Displacement Stage: Arquitetura extensível para mapas dinâmicos de deslocamento na GPU

Fábio Corato de Andrade Mani Shafaatdoost* Aura Conci Esteban W. G. Clua

Universidade Federal Fluminense, Instituto de Computação, Brasil
* Florida International University, School of Computing and Information Sciences, U.S.A.

Abstract

Tessellation and Displacement Mapping are well known methods in Computer Graphics, but rarely used in real-time application because of their high processing cost which directly affects run-time performance. However, progress in hardware presented new features in graphic pipeline. These techniques are getting stronger and are conquering their own position in real-time application and especially in digital games. This article proposes an extensible architecture which is responsible for managing and for applying displacement maps on GPU. This architecture composed of several kernels with specific functions. These kernels are extendible and customizable and it is possible to solve several kinds of problems by using this method. The proposed solution combined different techniques and uses a dynamic approach to control displacement maps which are used to influence geometries directly on GPU.

Keywords: Software Architecture; Displacement Mapping; GPU; Digital Games; Tessellation; Morphing.

Authors' contact:
{fcorato,aconci,esteban}@ic.uff.br
mshaf012@cs.fiu.edu

1. Introdução

Este trabalho tem como objetivo propor uma arquitetura extensível que visa retirar do desenvolvedor a responsabilidade de controle sobre os mapas de deslocamento, visando assim reduzir a complexidade e modularizar suas funções em um modelo de controle de fácil aprendizagem, domínio, utilização, extensão e manutenção. O foco está em criar um padrão de projeto que possa ser reutilizado quando as implementações precisarem fazer alterações em tempo real em malhas 3D que representem detalhes. Para que essa arquitetura possa existir é necessário que a *tessellation* e o *displacement mapping* estejam implementados no *shader*, utilizando-se os novos estágios disponíveis nas versões mais recentes das APIs gráficas. Também será necessário explorar a interoperabilidade existente entre as APIs e as arquiteturas para computação paralela nas GPUs.

As principais contribuições alcançadas neste trabalho foram: a separação da arquitetura em um modelo de camadas, permitindo que cada camada possa ser trabalhada de forma independente; a separação de cada funcionalidade em núcleos específicos, possibilitando que vários tipos de deformação sejam utilizados e combinados pelos desenvolvedores; a possibilidade de expandir a arquitetura através da criação de *kernels* personalizados, garantindo a flexibilidade da arquitetura.

2. Arquitetura

Apresentamos uma arquitetura que tem um módulo responsável por gerenciar um conjunto de *kernels* pré-fabricados. Esta estrutura permite que o desenvolvedor personalize seus *kernels* para que a arquitetura cresça conforme a necessidade de uso. Quando uma textura atuar como um mapa de deslocamento, o resultado será obtido através das operações feitas nos *kernels* aqui propostos, evitando que os *shaders* precisem ser programados para tais funções. O processamento feito na GPU, a utilização da interoperabilidade para manter a fluidez de dados em memória da GPU e a função específica de controle do mapa de deslocamento constituem a arquitetura extensível aqui proposta.

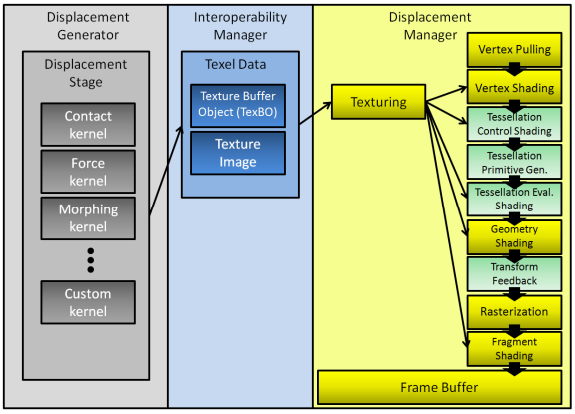

Figura 2.1: Displacement Stage: modelo em camadas.

A figura 2.1 mostra o modelo dividido em três camadas, as quais foram denominadas: Displacement Generator, responsável pelo módulo Displacement Module, o qual utiliza os *kernels* para manipular os mapas de deslocamento; Interoperability Manager, responsável por manter os mapas de deslocamento acessíveis para as demais camadas; e Displacement Manager, responsável por usar os mapas de deslocamento para influenciar a geometria ampliada pela *tessellation*.

2.1 Módulo de Contato

O módulo de contato é um *kernel* que representa casos de deformações geradas por contato entre duas superfícies: a superfície da malha a ser alterada e a superfície da entidade que a altera. Uma entidade do jogo, ao se locomover pelo ambiente, mantém contato com o mesmo, influenciando a geometria da malha em escala definida pelo desenvolvedor, de forma que a geometria do ambiente é alterada pelo contato com a entidade. O resultado é um rastro deixado no mapa de deslocamento pela entidade do jogo, o qual é usado para alterar a geometria ambiente, conforme mostrado na figura 2.2.

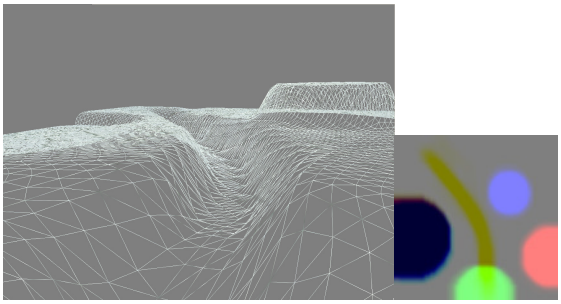

Figura 2.2: Resultado da geometria (esquerda), perspectiva, e mapa de deslocamento (direita), vista superior, após o uso do contact kernel.

O pseudocódigo abaixo representa a função usada no módulo de contato da arquitetura proposta. Esse *kernel*, como observado na figura 2.2 (direita), influenciou o mapa de deslocamento, deixando um rastro que inicia na parte central inferior, próximo ao círculo verde, e finalizando próximo à parte esquerda superior. A figura 2.2 (esquerda), mostra na geometria um caminho que corresponde ao rastro criado pelo uso do *kernel*.

```
kernel contact(image in, image out, float2
pos, float depth, float radius)
{
  float4 img = read_image(in,sampler,coords);
  float dist = distance(pos,coords);
  if(dist < radius)
    img.z -= depth;
  write_image(out,coords,img);
}
```

2.2 Módulo de Força

O módulo de força é um *kernel* que representa casos de deformações geradas por forças externas, podendo essas forças estar sob ou sobre a superfície da malha a ser alterada. Visando adequar os resultados das deformações ao contexto no qual forem utilizadas, optou-se por um vetor de quatro componentes para representar a força nesse *kernel*. As três primeiras componentes desse vetor de força são utilizadas normalmente para se extrair as informações características (módulo, direção e sentido), enquanto a quarta componente pode ser utilizada para influenciar os resultados, multiplicando a grandeza.

O resultado do *kernel* é uma deformação deixada no mapa de deslocamento pela ocorrência escolhida, a qual é usada para alterar a geometria do ambiente, conforme mostrado na figura 2.3.

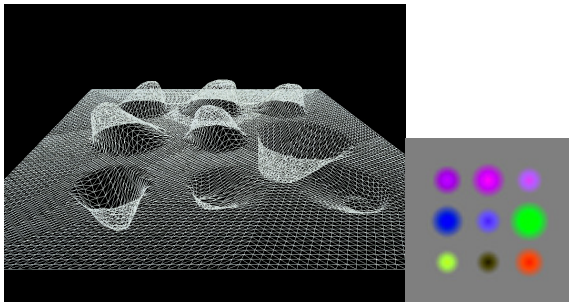

Figura 2.3: Resultado da geometria (esquerda) e mapa de deslocamento (direita) após o uso do force kernel.

O pseudocódigo a seguir mostra o algoritmo usado para a criação da imagem apresentada na figura 2.3. Ele recebe como parâmetros a posição onde a força será aplicada e o vetor que representa a própria força. Optou-se por calcular a área afetada utilizando o tamanho do vetor que representa a força, multiplicado por uma constante, a qual serve para adequar a escala da deformação ao ambiente que será afetado. Pode-se observar que o parâmetro força é do tipo *float4* e é exatamente nessa quarta componente que o valor da constante é enviado para ser usado na função.

```
kernel force(image in, image out, float2 pos,
float4 force)
{
  float4 img = read_image(in,sampler,coords);
  float area = length(force.xyz) * CONSTANT;
  float dist = distance(pos,coords);
  float dissipation = 1 - dist / area;
  if(dissipation > 0)
    img += force * dissipation;
  write_image(out,coords,img);
}
```

2.3 Módulo de Morphing

O módulo de *morphing* deforma a malha por transição usando um algoritmo de *morphing* linear. Ele simula o *morphing* 3D, pois o resultado encontrado é a deformação da geometria por interpolação ponto a ponto de seus vértices; porém esse resultado é alcançado influenciando o mapa de deslocamento, o qual é 2D. O pseudocódigo abaixo mostra como o módulo de *morphing* consegue o resultado descrito.

```
kernel morphing(image in1, image in2, write
image out, float time)
{
  float4 i1 = read_image(in1,sampler,coords);
  float4 i2 = read_image(in2,sampler,coords);
  float4 img  = i1 * (1-time) + i2 * time;
  write_image(out,coords,img);
}
```

O uso prático do módulo de *morphing*, denominado *morphing kernel* na figura 2.1, consiste na utilização de pelo menos dois mapas de deslocamento que representam o estado inicial e final da topografia. Essa alteração, feita de forma dinâmica durante um jogo, revela novas áreas para exploração do jogador, como: cavernas, fossos, esconderijos e outras áreas inicialmente ocultas ou inacessíveis, como ilustrado na figura 2.4.

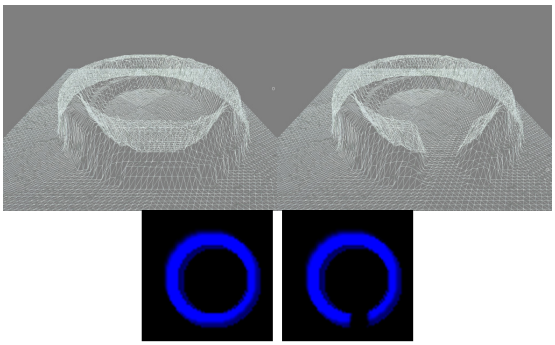

Figura 2.4: Geometria e mapa de deslocamento inicial (esquerda) e geometria e mapa de deslocamento final (direita).

2.4 Módulo Personalizado

O módulo personalizado, denominado de *custom kernel* na arquitetura apresentada neste artigo, é o módulo previsto como modelo para ser editado pelos desenvolvedores. Esse módulo é livre para receber quaisquer personalizações que possam ser pensadas e codificadas. O trabalho consiste inicialmente em editar ou adicionar um *custom kernel* no arquivo que contem todas as funções referentes aos demais kernels e nomeá-lo de acordo com sua funcionalidade. Nesse arquivo existe um *custom kernel* preparado com a estrutura mostrada abaixo.

```
kernel custom(image in, image out, float4
arg1, float4 arg2, float4 arg3, float4 arg4)
{
  float4 img = read_image(in,sampler,coords);
  // INSERT CODE HERE!
  write_image(out,coords,img);
}
```

Esse *kernel* encontra-se preparado para trabalhar com o mapa de deslocamento, o qual deve ser passado como parâmetro de entrada e saída de imagem. Ele também possui quatro parâmetros *float4*, nos quais podem ser passados até dezesseis valores para utilização, conforme a funcionalidade desejada. Essa quantidade de valores está acima da quantidade máxima usada nos *kernels* apresentados anteriormente, garantindo que qualquer um deles possa ser reescrito com o módulo personalizado.

O módulo personalizado poderá abranger funcionalidades complementares aos kernels existentes, conforme citado em [ANDRADE 2012].

3. Implementação

O diagrama apresentado na figura 3.1 representa o modelo em camadas referente à figura 2.1, considerando aspectos de padrão de projetos e o paradigma orientado a objetos. Neste trabalho o diagrama de classe serve para registrar a estrutura de classes utilizada, sendo possível através do mesmo observar detalhes da arquitetura, inclusive sob o ponto de vista dos desenvolvedores.

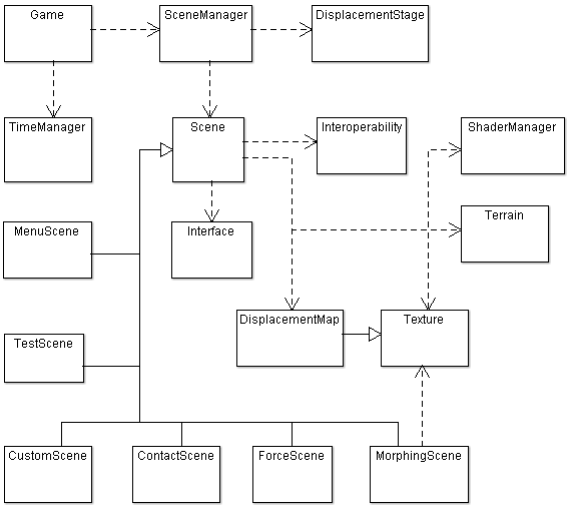

Figura 3.1: Diagrama de classes.

A descrição de cada classe, detalhando aspectos do funcionamento da arquitetura, pode ser encontrada em [ANDRADE 2012]. Como se pode observar, as partes mutáveis que precisam de maior atenção na implementação são os *kernels*, mas com o uso da arquitetura extensível, de acordo com a utilização proposta, basta escolher o *kernel* mais adequado para a função desejada ou adaptar o *custom kernel*. Nessa seção será mostrado o resultado do caso de teste do *custom kernel*, renomeado para *test kernel*, implementado na classe TestScene.

Para comprovar o funcionamento da aplicação do *custom kernel*, optou-se por mostrar uma implementação capaz de manipular o mapa de deslocamento através de uma interface simplificada, a qual utiliza mouse, teclado e o próprio mapa de deslocamento. O pseudocódigo abaixo pode ser observado e comparado ao apresentado na seção 2.4, pois é uma adaptação do *custom kernel*.

```
kernel test(image in, image out, float4 arg1,
float4 arg2, float4 arg3, float4 arg4)
{
  float4 img = read_image(in,sampler,coords);
  // INSERT CODE HERE!
  float dist = distance(arg1.xy,float2(x,y));
  if(dist < arg3.x)
    img += arg2;
  write_image(out,coords,img);
}
```

Na classe TestScene o *kernel* recebeu como parâmetros os argumentos: mouse, normal, radius e null respectivamente correspondentes à arg1, arg2, arg3 e arg4, mostrados no *custom kernel* acima.

O mapa de deslocamento foi colocado no canto superior esquerdo da tela e pode ser manipulado diretamente. Os dispositivos de entrada ajudam no controle das variáveis envolvidas, como o raio, a normal e a posição. A sequência de imagens abaixo demonstra cada etapa do processo, conforme consta em suas descrições.

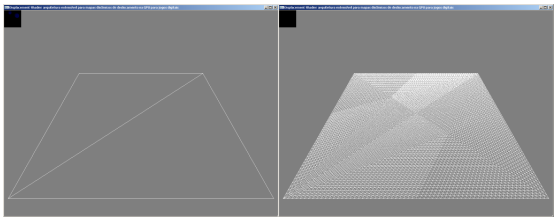

Figura 3.2: Malha original (esquerda) e malha após a tessellation (direita).

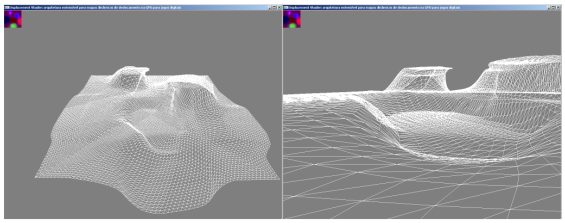

Figura 3.3: Malha após displacement mapping: visão distante (esquerda) e próxima (direita).

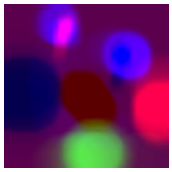

Figura 3.4: Mapa de deslocamento gerado pelo test kernel.

Como visto nas figuras acima, o exemplo demonstra que uma geometria simples, como um plano, pode ser passada para o *pipeline* gráfico, tornando-se uma malha densa, após passar pelo processo de *tessellation*. Nessa malha é aplicada a técnica de *displacement mapping*, que se responsabiliza por fazer as deformações que representam a geometria desejada. O mapa de deslocamento usado pode ser alterado dinamicamente nesse exemplo, conforme anteriormente citado, gerando diversas combinações.

O gráfico apresentado na figura 3.5 mostra que a implementação da TestScene obteve taxas interativas de *fps*, possibilitando que a arquitetura seja utilizada em tempo real. O resultado das demais cenas pode ser encontrado em [ANDRADE 2012], assim como informações de configuração do equipamento usado para os testes, os tempos gastos por cada kernel e outras etapas do processo.

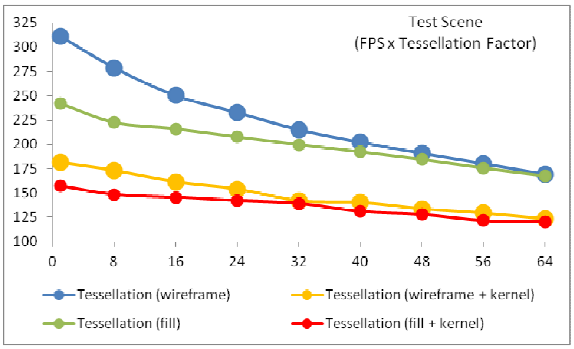

Figura 3.5: Resultados de testes.

4. Conclusão

A arquitetura proposta é um modelo que se utilizado de forma adequada, pode trazer benefícios e colaborar com o desenvolvimento de efeitos mais interessantes. Qualquer variação de kernel pensada pode ser escrita como um módulo personalizado e só tende a ajudar a enriquecer as aplicações e ampliar a própria arquitetura. Como já descrito, as características da arquitetura: processamento feito na GPU, utilização da interoperabilidade e funcionalidade dedicada, aliado ao modelo em camadas, fazem dela, do ponto de vista técnico, uma ferramenta simples de ser entendida, dominada, aplicada, estendida e mantida. Sua reutilização e, consequentemente, sua expansão mostram-se promissoras e quanto mais desenvolvedores colaborarem e compartilharem suas funções personalizadas, mais forte ela se tornará.

Referências

ANDRADE, F.C. AND CLUA, E.W.G., 2011. USING REAL TIME HARDWARE TESSELLATION FOR MORPHING OF GEOMETRY IN GPU. IN: II WORKSHOP ARGENTINO SOBRE VIDEOJUEGOS - WAVI 2011, BUENOS AIRES. ACTAS DEL SEGUNDO WORKSHOP ARGENTINO SOBRE VIDEOJUEGOS - WAVI 2011. BAHIA BLANCA : EDITORIAL DE LA UNIVERSIDAD NACIONAL DEL SUR, V.2. P.49-63.

ANDRADE, F.C., 2012. UMA PROPOSTA DE ARQUITETURA EXTENSÍVEL PARA MAPAS DINÂMICOS DE DESLOCAMENTO NA GPU PARA JOGOS DIGITAIS. DISSERTAÇÃO (MESTRADO EM COMPUTAÇÃO), UNIVERSIDADE FEDERAL FLUMINENSE, RIO DE JANEIRO.