

Danilo Tuler de Oliveira

**Uma abstração de alto nível para
programação do processador gráfico**

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática da PUC-Rio.

Orientador: Waldemar Celes

Rio de Janeiro

Março de 2003

Danilo Tuler de Oliveira

**Uma abstração de alto nível para
programação do processador gráfico**

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Waldemar Celes Filho

Orientador

Departamento de Informática – PUC-Rio

Prof. Marcelo Gattass

Departamento de Informática – PUC-Rio

Prof. Luiz Carlos Pacheco R. Velho

IMPA

Prof. Luiz Henrique de Figueiredo

IMPA

Prof. Ney Dumont

Coordenador Setorial do Centro

Técnico Científico – PUC-Rio

Rio de Janeiro, 09 de setembro de 2002

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Danilo Tuler de Oliveira

Graduou-se em Engenharia de Computação pela PUC-Rio em 1999.

Ficha Catalográfica

Tuler, Danilo

Uma abstração de alto nível para programação do processador gráfico / Danilo Tuler de Oliveira; orientador: Waldemar Celes Filho. - Rio de Janeiro: PUC, Departamento de Informática, 2003.

v., 70f.: il.; 29,7cm

1. Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Incluí referências bibliográficas.

1. pipeline programável. 2. linguagem de shading. 3. renderização.

Agradecimentos

Ao meu orientador Professor Waldemar Celes pelo estímulo e parceria para a realização deste trabalho.

Ao CNPq e à PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Aos meus amigos Sérgio, Renata, Suzana, Diogo, Diego, Mark, Fábio e Cacá por todo apoio.

À minha mãe, pela educação, atenção e carinho de todas as horas.

Aos professores que participaram da Comissão examinadora.

A todos os amigos e familiares que de uma forma ou de outra me estimularam ou me ajudaram.

Resumo

Tuler, Danilo; Celes, Waldemar. **Uma abstração de alto nível para programação do processador gráfico**. Rio de Janeiro, 2002. 65p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Os processadores gráficos disponíveis na atualidade se tornaram programáveis. No entanto, esta programabilidade é oferecida em um nível muito baixo de abstração, em geral utilizando uma linguagem de máquina. Portanto, é difícil explorar estas novas funcionalidades. Para solucionar este problema foi realizado um estudo das várias formas de acesso aos recursos de hardware e apresentamos uma abstração simples do pipeline programável. Nossa proposta é construída sobre a linguagem de programação Lua. O programador da aplicação escreve um código Lua especializado que, quando executado, gera o código de máquina equivalente que será carregado pelo processador gráfico. Diversas pesquisas tem sido feitas nesta área. Nós comparamos nossa proposta com outras recentemente publicadas ou ainda em desenvolvimento.

Palavras-chave

Pipeline programável, linguagem de shading, renderização.

Abstract

Tuler, Danilo; Celes, Waldemar (Advisor). **High-Level Abstraction for Graphics Hardware Programming**. Rio de Janeiro, 2002. 65p. MSc. Dissertation – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Currently available graphics hardware has become programmable. However, this programmability is offered at a very low-level of abstraction, in general using an assembly language. Therefore, it is hard to explore these hardware features. To overcome this problem a survey of the several ways to access the hardware features was done and we present a simple programmable pipeline abstraction. Our proposal is built on the top of the Lua programming language. The application programmer writes a specialized Lua code that, when executed, generates the equivalent assembly code to be loaded by the hardware. Several researches have been conducted on this subject. We compare our proposals to others recently published or still under consideration.

Keywords

Programmable pipeline, shading language, rendering

Sumário

Resumo	4
Abstract	6
Sumário	7
Lista de figuras	9
Lista de tabelas	10
1 Introdução	11
1.1 Motivação e objetivo	13
2 Pipeline Convencional	15
2.1 Arquitetura	15
2.2 API's	18
2.3 Extensibilidade	19
3 Pipeline programável	21
3.1 Arquitetura	21
3.2 Processamento de Vértices	21
3.3 Processamento de Fragmentos	33
4 Abstração	35
4.1 Stanford Real-time Shading Language	36
4.2 OpenGL 2.0	37
4.3 Cg	40
5 Lua Shading Language	43
5.1 Tipos de Dados	43
5.2 Arquitetura	44
5.3 Geração de código	45
5.4 Ambiente Global	45
5.5 Vertex Shader	46
5.6 Fragment Shader	52

6	Resultados	53
6.1	IslSimple	53
6.2	IslFresnel	56
6.3	IslBillboard	58
7	Conclusão	60
8	Referências Bibliográficas	62
	Apêndice A - Processadores Gráficos	64
	Apêndice B - API da Lua Shading Language	66
	Apêndice C - Uso de extensões OpenGL	68

Lista de figuras

Figura 1 - Evolução dos sistemas gráficos	12
Figura 2 - Estágios conceituais e funcionais do pipeline convencional	15
Figura 3 - Pipeline Programável	21
Figura 4 - Processador de Vértice	22
Figura 5 - Arquitetura de geração de código	45
Figura 6 - Alocação de objetos	48
Figura 7 – IslSimple	55
Figura 8 - IslFresnel	58
Figura 9 – IslBillboard	59

Lista de tabelas

Tabela 1 - Vertex Shader - registradores de entrada	24
Tabela 2 - Vertex Shader - registradores de saída	25
Tabela 3 - Vertex Shader - instruções	25
Tabela 4 - Vertex Shader - exemplo	27
Tabela 5 - NV_vertex_program - exemplo	28
Tabela 6 - EXT_vertex_shader – exemplo	30
Tabela 7 - ARB_vertex_program - exemplo	32
Tabela 8 - Stanford Real-time shading language - exemplo	37
Tabela 9 - Exemplo de criação de um shader em OpenGL 2.0	39
Tabela 10 - Exemplo de um vertex shader em OpenGL 2.0	40
Tabela 11 - Exemplo de um shader em Cg	41
Tabela 12 - Exemplo do ambiente global	46
Tabela 13 - Exemplo de vertex shader	47
Tabela 14 - Código gerado para NV_vertex_program	50
Tabela 15 - Código gerado para EXT_vertex_shader	52
Tabela 16 - Código nativo gerado por Cg	56
Tabela 17 - API da Lua Shading Language	67
Tabela 18 - Trechos do arquivo glsl.h	69
Tabela 19 - Importação de funções de extensão	69

1 Introdução

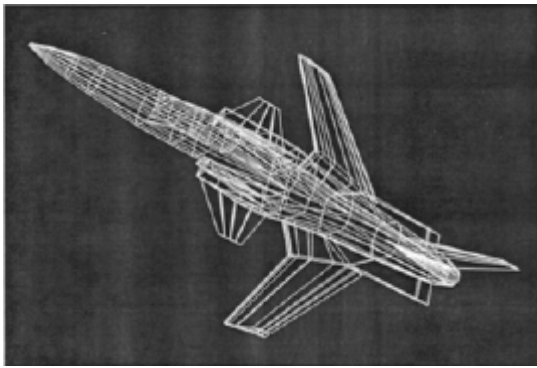
O subsistema responsável pela parte gráfica de uma aplicação tem evoluído muito nos últimos anos. Essa evolução percorre três caminhos distintos mas correlacionados. A primeira evolução diz respeito às funcionalidades e tarefas desempenhadas pelo subsistema gráfico, a segunda ao desempenho do processador e, por fim, a qualidade das imagens produzidas.

Os primeiros sistemas gráficos eram responsáveis somente por operações em duas dimensões. A tarefa básica desse sistema é copiar para a tela informações de imagens em duas dimensões tipicamente armazenadas em uma área de memória denominada *frame buffer*. Dois tipos de evoluções surgiram neste tipo de sistema: novas operações de manipulação do *frame buffer* e a maximização do desempenho de transferência de informações da aplicação para o *frame buffer*.

O grande salto em termos de funcionalidades ocorreu com a delegação de tarefas para a geração de imagens tridimensionais para o subsistema gráfico.

A primeira geração de *hardware* era capaz de produzir imagens somente em *wireframe* (Figura 1-a). Todo vértice do modelo a ser renderizado é passado ao sistema gráfico a fim de ser transformado e projetado. Nesta geração, somente pontos e linhas podem ser rasterizadas e os pixels produzidos simplesmente sobre-escrevem aqueles do *frame buffer*.

A segunda geração é marcada pela capacidade de produzir polígonos iluminados (Figura 1-b). Para cada vértice enviado para o sistema é calculada uma cor através de um modelo de iluminação simplificado. A rasterização de primitivas aqui é capaz de processar triângulos e produzir pixels com informação de profundidade, ou fragmentos. Esta informação viabiliza o algoritmo de *z-buffer* para determinação de superfícies visíveis.



(a) primeira geração



(b) segunda geração



(c) terceira geração



(d) quarta geração

Figura 1 - Evolução dos sistemas gráficos¹

A terceira geração de hardware é marcada pela introdução de texturas para aumentar a qualidade e o realismo das imagens produzidas (Figura 1-c). Cada vértice agora possui a informação de coordenada de textura, usada para controlar a forma com que a textura será aplicada ao modelo. O processo de rasterização também deve ser incrementado a fim de interpolar estas coordenadas para cada fragmento produzido.

O desempenho dos processadores gráficos tem aumentando em ritmo ainda maior do que aquele observado nos processadores centrais. Há diversas formas de medir este desempenho; as mais comuns são o número de triângulos e o número de fragmentos processados por segundo.

A tecnologia empregada nos processadores gráficos é similar àquela dos processadores de uso geral. Processadores gráficos hoje possuem dezenas de milhões de transistores, número superior ao de muitos processadores centrais ainda em uso.

¹ Imagens extraídas de <http://graphics.stanford.edu/courses/cs448a-01-fall/lectures/lecture1/>

Os sistemas gráficos são hoje o principal motor da evolução de arquiteturas de memória. As aplicações atuais requerem uma capacidade enorme de transferência de dados tanto entre aplicação e processador gráfico quanto entre diferentes estágios do processamento interno.

A qualidade das imagens produzidas pelos sistemas de renderização em tempo real tem aumentado cada vez mais, se aproximando da qualidade obtida por sistemas de produção cinematográfica.

O aumento do desempenho do processador gráfico possibilita que modelos geométricos cada vez mais detalhados possam ser enviados para o sistema, produzindo imagens conseqüentemente com maior grau de realismo.

As novas funcionalidades implementadas pelos hardware atuais possibilitam a geração de imagens com efeitos complexos de iluminação, como sombras, reflexões e refrações.

1.1 Motivação e objetivo

A complexidade das tarefas desempenhadas pelo subsistema gráfico tem aumentado cada vez mais. Para controlar essa evolução é preciso expor algum mecanismo para a aplicação ter domínio completo sobre a máquina de renderização. O elemento chave da quarta geração de hardware é a programabilidade, o que permite que a aplicação especifique um programa que será executado pelo processador gráfico. Este programa define exatamente como o processador irá manipular os vértices enviados para o sistema e os fragmentos produzidos pelo processo de rasterização.

Esta programabilidade é exposta para a aplicação através de uma linguagem de baixo nível de abstração, o que dificulta o uso e o reuso de código. Além disso cada fabricante de hardware expõe essa funcionalidade de uma forma diferente.

Para difundir o uso da programabilidade é necessária uma linguagem de alto nível de abstração, que funcione com uma diversidade de hardware. Este trabalho de pesquisa tem por objetivo:

- analisar os diversos mecanismos atualmente disponíveis para se programar o hardware gráfico;

- propor e implementar uma linguagem com alto nível de abstração que ofereça maior facilidade de uso e que funcione em uma variedade de hardware.

Esta dissertação está organizada da seguinte forma. O próximo capítulo descreve como o sistema de renderização é modelado tradicionalmente, mostrando o caminho percorrido pelos dados de um modelo geométrico até chegar à imagem gerada na tela. São descritas as principais API's gráficas utilizadas atualmente e como elas se encaixam no modelo apresentado.

O capítulo 3 apresenta o modelo de *pipeline* programável, proposto pelos principais participantes da indústria, e descreve a forma com que essa funcionalidade é exposta para a aplicação através das API's de maior popularidade.

O capítulo 4 descreve as principais iniciativas no sentido de prover uma abstração de alto nível para a programação do pipeline de renderização. A proposta de abstração deste trabalho é apresentada no capítulo 5 e alguns resultados obtidos no capítulo 6.

Por fim, o capítulo 7 apresenta as conclusões obtidas por este trabalho e sugere trabalhos futuros.

2 Pipeline Convencional

Este capítulo apresenta o modelo adotado pela maioria dos sistemas gráficos atuais. O processamento gráfico é modelado por um pipeline de renderização, que pode ser segmentado em estágios conceituais e funcionais.

O objetivo final do pipeline é gerar uma imagem bidimensional de uma cena tridimensional. A cena é descrita por uma câmera virtual, objetos tridimensionais, fontes de luz, aparências (materiais, texturas), entre outros elementos.

A localização e a forma dos objetos na imagem final são determinadas pela sua geometria e localização de uma câmera virtual. A aparência dos objetos é determinada pelas propriedades do material e textura, pelas luzes do ambiente e modelos de iluminação empregados.

Cada estágio do pipeline será analisado nas seções seguintes do ponto de vista conceitual e funcional. As principais implementações do pipeline serão apresentadas somente na seção 2.2.

2.1 Arquitetura

O pipeline de renderização em tempo-real, ilustrado na Figura 2, pode ser dividido em três grandes estágios conceituais: aplicação, geometria e rasterização [1].

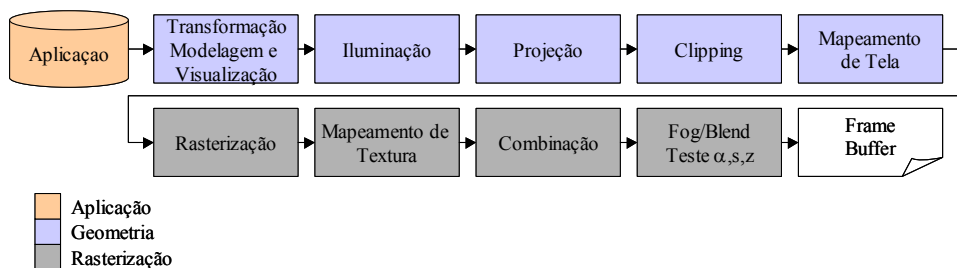


Figura 2 - Estágios conceituais e funcionais do pipeline convencional

Cada estágio conceitual pode ser segmentado em estágios funcionais, que serão detalhados nas seções seguintes.

2.1.1 Aplicação

O estágio de aplicação é controlado, como o nome diz, pela aplicação. Nesta fase são realizadas tarefas como detecção de colisão, animações e a atualização do mundo virtual de forma geral. Ao final deste estágio a geometria a ser renderizada é passada para o estágio de geometria. Este estágio é usado para a implementação de diversas técnicas de aceleração, tais como multi-resolução, *culling* e portais. Essas técnicas reduzem a quantidade de primitivas enviadas à geometria, afetando profundamente o desempenho do processo de renderização.

2.1.2 Geometria

O estágio de geometria é responsável pelas operações sobre os vértices de um modelo enviados pela aplicação. Este estágio pode ser subdividido em cinco estágios funcionais, como ilustrado na Figura 2.

No caminho até a tela, um modelo é transformado por uma seqüência de operações, caracterizando diferentes sistemas de coordenadas. Um modelo está originalmente em coordenadas do chamado espaço de modelo. Cada modelo da cena virtual pode ser associado a uma transformação de modelagem que irá posicioná-lo no mundo virtual. Esta transformação é aplicada a todos os vértices e normais, levando ao espaço do mundo, comum a todos os modelos da cena.

A câmera virtual está posicionada no espaço do mundo. Para facilitar as etapas de iluminação, projeção e *clipping* esta câmera e todos os modelos são transformados pela transformação de visualização. O objetivo desta transformação é posicionar a câmera na origem e alinhá-la com os eixos ortogonais. Este espaço é denominado espaço da câmera ou espaço do olho.

O próximo estágio do pipeline é a iluminação. Uma cena pode conter uma ou mais luzes que poderão afetar o cálculo da iluminação. Para cada vértice é calculada uma cor através de uma equação de iluminação. Esta equação procura simplificar de forma significativa o modelo real (físico) de iluminação. As variáveis consideradas nesta equação são as posições e propriedades das fontes de luz, a posição e a normal do vértice, propriedades do material a que o vértice

pertence e a posição do observador. O cálculo da iluminação é feito no espaço do olho.

O estágio funcional de projeção transforma o volume de visualização em um cubo unitário denominado volume canônico de visualização. Há basicamente dois métodos de projeção: ortográfica e perspectiva.

A etapa de *clipping* tem como função selecionar as primitivas que passarão para o estágio seguinte. Primitivas que estão completamente inseridas no volume de visualização são mantidas e aquelas completamente fora são descartadas. As primitivas que estão parcialmente dentro do volume precisam ser recortadas.

O mapeamento de tela transforma as coordenadas das primitivas para o espaço de tela. Isto exige tipicamente uma translação seguida de uma escala. A coordenada *z* é mapeada para uma coordenada do *z-buffer*, e as novas coordenadas *x* e *y* são chamadas coordenadas de tela. Toda esta informação é repassada para o estágio de rasterização.

2.1.3 Rasterização

A rasterização é o processo de converter vértices projetados, com informações de cor, coordenadas de textura, etc., em fragmentos que correspondem a pixels na tela. Diferentemente do estágio de geometria, que realiza operações por vértice, o estágio de rasterização realiza operações por fragmento.

O primeiro passo da rasterização é produzir um conjunto de fragmentos a partir das primitivas renderizadas. Um fragmento pode ser visto como um pixel com informações adicionais, como profundidade e coordenadas de textura. Cada tipo de primitiva, ponto, linha ou triângulo, possui um método distinto de rasterização.

Sobre todo fragmento produzido são realizadas operações a fim de determinar a cor final do pixel correspondente. Entre estas operações está o acesso a texturas e operações aritméticas para combinar as diversas informações do fragmento.

Também é no estágio da rasterização que é executado o principal algoritmo de visibilidade, o *z-buffer*. Um fragmento só é aproveitado caso ele tenha uma informação de profundidade, em geral, inferior àquela presente em um *buffer* especialmente mantido para este propósito, ou seja, somente os fragmentos mais

próximos do observador são desenhados. Além do *z-buffer* há uma diversidade de outros *buffers* que auxiliam a elaboração de algoritmos que precisam armazenar informações de cada fragmento.

Este é o modelo básico de pipeline adotado pelos sistemas gráficos modernos. Sua implementação pode ser feita inteiramente em *software* mas para desempenho de geração de imagens em tempo real grande parte do pipeline é hoje implementado em hardware. Tipicamente somente o estágio de aplicação é implementado em software e todo o resto é em hardware.

2.2 API's

Para utilizar as funcionalidades do hardware é disponibilizada um API para a aplicação. Um determinado hardware pode prover uma API inteiramente exclusiva para aquele sistema; entretanto há hoje duas APIs que atingiram um alto grau de popularidade – OpenGL [2] e DirectX [3]. Estas API's foram adotados tanto por programadores de aplicações gráficas quanto por desenvolvedores de hardware.

OpenGL é uma biblioteca gráfica multi-plataforma que nasceu no início da década de 90 nos laboratórios da Silicon Graphics como uma evolução da IRIS GL [4] e hoje se encontra na versão 1.4. O principal objetivo desta API é prover um acesso às funcionalidades gráficas do hardware de maneira independente de dispositivo.

O modelo de execução do OpenGL é um pipeline de topologia fixa similar àquele apresentado na seção anterior. Um conjunto de variáveis de estado controlam o funcionamento deste pipeline.

O DirectX é uma biblioteca criada pela Microsoft principalmente para o desenvolvimento de jogos na plataforma Windows. Ela é composta por diversos subsistemas que abrangem as funcionalidades de um jogo, como som, gráficos, dispositivos de entrada/saída e conectividade. A componente de especial interesse neste trabalho é o DirectX Graphics, responsável pela renderização tridimensional e bidimensional. A partir deste ponto qualquer referência ao termo DirectX considera apenas à componente gráfica DirectX Graphics.

A versão 8.1 do DirectX adotou o conceito de pipeline programável, que será discutido no capítulo seguinte. Entretanto ainda há a opção de se utilizar um pipeline convencional como o descrito anteriormente.

2.3 Extensibilidade

Uma característica importante de uma biblioteca gráfica é a sua capacidade de extensão para incorporar novas funcionalidades do hardware gráfico. Ortogonalidade da API é um fator favorável para esta extensibilidade.

Modelos diferentes de extensibilidade foram adotados pela bibliotecas gráficas em análise: OpenGL e DirectX.

2.3.1 OpenGL

O OpenGL é hoje administrado por um conselho denominado *Architecture Review Board* (ARB), que se reúne periodicamente para definir os caminhos de evolução da biblioteca. Este conselho é formado por membros da indústria de hardware e software, como NVIDIA, ATI, SGI e Microsoft.

A biblioteca OpenGL foi projetada desde o início visando a extensibilidade e provê para isso um mecanismo específico. O processo típico de extensão é descrito a seguir.

Um fabricante de hardware projeta uma nova funcionalidade e deve expô-la de alguma forma para a aplicação. Este redige então uma extensão, que é um documento que descreve como a especificação original do OpenGL é modificada com a introdução da nova funcionalidade. Novas funções e definições podem ser adicionadas e geralmente há uma relação de perguntas e respostas de como a extensão deve funcionar.

Quando uma extensão atinge um grau de maturidade maior, e é adotada por diversos fabricantes, esta pode ser enviada para análise do ARB para se tornar uma extensão “oficial” ou até mesmo parte do núcleo do OpenGL. O registro de todas as extensões existentes é mantido pela SGI e disponível online [5].

Uma aplicação que deseja usar uma extensão deve checar, em tempo de execução, se a extensão é implementada pelo hardware ou pelo *driver* da plataforma. Em caso afirmativo a aplicação deve obter referências para as novas

funções introduzidas pela extensão através de um mecanismo dependente da plataforma utilizada.

2.3.2 DirectX

A biblioteca DirectX foi criada e é mantida somente por uma empresa, a Microsoft. Desta forma o controle das modificações e evoluções é completamente centralizado. A solução adotada pela Microsoft para suportar novas funcionalidades emergentes é lançar uma nova versão da biblioteca que redefine a API's incluindo estas novas funcionalidades.

A tarefa dos fabricantes de hardware é implementar *drivers* que suportem a maior versão possível da biblioteca.

A aplicação que deseja utilizar o DirectX deve primeiramente obter uma referência para alguma versão da biblioteca, ou seja, a aplicação pode tentar obter uma referência para a versão 8.1 ou em caso de falha tentar obter uma referência para uma versão inferior. Outra forma de verificar as funcionalidades suportadas por um determinado hardware é pela consulta de um conjunto de *capabilities*.

O modelo de extensibilidade do OpenGL é interessante e tem sido efetivo na tarefa de garantir o acesso às novas funcionalidades. Entretanto o número de extensões existentes atualmente é muito grande, aproximadamente 300, e tem aumentado a cada dia. Várias extensões são correlacionadas, o que torna a manutenção e a administração muito difícil.

O modelo do DirectX é mais rígido mas não impediu a evolução da biblioteca, que hoje se encontra na versão 8.1, e com planos para as versões 9 [6] e 10. Entretanto uma nova versão da biblioteca pode trazer incompatibilidade com a versão anterior, dificultando muito o trabalho da aplicação.

As extensões ou versões expõem novas funcionalidades mas nunca atenderão a demanda contínua de novos recursos. Para solucionar este problema o funcionamento interno da máquina de renderização foi exposto para a aplicação, ou seja, o hardware gráfico passa a ser diretamente programado pela aplicação.

O capítulo seguinte descreve o modelo básico de hardware programável adotado pela indústria de hardware.

3 Pipeline programável

Para dar maior controle sobre o funcionamento do hardware gráfico o pipeline tem se tornado programável. Isto significa que além de configurar um estado de operação do pipeline, a aplicação pode especificar um programa que será executado em algum estágio específico do processamento.

Tanto o DirectX quanto o OpenGL têm evoluído no sentido de expor esta programabilidade para a aplicação. Será apresentado a seguir o modelo básico de pipeline programável, destacando os estágios do pipeline que podem ser substituídos por um programa desenvolvido pela aplicação.

3.1 Arquitetura

Este novo modelo de pipeline de renderização oferece programabilidade em basicamente dois pontos: o processamento de vértices e o processamento de fragmentos. Nas implementações atuais a aplicação pode optar por utilizar o processamento convencional ou especificar um programa para cada um dos estágios independentemente.

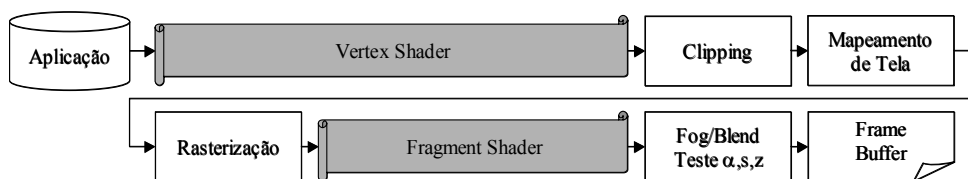


Figura 3 - Pipeline Programável

3.2 Processamento de Vértices

Como visto anteriormente cada vértice enviado para o pipeline de renderização deve sofrer uma série de transformações até corresponder a um pixel

na tela, além de ser iluminado. Estas transformações podem ser feitas pelo caminho convencional ou calculadas por um programa escrito pela aplicação e enviado para o processador gráfico, denominado *programa de vértice* [7].

O programa de vértice é uma seqüência de instruções de baixo nível de abstração que operam sobre conjuntos de registradores. Cada registrador é um vetor de quatro *floats*. Grandezas escalares ocupam uma parte de um registrador e matrizes ocupam seqüências de registradores. O modelo básico de processador de vértices é ilustrado na Figura 4.

O programa de vértice será invocado automaticamente para todo vértice que é enviado para o pipeline e substitui um conjunto de tarefas que antes eram realizadas pelo pipeline convencional:

- Transformação de modelagem e visualização
- Geração de coordenadas de textura
- Projeção
- Iluminação

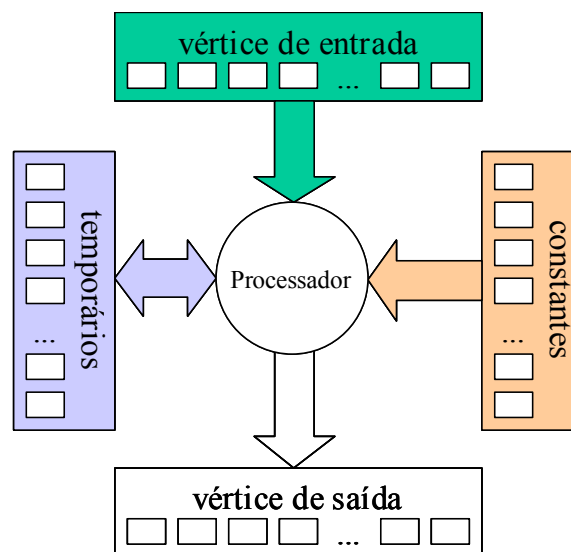


Figura 4 - Processador de Vértice

O processador de vértices recebe como entrada um conjunto de atributos relacionados a um vértice, armazenados em registradores. Estes registradores somente podem ser lidos pelo programa, não podem ser escritos. O número de atributos disponíveis é limitado pelo hardware específico.

A semântica de cada atributo do vértice é determinada pelo programa que irá executar no processador. Entretanto, o vértice tipicamente irá conter as seguintes informações:

- Posição, em coordenadas de modelo;
- Normal, em coordenadas de modelo e não necessariamente normalizada;
- Cores, primária e secundária;
- Fator de *fog*;
- Um conjunto de coordenadas de textura.

O programa deve obrigatoriamente produzir um vértice contendo as seguintes informações:

- Posição, em coordenadas do espaço de *clipping*;
- Cores, primária e secundária;
- Coordenadas de textura.

Para realizar as operações necessárias o programa conta com um conjunto de registradores temporários com permissão de escrita e leitura. O número de registradores disponíveis é também determinado pelo hardware.

Há ainda um conjunto de registradores fundamentais para o funcionamento do processador. Para produzir o vértice de saída o programa precisa de diversas informações do estado atual da máquina de renderização. Para calcular a posição do vértice, por exemplo, o programa precisa do valor das matrizes de modelagem, visualização e projeção. Estes valores são acessados através dos registradores constantes, que podem ser lidos pelo programa mas não podem ser escritos.

Os valores dos registradores constantes podem ser atualizados pela aplicação de forma explícita ou implícita, mas sempre antes do programa do vértice ser invocado.

O programador tem à sua disposição um conjunto reduzido de instruções para realizar os cálculos necessários. Normalmente não há instruções para controle de fluxo, o que significa que os programas não tem suporte a condicionais e construções de laços.

O DirectX 8.1 define uma linguagem de baixo nível, o Vertex Shader 1.1 [3]. O OpenGL possui algumas extensões que expõem a programabilidade de formas diferentes. A primeira extensão foi definida pela NVIDIA [8], que foi a primeira empresa a comercializar processadores gráficos programáveis. A segunda extensão foi proposta e implementada pela ATI [9] em parceria com a Matrox. Mais recentemente o ARB chegou a um acordo e definiu uma extensão

comum, que futuramente deve ser implementada [10]. A versão 9.0 do DirectX, ainda em desenvolvimento, promete a versão 2.0 do Vertex Shader e a NVIDIA já prepara a segunda versão de sua extensão para expor as funcionalidade de seu novo processador NV30, ainda em desenvolvimento [11]. Cada uma dessas API's serão descritas nas seções seguintes.

3.2.1 Vertex Shader 1.1

A linguagem definida pelo DirectX segue fielmente o modelo de processador de vértices apresentado na seção anterior. O programa, denominado *shader*, é definido por uma seqüência de instruções operando sobre registradores e pode estar definido em uma *string* na aplicação ou em um arquivo. O primeiro passo da aplicação é compilar o *shader* e transformá-lo para um formato binário através da chamada a uma das funções:

```
D3DXAssembleShader (strCode, strlen(strCode), 0, NULL, &pCode, NULL);
D3DXAssembleShaderFromFile ("shader.vs", 0, NULL, &pCode, NULL);
```

Uma vez criada a representação binária o shader pode ser criado, apagado e associado ao pipeline de renderização através das funções:

```
IDirect3DDevice8::CreateVertexShader
IDirect3DDevice8::DeleteVertexShader
IDirect3DDevice8::SetVertexShader
```

Os registradores de entrada e de saída que podem ser utilizados no programa são enumerados nas Tabelas 1 e 2.

Nome	Quantidade (mínima)	Permissão	Descrição
<i>an</i>	1 escalar	escrita/uso	endereçamento
<i>c[n]</i>	96 vetores	somente leitura	constantes
<i>rn</i>	12 vetores	leitura/escrita	temporários
<i>vn</i>	16 vetores	somente leitura	atributos do vértice

Tabela 1 - Vertex Shader - registradores de entrada

Nome	Quantidade (mínima)	Permissão	Descrição
<i>oDn</i>	2 vetores	somente escrita	cor
<i>oFog</i>	1 escalar	somente escrita	fator de <i>fog</i>

oPos	1 vetor	somente escrita	posição
oPts	1 escalar	somente escrita	tamanho de ponto
oTn	4 vetores	somente escrita	coordenadas de textura

Tabela 2 - Vertex Shader - registradores de saída

As instruções disponíveis são listadas na Tabela 3

Nome	Descrição
add	Soma
dp3	Produto interno de três componentes
dp4	Produto interno de quatro componentes
dst	Vetor distância
expp	Exponencial
lit	Coefficientes de iluminação
logp	Logaritmo
mad	Multiplicação e soma
Max	Máximo
min	Mínimo
mov	Cópia
mul	Multiplicação
rcp	Recíproco
rsq	Recíproco da raiz quadrada
sge	<i>Set</i> para maior que
slt	<i>Set</i> para menor que
sub	Subtração

Tabela 3 - Vertex Shader - instruções

A Tabela 4 mostra um exemplo de *vertex shader*. Esse exemplo transforma o vértice para o espaço de *clipping* e copia uma coordenada de textura. Além disso calcula a cor do vértice usando um modelo de iluminação com componentes difusa e especular. Diversos parâmetros são carregados nos registradores constantes, como matrizes de transformação e características de material.

A grande quantidade de comentários que se faz necessária ilustra a falta de clareza do código do *shader*.

```
; Transforms a vertex to homogenoeous clip space,
; copies texture coordinates, and lights it.
```

```

;
;
; MEMORY MAP
;
; Each vertex ultimately needs to write the following output registers:
; oPos    -- the vertex's clip-space coordinates
; oT0     -- the vertex's uv-coordinates
; oD0     -- the vertex's diffuse color
;
; Each vertex comes into the vertex shader with the following read-only data:
; v0 -- vertex's model space coordinates are in x, y, z;
;      w defaults to 1.0
; v1 -- vertex's normal in model-space in x, y, z;
;      w defaults to 1.0
; v2 -- vertex's u, v coordinates are in x, y
;      z defaults to 0.0
;      w defaults to 1.0
;
; Constant memory is set in Quad.cpp to contain:
; c0-c3 -- contains matrix to take model-space
;         coordinates to clip-space coordinates
; c4    -- contains normalized light vector in model-space coordinates;
;         the vector points towards the light
; c5    -- contains normalized halfway vector in model-space coordinates;
;         the vector points halfway between the eye and the light
; c6    -- contains the material color
; c7    -- contains constants (0.0f, 10.0f, 50.0f, 100.0f);
;         use to get specular power constant, but do not use zero as a
specular power
; c8    -- contains the normalized vector that points to the eye in
;         model-space coordinates
; c10   -- contains the color of the light

vs.1.0

; Transform position to clip space
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

; if the vertex is back-facing; then make it black!
dp3 r5, v1, c8
sge r5, -r5, -c7.x ; make it 0 for negative and zero values (1 for others

mul oD0, r5, c6

; Output texture coordinates
mov oT0, v2

```

```

; compute modified material color that includes back-face cull
mul r5, r5, c6

; compute DIFFUSE part:
; Dot normal with light direction in model space
dp3 r0, v1, c4
max r0, r0, c7.x      ; zero out negative values

; multiply diffuse component by material color and store
mul r0, r0.x, r5

; compute SPECULAR part
; Dot normal with halfway vector
dp3 r1, v1, c5
max r1, r1, c7.x      ; zero out negative values

; raise it to the specular index
mov r1.w, c7.z
lit r2, r1

; multiply specular component by material color and
; add the diffuse color
mad r1, r2.z, r5, r0

; multiply light color with material color
mul oD0, r1, c10

```

Tabela 4 - Vertex Shader - exemplo

3.2.2 NV_vertex_program

Essa extensão, desenvolvida pela NVIDIA para o OpenGL, é muito semelhante à proposta do *Vertex Shader* do DirectX. As semelhanças entre esta extensão e o *vertex shader* não são por acaso, já que foram criadas para dar acesso às funcionalidades do mesmo hardware. Serão descritas aqui somente as diferenças marcantes entre estas duas linguagens de baixo nível.

A primeira diferença é a terminologia empregada. *Vertex shader* aqui é chamado de *vertex program* e registradores constantes são parâmetros de programa.

O conjunto de registradores é o mesmo, entretanto o nome utilizado no código do *vertex program* é diferente. Também continua o mesmo conjunto de instruções.

Uma funcionalidade interessante introduzida por essa extensão é o acompanhamento automático do estado das matrizes do OpenGL. A aplicação pode informar que uma determinada matriz de transformação, sempre que alterada, deve ter seu valor copiado automaticamente para uma dada seqüência de 4 registradores constantes.

A tabela abaixo ilustra um exemplo de programa escrito na linguagem definida por esta extensão. Esse exemplo transforma o vértice e calcula a iluminação do vértice usando um modelo de iluminação difusa e especular considerando a luz e o observador no infinito.

```

!!VP1.0
#
# c[0-3] = modelview projection (composite) matrix
# c[4-7] = modelview inverse transpose
# c[32] = normalized eye-space light direction (infinite light)
# c[33] = normalized constant eye-space half-angle vector (infinite viewer)
# c[35].x = pre-multiplied monochromatic diffuse light color & diffuse material
# c[35].y = pre-multiplied monochromatic ambient light color & diffuse material
# c[36] = specular color
# c[38].x = specular power
#
# outputs homogenous position and color
#
DP4 o[HPOS].x, c[0], v[OPOS];
DP4 o[HPOS].y, c[1], v[OPOS];
DP4 o[HPOS].z, c[2], v[OPOS];
DP4 o[HPOS].w, c[3], v[OPOS];
DP3 R0.x, c[4], v[NRML];
DP3 R0.y, c[5], v[NRML];
DP3 R0.z, c[6], v[NRML]; # R0 = n' = transformed normal
DP3 R1.x, c[32], R0; # R1.x = Lpos DOT n'
DP3 R1.y, c[33], R0; # R1.y = hHat DOT n'
MOV R1.w, c[38].x; # R1.w = specular power
LIT R2, R1; # Compute lighting values
MAD R3, c[35].x, R2.y, c[35].y; # diffuse + emissive
MAD o[COL0].xyz, c[36], R2.z, R3; # + specular
END

```

Tabela 5 - NV_vertex_program - exemplo

Nessa extensão, programas de vértices são administrados pela aplicação de forma semelhante às texturas. Cada programa é associado a um identificador que pode ser criado por uma chamada à função:

```
glGenProgramsNV (1, &id);
```

Para ativar um programa a aplicação deve chamar a função abaixo, passando como parâmetro o identificador criado.

```
glBindProgramNV (GL_VERTEX_PROGRAM_NV, id);
```

Em seguida a aplicação deve carregar o código do programa, descrito por uma string ASCII:

```
glLoadProgramNV (GL_VERTEX_PROGRAM_NV, id, strlen(code), code);
```

Assim como para o caso de texturas há métodos para apagar programas e para verificar se um dado programa está residente na memória do processador.

3.2.3 EXT_vertex_shader

A ATI desenvolveu uma extensão para OpenGL bastante diferente das duas propostas vistas anteriormente. A primeira grande diferença é a forma de se especificar o código do *vertex shader*. Ao invés de ser carregado na forma de uma *string* o código é definido de forma procedural, via uma API definida pela extensão.

Esta abordagem possui vantagens e desvantagens. Uma vantagem é a possibilidade de organizar o código do *shader* em procedimentos ou utilizar laços da linguagem C para criar o programa. Uma desvantagem é a dificuldade adicional para se carregar códigos dinamicamente.

A tabela abaixo ilustra um *shader* escrito para essa extensão. Ele transforma o vértice e calcula a iluminação difusa de um vértice, combinando com a cor de um material.

```
/* Initialize global parameter bindings */
Modelview = glBindParameterEXT (GL_MODELVIEW_MATRIX);
Projection = glBindParameterEXT (GL_PROJECTION_MATRIX);
Vertex = glBindParameterEXT (GL_CURRENT_VERTEX_EXT);
Normal = glBindParameterEXT (GL_CURRENT_NORMAL_EXT);
glBindVertexShaderEXT (xform); //a simple diffuse shader
glBeginVertexShaderEXT ();
{
float direction[4] = { 0.57735f, 0.57735f, 0.57735f, 0.0f}; //direction vector
(1,1,1) normalized
float material[4] = { 1.00000f, 1.00000f, 0.00000f, 1.0f}; //yellow diffuse
material
float ambient[4] = { 0.20000f, 0.20000f, 0.20000f, 0.0f}; //scene ambient light
intensity
```

```

GLuint lightDirection;
GLuint diffMaterial;
GLuint sceneAmbient;
GLuint eyeVertex;
GLuint clipVertex;
GLuint eyeNormal;
GLuint intensity;
/* generate local values */
eyeVertex = glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);
clipVertex = glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);
eyeNormal = glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);
intensity = glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);
/* generate constant values */
lightDirection = glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_CONSTANT_EXT,
GL_FULL_RANGE_EXT, 1);
diffMaterial = glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_CONSTANT_EXT,
GL_FULL_RANGE_EXT, 1);
sceneAmbient = glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_CONSTANT_EXT,
GL_FULL_RANGE_EXT, 1);
glSetLocalConstantEXT (lightDirection, GL_FLOAT, direction);
glSetLocalConstantEXT (diffMaterial, GL_FLOAT, material);
glSetLocalConstantEXT (sceneAmbient, GL_FLOAT, ambient);
glShaderOp2EXT (GL_OP_MULTIPLY_MATRIX_EXT, eyeVertex, Modelview, Vertex);
glShaderOp2EXT (GL_OP_MULTIPLY_MATRIX_EXT, GL_OUTPUT_VERTEX_EXT, Projection,
eyeVertex);
/* assumes no scaling/shearing in modelview matrix */
glShaderOp2EXT (GL_OP_MULTIPLY_MATRIX_EXT, eyeNormal, Modelview, Normal);
glShaderOp2EXT (GL_OP_DOT3_EXT, intensity, lightDirection, eyeNormal);
glShaderOp2EXT (GL_OP_ADD_EXT, intensity, sceneAmbient, intensity);
glShaderOp2EXT (GL_OP_MUL_EXT, GL_OUTPUT_COLOR0_EXT, diffMaterial, intensity);
}
glEndVertexShaderEXT ();

```

Tabela 6 - EXT_vertex_shader – exemplo

Uma funcionalidade adicional dessa extensão é a capacidade de acompanhar automaticamente o valor de qualquer variável de estado do contexto OpenGL. Isto inclui as matrizes de transformação, o estado associado às luzes, materiais e texturas.

Essa extensão abstrai o conceito de registradores e define variáveis de três tipos: escalares, vetores e matrizes. Estas variáveis são identificadores instanciados pela aplicação e podem ser de quatro tipos:

- *variant*
- *invariant*
- *local constant*
- *local*

Variants correspondem aos registradores de entrada com os atributos de um vértice. Tem esse nome pois seu valor varia para cada vértice processado.

Invariants são usados para armazenar valores que não variam para todo vértice. Variáveis deste tipo são válidas para todo *shader* em uso pela aplicação. *Local constants* são semelhantes aos invariantes mas somente podem ser usados pelo *shader* em que foi especificado. Este tipo de variável não está disponível nas extensões apresentadas anteriormente. *Invariants* e *local constants* são equivalentes aos registradores constantes, também chamados de parâmetros de programa.

As variáveis do tipo *local* podem ser associadas aos registradores temporários da máquina de processamento de vértices.

Os registradores de saída da máquina de processamento de vértice são acessados através de identificadores pré-definidos pela extensão.

A extensão `EXT_vertex_shader` foi definida pela ATI mas também é suportada por um hardware da Matrox [12].

3.2.4 ARB_vertex_program

As extensões disponibilizadas pela NVIDIA e ATI conseguem expor a funcionalidade do hardware, entretanto a aplicação precisa implementar duas soluções caso queira oferecer suporte à ambos os hardware.

Pensando na padronização de uma linguagem de baixo nível para programação do processador de vértice, os diversos participantes do conselho OpenGL chegaram a um acordo de uma extensão. Esta nova extensão aproveita idéias tanto da `NV_vertex_program` quanto da `EXT_vertex_shader`.

O programa de vértice é descrito de forma textual e carregado pela aplicação dinamicamente, como na extensão da NVIDIA. Um programa tem tipicamente um cabeçalho que define as variáveis que serão utilizadas nas instruções. Estas variáveis não tem um nome fixo, como acontece nas extensões vistas anteriormente. Há variáveis de cinco tipos, cada um associado a um tipo de registrador:

- `ATTRIB`, atributos do vértice;
- `PARAM`, parâmetros de programa;
- `TEMP`, temporárias;
- `ADDRESS`, variável de endereçamento;

- OUTPUT, atributos do vértice de saída.

A idéia de acompanhar o estado OpenGL automaticamente foi adotado aqui. As variáveis do tipo PARAM especificam a variável de estado a que estão relacionadas.

O código apresentado na Tabela 7 transforma o vértice de entrada e calcula a cor do vértice por um modelo de iluminação difusa e especular.

```

!!ARBvp1.0
ATTRIB iPos          = vertex.position;
ATTRIB iNormal       = vertex.normal;
PARAM  mvinv[4]      = { state.matrix.modelview.invtrans };
PARAM .mvp[4]        = { state.matrix.mvp };
PARAM  lightDir      = state.light[0].position;
PARAM  halfDir       = state.light[0].half;
PARAM  specExp       = state.material.shininess;
PARAM  ambientCol    = state.lightprod[0].ambient;
PARAM  diffuseCol    = state.lightprod[0].diffuse;
PARAM  specularCol   = state.lightprod[0].specular;
TEMP   xfNormal, temp, dots;
OUTPUT oPos          = result.position;
OUTPUT oColor        = result.color;

# Transform the vertex to clip coordinates.
DP4   oPos.x,.mvp[0], iPos;
DP4   oPos.y,.mvp[1], iPos;
DP4   oPos.z,.mvp[2], iPos;
DP4   oPos.w,.mvp[3], iPos;

# Transform the normal to eye coordinates.
DP3   xfNormal.x, mvinv[0], iNormal;
DP3   xfNormal.y, mvinv[1], iNormal;
DP3   xfNormal.z, mvinv[2], iNormal;

# Compute diffuse and specular dot products and use LIT to compute
# lighting coefficients.
DP3   dots.x, xfNormal, lightDir;
DP3   dots.y, xfNormal, halfDir;
MOV   dots.w, specExp.x;
LIT   dots, dots;

# Accumulate color contributions.
MAD   temp, dots.y, diffuseCol, ambientCol;
MAD   oColor.xyz, dots.z, specularCol, temp;
MOV   oColor.w, difuseCol.w;
END

```

Tabela 7 - ARB_vertex_program - exemplo

Esta extensão define um conjunto de 27 instruções, um número maior do que aquele disponível existente anteriormente, mas ainda não há instruções para controle de fluxo.

3.2.5 Vertex Shader 2.0 e NV_vertex_program2

Já estão em desenvolvimento as novas versões de linguagens de baixo nível de abstração. A versão 2.0 do *vertex shader* fará parte da versão 9.0 do DirectX. A NVIDIA tem trabalhado também na segunda versão de sua extensão para processamento de vértices, NV_vertex_program2. Esta extensão será suportada pelo novo processador NV30 e poderá ser emulada nos processadores atuais.

A primeira evolução será o aumento do número máximo de instruções que podem ser executadas e do número de registradores disponíveis.

A segunda evolução, mais esperada, é o suporte a instruções de controle de fluxo, o que significa a possibilidade de laços e condicionais. Não há mais detalhes destas evoluções até a presente data.

3.3 Processamento de Fragmentos

O processador de fragmentos é responsável por combinar todas as informações de um fragmento para produzir uma cor e opcionalmente alterar a profundidade z.

Os atributos dos fragmentos gerados pelo rasterizador podem ter sido calculados a partir de atributos de um vértice vindo do pipeline convencional ou de um vértice produzido pelo processador de vértices.

Um fragmento possui tipicamente duas cores e um conjunto de coordenadas de textura. As coordenadas de textura são usadas para buscar informações das texturas associadas. Em seguida, efetuam-se operações matemáticas entre essas diversas informações. O produto de uma operação pode vir a servir como coordenada de textura para um novo endereçamento. Esta operação é denominada leitura de textura dependente.

Diferentemente do processamento de vértices, não há um modelo consolidado de processador de fragmentos.

O modelo implementado pelo DirectX é denominado *Pixel Shader* e hoje está na versão 1.4. Essa versão, lançada há alguns meses, é bem mais flexível e

ortogonal do que as anteriores. Está em desenvolvimento a versão 2.0, que deverá fazer parte do DirectX 9.0.

A NVIDIA implementa um modelo de processamento de fragmentos separado em duas partes, com uma extensão definida para cada parte, NV_texture_shader e NV_register_combiners. A primeira é responsável por fazer os acessos às texturas a partir das coordenadas de textura do fragmento, incluindo operações de leitura dependente. A segunda extensão é responsável por combinar as informações do fragmento para produzir a cor final.

Este modelo está sendo substituído por outro mais flexível e ortogonal a ser especificado na futura extensão NV_fragment_program.

A ATI apresenta a extensão ATI_fragment_shader que adota um modelo semelhante ao Pixel Shader 1.4, já que foi implementado para dar suporte ao mesmo hardware.

Está em desenvolvimento a extensão ARB_fragment_program, que promete consolidar um modelo de programação do processador de fragmentos.

4 Abstração

O modelo de programação proposto pela indústria e ilustrado na seção anterior é um primeiro passo, mas apresenta diversos problemas para que a programabilidade atinja um nível maior de popularidade entre os desenvolvedores.

O primeiro problema a ser destacado é o baixo nível de abstração da linguagem, exigindo do programador o trabalho de administrar diversos conjuntos de registradores e utilizar um conjunto reduzido de instruções de forma otimizada. A clareza do código em linguagem de máquina é muito prejudicada, principalmente devido à falta de identificadores com nomes que tenha alguma semântica. A flexibilidade do código também é muito baixa, dificultando uma modificação, por menor que seja, do algoritmo implementado.

A falta de recursos na linguagem também é um problema para o reuso dos programas desenvolvidos. A inexistência de funções impossibilita a criação de bibliotecas para realização de tarefas tradicionais, como iluminação ou geração de coordenadas de textura segundo um determinado modelo ou método. A falta de laços não permite a elaboração de um código que funcione para um número genérico de luzes, por exemplo.

O terceiro problema é a falta de padronização de uma linguagem comum compatível com a diversidade de hardware existente no mercado. Se o desenvolvedor quiser utilizar os novos recursos de programabilidade precisará implementar uma versão para cada hardware disponível no mercado. Para sanar esta deficiência, o conselho OpenGL aprovou a extensão `ARB_vertex_program` para processamento de vértices, já descrita no capítulo anterior. Ainda está em desenvolvimento a extensão para processamento de fragmentos (`ARB_fragment_program`), que provavelmente será apresentada como parte da versão 1.5 do OpenGL [10].

Frente a estas dificuldades, surge a necessidade do desenvolvimento de uma linguagem de alto nível de abstração que facilite o trabalho do programador e que seja compatível com diferentes hardware.

Há diversas pesquisas e iniciativas no sentido de prover uma linguagem de alto nível única, algumas mais ambiciosas, outras mais realistas. A maior parte desses trabalhos é recente e ainda está em desenvolvimento. Um trabalho importante foi desenvolvido na universidade de Stanford por Proudfoot et al. [13]. O conselho OpenGL tem se preocupado com esta questão e estuda há um ano a versão 2.0 [14] da biblioteca. A NVIDIA por sua vez trabalha na linguagem Cg [15] e a Microsoft em uma linguagem de alto nível a ser incorporada no DirectX 9.0. Cada uma dessas propostas é apresentada nas seções seguintes.

4.1 Stanford Real-time Shading Language

A principal contribuição do trabalho de Proudfoot et al. [13] foi de formalizar uma abstração do pipeline programável, como apresentado anteriormente.

Um conceito importante introduzido foi a frequência de computação, estendendo e generalizando o modelo de processamento de vértices e fragmentos adotado pelo OpenGL e DirectX. São definidas quatro frequências de computação: constante, grupo de primitivas, vértice e fragmento.

A linguagem definida por este trabalho foi inspirada em grande parte pela linguagem Renderman [16]. Não foram implementadas, no entanto, funcionalidades que não estavam disponíveis no hardware. Não há, por exemplo, suporte a laços e condicionais dependentes de dados. Também não foram implementadas uma série de funcionalidades que não eram necessárias para a pesquisa em questão, como uma extensa biblioteca de funções pré-existentes. A sintaxe da linguagem também foi um pouco alterada para refletir termos e técnicas usadas em renderização em tempo real.

Os tipos de dados definidos são: escalares, vetores de 3 e 4 componentes, matrizes 3x3 ou 4x4, booleanos e um tipo especial para referência a texturas. Associados a esses tipos, há modificadores que definem a frequência de

computação a que uma dada variável pertence. A partir deste modificador a frequência de um determinado trecho de código é implicitamente determinada.

O sistema implementado funciona em cima do OpenGL, gerando código para as extensões NV_vertex_program e NV_register_combiners.

O código ilustrado na Tabela 8 faz o cálculo de iluminação com coeficientes difuso e especular. Pode-se notar a existência de algumas variáveis globais pré-definidas, como a normal do vértice N .

```
// Useful constants

constant float4 Zero = { 0, 0, 0, 0 };

surface float4
lightmodel_diffuse (float4 a, float4 d)
{
    perlight float diffuse = dot(N,L);
    perlight float4 fr = d * select(diffuse > 0, diffuse, 0);
    return a * Ca + integrate(fr * Cl);
}

surface float4
lightmodel_specular (float4 s, float4 e, float sh)
{
    perlight float diffuse = dot(N,L);
    perlight float specular = pow(max(dot(N,H),0),sh);
    perlight float4 fr = s * select(diffuse > 0, specular, 0);
    return integrate(fr * Cl) + e;
}

surface shader float4
bowling_pin_base ()
{
    float4 Cd = lightmodel_diffuse({ 0.4, 0.4, 0.4, 1 }, { 0.5, 0.5, 0.5, 1 });
    float4 Cs = lightmodel_specular({ 0.35, 0.35, 0.35, 1 }, Zero, 20);
    return { 120 / 255, 120 / 255, 120 / 255, 1 } * Cd + Cs;
}
```

Tabela 8 - Stanford Real-time shading language - exemplo

4.2 OpenGL 2.0

O conselho OpenGL identificou a necessidade de evolução da biblioteca, principalmente no sentido de prover uma forma de acesso aos processadores programáveis que seja independente de hardware com elevado nível abstração.

Além disso, a biblioteca tem se tornado cada vez mais complexa e difícil de ser administrada, pois há uma longa lista de extensões especializadas com diversas interdependências entre si.

Além de uma linguagem de alto nível independente de hardware, outros pontos são discutidos para uma nova versão 2.0, entre eles o gerenciamento de memória mais simples, ainda que poderoso, e um controle maior sobre o tempo de renderização. Entretanto destacamos aqui somente a primeira.

A iniciativa na elaboração dos primeiros documentos foi da 3Dlabs [14] mas hoje há um grupo de trabalho formado por membros do conselho responsável em discutir as evoluções e implementar os primeiros protótipos de *drivers* compatíveis com OpenGL 2.0. O plano de evolução é criar primeiramente extensões “GL2” para que depois estas sejam incorporadas ao núcleo da biblioteca.

O modelo de pipeline proposto para o OpenGL 2.0 é semelhante àquele apresentado na seção 3.1. Destacam-se aqui o processador de vértices e o de fragmentos. A linguagem de alto nível especificada é a mesma para ambos os processadores e foi baseada na linguagem C. OpenGL 2.0 define dois conceitos: *shader* e programa. *Shader* é o código fonte do programa a ser enviado para os processadores e programa é um container de *shaders*. O código abaixo ilustra o processo de criação, compilação, ligação e ativação de um *shader* .

```
// Create a shader (shaderID != 0 if success)
shaderA = glCreateShaderObjectGL2(shaderType shaderType);

// Load source code
glLoadShaderGL2(shaderA, numStrings, strings);
glAppendShaderGL2(shaderA, string);

// Compile (status = TRUE if success)
status = glCompileShaderGL2(shaderA);

// Get information string from compile
infolog = glGetInfoLogGL2(shaderA);

// Create a program object
programZ = glCreateProgramObjectGL2();

// Attach shader object(s)
status = glAttachShaderObjectGL2(programZ, shaderA);
status = glAttachShaderObjectGL2(programZ, shaderB);
```

```

// Link the program
status = glLinkProgramGL2(programZ);

// get information on link
infolog = glGetInfoLogGL2(programZ);
glGetObjectParameterfvGL2(programZ, GL_SHADER_RELATIVE_SIZE_GL2, size);

// Make program current
status = glUseProgramObjectGL2(programZ);

```

Tabela 9 - Exemplo de criação de um shader em OpenGL 2.0

A linguagem de *shading* define os seguintes tipos básicos de dados: float, vec2, vec3, vec4, mat2, mat3, mat4, int, bool e arrays. Estes tipos podem ser acompanhados pelos seguintes modificadores: const, attribute, uniform e varying. Vetores e matrizes podem ter suas componentes acessadas de forma natural, entretanto não oferece suporte a inversões de componentes.

O suporte a controle de fluxo é completo, suportando construções if-else, for, while, do-while, return, break, continue e chamada de funções.

É permitido a definição de funções, inclusive com a capacidade de sobrecarga. Os parâmetros são passados por referência e somente há um valor de retorno. As regras de escopo são as mesmas da linguagem C. Há uma biblioteca de funções pré-definidas como: *sin*, *cos*, *pow*, *sqrt*, *normalize*, *dot* e *texture*.

Os atributos do vértice sendo processado e do vértice sendo produzido são acessados através de variáveis globais pré-definidas e com um nome determinado prefixados por “gl_” (ex: gl_Vertex). Da mesma forma são disponibilizadas as matrizes e demais estados do OpenGL.

A listagem abaixo apresenta um exemplo de *vertex shader* OpenGL 2.0 que transforma o vértice e calcula a cor pelo modelo de iluminação com componente difusa e especular.

```

// Functions defined in next example.
void main (void)
{
    float normalDotVP, normalDotHalfVector, powerFactor;
    vec3 color, normal, diffuseIntensity, specularIntensity;
    vec4 position;

    // Transform vertex to clip space
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    normal = gl_NormalMatrix * gl_Normal;
    normalDotVP = min (0, dot (normal, gl_Light0[gl_kPosition]));

```

```

normalDotHalfVector = min (0, dot (normal, gl_Light0[gl_kHalfVector]));

if (normalDotVP == 0.0)
    powerFactor = 0.0;
else
    powerFactor = pow (normalDotHalfVector, gl_FrontMaterialSpecularExponent);

diffuseLight = gl_Light0[gl_kDiffuseIntensity] * normalDotVP;
specularLight = gl_Light0[gl_kSpecularIntensity] * powerFactor;

color = gl_Light0[gl_kAmbientIntensity] * gl_FrontMaterial[gl_kAmbient] +
        diffuseLight * gl_FrontMaterial[gl_kDiffuse] +
        specularLight * gl_FrontMaterial[gl_kSpecular];

gl_FrontColor = clamp (vec4 (color, gl_FrontMaterial[gl_kDiffuseAlpha]),
                      0, 1);
}

```

Tabela 10 - Exemplo de um vertex shader em OpenGL 2.0

Um protótipo parcial do primeiro driver OpenGL 2.0 foi implementado pela 3Dlabs para ser executado no hardware Wildcat VP870. A proposta apresentada para esta versão do OpenGL é bastante ambiciosa. Sua linguagem possui claramente funcionalidades que ainda não estão disponíveis nos hardwares da atualidade.

4.3 Cg

A iniciativa da NVIDIA foi criar uma linguagem de alto nível – Cg (C for Graphics) - baseada na linguagem C. O modelo de pipeline adotado por Cg é o mesmo daquele apresentado nas seções anteriores, com dois processadores programáveis, um para vértices e outro para fragmentos.

Para lidar com a heterogeneidade dos hardware, Cg define o conceito de perfil. Um perfil Cg define um subconjunto da linguagem Cg completa que é suportado em uma plataforma de hardware ou API específica. Atualmente os seguintes perfis são suportados: DirectX 8 vertex shaders, DirectX 8 pixel shaders, ARB_vertex_program, NV_vertex_program, NV_vertex_program2 e futuramente NV_fragment_program.

O ponto de entrada de um *vertex shader* ou de um *fragment shader* é uma função de nome livre que recebe como parâmetros dados de entrada e retornam os valores calculados. Dois tipos de entrada são definidos, variantes e uniformes. Os

primeiros são dados que são especificados para cada elemento processado, vértice ou fragmento. Uniformes são valores que não variam para cada elemento, como por exemplo uma matriz de transformação ou a posição de uma luz.

O código da Tabela 11, assim como vários apresentados anteriormente, transforma o vértice calcula a iluminação difusa e especular.

```
// define inputs from application
struct appin
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};
// define outputs from vertex shader
struct vertout
{
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
};

vertout main(appin In,
             uniform float4x4 ModelViewProj : C0,
             uniform float4x4 ModelViewIT : C4,
             uniform float4 LightVec)
{
    vertout Out;
    Out.HPosition = mul(ModelViewProj, In.Position);
    // transform normal from model-space to view-space
    float4 normal = normalize(mul(ModelViewIT, In.Normal).xyzz);
    // store normalized light vector
    float4 light = normalize(LightVec);
    // calculate half angle vector
    float4 eye = float4(0.0, 0.0, 1.0, 1.0);
    float4 half = normalize(light + eye);
    // calculate diffuse component
    float diffuse = dot(normal, light);
    // calculate specular component
    float specular = dot(normal, half);
    specular = pow(specular, 32);
    // blue diffuse material
    float4 diffuseMaterial = float4(0.0, 0.0, 1.0, 1.0);
    // white specular material
    float4 specularMaterial = float4(1.0, 1.0, 1.0, 1.0);
    // combine diffuse and specular contributions
    // and output final vertex color
    Out.Color0 = diffuse * diffuseMaterial + specular * specularMaterial;
    return Out;
}
```

Tabela 11 - Exemplo de um shader em Cg

A semântica dos atributos do vértice deve ser definida pelo *vertex program*. Dois mecanismos são disponibilizados, o primeiro de associação explícita do atributo a um nome pré-definido, e a segunda de associação implícita.

O uso de associação explícita possibilita uma flexibilidade maior de interoperabilidade entre aplicação, *vertex program* e *fragment program*.

A linguagem Cg suporta cinco tipos básicos de dados, entre eles *float* e *bool*. Há variações para vetores, como *float1*, *float2*, *float3*, *float4* e tipos estruturados e arrays.

Cg suporta estruturas de controle como chamadas de função, *if-else*, *while* e *for*. O perfil NV_vertex_program2 suporta instruções de controle de fluxo, portanto neste perfil laços *for* e *while* são completamente suportados. Para outros perfis estas estruturas de controle só podem ser usadas se o laço puder ser aberto. Recursão não é suportada. A extensão NV_vertex_program2 ainda não é suportada por nenhum hardware disponível no mercado.

A Microsoft tem trabalhado na versão 9.0 do DirectX, que deve incluir uma linguagem de alto nível de abstração, denominada *DirectX High Level Shading Language (HLSL)*. Uma versão beta da biblioteca está em testes e ainda não há documentação disponível em domínio público.

Paralelamente a estas iniciativas, começamos a investir na concepção e implementação de uma linguagem de alto nível de abstração, denominada *Lua Shading Language*, que é objeto deste trabalho e será apresentada em detalhe no capítulo seguinte.

5 Lua Shading Language

As propostas apresentadas no capítulo anterior criaram uma nova linguagem de alto nível de abstração específica para a implementação de *shaders*. Em alguns casos a linguagem criada foi inspirada em alguma já existente, como C ou Renderman, entretanto foi necessário implementar todo o *front-end* do compilador, como analisador sintático, semântico e um gerador de código intermediário.

Este trabalho propõe uma linguagem de programação de *shaders* sobre a linguagem Lua [17], denominada *Lua Shading Language*. Lua é uma linguagem de programação de sintaxe simples e que pode ser estendida para qualquer domínio específico. Novos tipos de dados podem ser definidos e operações sobre estes dados podem ser interceptadas através de um mecanismo de eventos.

Na *Lua Shading Language* cada *shader* é implementado por uma função. Todos os tipos de construções da linguagem Lua são permitidos, exceto condicionais e laços envolvendo objetos especializados da *Lua Shading Language*. No entanto o programador pode, por exemplo, utilizar laços para integrar cada contribuição de fonte de luz, e utilizar condicionais para parametrizar o código do *shader*. Funções auxiliares podem ser definidas para realizar o cálculo tradicional de iluminação, ou bibliotecas de geradores de coordenadas de textura podem ser construídas, possibilitando o reuso de código. Conceitualmente, chamadas de função são substituídas pelo código da função e laços são linearizados antes do código ser compilado para o hardware.

5.1 Tipos de Dados

Há somente três novos tipos especializados de dados: escalares, vetores e matrizes. Escalares são números reais, vetores são *arrays* de quatro escalares, e

matrizes são quadradas de dimensão 4. A prática tem revelado que estes tipos são suficientes para codificar os *shaders*.

Vetores podem ser indexados pela componente ([xyzw], [uvst] ou [rgba]) ou por um índice numérico de 1 a 4. Esta indexação cria um objeto do tipo escalar que mantém uma referência para o vetor correspondente. Um vetor pode ter suas componentes invertidas, operação denominada *swizzle* (ex: `vec.zxyw`). Esta inversão, assim como a indexação, cria um objeto que mantém uma referência para o vetor original.

As matrizes podem ser indexadas numericamente para se ter acesso a uma determinada linha (ex: `mvit[1]`). Esta indexação cria um objeto do tipo vetor que mantém uma referência para a matriz de origem.

As operações disponíveis sobre estes objetos são limitadas por aquelas oferecidas pelo hardware programável em uso.

5.2 Arquitetura

Os *shaders* são escritos em Lua. Cada estágio funcional, processamento de vértices ou processamento de fragmentos, é representado por uma função Lua. O ambiente global de Lua pode definir objetos que serão usados por estas funções, como constantes ou matrizes de transformação da máquina de renderização. Uma API em C foi definida para criar e gerenciar *shaders*, bem como acessar os valores dos objetos globais definidos.

A execução da função Lua do *shader* dispara um gerador de código que traduz o código Lua em linguagem de máquina que será enviada para o hardware gráfico ou em uma seqüência de chamadas de funções da biblioteca gráfica para configurar o *pipeline* de renderização. A aplicação é responsável por selecionar o *shader* correto para renderizar a cena e por carregar as texturas apropriadas.

A linguagem Lua oferece construções procedurais e descritivas, o que possibilita a combinação de código de *shading* em descrições de cenas, de forma semelhante ao Renderman.

5.3 Geração de código

A arquitetura de geração de código é ilustrada na Figura 5. A implementação atual gera código para executar sobre extensões OpenGL, mas nada impede a implementação para DirectX, bastando para tal exportar determinadas funções do DirectX para o ambiente Lua.

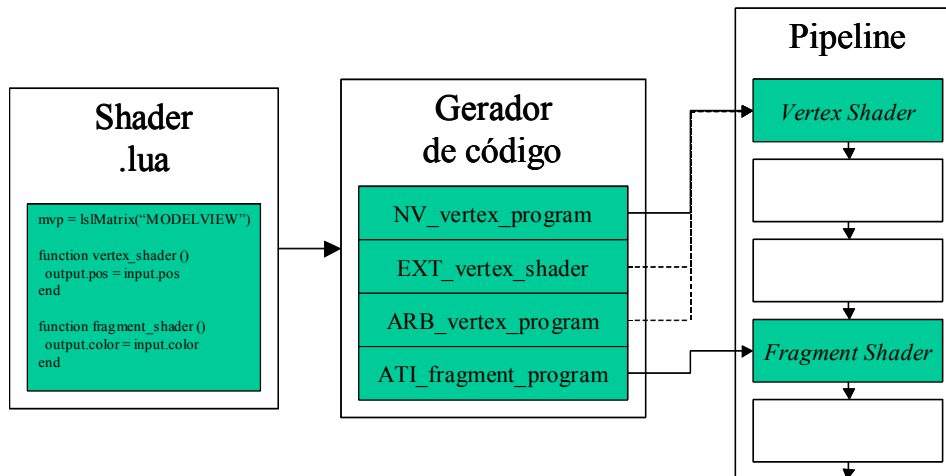


Figura 5 - Arquitetura de geração de código

As API's das extensões OpenGL para programação do hardware foram exportadas para Lua, assim o gerador de código, escrito em Lua, pode chamar funções para configurar o hardware.

O gerador de código é implementado em um alto nível de abstração já que é escrito em Lua. Caso um novo hardware ou extensão seja lançada é necessário apenas reescrever a geração do código. A linguagem utilizada – Lua – continua a mesma.

O *shader* é um programa Lua que possui um ambiente global e funções para processamento de vértices ou fragmentos.

5.4 Ambiente Global

O ambiente global do programa Lua carregado pode declarar variáveis que serão usadas pelas funções de *shader*. A Tabela 12 ilustra um código do ambiente global de execução.

```

mvp = lslMatrix("MODELVIEW_PROJECTION")
mvit = lslMatrix("MODELVIEW", "INVERSE_TRANSPOSE")

LightVec = lslVector(0.0, 0.0, 1.0, 1.0)
eye = lslVector(0.0, 0.0, 1.0, 1.0)

diffuseMaterial = lslVector(0.0, 0.0, 1.0, 1.0); -- blue diffuse material

specularMaterial = lslVector(1.0, 1.0, 1.0, 1.0); -- white specular material

```

Tabela 12 - Exemplo do ambiente global

Este ambiente global é análogo à frequência de computação constante do trabalho de Proudfoot et al. [13], e ao parâmetros *uniform* da linguagem Cg [15].

Os trechos de código Lua apresentados aqui podem ou não fazer parte de um mesmo arquivo Lua. Como segmentar o código em módulos é uma decisão do programador. Cada *shader* pode ser carregado separadamente.

5.5 Vertex Shader

Um *vertex shader*, também chamado de *vertex program*, é uma função que substitui os estágios de transformação, iluminação, projeção e geração de coordenadas de textura do *pipeline* convencional. O modelo de processador de vértices tipicamente implementado pelo hardware foi apresentado na seção 3.2.

Para ter acesso ao dados de entrada e saída do vértice sendo processado são definidas duas tabelas globais de Lua: *input* e *output*.

A tabela *input* contém os atributos convencionais do vértice, como posição, normal e coordenadas de textura. O *shader* também pode utilizar atributos genéricos, especificando o nome que desejar². A aplicação deverá passar estes atributos usando o mesmo nome definido no *shader*.

A tabela *output* contém os atributos de saída do vértice, como posição, cor e coordenadas de textura.

Elementos da tabela de entrada podem somente ser lidos e os de saída podem somente ser escritos. A função do *vertex shader* não pode definir novos objetos globais, somente variáveis locais.

² Esta generalização ainda não foi implementada.

A Tabela 13 lista um exemplo de *vertex shader* que calcula a iluminação difusa e especular de um vértice. Este código é similar àquele apresentado na seção de Cg.

```
function diffuse_and_specular ()
    output.pos =.mvp * input.pos

    -- transform normal from model-space to view-space
    local normal = mvit * input.normal
    normal = normalize(normal)

    -- store normalized light vector
    local light = normalize(LightVec)

    -- calculate half angle vector
    local half = normalize(light + eye)

    -- calculate diffuse component
    local diffuse = dot(normal, light)

    -- calculate specular component
    local specular = dot(normal, half)
    specular = pow(specular, 32)

    -- combine diffuse and specular contributions and output final vertex color
    output.color = diffuse * diffuseMaterial + specular * specularMaterial
end
```

Tabela 13 - Exemplo de vertex shader

Foram implementadas duas gerações de código nativo, uma sobre a extensão NV_vertex_program e outra sobre a extensão EXT_vertex_shader, ambas soluções para OpenGL. A primeira foi testada em processadores da NVIDIA e a segunda em processadores da ATI. A versão a ser usada é determinada automaticamente verificando as capacidades do hardware.

A implementação das duas versões compartilham parte do código mas serão analisadas aqui individualmente.

5.5.1 NV_vertex_program

Todo escalar, vetor ou matriz usado pelo *shader* possui referência para algum registrador do processador de vértice, seja ele constante, temporário, de entrada ou de saída.

A execução do ambiente global do *shader* apenas define os objetos globais (escalares, vetores e matrizes) que serão usados pelo *vertex shader*. Neste

Para dar suporte a atributos genéricos, ainda não implementado, pode-se usar o seguinte mecanismo. Caso o *shader* acesse um campo da tabela *input* que não esteja pré-definido, este campo é anotado e ao fim da execução associado a algum registrador de entrada ainda disponível. O primeiro registrador candidato é a coordenada de textura da maior unidade de textura que não estiver sendo utilizada.

A geração do código no formato definido pela extensão `NV_vertex_program` é feita através do mecanismo de eventos de Lua. As operações sobre os objetos constantes criados no ambiente global e sobre os objetos das tabelas *input* e *output* são interceptadas.

Tomamos como exemplo uma operação de multiplicação. A expressão

```
mvp * input.pos
```

dispara o evento de multiplicação da variável *mvp*. Este evento primeiramente cria um vetor temporário associado a um registrador temporário. Em seguida gera um conjunto de instruções (DP4) que realizarão o produto da matriz *mvp* com o vetor *input.pos*. Por fim retorna o vetor temporário criado. O mesmo acontece com os demais eventos aritméticos ou com as funções pré-definidas, como *normalize* ou *dot*.

A Tabela 14 apresenta o código gerado para o *shader* listado na Tabela 13.

```
!!VP1.0
DP4 R0.x, v[OPOS], c[0];
DP4 R0.y, v[OPOS], c[1];
DP4 R0.z, v[OPOS], c[2];
DP4 R0.w, v[OPOS], c[3];
MOV o[HPOS], R0;
DP4 R0.x, v[NRML], c[4];
DP4 R0.y, v[NRML], c[5];
DP4 R0.z, v[NRML], c[6];
DP4 R0.w, v[NRML], c[7];
DP3 R1.w, R0, R0;
RSQ R1.w, R1.w;
MUL R1.xyz, R0, R1.w;
DP3 R0.w, c[8], c[8];
RSQ R0.w, R0.w;
MUL R0.xyz, c[8], R0.w;
ADD R2, R0, c[9];
DP3 R3.w, R2, R2;
RSQ R3.w, R3.w;
MUL R3.xyz, R2, R3.w;
```

```

DP3 R2.x, R1, R0;
DP3 R2.y, R1, R3;
MOV R4.xy, R2.y;
MOV R4.zw, c[12].x;
LIT R4, R4;
MUL R5, c[10], R2.x;
MUL R6, c[11], R4.z;
ADD R7, R5, R6;
MOV o[COL0], R7;
END
# 28 instructions

```

Tabela 14 - Código gerado para NV_vertex_program

A alocação dos registradores temporários é auxiliada pelo mecanismo de coleta de lixo de Lua. Quando é criado um vetor temporário, associado a um registrador temporário, é criado também um *userdata* que permanece ligado ao vetor. Quando o vetor não é mais acessível, o coletor de lixo de Lua aciona a desalocação do *userdata* que estava associado ao vetor. Este método de desalocação marca o registrador antes utilizado pelo vetor como “livre”. Desta forma o registrador pode ser reutilizado por outro vetor. O efeito deste mecanismo pode ser observado no início do código apresentado na Tabela 14, com o reuso registrador R0.

5.5.2 EXT_vertex_shader

Como descrito antes, esta extensão abstrai o conceito de registradores. Há funções na API que instanciam escalares, vetores e matrizes, que ficam associados a um identificador, também denominado símbolo. Um objeto na *Lua Shading Language*, escalar, vetor ou matriz, referenciam aqui símbolos criados pela extensão EXT_vertex_shader.

Esta extensão não dá suporte ao acompanhamento de matrizes transformadas (inversas e transpostas). Esta funcionalidade teve de ser implementada em software.

O mesmo mecanismo de eventos usado na geração de código para a extensão NV_vertex_program foi utilizado aqui. Os atributos de entrada e saída do vértice também funcionam de forma similar.

A Tabela 15 ilustra uma simulação das chamadas feitas pelo gerador de código para a extensão EXT_vertex_shader quando processado o *shader*

apresentado na Tabela 13. Os valores de retorno das funções da API são emulados aqui para fins de depuração.

```

glBindParameterEXT (GL_MVP_MATRIX_EXT) =200
glGenSymbolsEXT (GL_MATRIX_EXT, GL_INVARIANT_EXT, GL_FULL_RANGE_EXT, 1) =100
glGenSymbolsEXT (GL_VECTOR_EXT, GL_INVARIANT_EXT, GL_FULL_RANGE_EXT, 1) =101
glGenSymbolsEXT (GL_VECTOR_EXT, GL_INVARIANT_EXT, GL_FULL_RANGE_EXT, 1) =102
glGenSymbolsEXT (GL_VECTOR_EXT, GL_INVARIANT_EXT, GL_FULL_RANGE_EXT, 1) =103
glGenSymbolsEXT (GL_VECTOR_EXT, GL_INVARIANT_EXT, GL_FULL_RANGE_EXT, 1) =104

glGenVertexShadersEXT (1) =1
glBindVertexShaderEXT (1)
glBindParameterEXT (GL_CURRENT_VERTEX_EXT) =201
glBindParameterEXT (GL_CURRENT_NORMAL) =202
glBindParameterEXT (GL_CURRENT_COLOR) =203
glBindTextureUnitParameterEXT (GL_TEXTURE0_ARB, GL_CURRENT_TEXTURE_COORDS) =204
glBindTextureUnitParameterEXT (GL_TEXTURE1_ARB, GL_CURRENT_TEXTURE_COORDS) =205
glBindTextureUnitParameterEXT (GL_TEXTURE2_ARB, GL_CURRENT_TEXTURE_COORDS) =206
glBindTextureUnitParameterEXT (GL_TEXTURE3_ARB, GL_CURRENT_TEXTURE_COORDS) =207
glBindTextureUnitParameterEXT (GL_TEXTURE4_ARB, GL_CURRENT_TEXTURE_COORDS) =208
glBindTextureUnitParameterEXT (GL_TEXTURE5_ARB, GL_CURRENT_TEXTURE_COORDS) =209

glBeginVertexShaderEXT ()
glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =105
glShaderOp2EXT (GL_OP_MULTIPLY_MATRIX_EXT, 0x105, 0x200, 0x201)
glShaderOp1EXT (GL_OP_MOV_EXT, GL_OUTPUT_VERTEX_EXT, 0x105)
glShaderOp2EXT (GL_OP_MULTIPLY_MATRIX_EXT, 0x105, 0x100, 0x202)
glGenSymbolsEXT (GL_SCALAR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =106
glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =107
glShaderOp2EXT (GL_OP_DOT3_EXT, 0x106, 0x105, 0x105)
glShaderOp1EXT (GL_OP_RECIP_SQRT_EXT, 0x106, 0x106)
glShaderOp2EXT (GL_OP_MUL_EXT, 0x107, 0x105, 0x106)
glShaderOp2EXT (GL_OP_DOT3_EXT, 0x106, 0x101, 0x101)
glShaderOp1EXT (GL_OP_RECIP_SQRT_EXT, 0x106, 0x106)
glShaderOp2EXT (GL_OP_MUL_EXT, 0x105, 0x101, 0x106)
glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =108
glShaderOp2EXT (GL_OP_ADD_EXT, 0x108, 0x105, 0x102)
glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =109
glShaderOp2EXT (GL_OP_DOT3_EXT, 0x106, 0x108, 0x108)
glShaderOp1EXT (GL_OP_RECIP_SQRT_EXT, 0x106, 0x106)
glShaderOp2EXT (GL_OP_MUL_EXT, 0x109, 0x108, 0x106)
glShaderOp2EXT (GL_OP_DOT3_EXT, 0x106, 0x107, 0x105)
glGenSymbolsEXT (GL_SCALAR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =110
glShaderOp2EXT (GL_OP_DOT3_EXT, 0x110, 0x107, 0x109)
glGenSymbolsEXT (GL_SCALAR_EXT, GL_INVARIANT_EXT, GL_FULL_RANGE_EXT, 1) =111
glGenSymbolsEXT (GL_SCALAR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =112
glShaderOp2EXT (GL_OP_POWER_EXT, 0x112, 0x110, 0x111)
glShaderOp2EXT (GL_OP_MUL_EXT, 0x108, 0x103, 0x106)
glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =113
glShaderOp2EXT (GL_OP_MUL_EXT, 0x113, 0x104, 0x112)
glGenSymbolsEXT (GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1) =114

```

```
glShaderOp2EXT (GL_OP_ADD_EXT, 0x114, 0x108, 0x113)
glShaderOp1EXT (GL_OP_MOV_EXT, GL_OUTPUT_COLOR0_EXT, 0x114)
glEndVertexShaderEXT ()
```

Tabela 15 - Código gerado para EXT_vertex_shader

O mesmo mecanismo de coleta de lixo, usado para minimizar o uso de temporários, foi utilizado aqui. Quando um símbolo não é mais referenciado é guardado em um *pool* de símbolos disponíveis para uso futuro.

5.6 Fragment Shader

Como descrito na seção 3.3 o modelo de programação do processador de fragmentos ainda não está totalmente consolidado.

O modelo proposto pela NVIDIA, exposto pelas extensões NV_texture_shader e NV_register_combiners, não se mostrou flexível e ortogonal o suficiente para a programação. De fato algumas referências nem o caracterizam como um modelo programável, mas apenas como um modelo configurável.

Apesar de não ter sido implementada uma geração de código para processamento de fragmentos, a linguagem Lua e os tipos estendidos de dados definidos parecem ser suficientes para oferecer a programabilidade nesse nível.

6 Resultados

Alguns exemplos foram implementados para avaliar o poder de expressão da linguagem desenvolvida e para comparar esta proposta com outras existentes.

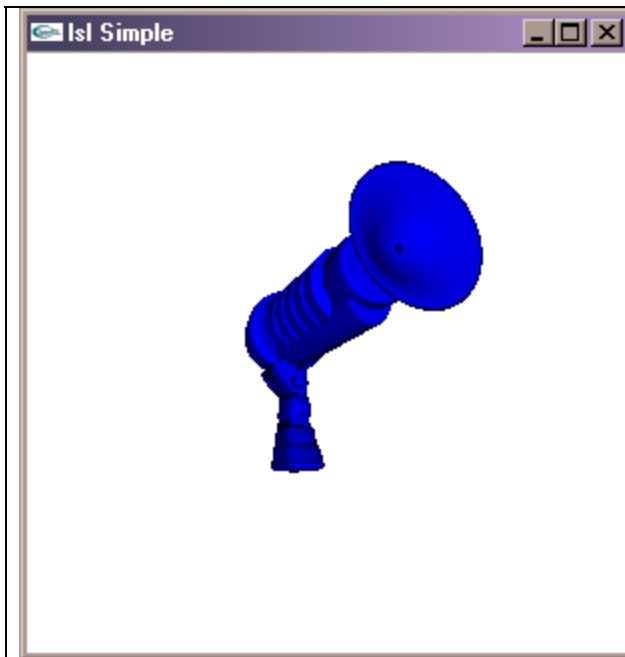
6.1 IsISimple

Esta aplicação é parte da demonstração da linguagem Cg e foi adaptada para a *Lua Shading Language*³. Um modelo é renderizado e sua iluminação é calculada por um *vertex shader* Lua.

Dois modelos básicos foram implementados para fins de comparação. A Figura 7 ilustra à esquerda um modelo somente com componente difusa e à direita com componente difusa e especular. Também são listados os códigos dos *shaders* em *Lua Shading Language* e os códigos nativos gerados para `NV_vertex_program`.

Este exemplo também foi compilado para a extensão `EXT_vertex_shader` e testado em um hardware da ATI. O código nativo foi apresentado na Tabela 15.

³ O modelo utilizado no exemplo é de propriedade da NVIDIA Corporation.



```

mvp = lslMatrix("MODELVIEW_PROJECTION")
mvit=lslMatrix("MODELVIEW","INVERSE_TRANSPOSE")

LightVec = lslVector(0.0, 0.0, 1.0, 1.0)
eye = lslVector(0.0, 0.0, 1.0, 1.0)

-- blue diffuse material
diffuseMaterial=lslVector(0.0, 0.0, 1.0, 1.0);

-- white specular material
specularMaterial=lslVector(1.0, 1.0, 1.0, 1.0);

function diffuse ()
    output.pos = mvp * input.pos

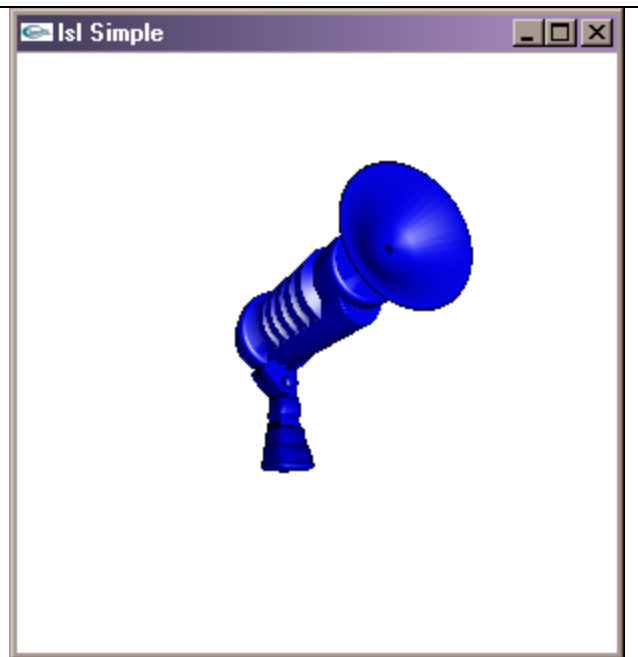
    -- transform normal from
    -- model-space to view-space
    local normal = mvit * input.normal
    normal = normalize(normal)

    -- store normalized light vector
    local light = normalize(LightVec)

    -- calculate diffuse component
    local diffuse = dot(normal, light)

    output.color = diffuse * diffuseMaterial
end

```



```

mvp = lslMatrix("MODELVIEW_PROJECTION")
mvit=lslMatrix("MODELVIEW","INVERSE_TRANSPOSE")

LightVec = lslVector(0.0, 0.0, 1.0, 1.0)
eye = lslVector(0.0, 0.0, 1.0, 1.0)

-- blue diffuse material
diffuseMaterial=lslVector(0.0, 0.0, 1.0, 1.0);

-- white specular material
specularMaterial=lslVector(1.0, 1.0, 1.0, 1.0);

function diffuse_and_specular ()
    output.pos = mvp * input.pos

    -- transform normal from
    --model-space to view-space
    local normal = mvit * input.normal
    normal = normalize(normal)

    -- store normalized light vector
    local light = normalize(LightVec)

    -- calculate half angle vector
    local half = normalize(light + eye)

    -- calculate diffuse component
    local diffuse = dot(normal, light)

    -- calculate specular component
    local specular = dot(normal, half)
    specular = pow(specular, 32)

```

	<pre>-- combine diffuse and specular --contributions and output final vertex color output.color = diffuse * diffuseMaterial + specular * specularMaterial end</pre>
<pre>!!VP1.0 DP4 R0.x, v[OPOS], c[0]; DP4 R0.y, v[OPOS], c[1]; DP4 R0.z, v[OPOS], c[2]; DP4 R0.w, v[OPOS], c[3]; MOV o[HPOS], R0; DP4 R0.x, v[NRML], c[4]; DP4 R0.y, v[NRML], c[5]; DP4 R0.z, v[NRML], c[6]; DP4 R0.w, v[NRML], c[7]; DP3 R1.w, R0, R0; RSQ R1.w, R1.w; MUL R1.xyz, R0, R1.w; DP3 R0.w, c[8], c[8]; RSQ R0.w, R0.w; MUL R0.xyz, c[8], R0.w; DP3 R2.x, R1, R0; MUL R3, c[9], R2.x; MOV o[COL0], R3; END # 18 instructions</pre>	<pre>!!VP1.0 DP4 R0.x, v[OPOS], c[0]; DP4 R0.y, v[OPOS], c[1]; DP4 R0.z, v[OPOS], c[2]; DP4 R0.w, v[OPOS], c[3]; MOV o[HPOS], R0; DP4 R0.x, v[NRML], c[4]; DP4 R0.y, v[NRML], c[5]; DP4 R0.z, v[NRML], c[6]; DP4 R0.w, v[NRML], c[7]; DP3 R1.w, R0, R0; RSQ R1.w, R1.w; MUL R1.xyz, R0, R1.w; DP3 R0.w, c[8], c[8]; RSQ R0.w, R0.w; MUL R0.xyz, c[8], R0.w; ADD R2, R0, c[9]; DP3 R3.w, R2, R2; RSQ R3.w, R3.w; MUL R3.xyz, R2, R3.w; DP3 R2.x, R1, R0; DP3 R2.y, R1, R3; MOV R4.xy, R2.y; MOV R4.zw, c[12].x; LIT R4, R4; MUL R5, c[10], R2.x; MUL R6, c[11], R4.z; ADD R7, R5, R6; MOV o[COL0], R7; END # 28 instructions</pre>

Figura 7 – lslSimple

O exemplo com iluminação difusa e especular acima é similar ao programa Cg apresentado na Tabela 11. A Tabela 16 apresenta o código nativo gerado pelo compilador da linguagem Cg para esse programa.

A *Lua Shading Language* gerou um código com apenas uma instrução a mais do que aquele gerado por Cg. Esta diferença ocorreu com a fusão de duas instruções de MUL e ADD em uma instrução de MAD.

De fato o código nativo gerado pela *Lua Shading Language* não é ótimo. Em geral instruções de MOV poderiam ser eliminadas e instruções de MUL e

ADD poderiam ser fundidas. Estas otimizações serão feitas futuramente através da geração de um código intermediário que será analisado antes de se gerar o código nativo.

```

DP4 R0.x, c[0], v[0];
DP4 R0.y, c[1], v[0];
DP4 R0.z, c[2], v[0];
DP4 R0.w, c[3], v[0];
MOV o[HPOS], R0;
DP4 R0.x, c[4], v[2];
DP4 R0.y, c[5], v[2];
DP4 R0.z, c[6], v[2];
DP4 R0.w, c[7], v[2];
DP4 R1, R0.xyz, R0.xyz;
RSQ R1, R1.x;
MUL R0, R1.x, R0.xyz;
DP4 R1, c[8], c[8];
RSQ R1, R1.x;
MUL R1, R1.x, c[8];
ADD R2, R1, c[9];
DP4 R3, R2, R2;
RSQ R3, R3.x;
MUL R2, R3.x, R2;
DP4 R2, R0, R2;
MOV R2.z, c[10].x;
MOV R2.w, c[10].x;
LIT R2, R2;
MUL R2, R2.z, c[11];
DP4 R0, R0, R1;
MAD R0, R0.x, c[9], R2;
MOV o[COL0], R0;
END
# 27 instructions

```

Tabela 16 - Código nativo gerado por Cg

6.2 IsIFresnel

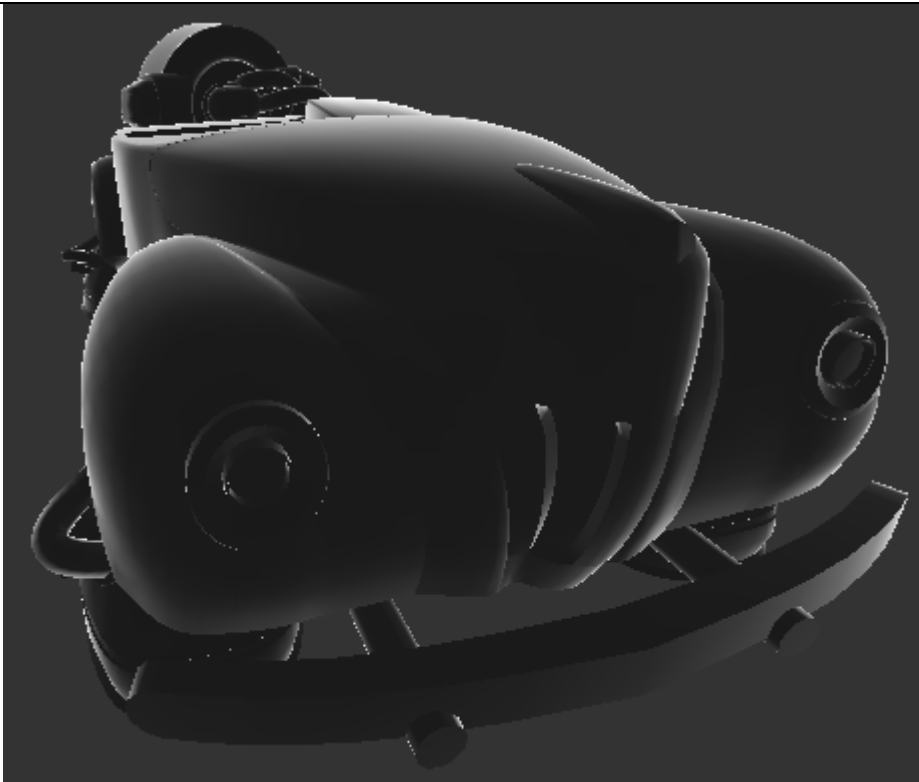
Este exemplo também foi baseado em uma aplicação para demonstração da linguagem Cg⁴.

O que está sendo visualizado na imagem abaixo é o coeficiente de reflexão de Fresnel. Quando a luz incide sobre uma superfície, parte dela é refletida e parte é refratada.

⁴ O modelo utilizado no exemplo é de propriedade da NVIDIA Corporation.

A quantidade de luz refletida depende da razão do índice de refração do meio e da superfície, da polarização e do comprimento de onda da luz e do ângulo de incidência.

Um modelo simplificado é calculado pelo *shader* apresentado abaixo. O vetor refletido é usado para acessar uma textura de reflexão de ambiente. Esse valor é combinado com a cor do vértice, que é exibida na imagem abaixo. Ou seja, quanto menor o ângulo de incidência maior será a parcela de luz refletida (a luz encontra-se no observador).



```

mvp = lsmatrix("MODELVIEW_PROJECTION")
mv  = lsmatrix("MODELVIEW")
mvit = lsmatrix("MODELVIEW", "INVERSE_TRANSPOSE")

function main()
    output.pos = mvp * input.pos

    local normal = normalize(mvit * input.normal)
    local eyeToVert = normalize(mv * input.pos)

    -- reflect the eye vector across the normal vector for reflection
    output.tex0.xyz = reflect(eyeToVert, normal)
    output.tex0.w = 1.0

    local f0 = .1;
    -- compute the fresnel term
    local oneMCosAngle = 1.2 + dot(eyeToVert, normal)

```

```

oneMCosAngle = pow(oneMCosAngle, 5)
local color = lerp(oneMCosAngle, 1, f0)
output.color = color.x
end

```

Figura 8 - lslFresnel

6.3 lslBillboard

Este exemplo implementa um algoritmo de *billboard*. A árvore na Figura 9 é uma textura com informação de transparência aplicada a um quadrado (dois triângulos). Todos os quatro vértices do quadrado possuem a mesma coordenada quando enviadas ao pipeline, porém cada vértice possui um dx e um dy que identifica qual dos quatro vértices ele é. Baseado nesse atributo a posição real do vértice é calculada. A coordenada de textura também poderia ser calculada de forma simples.

Os atributos dx e dy foram codificados no atributo *normal*. O ideal seria que estivessem em um atributo com nome genérico, mas esta opção ainda não foi implementada.



```
local right = input.normal.x * mv[1]
local up = input.normal.y * mv[2]

local pos = input.pos + right + up
output.pos = mvp * pos
output.tex0 = input.tex0
output.color = 1.0
end
```

Figura 9 – lslBillboard

7 Conclusão

O hardware programável é hoje uma realidade e não há dúvida que uma linguagem para programação de *shaders* com alto nível de abstração é necessária. Este fato justifica o grande número de trabalhos neste sentido, tanto acadêmicos quanto comerciais.

A nossa proposta é menos ambiciosa do que outras apresentadas, porém acreditamos que ela é capaz de expressar os *shaders* usados em renderização em tempo real nos hardware atuais.

A maior diferença da *Lua Shading Language* é o fato da abstração ter sido construída sobre a linguagem Lua, ao invés de se criar mais uma linguagem específica. O gerador de código foi completamente escrito em Lua em um alto nível de abstração, facilitando a modificação deste para novas funcionalidades de futuros hardware.

As facilidades da linguagem Lua permitem a codificação de *shaders* inseridos em uma descrição de grafo de cena, de forma semelhante ao suportado pelo Renderman. Proudfoot et al. [13] destacou a necessidade de uma linguagem de *shading* para artistas. Acreditamos que o uso de Lua pode ser um primeiro passo nesse sentido, já que Lua tem sido usada como uma linguagem de *script* na indústria de jogos [18].

A linguagem de *shading* definida cumpre com o objetivo de aumentar a clareza do código, como pode ser observado comparando-se os exemplos apresentados na *Lua Shading Language* com aqueles na linguagem de máquina nativa.

Também foi atingida a possibilidade de reuso de código. Modelos tradicionais de iluminação ou geradores de coordenadas de textura podem ser implementados por funções e organizados em bibliotecas.

Nossa solução de *vertex shader* foi baseada nas extensões OpenGL NV_vertex_program e EXT_vertex_shader, garantindo a portabilidade para pelo menos duas plataformas de hardware, NVIDIA e ATI.

A solução implementada também é independente de plataforma, podendo executar tanto em Windows quanto em sistemas Unix. Esta característica foi assegurada pela grande portabilidade das linguagens utilizadas, C e Lua.

Recentemente a extensão OpenGL ARB_vertex_program foi definida para processamento de vértices e diversos fabricantes de hardware já se comprometeram em implementá-la. Está em nossos planos um gerador de código para esta extensão.

A arquitetura de processamento de vértices está bem mais consolidada do que aquela para processamento de fragmentos. Foram feitas tentativas de se usar o modelo da NVIDIA de *texture shader* e *register combiner*, mas este não se mostrou flexível suficiente para a programação.

Novas arquiteturas para processamento de fragmentos, mais flexíveis e ortogonais, estão surgindo, o que deve facilitar a criação de abstrações. Devemos esperar a estabilização de um modelo de programação em baixo nível de abstração, o que deve acontecer com a publicação da extensão OpenGL ARB_fragment_program.

Um dos objetivos da *Lua Shading Language* é ser usada como ferramenta para o ensino de algoritmos para renderização em tempo-real.

8 Referências Bibliográficas

1. MÖLLER, T.; HAINES, E. **Real-Time Rendering**. 1. ed. Massachusetts: A K Peters, 1999. 481p.
2. WOO, M.; NEIDER, T.; SHREINER, D. **OpenGL Programming Guide**. 3. ed. Addison Wesley, 1999.
3. MICROSOFT. **DirectX**. Disponível em: <http://msdn.microsoft.com/directx/>. Acesso em: 03 set. 2002.
4. SEGAL, M.; AKELEY, K. **The Design of the OpenGL Graphics Interface**. Silicon Graphics Computer Systems, 1994.
5. SGI. **OpenGL Extension Registry**. Disponível em: <http://oss.sgi.com/projects/ogl-sample/registry/>. Acesso em: 03 set. 2002.
6. MICROSOFT. **DX9: DirectX Graphics Future Features Overview**. Disponível em: <http://www.microsoft.com/mscorp/corpevents/meltdown2001/ppt/DXG9.ppt>. Acesso em: 03 set. 2002.
7. LINDHOLM, E.; KILGARD, M. J.; MORETON, H. A user-programmable vertex engine. **In: Proceedings of ACM SIGGRAPH (2001)**, pp. 149-158.
8. NVIDIA. **NVIDIA OpenGL Extension Specifications**. Disponível em: <http://developer.nvidia.com/>. Acesso em: 03 set. 2002
9. ATI. **Hardware Shading with EXT_vertex_shader and ATI_fragment_shader**. Disponível em: <http://www.ati.com/developer/>. Acesso em: 03 set. 2002.
10. **Ata da reunião do OpenGL ARB em 18-19 jun. 2002**. Disponível em: http://www.opengl.org/developers/about/arb/notes/meeting_note_2002-06-18.html
11. NVIDIA. **NVIDIA “CineFX” Architecture**. Technical Brief, TB-00553-001_v02, 2002. Disponível em: http://developer.nvidia.com/docs/IO/3121/ATT/CineFX-TechBrief_v02.pdf

12. MATROX. **Matrox Parhelia OpenGL Extension Specification List v1.0**. 2002. Disponível em: <http://developer.matrox.com>. Acesso em: 03 set. 2002.
13. PROUDFOOT, K. et al. A Real-Time Procedural Shading System for Programmable Graphics Hardware. **In: Proceedings of ACM SIGGRAPH (2001)**, pp. 159-170.
14. 3DLABS. **OpenGL 2.0 Specifications**. Disponível em: <http://www.3dlabs.com/support/developer/ogl2/index.htm>. Acesso em: 03 set. 2002.
15. NVIDIA. **The NVIDIA Cg Compiler – C for Graphics**. Technical Brief, TB-00511-001-v01, 2002. Disponível em: <http://developer.nvidia.com/Cg/>. Acesso em: 03 set. 2002.
16. APODACA, A. A; GRITZ, L. **Advanced Renderman: Creating CGI for Motion Pictures**. 1. ed. Morgan Kaufmann, 2000. 544p.
17. IERUSALIMSCHY, R.; FIGUEIREDO, L. H.; CELES, W. Lua – an extensible extension language. **In: Software: Practice & Experience, 26 (6)**, 1996.
18. IERUSALIMSCHY, R.; FIGUEIREDO, L. H.; CELES, W. The Evolution of an Extension Language: A History of Lua. **In: V Brazilian Symposium on Programming Languages**, B-14-B28, 2001.

Apêndice A - Processadores Gráficos

As Tabelas abaixo listam os principais hardware disponíveis no mercado, com as respectivas extensões OpenGL relacionadas à programabilidade. Também são listadas as versões de *shaders* DirectX suportadas.

DirectX	
VS11	Vertex Shader 1.1
VS20	Vertex Shader 2.0
PS11	Pixel Shader 1.1
PS13	Pixel Shader 1.3
PS14	Pixel Shader 1.4
PS20	Pixel Shader 2.0
OpenGL	
VP	NV_vertex_program
VP2	NV_vertex_program2 (ainda não publicada)
VS	EXT_vertex_shader
ARBVP	ARB_vertex_program
FS	ATI_fragment_shader
MFS	MTX_fragment_shader (ainda não publicada)
FP	NV_fragment_program (ainda não publicada)
TS	NV_texture_shader
TS2	NV_texture_shader2
TS3	NV_texture_shader3
RC	NV_register_combiners
RC2	NV_register_combiners2

Nome	Codinome	OpenGL	DirectX
NVIDIA (www.nvidia.com)			
(não definido)	NV30	VP, ARBVP, VP2, TS3, RC2, FP	VS20, PS20
GeForce4 Ti	NV25	VP, ARBVP, TS3, RC2	VS11, PS13
GeForce3	NV20	VP, ARBVP, TS2, RC2	VS11, PS11
GeForce4 MX	NV17	RC	
GeForce2	NV15	RC	
GeForce2 MX	NV11	RC	
GeForce 256	NV10	RC	

Quadro4 XGL	NV25GL	VP, ARBVP, TS3, RC2	VS11, PS13
Quadro DCC	NV20GL	VP, ARBVP, TS2, RC2	VS11, PS11
Quadro4 550 XGL	NV17GL	RC	
Quadro2 Pro	NV15GL	RC	
Quadro2 Ex	NV11GL	RC	
Quadro	NV10GL	RC	
ATI (www.ati.com)			
Radeon 9700	R300	VS, ARBVP, FS	VS20, PS20
Radeon 8500	R200	VS, ARBVP, FS	VS11, PS14
Matrox (www.matrox.com)			
Parhelia-512		VS, MFS	VS20, PS20

Apêndice B - API da Lua Shading Language

```
#ifndef _LSL_H
#define _LSL_H

#ifdef WIN32
#include <windows.h>
#include <gl/gl.h>
#endif
#include <lua.h>

typedef int LSL_SHADER;

enum lslVertexShaderTarget {
    LSL_NV_vertex_program,
    LSL_EXT_vertex_shader
};

enum lslFragmentShaderTarget {
    LSL_DX_pixel_shader,
    LSL_ATI_fragment_shader
};

/* Initializes the library */
int lslOpen (lua_State *L);

/* Closes the library */
int lslClose (void);

/* Initializes an OpenGL extension */
int lslInitExtension (const char *extension);

/* Loads a Lua file */
int lslLoad (const char *filename);

/* Create the shader object
   A shader object may contain 2 types of programs
   - vertex shader
   - fragment shader
*/
LSL_SHADER lslCreateShader (void);
```

```

/* Create Vertex Shader, given a lua function */
int lslCreateVertexShader (LSL_SHADER shader, const char *function);

/* Create Fragment Shader, given a lua function */
int lslCreateFragmentShader (LSL_SHADER shader, const char *function);

/* Link a shader, checking compatibility between vertex and fragment shaders.
   At this step native code is generated
*/
int lslLinkShader (LSL_SHADER shader);

/* Bind shader to the pipeline
   A NULL value bind the conventional pipeline
*/
int lslBindShader (LSL_SHADER shader);

/* Delete the shader */
void lslDeleteShader (LSL_SHADER shader);

/* Change the value of the specified scalar variable */
int lslSetScalar (const char *name, float x);

/* Change the value of the specified vector variable */
int lslSetVector (const char *name, float x, float y, float z, float w);

/* Change the matrix being tracked by the specified variable */
int lslTrackMatrix (const char *name, GLenum matrix, GLenum transform);

void lslCheckError (const char *msg);
#ifdef _DEBUG
#define lslCheckError(msg)
#endif

/* This functions passes vertex attributes, mostly used when application
   defines custom data for vertex
*/
void lslVertexAttrib4f (const char *var, float x, float y, float z, float w);
void lslVertexAttrib4fv (const char *var, GLfloat *v);
void lslVertexAttribPointer (const char *var, enum type, uint stride, void *addr);
void lslEnableVertexAttribArray (const char *var);
void lslDisableVertexAttribArray (const char *var);

#endif

```

Tabela 17 - API da Lua Shading Language

Apêndice C - Uso de extensões OpenGL

A especificação de todas as extensões OpenGL estão disponíveis online no endereço <http://oss.sgi.com/projects/ogl-sample/registry/>. Neste mesmo endereço há três arquivos de *header* - *glext.h*, *wglext.h*, *glxext.h* - disponíveis para download. Estes arquivos são atualizados a cada nova extensão publicada e contém a definição dos novos *tokens*, protótipos e definições dos tipos de funções introduzidas pelas extensões.

A Tabela 18 lista três trechos do conteúdo do arquivo *glext.h*. O primeiro trecho apresenta a definição de alguns tokens introduzidos pela extensão NV_vertex_program. O segundo trecho mostra o protótipo de algumas funções desta mesma extensão. O último trecho define um tipo ponteiro de função para cada nova função introduzida pela extensão.

```

#ifndef GL_NV_vertex_program
#define GL_VERTEX_PROGRAM_NV          0x8620
#define GL_VERTEX_STATE_PROGRAM_NV   0x8621
#define GL_ATTRIB_ARRAY_SIZE_NV      0x8623
#define GL_ATTRIB_ARRAY_STRIDE_NV    0x8624
#define GL_ATTRIB_ARRAY_TYPE_NV      0x8625
#define GL_CURRENT_ATTRIB_NV         0x8626
#define GL_PROGRAM_LENGTH_NV         0x8627
#define GL_PROGRAM_STRING_NV         0x8628
#define GL_MODELVIEW_PROJECTION_NV   0x8629
#define GL_IDENTITY_NV               0x862A
#define GL_INVERSE_NV                0x862B
#define GL_TRANSPOSE_NV              0x862C

#ifndef GL_NV_vertex_program
#define GL_NV_vertex_program 1
#ifdef GL_GLEXT_PROTOTYPES
GLAPI void APIENTRY glBindProgramNV (GLenum, GLuint);
GLAPI void APIENTRY glDeleteProgramsNV (GLsizei, const GLuint *);
GLAPI void APIENTRY glExecuteProgramNV (GLenum, GLuint, const GLfloat *);
GLAPI void APIENTRY glGenProgramsNV (GLsizei, GLuint *);
GLAPI void APIENTRY glGetProgramParameterfvNV (GLenum, GLuint, GLenum, GLfloat *);
GLAPI void APIENTRY glGetProgramivNV (GLuint, GLenum, GLint *);
GLAPI void APIENTRY glGetProgramStringNV (GLuint, GLenum, GLubyte *);

```

```

GLAPI void APIENTRY glGetTrackMatrixivNV (GLenum, GLuint, GLenum, GLint *);
typedef void (APIENTRY * PFNGLBINDPROGRAMNVPROC) (GLenum target, GLuint id);
typedef void (APIENTRY * PFNGLDELETEPROGRAMSNVPROC) (GLsizei n, const GLuint
*programs);
typedef void (APIENTRY * PFNGLEXECUTEPROGRAMNVPROC) (GLenum target, GLuint id,
const GLfloat *params);
typedef void (APIENTRY * PFNGLGENPROGRAMSNVPROC) (GLsizei n, GLuint *programs);
typedef void (APIENTRY * PFNGLGETPROGRAMPARAMETERFVNVPROC) (GLenum target, GLuint
index, GLenum pname, GLfloat *params);
typedef void (APIENTRY * PFNGLGETPROGRAMIVNVPROC) (GLuint id, GLenum pname, GLint
*params);
typedef void (APIENTRY * PFNGLGETPROGRAMSTRINGNVPROC) (GLuint id, GLenum pname,
GLubyte *program);
typedef void (APIENTRY * PFNGLGETTRACKMATRIXIVNVPROC) (GLenum target, GLuint
address, GLenum pname, GLint *params);

```

Tabela 18 - Trechos do arquivo glxext.h

Para uma aplicação fazer uso de uma extensão ela deve verificar em tempo de execução se a extensão é suportada pelo hardware (ou driver) em uso. Em caso positivo a aplicação precisa importar as funções dinamicamente, através de um sistema dependente de plataforma.

A Tabela 19 ilustra a importação de algumas funções na plataforma Windows. A primeira seção declara as funções que serão importadas. Estas funções devem ser declaradas somente uma vez, caso contrário a aplicação acusará erro de ligação. A segunda seção ilustra o processo de importação através de chamada a função *wglGetProcAddress*. O código abaixo não faz nenhuma verificação do sucesso dessa importação, e só funciona para a plataforma windows.

```

PFNGLAREPROGRAMSRESIDENTNVPROC  glAreProgramsResidentNV = 0;
PFNGLBINDPROGRAMNVPROC          glBindProgramNV          = 0;
PFNGLDELETEPROGRAMSNVPROC       glDeleteProgramsNV       = 0;
PFNGLEXECUTEPROGRAMNVPROC       glExecuteProgramNV       = 0;

glAreProgramsResidentNV =
    (PFNGLAREPROGRAMSRESIDENTNVPROC)wglGetProcAddress("glAreProgramsResidentNV");
glBindProgramNV          =
    (PFNGLBINDPROGRAMNVPROC)      wglGetProcAddress("glBindProgramNV"      );
glDeleteProgramsNV       =
    (PFNGLDELETEPROGRAMSNVPROC)   wglGetProcAddress("glDeleteProgramsNV"   );
glExecuteProgramNV       =
    (PFNGLEXECUTEPROGRAMNVPROC)   wglGetProcAddress("glExecuteProgramNV"   );

```

Tabela 19 - Importação de funções de extensão

Hoje há mais aproximadamente 300 extensões definidas. Cada extensão pode acrescentar dezenas de novas funções. Desta forma é muito custoso realizar a importação de forma manual.

Foi elaborado então um script em Lua que interpreta os arquivos *glext.h*, *wglext.h* e *glxext.h* e gera um código C para fazer a importação das funções de uma dada extensão.

Este código C gerado é multi-plataforma e faz a verificação do sucesso da importação de cada função. Para usar o módulo a aplicação precisa apenas incluir este código e chamar a função:

```
int lsl_init_extensions(const char *origReqExts);
```

passando como parâmetro as extensões a serem inicializadas. Para verificar quais as extensões não puderam ser inicializadas, por qualquer razão, a aplicação pode chamar a função:

```
const char* lsl_get_unsupported_extensions();
```