4
Hardware Gráfico Programável

Nos últimos tempos, o hardware gráfico vem passando por uma profunda transformação: de um aparelho especializado na rasterização, para um processador de ponto flutuante, rápido, preciso, paralelizado e de uso geral. A figura 4.1 exibe um pouco dessa evolução, assinalando o avanço no processamento de gráficos em tempo real, e mostrando, da esquerda para direita, imagens geradas a partir de hardwares mais antigos para mais modernos.

Muitos esforços estão sendo feitos, de modo a tirar proveito desse potencial. Para se ter uma idéia, uma CPU Pentium4 funcionando a 3 GHz tem capacidade de 6 GFLOPS, enquanto uma GPU (Graphics Processor Unit) NVidia 6800 é capaz de 40 GFLOPS. Com essa evolução muitos pesquisadores buscam maneiras efetivas de acelerar algoritmos usando esses novos recursos.

4.1
Historia do Hardware Gráfico

Se o avanço do hardware gráfico vem andando a passos longos, isso se deve a alguns fatores. O principal é a necessidade de grande quantidade de processamento para simular um mundo 3D com grande precisão.

Esse avanço se deu de uma forma extremamente rápida. Em meados da década de 1990 o hardware gráfico não passava de vários circuitos integrados que trabalhavam juntos, de modo a gerar uma imagem na tela. Contemporaneamente a GPU possuí mais transistores que uma CPU. Por exemplo, um processador dual-core Opteron contém aproximadamente 233 milhões de transistores, enquanto uma GPU NVidia Gforce 7800 GFX possuí 304 milhões de transistores.

O termo GPU apareceu no final da década de 1990, quando o termo VGA (Video Graphics Array) já não servia para designar o hardware gráfico. A indústria nesse meio tempo identificou quatro gerações na evolução da GPU, descritos sucintamente na seqüência.

4.1.1

Figura 4.1: A evolução do pipeline

Aceleração Gráfica Pré-GPU

Várias companhias, como a Silicon Graphics (SGI) e Evans & Sutherland, antes do desenvolvimento da GPU já manufaturavam hardware gráfico especializado, porém, pouco acessível à programação. O desenvolvimento desse tipo de hardware criou vários dos conceitos atuais, tais como a transformação de vértices e o mapeamento de texturas.

4.1.2 Primeira Geração

A primeira geração de GPUs apareceu em 1998, incluindo NVidia TNT2, ATI Rage e 3dfx Voodoo3. Estas GPUs são capazes de rasterizar triângulos pré-transformados e aplicar uma ou duas texturas. Estas GPUs aliviavam completamente a CPU de atualizar os pixels individualmente, mas não podiam aplicar transformações aos vértices.

4.1.3 Segunda Geração

A segunda geração, de 1999 a 2000, inclui a NVidia GeForce 256, GeForce2, ATI Radeon 7500 e S3 Savage 3D. Estas GPUs são capazes de transformar os vértices e calcular iluminação sem o auxílio da CPU. O

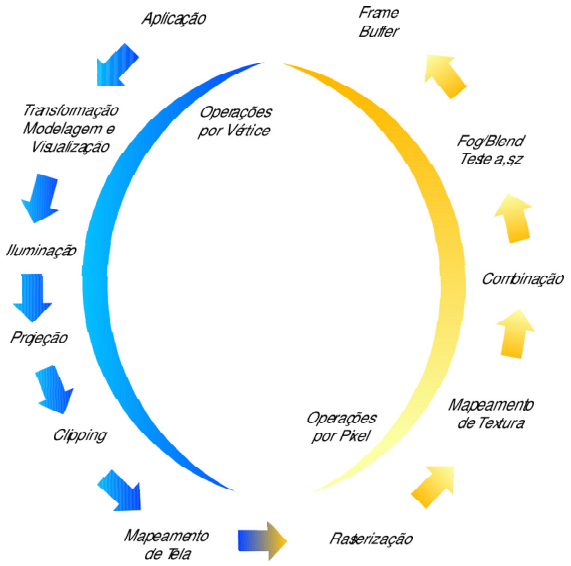

Figura 4.2: O pipeline hardware gráfico

conjunto de operações matemáticas para combinar texturas e colorir pixels foi amplamente aperfeiçoado, mas as possibilidades ainda eram limitadas.

4.1.4 Terceira Geração

As placas gráficas Nvidia GeForce3, GeForce4 Ti, Microsoft Xbox, e ATI Radeon 8500, que apareceram em 2001, fazem parte desta geração. Nesta geração já é possível a programação sobre os vértices. Apareceram, também, avanços no processamento dos pixels.

Como estas GPUs suportavam programação sobre a transformação dos vértices, mas não podiam ser programadas a nível dos pixels, esta geração foi considerada transacional. O acesso programável dos vértices se dava através da extensão do OpenGL ARB_vertex_program, ou usando DirectX 8.

4.1.5 Quarta Geração

A atual geração, que começou em 2002, inclui NVidia Geforce FX e ATI Radeon 9700. Estas GPUs já disponibilizam programação por vértice e por pixel. Essa abertura para programação permite transformações complexas para os vértices e para os fragmentos.

4.2 O Pipeline do Hardware Gráfico

Um pipeline é uma seqüência de estágios que operam em paralelo em uma ordem fixa. Cada estágio é altamente especializado, e a saída de um estágio é

a entrada do próximo. Esse tipo de arquitetura possui muitas vantagens, pois permite o aperfeiçoamento independente de cada estágio. A figura 4.2 mostra os estágios do pipeline convencional de hardware gráfico.

Como é mostrado na figura 4.2, a entrada do pipeline são vértices, que carregam atributos como a cor, a posição e o vetor normal. Esses vértices passam pelo primeiro estágio de *transformação dos vértices*, onde são transformados para o espaço de tela. A cada vértice é aplicada uma seqüência de instruções.

Em seguida, esses vértices entram na fase de *montagem de primitivas* e rasterização. Primeiro os vértices são transformados em primitivas, dependendo da forma como foram enviados ao pipeline, o que resulta numa seqüência de triângulos, linhas ou pontos.

Algumas dessas primitivas serão descartadas, dependendo de sua posição (frustum clipping e culling). As primitivas que não são descartadas precisam ser rasterizadas, isto é, será determinado o conjunto de pixels cobertos pela primitiva geométrica. Cada primitiva possui regras que determinam esse conjunto de pixels. Os resultados da rasterização são: um conjunto de localização de pixels e um conjunto de fragmentos. Uma observação importante é que daqui em diante a posição desses fragmentos não pode ser alterada.

A próxima fase é a *texturização* e *colorização*. Quando os fragmentos entram nesse estágio, os parâmetros necessários à aplicação de textura e iluminação são interpolados, e cada fragmento passa por uma seqüência de operações que determinam sua cor final. Esse estágio pode mudar a profundidade ou mesmo descartar o fragmento. A saída desse estágio são os fragmentos com sua cor calculada.

Finalmente, os fragmentos entram na fase de *testes* e *operações sobre fragmentos*, onde é aplicada uma série de operações, antes de atualizar o frame buffer. Nesse estágio são removidas as superfícies escondidas, é efetuado o blending, cada fragmento entra numa seqüência de testes (alpha, z-buffer) e, se falhar em algum, ele é descartado sem modificar o frame buffer.

4.3 O Pipeline do Hardware Gráfico Programável

Nessa nova arquitetura de processamento, é possível trocar algumas fases do pipeline gráfico por programas sobre os quais passamos a ter controle. Agora, através da entrada programável para a GPU, o usuário escolhe se quer que o hardware use o pipeline comum, ou se deve realizar outras rotinas de transformação de vértices e fragmentos.

Teremos basicamente duas etapas programáveis, como mostra a figura

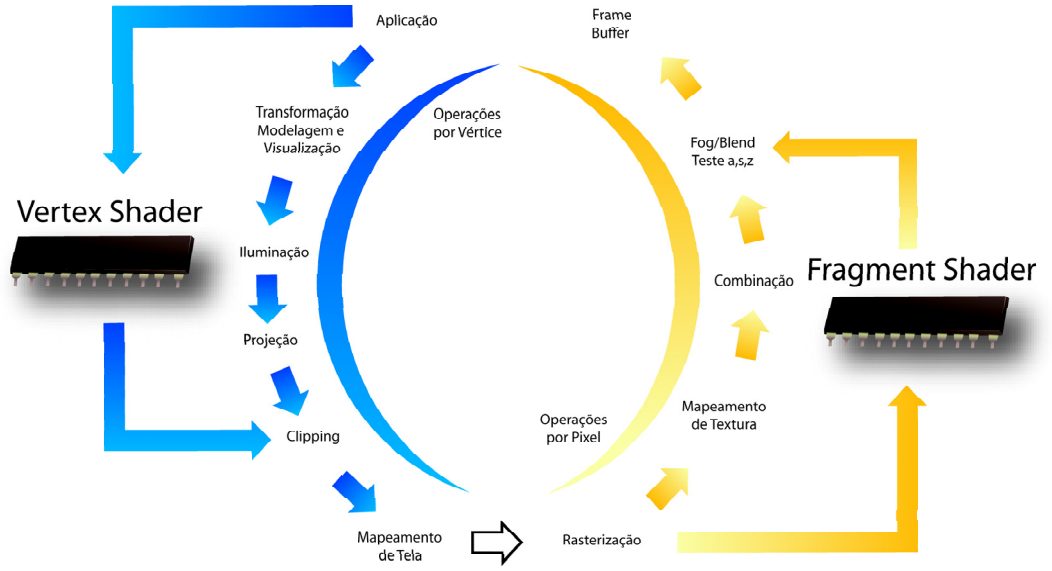

Figura 4.3: O Pipeline do Hardware Gráfico Programável

4.3. A primeira atuará sobre os vértices, onde cada vértice é processado e seus atributos são transformados. É possível derivar um novo atributo das propriedades do vértice.

A próxima etapa atuará nos fragmentos que foram gerados, interpolados com as propriedades dos vértices. Nesse estágio temos acesso às texturas e outras propriedades definidas na memória da GPU, assim como acesso às propriedades que foram interpoladas pelos vértices.

O programa responsável por transformar os vértices é chamado de *vertex shader*, e o que transforma os fragmentos de *fragment shader*.

4.3.1 Linguagens de Baixo Nível

A primeira linguagem para programação em GPU foi o Direct3D 8 Shading Language e é o equivalente às extensões do OpenGL: NV_vertex_program e ARB_vertex_program. Mais tarde, apareceram novas versões: Direct3D 9.0, NV_vertex_program2 e ARB_fragment_program.

Estas linguagens são compostas de um conjunto de instruções bastante limitadas, especializadas para programação gráfica. O tipo básico para os registradores são vetores de quatro componentes de valores ponto flutuante, e as operações são, geralmente, a soma componente a componente, produto componente a componente, produto interno e multiplicações de matriz por vetor. Essas linguagens são o equivalente ao assembly da GPU.

4.3.2

Linguagens de Alto Nível

Com a heterogeneização das linguagens de baixo nível, e também pela dificuldade gerada por elas, naturalmente foram desenvolvidas linguagens de alto nível para os shaders, equivalentemente ao que aconteceu às CPUs.

Essas linguagens são construídas com base nas instruções de baixo nível. A sintaxe da maioria dessas linguagens é baseada na linguagem de programação **C**, obviamente com várias limitações, porém, ainda assim, capaz de operar com ponto flutuante, inteiros e booleanos.

Um exemplo é Stanford Real-Time Shading Language. Existem ainda a GL Shading Language, que foi introduzida junto com o OpenGL 2.0 e também o Brook. Um outro exemplo, mais importante, é a linguagem Cg / Direct3D HLSL, que na próxima sessão será melhor descrita.

4.3.3 Cg

A linguagem Cg foi desenvolvida para facilitar e acelerar o uso do pipeline programável das GPUs. Com esse nível de abstração, não é mais necessário programar diretamente com o assembly do hardware gráfico. Essa linguagem foi desenvolvida em conjunto pela NVidia e Microsoft e é compatível, tanto com a API OpenGL, como a API do DirectX.

O Cg tem a sintaxe baseada na linguagem C, tanto que Cg é a abreviação para "C for graphics". Assim, se o usuário já é familiarizado com a linguagem C, fica muito fácil trabalhar com Cg.

Como o Cg é uma linguagem especializada, muitos dos conceitos da linguagem C não foram implementados, pois ela é destinada a programar o hardware gráfico. Esse tipo de linguagem é conhecida como Shading Language. De qualquer maneira, o Cg é capaz de, muito mais além de desenhar, realizar simulações físicas e outras aplicações, através do hardware gráfico.

Diferente das linguagens comuns, os programas Cg operam sobre vértices e fragmentos. Toda vez que o pipeline gráfico é disparado, o programa que processa os vértices, conhecido com vertex shader, atua sobre cada vértice e o programa que processa fragmentos, conhecido como fragment shader, atua sobre cada fragmento.

A linguagem é especializada em transformar vértices e fragmentos, porém, não contém várias das capacidades que as linguagens gerais, como C, C++ ou Java, implementam. O Cg não possui suporte para classes e outras capacidades de objeto orientado. A implementação atual não tem ponteiros e nem alocação de memória, e não é fornecido suporte a acesso de arquivos. Essas são basicamente as restrições do atual hardware gráfico.

O Cg permite o uso de vetores, estruturas e controladores de fluxo. Tem suporte nativo para matrizes e vetores, pois esses tipos estão fundamentalmente ligados à maioria das aplicações gráficas 3D. A linguagem conta também com uma biblioteca de funções, que é voltada para operações comuns em aplicações 3D. Por exemplo, conta com uma função **reflect**, que calcula reflexões de vetores.

Os programas Cg são executados de forma que as transformações sobre um vértice ou fragmento não interfiram nos demais vértices e fragmentos, para que possam fazer uso do paralelismo do pipeline gráfico.

Influência do OpenGL e Direct3D sobre CG

O Cg é apenas um componente de toda a infra-estrutura da renderização de cenas 3D complexas em GPU programável. Os próximos parágrafos mostram a maneira especial de como a linguagem interage com as atuais aplicações 3D.

Antes do uso das GPUs, todo o processamento de uma cena 3D era feito com uso da CPU, que era responsável pela transformação dos vértices e dos fragmentos. O hardware continha apenas o pixel buffer. Os programadores tinham que escrever todas as rotinas relativas ao pipeline gráfico, ou seja, o pipeline era inteiramente programável. O problema é que a CPU não pode acompanhar tamanha complexidade e quantidade de operações para produzir cenas realísticas.

Hoje em dia, as aplicações não programam seus próprios algoritmos de renderização 3D usando a CPU. Geralmente é usado o OpenGL ou o Direct3D, as duas principais interfaces de programação, para comunicar comandos de renderização para a GPU.

OpenGL

Em meados de 1990, as maiores empresas de sistemas gráficos criaram um consórcio chamado OpenGL Architeture Review Board (ARB), e comissionaram a Silicon Graphics no desenvolvimento do OpenGL. No começo, o OpenGL funcionava apenas em grandes estações gráficas UNIX. A Microsoft, um dos membros do ARB, implementou o OpenGL, de maneira a tratar os gráficos 3D em seu Windows NT, e depois o exportou para o Windows 95, e, subseqüentemente, para todos os sistemas operacionais para desktop.

O OpenGL não é limitado para apenas esses sistemas operacionais, possuindo também versões para Apple. Os usuários Linux podem usar a implementação Mesa ou uma implementação acelerada por hardware, como

no caso do NVIDIA´s OpenGL driver for Linux. Essa flexibilidade torna o OpenGL uma ótima interface multi-plataforma para gráficos 3D.

O desenvolvimento do OpenGL vem evoluindo com o hardware gráfico. Hoje em dia, extensões do OpenGL permitem acesso para os últimos avanços na GPU. Isso inclui as extensões ARB, para programação em vértices e em fragmentos. Ou seja, a cada versão do OpenGL são incluídos novos comandos, tanto em baixo nível, quanto em alto nível. Como o Cg opera como uma camada acima dessa interface, ele continuará compatível, mesmo com as futuras versões do OpenGL.

Direct3D

A Microsoft iniciou o desenvolvimento de interface de programação Direct3D em 1995, fazendo parte da interface de programação multimídia da Microsoft, denominada DirectX.

A maior vantagem do Direct3D é que ele tenta estar a par de todos os avanços que ocorrem no hardware gráfico. Assim, a Microsoft lança sempre novas versões dessa interface de programação. As últimas implementações já suportam o HLSL, a implementação Microsoft para a linguagem Cg.

O compilador Cg

Antes de serem processados pela GPU, para que possam ser executados, os programas escritos em Cg devem ser traduzidos em seqüências de operações mais simples. O compilador Cg, primeiro traduz em uma forma aceita pela API 3D, OpenGL ou Direct3D. Em seguida, a aplicação transfere essa tradução do programa Cg para a GPU, usando os comandos apropriados. A API 3D, finalmente, tem a tarefa de traduzir essas operações em comandos que o hardware possa executar.

Os detalhes dessa tradução dependem principalmente das capacidades da GPU em questão e da interface de programação 3D escolhida. Como o Cg é compilado em sua fase intermediaria de OpenGL ou Direct3D, ele depende do tipo e geração da GPU no computador. Pode ser que a placa gráfica em questão não seja capaz de executar um código Cg. Por exemplo, o programa pode ter mais instruções do que a capacidade do processador de fragmentos.

CgGL e CgD3D, bibliotecas CG especificas para OpenGL e Direct3D

Ao compilar um programa com uma linguagem-padrão, é gerado um executável, que roda diretamente na CPU. Uma vez compilado, um programa não necessita ser recompilado, a menos que exista uma mudança no código, o que é denominado de compilação estática.

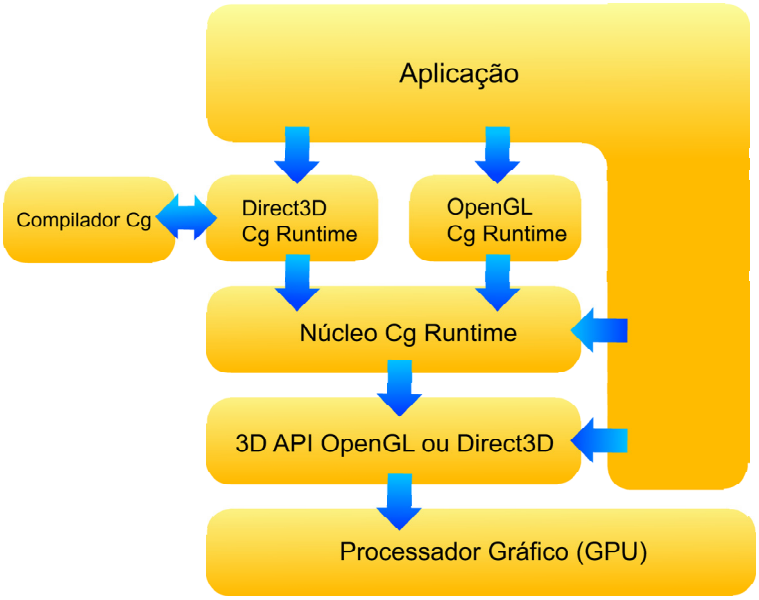

Figura 4.4: Como a biblioteca Cg Runtime se comunica com a API 3D

No caso do Cg acontece o que é chamado de compilação dinâmica. Embora também suporte programação estática, o compilador Cg não é um programa separado, mas sim uma biblioteca conhecida como Cg Runtime.

As aplicações que usam Cg devem ser linkadas com o Cg Runtime. As aplicações com Cg chamam as rotinas dessa biblioteca, todas com o prefixo cg, para compilar e manipular os programas Cg. A compilação dinâmica permite que os programas sejam otimizados para a GPU instalada no computador.

Junto com o núcleo do Cg Runtime, temos duas bibliotecas relacionadas, como é exibido na figura 4.4. Se a aplicação utiliza OpenGL, você usará a biblioteca CgGL para traduzir corretamente seu programa para o OpenGL; e, no caso do Direct3D, a biblioteca a ser usada é a CgD3D.

Basicamente essas bibliotecas possuem as mesmas rotinas, mas servem para traduzir o programa Cg para cada uma das APIs 3D. As rotinas da CgGL têm o prefixo cgGL, e as rotinas com prefixo cgD3D pertencem à biblioteca CgD3D.

5
Fluidos Estáveis

Neste capítulo é apresentado um método para resolver numericamente as equações de Navier-Stokes para fluidos incompressíveis. Esse método foi desenvolvido por Joe Stam em 1999 e é conhecido como Stable Fluids (stam1999). Ele é um método semi-lagrangeano, que é incondicionalmente estável.

5.1
Navier-Stokes para fluidos incompressíveis e o método da projeção

Ao supor que o fluido possui temperatura e densidade constantes, ele é descrito pelo campo de velocidades $\mathbf{V}$ e pelo campo escalar de pressão p, que são variáveis no tempo e no espaço, denotando por X a coordenada espacial.

Dados os campos $\mathbf{V}$ e p em um tempo inicial $t = 0$, esses campos evoluem de acordo com as equações de Navier-Stokes para fluidos incompressíveis:

$$\frac{\partial \mathbf{V}}{\partial t} = -(\mathbf{V} \cdot \nabla)\mathbf{V} - \frac{1}{\rho}\nabla p + \mu \nabla^2 \mathbf{V} + F \tag{5-1}$$

$$\nabla \cdot \mathbf{V} = 0, \tag{5-2}$$

onde μ é a constante de viscosidade, ρ é a constante de densidade e F uma força externa que atua sobre o fluido.

Suponha-se que o líqüido está delimitado em um domínio D. Neste caso, as condições de fronteira são dadas por uma função u_D, definida no bordo de D, ∂D, de forma que a componente normal da velocidade em relação a ∂D seja zero, isto é, o material não atravessa o bordo.

Acontece que, para o fluido em questão, os campos de pressão e velocidade que aparecem na equação de Navier-Stokes estão relacionados, ou seja, é possível combinar as duas equações em uma única equação para a velocidade. Para mostrar esse resultado é necessário o seguinte teorema:

Teorema 5.1 (Decomposição Helmholtz-Hodge) *Um campo vetorial W, em D, onde D é uma região do espaço com bordo ∂D suave, pode ser unicamente decomposto na forma*

$$W = \mathbf{V} + \nabla p,$$