM Daemon a execução do programa “Slave” em uma determinada máquina pertencente a rede WPVM. Sendo que é o próprio WPVM Daemon que irá escolher em qual máquina será disparado o processo. Juntamente com a solicitação de execução do “Slave” serão passados alguns parâmetros necessários à correta execução do programa “Slave”. São eles, o caminho (*path*) do arquivo do fragmento a ser tratado, o caminho (*path*) do arquivo que deverá ser criado caso o tratamento ocorra corretamente, a técnica de *antialiasing* que deverá ser utilizada e por fim caso a técnica selecionada seja a Monte Carlo, será passado pesos definidos na tela do *plug*-in necessários ao processamento da técnica.

Iniciado todos os processos paralelos, o programa passa a aguardar o recebimento de mensagens enviadas pelos processos através do WPVM Daemon informando o término do processamento.

Sendo que o processo de aguardo pela conclusão de todos os processos paralelos ocorre através de um *looping*, no qual só é quebrado quando todos os processos criados informarem através de mensagens ao programa a conclusão dos mesmos. No Quadro 9 pode ser visto a implementação parcial desta etapa.

Quadro 9 – Código fonte parcial da verificação de término dos processos paralelos.

```
// Inicia um looping que ira aguardar a resposta de todos os processos  quanto ao tempo de
// processamento de cada um e se processou o fragmento e criou o arquivo resultante com sucesso.
while ( TotConcluido < m_pMasterDlg->m_QtdeFragmentos )
{
        if ( m_pMasterDlg->m_pArrayFrag[ i ].Status != NAO_INICIADO && m_pMasterDlg-
        >m_pArrayFrag[ i ].Status != CONCLUIDO ) {
                ...
                // Verifica se recebeu alguma mensagem do slave com a tag TAG_CONCLUIDO do
                // processo em questao que informa que o processamento foi concluido com sucesso.
                // Sendo que slave envia msg apos ter conseguido criar o arquivo de frag. resultante.
                if ( m_pMasterDlg->RecebeMsgSlave( m_pMasterDlg->m_pArrayFrag[ i ].TID,
                Buffer, TAG_CONCLUIDO ) )     {
                        // Se recebeu atualiza campos da tela
                        strcpy( Aux, "STATUS: Escravo ( TID: " );
                        strcat( Aux, _itoa( m_pMasterDlg->m_pArrayFrag[ i ].TID, Buffer, 10 ) );
                        strcat( Aux, " ) concluido com sucesso!" );
                        m_pMasterDlg->m_cLstEventos.InsertString( 0, Aux );

                        // Atualiza o campo de Status de Processamento no array de informacoes,
                        // de acordo com o processo em questao.
                        m_pMasterDlg->m_pArrayFrag[ i ].Status = CONCLUIDO;
                        TotConcluido++;
                }
                ...
        }
}
...
```

Concluído com sucesso o processamento de todos os processos, inicia-se o processo de união dos fragmentos resultantes do tratamento em um único arquivo de imagem no formato *bitmap*.

Este processo baseia-se basicamente em eliminar dos fragmentos as faixas de *pixels* repetidas, conforme o processo de divisão da imagem, e posiciona-los devidamente em uma imagem para que tomem a forma da imagem original porém com o *aliasing* tratado.

O Quadro 10 mostra parcialmente o código fonte responsável por este processo.

Quadro 10 – Código fonte parcial responsável pela união dos fragmentos

```
…
DWORD           Coord_X_Ini = 0, Coord_X_Dest = 0;
for ( int i = 0; i < m_QtdeFragmentos; i++ ) {
        ...
        // Le o arquivo do fragmento para a memoria
        hBitmapFrag = (HBITMAP) LoadImage( NULL, Fragmento, IMAGE_BITMAP, 0, 0,
        LR_DEFAULTCOLOR | LR_LOADFROMFILE | LR_CREATEDIBSECTION |
        LR_DEFAULTSIZE);
        ...
        // Verifica lagura a ser copiado do fragmento, sendo que ao copiar
        // as faixas adicionadas no proc. divisao devem ser retiradas
        if ( ( i == 0 ) || ( i == ( m_QtdeFragmentos - 1 ) ) )
                LarguraFrag = (BitmapInfoFrag.bmWidth - 1/*faixa pixels*/ );
        else LarguraFrag = BitmapInfoFrag.bmWidth - 2/*faixa pixels*/;
        // Define a partir de que faixa de pixel sera copiado do fragmento para a imagem completa.
        if ( i == 0 ) Coord_X_Ini = 0;
        else Coord_X_Ini = 1;
        …
        // Copia literalmente o conteudo do fragmento para a imagem completa
        // Ira copiar para uma determinada posicao da imagem completa.
        // As posicoes sao determinadas conforme a divisao da imagem
        BitBlt( memDC, Coord_X_Dest, 0, LarguraFrag, BitmapInfoFrag.bmHeight, memDCFrag,
        Coord_X_Ini, 0, SRCCOPY );

        // Atualiza o controle da posicao na imagem completa de onde
        // comecara a ser copiado o conteudo da imagem fragmento.
        Coord_X_Dest += LarguraFrag;
}
...
```

Então concluída com sucesso a união dos fragmentos o programa encerra a sua execução com sucesso. Sendo que é através do seu encerramento que o *plug-in* irá identificar se o processamento ocorreu com sucesso ou não.

6.3.1.3 SLAVE

Inicialmente, o programa recebe e armazena as informações passadas por parâmetro pelo programa “Master".

Recebida as informações o programa irá copiar o arquivo de imagem que deve tratar para uma pasta local, ou seja, para a máquina que o irá tratar.

A pasta local é definida como a própria pasta do programa WPVM Daemon responsável por armazenar os processos permitidos a serem executados, geralmente especificada de “..\WPVM\bins”.

Conforme apresentado na seção 4.4, para efetuar o tratamento do *aliasing* é necessário que a imagem original seja amostrada a uma resolução maior que a desejada na saída.

O Quadro 11 mostra o código fonte parcial responsável pelo processo de amostrar a imagem original a uma escala duas vezes maior que a desejada para a saída.

Quadro 11 – Código fonte parcial responsável pela amostragem da imagem

```
...
// Determina-se a largura da imagem virtual ( superamostrada )
DWORD           LarguraImgVirtual = BmpWidth * 2,
                AlturaImgVirtual = BmpHeight * 2;

// Determina-se o fator de escala da imagem
double fx = dlBmpWidth / LarguraImgVirtual;
double fy = dlBmpHeight / AlturaImgVirtual;

double X_Amostra = 0, Y_Amostra = 0;
for ( DWORD Linha = 0; Linha < LarguraImgVirtual; Linha++ ) {
   // Soma a si mesmo o fator de escala em relacao ao eixo X.
   X_Amostra += fx;
   // Caso X_Amostra seja maior ou igual a largura da imagem, eh atribuido a ela a
   // largura da imagem menos um, devido que os pixels validos da imagem vao de zero
   // ate a largura menos um.Sendo assim o X_Amostra nao deve ultrapassar este limite.
   if ( X_Amostra >= BmpWidth ) X_Amostra = BmpWidth - 1;

   Y_Amostra = 0;
   // Looping dos pixels vai do zero ate a altura da img
   for ( DWORD Coluna = 0; Coluna < AlturaImgVirtual; Coluna++ )  {
         // Soma a si mesma o fator de escala em relacao ao eixo Y.
         Y_Amostra += fy;
         // Caso Y_Amostra seja maior ou igual a altura da imagem,
         // eh atribuido a ela a altura da imagem menos um, devido que
         // os pixels validos da imagem vao de 0 ate a altura menos um.
         // Sendo assim o Y_Amostra nao deve ultrapassar este limite.
         if ( Y_Amostra >= BmpHeight ) Y_Amostra = BmpHeight - 1;

         // Determina-se a cor do pixel na imagem original nas
         // coordenadas (X_Amostra, Y_Amostra )
         COLORREF cor = GetPixel( memDC, X_Amostra, Y_Amostra );

         // Eh atribuido na img virtual nas coord x,y a cor obtida na imagem original.
         SetPixel( hDCVirtual, Linha, Coluna, cor );
   }
}
...
```

Inicia-se então o processamento da imagem de acordo com a técnica de *antialiasing* especificada.

O Quadro 12 identifica o código fonte parcial responsável por aplicar o tratamento sobre a imagem de acordo com a técnica Superamostragem.

Quadro 12 – Código fonte parcial da técnica Superamostragem

```
HDC CSlaveDlg::TrataTecnicaSuperAmost( int xInicial, int yInicial, DWORD BmpHeight, DWORD
BmpWidth, HDC memDC ) {
	// Determina-se o fator de escala da imagem para com a imagem virtual
	// que no caso deste prototipo eh definido como padrao 2x.
	fx = (dlBmpWidth * 2) / dlBmpWidth;
	fy = (dlBmpHeight * 2) / dlBmpHeight;
	X_Amostra = 0;
	for ( x = 0; x < BmpWidth; x++ ) {
		// Multiplica-se o fator de escala pelo coord X, sendo que eh  atraves do X_Amostra
		// que sera buscado na img virtual o vlr da nova cor para o pixel atual
		X_Amostra = ( fx * x );

		double Y_Amostra = 0;
		// Looping dos pixels na vertical (de 0 ate altura da imagem)
		for ( DWORD y = 0; y < BmpHeight; y++ ){
			// Multiplica-se o fator de escala pelo coord X, sendo  que eh atraves do
			// Y_Amostra que sera pego na img virtual o vlr da nova cor p/ o pixel atual
			Y_Amostra = ( fy * y );

			// Chama funcao Converte que calcula a media de cor
			COLORREF cor = ConverteSuperAmost( hDCVirtual, (int)X_Amostra,
			(int)Y_Amostra );

			// Atribui ao pixel atual da img result a sua nova cor.
			SetPixel( memDC, x, y, cor );
		}
	}
	return memDC; // Retorna o HDC que passou pelo tratamento de aliasing
}

COLORREF CSlaveDlg::ConverteSuperAmost( HDC DC, int x, int y ) {
	int l, c, tr, tg, tb;
	tr = tg = tb = 0;
	// Leitura dos pixels
	for ( l = (x-1); l <= (x+1); l++ ) {
		for ( c = (y-1); c <= (y+1); c++ ) {
			// Obtem a intensidade de cor do pixel
			tr = tr + GetRValue( GetPixel(DC,l,c) );
			tg = tg + GetGValue( GetPixel(DC,l,c) );
			tb = tb + GetBValue( GetPixel(DC,l,c) );
		}
	}
	// Gera media simples
	tr = tr/9;
	tg = tg/9;
	tb = tb/9;
	return RGB(tr,tg,tb);
}
```

Os algoritmos, conforme apresentado no capítulo 4, consistem basicamente em percorrer todos os *pixels* da imagem, analisar e atribuir novos valores de cores à cena de acordo com uma média simples da intensidade do mesmo, no caso da técnica de Superamostragem, e no caso da técnica de Monte Carlo, atribuir ruídos aleatórios para formar a média simples de intensidade. Conforme pode ser visto no Quadro 13.

A média simples da técnica Superamostragem é alcançada pela soma da intensidade de cor dos *pixels* vizinhos e do próprio *pixel* analisado dividido por nove. Considerando que cada *pixel* possui oito vizinhos. No caso dos *pixels* que não possuírem os oito vizinhos, é atribuído que os vizinhos não encontrados possuam a mesma intensidade de cor do *pixel* analisado.

Quadro 13 – Código fonte parcial da técnica Monte Carlo

```
CSlaveDlg::TecnicaMonteCarlo(DWORD LarguraImagem,DWORD AlturaImagem,HDC memDC) {
...
          for ( Linha = 0; Linha < BmpWidth; Linha++ ) {
              // Multiplica-se o fator de escala pela coord X, sendo que eh atraves do X_Amostra
              // que sera buscado na imagem virtual o valor da nova cor para o pixel atual
              X_Amostra = ( fx * Linha );
              Y_Amostra = 0;
              // Looping dos pixels na vertical ( vai do 0 (zero) ate a altura da imagem - 1 )
              for ( DWORD Coluna = 0; Coluna < BmpHeight; Coluna++ ) {
                    // Multip. o fator de escala pelo coord X, sendo que é atraves d Y_Amostra
                    // que sera buscado na imagem virtual o valor da nova cor para o pixel atual
                    Y_Amostra = ( fy * Coluna );
                    // Chama funcao Converte para calcular a media de cor entre o pixel
                    COLORREF cor = ConverteMonteCarlo( hDCVirtual, (int)X_Amostra,
                                                          (int)Y_Amostra );
              }
              // Atribui ao pixel atual da imagem resultante a sua nova cor.
              SetPixel( memDC, Linha, Coluna, cor );
          }
          return memDC;  // Retorna o HDC que passou pelo tratamento de aliasing
}

COLORREF CSlaveDlg::ConverteMonteCarlo(HDC DC,DWORD X_Amostra,DWORD Y_Amostra){
          int tr,  tg, tb,ptot, PesoAtual;  tr = tg = tb = ptot = 0;
          // Leitura dos pixels
          for ( DWORD IndLinha = ( X_Amostra - 1 ); IndLinha <= ( X_Amostra + 1 ); IndLinha++ ) {
              for ( DWORD IndCol = (Y_Amostra - 1); IndCol <= ( Y_Amostra + 1 ); IndCol++ ) {
                    PesoAtual = m_Pesos[NumeroRandomico(0, 2 ) ][ NumeroRandomico( 0, 2 ) ];
                    ptot  += PesoAtual;
                    // Obtem a intensidade de cor do pixel
                    tr += GetRValue( GetPixel( DC, IndLinha,IndCol ) * PesoAtual );
                    tg += GetGValue( GetPixel( DC, IndLinha,IndCol ) * PesoAtual );
                    tb += GetBValue( GetPixel( DC, IndLinha,IndCol ) * PesoAtual );
              }
          }
          tr = arredonda(tr / ptot);
          tg = arredonda(tg / ptot);
          tb = arredonda(tb / ptot);
          return RGB( tr, tg, tb );
}
```

Concluído o tratamento da imagem, o programa efetua a cópia da imagem resultante para a pasta compartilhada informada pelo usuário na tela do *plug-in* e caso o processo da cópia ocorra com sucesso é enviado uma mensagem ao “Master” avisando

da conclusão com sucesso do programa através do comando “pvm_send”. Caso contrário é informado da mesma forma da conclusão sem sucesso.

6.3.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

O sistema é executado através do programa gráfico *Adobe Photoshop 6.0*.

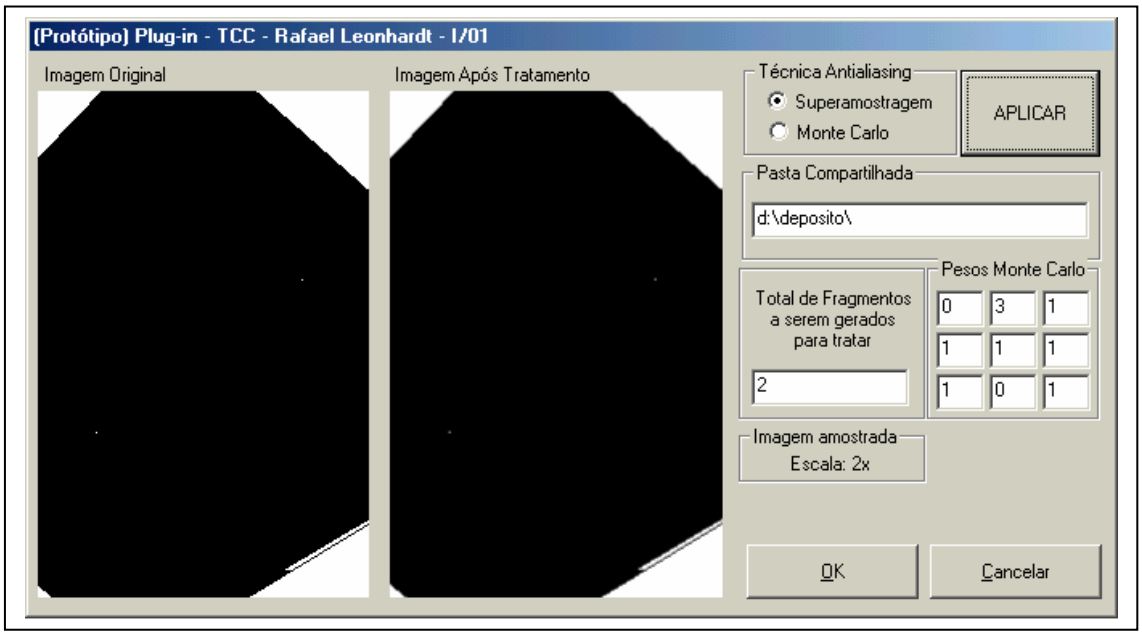
O usuário inicia o *Photoshop* e carrega imagem *bitmap*. Podendo fazer isto através da opção de menu “File” “Open”. Seleciona-se então a opção de menu “Filter”, submenu “TCC”, “Plugin WPVM – Antialiasing” para dar início ao *plug-in*.

As subseções a seguir apresentam a interface e operacionalidade do sistema.

6.3.2.1 *PLUG-IN*

A janela do *plug-in* possui campos e botões que permitem a interação do usuário com o *plug-in*. Na Figura 12, podemos ver a janela do *plug-in*.

Figura 12 – Janela do *Plug-in*



A janela demonstra o objeto inicial (lado esquerdo da janela) e a imagem resultante após o tratamento de *aliasing* (centro da janela). Têm-se no *plug-in* botões de seleção onde é escolhido pelo usuário a técnica de *antialiasing* (Superamostragem ou Monte Carlo) desejada para aplicar sobre a imagem.

No campo “Pasta compartilhada” é especificado a pasta compartilhada entre as máquinas que compõem a rede WPVM. Nesta pasta é depositado os arquivos criados na divisão da imagem original, bem como o arquivo resultante da junção de todos os fragmentos após terem sido tratado o *aliasing*.

No campo “Total de Fragmentos a serem gerados para tratar”, é especificado em quantas partes será divida a imagem, sendo que para cada fragmento será criado pelo menos um processo paralelo que tratará o *aliasing*.

A direita é especificado a matriz de pesos utilizada na técnica de Monte Carlo.

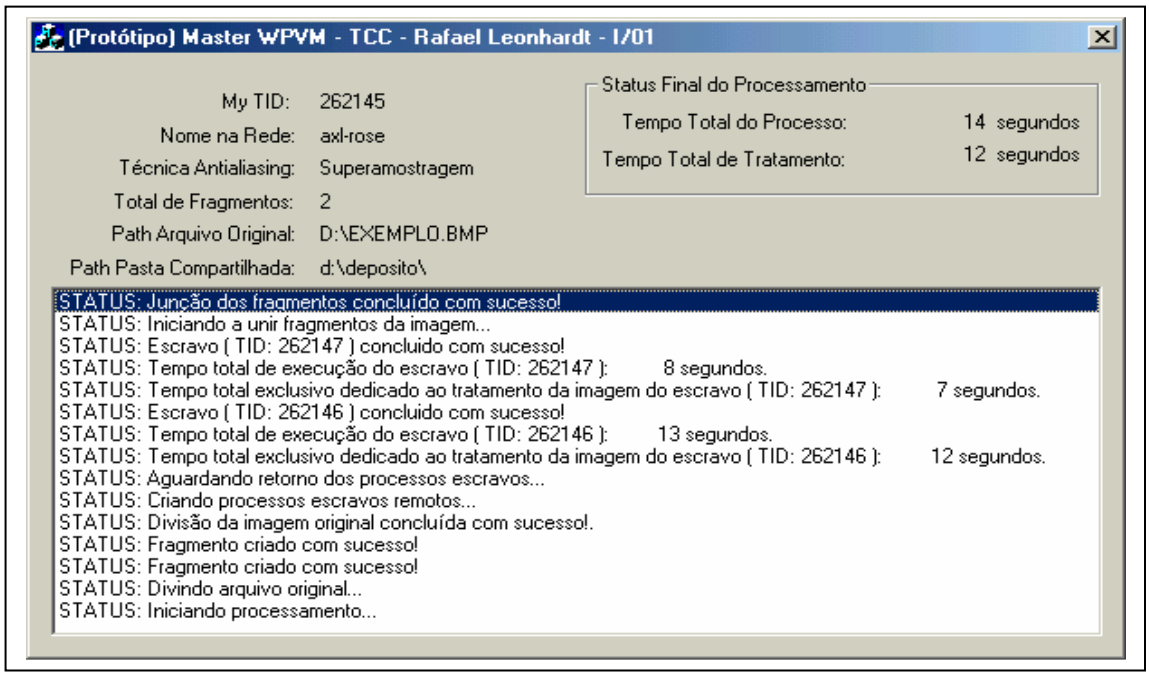
O campo “Imagem amostrada”, especifica a escala na qual a imagem original foi amostrada.

O botão “Aplicar” dará início ao processo de tratamento da imagem. O botão “OK” irá substituir a imagem original pela imagem gerada após tratado o *aliasing* e fechará a janela do *plug-in*. O botão “Cancelar” irá ignorar a imagem resultante e fechará a janela do *plug-in*.

6.3.2.2 MASTER E SLAVE

A janela do programa Master exibe informações sobre o andamento do processamento da imagem. Na Figura 13, podemos ver a janela do programa Master.

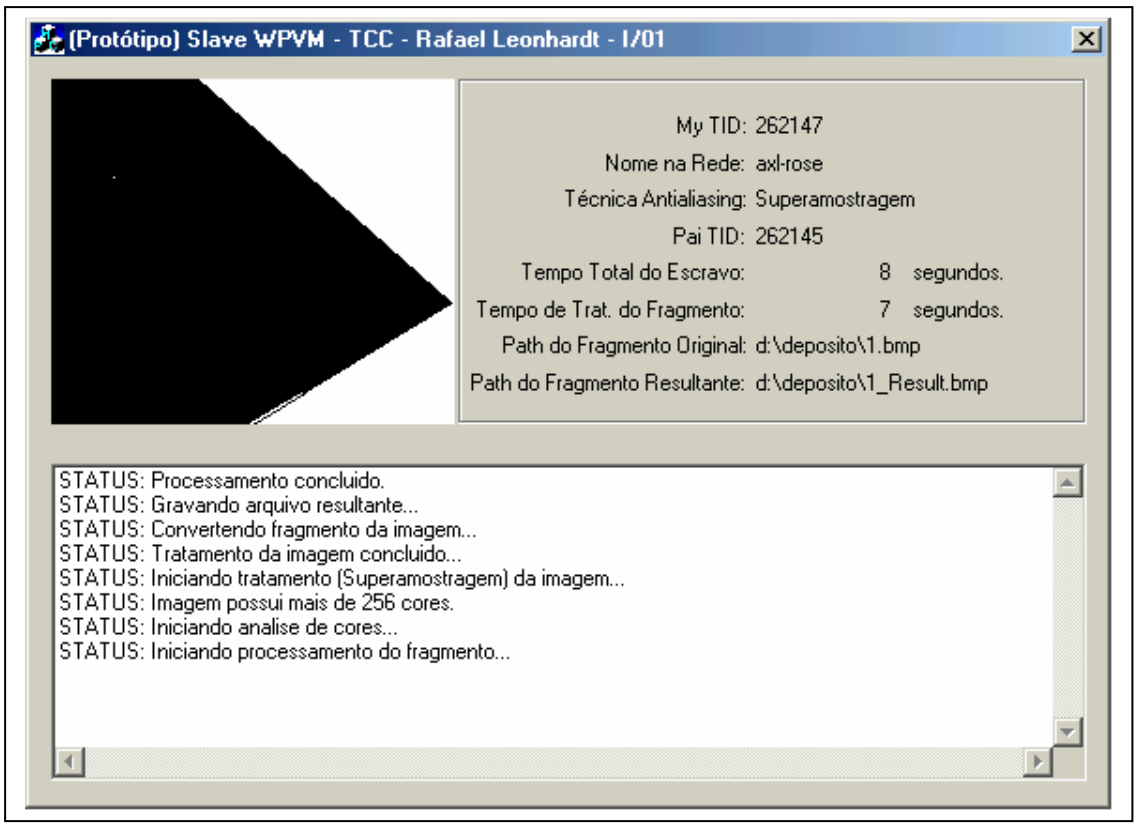
Figura 13 – Janela do programa Master



Nesta janela é mostrado o TID do processo em execução, identificação da máquina na rede, a técnica utilizada para tratar o *aliasing*, a quantidade total de fragmentos que será gerado a partir da imagem original, caminho do arquivo da imagem original que será tratada, caminho da pasta que está compartilhada dentre as máquinas que compõem a rede WPVM, tempo de vida em segundos do programa *Master*, tempo consumido exclusivamente para o tratamento do *aliasing* da imagem. Na parte inferior da tela, é exibida uma lista de status onde é informado o andamento da execução do programa.

A janela do programa Slave apresenta informações quanto ao andamento do processamento do fragmento da imagem. Na Figura 14, podemos ver a janela do programa Slave.

Figura 14 – Janela do programa Slave



Nesta janela é mostrado o TID do processo em execução, identificação da máquina na rede, a técnica utilizada para tratar o *aliasing*, o TID do processo que o criou, tempo de vida do programa em segundos, tempo que o processo consumiu para tratar o fragmento, nome e caminho do arquivo do fragmento da imagem original, nome e caminho do fragmento resultante que é gerado ao concluir o tratamento, lista de status,

informando passo a passo a execução do programa e por fim à esquerda é exibida parcialmente a imagem que está sendo tratada.

6.4 RESULTADOS E DISCUSSÃO

Objetivando-se analisar e validar os algoritmos descritos anteriormente, identificou-se como fator propenso à comparação, a ordem de complexidade baseada no tempo necessário para a execução de cada método.

Visando-se acentuar as diferenças entre o processamento distribuído e o processamento convencional, os testes comparativos foram realizados utilizando-se:

a) dois computadores Pentium III, 256 MB RAM, 700 MHz;

b) ambos com placa de rede Ethernet 10/100 Mbps.

Os algoritmos foram executados sem a execução de quaisquer aplicações em segundo plano, exceto o programa WPVM Daemon que se faz necessário em ambas as máquinas, e em uma das máquinas o programa gráfico *Adobe Photoshop 6.0*. A medição dos tempos foi realizada através do comando *GetCurrentTime*(), utilizando assim o relógio interno dos computadores.

Para efetuarmos a comparação de performance entre o processamento paralelo e o processamento convencional, foi utilizado o programa *Slave* desenvolvido neste trabalho. Porém executando-o independentemente dos outros programas (*Plug-in* e *Master*) e em um único computador. Desta forma a comparação entre os processamentos é mais precisa, devido que em ambos os processamentos a implementação dos algoritmos de tratamento da imagem são idênticos.

Foram analisadas três imagens no formato *bitmap* e com as seguintes dimensões: 2000x2000, 3000x3000 e 4000x4000 *pixels*.

Na Tabela 3 é possível verificar os resultados obtidos ao executar o programa *Slave* em um único computador.

Tabela 3 – Resultados do processamento convencional

Imagem	Computador	Técnica	Tempo Total
4000x4000 pixels	1 Pentium III	Superamostragem	**1843 segundos**
3000x3000 pixels	1 Pentium III	Superamostragem	**871 segundos**
2000x2000 pixels	1 Pentium III	Superamostragem	**251 segundos**
4000x4000 pixels	1 Pentium III	Monte Carlo	**2792 segundos**
3000x3000 pixels	1 Pentium III	Monte Carlo	**1378 segundos**
2000x2000 pixels	1 Pentium III	Monte Carlo	**799 segundos**

Antes de analisarmos a diferença de performance entre os modelos de processamento, é necessário esclarecer alguns pontos referente aos testes com processamento paralelo. São eles:

a) cada imagem foi fragmentada em duas partes, distribuídas entre as duas máquinas Pentium III. Neste item, também foi comparado o tempo de processamento quando cada uma das máquinas é a responsável por acionar os processos paralelos, ou seja, está com o programa *Photoshop* em execução;
b) utilizou-se os mesmos arquivos de imagens utilizados nos testes com processamento convencional;
c) nas Tabelas 4 e 5, os itens da coluna “Computadores” que possuírem na descrição o complemento “(Photoshop)”, significa que esta máquina possuía o *Photoshop* em execução, e foi responsável pela criação dos processos paralelos.

Na Tabela 4 verifica-se os resultados obtidos efetuando-se o tratamento das imagens com a técnica Superamostragem através de processamento paralelo.

Tabela 4 – Resultados do processamento paralelo (Superamostragem)

Imagem	Computador	Tempo *Slave*	Tempo Total
4000x4000 pixels	Pentium III Pentium III (Photoshop)	731 763	**920 segundos**
3000x3000 pixels	Pentium III Pentium III (Photoshop)	260 291	**367 segundos**
2000x2000 pixels	Pentium III Pentium III (Photoshop)	82 93	**123 segundos**

Na Tabela 5 verifica-se os resultados obtidos efetuando-se o tratamento das imagens com a técnica Monte Carlo através de processamento paralelo.

Tabela 5 – Resultados do processamento paralelo (Monte Carlo)

Imagem	Computador	Tempo *Slave*	Tempo Total
4000x4000 pixels	Pentium III Pentium III (Photos.)	1198 1253	**1433 segundos**
3000x3000 pixels	Pentium III Pentium III (Photos.)	550 587	**693 segundos**
2000x2000 pixels	Pentium III Pentium III (Photos.)	249 257	**301 segundos**

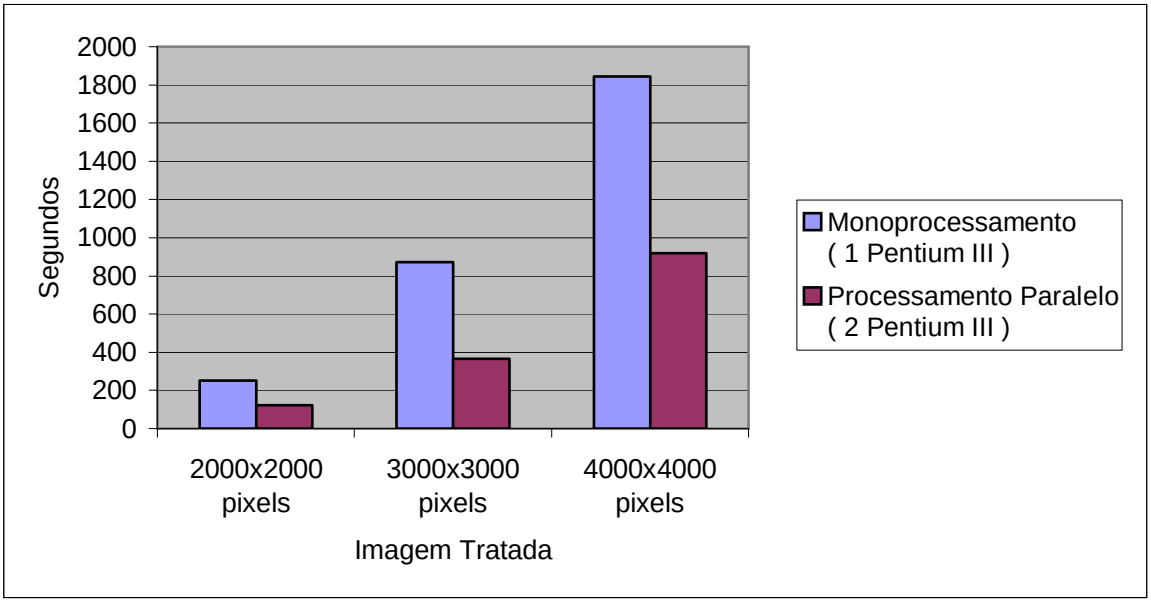
De acordo com as tabelas acima demonstradas, verifica-se uma redução drástica no tratamento do *aliasing*, quando este é tratado através de processamento paralelo. Como por exemplo, podemos observar que o processamento em paralelo da imagem de 3000x3000 *pixels*, através da técnica Superamostragem, obteve redução superior a 50% no tempo total de tratamento da imagem (ver Tabela 3 e 4). Conclui-se então, que para imagens destas dimensões, conforme for aumentando a quantidade de máquinas para processar, menor será o tempo necessário para alcançar o resultado desejado.

Nos testes realizados, notou-se que a máquina responsável pela criação dos processos consome alguns segundos a mais para processar a imagem. Como por exemplo, a imagem 3000x3000 *pixels*, com a técnica Superamostragem, processando juntamente com a execução do *Photoshop*, consumiu 31 segundos a mais que o

computador processando isoladamente. Possivelmente devido o *Photoshop* estar utilizando tempo de processador e memória.

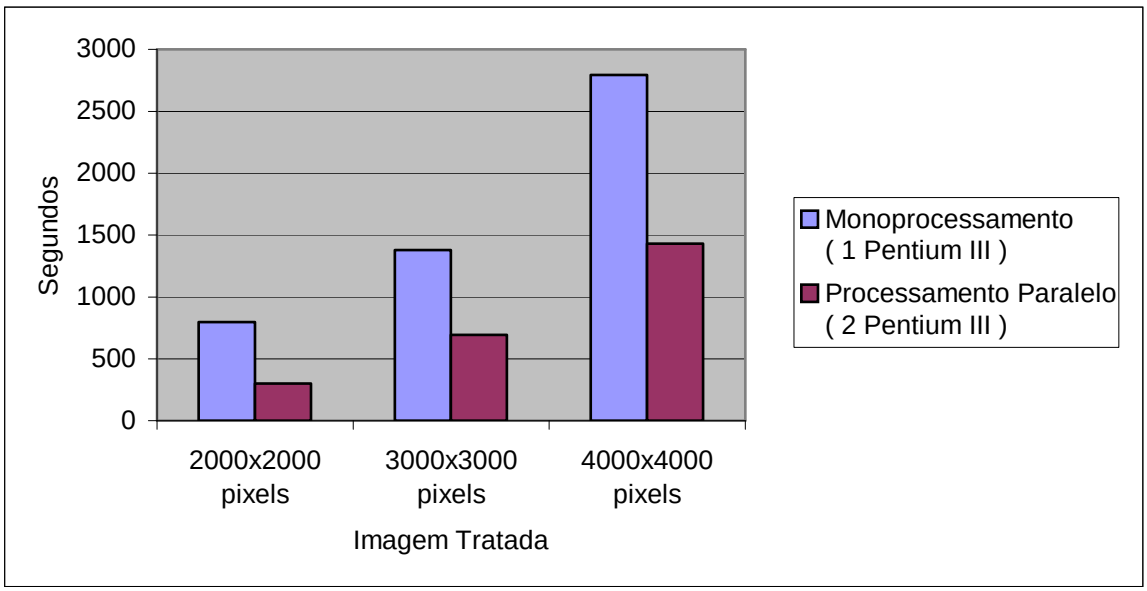
O Gráfico 1, demonstra os valores contidos nas Tabelas 2 e 3 referente a técnica de *antialiasing* Superamostragem.

Gráfico 1 – Comparação Processamento Técnica Superamostragem



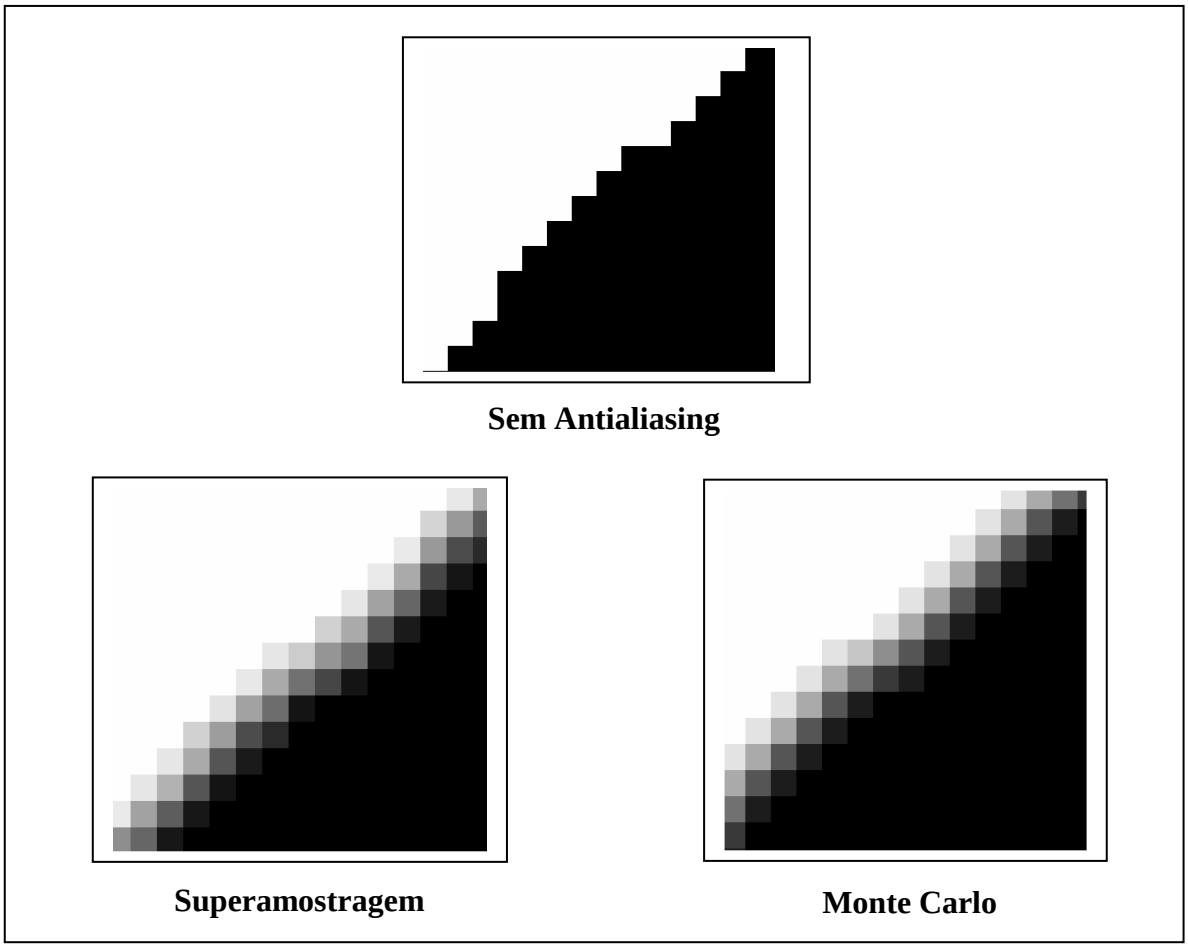
O Gráfico 2, demonstra os valores contidos nas Tabelas 2 e 4 referente a técnica de *antialiasing* Monte Carlo.

Gráfico 2 - Comparação Processamento Técnica Monte Carlo



Na Figura 15 é possível verificar a ação do tratamento de *aliasing*. As imagens foram aproximadas diversas vezes para verificarmos de forma mais precisa, podendo visualizar as fronteiras dos *pixels*.

Figura 15 – Imagens resultantes do tratamento de *aliasing*



7 CONCLUSÕES

O protótipo de *plug*-in para aplicação e demonstração de técnicas de *antialiasing* com processamento paralelo utilizando a biblioteca WPVM, cumpriu com os objetivos propostos. Abordou as características especificadas, entre elas o estudo, especificação e implementação do *plug-in* para o *Photoshop.* O programa “Master” foi desenvolvido fazendo uso da biblioteca WPVM e o programa “Slave” tratando o problema do *aliasing* através das técnicas de *antialiasing* Superamostragem e Monte Carlo juntamente com a biblioteca WPVM.

Durante a realização deste trabalho, o restrito e confuso conteúdo referente aos *plug-ins* do *Photoshop* foi a principal dificuldade encontrada.

Os testes e avaliações realizados revelaram, entre outros pontos, que o tratamento do *aliasing* através de processamento paralelo se mostrou bastante compensador, pois em sua maioria, conseguiu reduzir drasticamente o tempo gasto para este processo. Assim como o *plug-in* para o *Photoshop* se mostrou uma importante opção de *software* para a área gráfica, devido o usuário poder usufruir do sistema desenvolvido como também terá a sua disposição outras ferramentas disponíveis no próprio *Photoshop.* Outro ponto de destaque foi a biblioteca WPVM que se mostrou uma importante solução para aplicações que necessitem o uso de processamento paralelo. Sendo que a biblioteca é de fácil utilização e entendimento e ainda disponibiliza uma grande quantidade de funções que facilitam a implementação de aplicações deste gênero. Porém tendo como limitação, o uso da plataforma *Microsoft Windows*.

Atingidos os objetivos propostos, destaca-se algumas restrições referente ao sistema desenvolvido. São elas:

a) o sistema não permite visualização correta da imagem original e da resultante através da tela do *plug-in* devido não efetuar ajuste na escala de exibição, sendo assim, uma imagem muito grande não é totalmente visualizada na tela do *plug-in*.
b) não é possível utilizar a biblioteca WPVM diretamente pelo *plug-in* devido incompatibilidade de bibliotecas necessárias a sua execução. Sendo que os *plug-ins* utilizam biblioteca da própria da *Adobe* para manter compatibilidade

entre as plataformas *Microsoft Windows* e *Macintosh* pois os programas da *Adobe* focam ambas as plataformas o que não ocorre com a biblioteca WPVM que é de uso exclusivo na plataforma *Microsoft Windows*;

c) com relação às imagens tratadas pelo protótipo desenvolvido, sua capacidade limita-se à própria capacidade de manipulação de memória do Sistema Operacional;
d) devido o protótipo utilizar a biblioteca WPVM, fica restrito o uso do protótipo em plataformas *Microsoft Windows*.

7.1 EXTENSÕES

A área de tratamento de imagens é bastante vasta e abrangente, possibilitando a implementação de novos recursos e técnicas no protótipo desenvolvido. A título de sugestão, pode-se citar alguns tópicos a serem abordados em projetos futuros:

a) adicionar a implementação de técnicas de tratamento de imagens armazenadas em arquivos vetoriais como, por exemplo, a Pré-Filtragem;
b) possibilitar também a manipulação de imagens armazenadas em outros formatos de arquivos;
c) adicionar outras manipulações de imagens com processamento distribuído, como por exemplo, renderização;
d) adicionar controle de erros na execução do sistema, como por exemplo, caso ocorra algum erro em determinado processo, o sistema identifique e recrie o processo que falhou;
e) adicionar à implementação a opção de processar mais de uma técnica de *antialiasing* ao mesmo tempo.

REFERÊNCIAS BIBLIOGRÁFICAS

ADOBE SYSTEMS INCORPORATED. **Adobe developers association graphics and publishing SDK.** Disponível em: <http://partners.adobe.com/asn/developer/>. Acesso em 07 dez. 2000.

ALSPACH, Ted. **Guia incrível do Photoshop.** São Paulo: Makron Books, 1995.

ALVES, Alexandre; SILVA, Luis; CARREIRA, João; SILVA, João Gabriel. WPVM: Parallel computing for the people. In: HPCNs 95 High Performance Computing and Networking Europw, Milano – Italia, mai. 1995.

AMARAL, Antônio Carlos de C.; OLIVEIRA, Iraci; ANDRADE, José Pedro Pereira; ARAUJO, Sérgio Luiz M.. **Banco de dados dsitribuídos.** Porto Alegre, [1999?]. Disponível em: <http://planeta.terra.com.br/informatica/arruda/antiga/artigos/esp_cs_991/grupo04/>. Acesso em 10 maio 2001.

BANON, Gerald Jean Francis. **Bases da computação gráfica.** Rio de Janeiro: Campus, 1989.

BLANK, Guido. **Especificação e implementação de um protótipo de filtro gráfico utilizando técnicas de antialiasing.** 1998. 53 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

DOROW, Elisabeth Ignes. **Estudo das implementações algorítmicas de PVM.** 1997. 99 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

FACHINI, Ricardo. **Protótipo de ferramenta/plug-in para geração de imagens raster 2D em grayscale para o Photoshop.** 2000. 76 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HILL, Francis S. **Computer graphics.** Englewood Cliffs, USA: Macmillan, 1990.

KAUSHIK, Shudir R. **Antialiasing techniques.** Worcester, [1997?]. Disponível em: <http://www.cs.wpi.edu/~matt/courses/cs563/talks/antialiasing/>. Acesso em: 15 mar. 2001.

STRACK, Jair. **Sistemas de processamento distribuído.** Rio de Janeiro: LTC, 1984.

SUN MICROSYSTEMS INC. **An introduction to computer graphics concepts.** California, USA: Addison-Wesley Publishing Company. 1991.

SWAN, Tom. **Programação avançada em Borland C++ 4 para Windows**. São Paulo: Berkeley, 1994.

WIJEGUNARATNE, Inji; FERNANDEZ, George. **Distributed applications engineering.** Great Britain: Springer, 1998.