Um Tutorial sobre GPUs

Silvia Esparrachiari e Victor H. P. Gomes

Instituto de Matemática e Estatística – Universidade de São Paulo (USP)
São Paulo – SP – Brasil

{silviaeg,vhpgomes}@ime.usp.br

Abstract. *The present work makes a shot review about GPU (graphics processing unit) evolution and state of art. It also presents a small tutorial about the Cg language use in GPU programs and provides an example of non-conventional use of GPU (GPGPU – generic purpose GPU).*

Resumo. *O presente trabalho faz uma pequena introdução sobre a evolução e o estado da arte da tecnologia de GPUs* (*graphics processing unit*). *Também apresenta um simples tutorial sobre a utilização da linguagem Cg de programação em GPUs e um exemplo de utilização não convencional da GPU* (GPGPU – *generic purpose GPU*).

1. Introdução

Uma unidade gráfica de processamento (*graphics processing unit* - GPU) é um dispositivo dedicado à manipulação e renderização de objetos gráficos e está presente em quase todos os computadores pessoais (PCs), estações de trabalho, notebooks e consoles de jogos modernos. Por apresentarem uma arquitetura dedicada e altamente paralelizada, as GPUs modernas são muito mais eficientes que as CPUs (*central processing unit*) de propósito geral no que diz respeito à execução de algoritmos gráficos.

A evolução das GPUs tem ocorrido numa velocidade relativamente alta nos últimos anos devido, principalmente, a três fatores:

- O compromisso da indústria de semi-condutores em dobrar o número de transistores dentro de um microchip a cada 18 meses – fato conhecido historicamente como Lei de Moore;
- A capacidade do ser humano em perceber e compreender uma grande quantidade de dados e informações com alta precisão do mundo 3D;
- A vontade e o prazer que o ser humano possui em ser estimulado e entretido visualmente.

Considerando estes três fatores, podemos logo inferir ao menos duas áreas de desenvolvimento tecnológico nas quais a evolução das GPUs apresenta um forte impacto: a indústria de jogos e entretenimento – que criam jogos e outros produtos de entretenimento cada vez mais realistas – e a indústria de pesquisa técnica e científica, que utiliza as GPUs não apenas para a visualização de objetos 3D, mas também para a execução de cálculos científicos para os quais a arquitetura da GPU se apresenta mais vantajosa que a das CPUs.

2. História da GPU

Antes do efetivo surgimento dos processadores gráficos dedicados, surgiram hardwares que, apesar de não realizarem nenhum processamento gráfico efetivo, encarregavam-se da exibição das imagens e textos nos monitores da época. Tais dispositivos eram denominados "placas de vídeo" (*video cards*) e é por este nome que algumas pessoas ainda se referem às GPUs modernas.

5.1. Placas de Video

A primeira placa de vídeo a ser comercializada foi lançada pela companhia IBM juntamente com o primeiro IBM PC em 1981. O dispositivo de vídeo, que seguia a especificação MDA (*Monochrome Display Adapter*), era capaz de trabalhar apenas no modo texto, apresentando 80x25 linhas por vez, na tela do monitor. Ele também possuía 4KB de memória e trabalhava duas cores de pixel: a do fósforo do monitor e preto. Um exemplo de computador da época é mostrado na Figura 1.

Figura 1 – Computador IBM com placa de video com especificação MDA e monitor de fósforo verde.

Com o passar do tempo várias especificações de placas de video surgiram, cada uma com alguns avanços quanto à definição e memória em relação às anteriores. A Tabela 1 apresenta as principais características das especificações lançadas pela IBM durante a década de 80.

Tabela 1 Evolução das especificações das placas de vídeo na década de 80.

Especificação	Ano	Modo Texto	Modo Gráfico	Cores	Memória
MDA	1981	80x25	-	2	4KB
CGA	1981	80x25	640x480	4	16KB
HGC	1982	80x25	720x348	2	64KB
EGA	1984	80x25	640x350	16	256KB
IBM 8514	1987	80x25	1024x768	256	-
MCGA	1987	80x25	320x200	256	-
VGA	1987	80x25	640x480	256	256KB
SVGA	1989	80x25	800x600	256	512KB
XGA	1990	80x25	1024x768	65536	2MB

Em 1985, a companhia Amiga lançou o primeiro computador Commodore, que possuía um co-processador dedicado à rápida transferência de dados dentro da memória do computador. Tal dispositivo, denominado **blitter**, era capaz de copiar uma grande quantidade de dados de uma área de memória à outra em paralelo à CPU. O nome **blittler** tem sua origem no acrônimo BLIT (BLock Image Transfer) e era usado principalmente para movimentar grandes imagens bitmap dentro do espaço de memória.

Pouco tempo depois, no começo de 1987, a IBM lança o IBM 8514, que foi o primeiro computador a possuir tanto um dispositivo **blitter** quanto um *hardware* dedicado ao tratamento de primitivas 2D.

5.2. Primeira Geração de GPUs

A primeira geração de GPUs, que inclui as placas NVidia TNT2, ATi Rage e 3dfx Voodoo3, foram lançadas entre 1991 e 1998 e eram capazes de rasterizar triângulos pré-transformados e aplicar uma ou duas texturas sobre os mesmos. Estas placas também implementavam o conjunto de funcionalidades definidas pelo DirectX 6 e OpenGL 1.0. Apesar de liberarem a CPU de grande parte do processamento de objetos 2D e 3D, estas GPUs ainda não eram capazes de realizar transformações sobre os vértices dos objetos 3D e possuíam um conjunto bastante limitado de operações matemáticas que podiam ser executadas sobre as texturas a fim de definir as cores dos pixels durante o processo de rasterização.

5.3. Segunda Geração de GPUs

Entre os anos de 1999 e 2000, surgiram as GPUs de segunda geração, dentre elas: NVidia GeForce 256 e GeForce2, ATI Radeon 7500 e S3 Savage 3D. Tais GPUs eram capazes de realizar transformações e efeitos de iluminação (T&L) sobre os vértices dos objetos 3D, liberando a CPU de tal carga de processamento. Nesta mesma época, os padrões OpenGL e DirectX 7 também passaram a suportar T&L via *hardware*. O conjunto de operações matemáticas expandiu-se um pouco, passando a suportar *cube mapping* e operações com sinal, mas continuou bastante limitado.

Figura 2 – Test Drive 6, da Cryo Interactives (1999), utiliza o suporte em *hardware* de T&L.

5.4. Terceira Geração de GPUs

A terceira geração de GPUs foi bastante curta e restringiu-se às GPUs lançadas em 2001, como, por exemplo, as NVidia GeForce3 e GeForce4 Ti, a GPU do Microsoft Xbox e as ATI Radeon 8500. Esta foi a primeira geração a permitir a execução, na própria GPU, de programas para transformação de vértices (*Vertex Shaders*). Sendo assim, ao invés de oferecer suporte em *hardware* dos modos de iluminação e transformação especificados pelas bibliotecas OpenGL e DirectX 7, estas GPUs deixavam a aplicação especificar qual era a seqüência de operações a ser realizada sobre os vértices. Elas também possuíam mais opções de configuração de processamento dos pixels, mas, como este passo ainda não era programável, esta geração costuma ser considerada uma geração de transição.

Figura 3 – Ballistics, da GRIN (2001), utiliza *Vextex Shaders*.

5.5. Quarta Geração de GPUs

A quarta geração de GPUs, que compreende as GPUs lançadas entre os anos de 2002 e 2005, como é o caso da família NVidia GeForce FX e ATI Radeon 9700, permitiam tanto a programação de vértices quanto a de fragmentos (*pixel shaders*). Foi com esta geração de placas que surgiu o *"C for graphics"*, mais conhecido como Cg, desenvolvido pela NVidia, uma linguagem de programação baseada em C e voltada para a implementação de *vertex programs* e *fragment programs* que veremos mais adiante.

Figura 4 – Crysis, da Crytek (2007), utiliza *Geo Shaders*.

5.6. Quinta Geração de GPUs

Esta geração compreende todas as GPUs lançadas a partir de 2006, cuja principal característica é a arquitetura unificada de processamento e um *pipeline* cíclico. Exemplos de placas que apresentam tais características são NVidia GeForce 8800 e GeForce 200 e ATI Radeon HD. Estas GPUs também possuem suporte à linguagens de programação de mais alto nível que o Cg como, por exemplo, o CUDA no caso da NVidia. Mais informações a respeito da arquitetura destas GPUs modernas podem ser vistas na seção seguinte.

Figura 5 – Backbreaker, da Euphoria (2008), utilize CUDA.

3. Arquiteturas Modernas

Antes de abordarmos a estrutura das novas arquiteturas de GPUs, faremos uma pequena análise da evolução das mesmas.

3.1. Evolução das Arquiteturas de GPUs

Inicialmente, as GPUs de segunda geração possuíam uma arquitetura do tipo *pipeline* cujos estágios podem ser resumidos conforme esquematizado na Figura 6.

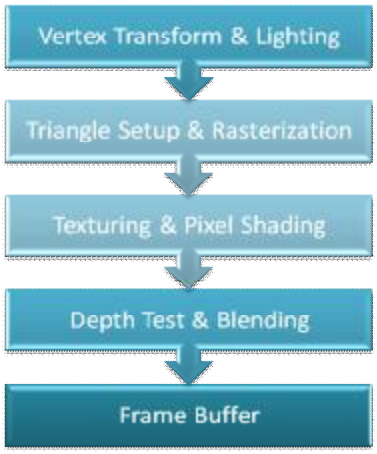

Figura 6 – *Pipeline* de processamento das primeiras GPUs

No primeiro estágio (*Vertex Transform & Lighting*), eram feitas operações de transformação da posição dos vértices e alteração da cor dos mesmos de acordo com suas propriedades de iluminação. No estágio seguinte (*Triangle Setup & Rasterization*), eram construídos e preenchidos os triângulos de acordo com a conectividade dos vértices, formando assim os componentes necessários ao processamento do próximo estágio: os fragmentos, comumente associados aos pixels do monitor. Logo em seguida (*Texturing & Pixel*

Shading), eram realizadas as aplicações de textura e outras transformações relativas à cor dos fragmentos. Por fim, no último estágio (*Depth Test & Blending*), eram realizados os testes de profundidade entre os objetos da cena e aplicados os devidos efeitos de composição, sendo que ao final deste estágio era gerada a imagem a ser exibida no monitor (*Frame Buffer*).

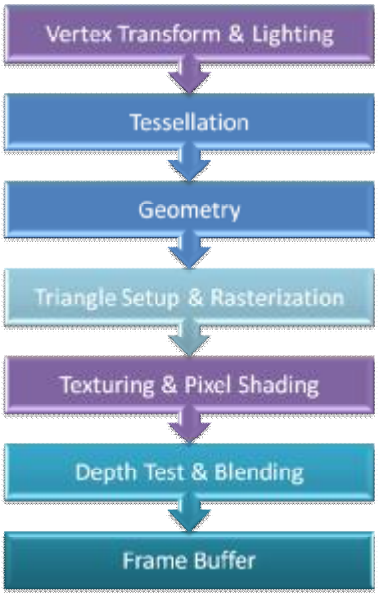

Figura 7 – Esquema genérico das GPUs até a quinta geração.

Com o passar do tempo, conforme visto nas inovações apresentadas pelas terceira e quarta gerações, os estágios de *Vertex Transform & Lighting* e *Texturing & Pixel Shading* se tornaram programáveis e foram adicionados novos estágios (*Tessellation* e *Geometry*), sendo que a quinta geração possuía uma arquitetura mais ou menos como a apresentada na Figura 7.

Um exemplo real de arquitetura em *pipeline* de GPU de quinta geração pode ser visto na Figura 8, que representa a arquitetura da NVidia GeForce 6. Repare que os processadores de vértices não são os mesmos que os processadores de texturas e fragmentos e que, nesta arquitetura, há mais processadores de vértices que de pixels.

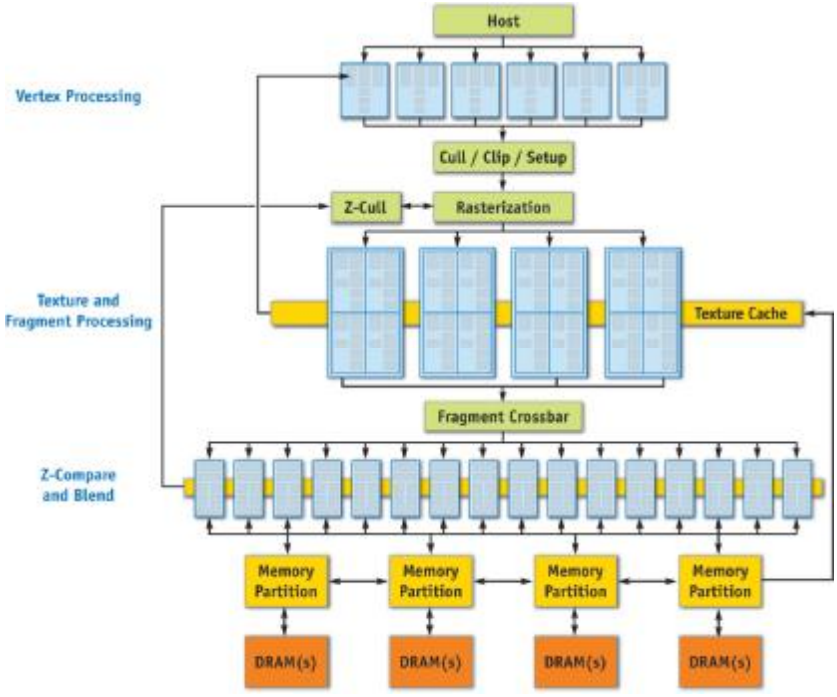

Figura 8 – Arquitetura da GPU NVidia GeForce 6.

3.2. As novas arquiteturas: o modelo unificado

A fim de planejar a nova geração de GPUs, os desenvolvedores, primeiramente, verificaram qual seria a nova proporção de processadores de vértices e de fragmentos. Após alguns testes envolvendo diversos aplicativos, chegou- se à conclusão que não era possível estabelecer uma proporção adequada, pois programas que trabalhavam com uma grande quantidade de modelos geométricos, como programas de modelagem do tipo CAD, utilizam principalmente os processadores de vértices, enquanto que programas que realizam uma grande quantidade de efeitos visuais, como programas de renderização de filmes, utilizam principalmente os processadores de fragmentos (Figura 9).

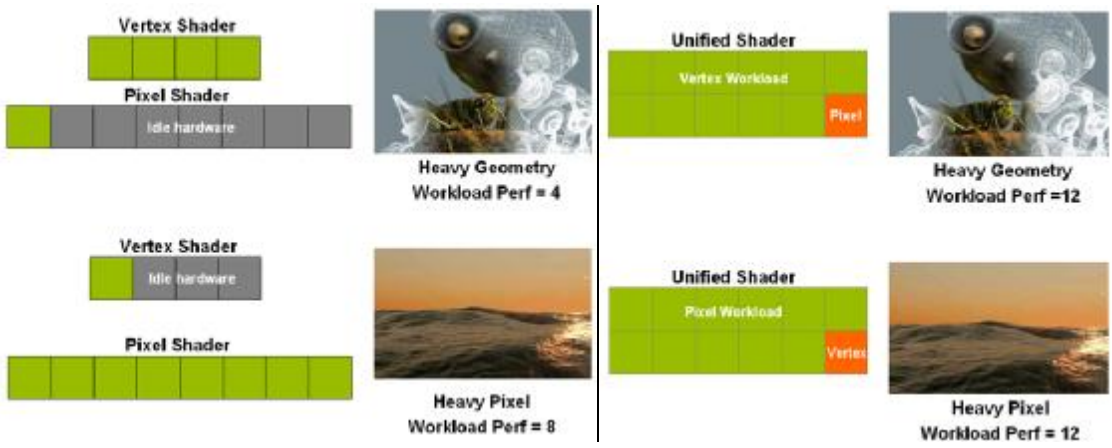

Figura 9 – À esquerda: exemplo de utilização de processadores de vértices e pixels por diferentes programas. À direita: utilização da proposta de processadores unificados.

Sendo assim, decidiu-se por unificar ambos os processadores e criar um único que fosse capaz de processar tanto vértices quanto pixels. Desta forma, não importa qual tipo de pixel a aplicação mais utiliza, sempre que houver algum dado a ser processado e um processador disponível, este nunca ficará ocioso.

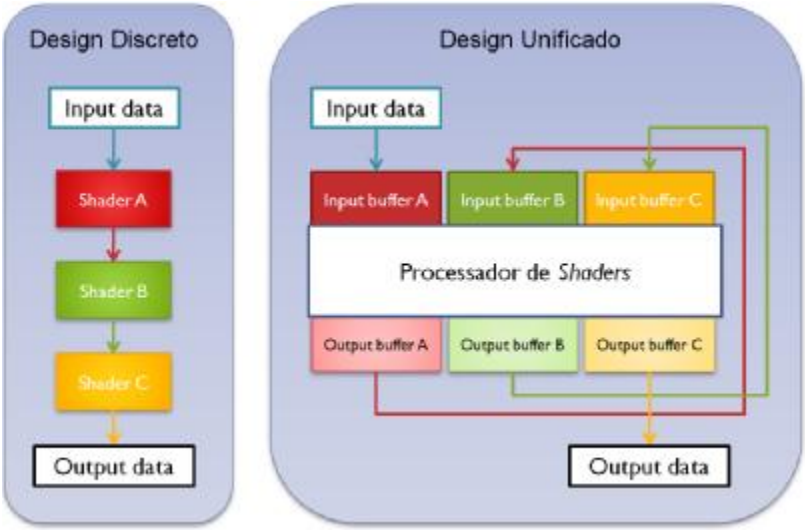

Figura 10 – Design Discreto x Design Unificado

Uma vez que ficou decidido que as novas arquiteturas teriam um único processador para executar todos os tipos de *shaders*, os diferentes tipos existentes – como *vertex shader, pixel shader,*

***geometry shader, physics shader,* etc – podem agora ser tratados simplesmente como diferentes *threads* a serem rodados em paralelo dentro de um mesmo cluster de processadores, necessitando apenas mapear a saída de um com a entrada de outro (**

Figura 10).

Uma das primeiras GPUs a apresentar esta nova arquitetura foi a NVidia GeForce 8800, lançada em fevereiro de 2007 e cuja estrutura é apresentada no esquema da Figura 11.

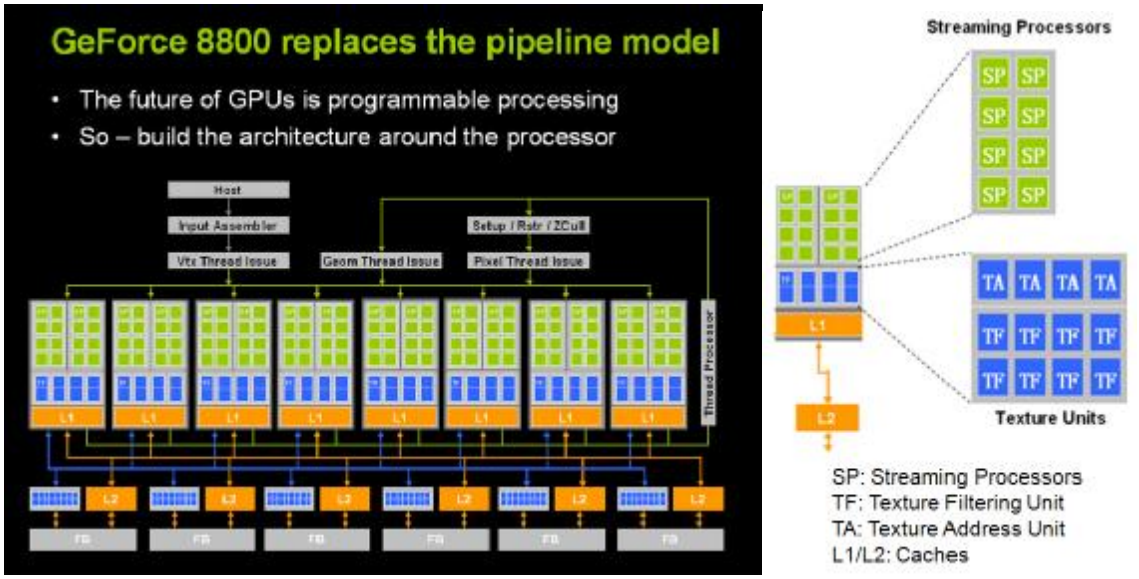

Figura 11 – À esquerda: arquitetura unificada da NVidia GeForce 8800. À diretira: detalhes dos componentes de um *cluster* de processamento.

4. Programando em CG + OpenGL

Uma vez que as GPUs se tornaram programáveis, tornou-se necessário criar uma linguagem de alto nível para que fosse possível a rápida e eficiente criação de programas para as mesmas. Foi com este intuito que os desenvolvedores da NVidia criaram o Cg, ou seja, *C for Graphics*. Por ser baseado em C, programadores que já possuem algum conhecimento desta linguagem são capazes de rapidamente aprender e implementar programas em Cg.

A linguagem Cg permite a criação de dois tipos de programas: *vertex programs* – para modificação das propriedades dos vértices – e *fragment programs* – para modificação das propriedades dos fragmentos. Vale à pena lembrar que, assim que as GPUs deixaram de implementar em *hardware* as especificações das bibliotecas OpenGL e Direct X, estas passaram a conter seus próprios *vertex programs* e *fragment programs* a serem rodados nas GPUs. Sendo assim, ao implementar e executar um *shader*, você estará substituindo o *shader default* pelo seu.

O Cg possui algumas diferenças importantes em relação à linguagem C. A primeira grande diferença está nos tipos de variáveis que podemos criar, pois, uma vez que a arquitetura das GPUs é voltada para a otimização de processamento de matrizes e vetores, o Cg possui uma vasta gama de tipos de matrizes e vetores que permitem a utilização de operações matemáticas otimizadas para estes tipos. Outra diferença é o fato de não ser possível incluir bibliotecas externas ao Cg a um programa. A única biblioteca

disponível, e que contém uma vasta gama de funções matemáticas, é a *Standard Cg Library* que é inclusa por padrão, sem a necessidade de declaração. Outro ponto diferente é o conceito de semântica, que nada mais é que dados sobre os objetos 3D que são passados automaticamente pela GPU para o seu programa. Veremos mais sobre os diferentes tipos de variáveis, *Standard Cg Library* e semântica nos exemplos de programas a seguir.

4.1. *Vertex Programs*

Um *vertex program* é executado uma vez para cada vértice contido na cena e tem como objetivo alterar suas propriedades, como posição, normal, cor, dentre outras. Vamos analisar um exemplo simples apresentado na Figura 12.

```
1   /* Simple Cg Program */
2   struct Test1v_Output
3   {
4       float4 position : POSITION;
5       float3 color    : COLOR0;
6   };
7
8   Test1v_Output Test1v_green( float2 position : POSITION )
9   {
10      Test1v_Output OUT;
11          OUT.position = float4(position,0.,1.);
12          OUT.color = float3(0.,1.,0.);
13
14      return OUT;
15  }
```

Figura 12 – Exemplo 1: *vertex program*

Neste exemplo, podemos verificar a declaração da função **`Test1v_green`**, que retorna uma estrutura do tipo **`Test1v_Output`**, declarada logo acima, e que recebe como parâmetro uma variável do tipo **`float2 position :POSITION`**, ou seja, ela recebe como parâmetro apenas as duas primeiras coordenadas referentes à posição de um vértice.

As declarações de variáveis do tipo **`float2`**, **`float3`** e **`float4`** definem vetores com precisão de ponto flutuante de 2, 3 e 4 posições, respectivamente. Já a declaração **`:POSITION`** e **`:COLOR0`** estabelecem a semântica da variável: posição, no primeiro caso, e cor, no segundo. Tais declarações associam uma propriedade do vértice ou fragmento a uma variável, sendo que o valor associado varia de vértice para vértice e de fragmento para fragmento. Se uma variável pertence ao conjunto de argumentos da função de entrada (neste caso a função **`Test1v_green`**) e está associada à uma semântica, então, seu valor será atualizado pela própria GPU antes do início da execução da função. Caso a variável associada a uma semântica seja retornada ou pertença a uma estrutura de retorno, o seu valor, no momento do retorno, substituirá o valor da semântica na memória da GPU.

Sendo assim, o que faz o programa acima?

Este programa projeta ortogonalmente todos os vértices no plano z=0 e os colori de verde (cor = RGB).

No entanto, para que estas transformações sejam executadas, devemos lembrar que um *shader* não faz absolutamente nada sozinho, é necessário utilizarmos uma biblioteca gráfica para gerar os objetos 3D, carregar o *shader* na GPU e executá-lo. Como exemplo, criaremos um programa simples, utilizando OpenGL, que desenha um triângulo vermelho:

```
static void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.,0.,0.);

    glBegin(GL_TRIANGLES);
      glVertex3f(-0.8,  0.8, -2.0);
      glVertex3f( 0.8,  0.8, -2.0);
      glVertex3f( 0.0, -0.8, -2.0);
    glEnd();

    glutSwapBuffers();
}
```

Figura 13 – Implementação de um triângulo vermelho em OpenGL

Para modificarmos as propriedades dos vértices do triângulo, utilizando o *vertex program* da Figura 12, precisamos, primeiramente, declarar um contexto Cg, um *profile* e um programa Cg. Um contexto Cg é semelhante a um contexto OpenGL. É ele quem gerencia a máquina de estados por trás das chamadas de métodos Cg. Um *profile* é um conjunto de operações matemáticas que deverão estar implementadas numa GPU para que esta possa rodar seu *shader*. Portanto, é uma boa prática de programação especificar sempre o *profile* que contém menor número de funções que não são utilizadas, permitindo que seu programa rode num maior número de GPUs. A declaração de um programa constitui em especificar algumas propriedades do seu programa Cg, como nome do arquivo e da função de entrada. No nosso exemplo, declaramos estas três variáveis como variáveis globais, para facilitar o acesso no decorrer do programa, e as inicializamos logo após a inicialização do contexto OpenGL (Figura 14).

```
static CGcontext   myCgContext;         // A Context To Hold Our Cg Program(s)

static CGprofile   myCgVertexProfile;   // The Profile To Use For Our Vertex Shader

static CGprogram   myCgVertexProgram;   // Our Cg Vertex Program

int main(int argc, char *argv[])
{
    ...

    myCgContext = cgCreateContext();            // Create A New Context For Our Cg Program(s)
      checkForCgError("creating context");
      cgGLSetDebugMode(CG_FALSE);
      cgSetParameterSettingMode(myCgContext, CG_DEFERRED_PARAMETER_SETTING);

    // Vertex Profile and Program initialization
    myCgVertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);  // Get The Latest GL Vertex Profile
      cgGLSetOptimalOptions(myCgVertexProfile);
      checkForCgError("selecting vertex profile");

    myCgVertexProgram = cgCreateProgramFromFile(
                              myCgContext,              /* Cg runtime context */
                              CG_SOURCE,                /* Program in human-readable form */
                              "Test1v.cg",              /* Name of file containing program */
                              myCgVertexProfile,        /* Profile: OpenGL ARB vertex program */
                              "Test1v_green",           /* Entry function name */
                              NULL);                    /* No extra compiler options */
    checkForCgError("creating vertex program from file");
      cgGLLoadProgram(myCgVertexProgram);
      checkForCgError("loading vertex program");

    glutMainLoop();

    return EXIT_SUCCESS;
}
```

Figura 14 – Inicialização de um contexto Cg, de um *Profile* e de um *Vertex Program* em OpenGL.

Uma vez declarado o programa, para utilizá-lo na modificação de um objeto 3D, é necessário carregar o *profile* (**`cgGLEnableProfile`**) e "ligar" o programa na GPU (**`cgGLBindProgram`**) antes do início do processo de construção do objeto e descarregar o *profile* ao final do processo (**`cgGLDisableProfile`**) (Figura 15). Note que não é preciso "desligar" o programa ao final do processo, pois isto é feito automaticamente.

```
static void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    cgGLBindProgram(myCgVertexProgram);
    checkForCgError("binding vertex program");

    cgGLEnableProfile(myCgVertexProfile);
    checkForCgError("enabling vertex profile");

    glColor3f(1.,0.,0.);

    glBegin(GL_TRIANGLES);
      glVertex3f(-0.8,  0.8, -2.0);
      glVertex3f( 0.8,  0.8, -2.0);
      glVertex3f( 0.0, -0.8, -2.0);
    glEnd();

    cgGLDisableProfile(myCgVertexProfile);
    checkForCgError("disabling vertex profile");

    glutSwapBuffers();
}
```

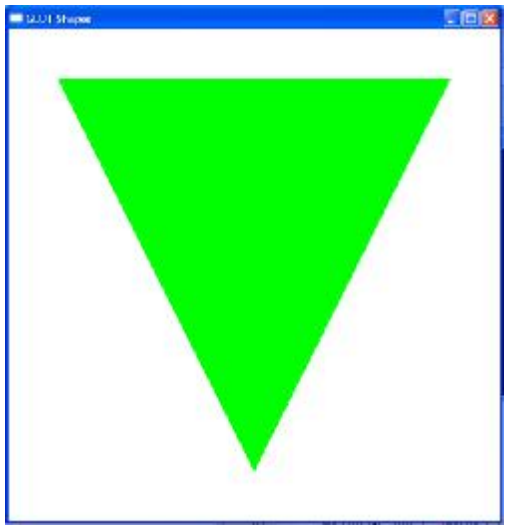

Figura 15 – À esquerda: carregamento e descarregamento do programa na GPU. À direita: resultado da execução do programa.

Agora, se executarmos o programa descrito, obteremos o triângulo da Figura 13 pintado de verde, como mostra a Figura 15.

4.2. *Passando Argumentos*

O *vertex program* apresentado na seção 4.1 é bastante simples. Vamos agora elaborá-lo um pouco mais passando uma matriz de transformação do objeto como parâmetro.

Um argumento passado a um programa Cg é o mesmo para todos os vértices declarados entre um **`cgGLBindProgram`** e outro, portanto, a nossa matriz de transformação será aplicada a todos os vértices do triângulo, de acordo com o programa da Figura 16.

```
/* Simple Cg Program */
struct Test2v_Output
{
     float4 position : POSITION;
     float3 color    : COLOR0;
};

Test2v_Output Test2v_green( float2 position : POSITION,
                            uniform float4x4 modelViewProj )
{
     Test2v_Output OUT;
         OUT.position = mul(modelViewProj, float4(position,0.,1.));
         OUT.color = float3(0.,1.,0.);

     return OUT;
}
```

Figura 16 – Passando argumentos a um programa Cg.

Os argumentos passados por um programa externo à GPU são declarados como uniformes (**`uniform`**), justamente devido ao fato de serem os mesmos para todos os vértices ou fragmentos. No programa da Figura 16, a matriz de projeção é passada por

meio da variável **`uniform float4x4 modelViewProj`**, que é uma matriz 4 por 4 com precisão de ponto flutuante. Para aplicar a transformação representada pela matriz a um vértice, utilizamos a função **`mul`** presente na *Standard Cg Library*, que retorna o resultado da multiplicação dos seus parâmetros – no nosso caso, uma matriz de transformação e um vetor de posição.

Para passarmos os valores da matriz de projeção ao programa Cg, devemos declarar uma variável do tipo **`CGparameter`**, inicializá-la e atualizá-la de acordo com os objetivos do nosso programa. No nosso exemplo, declaramos a variável globalmente e a inicializamos logo após o término da inicialização do *vertex program* (Figura 17).

```
static CGcontext   myCgContext;         // A Context To Hold Our Cg Program(s)

static CGprofile   myCgVertexProfile;   // The Profile To Use For Our Vertex Shader

static CGprogram   myCgVertexProgram;   // Our Cg Vertex Program

static CGparameter myCgVertexParam_modelViewProj; // Vertex Program External Parameter

int main(int argc, char *argv[])
{
    ...

    glClearColor(1,1,1,1);

    myCgContext = cgCreateContext();            // Create A New Context For Our Cg Program(s)
      checkForCgError("creating context");
      cgGLSetDebugMode(CG_FALSE);
      cgSetParameterSettingMode(myCgContext, CG_DEFERRED_PARAMETER_SETTING);

    // Vertex Profile and Program initialization
    myCgVertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);  // Get The Latest GL Vertex Profile
      cgGLSetOptimalOptions(myCgVertexProfile);
      checkForCgError("selecting vertex profile");

    myCgVertexProgram = cgCreateProgramFromFile(
                              myCgContext,              /* Cg runtime context */
                              CG_SOURCE,                /* Program in human-readable form */
                              "Test2v.cg",              /* Name of file containing program */
                              myCgVertexProfile,        /* Profile: OpenGL ARB vertex program */
                              "Test2v_green",           /* Entry function name */
                              NULL);                    /* No extra compiler options */
      checkForCgError("creating vertex program from file");
      cgGLLoadProgram(myCgVertexProgram);
      checkForCgError("loading vertex program");

    myCgVertexParam_modelViewProj = cgGetNamedParameter(myCgVertexProgram, "modelViewProj");
    checkForCgError("could not get modelViewProj parameter");

    glutMainLoop();

    return EXIT_SUCCESS;
}
```

Figura 17 – Declaração e inicialização de parâmetros de programas Cg em OpenGL.

Repare que, na função **`main`**, o método **`cgGetNamedParameter`** recebe como argumentos um objeto do tipo **`CgProgram`** (nosso *vertex program*), um *string* contendo o nome da variável no programa Cg e retorna um objeto do tipo **`CgParameter`**. Para atualizar a cada *frame* o valor desta variável, criamos uma matrix de transformação dentro do OpenGL e a passamos para o programa Cg utilizando o método **`cgGLSetStateMatrixParameter`**, conforme exibido na Figura 18.

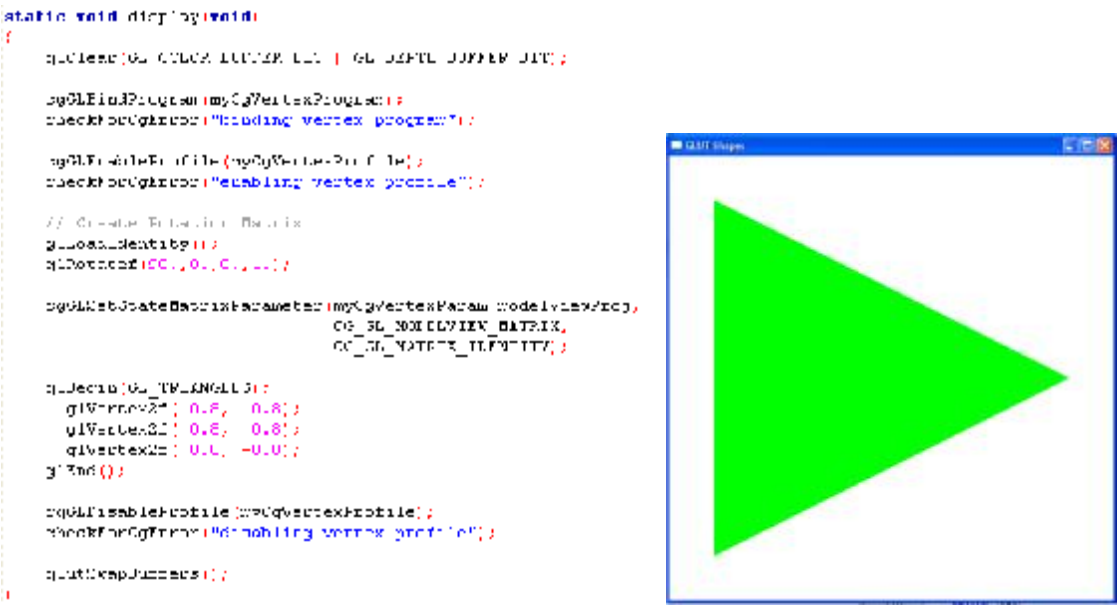

Figura 18 – À esquerda: atualizando parâmetros de programas Cg no OpenGL. À direita: resultado das modificações.

Uma vez feitas estas alterações, ao executarmos o programa, obteremos um triângulo verde rotacionado de 90 graus no sentido anti-horário (Figura 18).

4.3. *Fragment Programs*

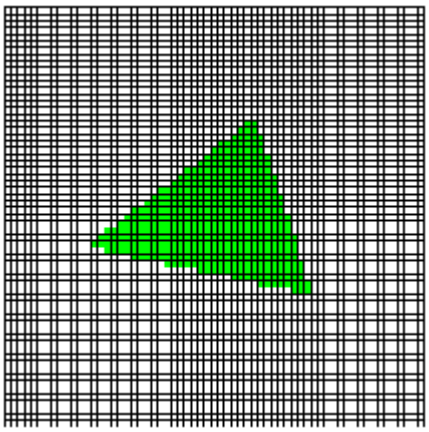

Figura 19 – Processo de rasterização de um triângulo.

Nas GPUs com arquitetura do tipo *pipeline*, para as quais foi feito o Cg, após a execução do *vertex program*, ocorre o processo de criação e preenchimento dos triângulos (rasterização), no qual são criados os fragmentos. Um *fragment program* é responsável por alterar as propriedades destes fragmentos, que não existiam no momento da execução do *vertex program*. Um fragmento pode ser entendido como sendo cada um dos pixels criados após a projeção dos objetos no plano da imagem. É importante ressaltar que um fragmento possui propriedades diferentes de um pixel, como, por exemplo, posição 3D, coordenadas de textura, normal, dentre outras, e um fragmento pode acabar não tendo nenhuma relação com o seu "pixel", pois pode ser eliminado em algum teste de profundidade (z-*buffer*). Na Figura 19 está representado um exemplo de rasterização de um triângulo após sua projeção no plano da imagem, sendo que os lados do triângulo estão fragmentados em pixels.

Apesar de possuir muitas propriedades cujos nomes são os mesmos que os dos vértices, muitas destas propriedades dos fragmentos não têm qualquer relação com as dos vértices. Por exemplo, a posição de um vértice é passada para a GPU pelo programa OpenGL e pode ser alterada pelo *vertex program*, enquanto que a posição de um fragmento é calculada durante o processo de rasterização e não pode ser alterada pelo *fragment program*.

Diferentemente de um *vertex program*, um *fragment program* só pode retornar componentes da semântica **COLOR**, ou seja, vetores de 1 a 4 posições. Tendo isto em mente, podemos apresentar nosso último exemplo de *shader*: um *fragment program* que aplica uma textura ao nosso triângulo transformado pelo *vertex program* da Figura 16. Tal programa é apresentado na Figura 20.

```
/* Texture Fragment Cg Program */

float4 Test3f_texture(float2 texCoord : TEXCOORD0,

                      uniform sampler2D decal) : COLOR
{

  float4 color = tex2D(decal, texCoord);

  return color;

}
```

Figura 20 - Exemplo de *fragment program*.

Note que não precisamos mais da declaração da estrutura de retorno e, ao invés dela, associamos o valor retornado à semântica **COLOR** adicionando a declaração **:COLOR** logo após o término da declaração dos parâmetros de entrada da função.

Uma vez que neste nosso exemplo utilizaremos tanto um *vertex program* quanto um *fragment program*, é necessário que todos os dados requeridos aos cálculos das propriedades utilizados pelo *fragment program* sejam retornados pelo *vertex program*. Quando sobrescrevemos apenas o *fragment program*, não há a necessidade de se preocupar com isto. Neste caso, então, como o exemplo de *fragment program* utiliza as coordenadas de textura dos fragmentos, é necessário que o *vertex program* contenha em sua estrutura de retorno os dados sobre as coordenadas de textura dos vértices. Para tal, devemos modificar *vertex program* conforme mostra a Figura 21.

```
/* Simple Cg Program */
struct Test3v_Output
{
     float4 position : POSITION;
     float2 texCoord : TEXCOORD0;
     float3 color    : COLOR0;
};

Test3v_Output Test3v_green( float2 position : POSITION, float2 texture : TEXCOORD0,
                            uniform float4x4 modelViewProj )
{
     Test3v_Output OUT;
        OUT.position = mul(modelViewProj, float4(position,0.,1.));
        OUT.color    = float3(0.,1.,0.);
        OUT.texCoord = texture;

     return OUT;
}
```

Figura 21 - Modificações necessárias ao *vertex program*.

Após realizadas estas pequenas modificações, precisamos criar e inicializar um *profile*, um programa e uma parâmetro Cg para o *fragment program*, dentro do código OpenGL, de forma muito semelhante à realizada durante a criação e inicialização do *vertex program*, conforme apresentado nas Figura 22 e Figura 23.

```
static CGcontext   myCgContext;          // A Context To Hold Our Cg Program(s)

static CGprofile   myCgVertexProfile,    // The Profile To Use For Our Vertex Shader
                   myCgFragmentProfile; // The Profile To Use For Our Fragment Shader

static CGprogram   myCgVertexProgram,    // Our Cg Vertex Program
                   myCgFragmentProgram; // Our Cg Fragment Program

static CGparameter myCgVertexParam_modelViewProj, // Vertex Program External Parameter
                   myCgFragmentParam_decal;       // Fragment Program External Parameter

int main(int argc, char *argv[])
{
    ...
    glEnable(GL_TEXTURE_2D);
    GLuint textureId;
        glGenTextures(1,&textureId);
    load_texture("bunny.jpg",256,256,textureId);

    glClearColor(1,1,1,1);

    myCgContext = cgCreateContext();             // Create A New Context For Our Cg Program(s)
      checkForCgError("creating context");
      cgGLSetDebugMode(CG_FALSE);
      cgSetParameterSettingMode(myCgContext, CG_DEFERRED_PARAMETER_SETTING);
    ...
    // Fragment Profile and Program initialization
    myCgFragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
      cgGLSetOptimalOptions(myCgFragmentProfile);
      checkForCgError("selecting fragment profile");

    myCgFragmentProgram = cgCreateProgramFromFile(
                                myCgContext,              /* Cg runtime context */
                                CG_SOURCE,                /* Program in human-readable form */
                                "Test3f.cg",              /* Name of file containing program */
                                myCgFragmentProfile,      /* Profile: OpenGL ARB vertex program */
                                "Test3f_texture",         /* Entry function name */
                                NULL);                    /* No extra compiler options */
    checkForCgError("creating fragment program from file");
    cgGLLoadProgram(myCgFragmentProgram);
    checkForCgError("loading fragment program");

    myCgFragmentParam_decal = cgGetNamedParameter(myCgFragmentProgram, "decal");
    checkForCgError("getting decal parameter");

    cgGLSetTextureParameter(myCgFragmentParam_decal, textureId);
    checkForCgError("setting decal 2D texture");

    glutMainLoop();

    return EXIT_SUCCESS;
}
```

Figura 22 - Declaração e inicialização de um *profile*, um programa e um parâmetro para o *fragment program*.

Assim como no *vertex program*, inicializamos o objeto do tipo **`CGprogram`** com o retorno da função **`cgCreateProgramFromFile`** que recebe, dente outras informações, o nome do arquivo no qual está descrito o código Cg e o nome da função de entrada. Criamos um objeto **`CGparam`** para conter o identificador da textura e o inicializamos com o retorno do método **`cgGetNamedParameter`**, que recebe como parâmetros o objeto **`CGprogram`** referente ao *fragment program* e o nome da variável dentro deste programa.

Após a inicialização do objeto **`CGparam`** que conterá o identificador da textura, associamos tal identificador ao parâmetro utilizando o método **`cgGLSetTextureParameter`**. Tal identificador é o número retornado pelo método **`glGenTextures`**, responsável pelo gerenciamento dos identificadores de texturas dentro do contexto OpenGL.

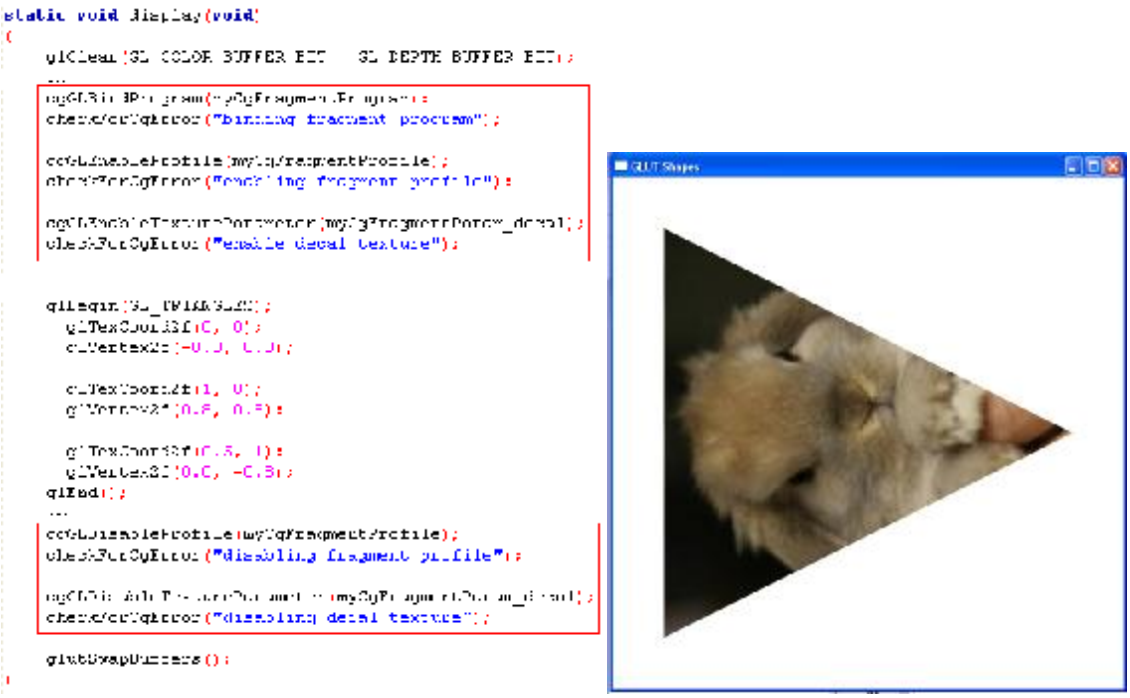

Figura 23 – À esquerda: ativação do *profile*, do programa e da textura para o *fragment program*. À direita: resultado da utilização do *vertex program* e do *fragment program* descritos neste tutorial.

Uma vez inicializados corretamente, assim como o *vertex program*, é necessário habilitarmos o *profile*, o programa e a textura associados ao *fragment program* antes do processamento dos objetos 3D e desabilitá-los ao final do processo.

O resultado da execução do programa descrito pode ser visto na Figura 23, na qual cada fragmento do triângulo rotacionado teve sua cor alterada de acordo com a sua coordenada de textura e a cor do pixel da imagem do coelhinho.

5. GPGPU

Como vimos nas seções anteriores, as GPUs a partir da terceira geração permitem que alguns estágios do *pipeline* gráficos sejam programados. Com o tempo, novas instruções e capacidades foram sendo adicionadas e as GPUs se tornaram capazes de realizar bem mais do que cálculos gráficos específicos para os quais foram desenvolvidas. Atualmente elas são co-processadores eficientes e sua alta velocidade as tornam úteis em uma grande variedade de aplicações.

As GPUs já foram utilizadas, com grandes ganhos de desempenho, em aplicações como Processamento de Sinais, Busca e Ordenamento, Visão Computacional, Compressão de Dados, dentre outras [REF - GPGPU]. De modo genérico, costumamos dar o nome de GPGPU (*general-purpose computation on GPU*) a toda e qualquer utilização da GPU diferente daquela para a qual foi desenvolvida originalmente, ou seja, qualquer aplicação não gráfica.

O *hardware* da GPU, altamente especializado para matrizes e vetores, faz com que ela seja muito mais rápida que a CPU em cálculos com dados desse tipo. Por outro lado, a CPU é mais eficiente que a GPU em cálculos com valores escalares.

Outra diferença diz respeito à precisão: o tipo *float* da GPU possui 4 *bytes* de precisão e é o máximo que ela suporta por *hardware*; já na CPU podemos realizar cálculos com o dobro (o *double* da linguagem C) e com o quádruplo (o *long double* da linguagem C) de precisão, o que torna a GPU não recomendada para cálculos que necessitem de uma precisão muito alta.

De maneira simplificada, um bom algoritmo para ser rodado em uma GPU tipicamente costuma:

- Envolver uma grande quantidade de dados, e que estes sejam, ou possam ser, representados por matrizes ou vetores;
- Ser fácil ou naturalmente paralelizável, isto é, o processamento de uma parte da massa de dados independe de todas as outras, o que permite que o *pipeline* gráfico fique quase ou totalmente preenchido ao longo da execução.

Para exemplificar, iremos mostrar como programar o Operador de Sobel numa GPU. Este operador é muito utilizado em processamento de imagens, especialmente para detecção de bordas.

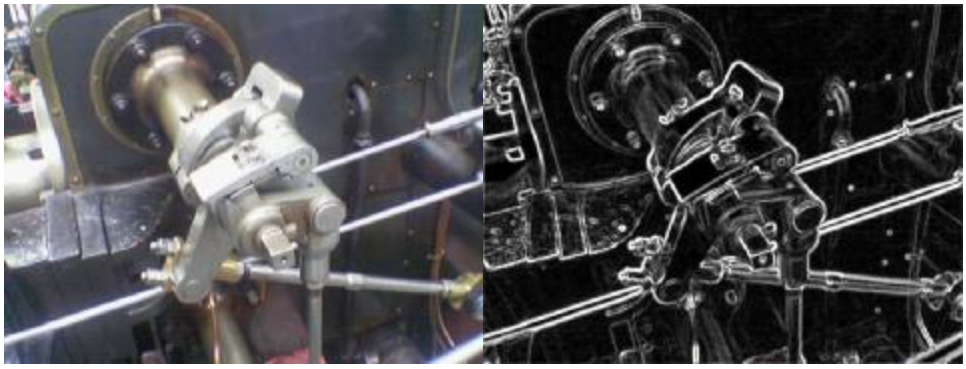

Figura 24 – À esquera, imagem original. À direita, imagem resultante após aplicação do Operador de Sobel.

Formalmente, o Operador de Sobel pode ser definido como:

$$G_X(i, j) = -P(i-1, j-1) - 2P(i-1, j) - P(i-1, j+1) + P(i+1, j-1) + 2P(i+1, j) + P(i+1, j+1)$$

$$G_Y(i, j) = -P(i-1, j-1) - 2P(i, j-1) - P(i+1, j-1) + P(i-1, j+1) + 2P(i, j+1) + P(i+1, j+1)$$

$$P(i, j) = |G_X| + |G_Y|$$

onde $P(i, j)$ representa o pixel da linha *i* e coluna *j* de uma imagem.

Ou seja, o operador utiliza dois *kernels* de processamento como mostrado na Figura 25. Os números nos *kernels* representam os pesos dos *pixels* mais próximos no cálculo dos valores G_X e G_Y de um *pixel* qualquer.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figura 25 – Os dois *kernels* de processamento do Operador de Sobel.

Uma primeira preocupação quando vamos fazer cálculos genéricos na GPU é como passar os seus dados para a GPU. Normalmente usamos texturas para esse fim, de modo que cada *texel* (*pixel* de textura) representa um dos dados a serem processados.

E como obter os dados calculados após o término da execução? A GPU escreve os resultados de seus cálculos em um *framebuffer*, e é daí que devemos retirar os resultados. O resultado do nosso processamento será a imagem processada pelo operador de Sobel, que será renderizada e copiada do *framebuffer*.

Outra questão fundamental é como garantir que a GPU irá processar todos os seus dados. Como vimos anteriormente, um *vertex program* é executado exatamente uma vez para cada vértice presente na pirâmide de projeção; e um *fragment program*, uma vez para cada fragmento gerado na etapa de rasterização. Como deveremos ler os resultados do *framebuffer*, nada mais natural que usar *fragment programs*, pois os fragmentos estão muito mais próximos do que efetivamente será desenhado no *framebuffer* que os vértices.

Assim, um processamento genérico na GPU possui os seguintes passos:

- Passar os dados a serem processados para a GPU no formato de uma textura;
- Criar uma cena 3D que garanta que cada fragmento corresponda a cada dado a ser processado;
- Ler o resultado do *framebuffer*.

Vimos nas seções anteriores como passar texturas como parâmetros para a GPU. Veremos como fazer o segundo e o terceiro passos a seguir.

Se os nossos dados são representados por uma textura de tamanho $m \times n$, fazer o segundo passo criando uma janela com o mesmo tamanho da textura e renderizando um quadrado texturizado com os nossos dados preenchendo todas a janela. Como a janela é totalmente preenchida por um quadrado, na etapa de rasterização, serão gerados um fragmento para cada pixel da janela, que corresponde por sua vez, a cada um de nossos dados.

O terceiro passo pode ser feito facilmente em OpenGL, com as funções **`glReadBuffer`** e **`glReadPixels`**.

```
70  static void display(void)
71  {
72      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
73
74      cgGLBindProgram(gCgProgram);
75      cgGLEnableProfile(gCgProfile);
76      cgGLEnableTextureParameter(gCgTexParam);
77
78      glViewport(0, 0, width, height);  // Este código renderiza a imagem
79      glMatrixMode(GL_PROJECTION);      //  preenchendo toda a janela,
80      glLoadIdentity();                 //  de modo que cada pixel da
81      gluOrtho2D(-1, 1, -1, 1);         //  janela (fragmento) corresponda
82      glMatrixMode(GL_MODELVIEW);       //  a cada pixel da imagem
83      glLoadIdentity();                 //  processada (texel)
84
85      glBegin(GL_QUADS);
86          glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
87          glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
88          glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
89          glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
90      glEnd();
91
92      cgGLDisableTextureParameter(gCgTexParam);
93      cgGLDisableProfile(gCgProfile);
94
95      glReadBuffer(GL_BACK);    // Copia o resultado
96      glReadPixels(0,0,width,height,GL_RED,GL_UNSIGNED_BYTE,result);
97  }
```

Figura 26 – Calculando o Sobel e lendo o resultado.

Na Figura 26, o código entre as linhas 78 e 90 desenha um quadrado preenchendo toda a janela, e as linhas 95 e 96 copiam o resultado do *framebuffer*, conforme discutido anteriormente.

O *fragment program* usado nesse exemplo pode ser visto na Figura 27.

```
/* Sobel operator on GPU - Fragment program */
float4 sobel(float2 coords : TEX0,

             uniform float2    offset,
             uniform sampler2D texture) : COLOR
{
    /* t = top    b = bottom
       l = left   r = right

       tl     t    tr
        l   pixel   r
       bl     b    br */

  float4 bl = tex2D(texture, coords - offset);
  float4 tr = tex2D(texture, coords + offset);

  float4 l  = tex2D(texture, coords + float2(-offset.x,         0));
  float4 tl = tex2D(texture, coords + float2(-offset.x,  offset.y));
  float4 t  = tex2D(texture, coords + float2(      0,    offset.y));

  float4 r  = tex2D(texture, coords + float2( offset.x,         0));
  float4 br = tex2D(texture, coords + float2( offset.x,  offset.y));
  float4 b  = tex2D(texture, coords + float2(      0,   -offset.y));

  float4 gx = -tl + tr - 2*l + 2*r - bl + br;
  float4 gy =  tl + 2*t + tr - bl - 2*b - br;

  return abs(gx) + abs(gy);
}
```

Figura 27 – Operador de Sobel na GPU.

Implementamos também o Operador de Sobel utilizando apenas a CPU e comparamos os resultados. A máquina de teste possuía um processador AMD Athlon 64 x2 5000+, 2GB de memória RAM e placa GeForce 8600 GT 512MB. Foram feitos

testes com imagens de tamanhos 32x32, 128x128, 512x512 e 2048x2048. A mostra Figura 28 os resultados.

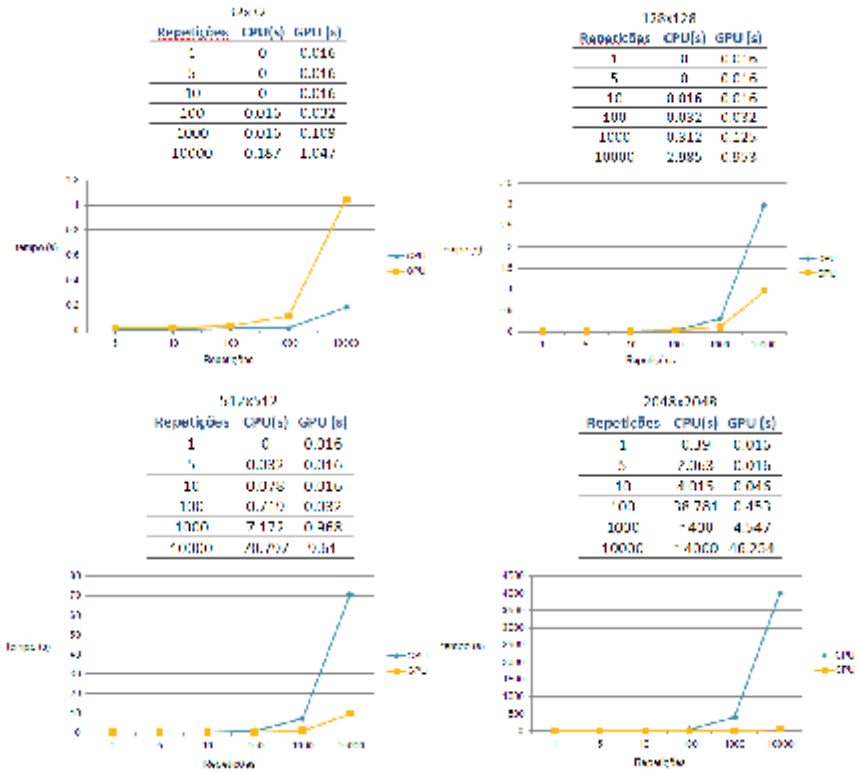

Figura 28 - Benchmark CPU x GPU

Como podemos ver a GPU se sai muito melhor em todos os casos, menos nas imagens pequenas 32x32. Nesse caso, as operações de transferência de dados entre a memória da CPU e da GPU consomem muito mais tempo que os cálculos em si.

Figura 29 – Resultados do Operador de Sobel. Da esquerda para direita: imagem original, imagem processada pela CPU e imagem processada pela GPU.

As pequenas diferenças visuais nos resultados da CPU e GPU que podem ser vistas na Figura 29 se devem justamente a diferença de precisão entre os dois processadores.

6. CUDA

Até a terceira geração de GPUs, não era possível alterar o *pipeline* de processamento, apenas alguns estágios específicos. A linguagem Cg reflete essa limitação, permitindo apenas a criação de programas que rodem como *vertex* ou *fragment programs*. Com o advento da arquitetura unificada esse limite foi quebrado, e surgiu uma nova forma de programação para GPU, o CUDA (*Compute Unified Device Architecture*).

Diferentemente do Cg, o CUDA não é uma nova linguagem de programação, mas uma biblioteca C. Com restrições, qualquer código C pode rodar na GPU.

No CUDA, a GPU é vista como um co-processador genérico, e não apenas gráfico, não sendo mais necessário criar um contexto OpenGL para acessá-la.

7. References

Cg Toolkit – User's Manual: a Developer's Guide to Programmable Graphics, release 1.4, September 2005;

The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, online version at http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html, November 2008;

Cuda Zone, available at http://www.nvidia.com/cuda, November 2008;

Kilgariff, E., Fernando, R., GPU GEMs 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation, chapter 30 - The GeForce 6 Series GPU Architecture, online version, 2005;

General-Purpose Computation Using Graphics Hadware, available at http://www.gpgpu.org, November 2008;

Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview, November 2006;

Luebke, D., GPU Architecture: Implications & Trends, presented on SIGGRAPH2008;

Wikipedia: Graphics processing unit, available at http://en.wikipedia.org/wiki/Graphics_processing_unit, November 2008;

Wikipedia: Video Card, avaiable at http://en.wikipedia.org/wiki/Video_card, November 2008;