UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

DESENVOLVIMENTO DE UMA FERRAMENTA PARA COMPARAÇÃO DE DESEMPENHO ENTRE ROTINAS IMPLEMENTADAS EM CPU E GPU

PABLO SIDNEY CORRÊA

BLUMENAU
2009

2009/1-15

PABLO SIDNEY CORRÊA

DESENVOLVIMENTO DE UMA FERRAMENTA PARA COMPARAÇÃO DE DESEMPENHO ENTRE ROTINAS IMPLEMENTADAS EM CPU E GPU

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Dalton Solano dos Reis, M.Sc. - Orientador

BLUMENAU
2009

2009/1-15

DESENVOLVIMENTO DE UMA FERRAMENTA PARA COMPARAÇÃO DE DESEMPENHO ENTRE ROTINAS IMPLEMENTADAS EM CPU E GPU

Por

PABLO SIDNEY CORRÊA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Dalton Solano dos Reis, M.Sc. – Orientador, FURB
Membro:	Prof. Paulo César Rodacki Gomes, Dr. – FURB
Membro:	Prof. Francisco Adell Péricas, M.Sc. – FURB

Blumenau, 09 de julho de 2009

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, que mesmo longe, sempre esteve presente.

Aos meus amigos, pelos empurrões e cobranças.

Ao George Ruberti Piva pela ajuda e cooperação ao trabalho.

Ao meu orientador, Dalton Solano dos Reis, por ter acreditado na conclusão deste trabalho.

RESUMO

No presente trabalho são descritas a especificação e implementação de técnicas utilizadas na representação de dinâmica de fluídos, sendo convertidas para a linguagem Cg e processadas pela GPU. A implementação do trabalho referente à dinâmica de fluídos foi desenvolvida pelo Piva, Gomes e Reis (2008), sendo que o objetivo desse trabalho é utilizar o aplicativo já pronto e realizar a conversão das técnicas para GPU. Foi realizada comparação de processamento, memória e representação gráfica obtida das técnicas de `colormap` e `vectorfield`. Entre os tempos encontrados no processamento das técnicas em GPU e CPU mostrou-se que a GPU é mais rápida, contudo sobre certas condições a CPU é mais indicada para realizar os cálculos.

Palavras-chave: GPU. *Shader*. Dinâmica de fluídos.

ABSTRACT

In this work are described in the specification and implementation techniques used in representing the dynamics of fluids and is converted to the Cg language and processed by the GPU. The implementation of the work on the dynamics of fluids was developed by Piva, Gomes and Reis (2008), with the aim of this work is to use the application already and ready to convert the techniques for GPU. Comparison was done processing, memory and graphics from the techniques of colormap and vectorfield. Between the times found in the processing techniques in GPU and CPU showed that the GPU is faster, but on certain conditions, the CPU is more suitable to perform the calculations.

Key-words: GPU. Shader. Dynamics of fluids.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

2D – 2 Dimensões

3D – 3 Dimensões

AGP – *Accelerated Graphics Port*

API – *Application Programming Interface*

ARB – *Architecture Review Boards*

ASHLI – *Advanced SHading Language Interface*

Cg – *C for graphics*

CGA – *Color Graphics Adapter*

COLLADA – *COLLAborative Design Activity*

CPU – *Central Processing Unit*

EGA – *Enhanced Graphics Adapter*

EISA – *Extended Industry Standard Architecture*

GB – *GygaBytes*

GHz - *GigaHertz*

GLSL – openGL *Shading Language*

GPGPU – *General Purpose for Graphics Processing Unit*

GPU – *Graphics Processing Unit*

HLSL – *High Level Shading Language*

IDE – *Integrated Development Environment*

ISA – *Industry Standard Architecture*

IP – Internet *Protocol*

MCA – Micro *Channel Architecture*

MDA – *Monocrome Display Adapter*

PCI – *Peripheral Component Interconnect*

PCIe - *Peripheral Component Interconnect express*

RAM – *Random Access Memory*

RAMDAC – *Random Access Memory Digital-to-Analog Converter*

SLI – *Scalable Link Interface*

SVGA – Super Video *Graphics Adapter*

TCP – *Transmission Control Protocol*

UFRGS – Universidade Federal do Rio Grande do Sul

UML – *Unified Modeling Language*

VESA – Video *Electronics Standards Association*

VGA – Video *Graphics Adapter*

VLB – VESA Local *Bus*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Com a evolução da tecnologia especialmente dos hardwares, o nível de realidade empregada nos softwares cresce cada vez mais. Os usuários destes aplicativos não se contentam mais com simples interfaces e desejam ambientes mais próximos da realidade cotidiana, que além de tudo o que já fazem também sejam capazes de processar grandes quantidades de dados e permitir manipulações interativas. As ferramentas criadas de forma tradicional, utilizando-se apenas da *Central Processing Unit* (CPU) para realizar estas tarefas, estão ficando obsoletas, isso porque o processador sozinho já não consegue mais gerenciar e processar todos os dados necessários para gerar o efeito desejado dentro de um tempo aceitável.

Para atingir estes objetivos se faz necesária a utilização de hardwares gráficos. Os dispositivos gráficos atuais vêm com processadores próprios acoplados a eles, também denominados de *Graphics Processing Unit* (GPU), que podem ser programados utilizando-se linguagens específicas para este fim. Com isso, consegue-se um maior nível de processamento das imagens e libera-se a CPU para realizar outras tarefas. Segundo Fernando (2004), os verdadeiros beneficiados com a possibilidade de programar a GPU serão os criadores de algoritmos para computação gráfica, pesquisadores, desenvolvedores de aplicativos e usuários finais.

A GPU, ou seja, o *chipset* de vídeo, não tem finalidades apenas no processamento gráfico. Para Pharr e Fernando (2005), a GPU pode ser utilizada também para detecção de colisão, cálculos de física e computação numérica, além de outras aplicações. Os desenvolvedores começam a deixar de lado a visão de que a GPU deve ser utilizada apenas para processamento gráfico e iniciam o estudo do desenvolvimento de aplicações de propósito geral utilizando o hardware gráfico. Esta idéia ainda está em fase inicial e um dos grandes problemas para se evoluir neste conceito é a limitação física das placas aceleradoras que não comportam este tipo de programação. Mas atualmente estão surgindo ferramentas que auxiliam no desenvolvimento de aplicativos para este propósito.

Diante da necessidade de maior processamento gráfico e de dados, pretende-se com este trabalho criar uma ferramenta que permita comparar o desempenho entre rotinas de dinâmica de fluídos implementadas em CPU e GPU.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta que permita realizar comparações entre rotinas implementadas em CPU e GPU, demonstrando as vantagens e desvantagens da utilização do processador gráfico.

Os objetivos específicos do trabalho são:

a) analisar as principais características da GPU e da linguagem *C for graphics* (Cg);
b) implementar em GPU uma rotina particular[1] que é utilizada em dinâmica de fluídos, já implementadas em CPU;
c) desenvolver rotinas para permitir realizar comparações de tempo de processamento, utilização de memória e aspectos visuais entre as rotinas de dinâmica de fluídos implementadas.

1.2 ESTRUTURA DO TRABALHO

O trabalho está estruturado em 4 capítulos. No segundo capítulo é apresentado a fundamentação teórica necessária para o entendimento do trabalho. Nele são expostos os conceitos sobre o hardware gráfico, sua evolução e funcionamento interno. Mostra também a definição das principais linguagens para o hardware gráfico disponíveis, como também ambientes desenvolvidos para utilização das mesmas. Neste capítulo é mostrada ainda uma definição sobre dinâmica de fluidos e alguns trabalhos correlatos. No terceiro capítulo será mostrado o desenvolvimento do sistema. Nesse capítulo pode ser encontrado os requisitos e especificação do trabalho como também a descrição dos algoritmos implementados e os resultados encontrados. Por fim, no quarto capítulo são descritas as conclusões encontradas e as extensões para trabalhos futuros.

[1] Foi utilizado o trabalho do Piva (2008), sendo que somente as técnicas `colormap` e `vectorfield` foram convertidas para GPU.

2 FUNDAMENTAÇÃO TEÓRICA

A seguir será explanado sobre o hardware gráfico, os conceitos que o formam e suas vantagens e limitações. Na sequência o funcionamento do *pipeline* gráfico, como o processador de vértices e fragmentos serão abordados. Após serão expostas as principais linguagens de programação existentes para GPU. Logo em seguida são detalhadas as principais *Integrated Development Environment* (IDE) disponíveis para desenvolvimento e as linguagens suportadas por cada uma. Na continuação são descritos alguns conceitos e características da simulação de dinâmica de fluídos. Por fim, em trabalhos correlatos, são mencionadas ferramentas existentes no mercado com funcionalidades semelhantes às da ferramenta que foi implementada.

2.1 HARDWARE GRÁFICO

Inicialmente as placas de vídeo eram dispositivos simples, cuja função era apenas mostrar o conteúdo da memória de vídeo no monitor. O *Random Access Memory Digital-to-Analog Converter* (RAMDAC), que é uma parte da placa gráfica, lia a imagem da memória de vídeo periodicamente e a enviava para o monitor (MORIMOTO, 2007). De acordo com Morimoto (2002), as placas de vídeo criadas até hoje já utilizaram todos os tipos de barramentos existentes. O primeiro barramento utilizado foi o *Industry Standard Architecture* (ISA), seguido do Micro *Channel Architecture* (MCA), *Extended Industry Standard Architecture* (EISA), VESA Local *Bus* (VLB), *Peripheral Component Interconnect* (PCI), *Accelerated Graphics Port* (AGP) e chegando no atual barramento para placas de vídeos o *Peripheral Component Interconnect express* (PCIe). Com a evolução dos barramentos, os mesmos tornavam-se cada vez mais rápidos incorporando mais recursos e um maior desempenho (MORIMOTO, 2002).

Juntamente ao barramento, outra tecnologia que se desenvolveu foi o padrão de vídeo utilizado. Este é responsável pela qualidade gráfica suportada pelos monitores, definindo a resolução em *pixels* com que esses podem atualizar a tela. Para que esse desenvolvimento acontecesse também era necessário que os monitores evoluíssem, tornando-se assim capazes de suportar as novas especificações criadas. Em seguida serão mostrados os principais

padrões de vídeo elaborados seguindo sua ordem cronológica:

a) *Monocrome Display Adapter* (MDA) e *Color Graphics Adapter* (CGA): utilizados nos primeiros computadores, esses dois formatos que limitados pelos recursos de processamento dos computadores da época, eram utilizados basicamente para interfaces somente textos. O MDA era o mais primitivo e barato, podendo suportar 25 linhas por 80 colunas e mostrava apenas duas cores simultaneamente. Já o CGA tinha a capacidade de mostrar resoluções gráficas de 320 x 200 utilizando quatro cores simultaneamente. Existia a capacidade de aumentar essa resolução para 640 x 200, com a consequência de utilizá-la apenas em modo monocromático (MORIMOTO, 2002);
b) *Enhanced Graphics Adapter* (EGA): foi desenvolvido pela IBM em 84 para equipar o PC AT. Suportava exibição de gráficos com resolução de até 640 x 350 utilizando 16 cores simultâneas (MORIMOTO, 2002). Mantinha total compatibilidade com o CGA;
c) Video *Graphics Adapter* (VGA): foi uma revolução sobre os padrões de vídeos antigos, suportando resoluções de 640 x 480 com 256 cores simultaneamente. Mais tarde esse padrão foi aperfeiçoado estendendo sua capacidade para definições gráficas de 800 x 600 com 16 cores. Continuava mantendo compatibilidade com o CGA e EGA, que assim permitia rodar aplicativos mais antigos (MORIMOTO, 2002);
d) Super Video *Graphics Adapter* (SVGA): padrão ainda utilizado atualmente é capaz de exibir até 32 *bits* de cores simultaneamente (MORIMOTO, 2002). Foram especificados três padrões para o SVGA, o Video *Electronics Standards Association* (VESA)1, VESA2 e VESA3 que definem resoluções diferentes cada um, sendo que o padrão VESA3 ainda é utilizado nos monitores atuais. Esse não foi o último padrão a ser elaborado, muitos outros já foram criados para satisfazerem computadores com resoluções diferentes.

Outro item importante das placas aceleradoras de vídeo que evoluiu foi a memória. Nas primeiras placas lançadas a memória era utilizada apenas para armazenar o conteúdo que seria exibido na tela do monitor. Quanto maior a capacidade de armazenamento da memória, maior seria a resolução e a quantidade de cores simultâneas suportadas pela placa. Atualmente as memórias das placas de vídeo são tão rápidas e com tamanho de armazenamento similar as utilizadas para a CPU. Um dos objetivos atuais da memória de vídeo é armazenar as texturas utilizadas pelos aplicativos gráficos (MORIMOTO, 2002). Isso se deve ao fato do acesso e

transferência de dados da memória *Random Access Memory* (RAM) para a placa de vídeo ser muito lenta se comparar com o acesso direto na memória da placa gráfica.

As primeiras placas evoluíram e começaram a aceitar recursos de aceleração, que melhorou o processamento de imagens em 2 dimensões (2D). Em seu próximo passo de evolução, começaram a aceitar a renderização de objetos em 3 dimensões (3D) e neste ponto começou a entrar a GPU, que é o processador gráfico da placa. A partir daí a GPU evoluía cada vez mais, suportando renderização de efeitos cada vez mais complexos e ficando cada vez mais rápida.

Como em processadores de CPU, o desempenho do *chipset* gráfico depende de vários fatores como a frequência de operação, *pixels* por ciclo de clock, *Fill Rate*, quantidade de polígonos processados por segundo, barramento de memória, resolução utilizada, efeitos 3D utilizados e os *drivers* instalados. Todos esses itens serão explanados brevemente abaixo.

A frequência de operação da GPU funciona como a da CPU, ela indica quantas instruções o processador gráfico pode executar em um segundo. Nas placas de vídeos mais recentes a velocidade do *chipset* pode chegar a ser superior a 500Mhz, ou seja, mais de 500 milhões de instruções por segundo. A principal diferença entre processadores da CPU e GPU é a quantidade de operações simultâneas que esta pode executar. Em um processador comum somente é executada uma instrução por vez. Já em placas gráficas a quantidade de operações simultâneas é determinada pela arquitetura do hardware gráfico, que geralmente não executa menos de duas instruções paralelamente. Esse processamento paralelo é denominado de *pixels* por ciclo de *clock*. (MORIMOTO, 2002).

Segundo Morimoto (2002), quando multiplicamos a quantidade de *pixels* processados por ciclo de clock pela frequência de processamento do *chipset* gráfico encontramos o *Fill rate*, que nada mais é do que o número de *pixels* por segundo que o hardware gráfico consegue processar. Essa definição não é somente aplicada ao *pixel*, mas também ao *texel*. *Texel* é o ponto que compõe a textura, que é utilizada em polígonos. Dessa forma, em uma placa com processamento de 500Mhz que seja capaz de processar duas instruções por ciclo e tenha suporte a multitextura (supondo que esta placa suporte duas texturas simultaneamente), terá um *Fill rate* de 1000 *megapixels* e 1000 *megatexels* (MORIMOTO, 2002). Ou seja, poderá processar 1 bilhão de pontos de imagem ou 1 bilhão de pontos de texturas por segundo.

A quantidade de polígonos processados por segundo define quantos polígonos o hardware consegue desenhar na tela em um segundo. Quanto maior for o número de polígonos, melhor será a desempenho do *chipset* gráfico. O barramento de memória define a

velocidade de acesso da GPU à memória de vídeo. Quanto mais largo for o barramento de acesso a memória, maior será a velocidade na transferência de dados entre o *chipset* e a memória (MORIMOTO, 2002).

Um dos fatores que contribui para um desempenho mais fraco das placas gráficas é a resolução utilizada na qual o hardware irá trabalhar. Como o *Fill rate* da placa é fixo, e é ele quem define de forma bruta a capacidade do *chipset* gráfico, quanto maior for a resolução utilizada, menor será a quantidade de quadros gerados no intervalo de um segundo (MORIMOTO, 2002).

Na maioria das placas existem efeitos 3D que podem ser configurados no próprio hardware, como por exemplo, o uso de 32 *bits* de cores, filtro anti-serrilhamento, nível de detalhamento dos objetos, entre outros. Esses efeitos causam uma melhor qualidade na imagem final, mas acabam consumindo mais tempo da GPU para poder processá-los (MORIMOTO, 2002).

Por fim os *drivers* podem se tornar um componente crítico para os dispositivos gráficos. São eles que definem todos os recursos que o hardware suporta e como fazer para acessá-los da forma mais rápida possível. São responsáveis por intermediar a comunicação entre o sistema operacional e a placa de vídeo (MORIMOTO, 2002).

Os hardwares gráficos atuais são muito rápidos, compostos por uma quantidade considerável de memória, que é acessada por um barramento muito mais rápido que a da placa mãe. O *chipset* de vídeo responsável pelo processamento gráfico também é muito mais complexo e mais rápido que o processador principal (MORIMOTO, 2007). Isso ocorre pois a GPU realiza processamento paralelo, diferente do sequencial utilizado em CPUs.

Para chegarem no estado atual, os hardwares gráficos foram evoluindo e podem ser divididos em 5 gerações. A primeira geração começou em 1998 e incluía os hardwares da NVidia TNT2, da ATI o Rage e da 3dfx o Voodoo3. Eram capazes de rasterizar triângulos pré-transformados e aplicar uma ou duas texturas. Contudo essas GPUs tinham suas limitações. Primeiro não conseguiam transformar os vértices de objetos 3D, o que acabava sobrando para a CPU realizar o trabalho. Em segundo, tinham uma quantidade limitada de operações matemáticas na combinação de texturas e para calcular a cor do *pixel* rasterizado (NVIDIA, 2009).

A segunda geração foi de 1999 a 2000, e compreendia os dispositivos gráficos GeForce 256 e GeForce2 da NVidia, Randeon 7500 da ATI e Savage3D da S3. As quantidades de operações matemáticas para combinação de texturas e para o cálculo da cor do *pixel* rasterizado foram estendidas, incluindo mapeamento de texturas para cubos e operações

matemáticas com sinal. Essa geração também retira da CPU o trabalho de realizar a rasterização dos vértices e das luzes. Acaba sendo mais configurável, mas ainda não totalmente programável (NVIDIA, 2009).

Na terceira geração que ocorreu em 2001, foram desenvolvidas as placas GeForce 3 e GeForce4 Ti da NVidia, as placas gráficas utilizadas no console da Microsoft o XBOX; e a Radeon 8500 da ATI. Nessa geração, o vértice era totalmente programável e não mais somente configurável. O nível de *pixel* podia ser configurável, mas não era suficientemente poderoso para ser considerado como verdadeiramente programável. Por causa dos dois fatos acima citados esta geração foi considerada como transitória (NVIDIA, 2009).

Já a quarta geração iniciou-se em 2002 e incluía as placas GeForce FX da NVIDIA e Radeon 9700 da ATI. Nessa geração ambos vértices e *pixels* eram totalmente programáveis. Esse nível de programação abre a possibilidade de levar para a GPU operações complexas de vértices e *pixels* (NVIDIA, 2009).

Em sua quinta geração, que começou por volta de 2007, os hardwares gráficos começaram a possuir mais de um núcleo de processamento (BABOO, 2009), e a possibilidade de interligar mais de uma placa para realizar um único processamento. Poderiam ser interligadas em modo *Scalable Link Interface* (SLI) para hardwares desenvolvidos pela NVidia, ou *CrossFire* para dispositivos fabricados pela ATI.

A principal vantagem de uma placa aceleradora 3D é o seu desempenho na renderização de gráficos e o fato dela poder ser programável, deixando assim livre a CPU para realizar outras tarefas necessárias. Mesmo sendo tão útil não é possível criar programas muito complexos para ela, já que possui limitações na quantidade de registradores, como na quantidade de declarações de constantes suportadas. Também não é indicada para realizar processamento sequencial, já que não utilizaria de sua capacidade de executar tarefas simultâneas. A CPU é mais adequada e muito mais rápida para processamento sequencial.

2.2 PROGRAMAÇÃO EM GPU

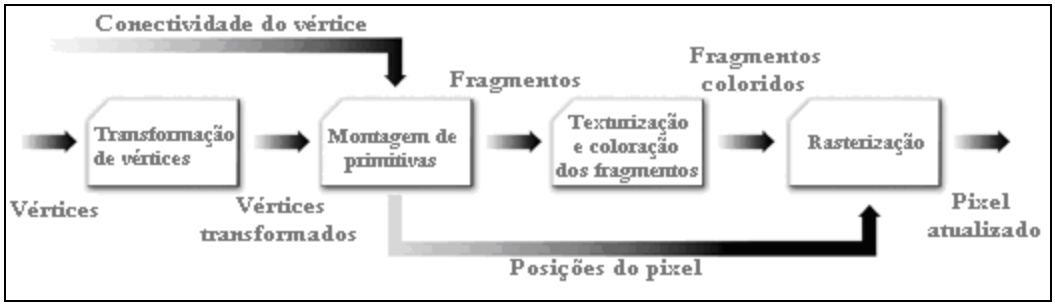
Os softwares desenvolvidos de forma tradicional não utilizam do processamento da GPU das placas de vídeos. Para poder ter acesso a todos os benefícios oferecidos por ela, é necessário codificar parte do programa com uma linguagem específica. Pelo fato de ter sido desenvolvida para processamento gráfico o modelo de programação não é usual. Com isso a

utilização não é tão simples (VIANA; CUNO, 2008).

Internamente, as operações da GPU podem ser divididas em 3 etapas (denominadas *pipeline* gráfico): transformação, rasterização e composição. A transformação consiste em converter a geometria de objetos 3D e espaço 2D. Na fase de rasterização são criados os *pixels* (fragmentos) a partir da geometria 2D gerada na etapa anterior. Já na etapa final, os fragmentos são combinados para formar uma imagem (VIANA; CUNO, 2008). Nos *chipsets* gráficos atuais é possível programar todas as etapas do *pipeline*, mas em placas mais antigas, somente as duas primeiras etapas são programáveis.

2.2.1 Funcionamento do *pipeline* gráfico

Segundo NVidia(2009), o *pipeline* é uma sequência de estágios operando paralelamente e em uma ordem fixa. Cada estágio recebe uma entrada do estágio anterior e envia uma saída para o próximo estágio. Usando uma analogia, pode-se comparar o *pipeline* gráfico com uma linha de montagem de automóveis. Para que o automóvel fique pronto ele passa por vários estágios, sendo que todos eles ocorrem simultaneamente. A Figura 1 mostra um exemplo de *pipeline* gráfico utilizado atualmente. Os quadros dessa figura serão descritos abaixo.



Fonte: Editado de NVidia (2009).

Figura 1 – Exemplo de um *pipeline* gráfico

Todos os vértices possuem uma posição, mas geralmente podem conter outras propriedades também como cor, conjunto de coordenadas de texturas, entre outras (NVIDIA, 2009). No primeiro estágio `Transformação de vértices`, são realizadas várias operações matemáticas em cada vértice. Essas operações consistem em transformar a posição do vértice em uma posição de tela que será usada em um próximo estágio pelo rasterizador, gerar coordenadas para texturização e criar a iluminação do vértice (utilizada para calcular a cor do mesmo) (NVIDIA, 2009).

Os vértices transformados seguem para o próximo estágio, `Montagem de primitivas`.

Nessa etapa acontecem primeiramente duas operações, o *clipping* e o *culling*. O *clipping* é a operação de remover todos os vértices que estão fora de um região visível do espaço 3D (também denominado de *view frustum*[2]), ou até num espaço em 2D. Já o *culling* é o processo que remove os vértices levando em consideração para qual lado esta virado a sua face (NVIDIA, 2009). Isto significa que quando uma esfera está sendo visualizada em um espaço tridimensional vão existir vértices que não poderão ser vistos da posição atual da câmera, pois sua face está direcionada para o lado contrário.

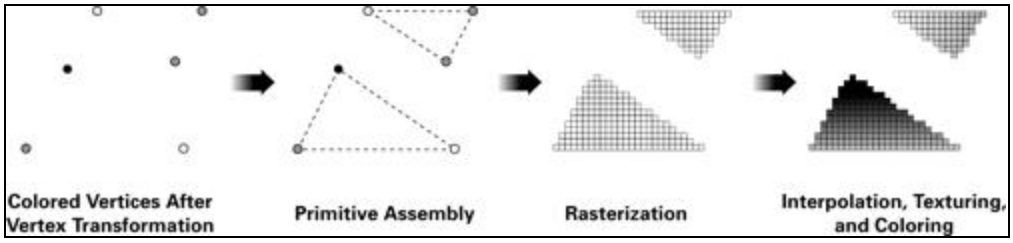
Polígonos que "sobreviverem" a esses passos iniciais são rasterizados. Rasterização é o processo de determinar o conjunto de *pixels* que estão cobertos por uma primitiva geométrica. Cada primitiva é rasterizada de acordo com as regras definidas para a mesma. O resultado desse processo é um conjunto compondo as localizações dos *pixels* e um conjunto de fragmentos (NVIDIA, 2009).

Após os fragmentos serem gerados, eles são encaminhados para o próximo estágio `Texturização e coloração dos fragmentos`. Esse passo executa uma sequência de texturizações e operações matemáticas afim de determinar a cor final do fragmento. Também possui o poder para descartar um fragmento evitando que o mesmo atualize o *pixel* correspondente a ele (NVIDIA, 2009).

Assim que os fragmentos são coloridos eles passam para a etapa final do *pipeline* gráfico, o `Rasterização`. Um dos passos a ser executado é eliminar os vértices que ficam escondidos atrás de outras superfícies, também denominado de *depth testing*. Nesse estágio também são executados vários outros testes em cada fragmento. Se qualquer um dos testes previstos falharem, o fragmento é descartado sem atualizar o *pixel* correspondente a ele. Depois de todos os testes serem executados, é realizada a combinação da cor do fragmento com a cor do *pixel* correspondente. Em seguida o *pixel* no *buffer* de tela é substituído pela cor combinada do fragmento (NVIDIA, 2009).

A Figura 2 demonstra os estágios de *pipeline* gráfico, onde 2 triângulos são rasterizados. Primeiro é realizada a transformação e coloração dos vértices. Em seguida são criadas as primitivas dos triângulos transformados. O próximo passo da imagem é a criação dos fragmentos, levando finalmente para a fase onde ocorre a interpolação, texturização e coloração dos mesmos (NVIDIA, 2009).

[2] Em ambientes de 3 dimensões esse espaço tem o formato de um tetraedro, e contém um limite de início denominado de *Z-near* e um limite para distância máxima, chamado de *Z-far*.



Fonte: NVidia (2009).

Figura 2 – Passos do *pipeline* gráfico renderizando 2 triângulos

2.2.1.1 Programação de vértices

Uma vantagem das GPUs modernas é a possibilidade de programação de alguns estágios do *pipeline* gráfico. Um desses estágios é a transformação de vértices, conhecido também como *vertex shader*. O processador de vértices necessita como entrada de alguns atributos como posição, cor, coordenadas de textura, etc. As instruções são decodificadas e executadas uma a uma até que o programa termine. Essas instruções podem acessar um conjunto de registradores distintos que contém vetores com valores como posição, normal[3] ou cor. Esses registradores podem ser acessados como somente leitura. Existem também registradores onde é possível realizar operações de escrita e leitura. Esses são utilizados para armazenamento temporário, salvando os valores contidos nas variáveis criadas dentro do programa. Quando a execução da aplicação termina, os vértices transformados são transferidos para o registrador de saída, onde é permitido somente escrever (NVIDIA, 2009).

O processador de vértices substitui todas as funcionalidades do primeiro estágio do *pipeline*. Isto é, os programas criados para processar os vértices precisam obrigatoriamente calcular uma posição, mas podem definir também outros atributos como cor, normais e coordenadas de textura. As principais operações dessa unidade são: a transformação de vértices, transformação de normais e sua normalização, geração de coordenadas de textura, transformação de coordenadas de textura, iluminação e cálculo de material (FERNANDES, 2006).

[3] Atributo do vértice utilizado nas técnicas de iluminação do mesmo.

2.2.1.2 Programação de fragmentos

Pode-se dizer que fragmento é um ponto dentro de uma primitiva geométrica. Conforme NVidia (2009), o fragmento está associado a localização de um pixel, a um valor de profundidade e a um conjunto de valores interpolados como a uma cor, uma segunda cor (especular), e um ou mais conjuntos de coordenadas de texturas. O fragmento é um *pixel* em potencial, basta conseguir passar por todos os testes no estágio de *raster* para atualizar os valores do *pixel* correspondente no *buffer* da tela (NVIDIA, 2009).

A programação de fragmentos (*pixel shader*) suporta as mesmas operações matemáticas realizadas com os vértices. Contudo esse estágio provê a possibilidade de realizar operações de textura também. Essa característica permite ao processador acessar uma imagem através de um conjunto de coordenadas de textura, e então retornar uma amostra filtrada dela (NVIDIA, 2009).

Como no processador de vértices, as instruções são executadas uma a uma até que o programa termine. No entanto, os atributos de entrada contidos nos registradores são diferentes. Eles contêm os fragmentos interpolados que foram gerados no processo anterior (processamento de vértices). Os cálculos executados para determinar a cor final do fragmento podem ser salvos nos registradores temporários. Nos programas de fragmentos é possível retornar somente um valor no registrador de saída, a cor do fragmento. Em hardwares gráficos mais modernos é possível retornar também a profundidade da cor (NVIDIA, 2009).

Essa etapa substitui todas as funcionalidades do terceiro estágio do *pipeline*. As principais operações que podem ser executadas são: operações sobre valores interpolados como cores, normais e coordenadas de texturas, acesso a texturas, aplicação de texturas, cálculo de nevoeiro e cálculo da cor. Os programas não podem alterar a posição x e y do *pixel* representado pelo fragmento (FERNANDES, 2006).

2.2.2 Linguagens de programação

Para se ter acesso aos recursos disponibilizados pelas placas aceleradoras de vídeo, é necessário codificar parte do programa em uma linguagem nativa do processador da placa. Da mesma forma que existe uma linguagem de baixo nível, que é utilizada na programação de aplicativos convencionais (atualmente os compiladores encarregam-se de gerar a linguagem

de baixo nível), existe também a mesma idéia para utilizar o processador da GPU. Pelo fato do hardware gráfico ser diferente da CPU, a linguagem também acaba mudando e a dificuldade de entender o que se está fazendo utilizando apenas linguagem de baixo nível, aumenta em efeitos mais complexos (C FOR ..., 2008).

Para ajudar no desenvolvimento de sistemas que utilizam dos recursos da GPU, os fabricantes de hardware, como também as empresas que desenvolvem *Application Programming Interfaces* (APIs) para criação de aplicativos gráficos, criaram linguagens de alto nível, abstraindo toda a dificuldade da linguagem de baixo nível, proporcionando assim um ganho em produtividade. Dentre as linguagens disponíveis, citam-se as seguintes: openGL *Shading Language* (GLSL), *High Level Shading Language* (HLSL) e Cg, que serão descritas abaixo.

2.2.2.1 GLSL

Desenvolvida pela empresa 3Dlabs em conjunto com a OpenGL *Architecture Review Boards* (ARB) (UNIVERSITY OF PENNSYLVANIA, 2008, p. 11), a GLSL é uma linguagem de alto nível, baseada na linguagem C/C++. Por ser parecida com uma linguagem já conhecida, torna a codificação muito mais fácil e intuitiva. Como foi desenvolvida para uma API, os aplicativos feitos com ela rodam em qualquer hardware com suporte a OpenGL 2.0[4].

A GLSL está atualmente na versão 1.2 e os principais recursos desta linguagem serão especificados a seguir. Existe um total de 52 palavras chaves utilizadas que vão desde a definição do tipo de variável até comandos de teste e laço. Há também 47 outras palavras reservadas para uso futuro na linguagem. Esses comandos para uso futuro hoje não tem função nenhuma na linguagem, mas o uso delas mesmo que para declarar o nome de uma variável acarretam em erro no compilador (KESSENICH; BALDWIN; ROST, 2006).

A declaração do identificador deve começar com qualquer letra ou o símbolo de sobre linha, seguida de qualquer número, letra ou sobre linha. Os identificadores que começarem com `gl_` são de uso exclusivo da OpenGL, não devem ser utilizados na declaração de variáveis e nem de funções. É importante frisar que a linguagem faz distinção de letras maiúsculas e minúsculas. GLSL suporta também vários tipos básicos e permite a

[4] Nome e versão da API criada pela empresa OpenGL.

possibilidade de utilizar estruturas e `arrays` (KESSENICH; BALDWIN; ROST, 2006). Esses tipos de variáveis podem ser divididos em vários grupos.

Primeiramente pode-se citar o grupo de variáveis escalares. Neste grupo encontram-se tipos nativos da especificação C\C++ que também são utilizados em GLSL, sendo esses os tipos `void, bool, int e float`. A utilização de `void` pode ser feita somente na declaração de funções e indica que não será retornado nada pela função. O tipo `bool` recebe valores de verdadeiro ou falso somente. Os inteiros representados pelo `int` possuem precisão de 16 *bits*. A única conversão de tipos possíveis é feita com o inteiro (`int`), onde é possível transformá-lo em um número de ponto flutuante. Pontos flutuantes (`float`) são utilizados para realizar uma variedade de cálculos escalares. Tanto os números inteiros como os pontos flutuantes não suportam números negativos. A utilização do sinal antes dos números é interpretada como um operador unário de negação (KESSENICH; BALDWIN; ROST, 2006).

Outro grupo existente é o de vetores, variáveis que podem conter 2, 3 e 4 componentes com valores dos tipos de pontos flutuantes, numéricos e boleanos. Vetores de pontos flutuantes são bastante úteis para armazenar valores muito utilizados para gráficos como cores, normais, posições, coordenadas de texturas entre outras. Vetores boleanos geralmente são utilizados para comparação de vetores numéricos. Nesse grupo constam os seguintes tipos: `vec2, vec3, vec4, bvec2, bvec3, bvec4, ivec2, ive3 e ivec4` (KESSENICH; BALDWIN; ROST, 2006). O número no final do tipo indica quantos componentes esse possui. A letra que antecede a palavra `vec` informa o tipo de vetor, b para boleano e i para inteiro.

Existe também um grupo de matrizes, onde os seus elementos são do tipo de pontos flutuantes. Podem possuir tamanhos de 2x2, 2x3, 2x4, 3x2, 3x3, 3x4, 4x2, 4x3 e 4x4. O primeiro número indica a quantidade de colunas e o segundo a quantidade de linhas. Encontram-se nesse grupo os tipos `mat2, mat3, mat4, mat2x2, mat2x3, mat2x4, mat3x2, mat3x3, mat3x4, mat4x2, mat4x3, mat4x4`. Não é possível criar matrizes com tamanhos superiores aos citados (KESSENICH; BALDWIN; ROST, 2006).

Outro grupo existente é composto por variáveis utilizadas para referenciar uma textura. Somente pode ser declarada como parâmetro das funções, ou variáveis do tipo `uniform`. Não é possível inicializar este tipo de variável em um *shader*. Quando declarado como `uniform` somente a OpenGL API pode inicializá-la. Os seguintes tipos de variáveis integram esse grupo: `sampler1D, sampler2D, sampler3D, samplerCube, sampler1Dshadow e sampler2Dshadow` (KESSENICH; BALDWIN; ROST, 2006).

O grupo de estrutura é composto por tipos de dados definidos pelo usuário. Podem integrar tipos de dados mencionados acima como também outras estruturas criadas pelo usuário. São declaradas utilizando a palavra reservada `struct`. Não é possível inicializar diretamente as variáveis declaradas nas estruturas (KESSENICH; BALDWIN; ROST, 2006).

Por fim existe o grupo dos `arrays`, que são utilizados para agrupar variáveis do mesmo tipo. Para utilizá-los deve-se colocar colchetes após o nome da variável. Os `arrays` podem ser inicializados com um tamanho que deve ser um inteiro maior que zero. É possível declarar um `array` sem um tamanho e depois redeclarar novamente atribuindo dessa vez o tamanho a ele. Contudo não é permitido declarar o `array` e definir seu tamanho e depois declará-lo novamente com outro tamanho. É permitido utilizar somente `arrays` de uma dimensão (KESSENICH; BALDWIN; ROST, 2006).

As variáveis possuem também qualificadoras que definem como será o acesso a memória entre outras coisas. As qualificadoras devem ser colocadas antes do tipo da variável. Quando uma variável local é declarada sem utilizar nenhum qualificador ela aloca memória para leitura e escrita. Se essa variável for um parâmetro de uma função, então será um dado de entrada. Se utilizado o qualificador `const`, a variável tem somente acesso de leitura na memória e deve ser inicializada na sua declaração. Os qualificadores nomeados `attribute` podem ser usados somente no processamento de vértices (*vertex shader*). Esse tipo de variável pode somente realizar a leitura da memória, e sua inicialização é de responsabilidade da OpenGL API. É possível utilizá-lo somente em variáveis de pontos flutuantes, vetores de pontos flutuantes e matrizes. Já o qualificador chamado `uniform` é utilizado para declarar variáveis globais que serão utilizadas por todas as primitivas processadas. Esse qualificador possui acesso para realizar somente a leitura da memória e sua inicialização é feita pela API do OpenGL. Existe ainda um último qualificador denominado `varying`. Este define uma ligação entre o processador de vértices e fragmentos. O processador de vértices pode escrever os valores nessa variável, que depois será lida pelo processador de fragmentos. Esse tipo de variável requer um escopo global e deve ser declarada fora do corpo das funções (KESSENICH; BALDWIN; ROST, 2006).

Ainda sobre os qualificadores, existem alguns que se aplicam somente aos parâmetros. O qualificador de entrada se chama `in` e serve para determinar quais parâmetros enviam valores para a função. Existem também os qualificadores de saída, denominados `out`. Seu principal objetivo é retornar o valor da função. Geralmente os parâmetros definidos como `out` pelo processador de vértices são recebidos como parâmetros de entrada no processador de

fragmentos (caso haja um *shader* de fragmentos). Por último existe um qualificador cujo nome é `inout`. Este tem tanto propriedades de entrada como de saída (KESSENICH; BALDWIN; ROST, 2006).

A linguagem GLSL também oferece a possibilidade de utilizar construtores para inicializar estruturas, vetores, matrizes e `arrays`. Os construtores devem começar com o tipo de dado que esta se tentando inicializar seguidos por um parêntese. Após deve-se colocar os valores que se deseja para o construtor. Pode-se colocar também outros tipos de dados dentro deste, desde que satisfaça o tipo de dado que deve ser inicializado. Por fim coloca-se outro parêntese para finalizar o construtor. A variável que se deseja inicializar deve receber este construtor como se fosse uma atribuição de valor (KESSENICH; BALDWIN; ROST, 2006).

Uma vantagem da linguagem GLSL é a possibilidade de realizar operações matemáticas simples entre matrizes do mesmo tamanho. Ou seja, é possível somar uma matriz de 4x4 com outra do mesmo tamanho simplesmente colocando o sinal de + entre elas (KESSENICH; BALDWIN; ROST, 2006). Para realizar esse mesmo processo em C\C++, seria necessário somar cada componente das matrizes separadamente. Operadores binários para alteração nos dados não são suportados nessa versão, somente operadores de lógica binária como `&&` ou `||` são utilizados.

Alguns comandos de seleção, interação e desvio também são suportados. Dentro dos comandos de seleção existe o `if-else`. É possível utilizar também o operador ternário de seleção (`expressão1 ? expressão2 : expressão 3`). Onde `expressão1` é a sentença que será validada, `expressão2` será o retorno do operador caso a sentença seja verdadeira e `expressão3` se for falsa. Para os comandos de interação existe o `for, while e do-while`. A sintaxe tanto para os comandos de seleção como para os comandos de interação são iguais à especificada em C\C++. Nos comandos de desvio podemos citar o `continue, break, return e discard`. Os comandos `continue e break` podem ser utilizados somente em laços. O primeiro permite pular de qualquer lugar do corpo de um laço, para a expressão que é validada para continuar no mesmo. Já o outro força a saída do laço mesmo que a condição não seja satisfeita. O comando `return` retorna o valor atribuído a ele e finaliza a função mesmo não chegando ao seu fim. Já o `discard` pode ser utilizado apenas com fragmentos. Ele descarta o fragmento que esta sendo processado, não permitindo assim que o *buffer* de vídeo seja atualizado pelo fragmento eliminado (KESSENICH; BALDWIN; ROST, 2006).

A própria linguagem GLSL possui várias funções pré-definidas que auxiliam na construção de programas de *shaders*. Existem funções para ângulos e trigonometrias, funções

exponenciais, funções comuns (como `abs` que retorna um número positivo), geométricas, para matrizes, outras que realizam comparações entre vetores, as que operam com texturas e finalmente as utilizadas somente em fragmentos (KESSENICH; BALDWIN; ROST, 2006).

Um detalhamento mais minucioso de todas as características dessa linguagem pode ser encontrado em Kessenich, Baldwin e Rost (2006).

2.2.2.2 HLSL

A HLSL foi produzida pela Microsoft para ser utilizada em sua API de desenvolvimento de aplicativos gráficos (Direct3D), bastante utilizada para desenvolvimento de jogos (MSDN, 2009a). Da mesma forma que o GLSL, possui uma sintaxe muito parecida ao C e compatível com qualquer hardware gráfico.

A linguagem HLSL possui 37 palavras reservadas e mais 70 palavras chaves. Tanto as palavras chaves como as reservadas não podem ser utilizadas como nome de identificadores (MSDN, 2009b). As variáveis podem ser divididas em vários grupos que serão descritos abaixo.

O grupo de `buffer` é utilizado para criar um `array` de tipos escalares. Geralmente utilizado em programas de geometria (*geometry shaders*), que é disponível somente na versão 10 ou superior do Direct3D. A leitura de um valor do `buffer` deve ser realizada através da chamada da função `Load`, passando como parâmetro um número inteiro referente ao índice do *array* que deseja ler (MSDN, 2009b).

Outro grupo de variáveis é o `escalar` e nesse grupo estão inclusos os tipos `bool`, `int`, `uint`, `half`, `float` e `double`. O tipo `bool` pode armazenar dados de verdadeiro ou falso somente. Os números inteiros são representados pelos tipos `int` e `uint`. Ambos possuem 32 *bits* de tamanho, sendo que o tipo `int` pode armazenar números negativos e `uint` não permite sinal. Nos números de pontos flutuantes constam os tipos `half`, `float` e `double`. O primeiro não é compatível com todas as versões da linguagem, mas já é suportado pelo Direct3D 10 e possui precisão de 16 *bits*. O segundo possui tamanho de 32 *bits*. Já no último é possível armazenar números de 64 *bits* de tamanho, mas somente é compatível com o Direct3D 11 (MSDN, 2009b).

Há um grupo para `vetores` que são tipos de variáveis escalares e podem ter de 1 a 4 componentes. A declaração para esses `vetores` é igual à linguagem GLSL, porém também é

possível declará-los utilizando a palavra reservada `vector`. A palavra `vector` deve ser seguida dos símbolos de `<` e `>`, sendo que entre esses símbolos deve-se colocar o tipo escalar a qual se refere o vetor seguido pela vírgula e a quantidade de componentes (`vector <float, 4> a`) (MSDN, 2009b).

Como na linguagem GLSL existem um grupo de matrizes que são vetores com mais de 1 posição. Seu tamanho é limitado em 16 componentes, sendo que é permitido o máximo de 4 linhas e 4 colunas. Pode ser declarado igual à linguagem GLSL, ou então parecido com a declaração do vetor. Sendo a segunda opção, deve-se substituir a palavra `vector` por `matrix` e adicionar um elemento a mais dentro dos símbolos. Por exemplo, pode-se declarar uma matriz de números inteiros contendo 4 colunas e 4 linhas da seguinte maneira: `matrix <int, 4, 4> a` (MSDN, 2009b).

Também existe um grupo para determinar as propriedades de uma textura denominada `sampler`. Nesse grupo encontram-se os tipos `sampler`, `sampler1D`, `sampler2D`, `sampler3D`, `samplerCUBE`, `sampler_state`, `SamplerState` e `SamplerComparisonState`. O último tipo mencionado somente é suportado pela versão 10 ou superior do Direct3D. Na declaração de uma variável desse grupo também é possível informar quais serão os estados e valores para a textura. A sintaxe de declaração de variáveis para este tipo é diferente da versão 9 para a 10 do Direct3D. A lista completa dos estados como a sintaxe para ambas as versões da API gráfica da Microsoft podem ser encontrados em Msdn (2009b).

Além das propriedades da textura, há um grupo para os tipos que são utilizados para acessar as texturas. Os seguintes tipos são suportados: `texture`, `texture1D`, `texture1Darray`, `texture2D`, `texture2Darray`, `texture3D` e `textureCube`. Para utilizar uma textura é necessário seguir 3 passos. O primeiro passo é declarar uma variável para a textura. O próximo passo é declarar e inicializar um objeto do tipo `sampler`. Por último realizar a chamada de uma função para trabalhar com operações na textura. As funções para textura são utilizadas de formas diferentes dependendo da versão do Direct3D (MSDN, 2009b).

Pode-se citar ainda um grupo para os *shaders*, cuja função dos tipos desse grupo é criar variáveis para executar um efeito. Na versão 9 do Direct3D são encontrados os tipos `VertexShader` e `PixelShader`. Já na versão 10 a mesma tarefa é realizada através da chamada das funções `SetPixelShader`, `SetGeometryShader` e `SetVertexShader`. Para ambas as versões é necessário informar um modelo de *shader* e a função que será executada quando o *shader* for ativado. O modelo de *shader* define quais serão as capacidades e

restrições da linguagem (MSDN, 2009b).

Existe também o grupo para as estrutura de dados. Sua declaração é igual à linguagem GLSL. A única parte que difere é um parâmetro opcional denominado modificador, que deve ser colocado antes de cada tipo dentro da estrutura. Os modificadores têm efeito apenas no estágio de *pixel shader* e foram inseridos a partir do modelo quatro do *shader*. Existem cinco modificadores: `linear`, `centroid`, `nointerpolation`, `noperspective` e `sample`. O primeiro é considerado o valor padrão e realiza interpolação entre os valores de entradas. O segundo realiza interpolação entre amostras que estão dentro de uma área coberta do pixel. Deve ser combinado com `linear` ou `noperspective`. O terceiro modificador determina a não realização da interpolação. Os dois últimos não realizam correção de perspectiva durante a interpolação, e não podem ser utilizados com nenhum outro modificador (MSDN, 2009b).

Por último podemos citar o grupo de definições, que são outros nomes atribuídos pelo usuário aos tipos de variáveis. Deve ser declarado utilizando a palavra reservada `typedef` seguida ou não pela palavra reservada `const`. Após deve ser colocado um dos tipos citados acima seguidos pelo nome que se deseja atribuir (MSDN, 2009b).

Existem outros aspectos que devem ser explanados sobre a sintaxe da declaração de variáveis na linguagem HLSL. Primeiramente é possível colocar modificadores de classes de armazenamento, o qual não é obrigatório. São eles: `extern`, `nointerpolation`, `shared`, `static`, `uniform` e `volatile`. O primeiro modificador marca a variável como uma entrada externa para o *shader*. É considerado como padrão para todas as variáveis globais declaradas e não pode ser combinada com `static`. O segundo modificador mencionado não realiza a interpolação da saída de um *vertex shader*. Já o terceiro marca a variável para ser compartilhada entre efeitos. O modificador `static` define que quando a variável é inicializada, o valor dela persiste entre as chamadas de funções. O penúltimo modificador define que o valor da variável será constante durante toda a execução de um *shader*. O último é o inverso do anterior, define que a variável mudará frequentemente. Esse modificador pode ser aplicado somente a variáveis locais. Após os modificadores de classe é possível colocar a palavra reservada `const`, que faz com que o valor da variável não possa ser trocado pelo *shader*. Todas as variáveis globais por padrão já são consideradas como constantes (MSDN, 2009b).

Para variáveis globais e parâmetros passados em um *shader* há ainda uma estrutura denominada semântica que precisa ser especificada. A semântica é utilizada para informar a GPU qual o objetivo da variável entre os estágios de um *shader*. É obrigatório o seu uso nas

variáveis de saída de um *vertex* e *pixel shader*. As semânticas para parâmetros de entrada em um *vertex shader* são as seguintes: `BINORMAL`, `BLENDINDICES`, `BLENDWEIGHT`, `COLOR`, `NORMAL`, `POSITION`, `PSIZE`, `TANGENT` e `TEXCOORD`. Como saída pode ser utilizado `COLOR`, `FOG`, `POSITION`, `PSIZE`, `TESSFACTOR` e `TEXCOORD`. Já para um *pixel shader* podem ser utilizadas as seguintes semânticas de entrada: `COLOR`, `TEXCOORD`, `VFACE` e `VPOS`. Para saída somente duas semânticas são suportadas, `COLOR` e `DEPTH`, sendo que a primeira é obrigatório aparecer em um *pixel shader* (MSDN, 2009b).

A linguagem HLSL também suporta declarações para fluxo de controle. As declarações disponíveis são: `break`, `continue`, `discard`, `do`, `for`, `if`, `switch` e `while`. Todos os comandos mencionados funcionam da mesma maneira da linguagem GLSL, e o comando `switch` têm sintaxe e funcionamento igual à linguagem C/C++ (MSDN, 2009b).

Há a possibilidade de declarar funções na linguagem, sendo que o *geometry shader* é uma função, mas com uma sintaxe um pouco diferente. Nas funções também é possível colocar modificadores, porém esses são opcionais. O modificador `inline` cria uma cópia do corpo da função para cada chamada dela (igual ao usado em C/C++). É o modificador padrão para todas as funções declaradas. Já o modificador `target` define a plataforma para a qual a função foi autora. Após a lista de parâmetros é possível ainda colocar uma semântica, que determina a saída para funções de *shader*. As funções podem ser sobrecarregadas mudando o nome, tipo de retorno, ou os parâmetros. Sobre os parâmetros existem os modificadores de entrada `in`, `out`, `inout` e `uniform`. Esses funcionam da mesma forma que na linguagem GLSL (MSDN, 2009b).

A linguagem oferece várias funções predefinidas que podem ajudar na implementação dos *shaders*. Uma lista completa das funções disponíveis, detalhes mais completos da linguagem HLSL como também as diferenças de sintaxe encontradas entre o Direct3D versão 9 e suas sucessoras, podem ser encontradas em Msdn (2009b).

2.2.2.3 Cg

Muito semelhante ao HLSL, esta linguagem foi desenvolvida pela parceria da NVidia com a Microsoft. Diferente das linguagens acima citadas, ela torna-se compatível com qualquer uma das APIs gráficas já mencionadas (NVIDIA, 2009). Como a sintaxe dessa linguagem é muito semelhante ao HLSL, será descrito abaixo somente as diferenças entre

elas.

A linguagem Cg possui no total 100 palavras chaves, sendo que 14 dessas palavras são sensíveis a caixa alta (NVIDIA, 2009). Em Cg o grupo `sampler` possui alguns tipos diferentes da linguagem HLSL. Nesse grupo podemos mencionar os tipos `sampler`, `sampler1D`, `sampler2D`, `sampler3D`, `samplerCUBE` e `samplerRECT`. O último tipo de variável mencionado não é suportado pelos perfis do DirectX (NVIDIA, 2005).

As estruturas na linguagem Cg podem conter funções membros. Essas funções podem acessar seus parâmetros ou as variáveis membros da estrutura onde ela foi declarada. Podem ser chamadas colocando o nome da estrutura, seguindo por um ponto e o nome da função (NVIDIA, 2005).

Algumas outras diferenças relacionadas aos grupos de tipos de variáveis também existem. Primeiramente não existe a possibilidade de criar um `buffer`. A palavra reservada `double` não é suportada pelos perfis atuais. A declaração de vetores e matrizes através das palavras reservadas `vector` e `matrix` também não é suportada (NVIDIA, 2005).

Existe ainda a possibilidade de declarar uma `interface`, disponíveis em outras linguagens como Java e C#. Sua declaração é parecida com a de uma estrutura, porém uma `interface` pode conter somente protótipos de funções. Opcionalmente uma estrutura pode implementar uma `interface`, sendo que para isto deve-se acrescentar ":" após o nome da estrutura seguindo pelo nome da `interface`. Os métodos de uma `interface` devem ser definidos dentro da estrutura que à esta implementando (NVIDIA, 2005).

Referente aos comandos de controle de fluxo, na linguagem Cg as palavras reservadas `switch`, `case`, `default`, `goto`, `break` e `continue` não são suportadas por nenhum perfil atual. Como na linguagem HLSL (porém não mencionado em sua seção), existe a possibilidade de criar efeitos através de técnicas. Pode-se utilizar técnicas para realizar efeitos para os diferentes perfis, tornando assim o efeito compatível com GPUs mais antigas. Para declarar uma técnica deve-se utilizar a palavra reservada `technique`. Cada técnica pode conter um ou mais passos que são declarados pela palavra chave `pass`. Cada passo pode conter estados para a renderização do efeito, como também os programas de *shaders* que irão executá-lo. Para os programas de *shaders* deve-se informar o perfil à qual se atribui o passo como também a função que irá executá-lo (NVIDIA, 2005).

Como em todas as outras linguagens citadas acima, a linguagem Cg oferece diversas funções pré-definidas para ajudar na implementação dos *shaders*. A lista completa das funções disponíveis, exemplos de *shaders*, funções utilizadas pela API para realizar a

comunicação com a GPU e mais detalhes da linguagem podem ser encontrados em Nvidia (2005).

2.2.2.4 Outras linguagens e especificações

Existe ainda uma linguagem de baixo nível para programação de *shaders* denominada ARB. Foi criada pela OpenGL para padronizar as instruções de GPU que controlam o hardware gráfico (ARB ..., 2008). É suportada por todas as placas gráficas, bem como por todas as APIs gráficas também.

Também pode-se mencionar, de acordo com Göddeke (2008), as linguagens de alto nível chamadas Sh e Brook, que como as anteriores têm uma semelhança ao C/C++, mas que ainda estão em desenvolvimento. Estas linguagens estão sendo desenvolvidas para serem utilizadas em *General Purpose for Graphics Processing Unit* (GPGPU), o qual serial desenvolver aplicações de propósito geral utilizando a GPU.

Também existe uma interface para transporte de dados gráficos denominada *COLLAborative Design Activity* (COLLADA). COLLADA define um esquema de base de dados em *eXtensible Markup Language* (XML) permitindo que aplicações proprietárias 3D troquem conteúdo digital livremente sem perda de informações. Suporta todas as características que aplicações modernas em 3D necessitam. O documento de esquema XML do COLLADA é público e acessível pela internet (COLLADA, 2007).

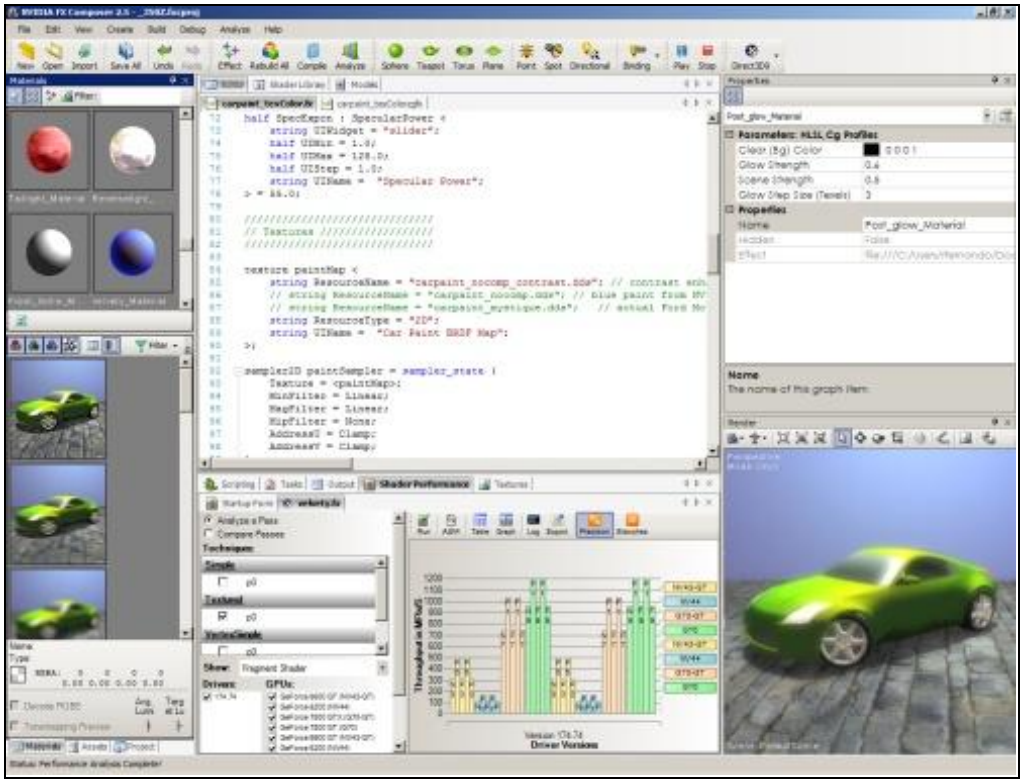
2.2.3 Ambientes de desenvolvimento

Mesmo com linguagens de alto nível, alguns efeitos desejados com a programação de shaders são complexos de visualizar apenas em código. Diante disso, algumas empresas desenvolveram ambientes gráficos para utilização destas linguagens. Algumas características destes ambientes vão além de somente destacarem as palavras chaves da sintaxe, podendo portanto mostrar interativamente os resultados obtidos. Alguns possuem também a possibilidade de estender a ferramenta através de *plugins*, anexando a estes ambientes *debbugers* para visualização dos valores das variáveis (NVIDIA, 2008). Nas seções subseqüentes serão descritas algumas ferramentas com essas características.

2.2.3.1 FX Composer

Ferramenta desenvolvida pela NVidia, foi criada para ser utilizada com a linguagem desenvolvida pela empresa, o Cg, mas possui compatibilidade com o HLSL também. Em sua versão mais atual possui suporte para as versões 9 e 10 da API DirectX e para OpenGL. Existe a possibilidade de integrar na ferramenta um *plugin* para realizar *debugger* dos *shaders* criados. Também possui suporte ao sistema de partículas, possibilidade de controlar remotamente o serviço através do protocolo *Transmission Control Protocol/*Internet *Protocol* (TCP/IP) e ainda realiza a comparação de técnicas de *shader*, mostrando o resultado em tabelas ou gráficos. Oferece ainda suporte total ao COLLADA, podendo acrescentar, excluir mover ou copiar conteúdo em documentos do COLLADA (NVIDIA, 2008).

Na Figura 3 é demonstrada a interface da ferramenta FX Composer. No centro se encontram o editor de *shaders* e uma tela com a comparação de desempenho. À esquerda se encontram os materiais utilizados e a direita a tela de visualização do efeito criado.



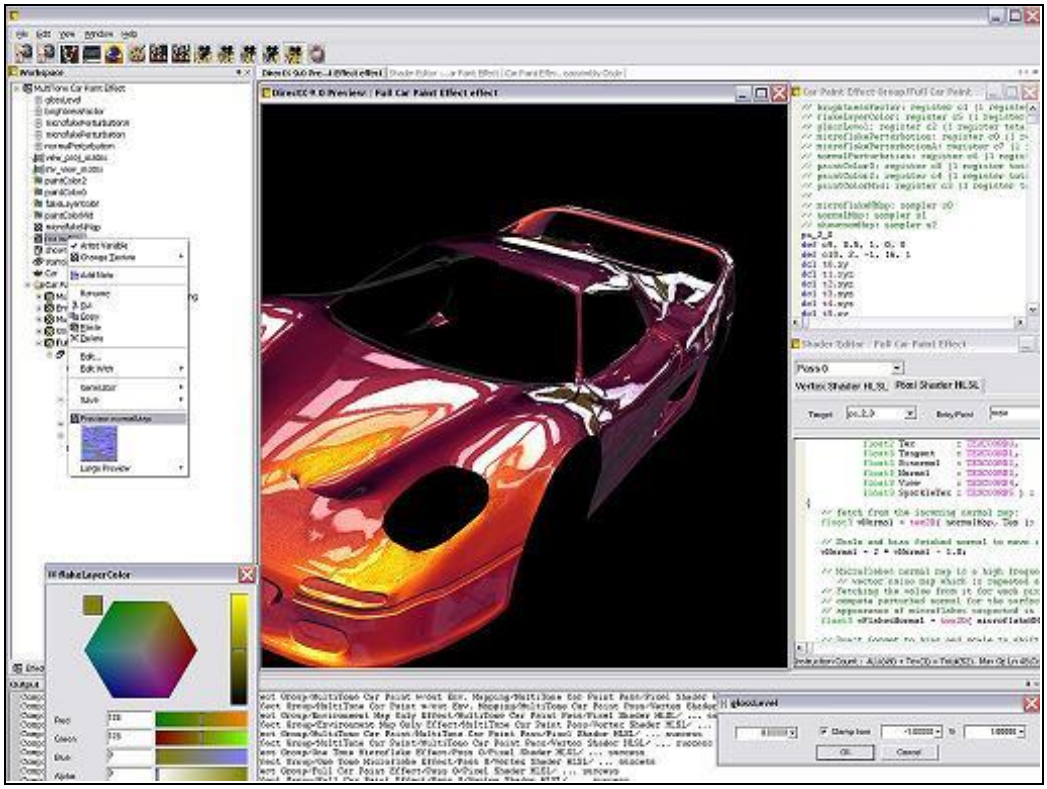
Fonte: NVidia (2008).

Figura 3 – Tela da ferramenta FX Composer

2.2.3.2 RenderMonkey

O RenderMonkey foi desenvolvido pela empresa AMD/ATI, e pode ser utilizado para criar programas de *shader* para as linguagens HLSL e GLSL. Possui um compilador interno facilitando a identificação dos erros durante o processo de desenvolvimento. Incorpora também uma quantidade de recursos para ajudar na depuração e otimização dos *shaders*. Na ferramenta encontra-se ainda uma janela para visualização do efeito que esta sendo criado, fornecendo retorno imediato de qualquer alteração realizada. Oferece também integração com o COLLADA, e possui vários exemplos de efeitos prontos que podem ser usados e alterados gratuitamente (AMD, 2009).

Na Figura 4 é demonstrada a interface da ferramenta RenderMonkey. Ao centro encontra-se a janela de visualização do efeito criado. No lado esquerdo encontra-se uma árvore de projeto contendo todos os arquivos e efeitos disponíveis. Ao lado direito tem-se o editor de código, sendo que o editor mais acima mostra o fonte convertido na linguagem *assembly* e no editor abaixo desse encontra-se o código na linguagem de programação para GPU. Por fim na parte inferior da ferramenta encontra-se a tela contendo as mensagens e erros.



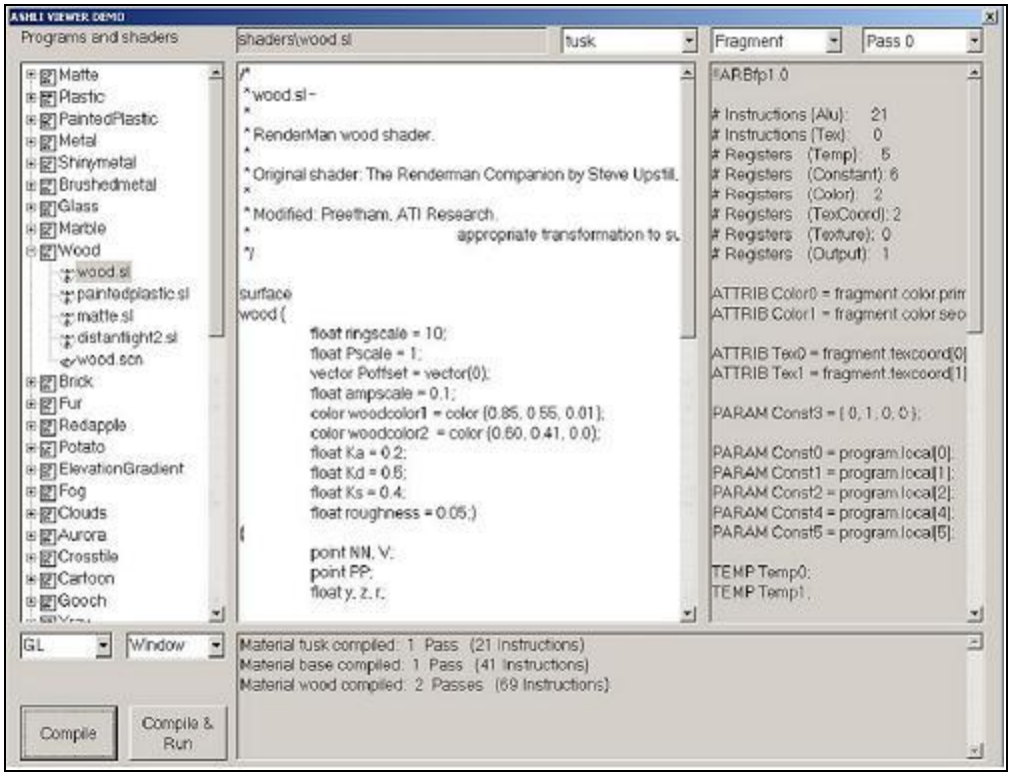
Fonte: AMD (2009).

Figura 4 – Tela do ambiente de desenvolvimento RenderMonkey

2.2.3.3 *Advanced SHading Language Interface* (ASHLI)

Além das IDEs citadas anteriormente, existe uma outra denominada ASHLI, que também foi desenvolvida pela empresa AMD/ATI. É uma API neutra, que pode ser utilizada para programação com HLSL ou GLSL, e possui integração com alguns *RenderMan*®[5]. Possui uma coleção com vários *shaders* prontos destinados para compilação (AMD, 2008).

A Figura 5 mostra a tela da ferramenta ASHLI. Do lado esquerdo da ferramenta encontram-se uma lista de efeitos de *shaders* prontos. No centro é mostrado o código fonte do *shader* selecionado e no lado direito encontra-se o código *assembly* para o fonte. Para visualizar o efeito basta clicar no botão *Compile & Run*.



Fonte: AMD (2008).

Figura 5 – Tela da ferramenta ASHLI mostrando o código fonte de um *shader*

[5] Marca registrada da Pixar. É uma interface para especificação de renderização foto realística (PIXAR, 2008).

2.3 VISUALIZAÇÃO DE DINÂMICA DE FLUÍDOS

Os fluídos estão em todos os lugares; é a água de um rio se movimentando, a fumaça saindo do cigarro, etc. (FERNANDO, 2004). Tudo isso pode ser simulado em softwares gráficos interativos, geralmente utilizados para estudar a aerodinâmica e hidrodinâmica (FLUID ..., 2008).

O que a dinâmica de fluídos estuda é o movimento de um fluído o qual é definido como escoamento. Podem ser utilizados vários métodos numéricos para simulação de grandes escalas de escoamentos, tais como os métodos de diferenças finitas, volumes finitos, elementos finitos e métodos espectrais. Os escoamentos podem ser representados de forma bidimensional, tridimensional, transientes, incompressíveis, isotérmicos e turbulentos (sendo este último representado obrigatoriamente em 3D). Além da simulação de grandes escalas de escoamentos, pode ser utilizada a simulação numérica direta. Esta apresenta resultados comparáveis a qualquer experimento de qualidade realizado, porém necessita de um processamento muito alto (OLIVEIRA JR., 2006).

É possível prever outras propriedades na dinâmica de fluidos computacional além do comportamento do fluido, como estimar a transferência de calor, massa, mudança de fase, reação química e movimento mecânico. A dinâmica dos fluídos computacional pode ser dividida em 3 etapas: pré-processamento, solução e pós-processamento. O pré-processamento consiste na geração da malha utilizando geradores próprios como CAD ou ICEM. No próximo passo é realizada a solução das equações de conservação, seguindo para o último passo onde os dados são representados de forma gráfica (SOUZA, 2008).

Um problema para realizar a simulação de dinâmica de fluídos é a quantidade enorme de dados gerados, que pode chegar na ordem de dezenas de *GygaBytes* (GB). A análise destes dados depende do desenvolvimento de técnicas específicas capazes de filtrar e realçar apenas informações realmente relevantes (GIRALDI; FEIJÓO, 2003). Esta análise geralmente é efetuada através de algoritmos, que possuem uma quantidade grande de cálculos que precisam ser efetuados.

Pela capacidade de processamento paralelo da GPU, pela grande massa de dados e cálculos necessários, utilizar placas gráficas para esta simulação é o mais indicado. Comparações feitas entre o processamento da CPU e da GPU, mostra que o hardware gráfico chegou a ser 6 vezes mais rápido (FERNANDO, 2004).

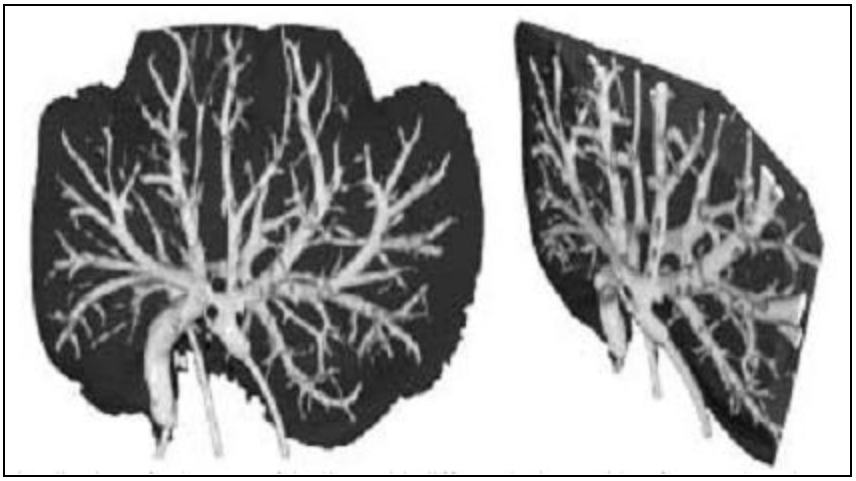
2.4 TRABALHOS CORRELATOS

Nesta seção são descritos alguns trabalhos que apresentam algumas das funcionalidades apresentadas ou que serão desenvolvidas de acordo com este projeto. O primeiro trabalho descreve uma técnica para recoloração de imagens para dicromatas (classe de indivíduos daltônicos). Nesse foi desenvolvido uma técnica para realizar o processamento das imagens em tempo real, reforçando a cor do contraste e tentando preservar as cores originais. A técnica desenvolvida foi implementada em CPU e GPU, sendo que a técnica implementada em GPU chega a ser de duas a três vezes mais rápida que em CPU. Na Figura 6 pode ser visualizado o resultado obtido pela técnica. A figura à esquerda é a foto original da flor, enquanto a figura à direita mostra a imagem depois de processada pela técnica desenvolvida (KUHN; OLIVEIRA; FERNANDES, 2008).

Fonte: Kuhn, Oliveira e Fernandes (2008, p. 1748).

Figura 6 – Diferença entre a imagem original e a imagem com contraste reforçado

O segundo trabalho é um projeto de visualização e manipulação de dados geométricos, disponível na página da Universidade Federal do Rio Grande do Sul (UFRGS), que utiliza a GPU para criar uma ferramenta capaz de gerar visualização em três dimensões dos órgãos humanos (Figura 7). Além da visualização, este projeto disponibiliza três ferramentas: uma capaz de realizar cortes no órgão visualizado, outra que permite apagar partes não relevantes da imagem e por fim uma ferramenta que possibilita ao usuário eliminar camadas de uma imagem 3D. Todas estas ferramentas são interativas e manipulam os dados em tempo real. Para tanto, se faz necessário a utilização da GPU neste projeto para poder resolver a grande quantidade de dados gerados pela representação 3D dos órgãos, que precisam ser representados de forma gráfica e em tempo real (DIETRICH et al., 2004).



Fonte: Dietrich et al. (2004, p. 8).

Figura 7 – Visualização de um pulmão escaneado, visto de 2 ângulos diferentes

3 DESENVOLVIMENTO

Neste capítulo é apresentado o desenvolvimento, a especificação e a operacionalidade da aplicação, como também os testes e resultados encontrados.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

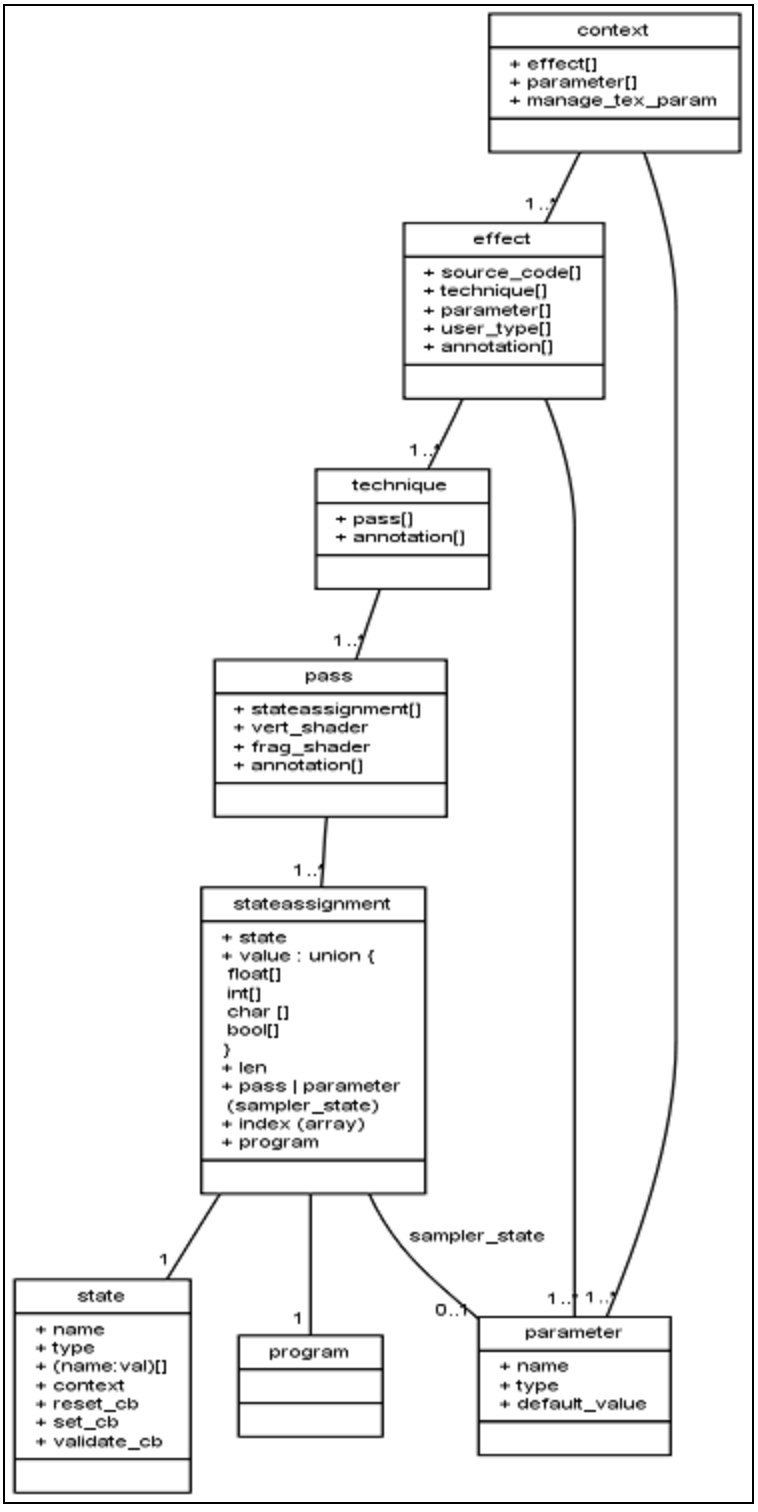
O sistema a ser desenvolvido terá os seguintes requisitos:

a) permitir ao usuário abrir um arquivo contendo as informações que deverão ser processadas pela ferramenta (Requisito Funcional - RF);
b) permitir ao usuário iniciar ou parar as simulações das rotinas de dinâmica de fluídos (RF);
c) permitir ao usuário visualizar as simulações graficamente (RF);
d) ser implementado utilizando a linguagem C++ para construção da ferramenta (Requisito Não-Funcional - RNF);
e) utilizar o ambiente *FX Composer* e a linguagem Cg para programação da GPU (RNF).

3.2 ESPECIFICAÇÃO

Os programas de *shaders* foram especificados através de um diagrama encontrado em um fórum da NVidia (NVIDIA, 2007), sendo que sua representação é muito parecida com um diagrama de classes. Foi utilizada a ferramenta Enterprise Architect para elaboração desse diagrama.

O diagrama é dividido em 6 níveis, sendo que todos eles são obrigatórios de serem representados caso os fontes sejam elaborados para se usar efeitos. Como pode ser visualizado na Figura 8 todo programa em *shader* é obrigado a possuir um contexto e todo contexto possui parâmetros e efeitos ligados a ele. Uma breve explicação do funcionamento dos efeitos foi descrita na seção 2.2.2.3.



Fonte: NVIDIA (2007).

Figura 8 – Diagrama de objetos para a linguagem Cg

As técnicas implementadas neste trabalho na linguagem Cg não foram elaboradas como efeitos, contudo é possível modificar a estrutura dos códigos para que sejam representados desta maneira. Sendo assim não é possível utilizar o diagrama totalmente para representar essas técnicas. Dessa forma serão utilizadas as caixas `context`, `effect` e `parameter` na representação das técnicas.

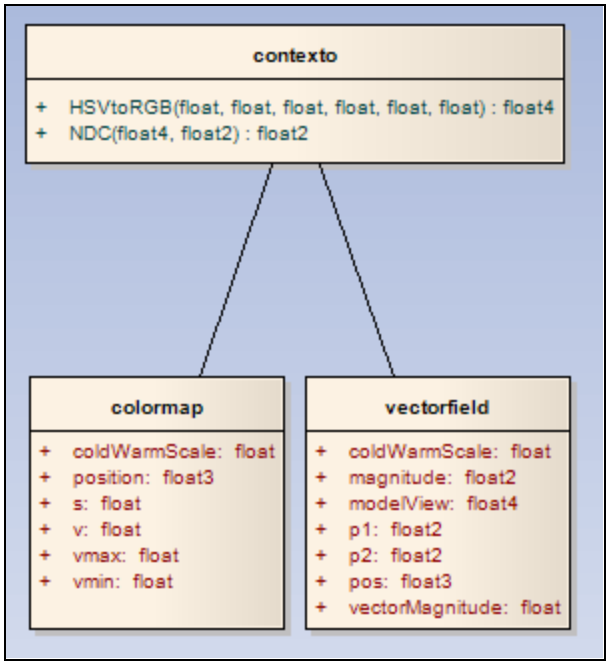


Figura 9 – Diagrama de objetos das técnicas implementadas

Como pode ser observado na Figura 9 foi utilizado um contexto e duas caixas para representar os efeitos implementados. Os parâmetros não foram representados separadamente tendo uma caixa para cada parâmetro, mas foram colocadas dentro da caixa de sua respectiva técnica. Podem ser visualizadas ainda na Figura 9 duas funções que foram adicionadas no contexto. Originalmente elas não aparecem no diagrama, contudo essas duas funções são utilizadas por ambas as técnicas. Por serem implementadas dentro de um mesmo contexto em que as duas técnicas foram desenvolvidas, e por serem indispensáveis para as mesmas elas foram representadas na caixa de contexto.

3.3 IMPLEMENTAÇÃO

A seguir serão mostradas as principais técnicas e ferramentas utilizadas como também a operacionalidade da implementação. Na seção 3.3.1 será descrito as implementações e alterações elaboradas sobre o trabalho original utilizado como base (PIVA; GOMES; REIS, 2008).

3.3.1 Alterações do trabalho

O trabalho atual não foi implementado do zero, foi utilizado como base um trabalho sobre dinâmica de fluídos (PIVA; GOMES; REIS, 2008). Dessa forma toda a parte gráfica de iteração com o usuário, como o desenvolvimento de todas as técnicas de simulação de dinâmica de fluídos foi desenvolvida somente na aplicação tomada como base (PIVA; GOMES; REIS, 2008). Assim o atual trabalho detinha-se em apenas realizar a conversão das técnicas implementadas em CPU para rodarem em GPU.

As principais alterações realizadas sobre (PIVA; GOMES; REIS, 2008), são as seguintes: conversão das técnicas de `colormap` e `vectorfield` para GPU, criação de rotina extra (rotina para normalização de coordenadas) em GPU para realizar a conversão, alteração de alguns códigos fontes para incluir instruções de comunicação com o hardware gráfico, inclusão de instruções para realizar os testes de desempenho de processamento e criação de um programa para comparação dos resultados gráficos gerados (sendo que este programa foi desenvolvido do zero, utilizando somente as imagens geradas pelas representações gráficas).

3.3.2 Técnicas e ferramentas utilizadas

O ambiente utilizado para a implementação do projeto foi o Microsoft Visual Studio C++ 2005. Também foi utilizado a IDE *FX Composer* para criar os *shaders* e verificar os erros de sintaxe. Foi escolhida esta ferramenta por ser desenvolvida pela mesma empresa que criou a linguagem Cg e por oferecer facilidades para encontrar erros de programação.

Foram desenvolvidas duas técnicas principais e mais duas funções de apoio em GPU. Uma das técnicas será descrita na seção 3.3.2.1 e trata da visualização do `colormap`. A segunda técnica desenvolvida é o `vectorfield` e será descrita na seção 3.3.2.2. As funções de apoio à essas técnicas serão descritas na seção 3.3.2.3. Por fim, será mostrado na seção 3.3.2.4 como é realizada a comunicação da API utilizada e a GPU.

3.3.2.1 Implementação do algoritmo `colormap`

Inicialmente foi criada uma `estrutura` para armazenar os dados de saída da função

como mostrado no Quadro 1. Essa estrutura é composta por 2 elementos que definem a cor e a posição do vértice que esta sendo processado pela GPU. É utilizada por todas as técnicas implementadas.

```
1  struct Ev_Output {
2
3    float2 position : POSITION;
4    float4 color    : COLOR;
5
6  };
```

Quadro 1 – Estrutura de saída das técnicas desenvolvidas

A principal função do `colormap` possui poucas linhas, pois utiliza de outras funções para realizar todo o processamento. Conforme pode ser visualizado no Quadro 2, entre as linhas 53 à 59 é realizado a declaração dos parâmetros utilizados pela função. Desses parâmetros os cinco primeiros são utilizados unicamente para o cálculo da cor do vértice processado. Na linha 58 encontra-se o parâmetro que armazena as dimensões de tela, utilizada pela API para definir os pontos visíveis na tela e para aplicar o efeito de *zoom*. O último parâmetro da função localizado na linha 59 fornece a posição `x` e `y` do vértice. A terceira dimensão desse parâmetro não é utilizado para definir a posição do obejto em um espaço tri-dimensional, mas para armazenar um valor utilizado também no cálculo da cor.

Na linha 61 é declarada uma variável contendo a estrutura de saída da função. Em seguida na linha 63 é realizado o cálculo para normalização do espaço de tela. Já na linha 65 os parâmetros da técnica são enviados para a função `HSVtoRGB` para calcular a nova cor do vértice (as funções `NDC` e `HSVtoRGB` são explicadas na seção 3.3.2.2). Por fim a técnica retorna os dados processados para serem visualizados na tela.

```
52  //função principal para o colormap
53  Ev_Output colormap (uniform float s,
54                      uniform float v,
55                      uniform float vmax,
56                      uniform float vmin,
57                      uniform float coldWarmScale ,
58                      uniform float4 modelView,
59                      float3 position: POSITION) {
60
61  Ev_Output OUT;
62
63  OUT.position = NDC(position.xy, modelView);
64  //a terceira dimensão é utilizada para armazenar o vertexvalue
65  OUT.color = HSVtoRGB(s, v, vmax, vmin, coldWarmScale, position.z);
66
67  return OUT;
68  }
```

Quadro 2 – Função principal da técnica `colormap`

3.3.2.2 Implementação do algoritmo `vectorfield`

Igualmente ao método anterior essa técnica possui alguns parâmetros que são utilizados para calcular a cor e posição do vértice a ser processado. Esses parâmetros encontram-se da linha 73 até a linha 77. Já os parâmetros localizados na linha 71 e 72 são utilizados nos cálculos da técnica para encontrar a posição dos outros vértices desejados. Nessa técnica a dimensão `z` do parâmetro `pos` não será utilizada para cálculo da cor, mas sim para definir quais serão os cálculos que deverão ser executados.

Como pode ser visualizado no Quadro 3 na linha 82, se a terceira dimensão do vértice for igual a 1, as instruções realizadas serão as mesmas da técnica `colormap`. A única diferença nessas instruções encontra-se na linha 84 nos parâmetros enviados para a função `HSVtoRGB`. Os dois primeiros parâmetros são constantes e não mais parâmetros da própria técnica. Nessa técnica também é definida outra constante localizada na linha 80 e denominada `t`, utilizada nos cálculos para encontrar a posição do vértice.

```
71    Ev_Output vectorfield (uniform float2 p1,
72                           uniform float2 p2,
73                           uniform float4 modelView,
74                           uniform float vectorMagnitude,
75                           uniform float2 magnitude,
76                           uniform float coldWarmScale,
77                           float3 pos : POSITION) {
78
79        Ev_Output OUT;
80        const float t = 0.8;
81
82        if (pos.z == 1){
83            OUT.position = NDC(pos.xy, modelView);
84            OUT.color = HSVtoRGB(1.0, 1.0, magnitude.y, magnitude.x,
85                                 coldWarmScale, vectorMagnitude);
86        }
87
88        ...
89
133       return OUT;
134   }
```

Quadro 3 – Execução dos cálculos da primeira condição do método `2D Vector Field`

Contudo se a terceira dimensão do vértice for igual a 2 como é mostrado no Quadro 4 na linha 86, então a técnica irá realizar uma série de cálculos. Da linha 87 até a linha 106 são realizados os cálculos para encontrar a posição do vértice. O resultado almejado pelos cálculos é encontrar a posição final de uma reta que se inicia na reta principal e estende-se paralelamente a ela, criando a forma de um vetor ao objeto desenhado. A reta principal é

formada pelas posições armazenadas nos parâmetros `p1` e `p2`. Após o fim dos cálculos nas linhas 107 e 108 são executadas as instruções para normalização da posição e para recalcular a cor do vértice processado (Quadro 4).

Agora se a terceira dimensão do parâmetro `pos` for diferente de 1 e 2, então outra condição será executada. Todavia dentro do laço dessa condição os cálculos executados são muito parecidos aos cálculos do Quadro 4. As divergências encontram-se entre as linhas 124 e 126, como é mostrado no Quadro 5. O que torna-se diferente entre os laços é o nome da variável que muda de `palpha` para `pbetha` e a operação matemática executada, alterando a operação de adição para subtração.

```
80        ...
81
82    if (pos.z == 1){
83
84        ...
85
86    } else if (pos.z == 2){
87       //parametric line equation
88       float2 pb;
89       pb.x = (1-t)*p1.x + t*p2.x;
90       pb.y = (1-t)*p1.y + t*p2.y;
91
92       //creates the normal vector
93       float2 originVector = p2.xy - p1;
94       float modulo = sqrt((originVector.x * originVector.x) +
95                           (originVector.y * originVector.y));
96
97       float2 normalVector = float2(-originVector.y, originVector.x);
98       normalVector.x /= modulo;
99       normalVector.y /= modulo;
100
101      float dx = sqrt((p2.x - pb.x) * (p2.x - pb.x) +
102                      (p2.y - pb.y) * (p2.y - pb.y)) / (3.0*0.67);
103
104      float2 palpha;
105      palpha.x = pb.x + dx*normalVector.x;
106      palpha.y = pb.y + dx*normalVector.y;
107      OUT.position = NDC(palpha, modelView);
108      OUT.color = HSVtoRGB(1.0, 1.0, magnitude.y, magnitude.x,
109                           coldWarmScale, vectorMagnitude);
110   }
111       ...
```

Quadro 4 – Cálculos da segunda condição do método `2D Vector Field`

O Quadro 5 mostra as diferenças encontradas da segunda para a última condição da técnica `2D Vector Field`, onde o nome da variável e a operação matemática empregada mudam.

```
123             ...
124             float2 pbetha;
125             pbetha.x = pb.x - dx*normalVector.x;
126             pbetha.y = pb.y - dx*normalVector.y;
127             ...
```

Quadro 5 – Diferença entre os cálculos da segunda condição para a última

3.3.2.3 Implementação das funções de apoio

Como muitas das instruções eram iguais em todas as técnicas implementadas, foram desenvolvidas duas funções para realizar o trabalho. A primeira função de apoio implementada calcula a cor de saída do vértice e pode ser visto no Quadro 6. Os parâmetros dessa função são definidos pelo programa e enviados para o método principal da técnica, para depois serem utilizados por essa função.

```
22   //realiza a conversão das cores
23   float4 HSVtoRGB(float s, float v, float vmax,
24                   float vmin, float coldWarmScale, float m) {
25     float4 OUT;
26
27     if (s == 0) {
28        OUT = float4(v, v, v, 1.0);
29     }  else {
30        float h = (m - vmin) / (vmax - vmin);
31        h *= coldWarmScale;
32        h -= coldWarmScale;
33
34        float var_h = abs(h) * 6;
35        float var_i = floor(var_h);
36        float var_1 = v * (1 - s);
37        float var_2 = v * (1 - s * (var_h - var_i));
38        float var_3 = v * (1 - s * (1 - (var_h - var_i)));
39
40        if      (var_i == 0) {OUT = float4(v    , var_3, var_1, 1.0);}
41        else if (var_i == 1) {OUT = float4(var_2, v    , var_1, 1.0);}
42        else if (var_i == 2) {OUT = float4(var_1, v    , var_3, 1.0);}
43        else if (var_i == 3) {OUT = float4(var_1, var_2, v    , 1.0);}
44        else if (var_i == 4) {OUT = float4(var_3, var_1, v    , 1.0);}
45        else                 {OUT = float4(v    , var_1, var_2, 1.0);}
46     }
47
48      return OUT;
49   }
```

Quadro 6 – Função que calcula a cor do vértice

Já a segunda função desenvolvida é responsável por realizar os cálculos de normalização das coordenadas enviadas para a GPU e pode ser vista no Quadro 7. Para esta função é necessário enviar as posições `x` e `y` do vértice e um vetor contendo o tamanho do

view na API utilizada.

```
 8   //realiza a normalização das coordenadas para o view da GPU
 9  float2 NDC (float2 position, float4 modelView) {
10      //modelView = left, rigth, bottom, top
11      float2 OUT;
12      float deltaOx = modelView.y - modelView.x;
13      float deltaOy = modelView.w - modelView.z;
14
15      //na GPU as coordenadas vão de -1 até 1
16      OUT.x = ((position.x - modelView.x) * (2/deltaOx)) - 1;
17      OUT.y = ((position.y - modelView.z) * (2/deltaOy)) - 1;
18
19      return OUT;
20  }
```

Quadro 7 – Função para calcular a normalização das coordenadas dos vértices

3.3.2.4 Comunicação da API com a GPU

Para poder utilizar a GPU, primeiro é necessário criar um programa de *shader*. Como pode ser visualizado no Quadro 8, para criar o programa é necessário declarar três variáveis para controle e também as variáveis que serão utilizadas para enviar os parâmetros para a GPU. Na linha 62 é declarada uma variável de contexto, que pode ser utilizada por qualquer programa de *shader*. A variável declarada na linha 63 armazena o programa criado e carregado. Já a variável da linha 64 define o perfil que será usado no programa. Atribuir um perfil menor que o programa de *shader* necessita fará com que esse não seja iniciado, porém nenhum erro será mostrado. A partir da linha 67 são declaradas as variáveis utilizadas para a GPU.

```
61      //GPU control variables
62      CGcontext Context;
63      CGprogram VertexProgram;
64      CGprofile VertexProfile;
65
66      //variables parameters from shader
67      CGparameter paramVMax;
68      CGparameter paramVMin;
69      CGparameter paramColdWarmScale;
70      CGparameter paramS;
71      CGparameter paramV;
```

Quadro 8 – Variáveis necessárias para criar um programa de *shader* para a técnica `colormap`

Tendo as variáveis declaradas é possível criar o programa para o *shader*. Como mostrado no Quadro 9 nas linhas 44 e 45, primeiramente procura-se o melhor perfil para a placa de vídeo instalada no computador. Como já mencionado anteriormente, se o *shader* for

criado com um perfil superior ao suportado pela placa de vídeo, a visualização não será mostrada e nenhum erro aparecerá. Após definir o perfil cria-se o contexto, visível na linha 47. Entre as linhas 50 à 53 é utilizada a instrução que cria o programa de *shader*. O primeiro parâmetro dessa função é o contexto que foi criado. Os próximos parâmetros que devem ser definidos é o tipo de arquivo do código fonte (texto ou binário) e o nome do arquivo, como pode ser visualizado na linha 51. O próximo parâmetro é o perfil definido para placa de vídeo utilizada (linha 52). Por fim, envia-se o nome da técnica que o programa irá executar. Se o programa for criado com sucesso então ele é carregado e inicializa-se as variáveis que enviarão os valores para os parâmetros do *shader*, mostrado nas linhas 58, 59 e 60. Cada parâmetro em um programa de *shader* precisa ser referenciado por uma variável local. Somente dessa forma é possível enviar os valores para os parâmetros dos *shaders*.

```
42      ...
43      /* Escolhe o profile mais adequado para o shader*/
44      VertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
45      cgGLSetOptimalOptions(VertexProfile);
46      /* Cria um contexto que será usado por todos os shaders */
47      Context = cgCreateContext();
48
49      //create the vertex shader program
50      VertexProgram = cgCreateProgramFromFile(Context,
51          CG_SOURCE, "colormap.cg",
52          VertexProfile,
53          "vectorfield", NULL);
54
55      if(VertexProgram != NULL)
56      {
57          //load program
58          cgGLLoadProgram(VertexProgram);
59          paramP1 = cgGetNamedParameter(VertexProgram, "p1");
60          paramP2 = cgGetNamedParameter(VertexProgram, "p2");
61      }
62      ...
```

Quadro 9 – Rotina que cria o programa de *shader* e inicializa as variáveis

Com o programa criado e carregado é possível executar a técnica, contudo antes são enviados os valores dos parâmetros para a GPU. Como pode ser visto no Quadro 10 nas linhas 200 à 202, os valores de variáveis locais são enviadas para a GPU. Na linha 239 e 241 o programa de *shader* é inicializado. A linha 245 e 247 enviam o último valor para a GPU. Cada uma dessas duas linhas corresponde a uma execução do programa. As instruções que definem as propriedades de um vértice (cor, posição, normal entre outras) são enviadas para a GPU automaticamente pela API. Por fim na linha 358 a execução do *shader* é desativada.

```
199       ...
200       cgSetParameter2dv(paramP1, p1);
201       cgSetParameter2dv(paramP2, p2);
202       cgSetParameter1f(paramVectorMagnitude, vectorMagnitude);
203   ⊞   ...
239       cgGLBindProgram(VertexProgram);
240       //enable profiles for shader
241       cgGLEnableProfile(VertexProfile);
242
243       glBegin(GL_LINES);
244           glColor4f(0.2, 0.2, 0.2, 1.0);
245           glVertex3f(p1[0],p1[1], 1);
246
247           glVertex3f(p2[0],p2[1], 1);
248       glEnd();
249
250   ⊞   ...
358       cgGLDisableProfile(VertexProfile);
359       ...
```

Quadro 10 – Rotina que inicia e termina a execução do *shader*

3.3.3 Operacionalidade da implementação

Nesta seção é apresentada a operacionalidade do sistema e suas principais funções. Mostra como carregar vários arquivos de configuração e como utilizar as várias técnicas de visualização existentes.

A imagem da Figura 10 mostra a tela de carregamento dos arquivos e das técnicas de visualização implementadas. Ao lado esquerdo da imagem encontra-se uma lista de arquivos para serem selecionados. Deve-se primeiramente executar o arquivo `vg.dat, pois` nele se encontram todas as configurações necessárias para rodar a simulação. Em seguida se faz necessário abrir os outros arquivos com extensão dat (`ug`, `vg` e `wg`).

Após os arquivos serem carregados é possível escolher o modo de visualização que se encontra ao centro da tela. Ao escolher a opção `Colormap` irá aparecer uma opção para selecionar um dos arquivos carregados. Cada arquivo mostrará uma visualização diferente. Quando se escolhe a opção `2D VectorField`(Figura 10) é possível selecionar 2 arquivos ao mesmo tempo. Ao escolher arquivos diferentes a visualização também mudará. Por fim é necessário somente pressionar o botão `Load` para carregar a tela de visualização.

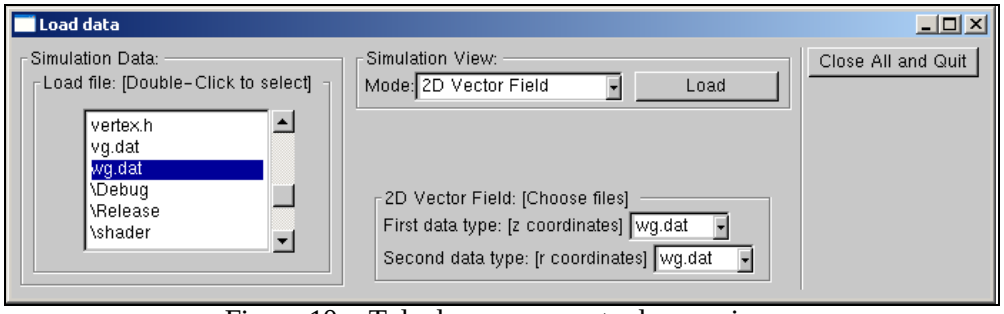


Figura 10 – Tela de carregamento dos arquivos

Ao pressionar o botão para carregar a tela de visualização, duas novas telas serão abertas. Uma delas contém a visualização da técnica e a outra armazena as opções da simulação que podem ser alteradas em tempo de execução. Ao selecionar a técnica `2D Vector Field` duas novas opções são mostradas na parte superior da tela, como pode ser visto na Figura 11. A primeira opção determina o espaço entre cada vetor fazendo com que mais ou menos vetores fiquem visíveis na tela. A segunda opção tem o efeito de escala redimensionando os vetores. Para que as alterações sejam visíveis na tela é necessário pressionar o botão `Update View`.

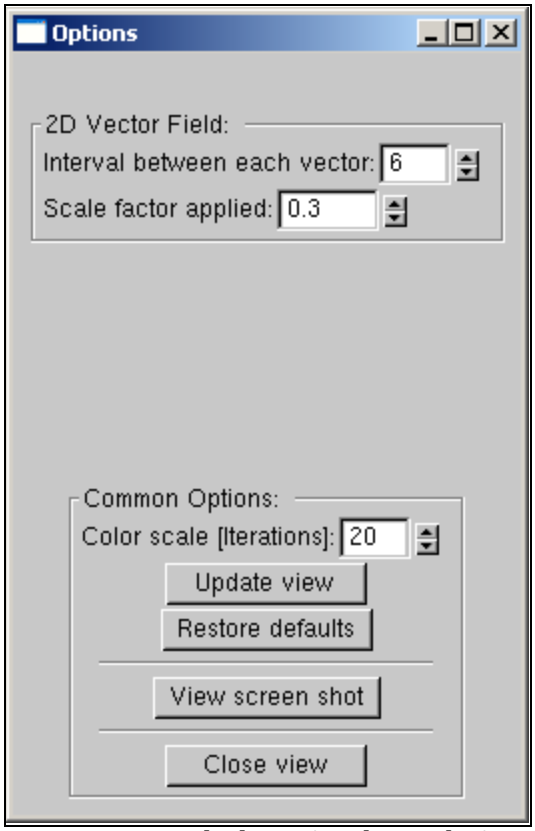


Figura 11 – Tela de opções da simulação

Para a técnica `Colormap` somente as opções do `Common Options` são visíveis. A opção `Color scale` determina a variação de cores que se encontra ao lado esquerdo da tela de visualização (Figura 12). O botão `Update View` aplica as alterações realizadas nas opções e

atualiza a tela de visualização. Já o botão `Restore defaults` restaura as opções para seus valores padrões e atualiza a tela de visualização. O próximo botão tira um *screen shot* da tela e o último botão fecha a tela de visualização e opções, e volta para a tela onde seleciona a técnica de visualização.

Na tela de visualização (Figura 12) também existe interação com o usuário. Pode-se mover o objeto representado e também aplicar o *zoom*. Para mover o objeto basta pressionar e segurar o botão esquerdo do mouse e movê-lo. Já o *zoom* pode ser obtido pressionando e segurando o botão de rolagem do mouse e movendo o mouse para cima ou para baixo. Após realizar as interações na tela é necessário pressionar o botão `Update View` da tela de opções para atualizar a tela.

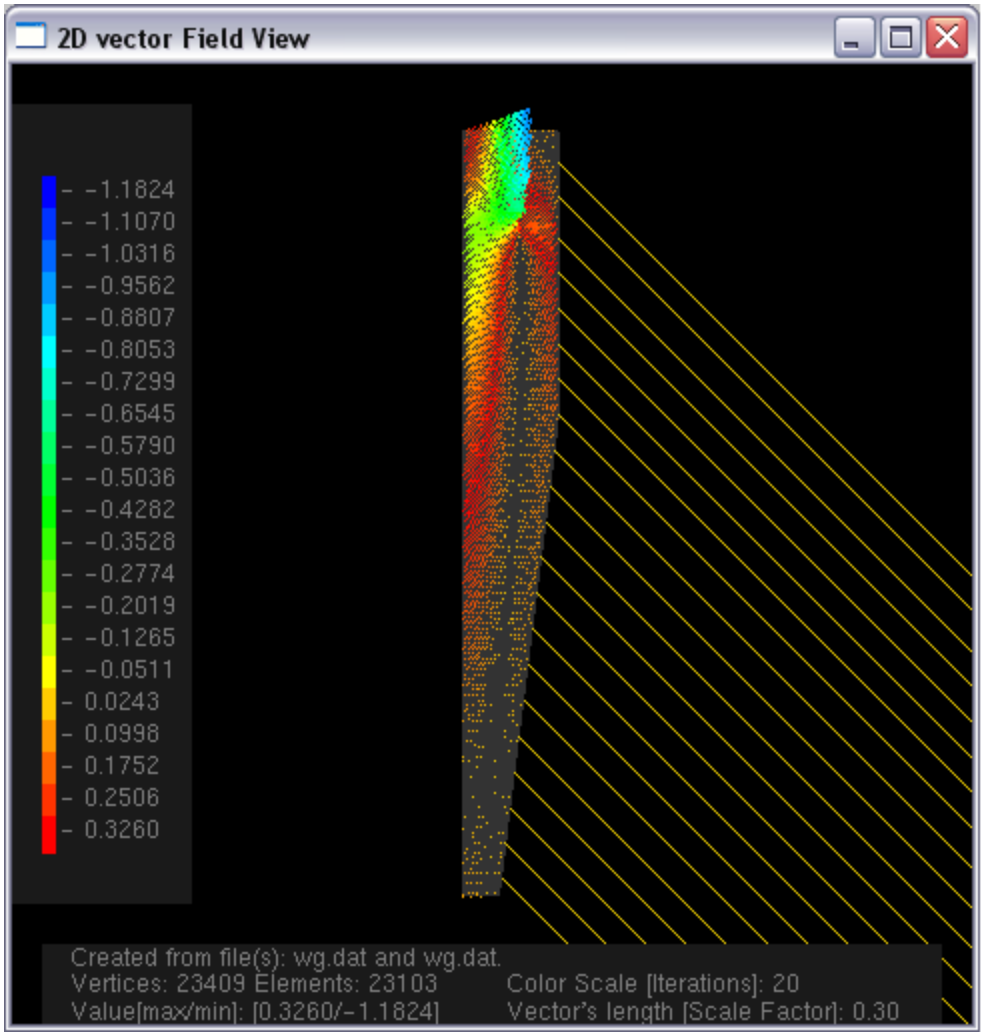


Figura 12 - Tela de visualização da técnica selecionada

3.4 RESULTADOS E DISCUSSÃO

Neste trabalho é possível observar as vantagens e desvantagens da utilização da GPU. Dessa forma na seção 3.4.1 são apresentados as limitações encontradas durante o desenvolvimento do projeto. Na seção 3.4.2 serão mostrados os testes de desempenho e memória realizados na GPU e CPU, a fim de observar os pontos positivos e negativos de cada hardware. Por fim, na seção 3.4.3 será mostrado a representação gráfica de cada técnica e as diferenças encontradas quando executadas em CPU e GPU.

3.4.1 Limitações encontradas

Na implementação dos algoritmos para GPU foi utilizada a placa de vídeo Geforce FX 5200 com 128MB de memória. Utilizando esse hardware gráfico um dos problemas encontrados na implementação do algoritmo `2D Vector Field` (seção 3.3.1.2) foi conseguir armazenar o valor de uma variável em um registrador da GPU, de forma que ficasse lá até o programa ser executado novamente. Mesmo criando variáveis globais de forma estática, elas eram inicializadas sempre que o programa era executado novamente. Então tentou-se buscar os valores calculados na GPU e trazê-los para variáveis locais na CPU, para poderem ser utilizados na próxima etapa da técnica na GPU. Existem funções em Cg que fazem esse trabalho, porém após tentar de várias formas, as variáveis locais não eram atualizadas com os valores calculados na GPU.

Para tentar resolver este problema, foi alterado o algoritmo para que esse realizasse duas vezes o mesmo cálculo (como descrito na seção 3.3.1.2). Como não foi possível armazenar o valor do cálculo realizado pela primeira vez na GPU, necessitou-se realizá-lo novamente. Nessa etapa a limitação foi o *profile* utilizado para o *shader*. A placa de vídeo utilizada suportava até a versão 3 do *profile* para *vertex shader* (o hardware da placa não suportava a versão mais atual), porém era necessário suporte a versão 4 para rodar o programa por causa das limitações do *profile*. O código foi reestruturado para trabalhar com a versão 3 e não houve nenhuma perda na qualidade final.

Resolvendo este problema acabou-se tirando um pouco do desempenho da GPU, isto porquê o algoritmo acaba realizando 2 vezes o mesmo cálculo, quando a CPU faz somente uma vez. Realizando os cálculos uma vez é possível encontrar coordenadas de posição para 2

vértices, contudo a GPU pode retornar somente coordenadas para um vértice por vez. Outro perda de desempenho foi a necessidade de criar uma função para realizar a normalização de coordenadas, pois as posições originais dos vértices ultrapassavam as dimensões 1 e -1, o qual define o *view frustum* (seção 2.2.1) da GPU.

Outra limitação da ferramenta foi a questão da interação com a tela. Em CPU quando o objeto era movido ou aplicado *zoom* sobre ele, a tela era atualizada em tempo real. Em GPU é necessário pressionar o botão *Update View* para visualizar as mudanças. Como as mudanças na função `Ortho` não influenciam a visualização em GPU, a única alternativa para interação em tempo real seria executar toda a técnica novamente.

3.4.2 Testes de desempenho e memória

Todos os testes para medição do desempenho de processamento entre CPU e GPU foram realizados utilizando sempre como base o arquivo de entrada `vg.dat`. Foram realizados dois tipos de teste de desempenho, tanto para a técnica `Colormap` como para a técnica `2D Vector Field`. O primeiro teste foi realizar várias iterações somente nos cálculos e instruções que fariam a representação da técnica na tela. O outro teste seria realizar várias iterações em toda a função. Para realizar os testes foi utilizado um computador com velocidade de 1,66 *GigaHertz* (GHz) e 1 GB de memória. O número máximo de iterações foi definido levando em consideração a quantidade de memória ocupada. Valores acima dessa quantidade (no caso, aproximadamente 800 MB) acabavam gerando despejo de memória. Todas as medições de tempo utilizam da grandeza dos milissegundos (ms).

Para o primeiro teste na técnica `Colormap`, pode ser observado uma velocidade de processamento maior pela GPU. Como pode ser visto no Quadro 11, a GPU consegue realizar o processamento da técnica pelo menos duas vezes mais rápido que a CPU.

Iterações	CPU	GPU
1.000.000	416	147
2.000.000	834	319
3.000.000	1.260	478
4.000.000	1.666	672
5.000.000	2.106	784
6.000.000	2.519	1.115
7.000.000	2.928	1.159

Quadro 11 – Tempos de processamento do primeiro teste da técnica colormap

Na Figura 13 pode-se observar um gráfico contendo os valores da tabela. No eixo y

estão os tempos tomados e são representados em ms. No eixo x encontram-se o número de iterações aplicadas no teste. Olhando o gráfico é possível observar que a CPU mantém um processamento constante levando em consideração a quantidade de iterações. Contudo como a GPU é mais rápida para realizar o processamento, quanto maior a quantidade de iterações maior será a diferença entre ambas.

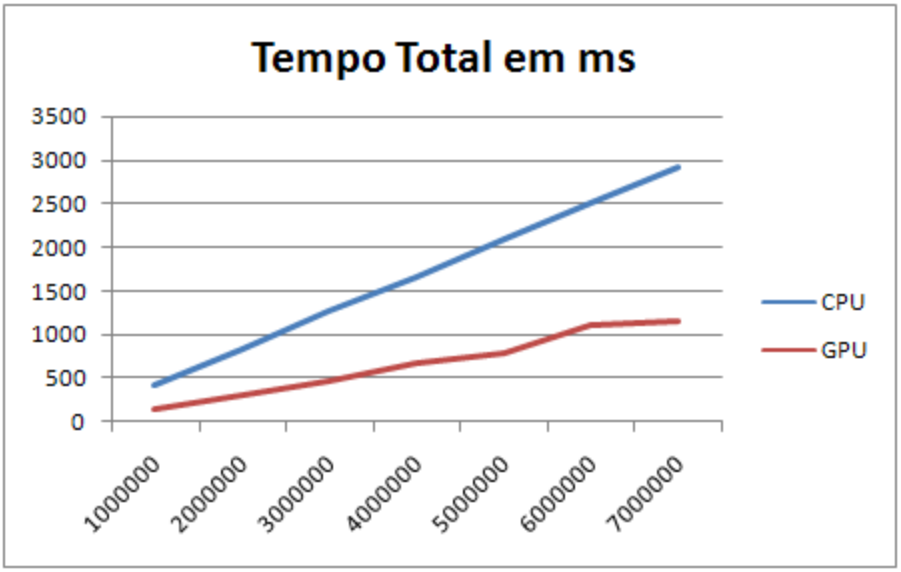


Figura 13 – Gráfico com os tempos do primeiro teste da técnica `colormap`

Vendo os resultados do segundo teste executado para a técnica `Colormap`, observa-se que a CPU acaba recebendo vantagem sobre a GPU. Contudo como pode ser visualizado no Quadro 12, existe uma função da OpenGL denomidada `CallList`, e quando a técnica é executada em GPU essa rotina acaba levando aproximadamente o dobro do tempo para ser executada. Se levarmos em consideração que o tempo de execução dessa função fosse o mesmo para ambos os testes, a GPU teria um desempenho semelhante a da CPU.

Iterações	CallList (CPU)	CallList (GPU)	Tempo Total (CPU)	Tempo Total (GPU)
1.000	63.784	114.764	85.984	139.594
2.000	128.094	230.347	170.734	281.078
3.000	193.256	337.831	257.875	400.984
4.000	259.155	460.755	346.921	563.875
5.000	322.558	576.877	433.656	707.360
6.000	384.547	692.815	508.437	847.297
7.000	451.272	804.788	600.297	976.359

Quadro 12 – Tempos de processamento do segundo teste da técnica `colormap`

Na Figura 14 tem-se um gráfico contendo os valores obtidos pelo segundo teste de desempenho da técnica `colormap`. É possível observar que o processamento pela GPU leva mais tempo que pela CPU. Contudo quando colocamos o mesmo tempo para o processamento da função `CallList`, observa-se que o desempenho é semelhante para ambos os hardwares

havendo pouca diferença entre eles. Esse processamento pela GPU com o tempo da função `CallList` igual à da CPU é demonstrado na cor verde pelo gráfico.

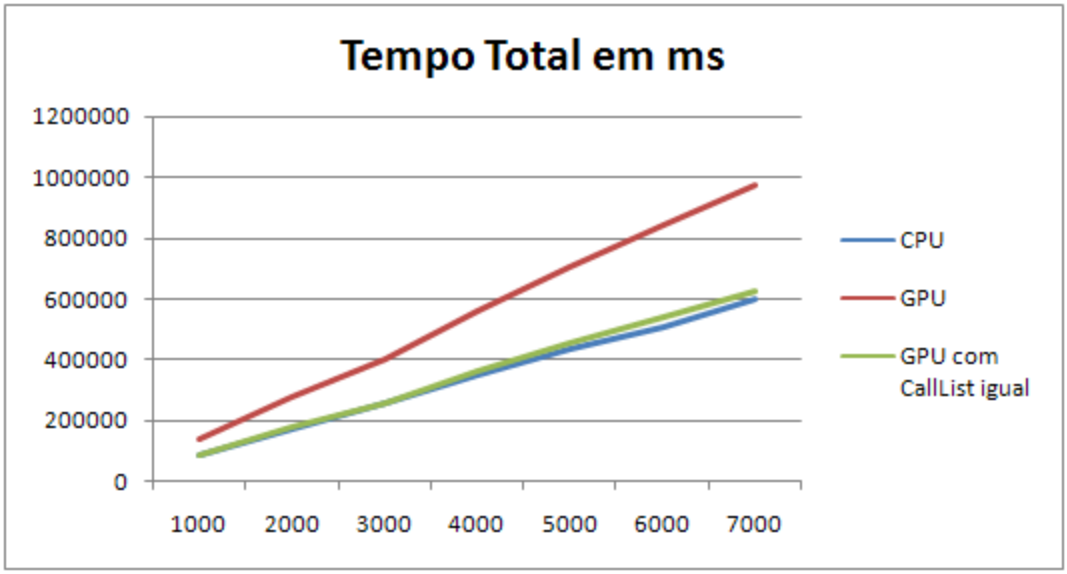


Figura 14 – Gráfico com os tempos do segundo teste da técnica `colormap`

Com relação à técnica `2D Vector Field` pode-se observar os mesmos resultados. No primeiro teste realizado a GPU possui um processamento mais rápido que da CPU. Separando essa técnica em duas partes, sendo a parte1 responsável por realizar o cálculo da cor e representação da haste principal do vetor, e a parte 2 responsável por encontrar as duas arestas que formam topo do vetor, é possível verificar que a CPU acaba sendo mais rápida que a GPU na segunda parte da técnica. Como pode ser observado no Quadro 13, mesmo a CPU conseguindo ser mais rápida em certas ocasiões, a GPU na técnica como um todo acaba tendo os melhores tempos.

Iterações	Parte 1 (CPU)	Parte 1 (GPU)	Parte 2 (CPU)	Parte 2 (GPU)	Tempo Total (CPU)	Tempo Total (GPU)
1.000.000	1.666	1.472	1.862	1.803	3.528	3.275
2.000.000	3.269	2.909	3.759	3.676	7.028	6.585
3.000.000	5.327	4.393	5.444	5.629	10.771	10.021
4.000.000	6.957	6.461	7.659	7.971	14.616	14.431

Quadro 13 – Tempos de processamento do primeiro teste da técnica 2D vectorfield

A Figura 15 mostra um gráfico contendo os tempos das duas partes da técnica `vectorfield`, coletados no primeiro teste realizado. Cada amostra de tempo no gráfico é dividida em quatro colunas, sendo as duas primeiras relativas a primeira parte da técnica e as duas seguintes responsáveis por mostrarem a segunda parte da técnica. Assim é possível observar que a primeira parte da técnica é mais rápida quando executada pela GPU, porém a segunda parte depois de algumas iterações é mais rápida pela CPU.

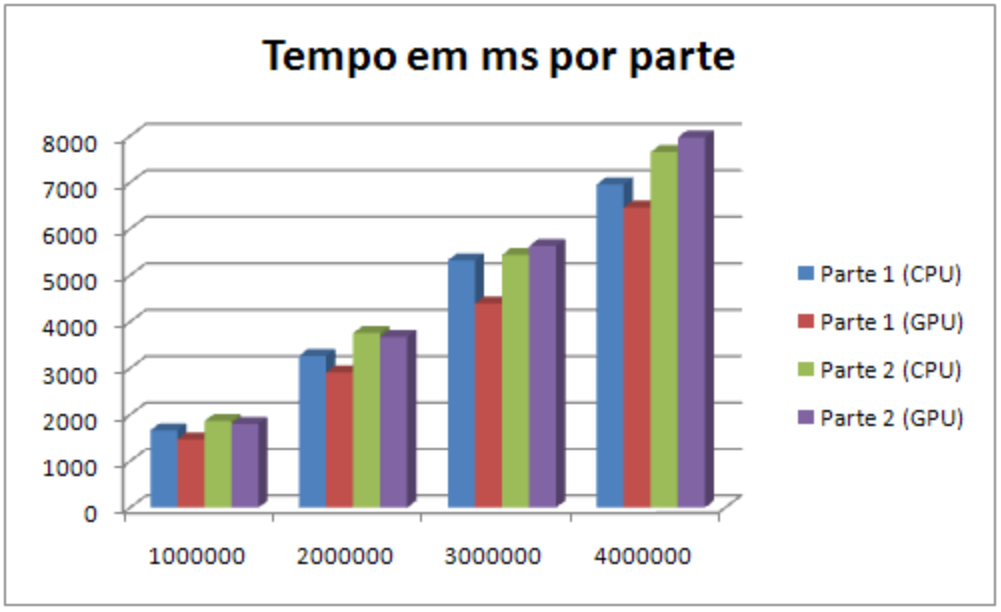


Figura 15 – Gráfico com os tempos do primeiro teste da técnica `vectorfield`

No segundo teste realizado observa-se o mesmo acontecimento da técnica `colormap`. O tempo da função `CallList` é maior quando a técnica é executada pela GPU. Como pode ser observado no Quadro 14 considerando que o tempo da função `CallList` seja o mesmo para os teste em CPU e GPU, pode-se notar que a técnica possui quase o mesmo tempo em ambos os testes.

Iterações	Calllist (CPU)	Calllist (GPU)	Tempo Total (CPU)	Tempo Total (GPU)
1.000	20.403	34.995	50.156	66.968
2.000	40.803	69.760	100.172	133.547
3.000	61.445	104.569	150.188	200.250
4.000	81.891	138.664	200.203	266.937
5.000	102.402	175.198	250.281	333.609
6.000	122.765	209.161	300.235	400.328
7.000	143.138	243.332	350.250	467.250

Quadro 14 – Tempos de processamento do segundo teste da técnica `vectorfield`

Olhando a Figura 16 pode-se observar que a GPU leva mais tempo para processar a técnica. Contudo quando os tempos da função `CallList` são iguais para a CPU e GPU, nota-se um desempenho muito parecido entre ambos os hardwares. O tempo requerido pela GPU para processar a técnica, considerando que a função `CallList` tem o mesmo desempenho que em CPU, é demonstrado na cor verde pelo gráfico.

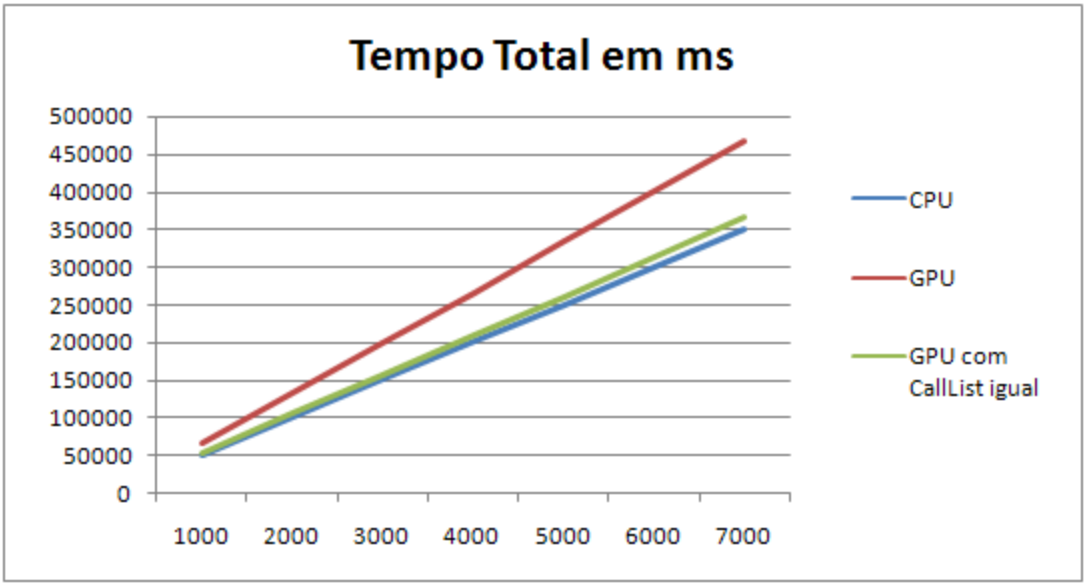


Figura 16 – Gráfico com os tempos do segundo teste da técnica `vectorfield`

Foi possível observar também que a quantidade de memória necessária para rodar os testes no sistema operacional Windows, variou de acordo com a técnica. As variações na quantidade de memória utilizada para os testes realizados em CPU e GPU foram encontradas principalmente no primeiro teste realizado em cada técnica. Para a técnica `colormap` quando eram realizados os testes com a GPU, observou-se que a quantidade de memória permaneceu estável em torno de 69 MB para todos os testes. Já com a CPU a quantidade de memória aumentava de acordo com o aumento de iterações realizadas, sendo que quando o teste era realizado com 7.000.000 de iterações a quantidade de memória ultrapassava os 700 MB. Para os testes realizados com a técnica `vectorfield` a situação se inverteu. Os testes realizados em ambos os hardwares aumentavam o consumo de memória de acordo como aumentavam a quantidade de iterações. Contudo a quantidade de memória consumida pela GPU acaba sempre sendo maior que a utilizada pela CPU, sendo que a diferença era de aproximadamente 100 MB. Já a memória interna da GPU necessita de 88 bytes para processar a técnica `colormap` e 116 bytes para processar a técnica `vectorfield`.

3.4.3 Comparação da representação gráfica

Na Figura 17 é mostrada a representação gráfica da técnica colormap obtida no processamento pela CPU. Já na Figura 18 pode-se visualizar a representação gráfica obtida no processamento pela GPU. Nota-se que em ambas as figuras a representação gráfica parece ser idêntica, contudo existem algumas diferenças entre elas.

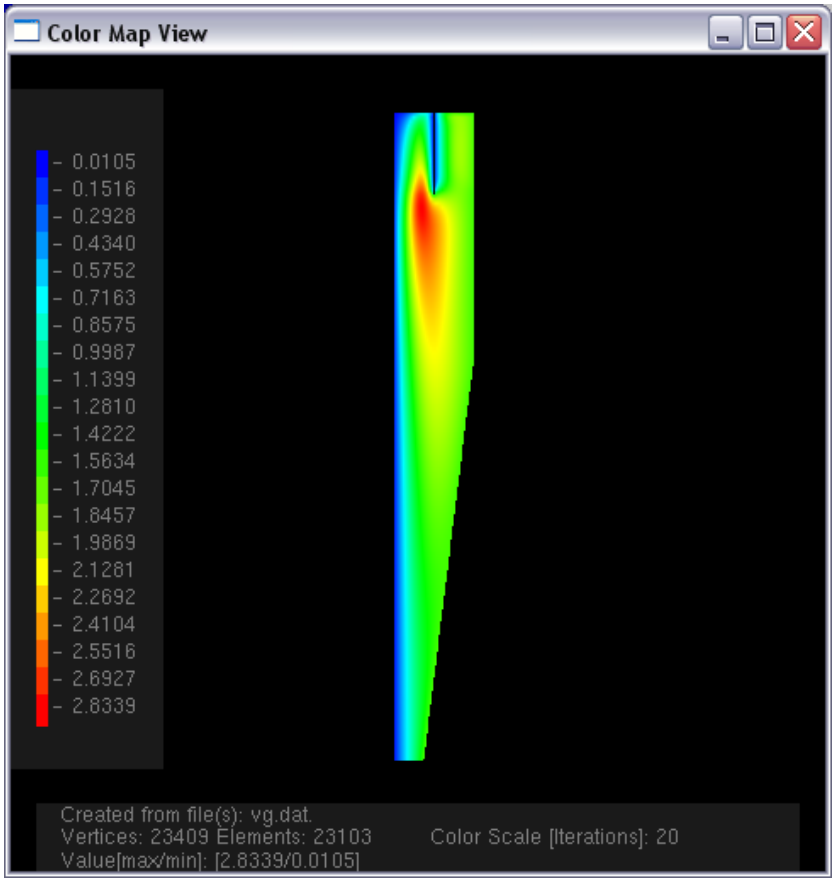


Figura 17 – Representação gráfica da técnica `colormap` pela CPU

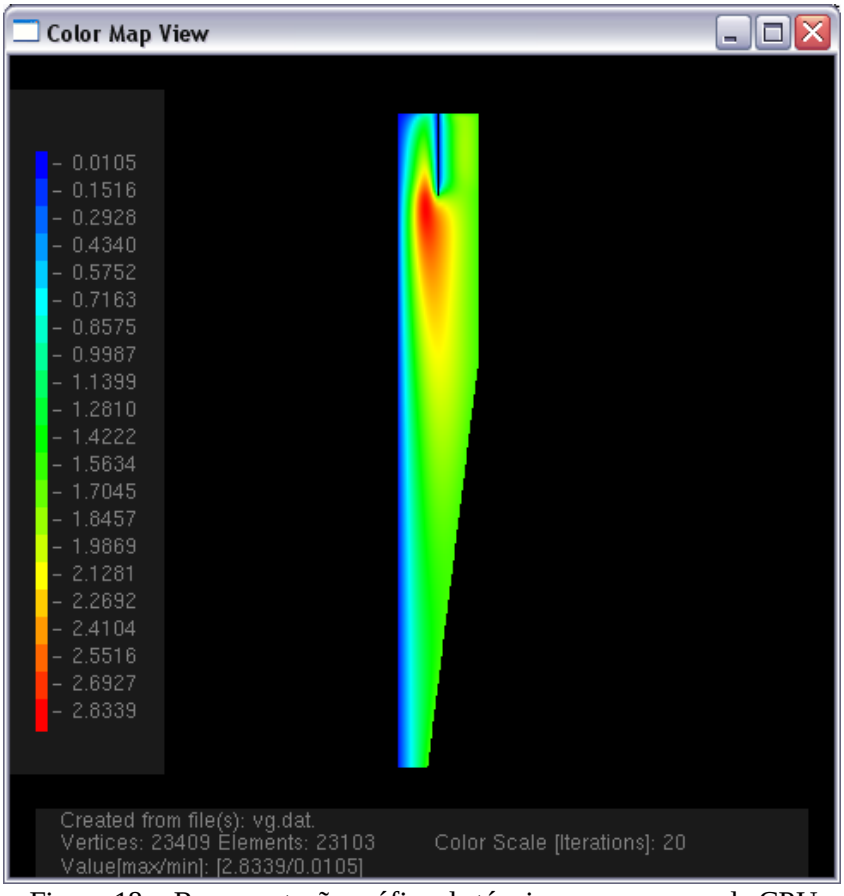


Figura 18 – Representação gráfica da técnica `colormap` pela GPU

Na Figura 19 pode-se observar as diferenças existentes na representação gráfica entre os dois processamentos. As áreas pintadas em branco representam os *pixels* que contém diferença na coloração. Sendo assim observa-se que uma grande quantidade da área onde é representada a técnica possui diferença de cor. Contudo como visualizado anteriormente as Figura 17 e Figura 18 são quase idênticas. Isso acontece por que as diferenças nos valores que compõe a cor são muito baixas, sendo difíceis de distinguir visualmente.

As imagens comparadas possuem uma quantidade de 250.832 *pixels* (488 x 514), sendo que dessa quantidade apenas 14.370 *pixels* correspondem ao escopo da técnica representada. Desse montante houve uma quantidade de 4.075 *pixels* diferentes, o que corresponde a aproximadamente 28,35% de diferença. Também foi calculado a diferença das cores geradas pela CPU e pela GPU, sobre a quantidade de iterações utilizadas na técnica. Encontrou-se uma variação média da cor de aproximadamente de 3,87% quando representado pela GPU.

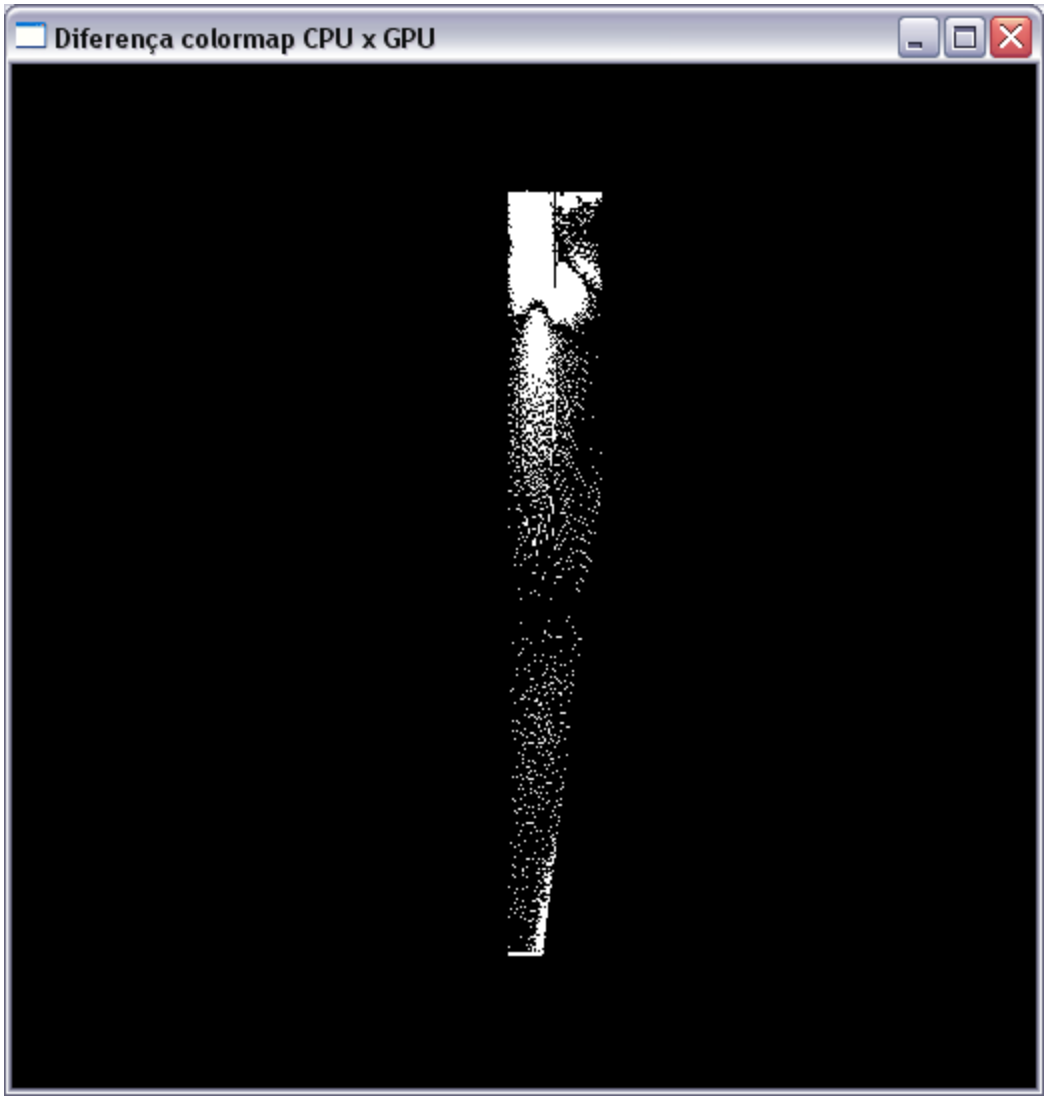


Figura 19 – Diferenças na representação gráfica da técnica `colormap`

Agora na Figura 20 é possível visualizar a representação gráfica da técnica `vectorfield` pela CPU, e na Figura 21 a representação gráfica pela GPU. Nessas figuras a representação gráfica é semelhante, porém as diferenças podem ser notadas visualmente.

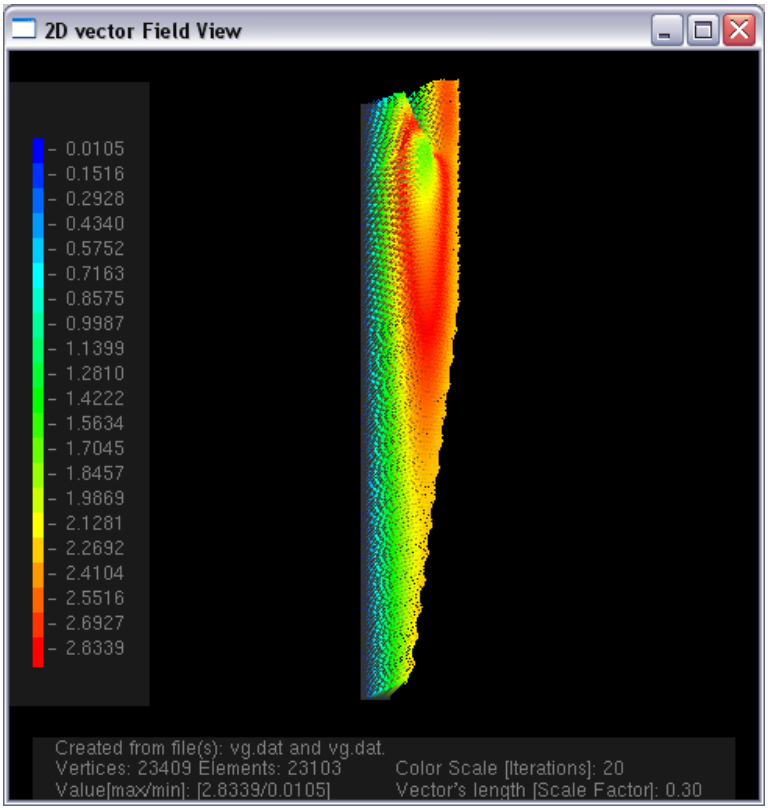


Figura 20 – Representação gráfica da técnica `vectorfield` pela CPU

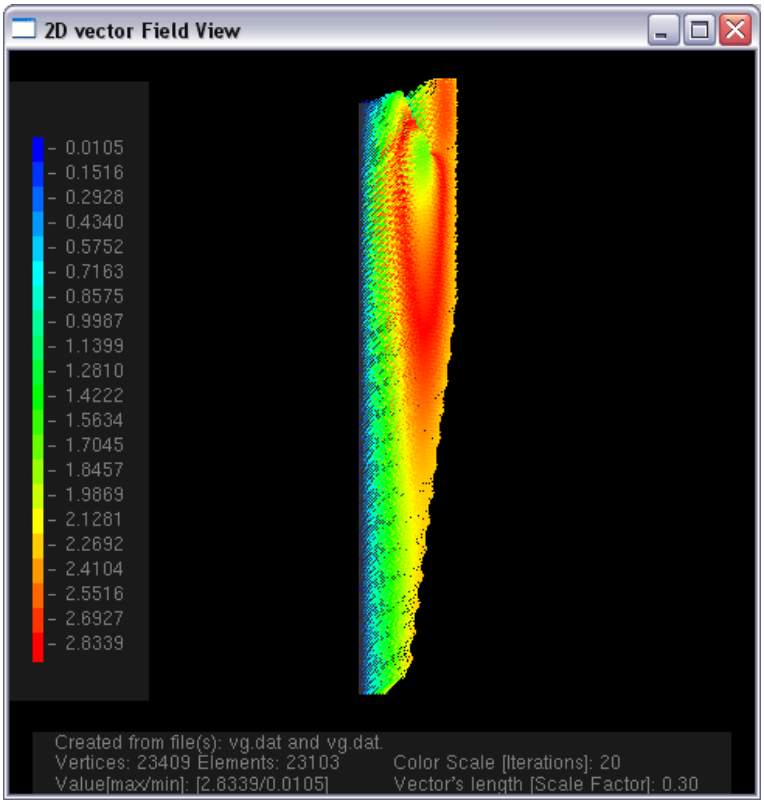


Figura 21 – Representação gráfica da técnica `vectorfield` pela GPU

Foi realizado o mesmo método de comparação da técnica `colormap` para verificar os *pixels* com diferenças nas cores. A Figura 22 demonstra essa diferença, onde as áreas em branco representam os *pixels* com cores diferentes. Nessa técnica a diferença ocorre não somente pela cor, mas também pelo espaçamento entre cada vetor representado. Como pode ser observado na Figura 20, a representação pela CPU parece conter espaçamentos maiores entre os vetores, geralmente visualizados pelas cores mais escuras. Já a GPU possui uma quantidade menor desses espaços, como pode ser observado na Figura 21.

A quantidade de *pixels* das imagens comparadas é de 250.832 (488 x 514), sendo que dessa quantidade apenas 19.993 *pixels* pertencem ao escopo da técnica representada. Desse montante houve 9.605 *pixels* com cores diferentes, o que corresponde a aproximadamente 48,04% de diferença. Realizando também a comparação da diferença das cores pela CPU e GPU, encontrou-se uma variação média de aproximadamente 136,74% na representação das cores pela GPU. Sendo que o cálculo foi realizado sobre a quantidade de iteração de cores utilizados na técnica.

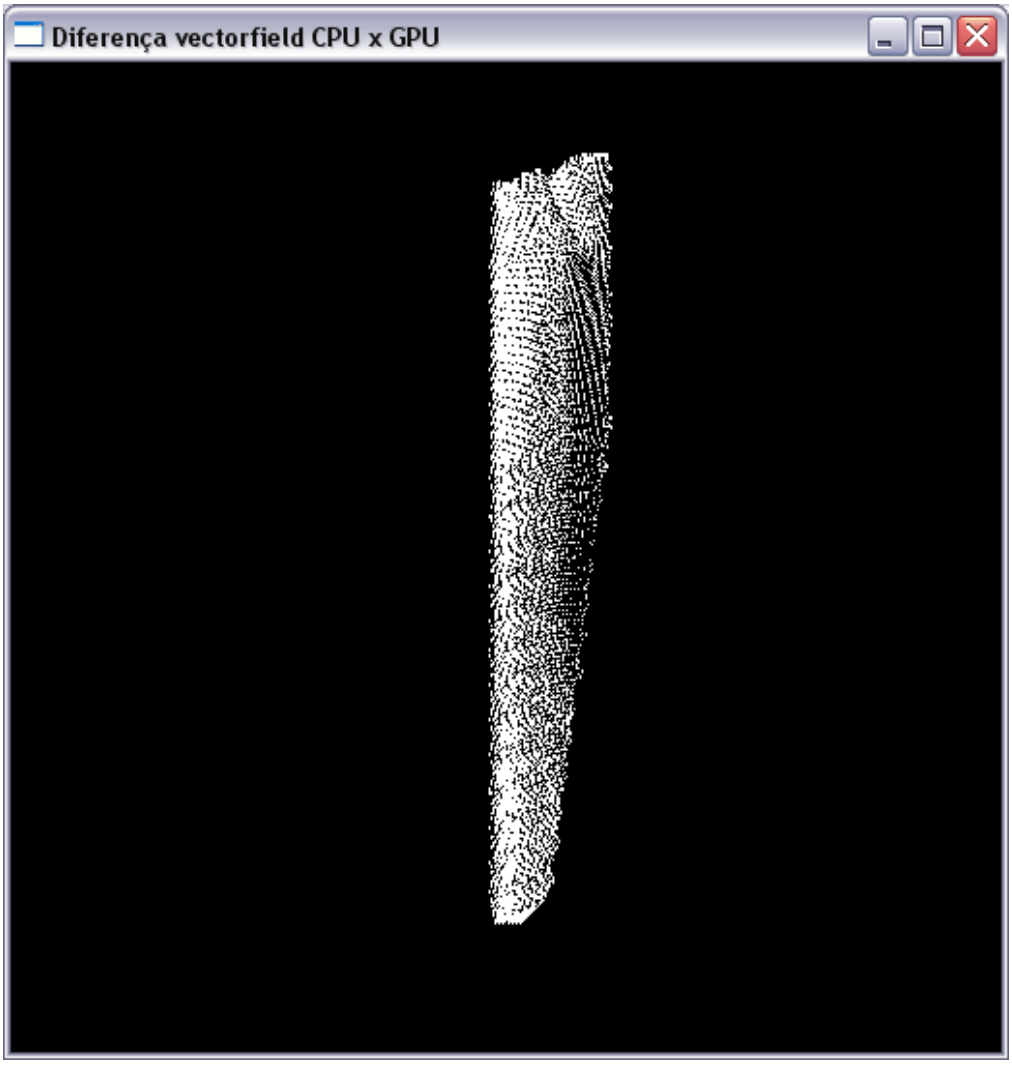


Figura 22 – Diferenças na representação gráfica da técnica `vectorfield`

4 CONCLUSÕES

Neste trabalho foram convertidas duas técnicas em GPU das três implementadas no trabalho utilizado como base para realização desse (PIVA; GOMES; REIS, 2008). Na técnica `colormap` observou-se um desempenho superior no processamento pela GPU. No entanto na técnica `vectorfield` mesmo o processamento sendo melhor na GPU, a segunda parte da técnica por algumas vezes torna-se mais rápida na CPU. Isso acontece pois para rodar a técnica em GPU foi necessário realizar os cálculos da técnica 2 vezes. Dessa forma os tempos se mantiveram muito próximos em um contexto geral. Por outro lado, para rodar os programas em GPU acaba-se consumindo uma quantidade de memória local maior.

Já referente a representação gráfica das técnicas, mostraram-se pequenas diferenças visuais entre ambas as partes, contudo quando realizada uma verificação mais minuciosa verificou-se uma quantidade razoável de pixels diferentes. Essa diferença pode ter sido ocasionada pela precisão das variáveis utilizadas em ambos os hardwares, sendo que será necessária uma verificação mais detalhada para encontrar as causas.

A GPU mostra-se superior ao executar cálculos para grandes quantidades de dados, porém a construção do algoritmo acaba por vezes tendo complicações. Isto se deve pois cada perfil de programação possui suas limitações. Pode-se esbarrar nessas limitações sem perceber e consequentemente não conseguindo gerar o efeito. Utilizar perfis mais atuais resolveria a maioria dos problemas, porém estaria restringindo-se os hardwares gráficos que podem ser utilizados pela aplicação. Sendo assim a relevância computacional desse projeto deve-se a implementação dos algoritmos, para que os mesmo possam ser executados pela maior quantidade de hardwares gráficos possíveis.

As ferramentas utilizadas mostraram-se bastante práticas para o desenvolvimento do projeto. Destaca-se a IDE FX Composer que foi utilizado para verificar erros de sintaxe no desenvolvimento das técnicas. A biblioteca utilizada para comunicação da API com o hardware gráfico também mostrou-se estável, existindo a possibilidade de encontrar e mostrar erros em tempo de execução.

Com relação aos trabalhos correlatos, verifica-se que o trabalho proposto possui algumas funcionalidades existentes relacionadas, como a utilização do hardware gráfico para processar os cálculos de grandes quantidades de dados. A principal diferença esta no objetivo do trabalho, pois trata de realizar a comparação das técnicas desenvolvidas em CPU e GPU, demonstrando as vantagens e limitações de ambos os hardwares.

Por fim, obteve-se uma ferramenta funcional possuindo todos os recursos da ferramenta original (PIVA; GOMES; REIS, 2008). Algumas limitações acabaram ocorrendo no projeto, porém nada que acarretasse um mal funcionamento do sistema.

4.1 EXTENSÕES

Para versões futuras sugere-se algumas melhorias como realizar os cálculos da técnica `2D Vector Field` com uma placa de vídeo mais moderna, ou ainda utilizando-se texturas para realizar os cálculos. Retornando os valores calculados para a memória local consegue-se um melhor desempenho na GPU, pois não será necessário realizar duas vezes os mesmos cálculos.

Incluir visualização das interações com os objetos em tempo real. Dessa forma é possível ter uma noção mais exata do que se esta fazendo na hora de interagir com o aplicativo.

Implementar a técnica `3D Stream Ribbons`, que não foi implementada por apresentar problemas quando executada nos sistemas operacionais Windows e MAC OS.

Desenvolver as técnicas baseando-se no diagrama mostrado na seção 3.2 Especificação. Implementando os algoritmos utilizando-se técnicas e passos é possível gerar códigos específicos para os perfis de hardwares, podendo aumentar assim o desempenho final para cada hardware.

Realizar uma verificação das técnicas já implementadas, tentando minimizar a quantidade de pixels com cores diferentes.

REFERÊNCIAS BIBLIOGRÁFICAS

AMD. **ASHLI:** advanced shading language interface. [Sunnyvale], [2008]. Disponível em: <http://developer.amd.com/gpu/archive/ashli/Pages/default.aspx>. Acesso em: 27 ago. 2008.

_____. **RenderMonkey Toolsuite.** [Sunnyvale], [2009]. Disponível em: <http://developer.amd.com/gpu/rendermonkey/Pages/default.aspx#download>. Acesso em: 13 abr. 2009.

ARB (GPU assembly language). In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2008. Disponível em: <http://en.wikipedia.org/wiki/ARB_(GPU_assembly_language)>. Acesso em: 11 set. 2008.

BABOO. **Galeria:** evolução das placas de vídeo. [S.l.], 2009. Disponível em: <http://www.baboo.com.br/absolutenm/templates/content.asp?articleid=34174&zoneid=262>. Acesso em: 08 mar. 2009.

BAIXAKI JOGOS. **Crysis.** [S.l.], [2008]. Disponível em: <http://www.baixakijogos.com.br/pc/crysis/analise.html>. Acesso em: 14 set. 2008.

C FOR graphics. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2008. Disponível em: <http://pt.wikipedia.org/wiki/C_for_graphics>. Acesso em: 21 set. 2008.

COLLADA. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: <http://www.collada.org/mediawiki/index.php/COLLADA>. Acesso em: 12 abr. 2009.

DIETRICH, C. A. et al. Real-time interactive visualization and manipulation of the volumetric data using GPU-based methods. In: SPIE MEDICAL IMAGING: VISUALIZATION, IMAGE-GUIDED PROCEDURES AND DISPLAY, 5., 2004, San Diego. **Proceedings...** Washington: The International Society for Optical Engineering, 2004. p. 181-192. Disponível em: <http://www.inf.ufrgs.br/cg/publications/cadietrich/spie-5367-19.pdf>. Acesso em: 15 set. 2008.

FERNANDES, A. R. **GLSL:** introdução e programação da aplicação OpenGL. [Braga], 2006. Disponível em: <http://sim.di.uminho.pt/disciplinas/mm/0607/t10/t10.pdf>. Acesso em: 15 mar. 2009.

FERNANDO, R. (Ed.). **GPU GEMS:** programing techniques, tips and tricks for real-time graphics. Santa Clara: Pearson Education, 2004. Disponível em: <http://http.developer.nvidia.com/GPUGems/gpugems_part01.html>. Acesso em: 27 ago. 2008.

FLUID dynamics. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2008. Disponível em: <http://en.wikipedia.org/wiki/Fluid_dynamics>. Acesso em: 14 set. 2008.

GIRALDI, G. A.; FEIJÓO, R. A. **Visualização de fluidos em dinâmica de fluidos computacional.** Petrópolis, [2003?]. Disponível em: <http://virtual01.lncc.br/monografia/monografia3/node1.html>. Acesso em: 14 set. 2008.

GÖDDEKE, D. **Languages and programming environments.** Dresden, 2008. Disponível em: <http://www.feast.uni-dortmund.de/showpdffile_Goeddeke2008.pdf>. Acesso em: 20 set. 2008.

KESSENICH, J.; BALDWIN, D.; ROST, R. **The OpenGL shading language.** Madison: 3Dlabs, 2006. Disponível em: <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf>. Acesso em: 20 mar. 2009.

KUHN, G. R.; OLIVEIRA, M. M.; FERNANDES, L. A. F. An efficient naturalness-preserving image-recoloring method for dichromats. In: IEEE VISWEEK 2008, 19., 2008, Columbus. **Anais...** Ohio: Journal IEEE Transactions on Visualization and Computer Graphics, 2008. p. 1747-1754. Disponível em: <http://www.inf.ufrgs.br/~oliveira/kuhn.pdf>. Acesso em: 15 jul. 2009.

MSDN. **HLSL.** [S.l.], 2009a. Disponível em: <http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx>. Acesso em: 20 fev. 2009.

_____. **Reference for HLSL.** [S.l.], 2009b. Disponível em: <http://msdn.microsoft.com/en-us/library/bb509638.aspx>. Acesso em: 10 jun. 2009.

MORIMOTO, C. E. **Hardware manual completo.** São Paulo: GDH Press e Sul Editores, 2002. Não paginado. Disponível em: <http://www.gdhpress.com.br/hmc/leia/index.php?p=cap8-1>. Acesso em: 04 mar. 2009.

_____. **Placas de vídeo 3D:** uma introdução. [S.l.], 2007. Disponível em: <http://www.guiadohardware.net/dicas/placas-video-3d.html>. Acesso em: 14 set. 2008.

NVIDIA. **FX Composer 2.5.** [Santa Clara], [2008]. Disponível em: <http://developer.nvidia.com/object/fx_composer_home.html>. Acesso em: 13 abr. 2009.

_____. **The Cg tutorial.** [Santa Clara], [2009]. Disponível em: <http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html>. Acesso em: 20 fev. 2009.

_____. **Developer forums.** [Santa Clara], 2007. Disponível em: <http://developer.nvidia.com/forums/index.php?showtopic=2777&hl=diagra>. Acesso em: 06 jun. 2009.

_____. **Cg user's manual:** a developer's guide to programmable graphics**.** [Santa Clara], 2005. Disponível em: <http://developer.download.nvidia.com/cg/Cg_2.1/2.1.0017/CgUsersManual.pdf>. Acesso em: 12 abr. 2009.

OLIVEIRA JR., J. A. A. **Desenvolvimento de um sistema de dinâmica dos fluidos computacional empregando o método de elementos finitos e técnicas de alto desempenho.** 2006. 112 f. Dissertação (Mestrado em Engenharia) - Curso de Pós-Graduação em Engenharia Mecânica, Universidade Federal do Rio Grande do Sul, Porto Alegre. Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/10617/000599814.pdf?sequence=1>. Acesso em: 18 abr. 2009.

PHARR, M.; FERNANDO, R. **GPU GEMS 2:** programming techniques for high-performance graphics and general-purpose computation. Santa Clara: Pearson Education, 2005. Não paginado. Disponível em: <http://http.developer.nvidia.com/GPUGems2/gpugems2_part01.html>. Acesso em: 11 set. 2008.

PIVA, G. R.; GOMES, P. C.; REIS, D. Poscyclo: a finite-volume post-processor visualization system. In: SEMINÁRIO INTERNO DE COMPUTAÇÃO DA FURB, 17., 2008, Blumenau, SC. **Anais...** Blumenau, SC: FURB, 2008. Disponível em: <http://www.assembla.com/wiki/show/brfurbcfd>. Acesso em: 27 out. 2008.

PIXAR. **What's RenderMan?** [Emeryville], [2008]. Disponível em: <https://renderman.pixar.com/products/whatsrenderman/index.htm>. Acesso em: 13 set. 2008.

SOUZA, F. J. **Dinâmica dos fluidos computacional.** [S.l.], 2008. Disponível em: <http://www.geocities.com/chicao99br/dia1.2.pdf>. Acesso em: 18 abr. 2009.

UNIVERSITY OF PENNSYLVANIA. **GPU programming "languages".** Philadelphia, [2008?]. Disponível em: <http://www.cis.upenn.edu/~suvenkat/700/lectures/21/gpusl.ppt>. Acesso em: 27 ago. 2008.

VIANA, J. R. M.; CUNO, A. **Conceitos de programação em GPU.** Rio de Janeiro, [2008?]. Disponível em: <http://www.lcg.ufrj.br/Cursos/GPUProg/gpuintro>. Acesso em: 15 set. 2008.