UNIVERSIDADE FEDERAL FLUMINENSE

FÁBIO CORATO DE ANDRADE

Uma proposta de Arquitetura extensível para mapas dinâmicos de deslocamento na GPU para jogos digitais

NITERÓI

2012

UNIVERSIDADE FEDERAL FLUMINENSE

FÁBIO CORATO DE ANDRADE

Uma proposta de Arquitetura extensível para mapas dinâmicos de deslocamento na GPU para jogos digitais

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Computação Visual. Linha de pesquisa: Jogos e Entretenimento Digital.

Orientadora:

Prof. Aura Conci, D.Sc.

NITERÓI

2012

Uma proposta de Arquitetura extensível para mapas dinâmicos de deslocamento na GPU para jogos digitais

FÁBIO CORATO DE ANDRADE

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Computação Visual. Linha de pesquisa: Jogos e Entretenimento Digital.

Aprovada por:

_____

Prof. D.Sc. Aura Conci IC-UFF

_____

Prof. D.Sc. Esteban Walter Gonzalez Clua IC-UFF

_____

Prof. Ph.D. Marcelo de Andrade Dreux PUC-Rio

Niterói, 03 de Setembro de 2012.

Dedico este trabalho ao meu tio e padrinho (in memorium), Edson Fernandes Corato, um grande amigo, irmão e mestre.

Agradecimentos

Primeiramente agradeço a Deus, por tudo, mas principalmente, pelo nascimento do meu filho Lucas (28/10/2011).

À professora Aura, pela paciência, incentivo e publicações.

Ao professor Esteban, pela amizade, apoio e incentivo.

Aos meus professores de graduação e mestrado, que me fizeram crescer profissionalmente.

Aos amigos do mestrado, pelos momentos de estudo e descontração.

Aos demais amigos, os quais sempre me incentivaram a seguir em frente.

Ao meu irmão, Cristiano, por me fazer entender o que são desafios desde sempre.

Aos meus sobrinhos e afilhados, que me prepararam para ser pai.

Ao meu filho, Lucas, por me fazer sorrir sempre que estou em casa.

À minha esposa, Juliana, pela paciência, apoio e carinho; por tudo.

Aos meus pais, Djalma e Cilene, que deram o seu melhor na minha criação.

Resumo

Tessellation e *Displacement Mapping* são métodos conhecidos em Computação Gráfica, mas pouco usados em aplicações em tempo real devido ao elevado custo de processamento, o que impacta diretamente no desempenho das implementações. Com o avanço dos *hardwares*, que apresentam novos estágios no pipeline gráfico, estas técnicas estão ganhando força e conquistando seus espaços nas aplicações em tempo real, especialmente nos jogos digitais. Este trabalho propõe uma arquitetura extensível que se responsabiliza por gerenciar e aplicar mapas de deslocamento em GPU. Esta arquitetura é composta por diversos *kernels* especializados em determinadas funções. Embora estes *kernels* possam ser estendidos e personalizados, é possível cobrir vários tipos de efeitos usando os que se apresentam neste trabalho. A proposta é de uma abordagem dinâmica, executando o controle sobre os mapas de deslocamento usados para influenciar as geometrias apresentadas diretamente na GPU, combinando diferentes técnicas. Apesar desta arquitetura permitir uma melhora no desempenho das aplicações, seu principal objetivo é a possibilidade de isolar e reutilizar o trabalho relacionado ao mapa de deslocamento, retirando tal responsabilidade dos *shaders* convencionais, facilitando a aplicação e o controle dos relevos e permitindo a utilização de várias aplicações numa mesma implementação.

Palavras-chave: Arquitetura de Software; Mapeamento de Deslocamento; Unidades de Processamento Gráfico; Jogos Digitais; Mosaico; Transformação.

Abstract

Tessellation and Displacement Mapping are methods known in Computer Graphics, but rarely used in real-time application because of their high cost of processing, which directly affects their implementation performance. However, the hardware progress has presented new stages in the graphic pipeline, and so these techniques are getting stronger and are conquering their own spaces in real-time application, especially in digital games. This work proposes an extensible architecture that is responsible for managing and for applying displacement maps on the GPU. This architecture is composed of several kernels with specific functions. Although these kernels can be extended and customized, it is possible to cover several kinds of effects using the ones presented in this work. The proposition is a dynamic approach, running that controls the displacement maps that are used to influence the geometries directly presented on the GPU, by combining different techniques. Although this technique provides an improvement in application performance, its main goal is the possibility to isolate and to reuse the work related to the displacement map, by taking the responsibility from the conventional shaders, and by improving the relief's application and control, and also by allowing the use of many applications in only one implementation.

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

Siglas e Abreviações

API	:	*Application Programming Interface*
CPU	:	*Central Processing Unit*
DDM	:	*Deformation Displacement Maps*
FPS	:	*Frames per second*
GPU	:	*Graphics Processing Unit*
IDE	:	*Integrated Development Environment*
LOD	:	*Level of Detail*
OpenCL	:	*Open Computing Language*
OpenGL	:	*Open Graphics Library*
POM	:	*Parallax Occlusion Mapping*
RAM	:	*Random Access Memory*
UML	:	*Unified Modeling Language*

Sumário

Lista de Figuras

Capítulo 1

Introdução

Para os desenvolvedores de *software* em geral, a utilização de padrões de projetos é cada vez mais explorada, principalmente pela necessidade de um controle mais apurado sobre seu trabalho, devido à necessidade de atender prazos cada vez mais curtos. Os desenvolvedores de jogos também estão incluídos nesse grupo e segundo Rabin [RABIN 2012], na área de jogos, muitas empresas estabelecem um padrão de codificação para facilitar a colaboração entre programadores. Esse padrão é um documento que descreve as diretrizes que os programadores devem seguir ao escreverem um código, sendo o padrão utilizado variado de empresa para empresa ou de um tipo de projeto para outro. Em relação ao rigor dos conteúdos da norma, algumas empresas a tratam como um conjunto sugerido de diretrizes que os profissionais podem querer seguir, enquanto outras a tratam como *frameworks* pré-definidos. Embora essas afirmações façam parecer que na área de jogos não se esteja aplicando tão rigorosamente os padrões em seus projetos, a adesão ao uso de *frameworks* e *engines* reflete o contrário, uma vez que estes estão repletos de padrões em seus modelos estruturais. Cada vez mais ferramentas e soluções de prateleira são disponibilizadas para os desenvolvedores de jogos, e elas têm realmente conseguido ser aceitas tanto na academia quanto no mercado. Isso demonstra que os desenvolvedores estão abertos para experimentações e que existe a

necessidade de ferramentas mais simples de serem entendidas e utilizadas, mas que gerem também um resultado com a qualidade desejada.

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, neste trabalho representadas pelo OpenGL [OPENGL]. Também será necessário explorar a interoperabilidade existente entre as APIs e as arquiteturas para computação paralela nas GPUs, representadas no contexto deste trabalho pelo OpenCL [OPENCL].

Esta arquitetura tem como objetivo apresentar uma forma de representação de detalhes nas malhas 3D em tempo real usando OpenCL, OpenGL, GLSL [GLSL], *tessellation* e *displacement mapping*, ao mesmo tempo em que possibilita que informações dos mapas de deslocamento sejam usadas para diversos fins. O foco é permitir que a união dessas técnicas gere novas possibilidades para que os desenvolvedores possam explorá-las ao criarem seus jogos, sem, no entanto, perder desempenho e realismo, o que justifica a exploração dos recursos disponibilizados pelas novas arquiteturas de GPU.

Unidades de processamento gráfico (GPU) são processadores especializados em operações relacionadas com computação gráfica. São extremamente poderosos devido à sua arquitetura paralela e sua eficiência, tanto no acesso à memória como nas operações vetoriais e de interpolação. Atualmente, devido ao grande aumento de flexibilidade da arquitetura e das linguagens de programação, as GPUs estão sendo usadas para substituir as CPUs na resolução de diversos algoritmos clássicos (GPU Computing). O uso cada vez mais frequente da GPU para programação genérica se

deve à grande eficiência que o processador gráfico possui em determinados tipos de operações, superando nestes casos o desempenho da CPU. Na arquitetura proposta o uso da GPU é de extrema importância para manter o desempenho das aplicações, pois seus kernels assumirão a responsabilidade pelo trabalho nos mapas de deslocamento, o qual convencionalmente é feito no *shader*.

Os módulos da arquitetura serão implementados como *kernels* de OpenCL. Para a arquitetura proposta foram pensadas situações distintas de deformação e geração de malha, criando-se para cada uma destas um *kernel* particular. Neste trabalho foram definidas três situações distintas e complementares: deformação por contato, deformação por força e deformação por transição de malhas. Mesmo sendo possível mapear grande parte dos problemas que envolvem *displacement mapping* para alguma destas situações, também será possível estender a arquitetura criando *kernels* personalizados. Os *kernels* terão por objetivo primordial influenciar os mapas de deslocamento, os quais serão utilizados pelos estágios do shader responsáveis pelo *displacement mapping*. Usando esse processo, estruturas simples podem ser transformadas em topologias detalhadas e utilizadas em jogos digitais com o auxílio da *tessellation* em conjunto com o *displacement mapping*.

Tessellation é o processo de preenchimento completo de uma área utilizando regiões poligonais convexas, as quais seguem um padrão geométrico, sem deixar ocorrer espaçamento e sobreposição entre as regiões. Pode ser entendido como um método que divide polígonos em polígonos menores. Assim, ao dividir, por exemplo, um quadrado através de sua diagonal, obtém-se dois triângulos. Isolada, a *tessellation* não incrementa mais detalhes na geometria, pois um plano quando dividido em dois triângulos continua a ser um plano; porém, quando este processo é utilizado para criar novos triângulos que serão usados em conjunto com outras técnicas, fazendo com que essas novas divisões representem novas informações topológicas, alterando a geometria da malha para representar um detalhe, pode-se ganhar muito em realismo. A *tessellation* também pode ser usada para controlar LOD (*level of detail*) em tempo real e diminuir a necessidade de usar modelos muito detalhados, sem perder a qualidade, reduzindo a quantidade de informação enviada para a placa de vídeo, o

que costuma ser um gargalo em diversas aplicações gráficas. A arquitetura aqui proposta encapsula a implementação da *tessellation* em *shader*, recurso disponível com a inclusão dos novos estágios inseridos nas versões mais recentes das APIs gráficas e *hardwares* que as suportam.

A técnica de *Displacement Mapping* atua diretamente sobre os vértices da geometria, diferentemente das técnicas de *Texture Mapping*, *Bump Mapping* e *Parallax Mapping*, as quais atuam sobre os pixels. Essa técnica consiste no uso de uma textura como um mapa de deslocamento, que ao ser aplicado sobre uma geometria base, causa um efeito onde a posição atual dos vértices dessa geometria é deslocada em alguma direção (geralmente a direção da normal da geometria), seguindo parâmetros fornecidos pelo mapa. Esse processo tem como resultado uma geometria real completamente modificada em relação à usada como base, aumentando o detalhamento e a complexidade da geometria, permitindo sombras e silhuetas mais definidas e precisas. Esse processo é realmente possível quando a malha alterada possui um número suficiente de vértices, o que pode ser ajustado mediante o uso da *tessellation*. É possível controlar essas técnicas em conjunto, direcionando sua aplicação de acordo com a distância e detalhes da geometria, otimizando sua utilização. A figura 1.1 ilustra o processo descrito. A arquitetura aqui proposta conta com a implementação do *displacement mapping* em *shader* e isola o trabalho sobre os mapas de deslocamento nos *kernels* do OpenCL. Junto com o recurso de *tessellation* em tempo real, a arquitetura aqui proposta permite que as malhas originais não possuam um número muito grande de vértices e polígonos.

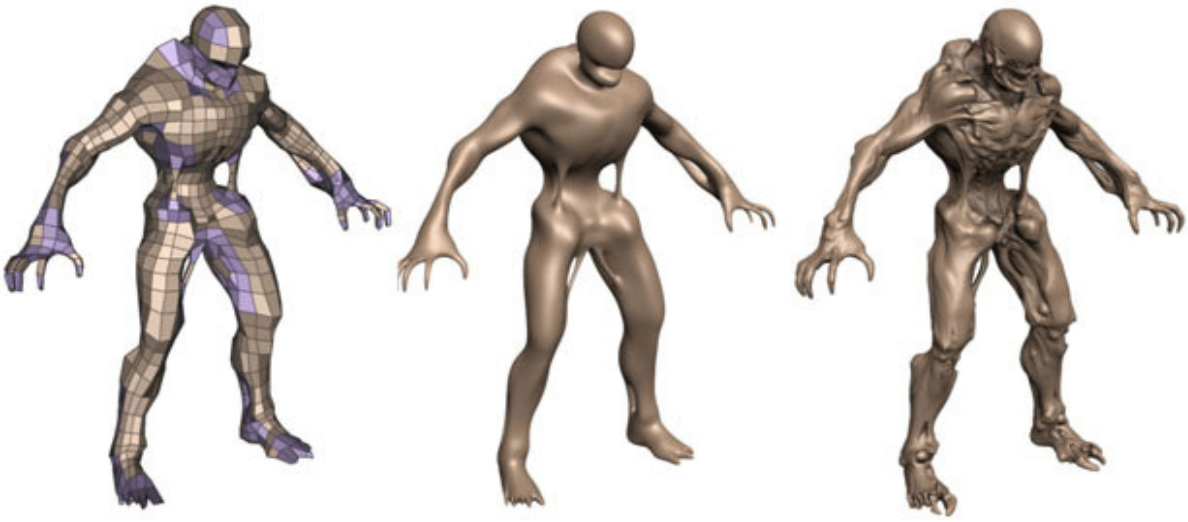

Figura 1.1: Depois de um modelo grosseiro (à esquerda) passar através do processo de tessellation, um modelo liso é produzido (meio). Ao aplicar o displacement mapping (direita), a personagem torna-se bem mais realista. [NVIDIA]

Sempre que se deseja alterar uma geometria dinamicamente utilizando *displacement mapping* no *shader* é necessário elaborar um algoritmo e escrever um novo código para alterar o mapa de deslocamento. Isso implica em alterar as chamadas aos *shaders*, seja incluindo um novo *shader* com o código criado, ou adaptando o *shader* existente incluindo esse código. Uma das principais contribuições desse trabalho é a modularização das funções responsáveis por influenciar os mapas de deslocamento. Isso permite que a criação, alteração ou adaptação feita anteriormente no *shader* torne-se mais simples de ser feita, controlada, validada e mantida, pois tudo é feito nos *kernels* do OpenCL, dentro do padrão de projeto proposto pela arquitetura.

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, tornando-a mais abrangente e flexível; a separação de cada funcionalidade em núcleos específicos da camada *Displacement Generator*, possibilitando que vários tipos de deformação sejam utilizados e combinados pelos desenvolvedores na busca de resultados interessantes para seus projetos; a possibilidade de expandir a arquitetura através da criação de *kernels*

personalizados, garantindo a flexibilidade da arquitetura em comportar novas contribuições que podem torná-la mais abrangente, além de garantir a liberdade criativa dos desenvolvedores. Esta proposta garante um controle sobre a aplicação mais simples de ser entendida, devido à abstração, permitindo também uma fácil extensão, graças a uma arquitetura que prevê novas funcionalidades.

O trabalho está dividido da seguinte forma: o capítulo 2 apresenta diversos trabalhos relacionados à proposta deste trabalho. O capítulo 3 descreve o processo de *tessellation*. O capítulo 4 aborda a utilização do *displacement mapping*. O capítulo 5 apresenta a arquitetura extensível para mapas dinâmicos de deslocamento na GPU para jogos digitais, seus módulos especializados e a possibilidade de extensão, enquanto no capítulo 6 são apresentados os resultados da implementação. Por fim, o capítulo 7 apresenta as conclusões do trabalho.

Capítulo 2

Trabalhos Relacionados

Displacement mapping foi introduzido inicialmente por Cook [COOK 1984] e vem sendo usado tradicionalmente em métodos baseados em *software*, usando *ray tracing* ou micro-polígonos. Pharr e Hanrahan [PHARR and HANRAHAN 1996] usaram geometrias em *caching* para acelerar a técnica. Wang [WANG et al. 2003] desenvolveu a *view-dependent displacement mapping*, que permite aplicar o *displacement mapping* baseado na direção da visualização da câmera. Diferentemente do método tradicional, esse método permite uma renderização de sombras e silhuetas sem adicionar complexidade na superfície base dos objetos.

Schein [SCHEIN et al. 2005] desenvolveu a chamada DDM, que é um método de deformação de geometrias em tempo real, tanto para superfícies racionais paramétricas quanto para superfícies poligonais usando o *hardware* gráfico.

Takahashi [TAKAHASHI and MIYATA 2005] descreve um modelo de deformação de superfícies base de objetos, por meio de *displacement mapping*, usando *vertex textures*. Também, no mesmo artigo, o autor descreve um método de cálculo de colisão que pode ser feito diretamente na GPU.

Tortelli [TORTELLI and WALTER 2007] destaca a evolução do *hardware* gráfico e aborda o uso da técnica de *displacement mapping* como uma das antigas técnicas custosas computacionalmente, podendo agora ser implementada em tempo real para melhorar a qualidade visual dos objetos que compõem as cenas dos jogos eletrônicos.

Tatarchuk [TATARCHUK et al. 2009] destaca que a *tessellation* em *hardware* fornece vários benefícios importantes que são cruciais para sistemas interativos como jogos digitais, tais como: compressão de malha, menor requerimento da largura de banda e escalabilidade dos modelos geométricos.

Nas versões mais recentes das APIs gráficas (DirectX11 [DIRECTX] e OpenGL4), foi adicionado o recurso do *Tessellator*, que permite a criação de vértices em massa na GPU. Nunes [NUNES 2011] em sua dissertação estuda este novo estágio da pipeline, bem como apresenta algoritmos clássicos (PN-*Triangles e Phong Tessellation*) que originalmente foram feitos para CPU e propõe novos algoritmos (Renderização de Tubos e Terrenos em GPU) para tirar proveito deste novo paradigma.

Este trabalho estende parte da proposta apresentada por Batista [BATISTA 2011], que visa simular e animar expressões faciais em faces humanas usando os algoritmos de *bump mapping* e *morphing* implementados em GPU. A implementação com *bump mapping* é visualmente interessante e tem aplicação em vários momentos de um jogo digital, mas por não alterar a geometria, traz uma série de limitações quando a deformação supera um limite, especialmente quando pode gerar auto-oclusão da superfície. A implementação com *displacement mapping*, juntamente com o processo de *tessellation*, descartando ou não o uso de *bump mapping*, pode cobrir uma ampla gama de situações, onde cada método pode ser utilizado de acordo com as necessidades das aplicações.

No próximo capítulo será apresentada a técnica de *tessellation*, seu uso e como a mesma pode colaborar no desenvolvimento de aplicações em tempo real, tais como a arquitetura aqui proposta.

Capítulo 3

Tessellation

Tessellation pode ser definida como uma divisão de um espaço em regiões poligonais convexas, sendo divisões do plano (R2) mais frequentemente discutidas. Enquanto no passado os autores reservavam o termo *tessellation* apenas para divisões do plano em polígonos regulares de mesmo tamanho, hoje o conceito é tratado de forma muito mais ampla, de modo a incluir quaisquer arranjos de formas não sobrepostas Rd, como também partições de outros espaços métricos [SCHOENBERG 2001].

O conceito da técnica de *tessellation* consiste basicamente em dividir um polígono em partes menores. Em sua forma mais básica, a *tessellation* é um método que subdivide polígonos em outros mais refinados. Por exemplo, cortar um quadrado através de sua diagonal, pode "tesselar" esse quadrado em dois triângulos. Sozinha a *tessellation* não melhora tanto o realismo, pois não importa se um quadrado é renderizado como dois triângulos ou dois mil triângulos, a *tessellation* só melhora o realismo se os novos triângulos forem usados para representar novas informações topológicas. A figura 3.1 mostra a evolução de um modelo simples em um mais detalhado após o uso da técnica de tessellation.

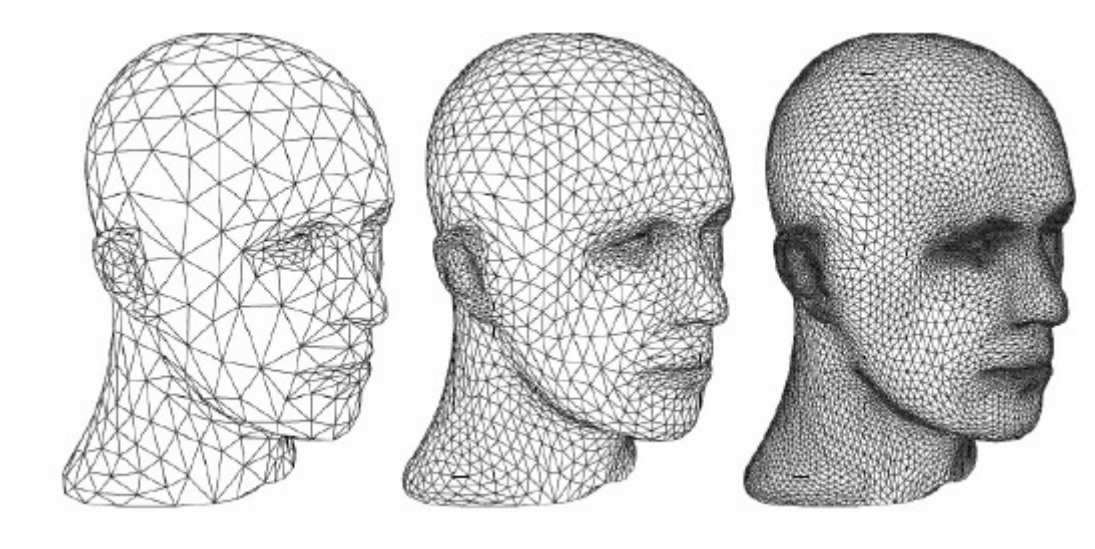

Figura 3.1: Modelo original (esquerda) sofrendo influência da tessellation em dois níveis diferentes, resultando em modelos mais detalhados (meio e direita). [DIGITAL DAILY]

Novas versões das APIs gráficas, especificamente o OpenGL 4 e DirectX 11, incluem três novos estágios no pipeline gráfico, tornando possível o uso da *tessellation* em tempo real em aplicações onde o desempenho é importante, como os jogos digitais. As figuras 3.2 e 3.3 representam respectivamente as arquiteturas dos pipelines gráficos das APIs DirectX e OpenGL. Elas destacam os novos estágios responsáveis pela *tessellation*. O DirectX 11 recebeu a inclusão de três novos estágios no seu pipeline gráfico, destacados na cor verde na figura 3.2. São eles: *Hull Shader*, *Tessellator* e *Domain Shader*. O OpenGL também compartilha dessa evolução, mas seus estágios são nomeados: *Tessellation Control Shader*, *Tessellation Primitive Generator* e *Tessellation Evaluation Shader*, os quais também estão em destaque na cor verde na figura 3.3.

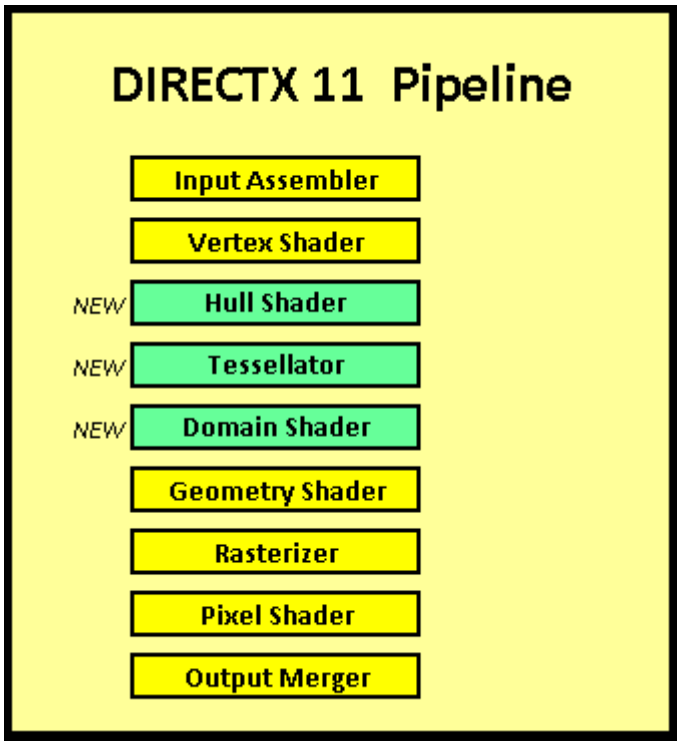

Figura 3.2: Pipeline gráfico do Direct3D 11.

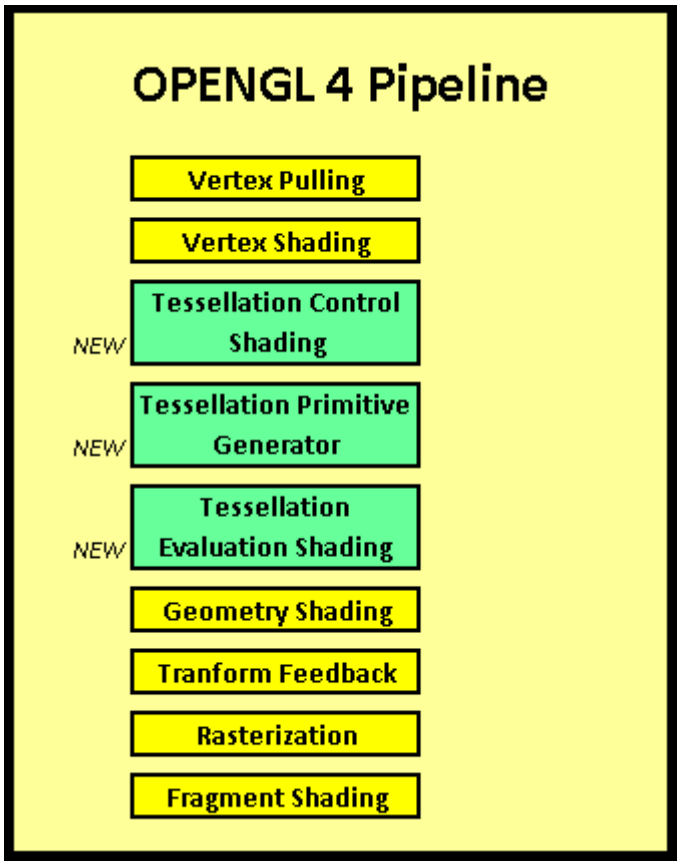

Figura 3.3: Pipeline gráfico do OpenGL 4.

O estágio denominado de *hull shader* (DirectX) ou *tessellation control shader* (OpenGL) é chamado para cada *patch*, utilizando os pontos de controle enviados pelo *vertex shader* como entradas. O *hull shader* tem duas principais funções. A primeira é (opcionalmente) converter os pontos de controle a partir de uma representação (base) para outra; por exemplo, implementando a técnica introduzida por Loop e Schaefer [LOOP and SCHAEFER 2008]. Os pontos de controle são enviados diretamente para o *domain shader* (DirectX) ou *tessellation evaluation shader* (OpenGL), ignorando o *tessellator* (DirectX) ou *tessellation primitive generator* (OpenGL). A segunda responsabilidade do *hull shader* é calcular fatores de *tessellation* adequados, que são passados para a fase de *tessellation*. Isto permite a *tessellation* adaptativa, que pode ser usada para controlar o LOD (*level of detail*) da malha. Os fatores de *tessellation* determinam o quanto uma aresta pode ser subdividida; eles são especificados por aresta e podem variar de 2 a 64.

O *tessellator* é um estágio fixo, mas configurável, que utiliza o fator de *tessellation* para subdividir o *patch* em vários triângulos. O *tessellator* não tem acesso aos pontos de controle, pois todas as decisões de *tessellation* são feitas com base na configuração e nos fatores de *tessellation* transferidos do *hull shader*. Cada vértice resultante do estágio de *tessellation* é enviado para o *domain shader*.

O *domain shader* opera na parametrização de coordenadas de cada vértice separadamente, embora ele também possa acessar os pontos de controle para transformar o domínio inteiro. O *domain shader* envia os dados completos do vértice (posição, coordenadas de textura, etc) para o *geometry shader*. Efetivamente, ele avalia a representação da superfície em cada vértice. Embora a técnica de *displacement mapping* também possa ser aplicada neste estágio, neste trabalho optamos por utilizá-la no *geometry shader*, prezando pela inteligibilidade.

Com o avanço das APIs gráficas e GPUs, a *tessellation* e o *displacement mapping* podem ser implementados em conjunto na GPU de forma satisfatória. Jogos populares, como Alien vs. Predator™ [SEGA] e Metro 2033™ [THQ], mostrados nas

figuras 3.4 e 3.5, usam a *tessellation* para criar modelos com uma aparência mais suave, e desenvolvedores da Valve [VALVE] e da id Software [ID SOFTWARE] têm feito trabalhos promissores com a utilização dessas técnicas em suas personagens.

Figura 3.4: Alien VS Predator™. [EVGA]

Figura 3.5: Metro 2033™. [EVGA]

Entendendo que o pipeline de *tessellation* é programável, embora o próprio *tessellator* seja apenas configurável, o mesmo pode ser usado para solucionar um grande número de problemas gráficos, como os descritos nas subseções abaixo.

3.1 Suavização

Os algoritmos de suavização são parceiros naturais da *tessellation*. O algoritmo de suavização utiliza um modelo grosseiro, e com o auxílio da tessellation, cria um modelo com aparência mais suave. Um exemplo popular são os Triângulos de Ponto Normal (*PN-Triangles*, também conhecidos como *N-patches*). O algoritmo de Triângulos PN converte modelos de baixa resolução em superfícies curvadas que são, em seguida, redesenhadas como uma malha de triângulos devidamente "tesselada". Muitas das imperfeições visuais que vemos nos jogos atuais, como articulações de personagens em formato cúbico, rodas de carro com aparência poligonal e recursos faciais grosseiros, podem ser eliminadas com a ajuda de tais algoritmos. Os Triângulos PN, por exemplo, são usados no jogo Stalker: Call of Pripyat [STALKER] para produzir personagens com aparência mais suave e realista. Os Triângulos PN permitem a suavização automática de personagens sem a manipulação do artista; tanto o realismo geométrico quanto os relacionados à iluminação são aperfeiçoados, conforme apresentado na figura 3.6.

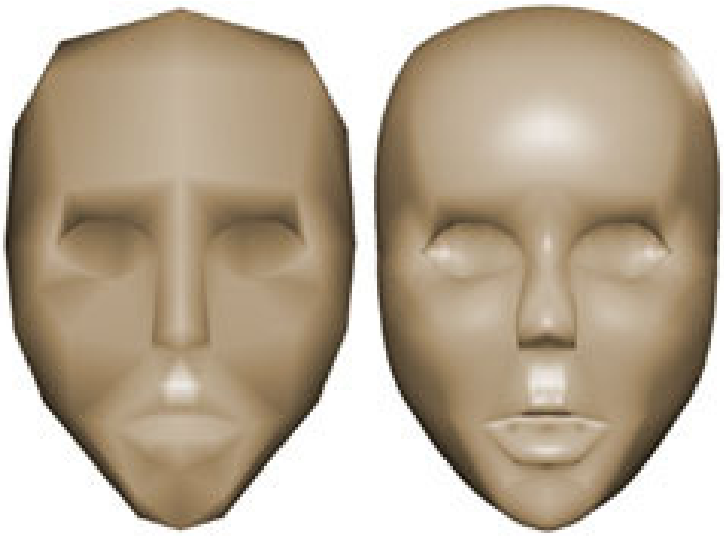

Figura 3.6: Suavização usando PN-Triangles. [NVIDIA]

3.2 Detalhamento

Em jogos com ambientes abertos e amplos, geralmente nota-se que objetos distantes normalmente mudam de aspecto de um momento para outro. Isso se deve ao mecanismo que altera diferentes LODs, para manter a carga de trabalho geométrica com menor custo. Sem o recurso de *tessellation* em *hardware*, não existia uma maneira fácil de variar continuamente o nível de detalhe, visto que isso exigiria a permanência de várias versões do mesmo modelo ou ambiente. A *tessellation* dinâmica soluciona esse problema ao variar o nível de detalhe em tempo de execução. Por exemplo, quando uma construção distante é avistada, ela pode ser renderizada com apenas três triângulos. À medida que o observador se aproxima, suas características notáveis surgem e triângulos adicionais são usados para destacar os detalhes, como janelas e telhado. Quando finalmente se alcança a porta, milhares de triângulos podem ser utilizados para renderizar somente a maçaneta, onde cada marca é esculpida meticulosamente com o *displacement mapping*. Com a *tessellation* dinâmica, o aparecimento e consequente desaparecimento de objetos não ocorrem mais, e os ambientes dos jogos podem dimensionar os detalhes geométricos de modo quase ilimitado. A figura 3.7 mostra uma malha usando tessellation com a técnica de LOD aplicada, onde a parte do terreno mais distante da câmera possui menos detalhes, enquanto a parte mais próxima da câmera possui maior quantidade de subdivisões.

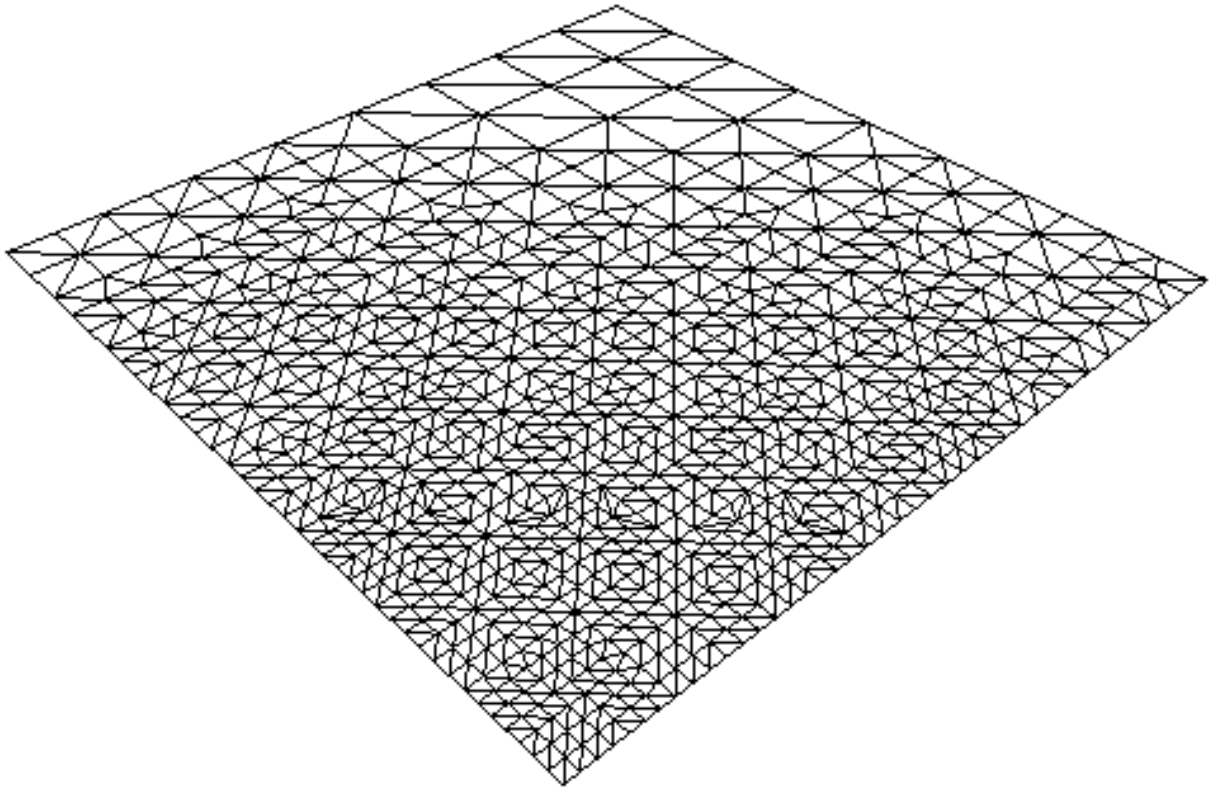

Figura 3.7: Level of details. [GAMEDEV]

3.3 Escalabilidade

Para os desenvolvedores, a *tessellation* aumenta consideravelmente a eficiência do pipeline de criação de conteúdos. Ao descrever sua motivação para usar a *tessellation*, Mitchell, J. afirmou: "Estamos interessados na capacidade de gerar ativos, que nos permitam tanto aumentar como reduzir a escala. Isto é, queremos criar um modelo uma vez só e conseguir aumentar a sua escala a uma qualidade semelhante àquela de filmes. De modo oposto, também queremos conseguir diminuir naturalmente a escala da qualidade de um ativo para atender às necessidades da renderização em tempo real em um determinado sistema" [NI et al. 2009]. A capacidade de criar um único modelo e usá-lo em várias plataformas, como destacado na figura 3.8, significa menos tempo gasto com desenvolvimento. É a obtenção da melhor qualidade de imagem possível das GPUs.

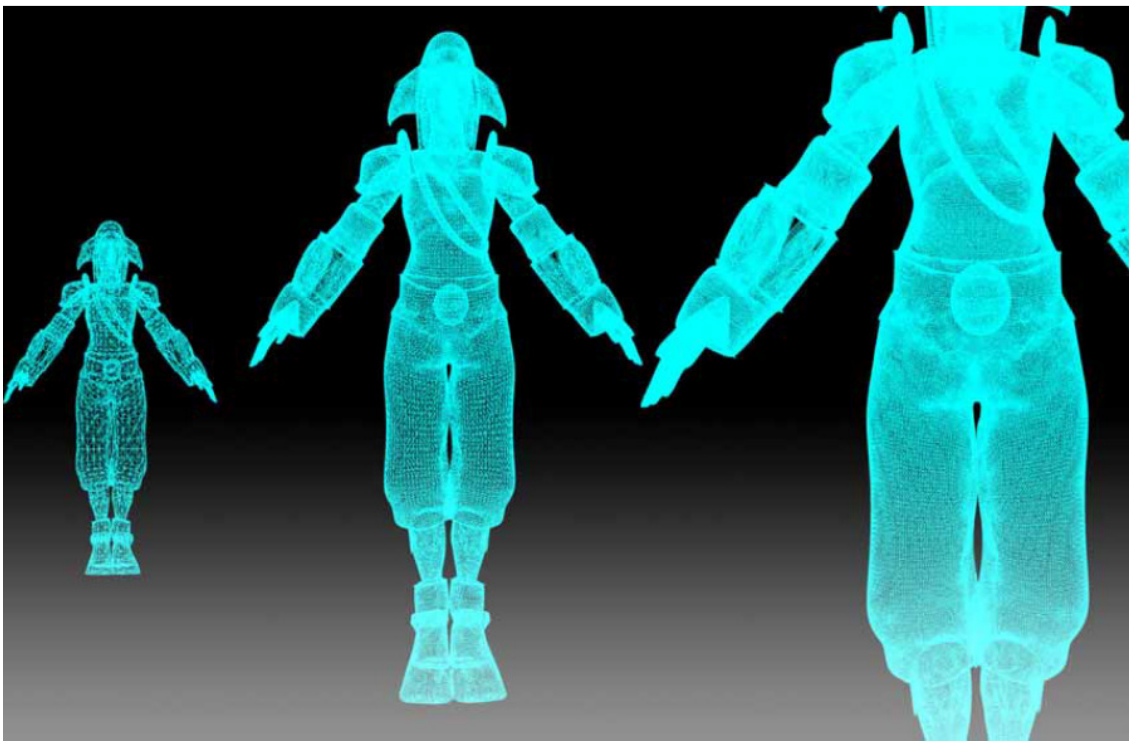

Figura 3.8: Escalabilidade. [TATARINOV 2008]

Após muitos anos de tentativas e erros, a *tessellation* finalmente se tornou realidade. Jogos impressionantes, como o Metro 2033™ [THQ], já demonstram o potencial da *tessellation*. Com o tempo, a *tessellation* poderá se tornar tão importante e indispensável quanto o sombreamento de pixels. Conscientes dessa possível importância, as fabricantes de GPUs estão criando arquiteturas para suportar a *tessellation* paralela, resultando em verdadeiros avanços no realismo geométrico e no desempenho da *tessellation*. Conforme demonstrado por Nunes [NUNES 2011], o *tessellator* ganha no desempenho quando comparado ao mesmo processo feito em CPU. A arquitetura extensível proposta neste trabalho utiliza a *tessellation* implementada em *hardware* na composição da sua camada *Displacement Generator*, visando manter taxas interativas para suportar o uso dinâmico do *displacement mapping*.

No próximo capítulo será apresentada a técnica de *displacement mapping*, seu uso e como a mesma pode colaborar no desenvolvimento de aplicações em tempo real, tais como a arquitetura proposta neste trabalho.

Capítulo 4

Displacement Mapping

O *displacement mapping* [COOK 1984] é uma técnica alternativa quando comparada com técnicas como texture mapping [CATMULL 1974] e bump mapping [BLINN 1978]. Essa técnica usa uma textura como um mapa de deslocamento que, aplicada sobre uma geometria base, causa um efeito onde a posição atual dos vértices dessa geometria é deslocada em alguma direção (geralmente a direção da normal da geometria), seguindo parâmetros fornecidos pela própria textura. Esse efeito proporciona à superfície uma grande sensação de profundidade, aumentando o detalhamento e a complexidade da geometria, permitindo sombras e silhuetas mais definidas e detalhadas.

Segundo Schein [SCHEIN et al. 2005], várias técnicas de computação gráfica foram desenvolvidos para adicionar detalhes em superfícies de geometrias, sendo o *texture mapping* e o *bump mapping* as mais comuns. Quando uma delas é usada para dar a impressão de uma superfície áspera, por exemplo, a técnica de *bump mapping* é mais eficaz. Entretanto, não ocorrem alterações na geometria do objeto, o que é facilmente notado quando se aplicam efeitos de sombra sobre essa geometria. Ou seja, o efeito é apenas visual, não alterando a geometria do objeto. Segundo Cook [COOK 1984], a técnica de *displacement mapping* não é considerada uma técnica de textura, e sim, um tipo de modelagem de geometria.

Por vários anos, a técnica de *displacement mapping* ficou restrita apenas aos sistemas de renderização de alto desempenho, como o PhotoRealistic RenderMan da Pixar [PIXAR], sendo inviável sua implementação em sistemas de renderização em tempo real. Um dos motivos dessa restrição foi que a proposta inicial da implementação da técnica necessitava de um sistema de criação de primitivas adaptativo da superfície da geometria para obter novos polígonos. Esse sistema de criação de primitivas era muito custoso em termos de processamento, principalmente em aplicações em tempo real. Felizmente, com a evolução dos dispositivos de *hardware* gráfico, APIs como OpenGL e Direct3D podem agora prover o uso dessa técnica em tempo real, criando assim novas possibilidades da sua utilização, principalmente em aplicações como jogos digitais.

Embora o *displacement mapping* já exista há bastante tempo, sem os recursos de *hardware* atuais seu uso era impraticável em aplicações de *rendering* em tempo real. O motivo é que para o *displacement mapping* ser eficaz, a superfície deve ser constituída de um grande número de vértices, motivo pelo qual essa técnica é classificada como custosa. Um relevo detalhado só pode ser formado se existirem vértices suficientes na malha base para esculpir a nova forma. Em essência, o *displacement mapping* precisa da *tessellation* e somente com os novos recursos de *hardware* isso agora é viável.

Em seu uso mais básico, o *displacement mapping* pode ser usado como uma substituição direta para técnicas existentes de mapeamento de saliências. Técnicas atuais, como o *normal mapping*, criam a ilusão de superfícies salientes através de alteração do sombreamento de pixels. Todas essas técnicas funcionam apenas em casos específicos, e são apenas parcialmente convincentes. Por exemplo, o uso do *parallax occlusion mapping* (POM), que essencialmente consiste em realizar um mapeamento de saliências, produz a ilusão de sobreposição da geometria, porém funciona apenas em superfícies lisas e somente no interior do objeto, de forma autocontida, sem influenciar as bordas. O *displacement mapping* não enfrenta esses problemas e produz resultados precisos a partir de todos os ângulos de visualização. A

figura 4.1 mostra um comparativo entre as técnicas de *bump mapping*, POM e *displacement mapping*, deixando evidente a diferença entre seus contornos e as limitações referentes ao ângulo de visão escolhido.

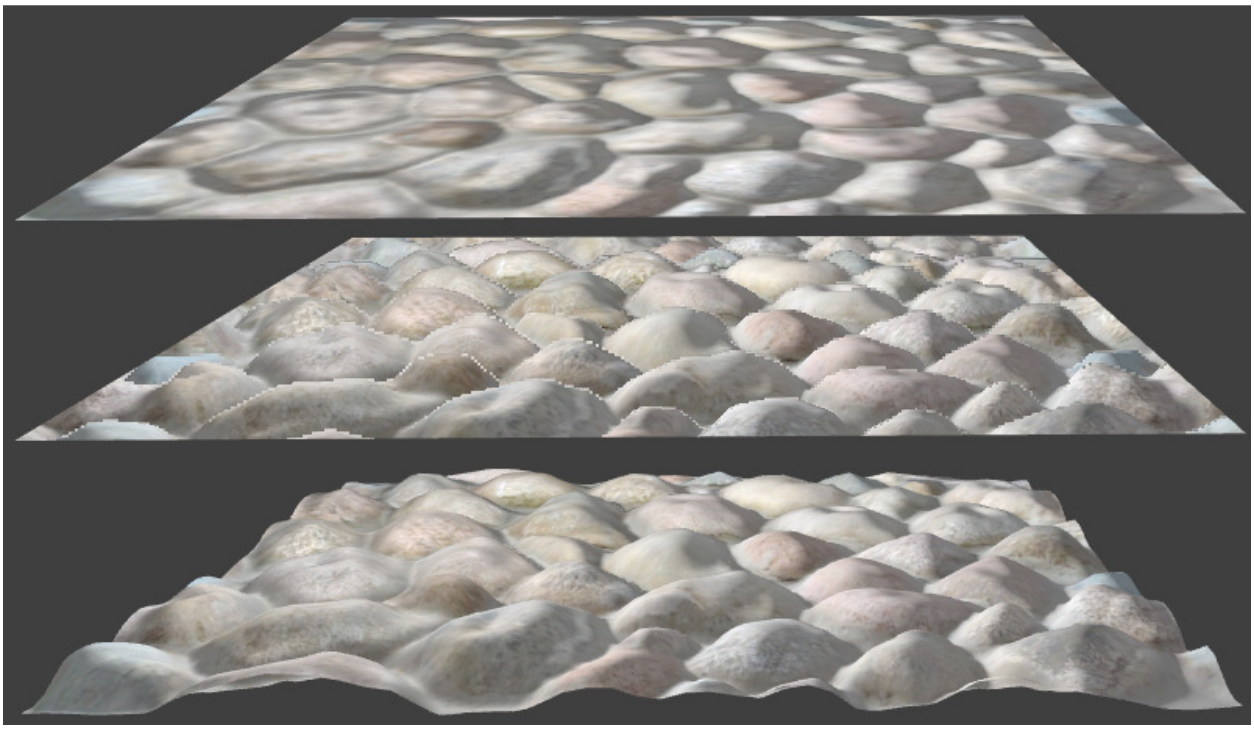

Figura 4.1: Comparação entre técnicas de bump mapping (acima), POM (meio) e displacement mapping (baixo).

Embora a *tessellation* seja indispensável para que a arquitetura proposta neste trabalho funcione, é o *displacement mapping* a técnica realmente explorada pela arquitetura. Tanto os *kernels* específicos quanto os personalizados têm como objetivo gerenciar os mapas de deslocamento, mapas esses que são usados dinamicamente pela técnica de *displacement mapping* durante a execução da aplicação. Os mapas de deslocamento são influenciados nos *kernels* específicos para representar, nas malhas criadas com *tessellation*, deformações por contato, deformações por transição de malhas e deformações por forças, enquanto os *kernels* personalizados podem representar novos casos de acordo com a necessidade e criatividade de seus desenvolvedores. O próximo capítulo apresentará a arquitetura e esclarecerá todas as funcionalidades referentes aos *kernels*.

Capítulo 5

Arquitetura Extensível

Segundo Pressman [PRESSMAN 2002], a arquitetura é a representação que permite ao engenheiro de software analisar a efetividade do projeto, satisfazer seus requisitos declarados, considerar alternativas arquiteturais num estágio em que fazer modificações de projeto é ainda relativamente fácil e reduzir os riscos associados com a construção do *software*. Baseado em Gamma [GAMMA et al. 1995], os padrões de projeto tornam fácil a reutilização de código e arquiteturas. Eles devem prover alternativas de projeto que tornem os sistemas reutilizáveis. Eles também aumentam a documentação e a manutenção de sistemas já existentes, por fornecerem uma especificação explícita de classes e interações de objetos e suas intenções subjacentes.

O foco deste trabalho é propor uma arquitetura extensível que visa retirar do *shader* a responsabilidade de controle sobre os mapas de deslocamento. A arquitetura restringe essa responsabilidade, colocando-a sobre os *kernels*. O objetivo é reduzir a complexidade e modularizar suas funções em um modelo de controle de fácil aprendizagem, domínio, utilização, extensão, manutenção e replicação. Da perspectiva técnica, segundo Bass, Clements e Kazman [BASS et al. 2003], existem três razões para que seja reconhecida a importância da arquitetura de software, sendo elas:

- Comunicação entre os envolvidos no projeto: a arquitetura representa uma abstração comum de um sistema que muitos, se não todos, dos envolvidos no projeto podem utilizar como a base de um mútuo entendimento, negociação, consenso e comunicação;

- Decisões de início de projeto: uma arquitetura de software manifesta as decisões iniciais de um sistema, estas possuindo profundo impacto com as próximas etapas, como o restante do desenvolvimento, o *deploy* do sistema, e o ciclo de manutenção. Este também é o ponto inicial onde decisões de projeto específicas sobre o sistema a ser desenvolvido podem ser analisadas;

- Abstração transferível de um sistema: a arquitetura constitui um modelo relativamente pequeno e intelectualmente inteligível de como um sistema é estruturado e como seus elementos trabalham em conjunto, sendo este modelo transferível entre os sistemas. Em particular, ela pode ser aplicada em outros sistemas que exibam atributos de qualidade e requisitos funcionais similares, podendo promover reutilização em larga escala.

A arquitetura proposta neste trabalho se beneficia desses conceitos. A aprendizagem e o domínio estão diretamente relacionados à comunicação entre os envolvidos no projeto, conforme sua descrição. Da mesma forma, as decisões de início de projeto englobam a utilização e a manutenção da arquitetura. A abstração transferível de um sistema está diretamente ligada à replicação do modelo, mas também pode considerar sua ampliação ou extensão.

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, sendo responsáveis apenas por acessar esse resultado e aplicá-lo conforme estabelecido. 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.

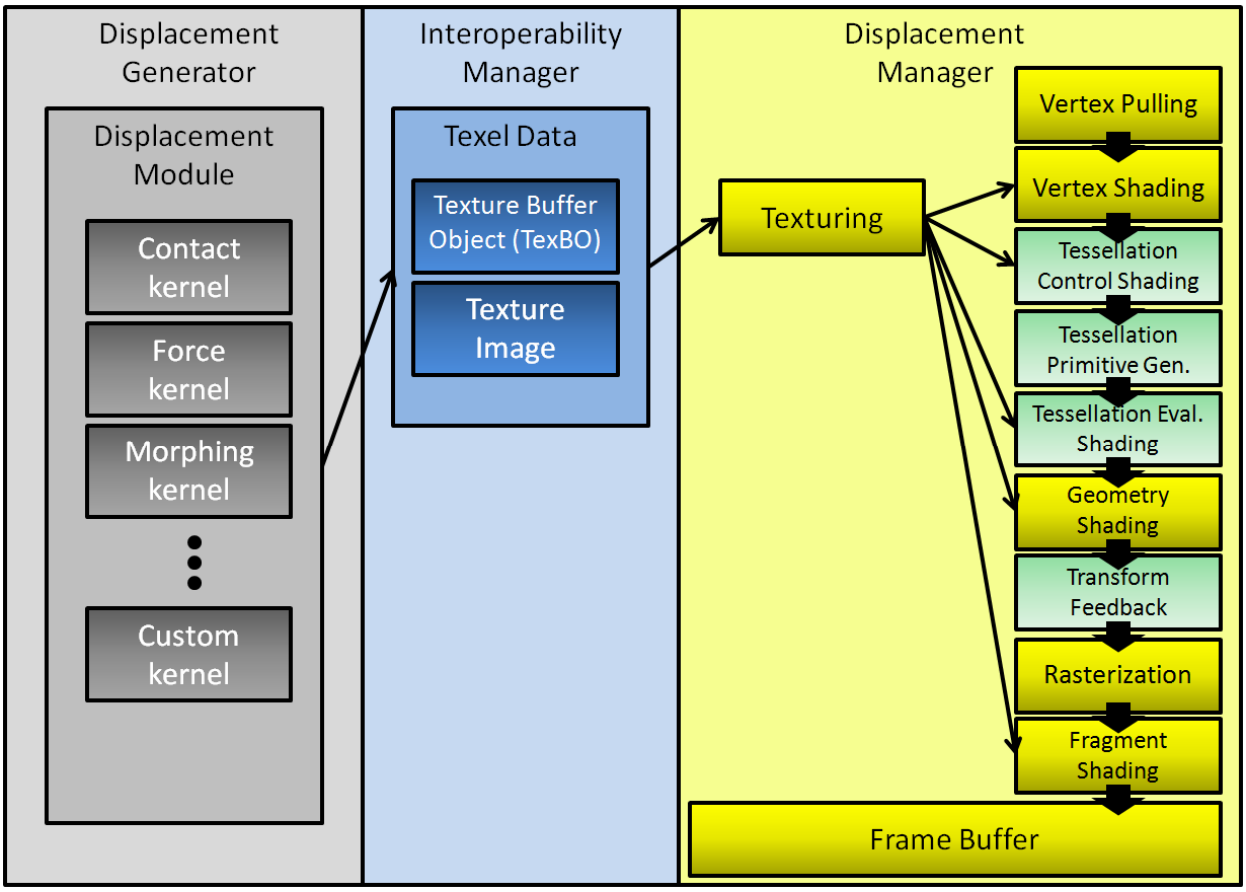

Figura 5.1: Arquitetura Extensível: modelo em camadas.

A figura 5.1 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*. Segundo Buschmann [BUSCHMANN et al. 1996], uma abordagem em camadas é considerada de melhor prática do que a implementação do protocolo como um bloco monolítico, porque a implementação separada de problemas que possuem conceitos

distintos resulta em diversos benefícios, como, por exemplo, possibilidade de desenvolvimento em equipes. A utilização de partes semi-independentes também fornece a possibilidade de se trocar com mais facilidade essas partes, posteriormente, caso necessário. Melhores tecnologias de implementação, novas linguagens ou algoritmos podem ser acrescentados, simplesmente reescrevendo uma seção de código delimitada. É exatamente isso que a arquitetura proposta permite que seja feito, pois permite trocar o *shader* usado na camada *Displacement Manager* ou escrever novos *kernels* na camada *Displacement Generator*, trabalhando de forma independente em cada camada. Como o recurso de *tessellation* é recente até o momento de desenvolvimento deste trabalho, espera-se que ocorram melhorias e, portanto, muitas alterações de código.

Na camada *Displacement Generator* definiu-se o módulo *Displacement Module*, o qual é composto inicialmente por três modelos de *kernels* distintos e específicos, sendo eles: *contact*, *force* e *morphing kernel*. Estes *kernels* representam diferentes conjuntos de tarefas que se podem realizar na aplicação e que de alguma maneira são implementações diferentes de *displacement mapping*. A arquitetura é citada como extensível e na figura 5.1 acima, destaca-se o *custom kernel*, o qual representa essa possibilidade de extensão. O *custom kernel* pode tanto ser utilizado para reescrever variações específicas dos *kernels* já propostos, sem perdê-los, quanto escrever *kernels* completamente novos para atender as necessidades específicas dos desenvolvedores no que se refere à deformação de superfícies por *displacement mapping*. O *custom kernel* permite que a arquitetura seja extensível. A camada *Interoperability Manager* garante à arquitetura a fluidez necessária para que sua utilização seja admissível, uma vez que não necessita que a informação transite entre CPU e GPU, permitindo acesso direto à textura tanto pela camada *Displacement Generator* quanto pela camada *Displacement Manager*. A camada *Displacement Manager* utiliza as informações contidas na textura para influenciar a nova geometria criada com a etapa de *tessellation*, usando a técnica de *displacement mapping* no *geometry shader*.

5.1 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. Em seu uso natural, 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 5.2. Essa entidade pode ser um jogador, um NPC, uma tropa, veículos, animais ou qualquer entidade criada pelos desenvolvedores.

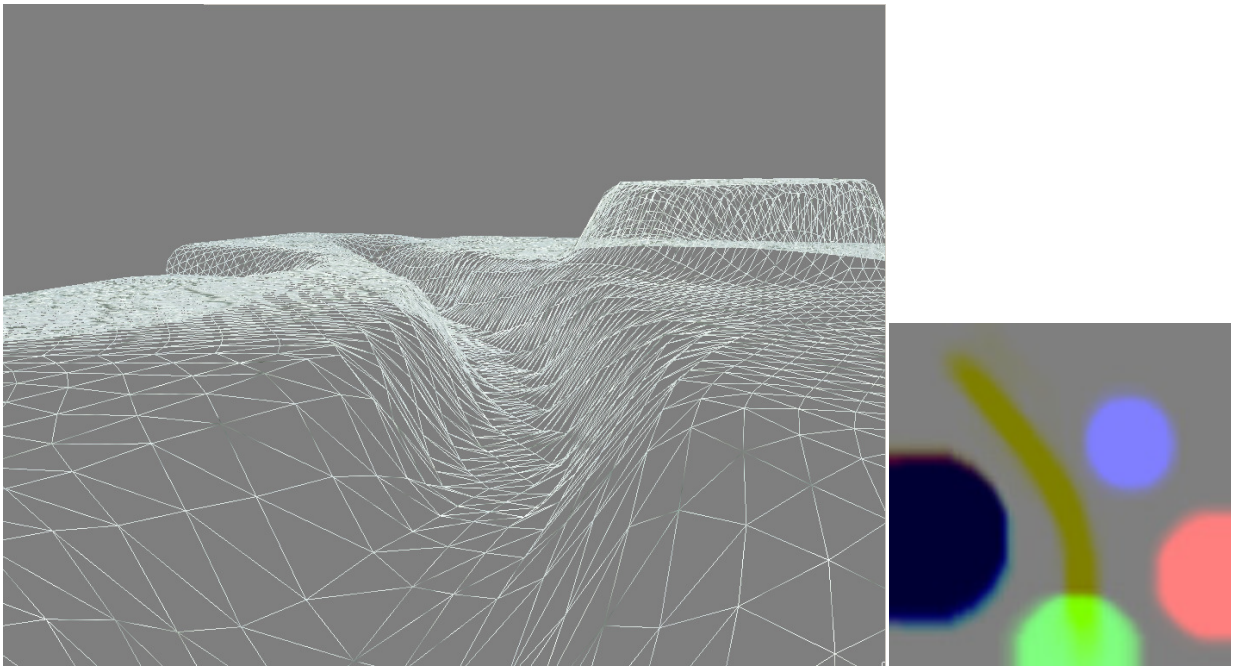

Figura 5.2: Resultado da geometria (esquerda), perspectiva, e mapa de deslocamento (direita), vista superior, após o uso do contact kernel.

O pseudocódigo abaixo representa a função usada no módulo de contato da arquitetura proposta, representado pelo *contact kernel* na figura 5.2. Esse *kernel*,

como observado na figura 5.2 (direita), vista superior, 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 5.2 (esquerda), perspectiva, mostra na geometria um caminho que corresponde ao rastro criado pelo uso do *kernel*. Foi usada uma escala maior de maneira a destacar visualmente o resultado, além do posicionamento específico da câmera.

```
kernel void contact(read image in, write image out, float2 pos,
float depth, float radius)
{
      sampler s = normal_mode|address_mode|filter_mode;
      int x = get_global_id(0);
      int y = get_global_id(1);
      int2 coords = (int2)(x,y);
      float4 image = read_image(in,s,coords);
      float dist = distance(pos,float2(x,y));
      if(dist < radius)
            image.z -= depth;
      write_image(out,coords,image);
}
```

Este *kernel*, pode também possuir outra funcionalidade: a de armazenar informações de deslocamento das entidades em um mapa de deslocamento que poderá servir como um mapa de localização espacial, como na figura 5.3, sendo possível usá-lo como fonte de consulta para informações necessárias para utilização em algoritmos de busca. Nesse caso, esse mapa não seria utilizado para influenciar a geometria da malha. Uma entidade do jogo provida de comportamento específico, por exemplo, pode usar as informações contidas no mapa para perseguir outras entidades. Uma entidade alvo transmitiria para o mapa de deslocamento algumas informações que seriam armazenadas pelo mesmo, como o rastro azulado na figura 5.3. Ao invés de processar um algoritmo de busca, o que poderia se tornar custoso dependendo do tamanho do mapa, entre outros fatores, a entidade perseguidora leria o mapa de

localização espacial referente à área atual e dependendo de suas habilidades, perseguiria o alvo com base nas informações encontradas.

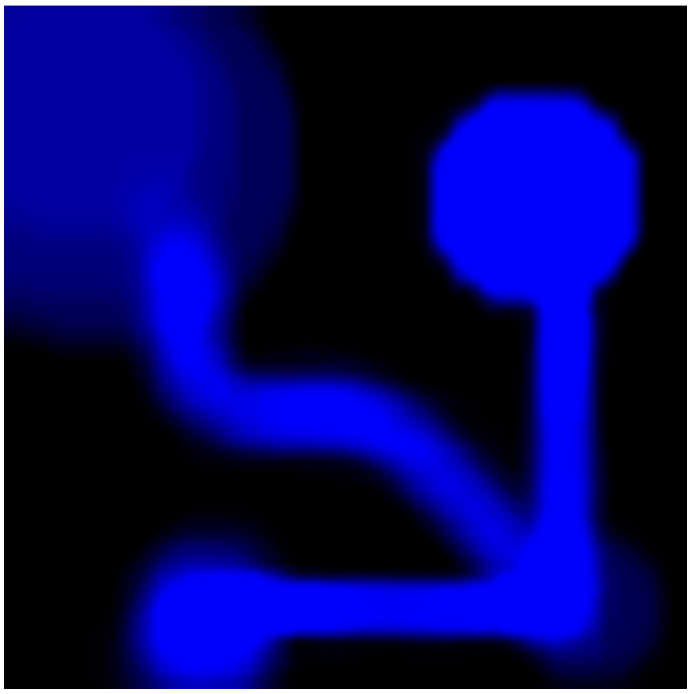

Figura 5.3: Mapa de deslocamento usado como mapa de localização espacial.

Uma terceira utilização poderia ser composta pelas duas funcionalidades juntas, usando um mapa de deslocamento de forma híbrida, que significa usar o mesmo mapa para armazenar informações que serão usadas para localização e para alterar a topografia do ambiente. Com essas duas últimas utilizações (localização e híbrida), novas possibilidades podem surgir, e cabe ao desenvolvedor adaptá-las e usá-las nos seus jogos. O uso dessas informações para avaliação pela inteligência artificial nos jogos digitais torna-se um campo passível de investigação.

Foram apresentadas três possibilidades de uso do mapa de deslocamento pelo *contact kernel*, sendo a última de caráter híbrido. O mapa de deslocamento é alterado e passa a ter outra função, além de representar detalhes topológicos, que consiste em guardar informações de trajetórias. Estas informações podem ser usadas pelos desenvolvedores para diversas atividades, tais como: *pathfinding*, redes de comunicação entre NPCs, etc. Dessa forma, um jogador que, por exemplo, marcar em

seu mapa uma exaustiva utilização de determinada localização, poderá ser surpreendido por voltar nesse local e encontrá-lo completamente alterado. Isso poderia ter ocorrido por um evento gerado aleatoriamente para destruir o que parecia ser mais importante para o jogador em determinado ponto do jogo ou mesmo para bonificá-lo através de um processo de *design* interativo.

5.2 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. A força é uma grandeza vetorial e possui características específicas, como: módulo, direção e sentido. 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 citadas nas características, enquanto a quarta componente pode ser utilizada para influenciar os resultados, multiplicando a grandeza. Essa influência pode ser muito útil, dada a variedade de representações que se pode ter, pois as forças podem ser resultado de explosões, choques com forte impacto, propagação de ondas sob a superfície, entre outros exemplos. 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 5.4. Essa deformação pode ser uma cratera, morro, fissura ou qualquer outra estrutura gerada de acordo com os parâmetros de força escolhidos pelos desenvolvedores.

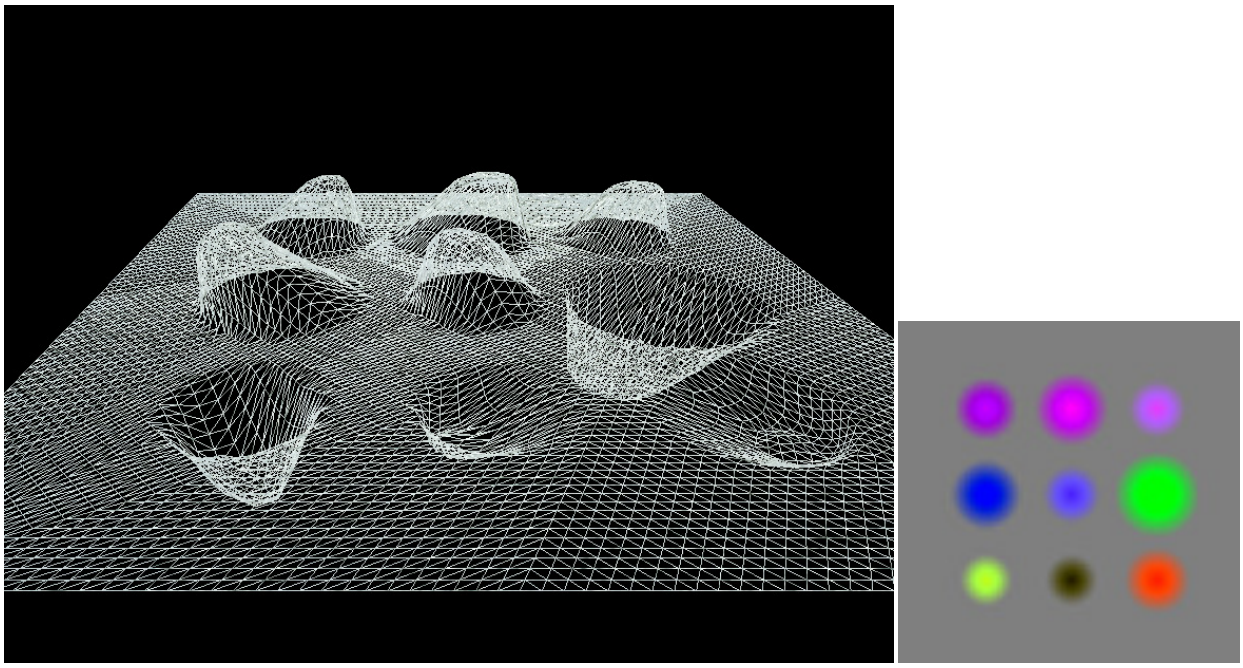

Figura 5.4: Resultado da geometria (esquerda) e mapa de deslocamento (direita) após o uso do force kernel.

O *force kernel* foi pensado principalmente para representar as consequências geradas pelas ações dos jogadores nos ambientes virtuais. Devido à grande variedade de jogos existentes atualmente, a possibilidade de interagir e modificar os cenários virtuais tornou-se um diferencial muito atrativo. Representações de deformação geradas por colisão entre entidades, geralmente, são perdidas quando uma área é deixada, fazendo com que uma nova visitação à área citada não mostre qualquer sinal de atividade anterior. O *force kernel* permite gerar as deformações para essas áreas, podendo as alterações ser armazenadas no mapa de deslocamento do local, o qual pode ser acessado posteriormente para remontar a área contendo esse histórico, visando dar mais realismo ao ambiente do jogo. O *force kernel* gera deformações por aplicação de forças, podendo simular o impacto da explosão de uma granada, por exemplo, deixando uma área completamente deformada; ou a colisão de uma entidade específica do jogo que tenha ou não o objetivo de causar dano ao ambiente, como: acidente de veículos, projéteis, entre outras variações.

```
kernel void force(read image in, write image out, float2 pos,
float4 force)
{
      sampler s    = normal_mode|address_mode|filter_mode;
      int x        = get_global_id(0);
      int y        = get_global_id(1);
      int2 coords  = (int2)(x,y);
      float4 image = read_image(in,s,coords);
      float area   = length(force.xyz) * CONSTANT;
      float dist   = distance(pos,float2(x,y));
      float dissipation = 1 - dist / area;
      if(dissipation > 0)
            image += force * dissipation;
      write_image(out,coords,image);
}
```

O pseudocódigo acima mostra o algoritmo usado para a criação da imagem apresentada na figura 5.4. 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, visando manter proporções consistentes, multiplicado por uma constante, a qual serve para adequar a escala da deformação ao ambiente que será afetado. Na imagem mostrada, o uso da constante garantiu que as estruturas geradas não se sobrepusessem, visando melhor exemplificação. 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.

Outra forma de representar esse tipo de deformação é a utilização de uma estrutura pronta, a qual é copiada de um mapa de deslocamento predefinido e aplicada no local da colisão. Geralmente essas estruturas predefinidas são denominadas de decalque ou adesivos, devido ao tipo de utilização. Um *custom*

kernel pode ser utilizado para personalizar essa funcionalidade, preferencialmente alterando a aplicação da estrutura em escala e rotação, visando evitar um padrão repetitivo, tratando com mais naturalidade a deformação. Exemplos desse tipo de técnica podem ser encontrados nas APIs gráficas, geralmente incluídos nos pacotes distribuídos para os desenvolvedores.

5.3 Módulo de Morphing

O módulo de morphing é um *kernel* que representa casos de deformação por transição de malhas; sendo essa transição feita com algoritmo de *morphing* linear. Segundo Conci, Azevedo e Leta [CONCI et al. 2008], *morphing* é uma redução da palavra metamorfose. O termo *morph* tem etimologia grega, *morphos*, que significa forma. Em computação gráfica, significa um processo de transição de um objeto para outro, de forma gradual. O *morphing* também é conhecido como uma técnica de processamento de imagens que objetiva transformar de forma gradual uma imagem em outra. Existem muitas aplicações tecnológicas e científicas para essas técnicas, tais como: assistência às cirurgias plástica e de reconstrução, análise da progressão no tempo de fotografias de pessoas desaparecidas e estudo da evolução das formas de organismos vivos, para análise de crescimento e desenvolvimento. Porém, ainda é na produção de efeitos especiais no cinema e na televisão que a aplicação dessas técnicas é mais popular.

O *morphing* realiza uma alteração de imagens, considerando o processo de deformação aliado à decomposição de suas cores. O *morphing* bidimensional provoca a sensação de uma mudança de forma através dos efeitos de distorção envolvidos; além disso, há uma variação de textura que aprimora esse efeito. Wolberg [WOLBERG 1998] menciona que a metamorfose entre duas imagens começa com um animador que estabelece a correspondência entre os pares de características primitivas, tais como: nós da malha, segmentos de linha, curvas ou pontos.

No *morphing* de malhas, um modelo tridimensional gerado com o formato do objeto tem sua forma transformada em um segundo objeto. Em cada estágio do *morphing*, o modelo criado é "renderizado" e mapeado para produzir uma representação na tela. O processo é complexo, pois devem ser definidas correspondências entre as imagens inicial e final através de suas formas. Essa é uma das razões dos objetos precisarem ser semelhantes, ao menos topologicamente, para

serem metamorfoseados. Quando a topologia dos objetos é similar, o mapeamento é feito ponto a ponto; quando ela difere, torna-se mais complicado estabelecer as correspondências entre os pontos.

Como citado, 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 void morphing(read image in1, read image in2, write image out,
float time)
{
      sampler s = normal_mode|address_mode|filter_mode;
      int x = get_global_id(0);
      int y = get_global_id(1);
      int2 coords = (int2)(x,y);
      float4 image1 = read_image(in1,s,coords);
      float4 image2 = read_image(in2,s,coords);
      float4 image = image1 * (1-time) + image2 * time;
      write_image(out,coords,image);
}
```

O uso prático do módulo de *morphing*, denominado *morphing kernel* na figura 5.1, consiste na utilização de pelo menos dois mapas de deslocamento que representam o estado inicial e final da topografia, como mostrado nas figuras 5.5 e 5.6. 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 nas figuras 5.5 e 5.6. Esse processo também pode ser aplicado à simulação de erosão, onde um monte pode se

tornar uma planície, um lago pode surgir em meio a um bosque ou mesmo uma ravina pode ser criada na lateral de uma floresta, durante os eventos do jogo.

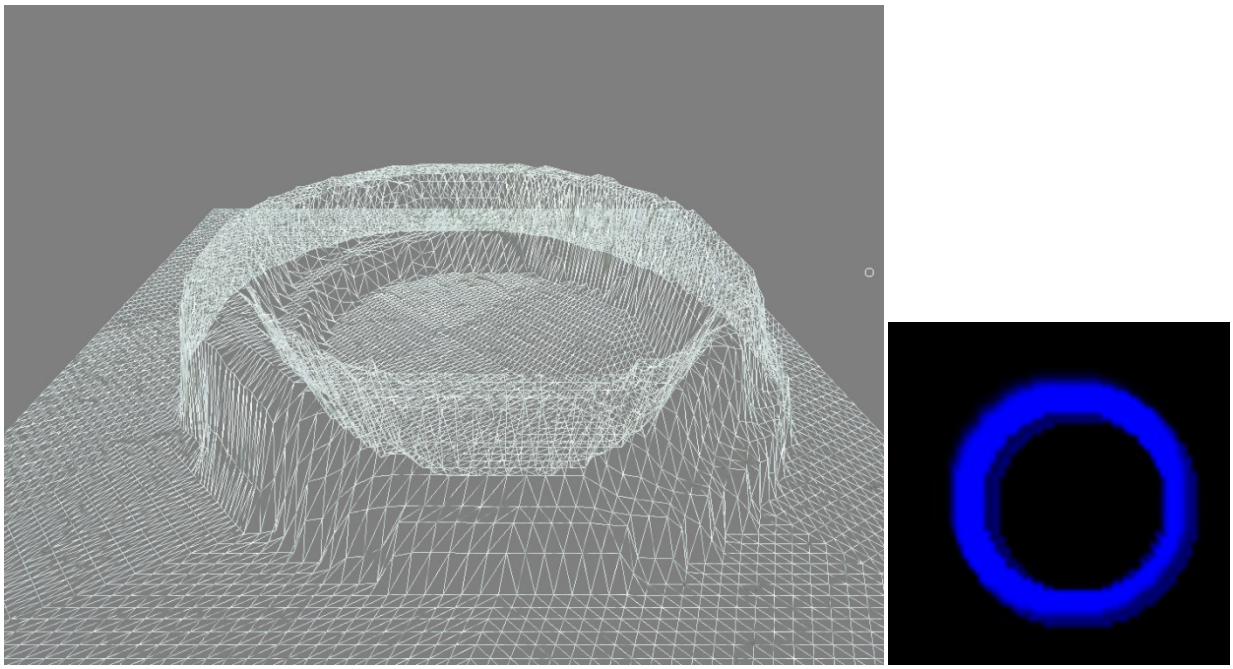

Figura 5.5: Estado inicial da geometria (esquerda) e mapa de deslocamento inicial (direita) que será usado no morphing kernel.

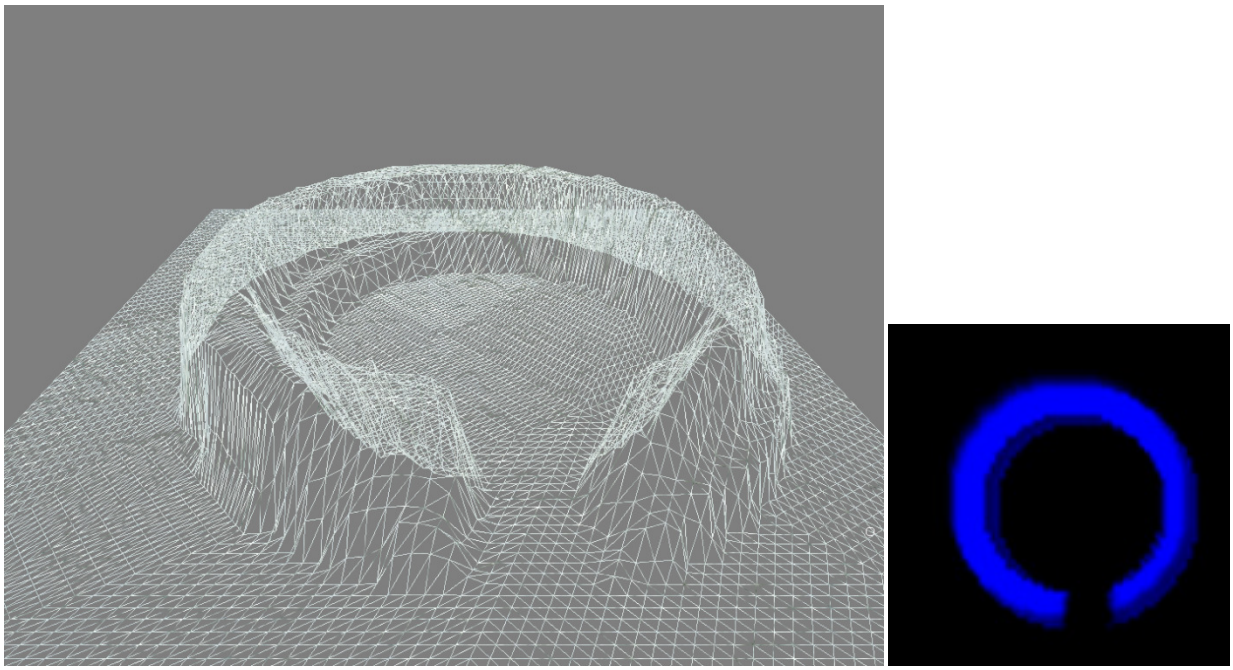

Figura 5.6: Resultado da geometria (esquerda) e mapa de deslocamento final (direita) após o uso do morphing kernel.

5.4 Módulo Personalizado

O módulo personalizado, denominado de *custom kernel* na arquitetura apresentada neste trabalho, 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 void custom(read image in, write image out, float4 arg1,
float4 arg2, float4 arg3, float4 arg4)
{
      sampler s = normal_mode|address_mode|filter_mode;
      int x = get_global_id(0);
      int y = get_global_id(1);
      int2 coords = (int2)(x,y);
      float4 image = read_image(in,s,coords);

      // INSERT CODE HERE!

      write_image(out,coords,image);
}
```

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 neste trabalho, garantindo que qualquer um deles possa ser reescrito

com o módulo personalizado. Essa construção foi escolhida como padrão predefinido do *custom kernel* visando servir para uso imediato ou como modelo para a criação de novos *kernels*.

A classe *DisplacementStage* da implementação, que será apresentada no próximo capítulo, está preparada para compilar o arquivo de OpenCL contendo todos os *kernels*. Ela também possui as funções que suportam o *custom kernel* na criação de *buffers*, preparação e passagem de parâmetros da CPU para a GPU, como mostrado na figura 5.7. Essa tarefa é simples, bastando utilizar o modelo pronto, o qual também serve de exemplo para novas implementações.

```
class DisplacementStage
{
public:
    DisplacementStage(cl::string fileName);
    ~DisplacementStage(void);

    void setKernel(cl::string kernelName);

    ...

    // CUSTOM KERNEL
    void createMemoryBufferToCustomKernel(void);
    void setCustomArg(void);
    void copyCustomToMemory(float*, float*, float*, float*);
    Buffer getBufferCustom1(void);
    Buffer getBufferCustom2(void);
    Buffer getBufferCustom3(void);
    Buffer getBufferCustom4(void);
    void releaseCustomBuffer(void);

    ....

    void runKernel(void);

private:
    ...
};
```

Figura 5.7: Funções preparadas para suportar o uso do Módulo Personalizado.

Finalizando essa etapa de preparação, basta utilizar o novo *kernel*, invocando-o em alguma parte do jogo através do uso das funções disponibilizadas pela classe *DisplacementStage*, responsável pela camada *Displacement Generator*.

Como citado nos módulos anteriores, o módulo personalizado poderá abranger as funcionalidades referentes ao mapa híbrido e de localização espacial citados na seção 5.1, poderá propor o uso de geometrias predefinidas como as citadas por decalques e adesivos na seção 5.2, poderá utilizar o morphing direcionado para sequências animadas, entre uma diversidade de soluções. No próximo capítulo o módulo personalizado será usado para representar uma ferramenta de modelagem, onde uma interface simples (o próprio mapa de deslocamento) será posicionada na tela visando sofrer interação direta do usuário, utilizando mouse e teclado, enquanto a geometria é deformada em tempo real, comprovando a utilização da arquitetura apresentada neste trabalho e confirmando a personalização do *custom kernel* como um módulo viável de uso.

Capítulo 6

Implementação

O fluxograma mostrado na figura 6.1 apresenta as atividades principais que compõem as etapas para utilização da arquitetura proposta neste trabalho. As etapas estão dividas em três partes, sendo elas: configuração, destacada na cor azul; atualização, destacada em amarelo; e desenho, destacada em verde.

Conforme descrito em [ANDRADE et al. 2012], acompanhando o fluxograma podemos entender que o contexto OpenCL é criado, logo depois os recursos de malha e textura são carregados pelo sistema nas atividades iniciais descritas no mesmo. As próximas atividades fazem com que o recurso textura seja registrado pelo OpenGL e em seguida pelo OpenCL, permitindo a interoperabilidade, que estará a cargo da camada responsável pela comunicação entre as outras duas camadas, como apresentado no capítulo anterior. Esse conjunto de atividades compõe a primeira etapa configuração, a qual faz a configuração do sistema e ocorre uma única vez por aplicação.

Na etapa de atualização, o mapa de deslocamento é tratado pelo *kernel* escolhido, o qual faz as alterações de acordo com sua funcionalidade, na única atividade dessa etapa registrada no fluxograma. Essa etapa representa a camada *Displacement Generator* descrita pela arquitetura extensível. Como pode ser

observado, o *kernel* no fluxograma é representado por um ícone de processo alternativo, significando que ele não é obrigatório, sendo o *kernel* invocado durante a atualização somente quando é necessário.

Na etapa de desenho, que é de responsabilidade da camada *Displacement Manager*, a primeira atividade é o envio do recurso malha para o shader (GLSL), a segunda atividade é o uso desse recurso pelo *Tessellator*, estágio do *shader* responsável por usar a técnica de *tessellation* para enriquecer a malha, aumentando seu número de vértices. A terceira atividade acontece no *geometry shader*, estágio do GLSL responsável por usar a técnica de *displacement mapping*, a qual utiliza informações do mapa de deslocamento resultante do *kernel* para influenciar a malha trabalhada com *tessellation*. A última atividade da etapa de desenho finaliza o processo "renderizando" o resultado criado pela arquitetura.

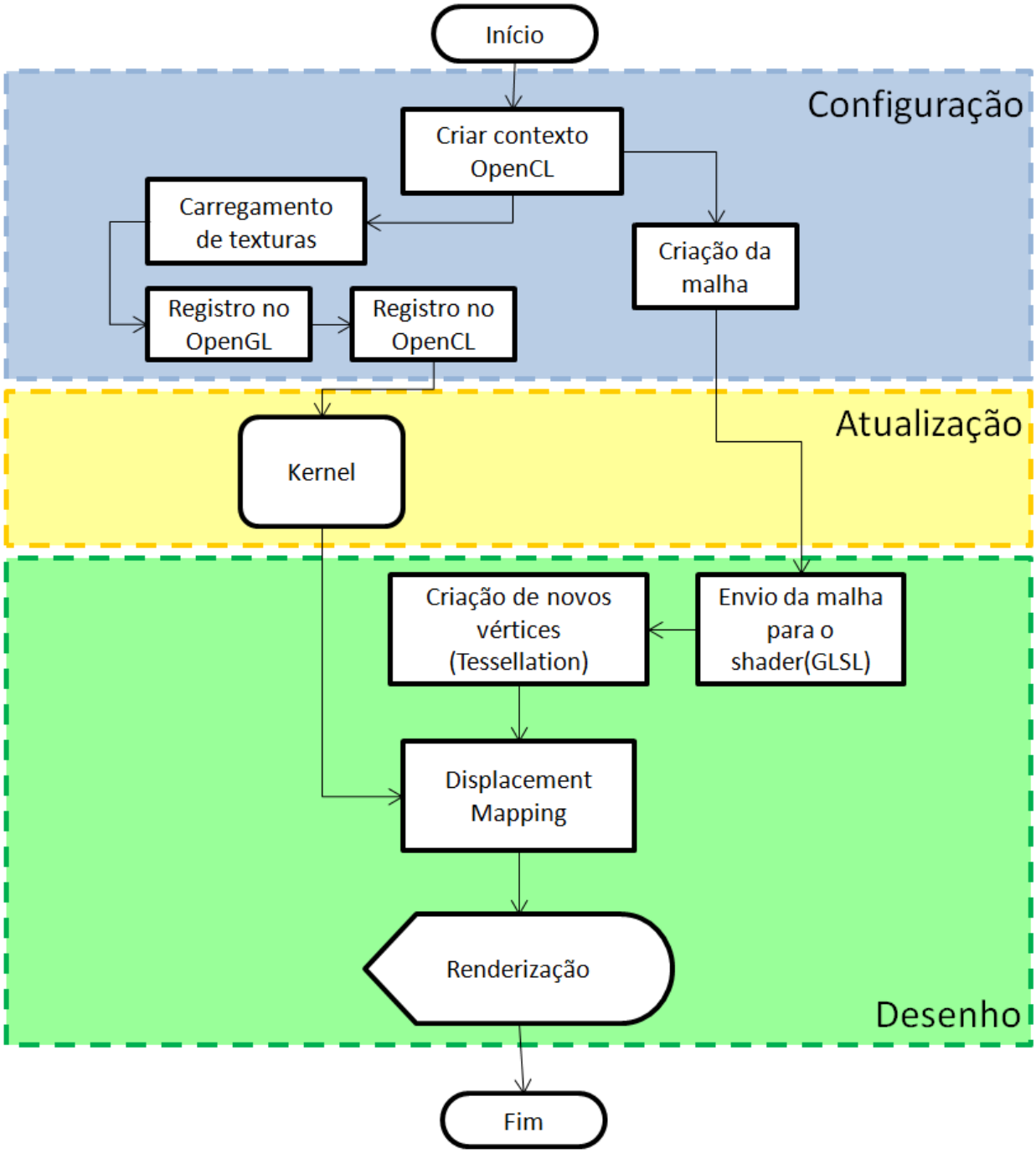

Figura 6.1: Fluxograma das principais atividades que compõem as etapas da implementação da arquitetura.

Com o objetivo de transformar as atividades do fluxograma apresentado em um exemplo prático e com intuito de validar a funcionalidade da arquitetura utilizando os *kernels* como caso de testes, foi necessário definir uma base de ferramentas e linguagens neste trabalho. A linguagem C++ foi escolhida devido ao seu caráter

multiplataforma, o qual condiz com as características da API gráfica OpenGL e da linguagem da arquitetura OpenCL, as quais possuem compatibilidade de uso com a linguagem escolhida. O Visual Studio 2010 [VS 2010] foi o IDE escolhido por ser um ambiente de fácil manipulação, com um rico sistema de suporte ao desenvolvimento de *softwares*, o qual ajuda a reduzir o tempo gasto com a escrita de códigos e visualização de resultados.

O Openframeworks [OF] é um kit de ferramentas C++ (*open source*) projetado para ajudar no processo criativo, fornecendo um *framework* simples e intuitivo para a experimentação. O conjunto foi projetado para funcionar de forma abrangente e inclui o uso de muitas bibliotecas de conhecimento comum. Ele também é compatível com cinco sistemas operacionais (Windows, OSX, Linux, iOS, Android) e possui compatibilidade com quatro IDEs, sendo o Visual Studio uma delas, encaixando-se perfeitamente com a linguagem, IDE e API escolhidos. Excluindo a parte arquitetural do modelo proposto, pode ser observado em [ANDRADE and CLUA 2011], o uso do DirectX11 ao invés do OpenGL; foi feita a mudança de plataforma visando alcançar uma maior quantidade de desenvolvedores e ferramentas de mercado.

Na introdução do trabalho, foi citado que a utilização de padrões de projetos é cada vez mais explorada, principalmente pela necessidade de um controle mais apurado sobre os processos. Os padrões de projeto (*design patterns*) originaram-se na área de construção civil, onde Christopher Alexander [ALEXANDER et al. 1977] afirmou que cada padrão descreve um problema do ambiente e o cerne da sua solução, de tal forma que se possa usar tal solução muitas vezes, sem nunca fazê-lo da mesma maneira. Erich Gamma [GAMMA et al. 1995] afirma que os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Compartilhando dessas afirmações e lembrando que padrões de projeto estão intrinsecamente ligados ao paradigma de orientação a objetos, a arquitetura proposta foi elaborada desde o início respeitando tais características.

O diagrama apresentado na figura 6.2 representa o modelo em camadas referente à figura 5.1, considerando aspectos de padrão de projetos e o paradigma orientado a objetos. A figura 6.2 ilustra esta arquitetura em UML, detalhando aspectos da implementação adotada. 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.

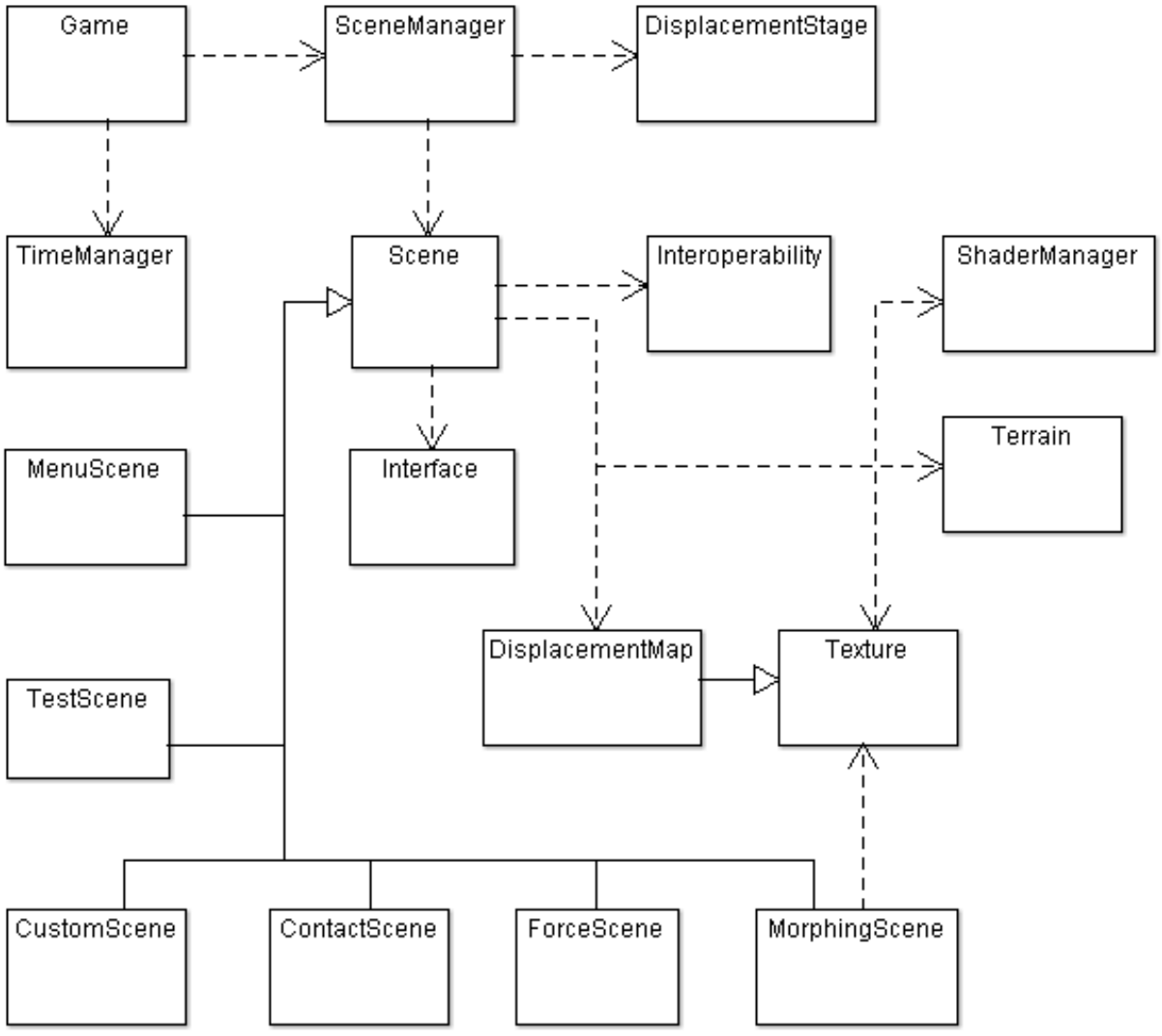

Figura 6.2: Diagrama de classes representando a implementação da arquitetura proposta.

A seguir serão descritas as respectivas classes, detalhando aspectos do funcionamento da arquitetura.

A classe *Game* é onde se encontra o laço principal da implementação deste trabalho. Ela é uma classe derivada da classe de aplicação do Openframeworks, que é um *framework* aberto e multiplataforma que utiliza a linguagem C++ e a API OpenGL. Ele ajuda disponibilizando recursos necessários para execução do projeto, evitando gasto de tempo na criação de funcionalidades não relacionadas diretamente com a atividade fim.

A classe *SceneManager* foi elaborada para atender o padrão de projeto comportamental *Mediator*, o qual segundo Gamma [GAMMA et al. 1995], tem a intenção de definir um objeto que encapsula a forma como um conjunto de objetos interage. O *Mediator* promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentemente. A classe *SceneManager* também adota o padrão de projeto de criação *Singleton*, que segundo Gamma [GAMMA et al. 1995], garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela.

A classe *TimeManager* disponibiliza o tempo entre *frames* através de métodos que retornam o valor da variável *deltaTime*, o qual é usado para manter as funcionalidades da aplicação independente da taxa de *frames* por segundo (*fps*). Essa classe também corresponde ao padrão de projeto de criação *Singleton*.

A classe *DisplacementStage* é responsável pelas atividades da camada *Displacement Generator*, as quais são responsáveis pela criação do contexto OpenCL, e principalmente, pela implementação e execução dos *kernels* apresentados no trabalho (*contact*, *force*, *morphing* e *custom kernel*).

A classe *Interoperability* é responsável pelo registro do mapa de deslocamento no OpenCL, para possibilitar a interoperabilidade com o OpenGL, onde a textura é originalmente alocada. A classe *Interoperability* originalmente era parte da classe *DisplacementStage* e foi separada com intuito de deixar mais evidente sua atividade e facilitar o reconhecimento da relação da implementação com o modelo em camadas.

A classe abstrata *Scene* gera um objeto polimórfico, o qual de acordo com o controle da classe *SceneManager*, pode assumir a função de umas das classes que herdam da classe *Scene*, sendo elas: *MenuScene*, *ContactScene*, *ForceScene*, *MorphingScene*, *CustomScene* e *TestScene*. A classe *Scene* possui um objeto da classe *Terrain*, o qual representa a malha utilizada pela implementação; ela também possui um objeto da classe *Texture*, o qual é usado para representar a textura usada para cobrir o terreno visualmente; um objeto da classe *DisplacementMap*, o qual representa o mapa de deslocamento e é utilizado pelas cenas que utilizam os *kernels*; um objeto da classe *Interoperability*, o qual faz o registro dos mapas e texturas na camada de interoperabilidade; um objeto da classe *ShaderManager*, o qual é responsável pelos *shaders* utilizados no trabalho e representa a camada *Displacement Manager*; e um objeto da classe *Interface*, a qual é explicada junto da classe *TestScene*.

A classe *DisplacementMap*, embora herde da classe *Texture*, foi criada exclusivamente para facilitar a compreensão da implementação e dar destaque ao objeto principal usado pelos *kernels*.

A classe *Texture* é importante porque carrega a imagem na memória, registra-a no OpenGL e disponibiliza uma identificação que é usada pela classe *Interoperability* para registrar o *buffer* no OpenCL e permitir que toda a manipulação da informação seja mantida no dispositivo gráfico. Essa mesma identificação possibilita que o mapa de deslocamento seja recuperado da memória e possa ser armazenado como arquivo, de forma que possa ser carregado posteriormente mantendo o histórico de deformações geradas em determinada área do ambiente virtual, conforme comentado no capítulo anterior.

A classe *Terrain* possui a geometria base, a qual recebe o tratamento das técnicas de *tessellation* e *displacement mapping* nos *shaders* GLSL para representar os resultados obtidos com o uso do mapa de deslocamento trabalhado nos *kernels*.

A classe *MenuScene* serve como tela de introdução e escolha das cenas que se desejam observar. Ela funciona como elo entre as cenas que apresentam os resultados dos *kernels* individualmente neste trabalho. Ela tem a responsabilidade de representar a organização da aplicação.

A classe *ContactScene* é responsável pelo uso do *contact kernel*. Conforme uma entidade se move em um ambiente virtual, sua posição e influência são passadas como parâmetros para o *kernel*, o qual gerencia o mapa de deslocamento usado para representar a topologia do terreno, criando trilhas sobre a superfície do ambiente. Nesta implementação foi criado um algoritmo que percorre o mapa automaticamente, gerando deformações no mesmo com intuito de validar o uso do módulo de contato.

A classe *ForceScene* é responsável pelo uso do *force kernel*. Quando uma deformação por força é necessária, são passados para o *kernel* a posição e o vetor força, os quais são utilizados para calcular a deformação na malha 3D. Essa influência pode ser agravada pelo uso da quarta componente do vetor força, a qual pode possuir um valor significativo para escalar os resultados. Nesta implementação foi criado um temporizador que aplica forças geradas aleatoriamente em nove pontos do mapa de deslocamento, gerando diversas deformações por força e validando o uso do módulo de força. O resultado da utilização dessa classe foi apresentado no capítulo anterior.

A classe *MorphingScene* é responsável pelo uso do *morphing kernel*. Essa classe possui dois objetos da classe *Texture*, os quais representam os estados inicial e final da geometria que será gerada pela interpolação controlada no *morphing kernel*. Nesta implementação foi criado um temporizador baseado em uma função de seno, de forma que a deformação por morphing resultante fique num ciclo que altera os estados inicial e final, validando o uso do módulo de *morphing*.

A classe *CustomScene* utiliza o *custom kernel*, mas como o mesmo não possui qualquer codificação que influencie o mapa de deslocamento usado, a cena serve exclusivamente como exemplo para observação de outros desenvolvedores.

A classe *TestScene* implementa o *custom kernel* e utiliza toda a estrutura de funções disponibilizadas para ele na classe *DisplacementStage*, assim como a classe *CustomScene*. A diferença entre elas é que o *custom kernel* foi editado para utilizar os dados enviados para ele, utilizando-os em tempo real para simular uma ferramenta de modelagem, usando as deformações geradas no mapa de deslocamento.

A classe *Interface* foi elaborada para servir como o padrão de projeto estrutural *Decorator*, que segundo Gamma [GAMMA et al. 1995], atribui responsabilidades adicionais a um objeto dinamicamente. Os *decorators* fornecem uma alternativa flexível a subclasses para extensão da funcionalidade. Um objeto dessa classe foi criado para prover uma interface de controle para auxiliar a classe *TestScene*. Esse objeto foi criado na classe *Scene*, mas não inicializado, deixando a cargo das subclasses que o utilizarem, fazê-lo. Esse objeto pode facilmente ser utilizado para auxiliar quaisquer das classes responsáveis pelo uso de mapa de deslocamento, pois mesmo sem se responsabilizar pelo controle dos parâmetros enviados ao *kernel*, pode disponibilizar visualmente o mapa de deslocamento utilizado na cena para observação em tempo real do funcionamento dos algoritmos usados para geração de deformação. A figura 6.3 abaixo ilustra o uso do objeto da classe *Interface* como *decorator* na cena resultante do uso da classe *ForceScene*, comprovando sua adaptação.

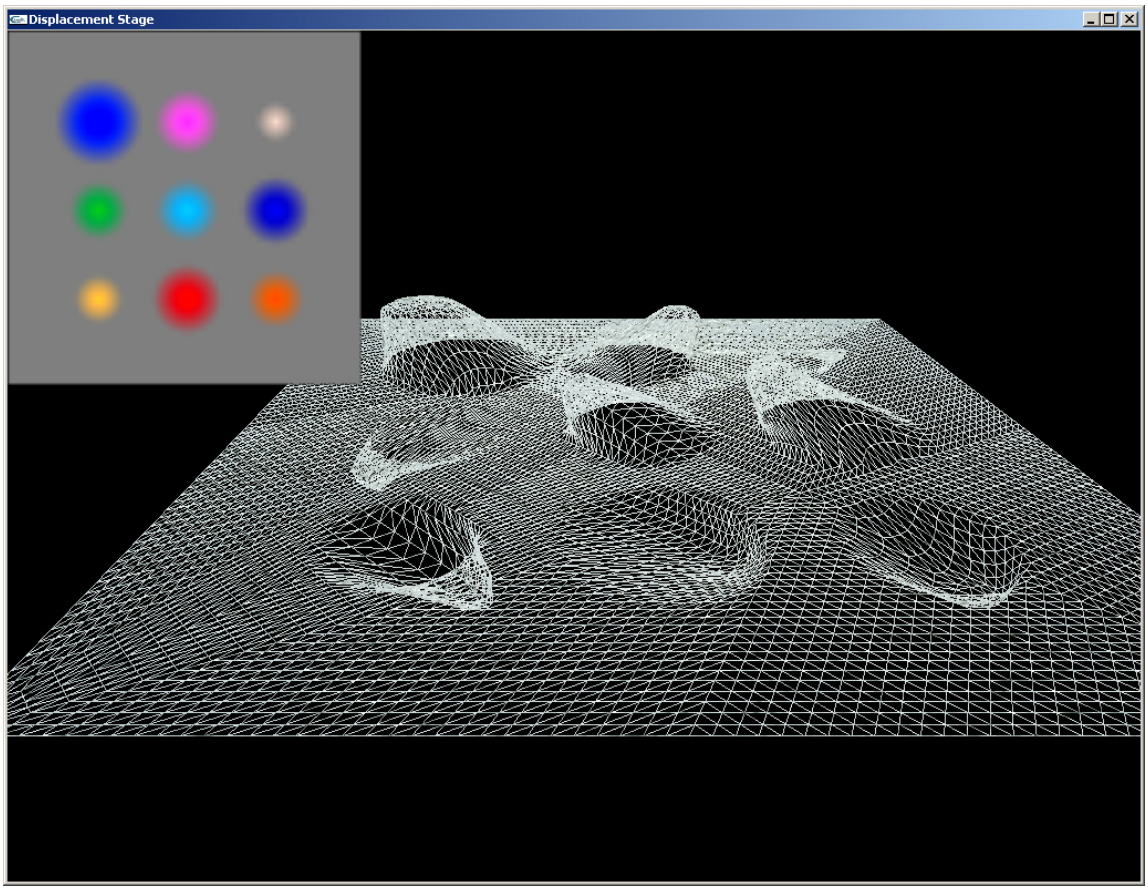

Figura 6.3: Padrão de projeto estrutural Decorator aplicado a ForceScene.

Os resultados visuais de cada um dos principais *kernels* (*contact*, *force* e *morphing kernel*) foram mostrados no capítulo anterior. Como visto, para uma aplicação que utiliza a posição de uma personagem para marcar o terreno, usamos o *contact kernel*. Em caso de multidão, é possível aumentar o raio de ação e passar valores mais expressivos, descartando a necessidade de passar cada indivíduo separado para o *kernel*, embora se possa considerar uma pequena mudança, que consistiria em reduzir a influência em direção às bordas da área. Mudar o formato da área influenciada também é uma opção viável para esse caso. Vale lembrar que qualquer alteração pode ser registrada na arquitetura extensível, simplesmente escrevendo um *kernel* personalizado, denominado *custom kernel*.

O resultado do uso do *force kernel* foi demonstrado anteriormente em imagem, a qual possui nove estruturas geradas pela chamada ao *kernel* utilizando valores gerados aleatoriamente para representar as forças passadas como parâmetros durante

os testes. O *force kernel* foi utilizado pela classe *ForceScene* descrita neste capítulo. As estruturas geradas possuem tamanho e forma diferentes, de acordo com os valores de força escolhidos, local onde são aplicadas essas forças e a utilização do valor de adequação descrito anteriormente. Esse *kernel* pode gerar deformações que representem vários tipos de estruturas, como: buracos, crateras, rachaduras, montes etc.

Utilizar o *morphing kernel*, por exemplo, faz com que dois mapas de deslocamento sejam usados com a técnica de *morphing* linear para gerar o mapa de deslocamento final, o qual será usado para influenciar a malha. No capítulo anterior foram mostrados os resultados obtidos, comprovando a utilização do kernel na classe *MorphingScene* da implementação. O pseudocódigo de exemplo mostrado no capítulo anterior servirá para ajudar na replicação do *kernel*.

Como se pode observar, as partes mutáveis que precisam de maior atenção na implementação são os *kernels*, mas com a utilização da arquitetura extensível, de acordo com a utilização proposta, basta escolher o *kernel* mais adequado para a função desejada e alimentá-lo com os parâmetros necessários. Outra opção é personalizar um *custom kernel*, e como já foram mostrados os resultados dos demais *kernels*, nesse capítulo será mostrado o resultado do caso de teste do *custom kernel*, renomeado para *test kernel*, implementado na classe *TestScene*.

Para visualmente 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 5.4, pois é uma adaptação do *custom kernel*.

```
kernel void test(read image in, write image out, float4 arg1,
 float4 arg2, float4 arg3, float4 arg4)
{
      sampler s = normal_mode|address_mode|filter_mode;
      int x = get_global_id(0);
      int y = get_global_id(1);
      int2 coords = (int2)(x,y);
      float4 image = read_image(in,s,coords);

      // INSERT CODE HERE!
      float dist = distance(arg1.xy,float2(x,y));
      if(dist < arg3.x)
            image += arg2;

      write_image(out,coords,image);
}
```

Na classe *TestScene* o *kernel* recebeu os parâmetros mostrados na linha de código abaixo. Sendo os argumentos *mouse*, *normal*, *radius* e *null* respectivamente correspondentes à arg1, arg2, arg3 e arg4, mostrados no *custom kernel* acima.

```
displacementStage->copyCustomToMemory(mouse, normal, radius, null);
```

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.

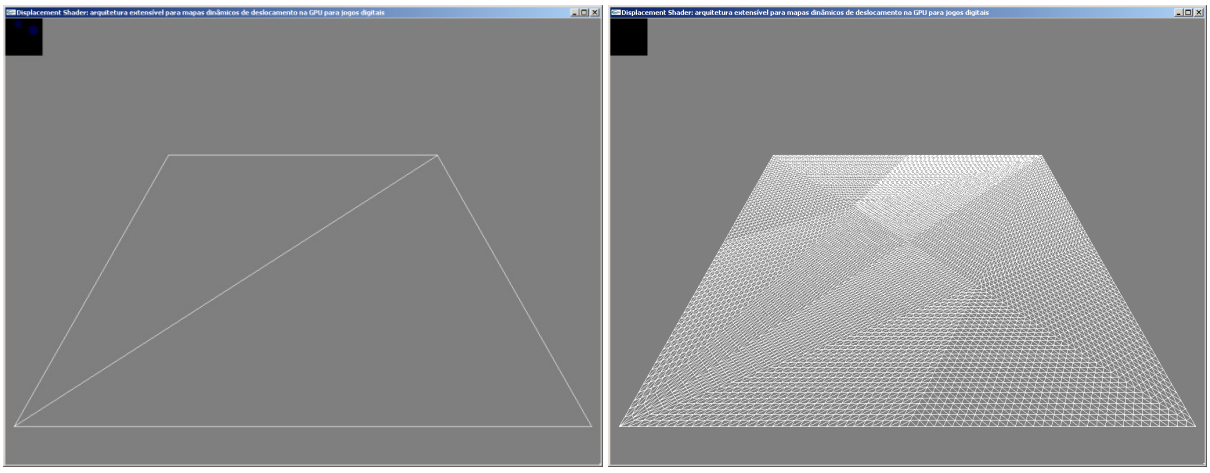

Figura 6.4: Malha original (esquerda) e malha após a tessellation (direita).

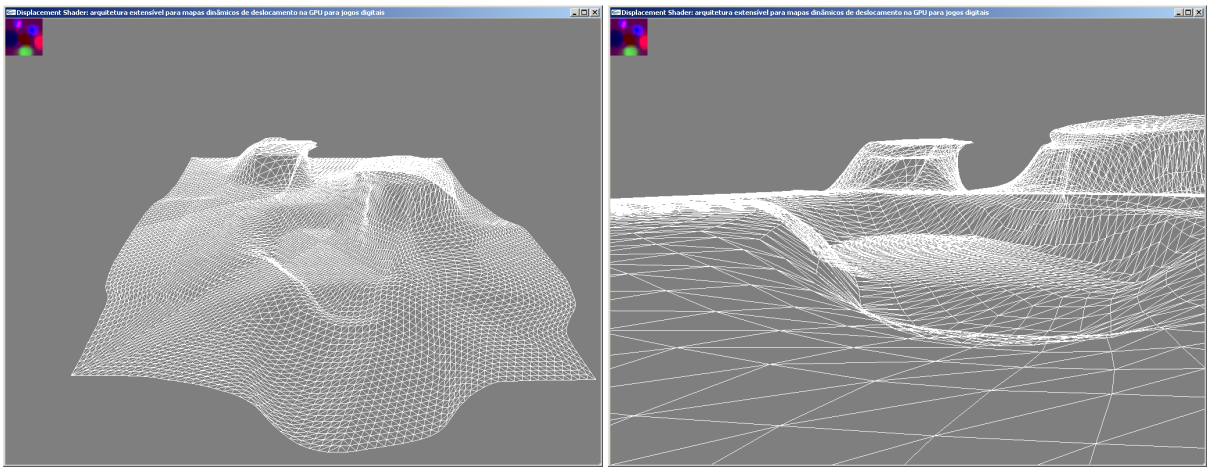

Figura 6.5: Malha após displacement mapping: visão distante (esquerda) e próxima (direita).

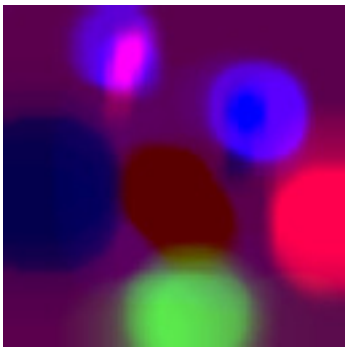

Figura 6.6: Mapa de deslocamento gerado pelo test kernel.

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.

A implementação apresentada neste trabalho, conforme mostrado no diagrama de classes, possui uma cena para demonstrar a funcionalidade de cada um dos módulos específicos (contato, força e *morphing*), os quais tiveram seus resultados apresentados no capítulo 5. As cenas *CustomScene* e *TestScene* são representações de duas vertentes do *custom kernel*. A primeira implementa todas as funcionalidades, mas não faz qualquer intervenção no *kernel*, resultando numa cena estática, onde a geometria é exposta para representar o mapa de deslocamento carregado. A segunda forma utiliza as mesmas funções, porém, reescreve e renomeia o *custom kernel* como *test kernel* e o utiliza dinamicamente com auxílio da interface proposta pelo *decorator*. Essas duas cenas servem para exemplificar o funcionamento do *custom kernel* e validam sua utilização, comprovando a extensibilidade da arquitetura apresentada e deixando clara sua contribuição. A arquitetura extensível proposta neste trabalho tende a crescer para atender as necessidades dos desenvolvedores, utilizando suas próprias contribuições.

A próxima seção apresentará o ambiente de teste, os resultados obtidos em cada etapa do processo, as medições de desempenho da aplicação quando usando diferentes fatores de *tessellation* combinados com a atividade dos *kernels*.

6.1 Testes

Para a construção dos protótipos e testes apresentados neste trabalho foi usado um computador com: placa-mãe Intel® D946GZIS, processador Intel® Core™ 2 CPU 6300 1.86GHz, RAM de 4096MB e sistema operacional Windows 7 Professional 64 bits. A placa de vídeo usada foi ATI Radeon HD 5450 com 80 unidades de processamento de fluxo, GPU Cedar 650MHz e memória de 512MB DDR2 400MHz.

Para o cálculo do valor médio gasto por cada etapa ou função, registrados nas tabelas e gráficos que são apresentados nesta seção, foram utilizados os resultados obtidos após dez execuções, de onde foram retiradas mil amostras de cada e descartados os valores limítrofes.

Na tabela 6.1 estão registrados os tempos gastos nas etapas de configuração referentes às três camadas do modelo: *Displacement Generator*, *Displacement Manager* e *Interoperability Manager* respectivamente.

Etapas de configuração	Tempo (segundos)
Criar contexto OpenCL	1,069039347
Carregar Textura e Registrar no OpenGL	0,005772850
Registrar no OpenCL	0,000009155

Tabela 6.1: Tempos utilizados nas etapas de configuração.

Na tabela 6.2 estão registrados os tempos gastos pelas funções utilizadas para influenciar o mapa de deslocamento. Foram feitas medições nos *kernels* OpenCL, as quais estão registradas na segunda coluna. Medições em funções executadas na CPU também foram registradas visando conseguir valores que pudessem ser comparados com a execução feita em GPU. Observou-se que as execuções feitas em CPU variavam

à medida que a área de abrangência gerada mudava de acordo com o raio escolhido, o que não acontecia com a execução em GPU. Então foram feitas medições utilizando os valores de raio mínimo e máximo para estas execuções, os quais foram registrados na terceira e quarta colunas. As execuções das funções *Morphing* e *Custom* não dependiam de raio e foram registradas apenas na terceira coluna, enquanto na quarta coluna foi usada a sigla N/A (não aplicável).

Função	kernel (GPU)	CPU (raio_mínimo)	CPU (raio_máximo)
Contact	0,333235	0,843431	1,636330
Force	0,366896	0,857993	1,823420
Morphing	0,309522	0,934399	N/A
Custom	0,278857	0,840553	N/A
Test	0,320722	0,852190	1,646820
** Tempos em milissegundos e texturas de tamanho: 64x64*			

Tabela 6.2: Tempos utilizados pelas funções responsáveis pelos mapas de deslocamento.

A figura 6.7 apresenta o gráfico correspondente à tabela 6.2, visando deixar mais fácil a identificação dos valores resultantes e suas comparações.

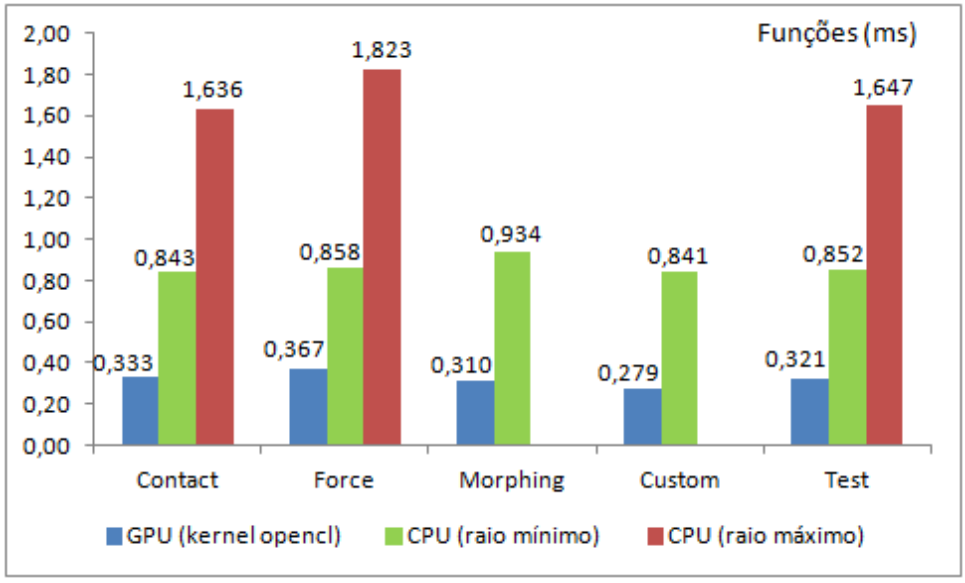

Figura 6.7: Gráfico comparativo dos tempos médios registrados por função usando GPU e CPU.

A seguir são apresentadas tabelas com o registro dos resultados obtidos nos testes feitos com cada cena criada no protótipo apresentado anteriormente. Intercalando essas tabelas estão gráficos referentes às mesmas, nos quais podem ser observadas as quedas de *fps* de acordo com o aumento dos fatores de *tessellation* e da utilização dos *kernels* a cada *frame*. Embora haja essa queda, as aplicações se mantiveram em taxas acima de 100 *fps*, mesmo em *hardware* modesto como o utilizado durante os testes, comprovando serem passíveis de utilização em conjunto com aplicações em tempo real. É importante destacar que em aplicações profissionais, o fator de *tessellation* não será utilizado no valor máximo durante toda a execução do *software*, como as funções do *kernel* também não serão requisitadas a cada quadro. Ambos os itens deverão ser utilizados conforme a necessidade, respeitando suas limitações e feitas as otimizações necessárias por parte dos desenvolvedores.

As tabelas citadas possuem três colunas primárias: modo de visualização, uso do *kernel* e *tessellation factor*. A primeira coluna pode ser preenchida com *wireframe* ou *fill*. O modo de visualização *wireframe* mostra apenas as arestas que formam a malha, enquanto no modo *fill* é mostrada uma textura cobrindo a estrutura. Na segunda coluna, uso do *kernel*, são computados valores *booleanos* (sim e não). No caso positivo, o *kernel* em questão é utilizado a cada *frame*. No caso negativo, o mesmo não é utilizado, confirmando a afirmativa apresentada no fluxograma no início do capítulo; onde o mesmo foi citado como processo alternativo. Qualquer utilização controlada dos *kernels* fará o desempenho da aplicação variar entre esses extremos. A última coluna, *tessellation factor*, é subdividida e abaixo de cada valor são registrados os resultados obtidos nos testes de desempenho, os quais combinam os atributos de cada coluna.

CONTACT SCENE (fps)										
Modo de Visualização	Uso do Kernel	Tessellation Factor								
		1	8	16	24	32	40	48	56	64
Wireframe	não	327,90	293,80	262,10	242,77	227,41	212,38	200,01	191,65	178,92
	sim	201,02	185,56	176,00	165,22	154,57	149,50	143,60	137,75	131,19
Fill	não	242,19	232,62	227,79	221,08	212,27	203,96	195,22	187,13	178,08
	sim	164,17	158,69	155,17	150,38	148,71	144,99	140,18	135,32	130,96

Tabela 6.3: Resultados de testes feitos com a cena de contato.

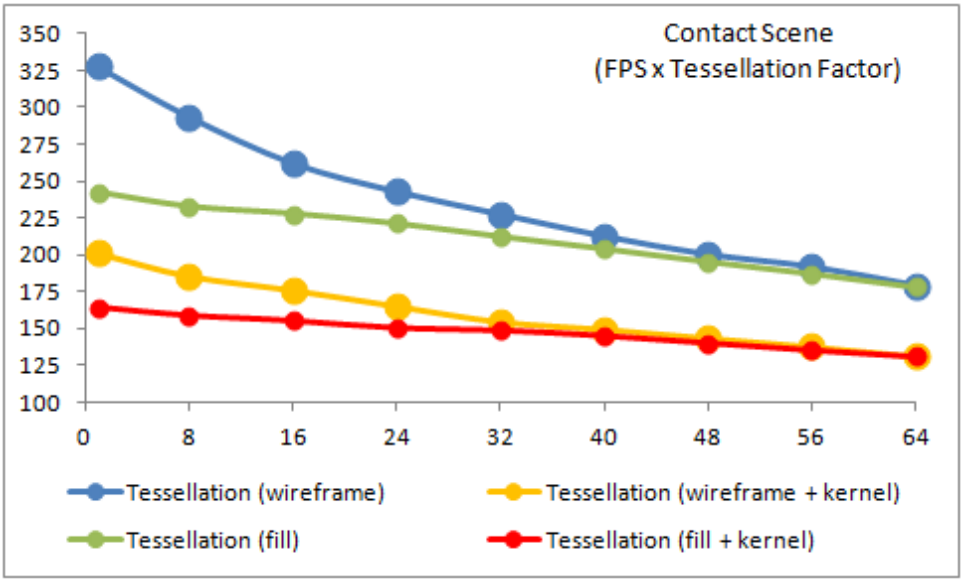

Figura 6.8: Gráfico referente ao desempenho registrado durante os testes com a cena de contato.

FORCE SCENE (fps)										
Modo de Visualização	Uso do Kernel	Tessellation Factor								
		1	8	16	24	32	40	48	56	64
Wireframe	não	330,24	294,50	263,21	243,98	230,16	213,94	201,03	191,36	180,80
	sim	191,21	180,17	165,61	156,69	151,98	143,21	138,01	132,36	129,55
Fill	não	243,63	235,19	231,82	222,76	216,09	205,45	197,10	188,70	180,37
	sim	158,82	156,29	152,87	149,96	146,55	142,73	137,78	131,65	129,09

Tabela 6.4: Resultados de testes feitos com a cena de força.

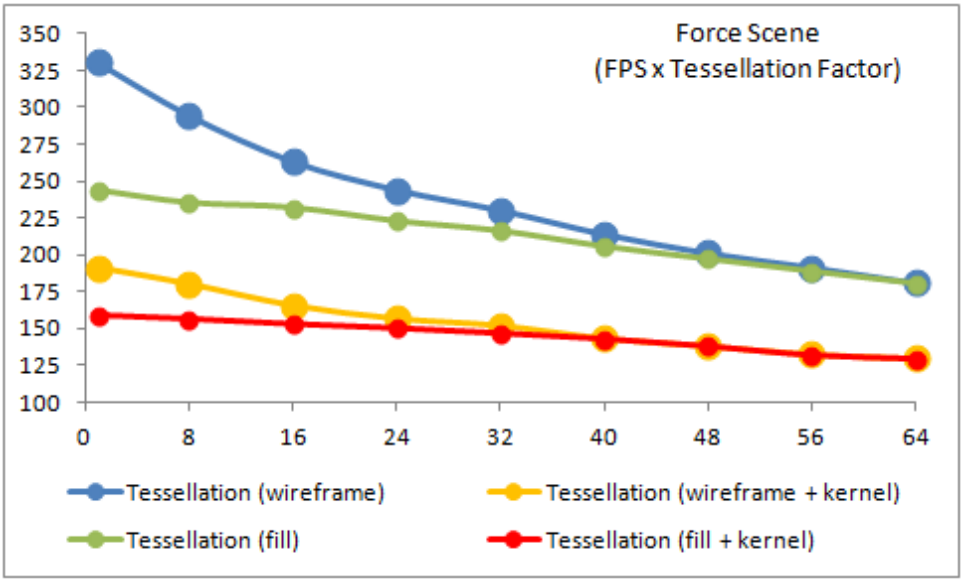

Figura 6.9: Gráfico referente ao desempenho registrado durante os testes com a cena de força.

MORPHING SCENE (fps)										
Modo de Visualização	Uso do Kernel	Tessellation Factor								
		1	8	16	24	32	40	48	56	64
Wireframe	não	330,53	291,20	254,40	238,35	216,35	201,79	189,94	178,66	168,09
	sim	216,45	199,79	183,64	171,01	161,61	154,85	146,77	140,34	133,38
Fill	não	264,80	241,87	226,30	216,70	206,34	199,32	193,40	182,36	172,68
	sim	186,05	170,41	165,97	160,14	156,72	150,79	145,89	141,80	136,59

Tabela 6.5: Resultados de testes feitos com a cena de morphing.

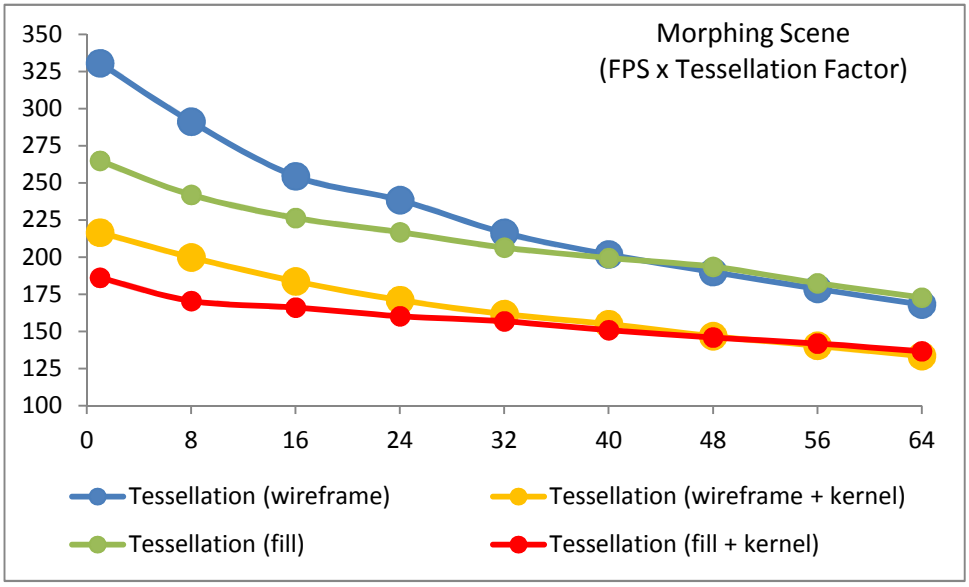

Figura 6.10: Gráfico referente ao desempenho registrado durante os testes com a cena de morphing.

CUSTOM SCENE (fps)										
Modo de Visualização	Uso do Kernel	Tessellation Factor								
		1	8	16	24	32	40	48	56	64
Wireframe	não	326,74	294,85	266,86	244,86	229,05	214,56	200,87	190,40	180,32
	sim	195,71	184,31	169,01	163,48	153,83	149,60	142,95	136,87	131,30
Fill	não	263,57	238,78	231,04	222,18	213,33	205,19	195,95	188,30	178,73
	sim	171,29	160,36	154,18	152,58	144,94	142,76	140,51	135,62	130,64

Tabela 6.6: Resultados de testes feitos com a cena personalizável.

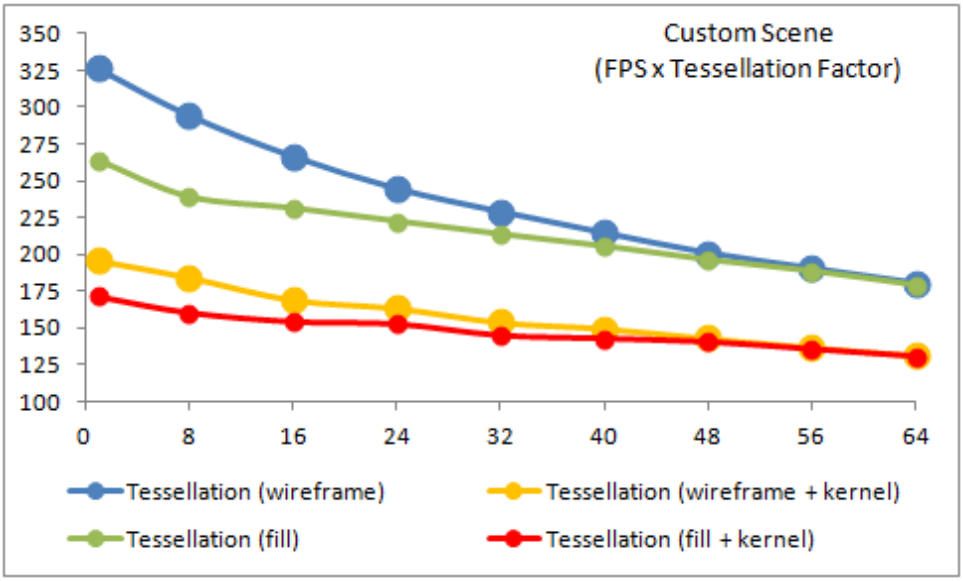

Figura 6.11: Gráfico referente ao desempenho registrado durante os testes com a cena personalizável.

TEST SCENE (fps)										
Modo de Visualização	Uso do Kernel	Tessellation Factor								
		1	8	16	24	32	40	48	56	64
Wireframe	não	311,18	278,87	250,26	232,79	215,23	202,26	191,01	180,37	169,41
	sim	181,23	172,94	161,34	153,91	141,81	140,04	133,62	129,19	123,15
Fill	não	242,14	222,51	216,06	207,59	199,59	192,33	184,39	175,49	167,43
	sim	157,17	148,33	145,42	142,07	139,14	131,06	127,96	121,61	120,35

Tabela 6.7: Resultados de testes feitos com a cena de teste.

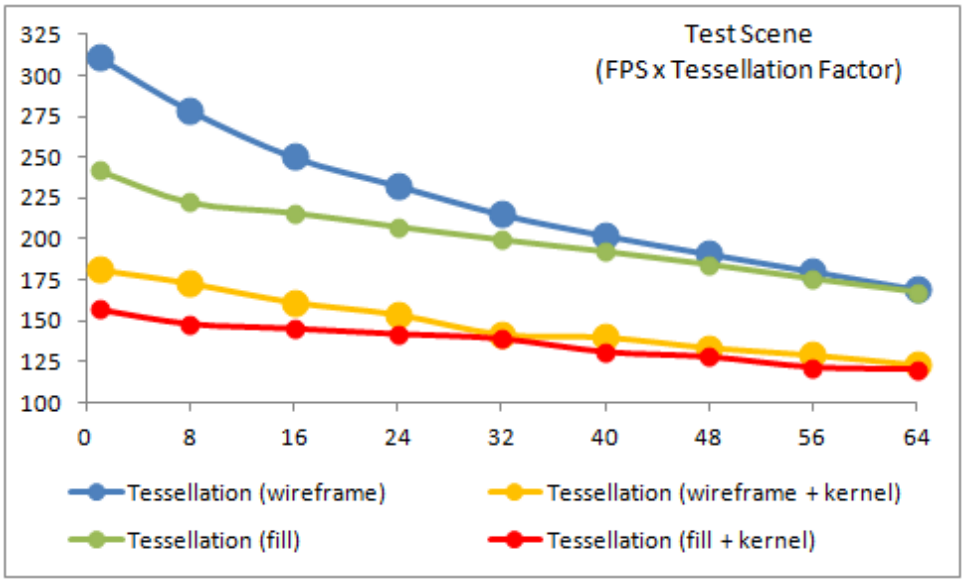

Figura 6.12: Gráfico referente ao desempenho registrado durante os testes com a cena de teste.

No último capítulo serão apresentadas as conclusões e algumas indicações de continuidade para o uso da arquitetura extensível apresentada ao longo deste trabalho.

Capítulo 7

Conclusão e Trabalhos Futuros

A inclusão dos três novos estágios no *pipeline* possibilitou que uma malha simples fosse usada para representar geometrias mais complexas, unindo a *tessellation* com o *displacement mapping*. Isso possibilitou cenas com mais detalhes, uma vez que o custo computacional dessas técnicas agora é direcionado para a GPU, o que não ocorria nos *pipelines* tradicionais. Isso pode trazer como consequência um resultado mais realista em determinadas aplicações, desde que respeitados os limites do *hardware* e com as otimizações necessárias por parte dos desenvolvedores.

O desenvolvimento de protótipos para este trabalho possibilitou um contato prático com o ambiente, esclareceu dúvidas referentes à sua utilização, colaborou para multiplicação da informação através deste trabalho e casou a teoria estudada e pesquisada, citada nas referências, com a experiência prática, gerando o resultado apresentado.

O método de representação de detalhes de topologia proposto neste trabalho foi implementado com sucesso. Sua simplicidade se mostrou um atrativo para implementações de jogos digitais que queiram usar mais detalhamento. O conjunto de antigas técnicas combinados em um novo contexto mostrou-se interessante e um

campo promissor a espera de novos experimentos, sejam utilizando novas técnicas ou simplesmente implementando antigas técnicas de forma criativa.

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.

Embora não seja o enfoque tratar a aplicação de forma mercadológica, não se pode deixar de citar que ao utilizar o OpenGL, a implementação se tornou multiplataforma; e ao se adotar o OpenCL, passou a não ter importância o fabricante de sua placa gráfica. Essas características só têm a acrescentar nas qualidades observadas, fazendo com que a arquitetura esteja disponível para um maior número de desenvolvedores.

Este trabalho apresentou uma proposta de arquitetura extensível para mapas dinâmicos de deslocamento na GPU, implementou algoritmos para utilização de funcionalidades específicas em três *kernels* diferentes (deformação por contato, deformação por força e deformação por morphing), disponibilizou um *kernel* personalizável para uso geral dos desenvolvedores, exemplificou a utilização do controle sobre os mapas de deslocamento e validou o uso do modelo em camadas planejado, comprovado através dos resultados obtidos em cada teste.

Foram pesquisadas e estudadas várias técnicas, metodologias e padrões de desenvolvimento de softwares para planejar a arquitetura proposta, de forma que ela se comportasse de forma prática, acessível e inteligível, visando realmente tentar se

tornar interessante ao ponto de sua característica de extensibilidade se tornar um atrativo para futuras investigações e consequentemente aumentar sua vida útil.

Como trabalhos futuros, propõe-se a união das técnicas apresentadas neste trabalho com as técnicas de *bump mapping* e *parallax mapping*, visando enriquecer os resultados obtidos com a complementação dessas técnicas.

A exploração da arquitetura proposta, dando ênfase à técnica de *morphing* pode ser interessante se colocada no contexto de animação de malhas em tempo real. Ela poderá ser explorada para suportar movimentação de tecidos, simulando animações de capas, bandeiras e outras variantes. Essa vertente também pode ser explorada para simular o envelhecimento de entidades, como personagens ou cenários. As mudanças ao longo do tempo podem ser corretamente exploradas para simular rugas de pele ou deformações e fissuras nos ambientes.

A exploração da arquitetura em conjunto com a física pode ser um campo amplo para novas experimentações, podendo simular o processo de erosão e de outros fenômenos que afetam os relevos ao longo do tempo.

A criação de um jogo digital 3D utilizando a arquitetura proposta nesse trabalho, visando pesquisar a necessidade de otimização de particularidades do processo de *tessellation* aplicado em tempo real e do uso exaustivo de um banco de mapas de deslocamento capazes de cobrir a estrutura do ambiente virtual, armazenando histórico das deformações geradas ao longo do processo de utilização do jogo, também se torna bastante interessante para complementar o trabalho apresentado.

A inclusão de novos módulos na arquitetura é um ponto que desperta bastante interesse. Módulos de física para reconhecimento da geometria de forma a suportar um nível mais realista de cálculos de colisão. Módulos de tratamento de textura para contribuir com mais realismo, visando influenciar dinamicamente o visual dos ambientes e entidades dos jogos. As texturas poderiam receber diversas informações

para complementarem as deformações apresentadas neste trabalho. Um exemplo simples, como uma explosão, além de deformar o ambiente com o uso do mapa de deslocamento, poderia escurecer a textura na área afetada dando mais realismo à ocorrência.

Enfim, muitos caminhos se mostram aptos para dar continuidade à arquitetura extensível proposta neste trabalho. Quais destes serão seguidos, somente o futuro poderá responder, mas a contribuição deixada é real e cabe aos desenvolvedores torná-la cada vez mais interessante.

Referências Bibliográficas

[ALEXANDER et al. 1977] ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M.; IACOBSON, M.; FIKSDAHL-KING, I.; and ANGEL, S. A Pattern Language. Oxford University Press, New York, 1977.

[ANDRADE and CLUA 2011] ANDRADE, F.C., CLUA, E.W.G. 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, 2011.

[ANDRADE et al. 2012] ANDRADE, F.C., SHAFAATDOOST, M., CONCI, A., CLUA, E.W.G. Displacement Stage: Arquitetura extensível para mapas dinâmicos de deslocamento na GPU. In XI Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2012, Brasília, 2012.

[BASS et al. 2003] BASS, L.; CLEMENTS, P.; KAZMAN, R. Software Architecture in Practice. 2. ed. Boston: Addison Wesley, 2003.

[BATISTA 2011] BATISTA, M.L.S. Simulação de emoções em faces humanas utilizando os algoritmos de bump mapping e morphing implementados na GPU. Dissertação (Mestrado em Computação), Universidade Federal Fluminense, Rio de Janeiro, 2011.

[BLINN 1978] BLINN, J. F. Simulation of wrinkled surfaces. In SIGGRAPH 78, p.286-292, 1978.

[BUSCHMANN et al. 1996] BUSCHMANN, F. et al. Pattern-Oriented Software Architeture. Chichester: Wiley, 1996.

[CATMULL 1974] CATMULL, E. A subdivision algorithm for computer display of curved surfaces. Ph.D. thesis, Department of Computer Science, University of Utah, Salt Lake City, UT, 1974.

[CONCI et al. 2008] CONCI, A., AZEVEDO, E., LETA, F. Computação gráfica: teoria e prática [vol.2]. Elsevier, Rio de Janeiro, 2008.

[COOK 1984] COOK, R. L. Shade trees. In SIGGRAPH 84, p.223-231, 1984.

[DIGITAL DAILY] Digital-Daily.com. Disponível em: http://www.digital-daily.com/video/radeon_hd2900xt/print [Acessado em 19 de julho de 2012].

[DIRECTX] DirectX 11. Disponível em: http://msdn.microsoft.com/en-us/directx/aa937781.aspx [Acessado em 19 de julho de 2012].

[EVGA] EVGA: DirectX 11 and Tessellation. Disponível em: http://www.youtube.com/watch?v=-uavLefzDuQ [Acessado em 19 de julho de 2012].

[GAMEDEV] Gamedev.net. Disponível em: http://www.gamedev.net/topic/531164-d3d11-hw-tessellation-for-terrain-rendering/ [Acessado em 19 de julho de 2012].

[GAMMA et al. 1995] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J.. Design Patterns, Elements of Reusable Object-Oriented Software. Indianapolis: Addison-Wesley, 1995.

[GLSL] GLSL. Disponível em: http://www.opengl.org/documentation/glsl/ [Acessado em 19 de julho de 2012].

[GPU GEMS] GPU GEMS 2. Disponível em: http://http.developer.nvidia.com/GPUGems2/gpugems2_frontmatter.html [Acessado em 19 de julho de 2012].

[HEINEMAN et al. 2009] HEINEMAN, G.T., POLLICE, G., SELKOW, S. Algoritmos: O Guia Essencial [vol.2]. Alta Books, Rio de Janeiro, 2009.

[HIGA 2008] HIGA, R.S. Uma proposta de morphing utilizando técnicas de interpolação de formas e média morfológica. Dissertação (Mestrado em Computação), Unicamp, Campinas, 2008.

[ID SOFTWARE] id Software. Disponível em: http://www.idsoftware.com/ [Acessado em 19 de julho de 2012].

[KIRK and HWU 2011] KIRK, D.B., Hwu, W.W. Programando para processadores paralelos: uma abordagem prática à programação de GPU. Elsevier, Rio de Janeiro, 2011.

[LIMA and BRAUN 2008] LIMA, D. S.; BRAUN, H. Exibição de terrenos em tempo real: Uma abordagem a terrenos com larga escala geométrica. Dissertação de mestrado. Pontifica Universidade Católica do Rio Grande do Sul, 2008.

[LOOP and SCHAEFER 2008] LOOP, C. and SCHAEFER, S. Approximating Catmull-Clark subdivision surfaces with bicubic patches. In ACM TOG v.27 n.8, 2008.

[VS 2010] Microsoft Visual Studio 2010. Disponível em: http://msdn.microsoft.com/pt-br/library/dd831853.aspx [Acessado em 19 de julho de 2012].

[NI et al. 2009] NI, T.; CASTAÑO, I.; PETERS, J.; MITCHELL, J.; SCHNEIDER, P.; and VERMA, V. Efficient substitutes for subdivision surfaces. In ACM SIGGRAPH 2009 Courses, n.13, 2009.

[NVIDIA] NVIDIA: DirectX 11 Tesssellation. Disponível em: http://www.nvidia.com/object/tessellation.html [Acessado em 19 de julho de 2012].

[NOVAK 2011] NOVAK, J.: Desenvolvimento de games. Cengage Learning, São Paulo, 2011.

[NUNES 2011] NUNES, G.B. Explorando aplicações que usam geração de vértices em GPU. Dissertação (Mestrado em Informática), Pontifícia Universidade Católica do Rio de Janeiro (PUC-RIO), Rio de Janeiro, 2011.

[OPENCL] OpenCL. Disponível em: http://www.khronos.org/opencl/ [Acessado em 19 de julho de 2012].

[OF] Openframewoks. Disponível em: http://www.openframeworks.cc/ [Acessado em 19 de julho de 2012].

[OPENGL] OpenGL. Disponível em: http://www.opengl.org/ [Acessado em 19 de julho de 2012].

[PHARR and HANRAHAN 1996] PHARR, M., and HANRAHAN, P. Geometry caching for ray-tracing displacement maps. 7th Eurographics Rendering Workshop, 31–40, 1996.

[PIXAR] PhotoRealistic RenderMan [online]. Disponível em: http://renderman.pixar.com/view/renderman [Acessado em 19 de julho de 2012].

[PRESSMAN 2002] PRESSMAN, R. S. Engenharia de Software. 5. ed. Rio de Janeiro: McGraw-Hill, 2002.

[RABIN 2012] RABIN, S. Introdução ao desenvolvimento de games [vol.2]. Cengage Learning, São Paulo, 2012.

[ROLLINGS and MORRIS 2000] ROLLINGS, A., MORRIS, D. Game architecture and design. The Coriolis Group, Scottsdale, Arizona, 2000.

[SCHEIN et al. 2005] SCHEIN, S., KARPEN, E., ELBER, G. Real-time geometric deformation displacement maps using programmable hardware. The Visual Computer (September), 791–800, 2005.

[SCHOENBERG 2001] SCHOENBERG, F.P. Tessellations. Department of Statistics University of California, Los Angeles, 2001.

[SEGA] SEGA®: Aliens VS. Predator™. Disponível em: http://www.sega.com/games/aliens-vs-predator/ [Acessado em 19 de julho de 2012].

[SNOOK 2003] SNOOK, G. Real-time 3D terrain engines using C++ and DirectX 9. Charles River Media Inc., Massachussetts, 2003.

[STALKER] S.T.A.L.K.E.R. Call of Pripyat. Disponível em: http://cop.stalker-game.com/ [Acessado em 19 de julho de 2012].

[TAKAHASHI and MIYATA 2005] TAKHASHI, M., MIYATA, K.: GPU based interactive displacement mapping. International Workshop on Advanced Image Technology, 105-108, 2005.

[TATARCHUK et al. 2009] TATARCHUK, N., BARCZAK, J., BILODEAU, B. Programming for real-time tessellation on GPU, 2009.

[TATARINOV 2008] TATARINOV, A. Instanced Tessellation in DirectX10. In GDC ’08: Game Developers’ Conference, 2008.

[THQ] THQ: Metro 2033™. Disponível em: http://metro2033.thq.com/es [Acessado em 19 de julho de 2012].

[TORTELLI and WALTER 2007] TORTELLI, D.M., WALTER, M.: Implementação da técnica de displacement mapping em hardware gráfico, In VI Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital, p.1-4, 2007.

[VALVE] Valve. Disponível em: http://www.valvesoftware.com/ [Acessado em 19 de julho de 2012].

[WANG et al. 2003] WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., SHUM, H. View-dependent displacement mapping. In ACM TOG v.22 n.3, 2003.

[WOLBERG 1998] WOLBERG, G. Image morphing: a survey. The Visual Computer 14, 8/9, 360-372, 1998.

[ZAMITH et al. 2009] ZAMITH, M., CLUA, E.W.G., MONTENEGRO, A., PASSOS, E., LEAL, R., CONCI, A.: Real time feature-based parallel morphing in GPU applied to texture-based animation. In: 16th International Workshop on Systems, Signals and Image Processing, Chalkida, Grécia. IEEE Proceedings of the 16th International Workshop on Systems, Signals and Image Processing. London : IEEE - Region 8, v. 16. p. 145-150, 2009.