UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CHRISTIAN AZAMBUJA PAGOT

Shadow Mapping com Múltiplos Valores de Profundidade

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. João Luiz Dihl Comba
Orientador

Prof. Dr. Manuel Menezes de Oliveira Neto
Co-orientador

Porto Alegre, maio de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Pagot, Christian Azambuja

Shadow Mapping com Múltiplos Valores de Profundidade / Christian Azambuja Pagot. – Porto Alegre: PPGC da UFRGS, 2005.

81 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2005. Orientador: João Luiz Dihl Comba; Co-orientador: Manuel Menezes de Oliveira Neto.

1. Sombras. 2. Hardware gráfico. 3. Shadow map. 4. PCF. I. Comba, João Luiz Dihl. II. Oliveira Neto, Manuel Menezes de. III. Título.

“– Calma! Voces ainda nao sabem o que está ai dentro. Vou na frente.”
— GANDALF

AGRADECIMENTOS

Ao meu orientador João Comba, e co-orientador Manuel Oliveira, por todo o suporte e estímulo recebido durante a execução deste trabalho.

A todo o pessoal do grupo de computação gráfica da UFRGS (e alguns que por lá já passaram), em especial: Profª. Carla Freitas, Profª. Luciana Nedel, Rui Bastos, Carlos Dietrich, Carlos Scheidegger, Atila Bohlke, Rodrigo Luque, Thiago Paim, Stefan Zanona, Diego Martins, Fábio Bernardon, Leandro Fernandes, João Prauchner, Eduardo Pons, Dalton, Raquel Pillat, Leonardo Machado e Marcus Farias.

Agradecimentos muito especiais à minha família: minha mãe Lilian, minha irmã Marcele e aos meus avós Celony e Walter.

Por fim, mas não menos importante, meus agradecimentos a todos os contribuintes deste país, que acreditam e ajudam a manter em funcionamento as nossas instituições públicas de ensino.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
GLSL	*OpenGL Shading Language*
HLSL	*High Level Shading Language*
NDC	*Normalized Device Coordinates*
PCF	*Percentage Closer Filtering*
PSM	*Perspective Shadow Maps*
RMSL	*RenderMan Shading Language*
RTSL	*Real Time Shading Language*
SIMD	*Single-Instruction/Multiple-Data*
SMMVP	*Shadow Map* com Múltiplos Valores de Profundidade
TSC	Teste de Sombra Composto

LISTA DE FIGURAS

RESUMO

Um dos algoritmos para cálculo de sombras mais eficientes existentes atualmente é o *Shadow Mapping* de Williams. Ele é simples, robusto e facilmente mapeável para o *hardware* gráfico existente. Este algoritmo conta com duas etapas. A primeira é responsável pela geração de um *depth buffer* (*Shadow Map*) a partir do ponto de vista da luz. Na segunda etapa a imagem final da cena é gerada a partir do ponto de vista da câmera. De maneira a determinar se os *pixels* da imagem final estão iluminados ou em sombra, cada *pixel* é transformado para o espaço da luz e testado contra o *Shadow Map*.

Shadow Maps tradicionais armazenam apenas um valor de profundidade por célula, fazendo com que os testes de sombra retornem valores binários. Isso pode ocasionar o surgimento de *aliasing* nas bordas das sombras. Este trabalho apresenta uma nova abordagem capaz de produzir melhores resultados de suavização que, em conjunto com o algoritmo de *PCF* (*Percentage Closer Filtering*), reduz o serrilhado das bordas das sombras através do uso de filtros de menor tamanho. O novo algoritmo estende os conceitos de *Shadow Map* e de *teste de sombra* de forma a suportarem múltiplos valores de profundidade. Esta nova abordagem apresenta potencial para implementação em *hardware*, e também pode ser implementada explorando a programabilidade das recentes placas gráficas.

Palavras-chave: Sombras, hardware gráfico, shadow map, PCF.

Multiple Depth Shadow Maps

ABSTRACT

William's Shadow Mapping is one of the most efficient hard shadow algorithms. It is simple, robust and can be easily mapped to the actual grapics hardware. It is a two-pass technique. In the first pass a depth buffer (Shadow Map) is created from the light's view point. In the second pass the final image is rendered from the camera's view point. In order to decide whether each pixel in the camera's view is lit or in shadow with respect to the light source, the pixel is transformed into the light space, and tested against the Shadow Map.

Shadow maps store a single depth value per cell, leading to a binary outcome by the shadow test, and are prone to produce aliased shadow borders. This work presents a new approach that produces better estimates of shadow percentages and, in combination with percentage closer filtering (PCF), reduces aliasing artifacts using smaller kernel sizes. The new algorithm extends the notions of shadow map and shadow test to support the representation of multiple depth values per shadow map cell, as well as multi-valued shadow test. This new approach has the potential for hardware implementation, but can also be implemented exploiting the programmable capabilities of recent graphics cards.

Keywords: shadows, graphics hardware, shadow map, PCF.

1 INTRODUÇÃO

A última década tem representado uma verdadeira revolução para a área da computação gráfica interativa. O surgimento e a disponibilização em massa de *hardware* gráfico dedicado para computadores pessoais, a preços acessíveis, pode ser considerado como um dos principais fatores dessa revolução. Rotinas referentes à geração de imagens, tradicionalmente onerosas e repetitivas, passaram a ser executadas de forma mais eficiente no novo *hardware*, enquanto que a *CPU* pode se dedicar mais à gerência das aplicações. Inicialmente esse *hardware* era pouco flexível, sendo apenas configurável, mas o posterior surgimento das *GPU*s (*Graphics Processing Unit*), com suas unidades programáveis, reduziu em muito essas limitações. Agora, além de *hadware* gráfico extremamente eficiente, unidades completamente programáveis passaram a estar ao alcance do público em geral.

Várias forças favoreceram o surgimento e a evolução deste *hardware*. Segundo (FERNANDO; KILGARD, 2003), a primeira é o comprometimento da própria indústria em dobrar o número de transistores dos processadores a cada 18 meses aproximadamente. Este aumento constante no número de transistores, conhecido como *Lei de Moore* (MOORE, 1965), pode ser interpretado como o desenvolvimento de *hardware* cada vez mais poderoso a preços mais acessíveis. A segunda força é a grande capacidade computacional exigida pelas técnicas de computação gráfica. A terceira é o desejo do ser humano de ser visualmente estimulado. Essa é a força que conecta a evolução do *hardware* à busca de mais realismo nas imagens sintéticas.

Embora as *GPU*s sejam processadores programáveis assim como as *CPU*s, sua arquitetura é bastante diferente. As *GPU*s são processadores especializados que implementam em *hardware* o algoritmo de *scan-based rasterization*. Por ser um algoritmo facilmente paralelizável, ainda mais se forem considerados modelos locais de iluminação, nada mais natural que a *GPU* seguisse a mesma linha. Atualmente as *GPU*s contam com um arquitetura do tipo *SIMD* (*Single-Instruction/Multiple-Data*), composta por diversas unidades de processamento paralelas, que implementam um modelo de computação baseada em fluxos.

A disponibilidade de um processador que implemente este eficiente modelo de computação tem provocado uma revisita dos algoritmos tradicionais. Diversos estudos e trabalhos têm sido publicados sobre a adaptação de algoritmos para o modelo de computação baseada em fluxos, e isto não tem sido diferente em relação aos algoritmos para determinação de sombras em cenas 3D.

1.1 Motivação

O algoritmo para geração de sombras que melhor se adapta à implementação em *hardware* é o *Shadow Mapping* de Williams (WILLIAMS, 1978). Além de se tratar de um método genérico para o cálculo de sombras, não impondo restrições à geometria da cena, apresenta uma grande similaridade com o próprio *pipeline* gráfico implementado no *hardware*.

Embora recursos dedicados exclusivamente à geração de sombras através de *Shadow Mapping* estejam disponíveis nas placas gráficas, eles têm sido pouco explorados na prática. Isso se deve principalmente ao problema de *aliasing* presente nas bordas das sombras geradas por este método.

Para que o uso do algoritmo de *Shadow Mapping* torne-se uma alternativa viável na prática, seus problemas devem ser contornados. O algoritmo de *PCF*, proposto por Reeves em 1978 (REEVES; SALESIN; COOK, 1987), tem como objetivo minimizar os efeitos de *aliasing* através do emprego de um filtro de convolução sobre os resultados dos testes de sombra do *Shadow Mapping*. O algoritmo estima a atenuação final da cor de um pixel em sombra, ou próximo às suas bordas, através de uma média aritmética feita com um determinado conjunto de resultados de testes de sombra individuais. O resultado visual gerado por este algoritmo é a transição suave e gradual entre regiões de sombra e regiões iluminadas.

Entretanto, o resultado da filtragem obtida com o *PCF* depende da quantidade de elementos de seu filtro, sendo que resultados satisfatórios são alcançados somente quando o seu número de elementos começa a interferir na performance da aplicação, tornando-a pouco atraente em se tratando de aplicações de tempo real. Adaptações ao algoritmo de *PCF*, que permitissem melhores suavizações de bordas a partir de um filtro de dimensões menores (com menor impacto sobre a performance), poderiam tornar o *Shadow Mapping* uma opção interessante para a geração de sombras em tempo real.

A observação de que o *PCF* executa um procedimento de filtragem sobre um conjunto de valores binários foi o principal motivador deste trabalho. De certa forma, isso limita a capacidade de suavização do *PCF*, visto que caso seus valores não estivessem limitados a um conjunto binário, um número maior de níveis de atenuação poderiam ser obtidos com o uso de um filtro de tamanho fixo. Nosso trabalho partiu desta observação.

1.2 Contribuições

As contribuições deste trabalho são:

- Uma generalização do *Shadow Map*, no sentido de agora a nova estrutura suportar múltiplos valores de z por célula.
- Uma generalização dos testes de sombra, no sentido de que agora esses novos testes verificam relações entre mais de 2 valores de z.
- Apresentação de um método de geração de sombras, baseado no espaço da imagem, que:

  - pode ser utilizado em aplicações que demandam interatividade.
  - apresenta qualidade de filtragem das bordas de sombra, através do uso de filtros de menores dimensões, equivalente aos obtidos atualmente através de filtros maiores.
  - apresenta melhor performance no caso de cenas que geram um grande número de fragmentos.

- Análise, a nível de fragmento, dos desempenhos dos algoritmos de *Shadow Mapping* com *PCF* e do novo método.

1.3 Estrutura do trabalho

O capítulo dois apresenta uma revisão sobre o *hardware* gráfico atual. O capítulo começa apresentando o *pipeline* gráfico, e a seguir a estrutura do *hardware* que o implementa. Nas sessões seguintes cada elemento do hardware é apresentado em detalhe. O capitulo três fala sobre sombras e trabalhos relacionados. Inicialmente é abordado o fenômeno físico sombra, suas causas, importância na percepção e no aumento do realismo. A seguir é apresentado um modelo simplificado e eficiente para determinação de sombra. A seguir são apresentados em detalhe os algoritmos de *Shadow Mapping* e *PCF*, fundamentais ao entendimento do método apresentado neste trabalho. O capítulo termina com a apresentação de uma série de trabalhos relacionados que representam o estado da arte na área.

O capítulo quatro apresenta o novo método. Inicialmente são apresentadas as motivações do trabalho e as intuições seguidas no seu desenvolvimento. A seguir são apresentadas as formalizações, e os diferentes métodos utilizados na construção dos *Shadow Maps* com múltiplos valores de profundidade, bem como suas limitações. Por fim, é apresentada a implementação do nosso programa seguida de alguns trechos de código. O capítulo cinco apresenta os resultados obtidos seguidos de uma discussão detalhada acerca do desempenho do algoritmo. Também são apresentados e discutidos os resultados obtidos com uma aplicação desenvolvida especificamente para o teste de desempenho do novo método. São apresentados os resultados obtidos em dois modelos de placas gráficas diferentes. O trabalho encerra com o capítulo seis, onde são apresentadas as conclusões e trabalhos futuros.

2 O *HARDWARE* GRÁFICO

O objetivo deste trabalho é a definição de um algoritmo de suavização de bordas de sombra que tenha boa performance e cuja implementação tire proveito do paralelismo disponível no *hardware* gráfico atual. De forma a facilitar o entendimento do algoritmo e justificar algumas escolhas feitas durante seu desenvolvimento, resolvemos dedicar este capítulo inteiramente a uma introdução ao *hardware* gráfico. Inicialmente falaremos sobre a estrutura do *pipeline* gráfico – que serviu como base para a construção do *hardware* – e a seguir da evolução deste *hardware* até os dias atuais. A seguir comentamos em mais detalhe alguns de seus elementos principais, enfatizando seu funcionamento e organização. Por fim apresentamos uma visão geral sobre as linguagens utilizadas na sua programação e a utilização de *caches* de textura em *hardware*.

2.1 O *pipeline* gráfico

O *pipeline* gráfico representa uma estrutura composta por diversos estágios que, a partir da especificação de uma cena 3D (composta por câmera, objetos tridimensionais, fontes de luz, etc.), gera uma imagem bidimensional. Como esta breve introdução ao *pipeline* gráfico servirá apenas como subsídio para o entendimento da estrutura do *hardware*, tomaremos como base a estrutura apresentada por Möller e Haines em (MOLLER; HAINES, 2002).

Os principais estágios de um pipeline gráfico para geração de imagens em tempo real são: aplicação, geometria e rasterização. Essa estrutura pode ser observada na figura 2.1. O estágio da aplicação refere-se a aplicação em si. Nesse estágio é que são

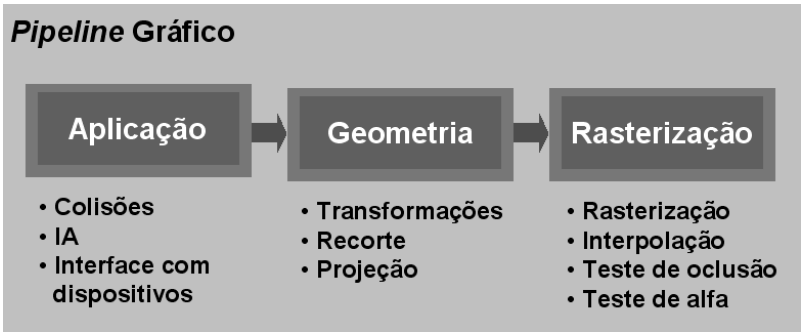


Figura 2.1: Organização de um *pipeline* gráfico para geração de imagens em tempo real. Adaptado de (MOLLER; HAINES, 2002).

feitos os cálculos referentes à detecção de colisões, inteligência artificial, física, interface com periféricos, etc. Esse estágio será responsável por enviar ao estágio seguinte a lista de vértices que compõem os polígonos da cena e as respectivas matrizes de transformação. O próximo estágio, da geometria, é responsável por aplicar transformações sobre os vértices, de forma a transferí-los do espaço geométrico para o espaço da tela. Essa transformação é dividida em várias etapas. Primeiro são aplicadas sobre os vértices as transformações afins, tais como rotação, translação, escala, etc. Isso transformará os vértices, originalmente no espaço do objeto, para o espaço do universo. A seguir uma segunda transformação (referente à projeção) transfere esses vértices para o espaço *NDC* (*Normalized Device Coordinates*). Este espaço é definido por um cubo, cujos pontos extremos se encontram geralmente em (-1,-1,-1) e (1,1,1). A seguir operações de recorte são aplicadas, removendo da cena polígonos que se encontram total ou parcialmente fora do volume de visualização. Como último passo desse estágio, os polígonos restantes são finalmente projetados sobre a tela.

No estágio de rasterização, os polígonos, já no espaço da tela, são rasterizados. Essa discretização dos polígonos dará origem a uma série de fragmentos que poderão eventualmente se tornar *pixels* da imagem final. As propriedades desses fragmentos, tais como cor, profundidade, coordenadas de textura, são geradas a partir da interpolação linear das respectivas propriedades do polígono que os originou. Nesse estágio também é feito o procedimento de mapeamento de textura. Depois de definida a cor final do fragmento, este será submetido a uma série de testes, tais como oclusão, alfa, etc. Ao passar com sucesso por estes testes, o fragmento passa a contribuir para a imagem final, na forma de pixel.

A estrutura de funcionamento do estágio de aplicação pode ser bastante complexa, e depende fundamentalmente do objetivo da aplicação. Por outro lado, os estágios de geometria e rasterização possuem uma funcionalidade bem específica, e que varia muito pouco em função do tipo de aplicação. Transformações de vértices e rasterização de polígonos continuarão sendo feitas da mesma forma, não importando se a aplicação é um simulador mais sofisticado ou simples jogo. O fato de os estágios de geometria e rasterização terem uma funcionalidade bem específica, e executarem operações repetitivas sobre elementos simples, fez com que pudessem ser implementados em hardware, aumentando sua eficiência. Este será o assunto do restante deste capítulo.

2.2 As origens do *hardware* gráfico

Os primórdios do *hardware* encontrado atualmente nas placas gráficas remete aos tempos das *Connection Machines* (HILLIS, 1985). As *Connection Machines* foram uma série de computadores de arquitetura *SIMD* massivamente paralelas produzidos pela companhia *Thinking Machines Corporation* no início dos anos 80. Uma *Connection Machine* era composta de um módulo central que controlava milhares de unidades de processamento adicionais extremamente simples. Cada uma destas unidades de processamento possuía uma unidade própria de memória. A topologia de intercomunicação entre essas unidades de processamento definia um hipercubo. O módulo central era encarregado de decodificar o programa e propagar as instruções para as unidades de processamento, que as executavam em paralelo. A arquitetura das *Connection Machines* foi utilizada na ex-

ploração de algoritmos para diversas áreas, entre elas dinâmica de fluídos, programação dinâmica, *ray tracing*, problemas de geometria computacional, etc.

Na mesma época, Henry Fuchs desenvolveu a arquitetura *Pixel-Planes* (FUCHS; POULTON, 1982). Tratava-se, assim como as *Connection Machines*, de uma arquitetura *SIMD* massivamente paralela, mas voltada especificamente à geração de imagens. Nessa nova arquitetura, cada *pixel* contava com um processador geral bastante simples e uma quantidade de memória suficiente para armazenar sua cor, profundidade e outras informações. Uma matriz destes *pixels* compunha o que era chamado de *Frame-Buffer* Inteligente. Este *Frame-Buffer* podia ser visto como uma superfície que recebia descrições de primitivas geométricas, na forma de coeficientes, e algumas instruções que eram localmente executadas, a nível de *pixel*. Como as instruções, os endereços de memória e os coeficientes das primitivas eram propagados para todos os processadores dos *pixels*, o *Frame-Buffer* Inteligente formava um processador do tipo *SIMD* que contava uma topologia de interconexão bem simples entre seus elementos.

No caso específico da computação gráfica esses projetos serviram para mostrar como o paralelismo inerente ao *pipeline* gráfico e aos modelos locais de iluminação neles implementados poderiam ser explorados como uma forma de aumentar a sua performance. Implementações de alguns efeitos globais de iluminação, tais como sombras, também foram possíveis em arquiteturas desse tipo (FUCHS et al., 1985), mostrando o quão gerais elas poderiam ser. A *Evans&Sutherland* e a *SGI* foram as primeiras empresas a fabricar *hardware* gráfico dedicado em larga escala. Os sistemas gráficos desenvolvidos por essas empresas introduziram muitos dos conceitos, tais como as unidades de vértice e de fragmento, encontrados no *hardware* atual. Suas máquinas eram extremamente poderosas, mas ao mesmo tempo muito caras para serem adquiridas pelo público em geral. Seu mercado se restringia a grandes instituições ou empresas que empregavam suas máquinas em pesquisa, simulações ou processos de visualização que exigiam grande poder computacional.

À medida que os jogos de computador foram se popularizando, e as interfaces gráficas dos sistemas tornando-se mais complexas, iniciativas em direção a aceleração gráfica começaram a ser tomadas por alguns fabricantes de *hardware* para computadores pessoais. Os primeiros aceleradores gráficos para computadores pessoais não passavam de *framebuffers* construídos em cima de uma memória dedicada mais rápida que as utilizadas tradicionalmente. Esse novo *hardware* limitava-se a exibição mais eficiente de grandes quantidades de *pixels*, previamente rasterizados e coloridos. Em 1996 a já extinta empresa *3Dfx* apresentou o que veio a ser considerado o primeiro acelerador gráfico de fato para computadores pessoais: o processador *Voodoo3* (FERNANDO; KILGARD, 2003). Este processador era capaz de receber triângulos já transformados pela *CPU*, rasterizá-los, mapear texturas devidamente filtradas e com correção perspectiva, etc. Com o lançamento da placa *GeForce256* a NVIDIA introduziu no mercado de placas gráficas para computadores pessoais o estágio de aceleração de geometria, conhecido como *Transform and Lighting* (*T&L*). Entretanto, a utilização de funções fixas limitava o seu uso.

A posterior introdução dos processadores de vértices e de fragmentos nestes aceleradores permitiu a utilização de linguagens de *shading* (seção 2.5) na geração de imagens em tempo real. Essas novas unidades, agora programáveis, deixaram de ser meros acelera-

dores gráficos para se transformarem em verdadeiras unidades de processamento gráfico, ou *GPU*s (*Graphics Unit Processors*). Agora as *GPU*s contavam com praticamente todos os recursos necessários à completa transferência do estágio de rendering da *CPU* para a *GPU*.

Atualmente existem duas empresas no mundo que dominam o mercado de fabricação de *GPU*s: a *NVIDIA* e a *ATi*. Além dessas, outras empresas de menor expressão no mercado das placas gráficas, tais como a *XGI* e *S3* entre outras, também têm produzido *hardware* gráfico dedicado. Embora existam diferenças nas arquiteturas dos processadores produzidos por esses fabricantes, essas arquiteturas têm convergido. Atualmente diversos elementos são comuns ao *hardware* de muitos desses fabricantes, com variações apenas no que diz respeito às suas implementações. A seguir apresentamos uma descrição mais detalhada de alguns desses elementos, de forma geral, sem nos preocuparmos com detalhes específicos de algum fabricante em particular.

2.3 *Buffers*

Praticamente todas as placas gráficas atuais contam com algum tipo de processador gráfico (*GPU*) e uma memória dedicada. Essa memória é compartilhada tanto pelas texturas quanto pelos *buffers* do sistema. O *hardware* gráfico atualmente conta com uma série de diferentes tipos de *buffers*. Enquanto alguns *buffers* mais consagrados são encontrados em praticamente qualquer sistema, como por exemplo o *color-buffer* ou o *depth-buffer*, outros menos populares, como os *pbuffers*, aparecem apenas em placas mais recentes. Embora os *buffers* já possuam estruturas específicas definidas de acordo com sua função, alguns parâmetros adicionais, tais como resolução, precisão ou política de amostragem, podem por vezes serem definidos pelo próprio usuário.

O *color-buffer* e o *depth-buffer* compõem o *framebuffer*. Os elementos do *color-buffer* armazenam as componentes de cor de cada *pixel* da tela. Embora os sistemas permitam ao usuário escolher entre diferentes modos de funcionamento, a evolução do *hardware* tem permitido a popularização dos modos *high-color* e *true-color*. A representação interna e o espaço de memória ocupado pelo *color-buffer* dependerão do modo utilizado. No modo *high-color* cada elemento do *color-buffer* pode ocupar 15 ou 16 bits, fazendo com que o número máximo de cores distintas fique em 32.768 ou 65.536 cores, respectivamente. No modo *true-color* os elementos do *color-buffer* podem ocupar 24 ou 32 bits. No caso de se utilizar 24 bits para a representação da cor do pixel, cada um de seus elementos de cor (RGB) fica sendo representado por 8 bits. Isso permite a representação de aproximadamente 16 milhões de cores distintas. No caso da cor ser especificada com 32 bits, as componentes da cor continuam sendo representados por cerca de 8 bits, enquanto que os 8 bits restantes ficam dedicados à componente *alpha*, permitindo a representação do *pixel* com até 256 níveis distintos de transparência. Dois tipos de configurações têm sido freqüentemente adotadas na utilização do *color-buffer*: *single-buffering* ou *double-buffering*. Na configuração de *single-buffering*, o *color-buffer* onde está sendo feito o rendering da imagem é o mesmo que está sendo exibido na tela. Isso pode resultar em um efeito chamado de *flickering*, que ocorre quando o conteúdo do *color-buffer* é atualizado ao mesmo tempo em que este está sendo exibido na tela. O efeito pode ser visualmente percebido como uma discontinuidade em alguns pontos do quadro atualmente exibido na

tela. Uma maneira de solucionar esse problema é a utilização do *double-buffering*, onde o *color-buffer* a ser exibido na tela (*front-buffer*) é isolado do *color-buffer* onde está sendo feita a atualização de conteúdo (*back-buffer*).

O algoritmo de *z-buffer* (CATMULL, 1974), devido a sua simplicidade e generalidade, tem sido o método de oclusão preferido para implementação em *hardware*. Todas as placas gráficas atuais que implementam oclusão utilizam esse algoritmo, e por conta disso, contam com a presença de um *depth-buffer*. O *depth-buffer* encontrado nos aceleradores gráficos usualmente armazena valores de profundidade normalizados, que se situam no intervalo $[0.0, 1.0]$. Internamente os valores de profundidade das células do *depth-buffer* são representados em notação de ponto-fixo, e em geral com uma precisão de 24 bits. Os modelos mais atuais de placas gráficas têm contado com *depth-buffers* hierárquicos (GREENE; KASS; MILLER, 1993), (ATI, 2005), permitindo um aumento de performance dos testes de oclusão.

O *stencil-buffer* é um *buffer* de propósitos gerais e apresenta a mesma resolução espacial do *color-buffer*. Cada um dos seus elementos está relacionado a um elemento do *color-buffer*, e podem ser representados por um conjunto de 1 até 8 bits. *Stencil-buffers*, cujos elementos são representados por apenas 1 bit, são utilizados em procedimentos que necessitam mascarar regiões do *color-buffer*. Um exemplo desse tipo de aplicação é a simulação de superfícies reflexivas (DIEFENBACH, 1996). *Stencil-buffers* que contam com mais bits por elemento podem ser utilizados para os mais diversos fins. Implementações do algoritmo de *Shadow Volumes*, que tiram proveito dos recursos fornecidos pelo *hardware*, utilizam *stencil-buffers* de 8 bits para calcularem se o ponto da imagem se encontra iluminado ou em sombra através de um procedimento de contagem (HEIDMANN, 1991).

Efeitos de *motion blur*, profundidade de campo ou *antialiasing* podem ser obtidos através do uso de *buffers* de acumulação (HAEBERLI; AKELEY, 1990). Esses *buffers* são utilizados para acumular os resultados de diversas passadas[1], precisando desta forma de uma faixa de representação maior para evitar perdas. Essa maior faixa de representação é obtida através da utilização de um número maior de bits em seus elementos.

Um outro tipo de *buffer* suportado pelo *hardware* gráfico atual é o *pixel-buffer*, chamado também de *pbuffer*. Os *pbuffers* podem ser considerados como contextos de rendering independentes, não visíveis, capazes de armazenar uma variada gama de informações. Esses *buffers* são úteis no armazenamento dos passos intermediários do processo de rendering. A possibilidade de se gerar texturas dinâmicas de forma eficiente é relativamente nova. Antigamente essa geração era possível através do rendering da imagem para um *pbuffer*, leitura de seu conteúdo pela *CPU*, e posterior reenvio deste conteúdo, agora em forma de textura, de volta a unidade gráfica. A evolução do *hardware* permitiu que os *pbuffers* atuais possam ser interpretados como texturas, eliminando assim a necessidade de envio de seu conteúdo para a *CPU*. Pelo fato de poderem ser interpretados como texturas, os *pbuffers* se beneficiam de todos os esquemas de filtragem disponíveis às texturas.

[1]Neste contexto, "passada"trata de todo o processo envolvido no *rendering* de uma determinada cena. Esse processo inclui desde os ajustes das propriedades da câmera, reenvio das primitivas geométricas que compõem a cena e habilitação das respectivas unidades de textura, até o *rendering* da cena propriamente dito.

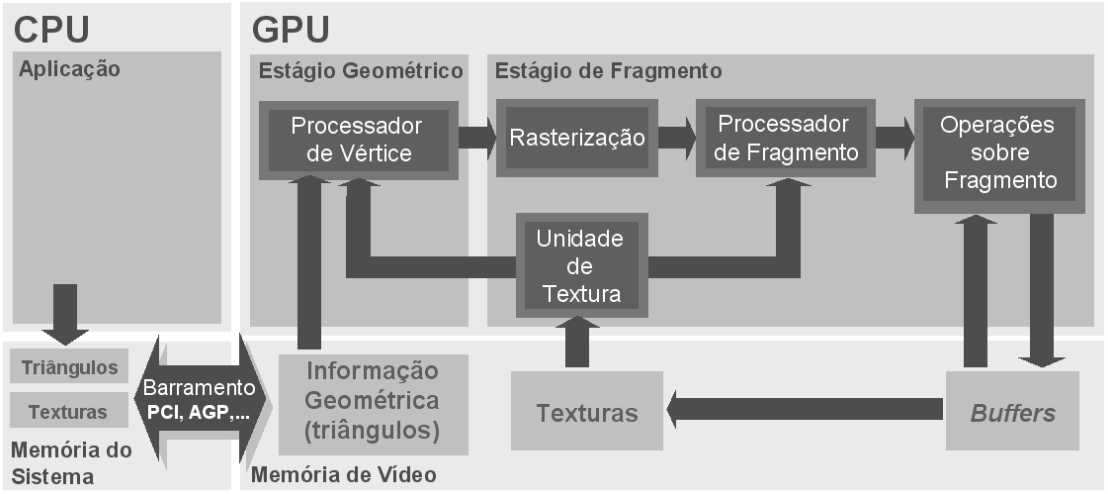


Figura 2.2: Organização interna da *GPU* e sua interface com a *CPU*. Adaptado de (FERNANDO et al., 2004)

A sua resolução espacial não fica restrita à resolução do *color-buffer*, como no caso do *stencil*. Os formatos dos elementos dos *pbuffers*, e o número de canais de cada elemento, são bastante variáveis e podem ser definidos pelo usuário no momento de sua criação.

Mais recentemente, devido à generalização das *GPU*s, os possíveis formatos das texturas, e por conseguinte dos *pbuffers*, foram expandidos. Atualmente cada elemento de um *pbuffer* pode contar com até 4 canais, cada um representado por um número de 32 bits em ponto flutuante, o que faz com que cada elemento desse *pbuffer* ocupe um total de 128 bits. Embora esse aumento de precisão faça com que alguns dos métodos de filtragem de textura não possam ser utilizados, uma faixa maior de números reais, não restritos ao intervalo $[0.0, 1.0]$, passa a ser representável. Essa maior precisão tem facilitado a utilização das *GPU*s em computações de propósito geral. Vários métodos específicos da computação gráfica também têm se beneficiando desse aumento de precisão, como por exemplo *volume rendering*, *blending*, entre outros.

2.4 Arquitetura

O *hardware* gráfico atual é estruturado na forma de um *pipeline*, ou seja, uma seqüência de estágios operando em paralelo e em uma ordem fixa. Cada estágio recebe como entrada a saída do estágio anterior e envia sua saída para o estágio seguinte. Desta forma são transformados os diversos vértices, primitivas geométricas e fragmentos que transitam em fluxo pelo *pipeline*. Essa organização do *hardware* é praticamente uma versão do *pipeline* gráfico visto na seção 2.1. A organização interna deste *hardware*, e sua interface com a *CPU*, podem ser vistos na figura 2.2.

2.4.1 Fluxo de funcionamento

Inicialmente a *CPU* envia para o processador de vértices da *GPU* um fluxo de vértices acompanhados de seus atributos. Entre esses atributos figuram suas coordenadas no espaço do objeto, cor, coordenadas de textura, vetor normal, etc. Nesse estágio alguns

dos atributos dos vértices são modificados através da aplicação de transformações geométricas. Estas operações geralmente incluem a transformação das coordenadas do vértice para o sistema de coordenadas da tela, transformações das coordenadas de textura, etc.

Os vértices transformados são então enviados para o próximo estágio, chamado de *Rasterização*. Baseando-se em informações que acompanham o conjunto de vértices, o estágio de rasterização reagrupa os vértices em primitivas geométricas. As primitivas possíveis de serem geradas são: triângulos, linhas e pontos. Eventualmente algumas primitivas serão recortadas contra o volume de visualização ou contra eventuais planos de recorte definidos pela aplicação. O procedimento de *backface culling*, caso esteja habilitado, também ocorre nesse estágio e resulta na eliminação de triângulos cujas faces frontais não estejam voltadas para a câmera. Os triângulos que passam por essa etapa são rasterizados gerando uma série de fragmentos. Cada fragmento possui uma série de atributos tais como posição na tela, cor, profundidade e uma ou mais coordenadas de textura associadas, atributos esses gerados a partir da interpolação dos atributos correspondentes dos vértices transformados. A utilização do nome fragmento no lugar de *pixel* é proposital, pois um fragmento não representa necessariamente um *pixel* na imagem final. Um fragmento é um candidato a *pixel*, e se tornará um *pixel* na imagem final somente depois de passar com sucesso pelo processador de fragmento e pelos diversos testes feitos no estágio seguinte, o estágio de operações sobre fragmentos.

No processador de fragmentos cada fragmento pode sofrer diversas operações que podem mudar sua aparência ou mesmo remover o fragmento do *pipeline*. Maiores detalhes acerca do processador de fragmento serão vistos a seguir.

O estágio de operações sobre fragmentos executa a seqüência final de operações a serem aplicadas aos fragmentos. Nesse estágio vários testes são executados: *depth test*, *scissor test*, *alpha* e *stencil*. Se algum destes testes falhar, o fragmento é descartado e não é enviado ao *frame-buffer*. Caso passe com sucesso pelos testes, uma operação de *blending* combinará a cor do fragmento com a cor do *pixel* correspondente do *frame-buffer*, resultando na atualização da cor do *pixel*. Neste caso o *depth-buffer* também pode ser atualizado.

Cabe aqui uma nota sobre o *depth test*: o teste apresentado anteriormente ocorre ao final do *pipeline*, imediatamente antes da atualização do *framebuffer*. Isso implica que os fragmentos, mesmo os que serão descartados futuramente pelo *depth test*, sejam processados pelo processador de fragmentos. Um método utilizado para aumentar a performance do rendering, eliminando o custo de processar fragmentos que serão eliminados pelo *depth test*, é o uso do teste precoce de z (NVIDIA, 2004c). O teste precoce de z não passa de um *depth test* feito precocemente, logo que o fragmento abandona o estágio de rasterização. Esse teste é feito com o valor de profundidade interpolado do fragmento. A utilização de triângulos transparentes, alteração do valor de profundidade do fragmento pelo processador de fragmentos, e a ordem do envio da geometria da cena para a *GPU* podem diminuir a eficácia desse teste no sentido de aumentar a performance da aplicação.

2.4.2 Programabilidade

Há não muito tempo atrás, os processadores de vértices e de fragmentos apresentavam funcionalidades fixas. Com o passar do tempo se tornaram configuráveis, e mais tarde foram substituídos por unidades verdadeiramente programáveis.

Essa evolução no sentido de uma maior programabilidade e generalização veio em resposta ao apelo da comunidade por processadores gráficos flexíveis o bastante, que permitissem a transferência completa do *pipeline* gráfico da *CPU* para a *GPU*. Outro fator que também impulsionou a evolução da arquitetura desses processadores foi a utilização, com sucesso, da *GPU* como um processador de uso geral. A seguir examinamos cada uma dessas duas unidades programáveis em detalhe.

2.4.3 Processador de Vértices

O processador de vértices é responsável por executar uma série de operações sobre o fluxo de vértices que por ele trafega. Cada vértice que chega ao processador vem acompanhado de seus atributos que serão carregados em registradores somente para leitura. Uma série de registradores de leitura e escrita também estão disponíveis no processador, como forma de facilitar o armazenamento de resultados intermediários necessários às transformações dos vértices. Registradores somente para escrita são responsáveis por armazenar os resultados das transformações dos vértices e de seus atributos. Alguns dos valores desses registradores somente para escrita, como os dos registradores de cor ou coordenadas de textura, serão interpolados durante a rasterização e então enviados ao processador de fragmentos, compondo dessa forma os atributos dos fragmentos. As operações a serem feitas sobre os vértices que entram no processador estão previamente definidas no programa de vértices, também chamado de *vertex shader*.

As operações que podem estar definidas nesses programas são as mais variadas. A operação mais comum é a transformação do vértice do sistema de coordenadas do universo para o sistema de coordenadas da câmera. Outras transformações também podem ser feitas nesse estágio. As coordenadas dos vértices podem ser alteradas de forma a criarem efeitos de deformação, como ocorre com a técnica de *skinning* (NVIDIA, 1999).

O processador de vértices opera sobre um fluxo de vértices, um vértice por vez, em seqüência. Um vértice entra, é transformado, e então enviado para o próximo estágio. Nesse processo vértices não podem ser criados ou destruídos. Um vértice na entrada sempre corresponderá a um vértice na saída.

Os processadores de vértice são atualmente implementados através de uma arquitetura *SIMD*, significando que existem na realidade diversas unidades de processadores de vértices operando em paralelo. Todas essas unidades executam o mesmo programa de vértice simultaneamente. Devido a essa organização, esses processadores apresentavam, até recentemente, uma série de limitações quanto à sua programabilidade. O tamanho dos programas de vértice era limitado em cerca de 1024 instruções, e o processador não tinha acesso à memória. Atualmente essa limitação de tamanho do programa praticamente inexiste, e os processadores já podem acessar texturas. O acesso à textura permite a implementação de uma série de técnicas, tais como *displacement mapping*, simulação de

água, explosões, etc (NVIDIA, 2004a). Até recentemente os processadores de vértices também não possuíam instruções de salto condicional, somente instruções de atribuição condicional. Isso permitia, em um alto nível, a simulação de instruções condicionais, mas resultava, no baixo nível, em um código que precisava avaliar todos os blocos de código referentes à instrução condicional simulada. Recentemente a NVIDIA lançou, com as *GPUs* da série 6 (NVIDIA, 2004b), o "*true branching*", ou seja, instruções condicionais verdadeiras, uma exigência da última versão do padrão *DirectX* (*Microsoft DirectX 9.0, Vertex Shader 3.0*). Isso permite um aumento de eficiência na execução do programa de vértice já que não é mais necessário arcar com o custo de execução de instruções que não contribuirão para o resultado final.

2.4.4 Processador de Fragmentos

O processador de fragmentos opera sobre o fluxo de fragmentos que lhe é passado pelo estágio de rasterização. Assim como no caso do processador de vértices, o processador de fragmentos não é um processador único. Trata-se na realidade de um conjunto de processadores paralelos operando sob o modelo *SIMD*. Também, à semelhança do processador de vértices, as operações a serem aplicadas sobre os fragmentos estão descritas em um programa, neste caso chamado programa de fragmento, ou *fragment shader*.

Vale aqui uma observação em relação à velocidade relativa dos processadores de vértices e de fragmentos. Desconsiderando-se os casos degenerados, que não são a regra, o número médio de fragmentos gerados por um polígono é maior do que o número de vértices desse polígono. Isso significa que existe um aumento no número de elementos do fluxo que atravessa o *pipeline* logo após o estágio de rasterização, e que precisa ser absorvido pelos estágios seguintes. Desta forma o processador de fragmentos é construído para operar numa freqüência maior que a do processador de vértices.

Os elementos de saída de um processador de fragmentos, os fragmentos transformados, passarão ainda pelo estágio de operações sobre fragmentos, estando sujeitos a todos os seus testes. Se passarem com sucesso por esse último estágio, serão enviados para o *buffer* final. O *pipeline* pode ser configurado de forma a direcionar o resultado do rendering para diversos *buffers* possíveis, como por exemplo o *color-buffer*, o *depth-buffer*, os *pbuffers*, etc.

A característica mais importante em relação aos fragmentos é que estes se encontram no sistema de coordenadas da tela. Isso significa que o fragmento já está associado à posição do *buffer* que irá eventualmente atualizar. Desta forma, embora um programa de fragmento possa amostrar diferentes texturas (*gathering*), não é possível alterar a posição aonde um fragmento será escrito (*scattering*).

Algumas das restrições existentes no modelo de programação dos processadores de vértices também existiam ou ainda existem nos processadores de fragmentos. O tamanho dos programas de fragmento possui atualmente um limite máximo, girando em torno de 512-1024 instruções (dependendo da *API*). Recentemente instruções verdadeiramente condicionais passaram a estar disponíveis também para esses processadores. Diferentemente dos processadores de vértices, os processadores de fragmentos sempre tiveram acesso à memória, pois o processo de amostragem de textura é feito exatamente nesse

estágio.

2.5 Linguagens de *shading*

A partir da observação de que um único modelo de *shading* não seria suficiente para representar os diferentes tipos de superfícies presentes em uma cena sintética, Rob Cook criou as *Shade Trees* (COOK, 1984). Uma *Shade Tree* organiza de forma hierárquica as várias operações de *shading* necessárias à correta representação de um determinado tipo de superfície. As folhas das *Shade Trees* armazenam as propriedades da superfície enquanto seus nós internos contém as operações de *shading*. A definição da cor final da superfície é obtida através de um percorrimento recursivo desta árvore.

Enquanto bastante flexíveis, no sentido de poderem representar diversos tipos distintos de superfícies, as *Shade Trees* tornavam-se muito grandes e difíceis de manipular quando utilizadas na modelagem de superfícies complexas. Desta forma, com o objetivo de tornar mais flexível e simples a definição de propriedades de superfícies sintéticas, é que a *Pixar* cunhou a expressão *shading language* através da criação da *RenderMan Shading Language* (*RMSL*)(UPSTILL, 1990). A *RMSL* é atualmente a linguagem de *shading* para rendering *offline* mais conhecida e utilizada.

O fato de ter sido desenvolvida para sistemas de *rendering offline* inviabilizou o uso da *RMSL* em aplicações de tempo real. Pesquisadores da Universidade da Carolina do Norte, em Chapel Hill (*UNC-Chapel Hill*), desenvolveram pesquisas com *hardware* gráfico na década de 90 que culminaram com a criação da *PixelFlow* (OLANO, 1998): uma máquina com a arquitetura voltada exclusivamente à implementação de aplicações gráficas de tempo real, programável através de uma linguagem de *shading*. Marc Olano desenvolveu a linguagem de *shading* chamada *pfman* para uso exclusivo com a *PixelFlow*. Embora tenham conseguido a geração de imagens a taxas realmente interativas (cerca de 30 *frames* por segundo), a *PixelFlow* era muito cara, e falhou comercialmente.

No final dos anos 90 o *hardware* gráfico disponibilizado comercialmente por alguns fabricantes já contava com implementações do *OpenGL* (OpenGl Specification, 2004). Isso inspirou a pesquisa e criação de linguagens de *shading* que, quando compiladas, geravam código *OpenGL* capaz de gerar os efeitos visuais desejados através de renderings de múltiplas passadas. Um exemplo dessas iniciativas foi a *Real Time Shading Language* (*RTSL*), desenvolvida na Universidade de Stanford (PROUDFOOT et al., 2001).

A percepção de que o mercado estaria se movendo na direção da utilização de linguagens de *shading* fez com que fabricantes e organizações começassem um trabalho concentrado no sentido de definir linguagens e *hardware* que interoperassem. Os principais fabricantes de *hardware* criaram suas próprias linguagens de *shading* e o seu próprio *hardware* programável, enquanto que empresas de *software* criavam também suas linguagens de *shading*, muitas vezes contando com características ainda não suportadas pelo *hardware*. O resultado desse duelo têm sido uma evolução acelerada tanto da arquitetura dessas máquinas, quanto das suas linguagens de programação, aonde um ponto de convergência já começa a ser vislumbrado.

As principais linguagens de *shading* atuais, destinadas à programação de *hardware* dedicado, são o *HLSL* (*High Level Shading Language*) da *Microsoft*, o *Cg* (*C for Graphics*), desenvolvido em conjunto pela NVIDIA e Microsoft, e o *GLSL* (*OpenGL Shading Language*) da *3Dlabs*. O *HLSL* foi um projeto desenvolvido pela Microsoft em parceria com a NVIDIA e atualmente faz parte da API *DirectX* da Microsoft. O *Cg* é um iniciativa exclusiva da NVIDIA. Tendo sido criado a partir do *HLSL*, o *Cg* possui sintaxe e estrutura muito semelhante a este. Diferentemente do *HLSL*, o *Cg* foi desenvolvido tendo em mente a portabilidade, ou seja, um programa escrito em *Cg* pode ser portado para qualquer outra plataforma de *hardware*, bem como para qualquer outro sistema operacional. O *GLSL* é uma proposta, feita pela 3Dlabs, de extensão à *API OpenGL*. Assim como o *Cg*, também destina-se a ser portável entre diversas plataformas de *hardware* e *software*.

2.6 Textura: *cache* e *prefetching*

O aumento de performance dos processadores, em termos de milhões de operações em ponto flutuante por segundo, tem crescido de forma exponencial por vários anos. Entretanto, a latência e a largura de banda no acesso à memória não tem acompanhado esse ritmo. Adicionalmente, no contexto do *hardware* gráfico, a tendência tem sido o uso de um número cada vez maior de texturas por primitiva. Esse aumento no número de acessos à textura tem se tornado um dos principais gargalos dos sistemas gráficos, fazendo com que um cuidado especial seja dedicado à implementação dos subsistemas responsáveis pela realização de acesso à texturas (MOLLER; HAINES, 2002).

Embora existam diferenças entre as arquiteturas dos diversos fabricantes, variações de técnicas tradicionais, adaptadas à realidade da computação gráfica, têm sido utilizadas na minimização dos problemas de latência e largura de banda. Problemas relacionados a largura de banda têm sido tradicionalmente resolvidos através do uso de *caches*. *Caches* são memórias rápidas que se localizam no próprio *chip* da *GPU*, e são responsáveis por armazenar os elementos de textura recentemente acessados. Dessa forma, elementos acessados com mais freqüência ficam mantidos no *cache*, reduzindo o número de acessos à memória. Esquemas mais sofisticados de *cache*, considerando níveis hierárquicos, podem ser também implementados como forma de aumentar o desempenho do sistema.

A latência está relacionada ao tempo que a memória leva para retornar o valor de um *texel* depois de feita uma solicitação. À medida que os fragmentos vão executando operações de *fetch* de textura, eles são colocados em filas de espera, e saem de lá somente depois que a memória retorna o valor solicitado. Essas solicitações podem eventualmente ocorrer numa freqüência superior à capacidade de resposta da memória, causando uma parada em todo o *pipeline*. Uma maneira de se reduzir a latência é através da previsão do próximo valor de memória a ser acessado. De acordo com a natureza e comportamento do sistema, tenta-se prever qual o próximo valor de memória a ser acessado, e antes da solicitação efetivamente ocorrer, resgata-se esse valor deixando-o disponível na *cache*.

3 ALGORITMOS PARA GERAÇÃO DE SOMBRAS

Sombras são elementos muito importantes que auxiliam na percepção do mundo ao nosso redor. Muitos cientistas têm desenvolvido estudos sobre as sombras, indo da análise de sua geometria até o seu impacto na percepção humana (WANGER, 1992) (MAMASSIAN; KNILL; KERSTEN, 1998) (HUBONA et al., 1999). As sombras auxiliam no entendimento das relações espaciais entre objetos de uma cena. Dependendo de suas configurações, podem fornecer informações sobre a geometria do objeto que recebe a projeção da sombra, bem como do objeto que está gerando a sombra. Além disso, a presença de sombras aumenta o realismo de cenas sintéticas, tornando-as mais convincentes.

3.1 Modelos de iluminação

Do momento em que acordamos até a hora de dormir passamos o dia vendo coisas: nossa própria imagem em um espelho, um programa de TV, livros, outras pessoas, etc. Podemos ver todas essas coisas sem que exista algo que nos conecte diretamente a elas. O elo visual com o mundo que nos cerca se dá apenas através da luz que parte desses objetos e atinge nossa retina.

A luz pode ser repassada ao observador de duas maneiras: direta ou indiretamente. A luz é passada diretamente ao observador quando o atinge sem que tenha havido nenhum tipo de interação desta com outros objetos durante o seu trajeto. A luz também pode atingir o observador de forma indireta, após interagir com um ou mais objetos durante o seu trajeto. A interação da luz em um certo ambiente pode assumir configurações complexas, fazendo com que se espalhe, de maneira não uniforme, por todo o ambiente. Visto que um dos elementos chave para a obtenção do realismo na computação gráfica é a correta representação das cores de um objeto em uma cena (HALL, 1988), a precisa simulação do fenômeno físico referente ao comportamento da luz torna-se crucial na síntese de imagens.

Na tentativa de aproximar cada vez mais o fenômeno físico representado pela luz, uma série de modelos de iluminação foram desenvolvidos. Alguns modelos são basicamente empíricos, pois apenas aproximam o efeito visual de iluminação tais como os de Warnock (WARNOCK, 1969), Gouraud (GOURAUD, 1971) e Phong (PHONG, 1975), entre outros. Outros modelos são ditos analíticos, pois modelam a interação da energia luminosa em um ambiente, como por exemplo em Cook e Torrance (COOK; TORRANCE, 1982), Goral (GORAL et al., 1984) e Wallace (COHEN; WALLACE, 1993).

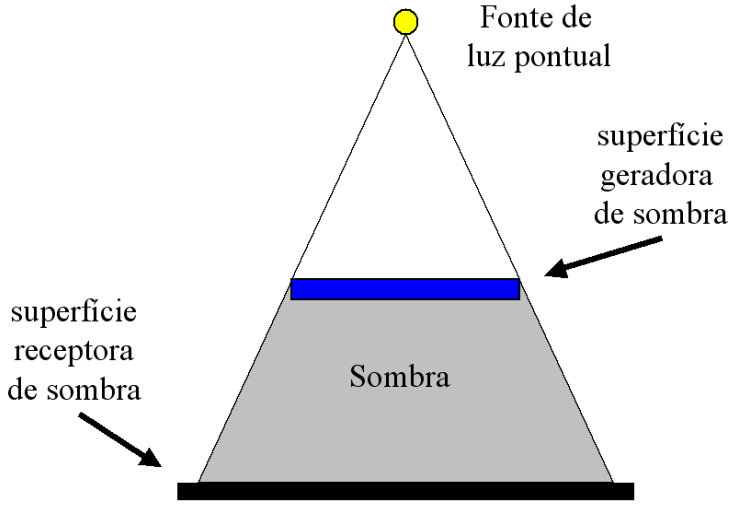


Figura 3.1: Modelo utilizado na representação de luz pontual. A determinação das regiões que estão em sombra se reduz a uma questão de visibilidade. É considerado como estando em sombra tudo o que não é visível diretamente a partir do ponto que representa a fonte de luz.

Os modelos analíticos, por modelarem a interação da luz em um ambiente, indiretamente calculam as suas sombras (regiões onde há menor concentração de energia). Os resultados visuais proporcionados por esses modelos são muito próximos dos obtidos em uma cena real, mas o custo computacional envolvido em suas resoluções os tornam inadequados quando o objetivo é a geração de cenas a taxas interativas. Por outro lado, os resultados obtidos através de modelos empíricos de iluminação nem sempre aproximam de maneira precisa o que ocorre no mundo real, mas, por serem modelos simplificados e explorarem coerências existentes nas cenas, tendem a ser mais eficientes, sendo a escolha preferida para aplicações que demandam interatividade.

3.2 Uma abordagem mais prática e eficiente

Em detrimento de uma maior eficiência, modelos empíricos de iluminação geralmente desconsideram a interação luminosa entre as superfícies de um ambiente. Esses algoritmos, em sua maioria, limitam-se a avaliar, para uma fonte de luz pontual, um ponto de uma superfície como iluminado quando é diretamente visível a partir da fonte de luz, ou em sombra, caso contrário. A questão fica, deste modo, reduzida a um problema de visibilidade. Esse modelo pode ser observado na figura 3.1.

Partindo deste modelo de luz pontual, uma série de algoritmos de sombra foram desenvolvidos. Até o fim dos anos 70, os algoritmos para geração de sombras eram restritos a cenas compostas por polígonos. Os cálculos de sombra eram feitos no espaço geométrico. Assim funcionavam os algoritmos de Blinn (BLINN, 1988), Crow (CROW, 1977), entre outros. Em 1978 Lance Williams apresentou um novo algoritmo para o cálculo de sombras geradas por uma fonte de luz pontual. Diferentemente dos algoritmos existentes até então, o algoritmo de Williams funciona no espaço da imagem. Inicialmente a intenção era resolver o problema da geração de sombras em cenas compostas por superfícies curvas, tais como superfícies bi-cúbicas. O algoritmo mostrou-se bastante geral, eficiente

e robusto, e acabou servindo de base para uma série de outras soluções, que passaram a compor uma nova classe de algoritmos de sombra, baseados no domínio da imagem. Mais tarde o algoritmo de Williams ficou popularmente conhecido pelo nome de algoritmo de *Shadow Mapping* (WILLIAMS, 1978).

Apesar de todas as qualidades, o algoritmo de *Shadow Mapping* apresenta alguns problemas, sendo o mais evidente deles o forte *aliasing* presente nas bordas de sombras produzidas pelo algoritmo. Na medida em que o presente trabalho trata deste problema, uma breve explanação sobre algoritmo de *Shadow Mapping*, bem como das técnicas que tentam minimizar os desagradáveis efeitos do *aliasing*, torna-se necessária. Inicialmente será apresentado o algoritmo de *Shadow Mapping*. Na próxima seção é apresentado o algoritmo de *PCF*, considerado o primeiro algoritmo para redução do *aliasing* do *Shadow Mapping*, e que também é tema deste trabalho. Na última seção deste capítulo são apresentados algoritmos que representam o estado da arte no que se refere à redução do *aliasing* gerado por *Shadow Mapping*.

3.3 O algoritmo de *Shadow Mapping*

No final dos anos 70, o algoritmo de visibilidade proposto por Catmull, o *z-Buffer* (CATMULL, 1974), era o único algoritmo de visibilidade capaz de permitir a correta visualização de superfícies compostas por superfícies bi-cúbicas. Apesar do consumo adicional de memória causado pelo uso de um *depth-buffer*, o algoritmo de Catmull apresentava características interessantes, tais como crescimento linear de custo em decorrência do aumento da complexidade da cena e capacidade de ser aplicado a cenas complexas.

Por esta época também, Williams observou que o cálculo da sombra (sua determinação) poderia ser visto como um problema de visibilidade, considerado a partir do ponto de vista da fonte de luz, podendo ser resolvido através do emprego de *depth-buffers*. Desta forma, Williams estruturou o algoritmo de *Shadow Mapping* em dois passos:

Passo 1: Uma imagem contendo apenas o *depth-buffer* da cena é gerada a partir do ponto de vista da luz. Nenhum cálculo relativo a *shading* é feito neste momento. Esta imagem é denominada *Shadow Map*. Este passo pode ser observado na figura 3.2.

Passo 2: A imagem final da cena é gerada a partir do ponto de vista da câmera. Nesta etapa cada ponto gerado pela cena é transformado para o sistema de coordenadas da luz, e testado contra o *Shadow Map* a fim de avaliar sua visibilidade em relação a fonte de luz (teste de sombra). Se o ponto não é visível a partir do ponto de vista da luz, este é considerado como estando em sombra. Caso contrário, o ponto é considerado iluminado. O resultado deste teste é então usado para atenuar a cor final do *pixel*, calculada de acordo com as propriedades da superfície e o modelo de iluminação utilizado. Este passo é ilustrado na figura 3.3.

Apesar de bastante eficiente e robusto, o algoritmo de *Shadow Mapping* apresenta certas exigências e limitações que devem ser consideradas no momento de sua utilização. Além do fato da estrutura do *Shadow Map* exigir um espaço extra de memória, a sua geração sofre de problemas relacionados à quantização e amostragem.

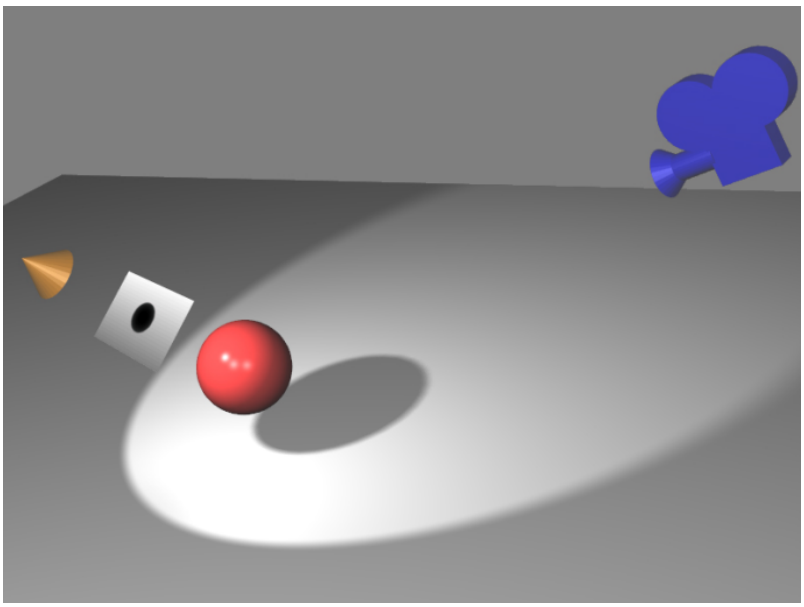


Figura 3.2: Primeiro passo do algoritmo de *Shadow Mapping*: geração do *Shadow Map* a partir do ponto de vista da luz.

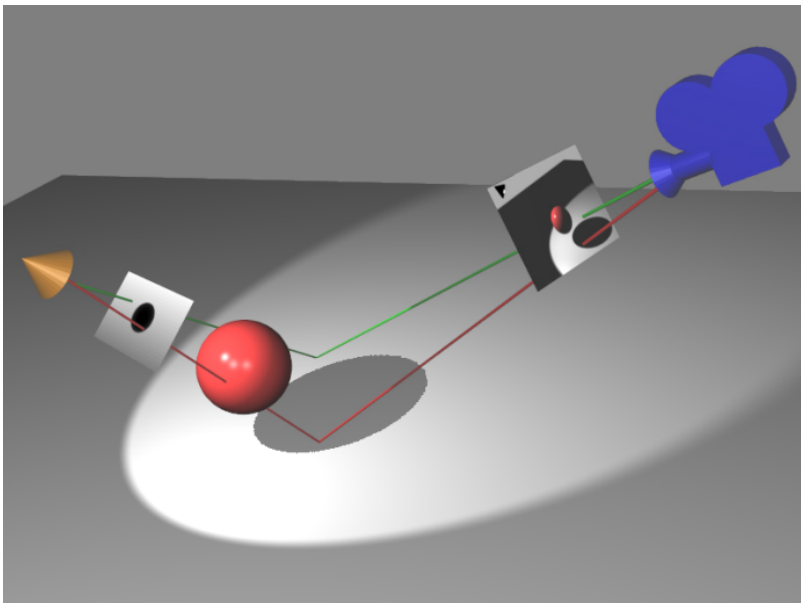


Figura 3.3: Segundo passo do algoritmo de *Shadow Mapping*: geração da imagem final a partir do ponto de vista da câmera. Cada ponto gerado pela câmera é testado contra o *Shadow Map* para a determinação da sombra.

O primeiro problema refere-se à quantização dos valores de profundidade a serem registrados nas células do *Shadow Map*, e está vinculado diretamente à precisão numérica disponível para tal. Um *pixel* da cena final que está iluminado é visível a partir do ponto de vista da luz, e a porção da superfície da qual faz parte está registrada em alguma célula do *Shadow Map*. Durante a geração da cena final esse *pixel* será transformado para o sistema de coordenadas da luz. Essa transformação entre sistemas de coordenadas implica na execução de uma série de operações aritméticas que, em conjunto com a imprecisão numérica inerente aos sistemas computacionais, pode introduzir pequenos erros nos valores de suas coordenadas. A introdução desses erros pode eventualmente fazer com que o *pixel*, depois de transformado, situe-se um pouco abaixo da superfície a qual pertence, sendo avaliado incorretamente como estando em sombra. Os efeitos deste auto-sombreamento são percebidos visualmente como pontos escuros na imagem final.

A solução proposta por Williams para a redução dos efeitos do auto-sombreamento foi a subtração de um pequeno valor de *bias* das coordenadas z dos *pixels* (já no espaço da luz). O valor deste *bias* é determinado de forma empírica, é igual para todos os *pixels* da imagem final, e dependente de uma série de fatores. Alguns dos fatores que podem interferir na definição do valor do *bias* são a configuração da cena, precisão numérica dos elementos do *Shadow Map*, entre outros. A utilização de valores de *bias* demasiadamente grandes pode causar um perceptível deslocamento da sombra na cena. O aumento da precisão numérica dos elementos do *Shadow Map* geralmente permite a utilização de valores de *bias* menores.

O segundo problema na geração do *Shadow Map* refere-se a própria discretização da cena, estando relacionado ao tipo de amostragem feita e a resolução espacial do *Shadow Map*. A resolução pode em alguns casos não ser suficiente para o registro de determinadas regiões da cena, resultando em perda de informação. A regularidade com que é feita a amostragem da cena também pode causar alguns efeitos indesejáveis, tais como *aliasing*. Soluções a estes problemas seriam, respectivamente, o aumento da resolução do *Shadow Map*, com um conseqüente aumento no consumo de memória, e a amostragem estocástica da cena, que resultaria em um aumento do custo computacional.

Um terceiro problema está relacionado à amostragem do *Shadow Map* no momento da realização dos testes de sombra. Esse problema pode ser observado na imagem final através de serrilhados nas bordas das sombras. Embora o aumento da resolução do *Shadow Map* ajude a minimizar esse efeito, na maioria das vezes não é possível se prever qual a resolução necessária para a geração de sombras sem serrilhado nas bordas. Este problema torna-se mais evidente no caso de cenas dinâmicas que contam com o movimento de objetos, fontes de luz ou de câmeras. A figura 3.4 apresenta o detalhe de uma imagem onde pode ser observada a presença de *aliasing* na borda da sombra gerada através de *Shadow Mapping*.

Problemas relacionados a *aliasing* são tradicionais em procedimentos onde existe a necessidade de se amostrar imagens, e geralmente são tratados através da utilização de técnicas de filtragem. No caso do *Shadow Map*, a técnica de filtragem tradicionalmente utilizada no momento da amostragem é o algoritmo de *PCF*, que será discutido em detalhe na seção 3.4.

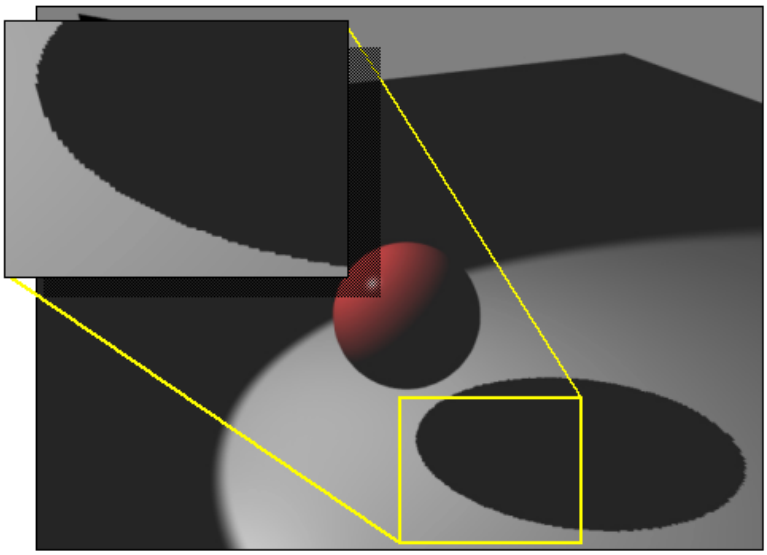


Figura 3.4: Detalhe da imagem: presença de *aliasing* na borda da sombra gerada através de *Shadow Mapping*.

3.3.1 Implementação em *hardware*

O *Shadow Mapping* é um dos algoritmos que melhor se adapta a implementação em *hardware*. Tanto seu funcionamento quanto as estruturas das quais se utiliza são facilmente mapeados para o *hardware* gráfico atual. A geração do *Shadow Map* pode ser feita através do processo de rasterização convencional implementado nas *GPU*s e cujo armazenamento pode ser feito em uma textura. Os acessos ao *Shadow Map* podem ser feitos através do mesmo esquema empregado no acesso a uma textura convencional. O *Shadow Mapping* não precisa de complexas estruturas de dados e de manipulações geométricas feitas pela *CPU*, sendo capaz de aproveitar o paralelismo fornecido pela *GPU*. Uma outra vantagem, um pouco mais sutil, surge do fato do *Shadow Map* ser gerado pela própria *GPU*. Isso significa que eventuais alterações na geometria da cena, causadas pela unidade de processamento de vértices, serão devidamente capturadas pelo *Shadow Map*.

A primeira implementação de *Shadow Mapping* utilizando recursos disponíveis em *hardware* foi proposta por Segal em 1992 (SEGAL et al., 1992). Ele observou que a obtenção dos valores do *Shadow Map*, para utilização nos testes de sombra, poderia se beneficiar do mapeamento projetivo de textura, recurso já disponível no *hardware* de algumas máquinas da época. O algoritmo de Segal baseava-se em múltiplas passadas. Na primeira, um *depth-buffer* é gerado a partir do ponto de vista da luz e convertido em uma textura (*Shadow Map*). Em uma segunda passada a cena final é gerada a partir do ponto de vista da câmera, apenas com a iluminação ambiente. Na terceira passada os pontos da cena final são transformados para o espaço da fonte de luz e projetivamente testados contra o *Shadow Map*. Os *pixels* têm então seus canais alfa marcados de acordo com o resultado dos testes de sombra: 1 caso o *pixel* esteja iluminado, ou 0 caso o *pixel* esteja em sombra. No quarto e último passo a imagem final é renderizada novamente utilizando o modelo completo de iluminação apenas para os pontos iluminados.

Em sua tese de doutorado Wolfgang Heidrich (HEIDRICH, 1999) usou uma implementação semelhante à de Segal, adaptada de forma a poder tirar proveito do *hardware* encontrado em algumas placas gráficas da época. Sua implementação utiliza a geração automática de coordenadas de textura, em conjunto com o canal alfa dos *pixels*, de forma a gerar máscaras de sombra para os *pixels* da cena final.

Em 2001 a NVIDIA lançou o *chip NV20* (encontrado na placa *GeForce3*), que possui *hardware* dedicado à geração de *Shadow Maps* (EVERITT, 2001b). A nova placa estende a unidade de mapeamento de texturas de forma a suportar um formato capaz de armazenar componentes de profundidade. Acessos feitos à essa nova textura não retornam os valores de profundidade lá armazenados, mas sim os resultados dos testes de sombra: iluminado ou em sombra. O *hardware* também oferece a opção de se filtrar os valores desta amostragem através de um filtro *PCF* de 4 amostras, com posterior interpolação bilinear desses resultados, fazendo com que o resultado final da amostragem não fique restrito ao conjunto de valores $\{0, 1\}$, mas dentro do intervalo $[0.0, 1.0]$ (FERNANDO, 2004).

A evolução do *hardware* gráfico tem feito com que recursos dedicados, tais como o de *Shadow Mapping*, sejam descontinuados em prol de uma arquitetura mais geral, capaz de permitir a implementação de uma gama maior de algoritmos. Atualmente texturas com uma maior capacidade de representação permitem que *Shadow Maps* sejam implementados com uma precisão de até 32 bits por elemento, com representação em ponto flutuante. Programas de vértice e de fragmento permitem ainda a utilização de diferentes técnicas de amostragem no momento da filtragem do *Shadow Map*.

3.4 Filtrando sombras geradas com *Shadow Mapping*: *PCF*

O algoritmo de *PCF* foi criado por William T. Reeves e colaboradores (REEVES; SALESIN; COOK, 1987), e teve como objetivo amenizar o efeito de serrilhado presente nas bordas das sombras geradas pelo algoritmo de *Shadow Mapping*. Como dito pelo próprio Reeves em seu artigo: *"A singular versatilidade, eficiência e simplicidade do algoritmo de z-buffer fazem com que torne-se tentadora a busca por soluções aos seus problemas."*.

O acesso à textura geralmente é acompanhado de mecanismos de filtragem. Uma região da textura é amostrada e então processada pelo filtro. No caso da amostragem do *Shadow Map*, esse mecanismo mostra-se inapropriado, pois o valor de profundidade filtrado seria utilizado no teste de sombra, significando que o resultado do teste continuaria sendo binário, não ocorrendo nenhum tipo de suavização das bordas das sombras. Um outro efeito indesejável seria a deformação das sombras. O procedimento de filtragem alteraria os valores de profundidade originais do *Shadow Map*, fazendo com que as sombras perdessem a relação com a geometria da cena.

Para permitir que as sombras da cena possam ser devidamente filtradas, o algoritmo de *PCF* inverte a ordem em que os testes de sombra e a filtragem da textura são feitos. Desta forma, os testes de sombra são feitos primeiro, e o filtro é aplicado posteriormente sobre os resultados binários desses testes. Este procedimento resulta na correta suavização das bordas das sombras.

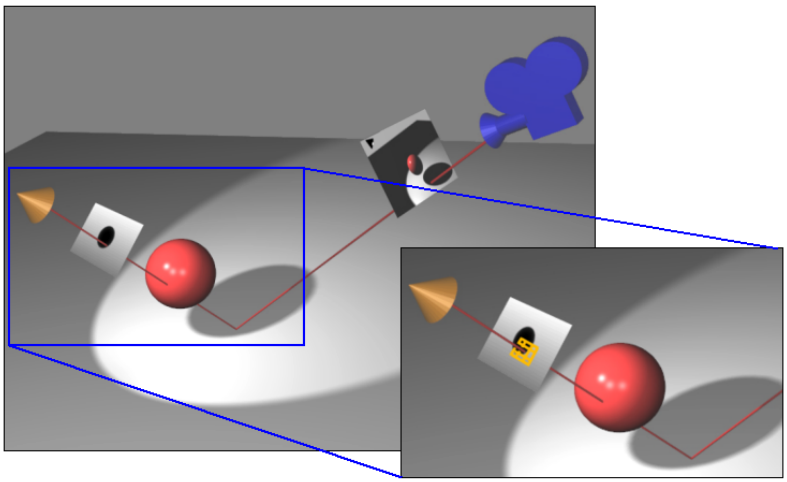


Figura 3.5: Projeção de um ponto da cena sobre o *Shadow Map*, e destacado sobre este, a área utilizada na filtragem com o *PCF*.

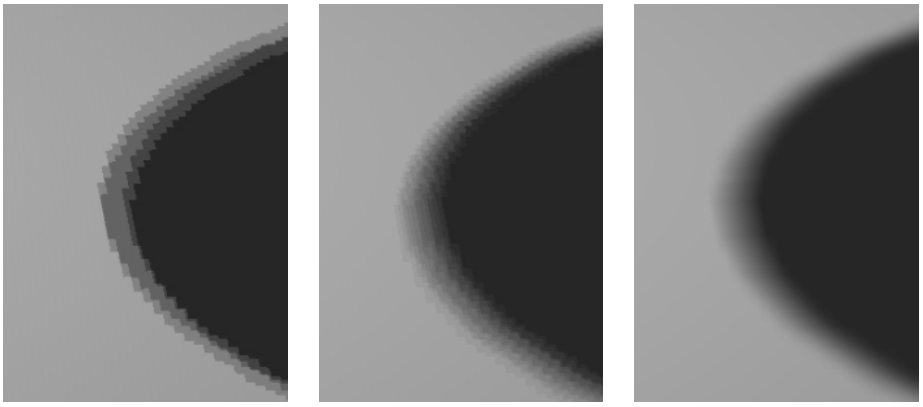
O procedimento para gerar uma imagem com sombras suavizadas pelo *PCF* divide-se em dois passos, sendo muito semelhante ao *Shadow Mapping*:

Passo 1: O *Shadow Map* é gerado de maneira tradicional. No caso de existirem várias fontes de luz, um *Shadow Map* deve ser gerado para cada uma.

Passo 2: A imagem é gerada a partir do ponto de vista da câmera. Cada *pixel* da imagem final define uma área sobre as superfícies da cena. Essa área é transformada para o sistema de coordenadas da luz, definindo uma região sobre o *Shadow Map*. A função do *PCF* é determinar a porcentagem da área desta região que gera sombra sobre a região correspondente da cena. Esta porcentagem geralmente é aproximada através de testes de sombra feitos com amostras obtidas a partir do *Shadow Map* e seus pontos correspondentes na cena. O conjunto dos resultados desses testes de sombra será então filtrado, e o resultado usado na determinação da atenuação final do *pixel*. A figura 3.5 ilustra a projeção de um ponto da cena sobre o *Shadow Map*, e destaca (sobre o *Shadow Map*) a área amostrada para a determinação da atenuação final do *pixel*.

Uma série de fatores pode interferir na qualidade da filtragem obtida com o uso do *PCF*: o formato da região de amostragem sobre o *Shadow Map*, a distribuição das amostras sobre esta região e a quantidade de amostras. Diferentes políticas podem ser usadas na determinação da região a ser amostrada no *Shadow Map* e na maneira como ela será amostrada. As regiões de amostragem podem ser determinadas através da exata projeção da área do *pixel* da imagem final sobre o *Shadow Map*. Isso acarretaria na projeção dos 4 cantos da área do *pixel* sobre a geometria da cena, e a reprojeção desse novo polígono sobre o *Shadow Map*. Esses cálculos, se feitos para todos os *pixels* da tela, podem acarretar em um custo muito elevado, diminuindo significativamente a performance do algoritmo. Uma outra alternativa, menos onerosa, seria a aproximação desta área através de *bounding boxes*. Uma terceira possibilidade seria a utilização de uma região de amostragem de formato fixo, e igual para todos os pontos gerados pela cena.

Para cada um desses tipos de regiões as distribuições das amostras podem assumir di-



(a) (b) (c)

Figura 3.6: Comparação entre os resultados da filtragem de uma sombra gerada com *Shadow Mapping* através da utilização de filtros *PCF* com diferentes números de amostras: (a) *PCF* com 4 amostras. (b) *PCF* com 9 amostras. (c) *PCF* com 16 amostras.

versas configurações: regular, gaussiana ou regular com pequenos deslocamentos. Além disso, o número de amostras também influencia os resultados finais da filtragem feita com o *PCF*. Um número maior de amostras tende a gerar transições mais suaves entre regiões iluminadas e em sombra. Um exemplo do efeito que o número de amostras tem sobre o resultado da filtragem pode ser observado na figura 3.6.

Um efeito interessante obtido através do aumento exagerado da área dessas regiões é um borramento nas bordas das sombras, cujo efeito visual se assemelha ao das penumbras das sombras reais. Deve-se no entanto atentar para o fato de que este é apenas um efeito ocasional do método, e que essas penumbras casuais apresentam limitações em relação às obtidas em sombras reais. Uma dessas limitações é a não alteração da espessura da penumbra quando se levam em consideração as variações nas distâncias entre superfícies, oclusores e fontes de luz. Penumbras são obtidas através de fontes de luz de área, e os formatos dessas áreas tem influência direta sobre os formatos das penumbras, fenômeno que também não é representado de forma correta pelo *PCF*.

3.5 Trabalhos Relacionados

A literatura acerca da geração de sombras em computação gráfica é bastante vasta, e vai além do escopo deste trabalho. Revisões sobre um grande conjunto de algoritmos para geração de sombras podem ser vistos em (WOO; POULIN; FOURNIER, 1990), (HASENFRATZ et al., 2003) e (MOLLER; HAINES, 2002). Nesta seção nos concentramos nos algoritmos que estão diretamente relacionados ao tema deste trabalho.

O algoritmo de *PCF* apresentado anteriormente (seção 3.4) reduz o *aliasing* nas bordas das sombras através da filtragem dos resultados dos testes de sombra. Lokovic e Veach criaram os *Deep Shadow Maps* (LOKOVIC; VEACH, 2000). Esta técnica, utilizada no cálculo de sombras para pêlos, nuvens e fumaça, conta com um *Shadow Map* capaz de armazenar funções de atenuação em suas células. À medida que um raio de luz penetra em algum desses meios, a função de atenuação da célula correspondente do *Shadow Map*

informa o grau de atenuação. O problema de *aliasing* é bastante reduzido, visto que um procedimento de filtragem está implícito no funcionamento do algoritmo. O problema relacionado ao *Deep Shadow Map* reside no fato de que um pré-processamento da cena, de forma a definir as funções de atenuação para cada célula do *Shadow Map*, é necessário, inviabilizando sua implementação, como originalmente proposto, para aplicações de tempo real.

Embora a técnica de filtragem seja um método clássico na redução de *aliasing* em imagens, outros métodos tentaram atacar esse problema através do aumento da resolução do *Shadow Map*. Esse tipo de abordagem foi utilizada por Fernando na técnica intitulada "*Adaptive Shadow Maps*"(FERNANDO et al., 2001). Neste trabalho, uma extensão da estrutura do *Shadow Map*, de forma a garantir que este venha a ter uma resolução adequada, é proposta. Fernando observou que a resolução do *Shadow Map* não precisa ser uniforme, dado que o problema de *aliasing* torna-se visível somente nas regiões de fronteira de sombra. Desta forma, o algoritmo subdivide o *Shadow Map* de forma hierárquica, garantindo uma maior resolução nas regiões que representam fronteiras de sombra ou em regiões onde a resolução da tela é maior que a do *Shadow Map*. Embora o método seja efetivo no que diz respeito a eliminação do *aliasing*, o custo envolvido na manutenção da estrutura hierárquica do *Shadow Map* torna proibitiva a sua utilização em aplicações que demandam interatividade.

Stamminger e Drettakis desenvolveram o "*Perspective Shadow Maps*"(*PSM*) (STAMMINGER; DRETAKKIS, 2002) como forma de reduzir o *aliasing* através da utilização de *Shadow Maps* no espaço pós-projetivo. Após a transformação perspectiva, o volume de visualização da câmera é transformado em um cubo, fazendo com que os objetos mais próximos da câmera fiquem maiores do que os que se encontram mais distantes. Desta forma, um *Shadow Map*, de resolução fixa e constante, posicionado logo acima desta cena é capaz de amostrar de forma mais precisa os objetos que se encontram próximos da câmera. A técnica pode ser implementada de forma a se beneficiar da eficiência do *hardware* gráfico, permitindo dessa forma a geração de imagens a taxas interativas. Apesar de eficiente, a qualidade obtida com o *PSM* depende fortemente das posições relativas entre a câmera e a fonte de luz. O problema de *biasing* também torna-se mais complicado no *PSM*, visto que a distribuição das células do *Shadow Map* não ocorre de maneira uniforme sobre a cena, significando que o valor do *bias* não pode mais ser fixo. Chong e Gortler apresentam em "*A Lixel for Every Pixel*"(CHONG; GORTLER, 2004) um trabalho semelhante. Diferentemente do *PSM*, esta solução não opera no espaço pós-projetivo. Baseando-se na posição da câmera, da fonte de luz e de planos de interesse definidos para a cena, matrizes que transformam os pontos do sistema do universo para o espaço da luz são geradas. Essas matrizes mapeiam cada ponto amostrado pela câmera para um *lixel*[1] do *Shadow Map*. A técnica apresenta boa performance (cerca de 30 quadros por segundo para os exemplos do artigo), o *aliasing* eventualmente surge em superfícies que não estão alinhadas com os planos de interesse. Além do mais, cada plano de interesse exige um *Shadow Map*, e conseqüentemente uma passada, adicional.

Martin e Tan criaram o "*Trapezoidal Shadow Maps*"(MARTIN; TAN, 2004), que também opera no espaço pós-projetivo. Eles observaram que uma das causas do *aliasing* é o desperdício da resolução do *Shadow Map* sobre regiões do universo que não estão no

[1]Na nomemclatura adotada neste trabalho, um *lixel* é um elemento do *Shadow Map*

campo de visão da câmera. A solução proposta faz com que o volume de visualização da fonte de luz se ajuste geometricamente ao volume de visualização da câmera, diminuindo esse desperdício. Para permitir esse ajuste, o volume de visualização da câmera, como visto pela fonte de luz em seu espaço pós-projetivo, é envolvido por um trapézio (como uma envoltória convexa). Esse trapézio definirá o formato do novo *Shadow Map* (ajustado ao formato do volume de visualização). Um ponto da cena, para ser testado contra este novo *Shadow Map*, deve ser transformado para esse novo espaço trapezoidal. O método apresenta boa performance, e pode ser implementado em *hardware*. Problemas de amostragem relacionados ao desperdício de resolução do *Shadow Map* são minimizados. O método não resolve os problemas de amostragem advindos do duelo entre as direções de visualização dos volumes.

Aila et al. (AILA; LAINE, 2004) e Johnson et al. (JOHNSON; MARK; BURNS, 2004) apresentam métodos semelhantes para redução do *aliasing* em sombras geradas pelo *Shadow Mapping*. Em ambas as técnicas, o primeiro passo é a amostragem da cena a partir do ponto de vista da câmera. No segundo passo, esses pontos são convertidos para o sistema de coordenadas da luz, e então utilizados na rasterização da cena a partir do ponto de vista da luz. Embora os resultados sejam comparáveis àqueles obtidos com os *Shadow Volumes* (CROW, 1977), ou mesmo com os dos *Shadow Rays* (WHITTED, 1980), o algoritmo não é eficientemente mapeável sobre a arquitetura do *hardware* gráfico disponível atualmente. O trabalho de Johnson et al. conta também com uma proposta de arquitetura, capaz de permitir a implementação de seu algoritmo em tempo real.

Brabec e Seidel propuseram duas implementações distintas do algoritmo de *PCF* capazes de tirar proveito do paralelismo encontrado no *hardware* (BRABEC; SEIDEL, 2001). A primeira implementação conta com um *Shadow Map* que armazena múltiplos valores de profundidade por célula, obtidos a partir de 4 imagens (contendo informação de profundidade) geradas a partir do ponto de vista da luz. Antes da geração de cada uma das 4 imagens, um pequeno valor de deslocamento é aplicado ao plano da imagem onde é feito o registro dos valores de profundidade da cena. Adicionalmente, durante a geração de cada uma das 4 imagens, somente um dos canais de cor da textura (*Shadow Map*) está habilitado para escrita. Isso fará com que, ao final das 4 passadas, cada célula do *Shadow Map* contenha 4 valores de z. Durante a geração da cena final, a partir do ponto de vista da câmera, cada ponto da cena é testado contra os 4 valores de profundidade armazenados na célula correspondente do *Shadow Map*. O resultado final destes testes indicará a atenuação total a ser aplicada sobre a cor do fragmento. Pode-se observar que o esquema proposto trata-se na realidade se uma superamostragem da cena, fazendo com que a resolução efetiva do *Shadow Map* seja multiplicada por um fator de 2 em cada dimensão. Este método, apesar de eficaz em relação à suavização das bordas das sombras, conta com o custo de 4 passadas adicionais somente para a geração do *Shadow Map*.

Devido a esse custo adicional resultante da geração do *Shadow Map*, Brabec e Seidel propuseram também uma pequena modificação no algoritmo tornando-o capaz de gerar resultados semelhantes de forma mais eficiente. Nesta segunda proposta um *Shadow Map* tradicional é gerado a partir do ponto de vista da luz, sendo que o valor de profundidade registrado por cada célula é replicado em seus 4 canais de cor. Em uma segunda passada sobre o *Shadow Map* um processo de convolução, com pesos ajustados especialmente para este fim, faz com que cada uma de suas células propague para suas vizinhas o seu va-

lor de profundidade. Ao final desta segunda passada, cada célula do *Shadow Map* conterá 4 valores de z: um obtido a partir do primeiro passo e mais três valores de z recebidos a partir de suas vizinhanças através do processo de convolução. Depois dessa etapa a cena final é gerada da mesma forma que no método anterior, sendo cada ponto da cena testado contra os 4 valores de profundidade da respectiva célula do *Shadow Map*. Os resultados visuais obtidos através deste segundo método foram muito semelhantes aos obtidos com o primeiro. Quanto à performance, o segundo método apresentou um aumento significativo quando comparado ao primeiro.

Os dois algoritmos apresentados por Brabec e Seidel, assim como a técnica proposta neste trabalho, armazenam múltiplos valores de profundidade por célula. Enquanto o primeiro algoritmo conta com uma superamostragem da cena para a geração desses valores adicionais, o segundo executa uma passada adicional sobre o próprio *Shadow Map*. A vantagem do segundo algoritmo reside no fato de não ser necessária uma nova passada pela geometria da cena, estando o seu custo vinculado somente à resolução do *Shadow Map*. Enquanto o trabalho de Brabec e Seidel trata apenas do caso em que cada célula possui 4 valores de profundidade, o nosso trabalho generaliza esta idéia para o caso de n valores de profundidade por célula, onde $n \geq 1$.

4 *SHADOW MAPPING* COM MÚLTIPLOS VALORES DE PROFUNDIDADE

"The z-buffer algorithm's singular versatility, efficiency, and simplicity make it tempting to look for ways to overcome its drawbacks, particularly the more serious aliasing problem."
— WILLIAM T. REEVES ET AL.

Como discutido no capítulo anterior, o *Shadow Mapping* é atualmente uma das técnicas de geração de sombras que melhor se adapta à implementação em *hardware*, sendo o serrilhado presente nas bordas das sombras o principal empecilho relacionado ao seu uso. O algoritmo de *PCF*, apesar de não muito eficiente, mostrou-se bastante eficaz em relação à atenuação dos desagradáveis efeitos visuais provocados por este serrilhado. Além de bastante geral e robusto, o *PCF* permite ainda a geração de efeitos visuais semelhantes a penumbras, obtidos por meio da manipulação de alguns de seus parâmetros.

Uma observação importante a ser feita é quanto à qualidade da suavização das bordas de sombra obtida através do uso do *PCF*. Essa qualidade está relacionada a três pontos principais: o formato da região utilizada no momento da amostragem do *Shadow Map* e a quantidade e distribuição das amostras dentro dessa região. Como visto anteriormente, melhores resultados de suavização são obtidos sempre que a região a ser amostrada aproxima de forma mais precisa a região coberta pela projeção do *pixel* sobre a cena e/ou quando o número e distribuição das amostras cobre de maneira apropriada a região de amostragem no *Shadow Map*. Embora resultem em melhora de qualidade, a utilização de políticas de amostragem sofisticadas (com distribuições estocásticas ou gaussianas) e, principalmente, a precisa avaliação da região de amostragem, representam custos adicionais demasiadamente grandes quando comparadas aos benefícios que proporcionam. Por estas razões, diversas implementações ainda têm buscado a melhora da qualidade da filtragem proporcionada pelo *PCF* através do aumento do número de amostras feitas sobre o *Shadow Map* (BRABEC; SEIDEL, 2001), (FERNANDO, 2004).

O problema destas aplicações é que antes mesmo que o número de amostras seja suficiente para permitir a geração de níveis satisfatórios de filtragem, o custo envolvido na sua obtenção já compromete de forma significativa a performance da aplicação. A observação de que o *PCF* executa um procedimento de filtragem sobre um conjunto de valores binários foi o principal motivador deste trabalho. De certa forma, isso limita a capacidade de suavização do filtro *PCF*, visto que caso seus valores não estivessem limitados a um conjunto binário, um número maior de níveis de atenuação poderiam ser obtidos com o uso de um mesmo filtro de tamanho fixo (PAGOT; COMBA; OLIVEIRA, 2004). Este

capítulo irá discutir em detalhes a maneira como essa observação levou à generalização do algoritmo de *Shadow Mapping*, resultando em uma melhora significativa da qualidade visual das sombras e reduzindo o custo da sua filtragem em determinados casos.

4.1 Limitações do *PCF*

Para que a implementação do *PCF* possa tirar proveito dos recursos disponibilizados pelo *hardware*, a ponto de viabilizar a sua utilização em aplicações de tempo real, uma série de simplificações devem ser feitas. Como visto na seção 3.5, o *PCF* tem sido implementado como um teste de visibilidade entre um ponto da cena e uma região do *Shadow Map*, sendo o formato fixo para todos os pontos da cena testados contra o *Shadow Map*. Limitações na largura de banda existentes na comunicação entre a *GPU* e a memória de vídeo reduzem o número de amostras que podem ser feitas nas regiões de amostragem definidas sobre o *Shadow Map*.

Todas essas simplificações resultam no aumento de performance do algoritmo de *PCF*, ao mesmo tempo em que causam uma diminuição na qualidade de sua filtragem. Como visto na seção anterior, embora outras alternativas estejam disponíveis, a maneira mais simples de se aumentar a qualidade da filtragem seria através do aumento do número de amostras feitas sobre o *Shadow Map*. Mesmo assim, mais cedo ou mais tarde esbarraríamos novamente no problema da largura de banda da memória, e a solução se tornaria ineficiente. Uma observação feita neste ponto se refere à importância do número de amostras feitas sobre o *Shadow Map*. A sua importância reside no fato de que cada amostra será submetida a um teste de sombra, sendo o conjunto dos resultados desses testes filtrados pelo *PCF*. Quanto maior o número de amostras feitas sobre o *Shadow Map*, maior será o conjunto de valores que irão alimentar o filtro do *PCF*, significando que uma gama maior de valores distintos de atenuação poderão ser retornados.

Até esse momento o número de valores distintos de atenuação que o filtro do *PCF* pode retornar está relacionado exclusivamente ao número de amostras obtidas a partir do *Shadow Map*. Pode-se dizer desta forma, e sem erro, que um filtro de 4 elementos é capaz de retornar até 5 valores distintos de atenuação, enquanto que um filtro de 6 elementos pode retornar até 7 valores distintos. Essa idéia pode ser inclusive generalizada para um filtro de n elementos. Assumindo que cada elemento pode possuir apenas um valor do conjunto $\{0, 1\}$ (que são os dois valores possíveis de serem retornados pelos testes de sombra), o número de valores distintos de atenuação (k) possíveis de serem retornados por um filtro de n elementos pode ser expresso como:

$$k = n + 1 \tag{4.1}$$

A fórmula acima apresenta a capacidade do *PCF* em gerar valores distintos de atenuação como uma função exclusiva do número de elementos do seu filtro. Uma observação sutil, e que motivou esse trabalho, é a de que a capacidade do filtro em gerar essas diferentes gradações não é uma função exclusiva do número de elementos do filtro, mas também do número de valores distintos que cada um desses elementos pode assumir. A fórmula acima foi derivada a partir da premissa de que cada elemento do filtro poderia assumir um valor entre dois possíveis. Neste ponto, a pergunta é: o que aconteceria com o número de valores distintos de atenuação se os elementos do filtro pudessem assumir valores dentre

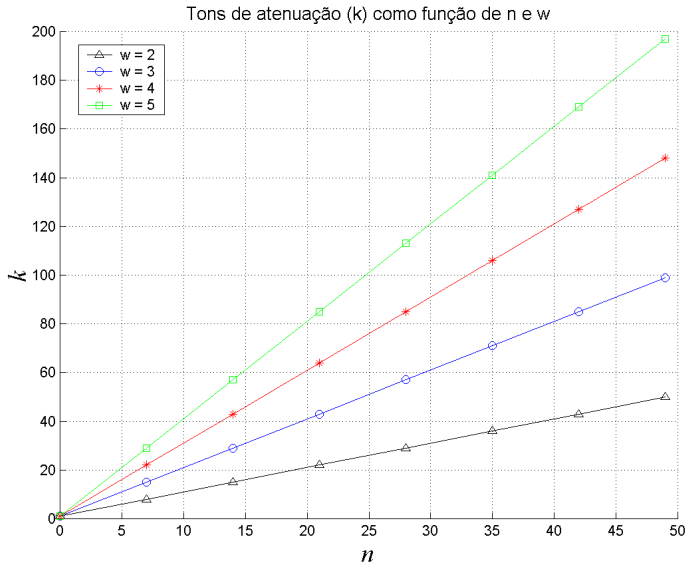


Figura 4.1: Capacidade de geração de tons distintos de atenuação de um filtro *PCF* representado em função de n (número de elementos de seu filtro) e w (número de possibilidades de valores para os elementos do filtro do *PCF*).

uma gama maior?

Considerando-se agora que cada elemento do filtro não está mais restrito a um conjunto binário de valores, podendo assumir um entre w valores distintos, o número total de valores de atenuação (k) gerados por um filtro de n elementos pode ser expresso como:

$$k = (w - 1) \times n + 1 \tag{4.2}$$

Fazendo-se uma comparação entre um filtro *PCF* aplicado sobre um conjunto de n elementos binários ($w = 2$), e outro aplicado também sobre um conjunto de n elementos, desta vez capazes de assumir 1 dentre 3 valores distintos ($w = 3$), fica evidente que no segundo caso haverá um aumento no número de valores distintos de atenuação, sem que isso tenha implicado em um aumento no número de elementos do filtro. A figura 4.1 apresenta a variação no número de tons gerados como uma função dos valores de n e w.

Dado que os valores dos elementos do *PCF* são os resultados dos testes de sombra, para que possamos realmente tirar proveito de uma situação onde $w > 2$, uma alteração no teste de sombra, de forma a permitir que ele retorne um número maior de resultados, é necessária. Como se poderia alterar o teste de sombra para que ele seja capaz de retornar valores adicionais?

4.2 Estendendo o teste de sombra

Como visto na seção anterior, uma situação onde $w > 2$ implica que os resultados dos testes de sombra não podem estar limitados a um conjunto binário. Para isto, o teste de sombra tradicional deve ser estendido, permitindo que ele possa retornar uma gama maior de valores.

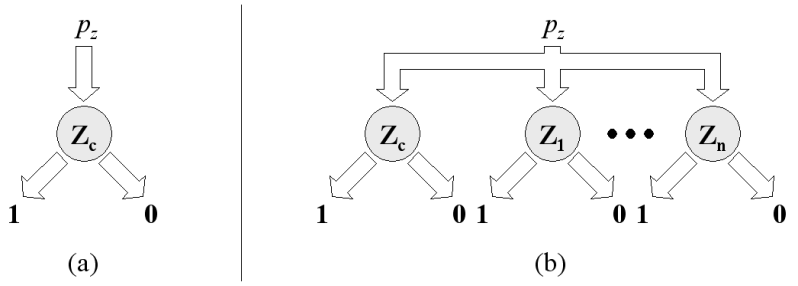


Figura 4.2: (a) Teste de sombra tradicional, executado entre um ponto da cena e um elemento do *Shadow Map*, apresentando resultado binário. (b) Teste de sombra composto, executado entre um ponto da cena e diversos elementos do *Shadow Map*, sendo seu resultado representado pela média dos resultados dos testes individuais.

Embora possam existir diversas alternativas para a extensão do teste de sombra, uma solução possível, e adotada neste trabalho, foi a sua extensão através da composição de um conjunto de testes de sombra tradicionais. Desta forma, o novo teste de sombra não trata apenas de um teste de um ponto da cena contra um único valor de z do *Shadow Map*, mas contra um conjunto de valores de z. Cada teste entre o ponto da cena e um dos valores de z desse conjunto compõem um teste de sombra tradicional que resultará em 0 (iluminado) ou 1 (em sombra). O resultado final desse novo teste será obtido através da média dos resultados dos testes de sombra tradicionais que o compõem. A figura 4.2 mostra uma comparação entre o teste de sombra tradicional, capaz de retornar um valor binário, e o novo teste de sombra, capaz de retornar uma gama maior de valores.

A pergunta que obviamente surge é sobre quais poderiam ser os valores adicionais de z a serem utilizados no novo teste de sombra. Embora outras técnicas, tais como o *Depth Peeling* (EVERITT, 2001a), possam ser exploradas na obtenção desses valores adicionais, optou-se por extraí-los a partir do próprio *Shadow Map*, evitando desta forma o custo de várias passadas pela cena. Esses valores adicionais devem ser escolhidos de maneira que os resultados do novo teste continuem gerando as sombras da cena de forma correta. Deste modo, a escolha desses valores adicionais começará pelo valor de z utilizado pelo teste de sombra tradicional.

No teste de sombra tradicional, um ponto p da cena é transformado para o sistema de coordenadas da luz, e posteriormente projetado sobre o *Shadow Map*. O valor de z do *Shadow Map*, sobre o qual p é projetado, será chamado de z_c, e será obrigatoriamente um dos valores a serem utilizados no novo teste de sombra. A presença de z_c no teste composto de sombra é importante pois é ele que irá ajudar na preservação da correta estrutura da sombra.

Os demais valores podem ser amostrados a partir de uma vizinhança no *Shadow Map* definida ao redor de z_c, à semelhança do que ocorre com o *PCF*. Uma das vantagens da seleção desses valores adicionais a partir do mesmo *Shadow Map* é que apenas uma passada adicional sobre o *Shadow Map* é necessária. Essa vizinhança pode ter diversos formatos e extensões, e a política adotada em sua amostragem também pode variar. Os valores adicionais amostrados ao redor de z_c serão chamados respectivamente de z_1, z_2, ..., z_n, sendo n o número total de amostras feitas, excluindo z_c. Como forma de facilitar

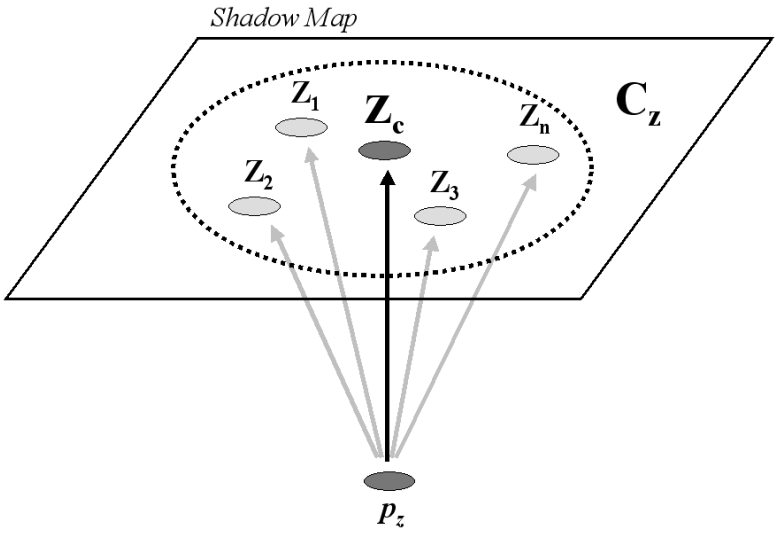


Figura 4.3: O conjunto valores de profundidade (C_z) utilizados pelo novo teste de sombra.

a referência a estes valores, chamaremos a partir de agora o conjunto de valores de z utilizados pelo novo teste de sombra de $C_z = \{z_c, z_1, z_2, ..., z_n\}$. A figura 4.3 ilustra essa situação.

Maiores detalhes acerca das diversas políticas para a seleção dos valores adicionais de z para o novo teste de sombra serão apresentados na seção 4.5.

4.3 O *Shadow Map* com Múltiplos Valores de Profundidade (SMMVP)

Na seção anterior foi definido que z_c é o valor de profundidade da célula do *Shadow Map* sobre o qual um ponto p da cena se projeta no momento do teste de sombra. O *rendering* de uma cena, com *Shadow Map*, implica que todos os pontos da cena, visíveis pela câmera, serão transformados para o sistema de coordenadas da luz e projetados sobre o *Shadow Map*.

Devido à organização da cena, à posição da fonte de luz, ou mesmo à posição da câmera, muitos pontos da cena podem ser projetivamente equivalentes em relação ao *Shadow Map*, sendo projetados sobre uma mesma célula. Pontos da cena que se projetarem sobre uma mesma célula irão recuperar exatamente o mesmo valor de z_c. Pequenas diferenças de deslocamento dentro de uma mesma célula do *Shadow Map* não resultarão em diferenças no valor de z_c pois o *Shadow Map* é representado por uma grade regular, e o seu conteúdo nunca é filtrado. Além disso, os valores adicionais de C_z ($z_1, z_2, ...z_n$) são amostrados a partir das redondezas de z_c, significando que, em decorrência da coerência espacial, pontos distintos dentro de uma mesma célula do *Shadow Map* darão origem a conjuntos C_z provavelmente distintos, mas cujos valores médios serão próximos. Essa situação é ilustrada na figura 4.4.

A coerência acerca dos valores de C_z faz com que uma otimização possa ser feita: para cada célula do *Shadow Map*, o seu C_z correspondente pode ser previamente calcu-

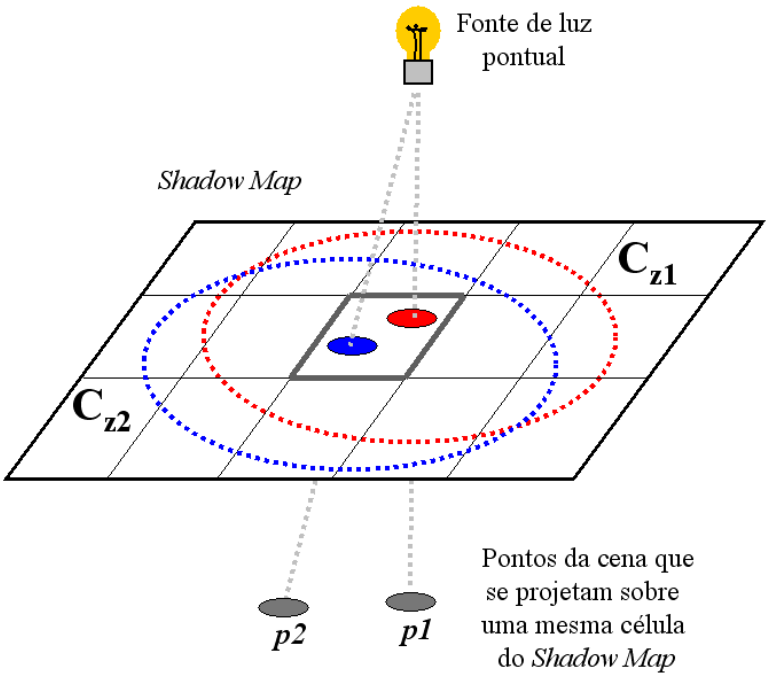


Figura 4.4: A figura acima apresenta a intersecção de duas regiões de amostragem geradas por dois pontos distintos da cena ($p1$ e $p2$), que se projetam sobre uma mesma célula do *Shadow Map*. Isso significa que, embora os elementos dos conjuntos C_{z1} e C_{z2} possam assumir valores distintos, os valores médios desses conjuntos serão próximos em decorrência da coerência espacial dos elementos do *Shadow Map*.

lado, eliminando a necessidade de recálculo nos casos em que mais de um ponto da cena se projeta sobre uma mesma célula. Esse pré-cálculo de C_z pode resultar em trabalho desnecessário no caso de células do *Shadow Map* que não recebam a projeção de nenhum ponto da cena. Isso ocorrerá sempre que o *Shadow Map* amostrar regiões da cena que não são visíveis a partir do ponto de vista da câmera. Embora essa seja uma possibilidade, em se tratando de *Shadow Mapping*, geralmente se tem o cuidado para que isso não ocorra, pois a resolução do *Shadow Map* é limitada e o seu desperdício sobre partes da cena que não contribuirão à cena final pode acarretar em degradação acentuada das sombras. Brabec desenvolveu um trabalho onde explora técnicas para evitar que esse tipo de desperdício ocorra (BRABEC; ANNEN; SEIDELL, 2002). Uma outra vantagem relacionada ao pré-cálculo do C_z é que a geração desses valores adicionais representa um custo fixo, independente da complexidade da cena, sendo proporcional à resolução do *Shadow Map*. A análise desses custos é apresentada em detalhe no capítulo 5.

Essa possibilidade de pré-seleção dos valores de C_z para cada elemento do *Shadow Map* dá origem ao conceito de *Shadow Map* com Múltiplos Valores de Profundidade (SMMVP) (PAGOT; COMBA; OLIVEIRA, 2004), onde cada célula do *Shadow Map* contém, além de seu valor de profundidade original, os valores adicionais de profundidade selecionados a partir de sua vizinhança.

Antes de entrarmos em uma discussão mais detalhada sobre a seleção dos valores adicionais de C_z, apresentaremos as formalizações dos conceitos introduzidos até aqui, como forma de facilitar a explanação do método.

4.4 Formalização

A seguir apresentamos a formalização dos conceitos referentes ao novo teste de sombra e ao *Shadow Map* com Múltiplos Valores de Profundidade. Uma formalização adicional, referente à aplicação do *PCF* sobre um SMMVP, também será apresentada.

***Shadow Map* com Múltiplos Valores de Profundidade (SMMVP):**

Um *Shadow Map* com Múltiplos Valores de Profundidade é um *Shadow Map* capaz de armazenar múltiplos valores de profundidade em cada uma de suas células. Um dos valores de profundidade da célula será obrigatoriamente o z_c, obtido através da geração do *Shadow Map* tradicional. Os valores adicionais são obtidos a partir de uma amostragem feita sobre uma vizinhança definida ao redor da célula que contém z_c. A notação n-SM especifica um SMMVP que possui n valores de profundidade armazenados em cada uma de suas células. Fica claro que essa notação não especifica o formato e extensão da região de amostragem, nem a política de amostragem adotada na seleção dos valores adicionais de profundidade. Essas informações, quando relevantes, deverão ser explicitamente apresentadas. Dentro deste contexto, o *Shadow Map* tradicional passa a ser um caso especial de um SMMVP, onde $n = 1$, podendo desta forma ser descrito como 1-SM.

Teste de Sombra Composto (TSC):

O novo teste de sombra, chamado agora de Teste de Sombra Composto, é uma extensão do teste de sombra tradicional. Um n-TS executa n testes lógicos entre um ponto p da cena e os n valores de profundidade contidos no C_z da célula de um n-SM. O resultado do n-TS é dado pela razão s/n, onde s é a quantidade de testes lógicos cujos resultados são "verdade", e $0 \leq s \leq n$. Um 1-TS pode ser interpretado como o teste de sombra do *Shadow Map* tradicional, enquanto que um 2-TS é um teste de sombra aplicado sobre um 2-SM.

Filtro PCF(k,n):

Esta formalização adicional foi criada como uma forma de representar, de forma mais compacta, a aplicação de um filtro *PCF*, com k elementos, sobre um n-SM. Deve ficar claro aqui que os valores dos elementos do filtro *PCF* serão definidos pelos respectivos TSCs. Da mesma forma como ocorria com o n-SM, a notação utilizada aqui para o *PCF* não informa a área de extensão ou política de amostragem utilizada na obtenção dos k elementos que alimentarão o filtro.

Através desta notação, um *PCF*(1,1) representa o algoritmo de *Shadow Mapping* tradicional sem a aplicação de filtro *PCF*. Um *PCF*(4,1) pode ser interpretado como um *PCF*, cujo filtro possui 4 elementos, e que está sendo aplicado sobre um *Shadow Map* tradicional. A notação *PCF*(3,2) define a aplicação de um *PCF*, cujo filtro possui 3 elementos, sobre um 2-SM. Um *PCF*(k,n) é capaz de retornar $n \times k + 1$ valores diferentes de atenuação.

4.5 Selecionando os valores adicionais de profundidade para o SMMVP

Como visto na seção 4.2, a seleção dos valores adicionais das células de um SMMVP será feita a partir de uma região definida ao redor das próprias células. Uma otimização proposta, e que resulta no bom desempenho do método, é a seleção desses elementos adicionais através de um pré-processamento executado sobre o *Shadow Map*, antes do início da geração da imagem final, com sombras.

A seleção desses elementos adicionais deve ser feita de forma a permitir que os TSCs, a serem aplicados posteriormente, sejam capazes de gerar valores adicionais de atenuação, principalmente nas regiões próximas às bordas das sombras. Esses valores adicionais de atenuação serão posteriormente filtrados pelo *PCF*, o que permitirá a geração de uma gama ainda maior de valores de atenuação, resultando em transições mais suaves entre regiões iluminadas e em sombra.

No decorrer desta seção desenvolveremos a intuição por trás da escolha dos valores adicionais para as células do SMMVP. Em seguida apresentamos a construção de diversos SMMVPs, cada um com um determinado número de valores por célula. Por fim, é apresentada uma política de seleção de valores aplicável nos casos em que o número de elementos por célula é maior que 3.

4.5.1 A intuição por trás da escolha dos valores adicionais do SMMVP

Para que a idéia por trás da seleção dos valores adicionais de profundidade das células do SMMVP fique mais clara, vamos voltar ao algoritmo de *Shadow Mapping*, ver o comportamento do seu teste de sombra, e então desenvolver a intuição para a escolha desses valores. Na figura 4.5 são apresentadas duas situações: uma onde um ponto é testado contra o *Shadow Map* e avaliado como estando em sombra, e outra onde um ponto é avaliado como completamente iluminado.

Como se pode observar, o teste de sombra utilizado retorna apenas os valores "iluminado"ou "em sombra". A extensão desse teste, de forma que o ponto da cena possa ser testado contra múltiplos valores de profundidade do *Shadow Map*, resultaria no retorno de uma gama maior de valores de atenuação. Essa situação é demonstrada na figura 4.6.

Podemos observar que os valores de atenuação retornados pelo TSC não ficam mais restritos a um conjunto binário. Os possíveis valores de atenuação são múltiplos e estão relacionados à quantidade de elementos da região do *Shadow Map* que se interpõem entre o ponto da cena e a fonte de luz. Pontos da cena que estão em sombra, e suficientemente distantes [1] das bordas da sombra, são avaliados como estando completamente em sombra. No caso oposto, onde o ponto encontra-se iluminado e suficientemente distante da borda da sombra, este será avaliado como estando completamente iluminado.

O retorno de valores de atenuação intermediários ocorre quando p encontra-se sufi-

[1] A expressão "suficientemente distante"é utilizada, neste caso, para informar que a região de amostragem gerada pelo ponto da cena não intercepta as discontinuidades dos valores de profundidade do *Shadow Map*.

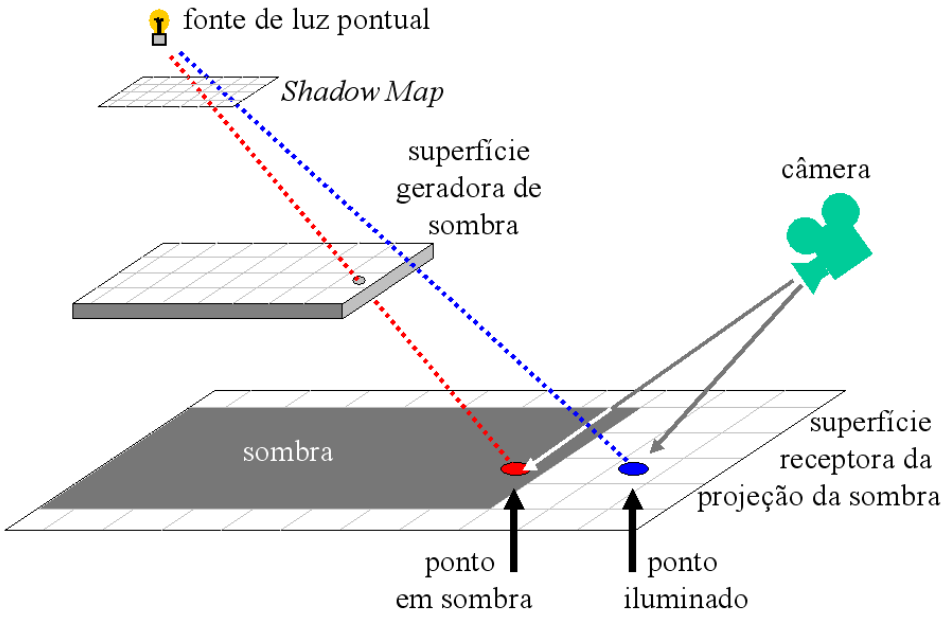


Figura 4.5: O teste de sombra tradicional, utilizado no algoritmo de *Shadow Mapping*, retorna apenas valores binários ("iluminado"ou "não iluminado"), mesmo quando esses testes são feitos nas proximidades de fronteiras de sombra.

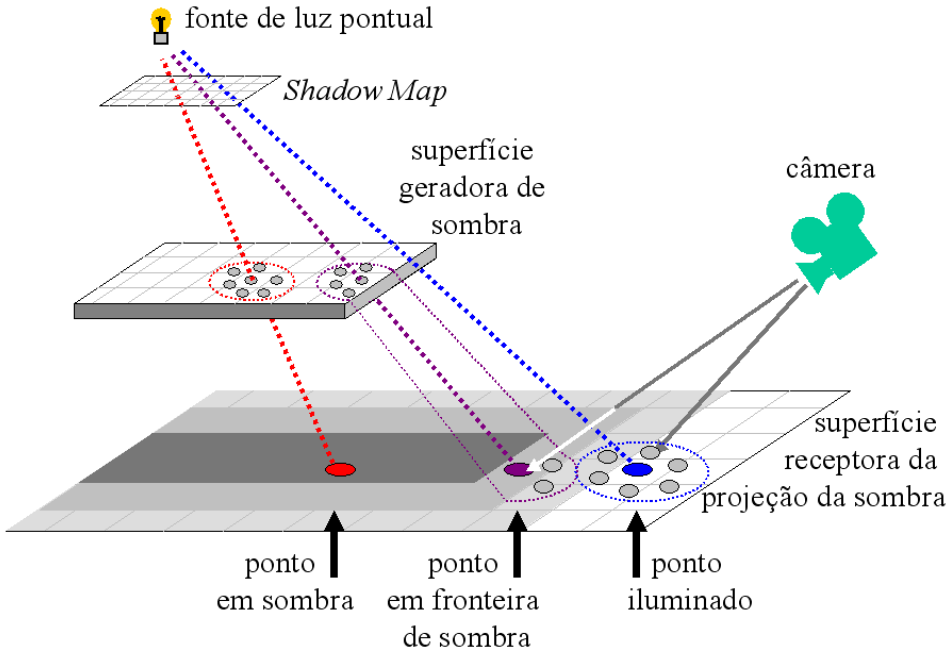


Figura 4.6: Resultados obtidos com o novo teste de sombra. Nesse novo teste, cada ponto gerado pela câmera é testado contra um conjunto de pontos do *Shadow Map*, a exemplo do que ocorre com o *PCF*. Pode ser observado que o novo teste gera resultados intermediários de atenuação (no intervalo $[0.0, 1.0]$) quando a região de amostragem intersecciona a fronteira de sombra.

cientemente próximo[2] da borda de sombra, de forma que a região de amostragem passa a interseccionar a fronteira de sombra. Desta forma alguns pontos amostrados e testados contra p indicarão que este se encontra em sombra, enquanto outros indicarão que p encontra-se iluminado. A filtragem desses resultados individuais dará origem ao valor de atenuação final do TSC, que estará no intervalo $[0.0, 1.0]$ (sendo que 0.0 significa completamente iluminado e 1.0 significa completamente em sombra).

As descontinuidades detectadas sobre o *Shadow Map* permitem a geração de valores intermediários de atenuação nas regiões de fronteira de sombra. Esse era o comportamento esperado pelo TSC, e isso faz com que esses valores, vizinhos às células, sejam candidatos à se tornarem os valores adicionais da célula. Cautela deve ser tomada no momento de se definir a resolução do *Shadow Map*, pois *Shadow Maps* com baixa resolução podem levar à detecção de discontinuidades que não representam necessariamente aquela gerada por superfícies geradoras de sombra, resultando em indevido auto-sombreamento de algumas regiões da cena.

Tendo visto a importância dos valores vizinhos à célula na construção do TSC, veremos agora como utilizar esse conceito na construção de diversos tipos de SMMVPs.

4.5.2 SMMVP com 2 valores de profundidade por célula (2-SM)

Um 2-SM é um SMMVP capaz de armazenar 2 valores de profundidade em cada uma de suas células. Como visto anteriormente, um dos valores de profundidade da célula será o valor gerado durante a geração do *Shadow Map*. O segundo valor de profundidade da célula será obtido a partir de amostragem de uma região ao redor da célula.

Valores intermediários de atenuação serão retornados pelo TSC quando os valores adicionais da célula em questão conseguirem capturar a discontinuidade de profundidade existente nas fronteiras de sombra. O que pode ocorrer no caso de um 2-SM é que se o segundo valor for amostrado de forma estocástica, ou através de um padrão fixo, tal discontinuidade pode não ser detectada, fazendo com que o TSC não retorne os valores esperados. Isso pode se tornar bastante evidente nos casos em que as superfícies definam discontinuidades com determinadas orientações. Essas situações podem ser observadas na figura 4.7.

Por esta razão, no caso da utilização de um 2-SM, uma política diferente de seleção do segundo valor deve ser usada, de forma a aumentar a probabilidade de detecção de eventuais discontinuidades dos valores de profundidade do *Shadow Map*, caso elas existam. Uma maneira de se obter esse resultado é percorrer a vizinhança da célula e então selecionar o maior ou o menor valor de profundidade nesta vizinhança. A seleção do maior ou menor valor apresenta custo linear $O(n)$, sendo n o número de amostras feitas ao redor da célula.

[2]Neste caso, a expressão "suficientemente próximo"se refere à região de amostragem, gerada por um ponto da cena, que intercepta a discontinuidade de valores do *Shadow Map*.

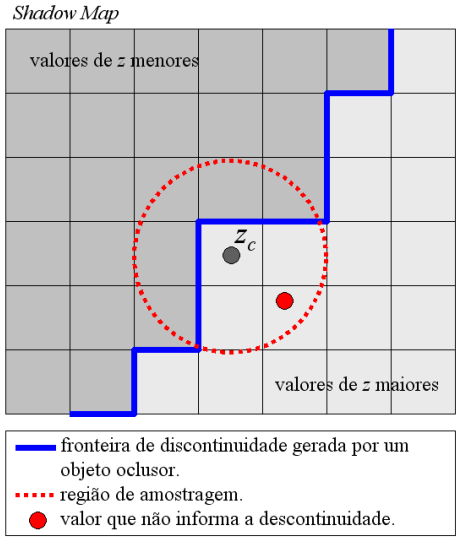


Figura 4.7: A seleção estocástica, ou com distribuição fixa, do segundo elemento de um 2-SM pode fazer com que as discontinuidades dos valores de profundidade do *Shadow Map* não sejam detectadas.

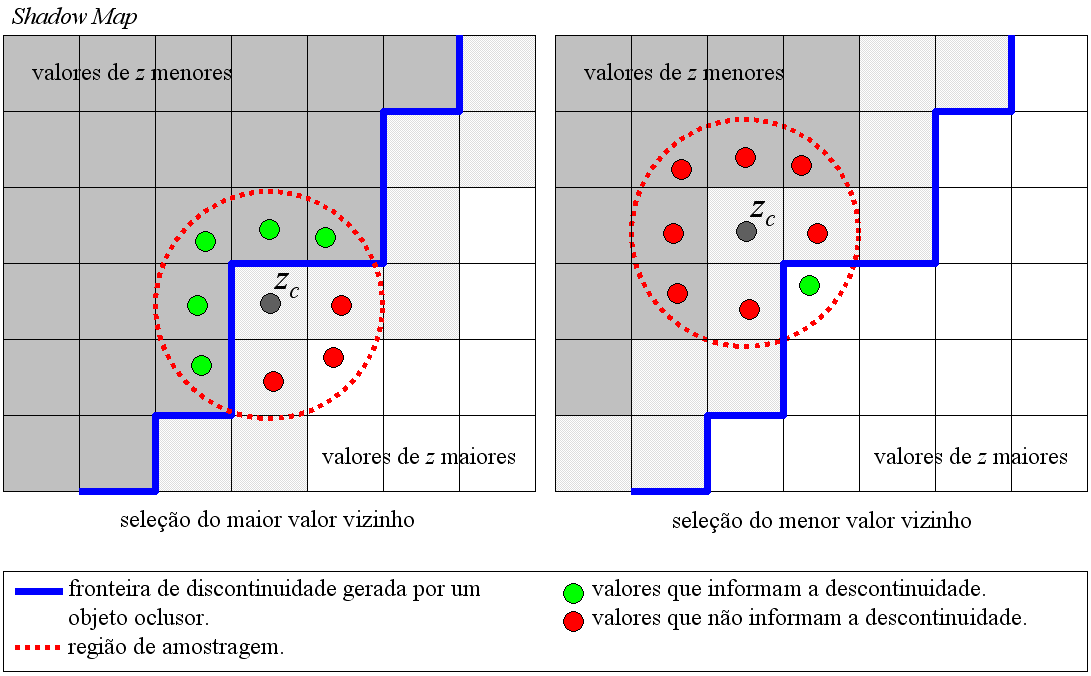


Figura 4.8: Uma maneira de se selecionar o segundo valor da célula de um 2-SM, de forma a aumentar a probabilidade de detecção de discontinuidades do *Shadow Map*, é através de uma amostragem maciça da vizinhança de z_c, com a posterior seleção do maior ou menor elemento como segundo elemento da célula do 2-SM.

4.5.3 SMMVP com 3 valores de profundidade por célula (3-SM)

O 3-SM é um SMMVP capaz de armazenar 3 valores de profundidade em cada uma de suas células. Novamente, um desses valores será o valor gerado durante a geração do *Shadow Map*, e os dois valores adicionais serão obtidos através da amostragem da região definida na vizinhança da célula. Assim como o 2-SM, o 3-SM pode apresentar problemas na geração dos múltiplos valores de atenuação caso a posição dessas amostras sobre a região de amostragem seja fixa ou estocástica.

No caso do 3-SM, uma política que melhore a detecção de eventuais discontinuidades dos valores de profundidade armazenados no *Shadow Map* também deve ser usada. Deste modo, optou-se por utilizar o mesmo sistema utilizado no caso do 2-SM. Primeiro amostra-se de forma regular a vizinhança da célula, e a seguir selecionam-se os 2 valores mais representativos como valores adicionais da célula. Neste caso optou-se por fazer com que um dos valores seja o maior amostrado, e o outro seja o menor amostrado. Isso fará com que caso a região de amostragem interseccione uma fronteira de sombra, uma amostra venha de uma lado da fronteira enquanto a outra venha do outro lado.

4.5.4 SMMVP com n valores de profundidade por célula (n-SM, para $n \geq 4$)

Foi mostrado, nas seções 4.5.2 e 4.5.3, que os SMMVPs com 2 ou 3 elementos por célula permitem a utilização de padrões de amostragem que podem falhar na detecção de bordas de sombra, tornando-se este problema mais evidente no caso de discontinuidades que apresentem orientações sobre o *Shadow Map*. Também foram apresentadas técnicas que, apesar de onerarem o algoritmo, garantem uma melhor detecção dessas fronteiras.

A partir de 4 elementos por célula as amostras podem ser dispostas ao redor do valor de profundidade central de forma a garantirem uma melhor detecção de eventuais fronteiras de sombra. A figura 4.9 apresenta um padrão de amostragem possível, definido para o 4-SM, onde as amostras adicionais são dispostas de forma regular em torno da amostra central.

O padrão de amostragem apresentado não garante que fronteiras de sombra serão sempre detectadas, existindo casos em que essa detecção pode falhar. Esses casos podem ocorrer devido a utilização de um *Shadow Map* de baixa resolução ou a variações abruptas e repentinas na geometria da cena. A figura 4.10 ilustra esta situação.

Esse padrão de amostragem pode ser facilmente extendido para um n-SM com $n >$ 4, bastando para isso apenas a distribuição uniforme das amostras ao redor de célula. A figura 4.11 apresenta possíveis padrões de amostragem para um 5-SM e um 6-SM.

4.6 Implementação

Como visto nas seções anteriores, o algoritmo de SMMVP é muito semelhante ao algoritmo de *Shadow Mapping*. O primeiro passo gera o *Shadow Map* da cena a partir do ponto de vista da luz. Nesse momento pode-se considerar que as células do SMMVP possuem apenas um valor de profundidade. O segundo passo é encarregado de selecionar os valores adicionais das células, sendo necessário, neste caso, o percorrimento de todas

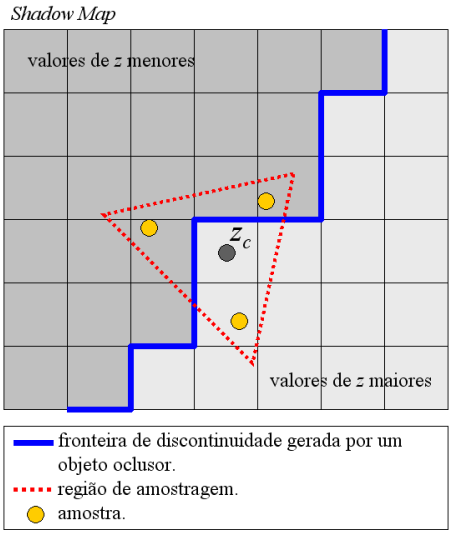


Figura 4.9: Um possível padrão de amostragem para geração dos valores adicionais de um 4-SM. O fato de as amostras estarem distribuídas uniformemente ao redor da amostra central aumentam a sua eficácia em detectar discontinuidades sobre o *Shadow Map*.

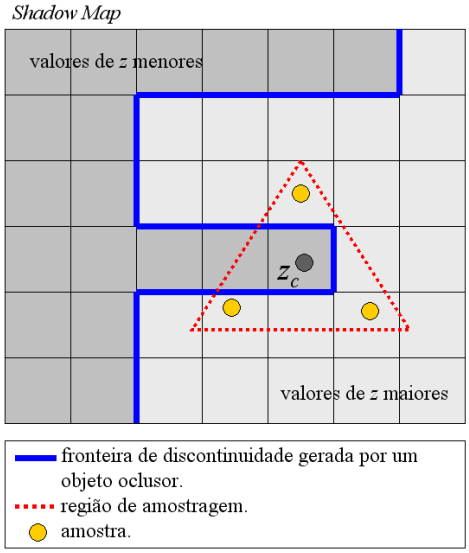


Figura 4.10: Apesar da distribuição uniforme das amostras, o padrão de amostragem pode eventualmente não detectar a discontinuidade de valores de profundidade de um *Shadow Map*. Essa falha pode ocorrer devido à baixa resolução do *Shadow Map*, ou às mudanças bruscas e repentinas na geometria da cena.

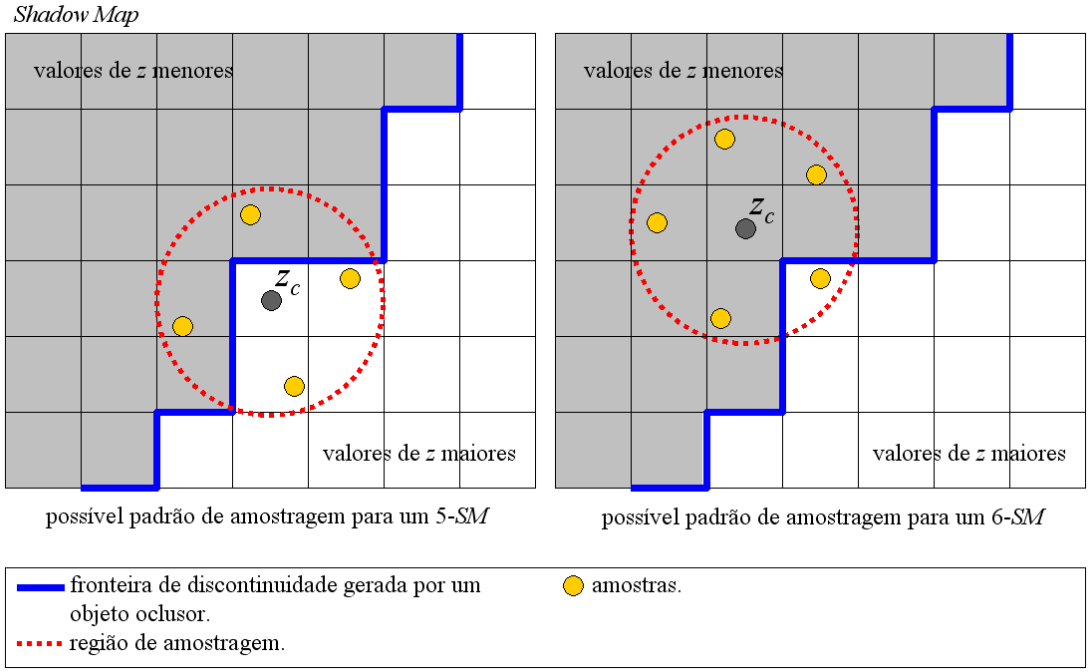


Figura 4.11: Possíveis padrões de amostragem para geração dos valores adicionais de um 5-SM e um 6-SM.

as células do *Shadow Map* e a respectiva amostragem de suas vizinhanças. Ao final deste passo, as células do SMMVP já possuem os valores adicionais. O terceiro e último passo é responsável pela geração da imagem final da cena, com sombras, a partir do ponto de vista da câmera. No caso de *shading* postergado (DEERING et al., 1988), este algoritmo contaria com um passo adicional, referente ao cálculo de visibilidade da cena a partir do ponto de vista da câmera. No decorrer desta seção apresentaremos detalhes de como o algoritmo, sem *shading* postergado, foi implementado.

4.6.1 Ambiente

O algoritmo de SMMVP foi implementado utilizando-se a linguagem C++ e *OpenGL* como *API* gráfica. A seleção dos valores adicionais de profundidade das células, por ser um processo repetitivo e local a cada célula, foi implementado de forma a tirar proveito do paralelismo disponibilizado pelo *hardware* gráfico. As placas gráficas utilizadas durante o desenvolvimento do protótipo foram uma *GeforceFX 5800* e uma *GeforceGT 6800*, ambas da NVIDIA. A escolha dessas placas deveu-se ao fato de ambas possuírem processadores de fragmento programáveis. O fato de pertencerem a gerações diferentes permitiu também uma avaliação mais robusta acerca do comportamento do algoritmo quando executado sobre diferentes arquiteturas (diferentes políticas de *cache*, largura de banda, etc.). A linguagem de *shading* utilizada na programação das *GPU*s foi a *Cg* da NVIDIA devido à sua popularidade. Nas seções seguintes a implementação de cada etapa do algoritmo, bem como as estruturas utilizadas, são explicadas em detalhe.

4.6.2 Primeiro passo: Gerando o *Shadow Map*

O primeiro passo do algoritmo trata da geração do *Shadow Map*. Para isto, posiciona-se uma câmera na posição da fonte de luz e faz-se o *rendering* da cena, registrando-se somente a coordenada z dos pontos amostrados. Como no passo seguinte será feita a seleção dos valores adicionais de profundidade das células a partir do *Shadow Map*, este deverá ser armazenado em uma estrutura que permita a posterior leitura de seu conteúdo.

Atualmente a única maneira de se disponibilizar o resultado de uma etapa de *rendering* para leitura pela própria *GPU* é através de sua transformação em uma textura, sendo isso feito de duas formas possíveis: através do *rendering* para o *color-buffer* e posterior cópia de seu conteúdo para uma textura, ou através do redirecionamento do *rendering* diretamente para um *pbuffer*.

A primeira alternativa, apesar de eficaz, é bastante ineficiente devido a necessidade de transferência de grandes quantidades de dados de uma região da memória para outra. Outra limitação imposta pelo *color-buffer* está relacionada à precisão de seus elementos. Cada elemento conta com apenas 24 bits (8 bits por canal) para a sua representação, e os valores representáveis ficam limitados ao intervalo normalizado $[0.0, 1.0]$.

Os *pbuffers* das placas atuais, por outro lado, disponibilizam até 32 bits para a representação, em ponto flutuante, de cada um dos quatro canais de seus elementos. Os valores desses elementos não precisam estar normalizados, significando uma maior flexibilidade no armazenamento do *Shadow Map*. *Pbuffers* também podem ser acessados como texturas imediatamente após o término do *rendering*, sem a necessidade de cópias de regiões de memória. Todos esses fatores contribuíram para a implementação do *Shadow Map* através de um *pbuffer*.

Para que se tire o máximo proveito da precisão oferecida pelo *pbuffer*, os valores interpolados de z que chegam ao processador de fragmento não são normalizados através da divisão por w, sendo enviados diretamente para o *pbuffer*. As figuras 4.12 e 4.13 apresentam os dois trechos de código utilizados respectivamente na programação do processador de vértices e de fragmento para a geração do *Shadow Map*.

O programa de vértice da figura 4.12 transforma os vértices do sistema de coordenadas do objeto para o sistema de coordenadas da luz. Os dados desses vértices transformados serão posteriormente interpolados (no estágio de rasterização) e enviados ao processador de fragmentos.

O programa de fragmentos da figura 4.13 recebe os fragmentos gerados pela interpolação dos polígonos da cena, já no espaço da luz. Nenhuma transformação é aplicada ao fragmento, e a saída do programa é o valor da coordenada z do fragmento. Como pode ser observado, a coordenada z não é dividida pela componente w de forma a garantir uma representação mais precisa do valor.

```
// programa de vertice
void main(
          // vertice no espaco do objeto
          float4       i_position       : POSITION,
          // vertice no espaco da luz
          out float4   o_position       : POSITION,
          // vertice no espaco da luz
          out float4   o_pos_lightspace : TEXCOORD0,

          // transformacao para o espaco da luz
          const uniform float4x4 modelViewProj
)
{
    o_position       = mul(modelViewProj, i_position);
    o_pos_lightspace = o_position;
}
```

Figura 4.12: Programa de vértices utilizado na geração do *Shadow Map*: transforma as coordenadas de um vértice (*i_position*), originalmente no espaço do objeto, para os sistemas de coordenadas do universo (*o_position*) e da luz (*o_pos_lightspace*).

```
// programa de fragmento
void main(
          // fragmento interpolado no espaco da luz
          float4 i_pos_lightspace : TEXCOORD0,
          // valor de saida do processador de fragmento
          out float4 color : COLOR
)
{
    // valor nao normalizado, com 32 bits de precisao
    // em ponto flutuante
    color.x = i_pos_lightspace.z;
}
```

Figura 4.13: Programa de fragmentos utilizado na geração do *Shadow Map*: recebe um fragmento no espaço da luz (*i_pos_lightspace*), e gera como saída o valor de profundidade (*i_pos_lightspace.z*) do fragmento. Neste caso, a coordenada z do fragmento não é dividida por w de forma a garantir uma representação mais precisa do seu valor.

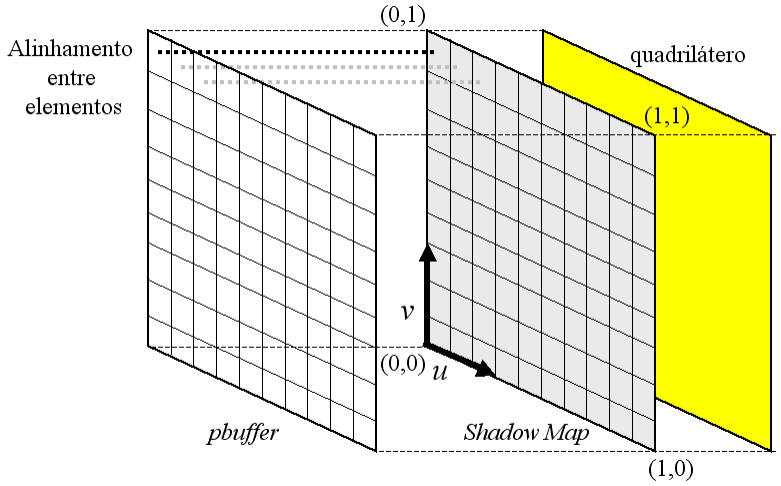


Figura 4.14: Alinhamento entre o *Shadow Map*, mapeado sobre um quadrilátero, e o *pbuffer* para geração do SMMVP.

4.6.3 Segundo Passo: Gerando o SMMVP

O passo anterior apresentou a implementação do *Shadow Map* através de um *pbuffer* como uma forma eficiente de disponibilizar seu conteúdo para a leitura pelo próximo estágio, responsável pela geração do SMMVP. Assim como no caso do *Shadow Map*, o conteúdo do SMMVP também deverá ser acessado pelo estágio seguinte, fazendo com que a opção de implementá-lo através de um *pbuffer* seja imediata.

A seleção dos valores adicionais das células do SMMVP, a partir da amostragem do *Shadow Map*, é um procedimento local, e o padrão de amostragem a ser utilizado pode ser o mesmo para todas as células. Isso faz com que o procedimento de seleção possa ser facilmente mapeado sobre a arquitetura paralela do *hardware*. Para isso temos que forçar a execução de um programa de fragmento, responsável pela seleção dos valores adicionais, para cada elemento do *Shadow Map*. Isso pode ser feito através do mapeamento do *Shadow Map*, como uma textura, sobre um quadrilátero que cubra toda a área do *pbuffer*, e que seja projetado ortogonalmente sobre o mesmo. Cuidado deve ser tomado para que a resolução do *pbuffer* seja a mesma do *Shadow Map*, pois isso garante que cada elemento do *Shadow Map* vai estar alinhado com um *texel* do *pbuffer*. Essa organização pode ser observada na figura 4.14.

O programa de vértices, neste caso, tem a única função de repassar as coordenadas de textura do quadrilátero para o estágio de rasterização para que estas sejam interpoladas e atribuídas aos fragmentos. Não existe a necessidade de se multiplicar os vértices do quadrilátero pela matriz *MODEL_VIEW*, pois, devido à posição da câmera e ao tipo de projeção utilizada (ortogonal), esta corresponde à matriz identidade. A figura 4.15 mostra o programa de vértices utilizado na geração dos valores adicionais do SMMVP.

Uma vez montada essa estrutura, vários tipos de SMMVPs podem ser gerados. As únicas diferenças existentes entre os diferentes tipos de SMMVPs apresentados na seção 4.5 são em relação ao número de amostras feitas nas vizinhanças das células e ao critério de seleção dos valores mais representativos dessas amostras. Como essas tarefas

```
void main(
           // vertice no espaco do objeto
           float4   i_position  : POSITION,
           // coordenada de textura do vertice
           float4   i_tex_coord : TEXCOORD0,

       // vertice gerado como saida
       out float4   o_position   : POSITION,
       // coordenada de textura do vertice
       out float4   o_tex_coord  : TEXCOORD0
)
{
       o_position   = i_position;
       o_tex_coord  = i_tex_coord;
}
```

Figura 4.15: Programa de vértices utilizado na geração dos valores adicionais de profundidade das células do 2-SM. A única função deste programa é a transferência das coordenadas de textura (*i_tex_coord*) do quadrilátero (figura 4.14) para o estágio de rasterização, para que elas sejam interpoladas e passadas aos fragmentos.

ficam a cargo dos programas de fragmentos, a geração de diferentes SMMVPs pode ser obtida através de programas de fragmentos diferentes, que façam amostragens e usem critérios de seleção de acordo com o SMMVP desejado.

A figura 4.16 apresenta um trecho de um programa de fragmentos utilizado na geração de um 2-SM. O programa amostra o valor de z da célula central do *Shadow Map* (alinhada com o *texel* corrente do *pbuffer*), e mais 8 valores adicionais dispostos de forma regular ao seu redor. O cálculo da coordenada de textura dessas 8 amostras adicionais é feito através da adição de um valor de deslocamento à coordenada de textura da célula central. A seguir o programa verifica, dentre essas 9 amostras, a que tem o maior valor, e a define como o valor adicional da célula do 2-SM.

Como o objetivo do algoritmo de SMMVP é diminuir o número de acessos à memória, é necessário que o próximo passo do algoritmo seja capaz de recuperar os 2 valores de profundidade da célula do 2-SM através de uma única operação de busca. Desta forma, como pode ser observado no código (figura 4.16), após a escolha do segundo valor o programa de fragmento gera como saída o valor de profundidade da célula central e o valor adicional selecionado, gravando-os em diferentes canais de cor de um mesmo *texel* do *pbuffer*. Assim, no próximo estágio, uma única operação de leitura será suficiente na obtenção dos 2 valores de profundidade de uma célula do 2-SM.

O programa de fragmento responsável pela geração de um 3-SM segue a mesma idéia apresentada anteriormente. Neste caso, são selecionados o maior e o menor valor de z obtidos a partir de uma amostragem feita ao redor da célula central. Os três valores são posteriormente gravados em 3 canais de cor de um mesmo *texel* do *pbuffer* que representa o 3-SM. Como no caso anterior, o estágio seguinte precisará executar uma única operação de busca para obter os 3 valores de profundidade da célula do 3-SM.

Apesar da evolução que o *hardware* gráfico tem apresentado nos últimos anos, e da

```
. . .
// corrige a interpolacao das coordenadas
// de textura do ípolgono.
i_tex_coord.xy  = i_tex_coord.xy/i_tex_coord.w;
. . .
// amostra 9 valores do Shadow Map
test1.x = f4texRECT(projectiveMap1,samp1).z;
test1.y = f4texRECT(projectiveMap1,samp2).z;
test1.z = f4texRECT(projectiveMap1,samp3).z;

test2.x = f4texRECT(projectiveMap1,samp4).z;
test2.y = f4texRECT(projectiveMap1,i_tex_coord.xy).z; // celula central
test2.z = f4texRECT(projectiveMap1,samp6).z;

test3.x = f4texRECT(projectiveMap1,samp7).z;
test3.y = f4texRECT(projectiveMap1,samp8).z;
test3.z = f4texRECT(projectiveMap1,samp9).z;

// pesquisa o maior valor de z amostrado,
// atraves da regra "maior que"
float greater_z = test1.x;
greater_z = (test1.y>greater_z)?test1.y:greater_z;
greater_z = (test1.z>greater_z)?test1.z:greater_z;

greater_z = (test2.x>greater_z)?test2.x:greater_z;
greater_z = (test2.y>greater_z)?test2.y:greater_z;
greater_z = (test2.z>greater_z)?test2.z:greater_z;

greater_z = (test3.x>greater_z)?test3.x:greater_z;
greater_z = (test3.y>greater_z)?test3.y:greater_z;
greater_z = (test3.z>greater_z)?test3.z:greater_z;

// color.x --> valor do z da celula atual
// color.y --> valor do maior vizinho
color.xyz = float4(test2.y,greater_z,0.0,1.0);
```

Figura 4.16: Trecho do programa de fragmentos utilizado na geração dos valores adicionais de profundidade das células do 2-SM. O programa recebe a coordenada de textura do fragmento (*i_tex_coord*), e a utiliza para fazer a busca de z_c (seção 4.2). Coordenadas de texturas vizinhas a *i_tex_coord* são geradas (*samp1..9*), de forma a possibilitar a amostragem da vizinhança de z_c. Dentre os valores amostrados, o maior é selecionado (*greater_z*), e definido como segundo valor de profundidade da célula.

sua aplicação em diferentes outras áreas que não a computação gráfica, muitas das suas estruturas ainda carregam o legado de suas origens, e isso não é diferente quando se fala em *pbuffers*. Apesar de seus elementos atualmente suportarem representações de grande precisão, o número de canais por elemento dos *pbuffers* ainda continua limitado a 4, e isso decorre naturalmente da tradicional estrutura de representação das cores em computadores. De certa forma isso limita a quantidade de elementos adicionais que um SMMVP pode manter em cada uma de suas células. Um 4-SM poderia ser facilmente armazenado em um único *pbuffer*, onde cada um dos 4 elementos de uma célula do SMMVP ocuparia um dos 4 canais do *texel* do *pbuffer*. SMMVPs de ordem maior ($n > 4$) precisariam ser implementados através do uso de mais de um *pbuffer*. A escrita desses valores em múltiplos *pbuffers* poderia ser eficientemente implementada através da técnica de *Multiple Render Targets* (NVIDIA, 2004c) (ATI, 2005), disponível nas placas mais modernas, mas o custo envolvido na recuperação desses valores no estágio seguinte começaria a inviabilizar sua utilização em aplicações de tempo real.

4.6.4 Terceiro Passo: Gerando a cena final

A seção anterior mostrou como o SMMVP é criado e como seu conteúdo torna-se disponível para leitura através de sua implementação como um *pbuffer*. O terceiro e último passo do algoritmo irá gerar a cena final com *shading*, utilizando a informação contida no SMMVP para o cálculo das sombras.

Nesta última etapa a câmera é posicionada no local a partir de onde será feito o registro da cena final. Cada fragmento gerado pela câmera deverá ser transformado para o sistema de coordenadas da luz e testado contra múltiplas células do SMMVP através de TSCs. Os resultados dos TSCs serão posteriormente filtrados pelo *PCF*, e assim será obtida a atenuação final a ser aplicada sobre a cor do *pixel*.

Como pode ser observado, o último passo do algoritmo, a exemplo do que ocorria nos passos anteriores, baseia-se na execução de operações repetitivas e locais a cada elemento (fragmentos gerados pela cena). Desta forma, optou-se também por implementar essa última etapa de forma que ela pudesse tirar proveito do paralelismo existente no *hardware* gráfico.

A geração da cena final implica na transformação dos vértices da cena, inicialmente no sistema de coordenadas do objeto, para o espaço da tela, sendo isso obtido através de sua multiplicação pela matriz *MODEL_VIEW_PROJ* da câmera. Para que o cálculo da sombra possa ser feito, esses vértices devem também ser projetados sobre o SMMVP. Essa projeção é obtida transformando-se os vértices para o sistema de coordenadas da luz. Isso requer que o programa mantenha informações acerca da *MODEL_VIEW_PROJ* gerada a partir do ponto de vista da luz. O trecho do programa de vértices apresentado na figura 4.17 mostra a transformação de um vértice, inicialmente no sistema de coordenadas do objeto, para o sistema da câmera e também para o sistema da luz.

Os vértices transformados para o espaço da câmera irão compor primitivas geométricas que serão posteriormente rasterizadas e implicarão na interpolação de algumas informações. Entre essas informações estão as coordenadas dos vértices no espaço da luz (salvas no registrador de coordenada de textura *TEXCOORD0*). Esse truque de implemen-

```
void main(
           // vertice no sistema de coordenadas do objeto
           float4   i_position           : POSITION,
           . . .
           out float4   o_position       : POSITION,
           out float4   o_tex_coord_proj : TEXCOORD0,

           // Matriz ViewProj gerada a partir do ponto de
           // vista da camera
           const uniform float4x4 modelViewProj,
           // Matriz Model
           const uniform float4x4 modelView,
           // Matriz ViewProj gerada a partir do ponto de
           //vista da luz
           const uniform float4x4 textureMatrix,
           . . .)
{
  . . .
  // vertice no sistema de coordenadas da camera
  o_position       = mul(modelViewProj, i_position);
  // vertice no sistema de coordenadas da luz
  o_tex_coord_proj = mul(mul(textureMatrix,modelView),i_position);
}
```

Figura 4.17: Trecho do programa de vértices utilizado na geração da imagem final: transforma um vértice, inicialmente no espaço do objeto (*i_position*), para os espaços da câmera (*o_position*) e da luz (*o_tex_coord_proj*). As coordenadas do vértice no espaço da luz são gravadas em um registrador de textura (*TEXCOORD0*), de forma que possam ser interpoladas no momento da rasterização. Isso permite que os fragmentos da cena "saibam"sua posição correspondente no espaço da luz através da consulta ao registrador *TEXCOORD0*.

```
. . .
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef((SMMVP_DIMENSAO-1)/2, (SMMVP_DIMENSAO-1)/2, 0.0);
glScalef((SMMVP_DIMENSAO-1)/2, (SMMVP_DIMENSAO-1)/2, 1.0);
gluPerspective(. . .);
. . .
```

Figura 4.18: Trecho de código C++/*OpenGL*, responsável pela inclusão, na matriz *MODEL_VIEW_PROJ* da fonte de luz, dos cálculos de renormalização das coordenadas dos fragmentos, de forma a garantir que essas fiquem no intervalo $[0.0, <$*dimensão da textura em pixels*$>$*-1*$]$.

tação permite que o processador de fragmentos tenha acesso, através do registrador *TEXCOORD0*, às coordenadas dos fragmentos no espaço da luz. Essas coordenadas, depois de divididas por w, serão utilizadas como coordenadas de textura do fragmento durante o acesso ao SMMVP. Apenas um pequeno detalhe impede a utilização imediata dessas coordenadas como coordenadas de textura dos fragmentos: a divisão por w as coloca no espaço *NDC*, cujo domínio (para qualquer coordenada) abrange o intervalo $[-1.0, 1.0]$. Como o domínio das coordenadas de textura é representado pelo intervalo $[0.0, 1.0]$, uma renormalização desses valores, logo após a divisão por w, torna-se necessária. Dado um valor qualquer $x_{NDC} \in [-1.0, 1.0]$, a sua renormalização para o intervalo $[0.0, 1.0]$ pode ser feita da seguinte forma:

$$x_{textura} = x_{NDC} \times 0.5 + 0.5 \tag{4.3}$$

O fato de o SMMVP ter sido implementado através de um *pbuffer* com precisão de 32bits em ponto flutuante por canal implica um pequeno ajuste na equação apresentada acima. Por determinação dos fabricantes do *hardware* as coordenadas de textura para texturas de ponto flutuante não ficam limitadas ao intervalo $[0.0, 1.0]$. Neste caso, as suas coordenadas de textura podem assumir valores no intervalo:

$$[0.0, <\textit{dimensão da textura em pixels}>\textit{-1}]$$

Desta forma, a equação de renormalização deve ser reescrita levando esse fato em consideração:

$$x_{textura} = x_{NDC} \times (<\textit{dimensão}> - 1)/2 + (<\textit{dimensão}> - 1)/2 \tag{4.4}$$

Sendo esta renormalização uma operação necessária a todos os fragmentos gerados pela câmera, optou-se por incluí-la diretamente na matriz *MODEL_VIEW_PROJ* da fonte de luz. Desta forma pode-se diminuir o número de instruções a serem executadas pelo programa de fragmento. A figura 4.18 apresenta o trecho de código C++/*OpenGL* que inclui esta renormalização na *MODEL_VIEW_PROJ* da fonte de luz (ao fim de todas as outras transformações). Observe que apenas as coordenadas x e y dos vértices, utilizadas na indexação do SMMVP, são afetadas. A coordenada z não deve ser alterada pois será utilizada mais tarde no TSC.

As operações de transformação e renormalização de coordenadas vistas até agora nesta seção garantem a correta projeção dos fragmentos gerados pela câmera sobre o

SMMVP. Basta agora que o programa de fragmentos recupere os valores das células do SMMVP, execute os TSCs sobre esses valores e filtre os resultados através do *PCF*. A figura 4.19 apresenta o trecho de código de um programa de fragmentos que gera a imagem final com sombras. Inicialmente 9 amostras são obtidas a partir do 2-SM (totalizando 18 valores de profundidade). O valor de profundidade do ponto da cena, no espaço da luz (*i_tex_coord_proj.z*), será testado, através de 2-TSs, contra os 9 pares de valores de profundidade amostrados. Os resultados desses testes alimentarão um filtro *PCF*(9,2), que retornará o valor final de atenuação a ser aplicada sobre a cor do fragmento.

Os resultados de cada 2-TS, e do *PCF*(9,2), são obtidos através do cálculo de médias, que implicam em divisões. De forma a evitar a execução de divisões (embora, em alguns casos, o próprio compilador *Cg* resolva isso), optou-se por implementar o cálculo da filtragem através de multiplicações. Como se está trabalhando na realidade com 18 valores de profundidade, a contribuição de cada um desses valores, para o resultado final de atenuação, será de $1/18 = 0.0555555$. Desta forma, cada teste de sombra feito com um dos 18 valores de z do 2-SM pode retornar 0.0 ou 0.0555555. A posterior soma desses resultados será equivalente ao resultado obtido com o *PCF*(9,2), e determinará a atenuação final da cor do fragmento.

Ao final da execução do programa da figura 4.19, a variável *att* contém o valor de atenuação (limitado ao intervalo $[0.0, 1.0]$) a ser aplicado sobre a cor do fragmento. O exemplo acima ilustrou o cálculo para a atenuação de um fragmento a partir de um 2-SM. Esse programa de fragmentos pode ser facilmente estendido para implementar a atenuação causada por um 3-SM. Neste caso, o trecho referente à obtenção dos valores a partir do SMMVP deve buscar os valores de profundidade armazenados em 3 canais de cor (*xyz*) ao invés de apenas dois (*xy*), como aparece no código atual. O número de elementos do filtro do *PCF* também podem ser facilmente alterado. Para isto, basta mudar o número de consultas ao SMMVP (de acordo com o número de elementos do filtro), o número de TSCs a serem feitos e o valor de contribuição de cada teste de sombra. A seção seguinte apresenta os resultados obtidos através da implementação do SMMVP.

```
// recupera os valores de 9 celulas do 2-SM
// (18 valores de profundidade no total)
test1.xy = f4texRECT(projectiveMap2,samp1).xy;
test1.zw = f4texRECT(projectiveMap2,samp2).xy;
test2.xy = f4texRECT(projectiveMap2,samp3).xy;

test2.zw = f4texRECT(projectiveMap2,samp4).xy;
test3.xy = f4texRECT(projectiveMap2,samp5).xy; // central
test3.zw = f4texRECT(projectiveMap2,samp6).xy;

test4.xy = f4texRECT(projectiveMap2,samp7).xy;
test4.zw = f4texRECT(projectiveMap2,samp8).xy;
test5.xy = f4texRECT(projectiveMap2,samp9).xy;

// executa os 2-TSs e a filtragem
test1 = (test1 > (i_tex_coord_proj.zzzz - 0.075)) ?
        float4(0.0555555,0.0555555,0.0555555,0.0555555) :
        float4(0.0,0.0,0.0,0.0);
test2 = (test2 > (i_tex_coord_proj.zzzz - 0.075)) ?
        float4(0.0555555,0.0555555,0.0555555,0.0555555) :
        float4(0.0,0.0,0.0,0.0);
test3 = (test3 > (i_tex_coord_proj.zzzz - 0.075)) ?
        float4(0.0555555,0.0555555,0.0555555,0.0555555) :
        float4(0.0,0.0,0.0,0.0);
test4 = (test4 > (i_tex_coord_proj.zzzz - 0.075)) ?
        float4(0.0555555,0.0555555,0.0555555,0.0555555) :
        float4(0.0,0.0,0.0,0.0);
test5 = (test5 > (i_tex_coord_proj.zzzz - 0.075)) ?
        float4(0.0555555,0.0555555,0.0     ,0.0     ) :
        float4(0.0,0.0,0.0,0.0);

test1.x = dot(float4(1.0,1.0,1.0,1.0), test1);
test2.x = dot(float4(1.0,1.0,1.0,1.0), test2);
test3.x = dot(float4(1.0,1.0,1.0,1.0), test3);
test4.x = dot(float4(1.0,1.0,1.0,1.0), test4);
test5.x = dot(float4(1.0,1.0,0.0,0.0), test5);

// soma os resultados dos testes individuais
// e calcula o valor de atenuacao final do pixel
float att = test1.x + test2.x + test3.x + test4.x + test5.x;
. . .
```

Figura 4.19: Trecho do programa de fragmentos que gera a imagem final, com sombras. Inicialmente são obtidas 9 amostras a partir do 2-SM (18 valores de profundidade). A seguir cada um dos 18 elementos participa de um teste de sombra simples, onde o resultado pode ser 0.0 ou $1/18$ (0.0555555). Desta forma o resultado dos 2-TSs e do *PCF*(9,2) acabam sendo integrados em um mesmo procedimento, e divisões (necessárias aos cálculos de média dos 2-TS e *PCF*) são substituídas por multiplicações. Por fim, a soma dos resultados dos testes de sombra individuais determina o valor final de atenuação a ser aplicado à cor do fragmento.

5 RESULTADOS E ANÁLISE

5.1 Resultados

Para comparar os resultados do nosso método àqueles obtidos através do *Shadow Mapping* tradicional combinado com o *PCF*, quatro cenas de teste foram escolhidas. Estas cenas permitem a avaliação dos resultados visuais possibilitados pelo novo algoritmo quando aplicado a cenas com configurações geométricas distintas.

A cena da figura 5.1a apresenta tanto variações suaves quanto variações mais abruptas de geometria, enquanto que a figura 5.1b apresenta uma configuração geométrica bastante complexa, composta por variações repentinas e sobreposição de elementos. A cena da figura 5.1c gera discontinuidades alongadas e orientadas nos valores do *Shadow Map*. Por fim, a figura 5.1d apresenta uma cena onde prevalecem elementos geométricos curvos e de variação suave.

As figuras 5.2(a,b) apresentam detalhes dos resultados obtidos através da aplicação dos algoritmo de *Shadow Mapping* e 2-SM sobre a figura 5.1a. Nestas duas imagens, a cor dos *pixels* está sendo modulada diretamente pelos resultados dos testes de sombra, sem filtragem através de *PCF*. Os quadros desenhados sobre as imagens representam os resultados dos testes de sombra e TSCs para pequenas regiões (3x3 *pixels*) das imagens. Observa-se que enquanto o *Shadow Mapping* apresenta apenas resultados binários para os testes de sombra, os TSCs aplicados sobre o 2-SM são capazes de retornar um terceiro valor de atenuação $(0,5)$ em regiões de fronteira de sombra.

As figuras 5.2(c,d) apresentam em detalhe os resultados da aplicação de filtros *PCF* sobre as sombras das imagens 5.2(a,b). A imagem da figura 5.2c apresenta o resultado da filtragem das sombras da figura 5.2a através de um filtro *PCF* de 16 (4x4) amostras, enquanto que a imagem da figura 5.2d mostra o resultado da filtragem das sombras da figura 5.2b através de um *PCF* de 9 (3x3) amostras. Pode-se observar que o nível de suavização obtido nas bordas das sombras das duas imagens são muito semelhantes, embora haja uma diferença considerável no número de elementos dos filtros *PCF* utilizados em cada uma. Como pode ser observado, os resultados visuais obtidos através da filtragem das sombras com o SMMVP, e *PCFs* de dimensões menores, podem ser comparáveis aos obtidos com o *Shadow Mapping* tradicional e *PCF*s de dimensões maiores.

A figura 5.3 apresenta em detalhe as projeção de sombras geradas através de diferentes combinações de SMMVP e *PCFs* para uma cena com variações repentinas na geometria. Neste exemplo são ilustrados os resultados obtidos através do uso do *Shadow Mapping*

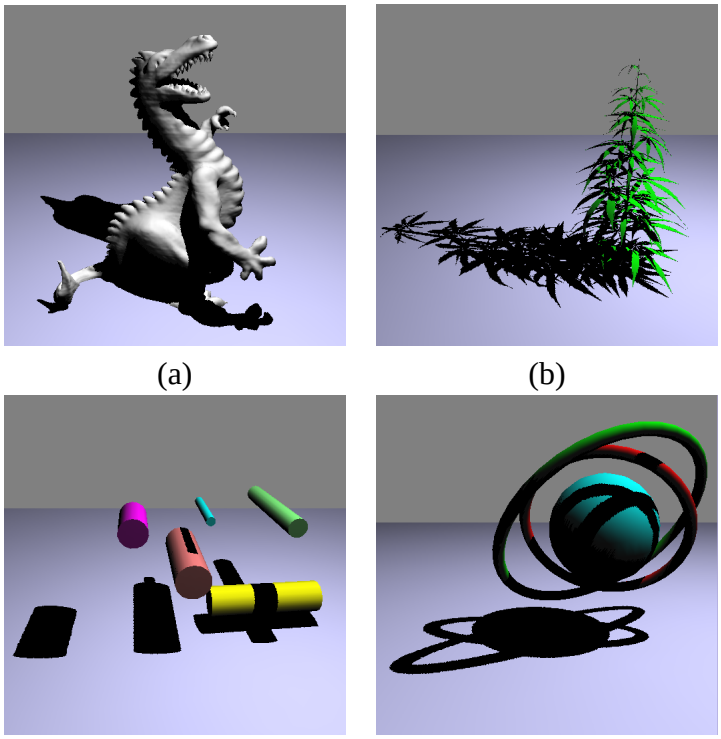


Figura 5.1: Cenas utilizadas para o teste do algoritmo de SMMVP: (a) cena com variações suaves e repentinas na geometria, (b) cena complexa com variações abruptas, (c) cena que gera discontinuidades alongadas e orientadas nos valores do *Shadow Map*, (d) cena com elementos geométricos curvos e de variação suave.

tradicional em conjunto com *PCF*s de 9 (3x3) e 16 (4x4) amostras, bem como os resultados obtidos com a utilização de 2-SMs e 3-SMs com *PCFs* de dimensões menores.

Na figura 5.4 apresentamos sombras projetadas sobre superfícies curvas, e comparamos o uso do *Shadow Mapping* tradicional com um *PCF* de 4 (2x2) amostras contra um 2-SM e um 3-SM utilizados em conjunto com *PCFs* de 9 (3x3) amostras. Como visto anteriormente, os resultados de filtragem são bastante similares, demonstrando a eficácia do SMMVP em garantir, mesmo com o uso de *PCFs* de menores dimensões, níveis de suavização equivalentes aos obtidos com o *Shadow Mapping* tradicional em conjunto com *PCFs* de maiores dimensões.

5.2 Discussão

Esta seção trata de discussões a respeito dos SMMVPs. Inicialmente é feita uma breve discussão sobre os efeitos dos tamanhos dos filtros utilizados com o SMMVP. A seguir uma avaliação teórica acerca dos custos envolvidos na técnica de SMMVP é apresentada. Uma comparação entre seus custos e os envolvidos no *Shadow Mapping*, em conjunto com *PCF*, também é feita. Na seqüência são apresentados dados de desempenho obtidos através de uma aplicação construída especificamente para este fim. Por fim, é apresentada uma breve discussão a respeito de uma possível implementação do algoritmo de SMMVP

em *hardware*.

5.2.1 Tamanhos dos filtros

Como pode ser observado na seção 5.1, a utilização de SMMVPs, em conjunto com *PCFs* de pequenas dimensões, permite a obtenção de níveis de filtragem das bordas de sombra visualmente equivalentes àqueles obtidos com o *Shadow Mapping* tradicional e *PCFs* de dimensões maiores. O fato dos filtros utilizados em conjunto com os SMMVPs terem dimensões menores também tem implicações nos resultados finais, como será visto a seguir.

Observando-se a figura 5.3: a imagem da figura 5.3a foi gerada através de *Shadow Mapping* tradicional, em conjunto com um *PCF* de 9 amostras, e a imagem da figura 5.3c foi gerada com um 3-SM, em conjunto com um *PCF* de 4 amostras. Segundo a equação 4.2, o número de tons de atenuação possivelmente gerados para a imagem 5.3a é de no máximo 10 tons, enquanto que para a imagem 5.3c, o número de tons de atenuação é no máximo 13. Isso explica a semelhança entre os níveis de filtragem obtidos para as duas imagens. Agora, se forem observados os cantos inferiores esquerdos dessas duas imagens, pode-se notar que existem pequenas regiões claras, sem sombra, que são percebidas na imagem 5.3c e que praticamente inexistem na imagem 5.3a. Isso se deve ao tamanho dos filtros utilizados. Embora ambos possam gerar praticamente o mesmo número de tons de atenuação, os seus tamanhos diferem. O filtro utilizado na imagem 5.3a, por ser maior (3x3 amostras), acaba afetando a cor de *pixels* que se encontram mais distantes das bordas de sombra, enquanto que o filtro utilizado na imagem 5.3c, por ser menor, tem efeito mais localizado (mais próximo às bordas de sombra).

5.2.2 O custo envolvido na geração de cenas com SMMVP

As vantagens propiciadas pelo uso de SMMVPs são obtidas ao custo da seleção e armazenamento dos múltiplos valores de profundidade adicionais das células do SMMVP. Deve-se considerar também neste custo a execução dos TSCs, mais complexos que os testes de sombra tradicionais.

Apresentado desta forma, pode parecer que simplesmente transferimos parte do custo relacionado à filtragem feita pelo *PCF* para a construção dos SMMVPs. Entretanto, uma análise mais cuidadosa mostra que, para determinados tipos de cenas, os custos relacionados ao SMMVP, em conjunto com *PCF*, são menores do que os relacionados ao *Shadow Mapping* tradicional em conjunto com o *PCF*. De forma a facilitar essa análise, começaremos pelo cálculo do custo envolvido na geração de uma cena com *Shadow Mapping* e *PCF*, e então expandiremos essa idéia para o caso das cenas geradas com SMMVP.

O custo de geração de uma cena com *Shadow Mapping* está relacionado a:

1. número de fragmentos gerados pela cena (f)
2. custo de transformação dos fragmentos da cena para o espaço da luz ($custo_{transf}$)
3. acesso à textura para obtenção do valor de profundidade do *Shadow Map* ($custo_{textura}$)

4. custo do teste de sombra ($custo_{ts}$)

Desta forma, podemos expressar o custo envolvido na geração de uma cena com *Shadow Mapping* ($custo_{cena\_sm}$) através da fórmula a seguir:

$$custo_{cena\_sm} = f \times (custo_{transf} + custo_{textura} + custo_{ts}) \tag{5.1}$$

O uso do algoritmo de *PCF* implica na obtenção de um número maior de valores de profundidade a partir do *Shadow Map*, e também em um aumento correspondente no número de testes de sombra a serem feitos por fragmento. Isso significa que a utilização de um filtro *PCF* de k elementos implica um aumento, por um fator igual a k, no número de acessos a textura e de testes de sombra. Isso pode ser expresso da seguinte forma:

$$custo_{cena\_sm} = f \times (custo_{transf} + k \times (custo_{textura} + custo_{ts})) \tag{5.2}$$

Instruções de acesso a textura apresentam um custo significativo, correspondendo, na melhor hipótese, ao mesmo custo que a execução de uma operação matemática. Desta forma optamos por reescrever a equação anterior de forma a expressar o custo do *Shadow Mapping* apenas em termos de acessos à textura ($custo_{cena\_sm\_textura}$):

$$custo_{cena\_sm\_textura} = f \times k \tag{5.3}$$

Como visto na seção 4.6, o algoritmo de SMMVP conta com apenas um passo adicional, se comparado ao *Shadow Mapping*, responsável pela seleção dos valores adicionais de profundidade de suas células. A seleção destes valores é feita através da amostragem das vizinhanças de cada uma das células do *Shadow Map*. Supondo que as dimensões do *Shadow Map* são expressas por h_{sm} e w_{sm}, e que os valores adicionais de uma célula são obtidos a partir de q amostras feitas em sua vizinhança, o custo relacionado à geração do SMMVP ($custo_{smmvp}$), em termos de acesso à textura, pode ser descrito como:

$$custo_{smmvp} = h_{sm} \times w_{sm} \times q \tag{5.4}$$

O custo relacionado à geração de uma cena através do uso de SMMVP em conjunto com *PCF* pode agora ser expressa como o custo da geração do SMMVP (equação 5.4) acrescido do custo de geração da cena final com sombras (equação 5.3):

$$custo_{cena\_smmvp} = h_{sm} \times w_{sm} \times q + f \times k \tag{5.5}$$

O número de fragmentos (f) gerados por uma cena é arbitrário. Se a cena apresenta muitos polígonos sobrepostos, o número de fragmentos que ela pode gerar pode ser muito maior que a própria resolução da tela[1]. Por outro lado, o número de fragmentos gerados durante a seleção dos valores adicionais do SMMVP (seção 4.6.3) é fixo e igual à resolução do *Shadow Map*. Desta forma, pode-se concluir que, à medida que o número de fragmentos gerados por uma determinada cena aumenta, existirá um ponto onde o custo de um *PCF*(k+r, 1)[2], onde $r \in \mathbb{N}^*$, será maior que o custo de um *PCF*(k, 2), podendo esse ponto ser expresso como:

[1] A quantidade de fragmentos gerados pela cena é influenciado pela quantidade de sobreposições de polígonos na cena, da ordem em que esses polígonos são enviados para o *pipeline* e também do status do *depth test* (habilitado ou não.)

[2] Ver a seção 4.4 para maiores detalhes sobre esta notação.

$$f > \frac{q \times h_{sm} \times w_{sm}}{r} \tag{5.6}$$

Por exemplo, considere as resoluções utilizadas em nossos testes (512^2 tanto para a imagem final quanto para o *Shadow Map*), e compare o custo do *Shadow Mapping* tradicional usando um *PCF* com 9 elementos (3x3) contra o custo de um 2-SM em conjunto com um *PCF* com 4 elementos (2x2). Sendo q definido como 9, e r igual a $9 - 4 = 5$, pode-se verificar que o 2-SM representará um custo menor quando $f > 1,8 \times 512^2$. Este valor representa um número de fragmentos maior do que a resolução da imagem final, podendo ocorrer facilmente em cenas de maior complexidade.

Alguém pode argumentar que a visibilidade a partir do ponto de vista da câmera pode ser resolvida antes que se iniciem os cálculos referentes à geração das sombras (*shading* postergado (DEERING et al., 1988). Supondo uma cena que cubra toda a tela, podemos assumir que o cálculo da sombra vai ser executado apenas uma vez para cada fragmento gerado ($f = h_s \times w_s$), onde h_s e w_s representam as dimensões da tela. Se expressarmos $h_s = \lambda_h \times h_{sm}$ e $w_s = \lambda_w \times w_{sm}$ como valores escalados das dimensões do *Shadow Map*, o uso do SMMVP torna-se mais barato que o *Shadow Mapping* tradicional com *PCF* de dimensão maior quando:

$$\lambda_h \times \lambda_w > \frac{q}{r} \tag{5.7}$$

Se a imagem final e o *Shadow Map* possuem as mesmas dimensões, esta desigualdade nunca será satisfeita. Entretanto não é incomum a utilização de *Shadow Maps* com resoluções menores do que a da imagem final, o que viabiliza a utilização do algoritmo de SMMVP também nesses casos.

5.2.3 O desempenho na prática

De maneira a poder verificar o desempenho do algoritmo na prática, uma aplicação adicional foi desenvolvida. Esta aplicação conta com a implementação de todos os programas de vértices e de fragmentos responsáveis por gerar o *Shadow Map*, executar as filtragens do *PCF* e gerar os valores adicionais do SMMVP. Como a avaliação se baseia nas operações executadas para cada fragmento da cena, a aplicação procurou saturar o processador de fragmentos, sem sobrecarregar o processador de vértices. Uma maneira de se obter essa saturação da unidade de fragmentos, a sobrecarga da unidade de vértices, é através da renderização de cenas compostas por quadriláteros (descritos por apenas 4 vértices) que cubram grandes porções da tela.

O tipo de SMMVP escolhido para teste foi o 2-SM. A resolução escolhida para a imagem final, *Shadow Map* e 2-SM foi de 512^2 *pixels*. O filtro *PCF* utilizado em conjunto com o *Shadow Mapping* possui 9 elementos (3x3), enquanto que o *PCF* utilizado com o 2-SM tem 4 elementos (2x2).

A aplicação começa renderizando um pequeno quadrilátero, que cobre apenas a região inferior da tela, coletando ao final dados referentes ao tempo total de renderização da cena (geração de *Shadow Map*, seleção dos valores adicionais do 2-SM e aplicação de *PCF*). A seguir esse quadrilátero é aumentado, cobrindo uma região maior da tela e gerando mais fragmentos. Novas medições de tempo são feitas. Assim que o primeiro quadrilátero passa a cobrir toda a tela, um segundo quadrilátero é adicionado à cena, de

forma a aumentar o número de fragmentos gerados. A estrutura da cena utilizada pela aplicação pode ser observada na figura 5.5. A figura 5.6 apresenta um gráfico que mostra o tempo de rendering da cena (em segundos) como uma função do número de fragmentos. Esses são os resultados obtidos pela execução da aplicação em uma placa *GeForceFX 5800* da NVIDIA.

O número de fragmentos, no ponto de cruzamento dos dois gráficos, não coincide exatamente com o valor obtido no exemplo teórico apresentado. Isso pode ser explicado pela não inclusão, nas fórmulas, dos custos referentes às operações matemáticas e lógicas requeridas pelos algoritmos. A presença de *caches* na *GPU*[3] também pode fazer com que os custos de algumas operações sejam variáveis, resultando em desvios dos valores obtidos na prática em relação aos valores teóricos. O importante é que o comportamento do sistema foi o esperado.

A figura 5.7 apresenta uma ampliação da figura 5.6, próxima à origem dos eixos. Pode-se observar nesta imagem que o gráfico referente ao custo do *Shadow Mapping* tradicional começa próximo de 0 (zero). Isso se deve ao fato de inicialmente a cena não gerar nenhum fragmento. À medida que os quadriláteros vão aumentando de tamanho, o número de fragmentos aumenta, e os acessos ao *Shadow Map* também. Diferentemente do *Shadow Mapping*, o custo inicial do 2-SM não inicia em 0 (zero). Isso ocorre pois a seleção dos valores adicionais das células do SMMVP é um pré-processamento que, independentemente do número de fragmentos gerados pela cena, sempre será feito. Apesar do custo da geração do 2-SM estar sempre presente, ele é fixo. Além do mais, o número de acessos a textura gerado pelos fragmentos da cena será menor do que o gerado pelo *Shadow Mapping*, pois o *PCF* utilizado em conjunto com o 2-SM possui dimensão menor. Isso significa que à medida que o número de fragmentos da cena aumentar, o ganho de performance obtido pelo uso de um *PCF* menor irá compensar o custo fixo relacionado à geração do 2-SM, tornando-o mais eficiente que o próprio algoritmo de *Shadow Mapping* com *PCF*.

A figura 5.8 mostra os resultados obtidos com o uso de uma placa *GeForceGT 6800* da NVIDIA – as escalas utilizadas para os eixos x (fragmentos) e y (tempo em segundos) foram as mesmas da figura 5.6, de forma a permitir ao leitor uma comparação aproximada do desempenho dos algoritmos nas duas placas. Como se pode observar, o custo fixo relacionado à geração dos valores adicionais do 2-SM é menor do que o obtido para a *GeForceFX 5800*. A taxa de aumento dos valores de tempo, em função do número de fragmentos, para os dois gráficos (*Shadow Mapping* e 2-SM) também é menor. Isso se deve ao fato desta placa possuir um número maior de unidades de processamento de fragmentos do que a *GeForceFX 5800*.

Pode-se observar também que, embora os dois gráficos da figura estejam convergindo, indicando que se cruzarão mais adiante (como previsto), esse cruzamento não ocorre. Uma das razões para este comportamento pode ser atribuído à inclusão de mais um nível de *cache* de textura na arquitetura da *GeForceGT 6800* (TOMSHARDWARE, 2004). Enquanto a *GeForceFX 5800* conta com somente um nível de *cache* de textura, a *GeForceGT 6800* conta com dois níveis. Essa organização hierárquica dos *caches* pode reduzir o tempo médio de acesso à textura, a ponto de (como dito pela própria NVIDIA (NVIDIA,

[3] *Caches* de textura, de vértices, e de fragmentos.

2004c)), igualá-lo, em alguns casos, ao tempo de execução de uma instrução aritmética. Dado que os algoritmos de *Shadow Mapping*, *PCF* e o SMMVP fazem acessos coerentes às texturas, todos se beneficiam fortemente dessa nova política de *cache*. Essa nova realidade vai de encontro com a premissa assumida no momento da definição da fórmula 5.5, de que o acesso à textura tinha um custo tão maior do que o das operações lógicas e aritméticas, que justificaria a eliminações destas últimas da fórmula.

5.2.4 Implementação em *hardware*

Embora esteja claro que o hardware está evoluindo em direção a uma arquitetura cada vez mais geral, o SMMVP apresenta uma estrutura que possibilitaria sua eventual implementação em *hardware*. No caso de cenas estáticas (somente com movimentos de câmera), o SMMVP pode ser calculado apenas uma única vez, não havendo necessidade de nova seleção de valores adicionais para suas células. Para cenas dinâmicas, seria vantajosa a presença de um *hardware* dedicado para seleção dos valores adicionais das células (o mínimo ou máximo valor vizinho de cada célula do *Shadow Map*).

Uma outra possibilidade seria considerar a utilização do *hardware* de oclusão existente atualmente nas placas. A technologia *Hyper-Z* (ATI, 2005) da *ATi* mantém o menor valor de profundidade para cada grupo de 4 e 16 elementos do *Shadow Map*. Alguns experimentos, no sentido de simular o acesso a essa estrutura hierárquica, permitiram a observação dos resultados visuais desta abordagem. Embora o número de tons de atenuação nas bordas de sombra tenha realmente aumentado, o fato das amostras do *Hyper-Z* serem obtidas a partir de uma subdivisão regular do *Shadow Map*, acabaram por gerar efeitos de *aliasing* bastante severos.

Embora uma seleção mais eficiente dos valores adicionais do SMMVP implique eventualmente em alguma alteração de *hardware*, o armazenamento de um 2-SM, 3-SM ou mesmo um 4-SM não precisam de nenhuma modificação. Os formatos de textura encontrados no *hardware* atual permitem a utilização de até 4 canais por *texel*, permitindo o armazenamento de até 4 valores de profundidade, que seriam eficientemente lidos através de uma única operação de leitura.

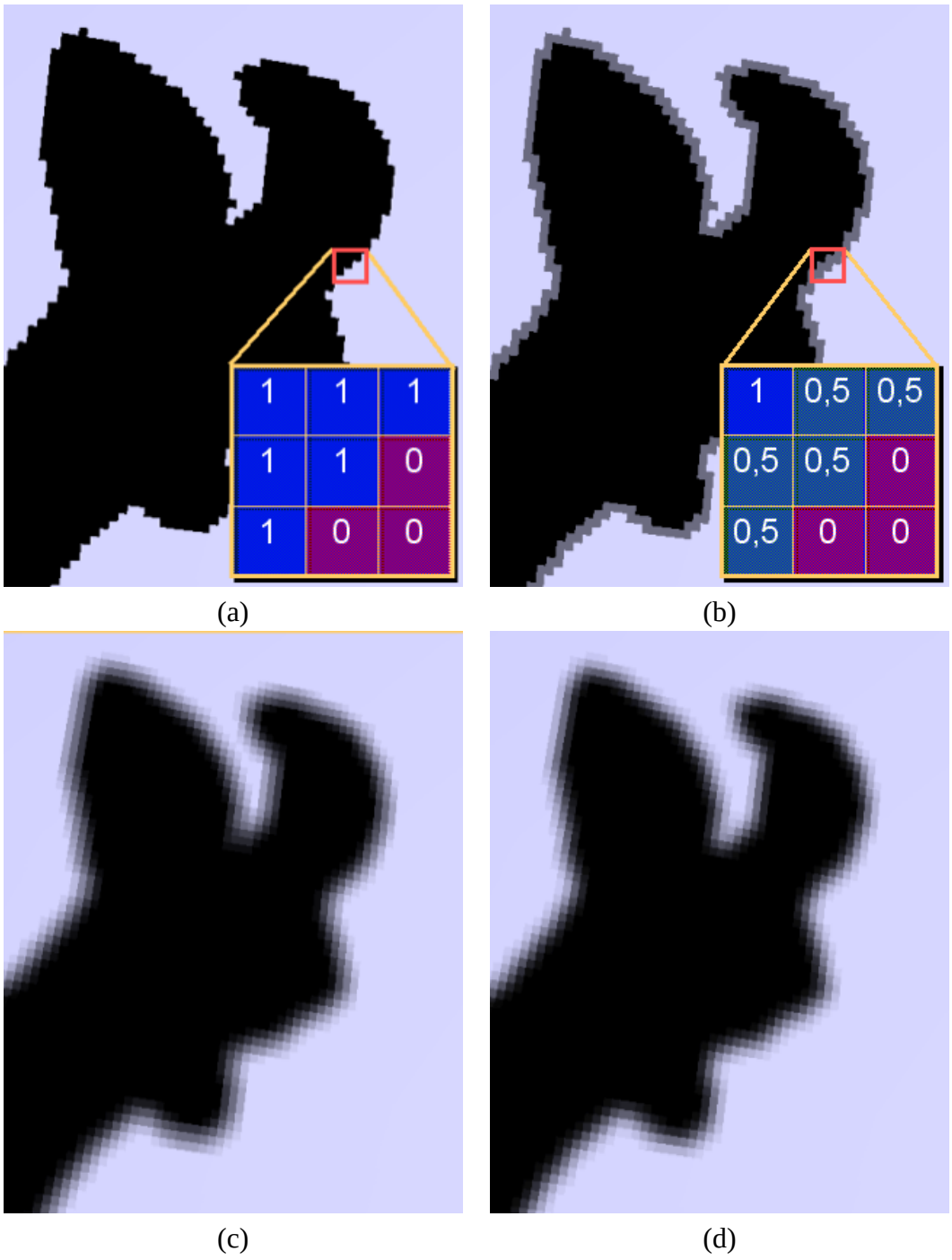
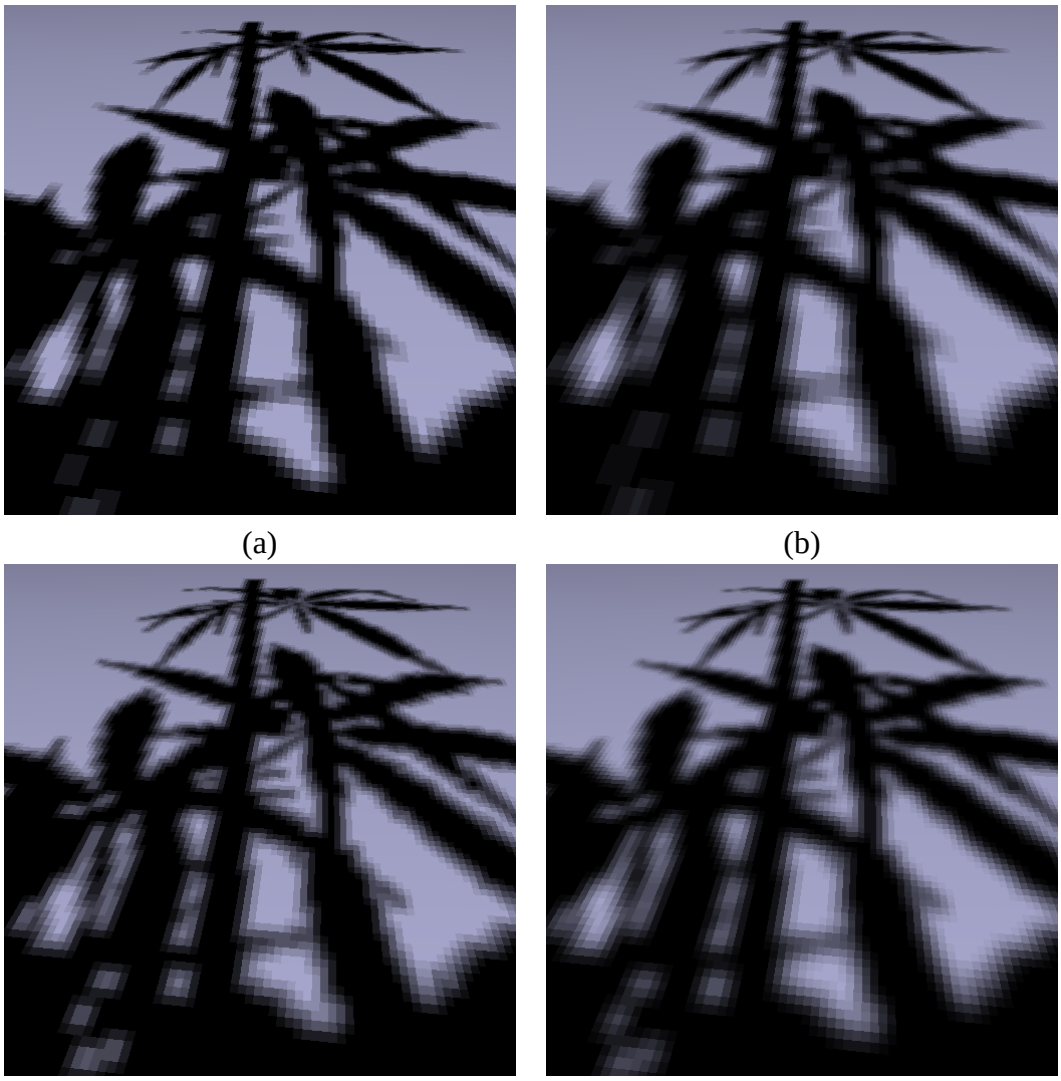


Figura 5.2: (a) Cena com cores moduladas de acordo com o resultados de testes de sombra feitos contra um *Shadow Map* tradicional. (b) Cena com cores moduladas de acordo com o resultado de TSCs feitos contra um 2-SM. Em nenhuma destas cenas (a e b) os resultados dos testes de sombra, ou TSCs, foram filtrados através de *PCF*. Observa-se que os TSCs são capazes de retornar um número maior de valores de atenuação se comparados a um teste de sombra tradicional. (c) Resultado da filtragem das sombras da figura *a* através de um filtro *PCF* de 16 amostras (4x4). (d) Resultado da filtragem das sombras da figura *b* através de um filtro *PCF* de 9 amostras (3x3). Observe a semelhança entre as imagens *c* e *d*, mesmo tendo sido filtradas com filtros de dimensões diferentes.



(a) (b)

(c) (d)

Figura 5.3: Comparação entre os resultados obtidos com o algoritmo de *Shadow Mapping* tradicional em conjunto com o *PCF*, e os obtidos com o SMMVP. (a) e (b) *Shadow Mapping* tradicional utilizando *PCFs* com 9 (3x3) e 16 (4x4) elementos respectivamente. (c) e (d) 3-SM utilizando *PCFs* com 4 (2x2) and 9 (3x3) elementos respectivamente. Novamente, pode-se observar que os resultados obtidos com o SMMVP são comparáveis aos obtidos com o *Shadow Mapping* tradicional (compare (a) com (c) e (b) com (d)).

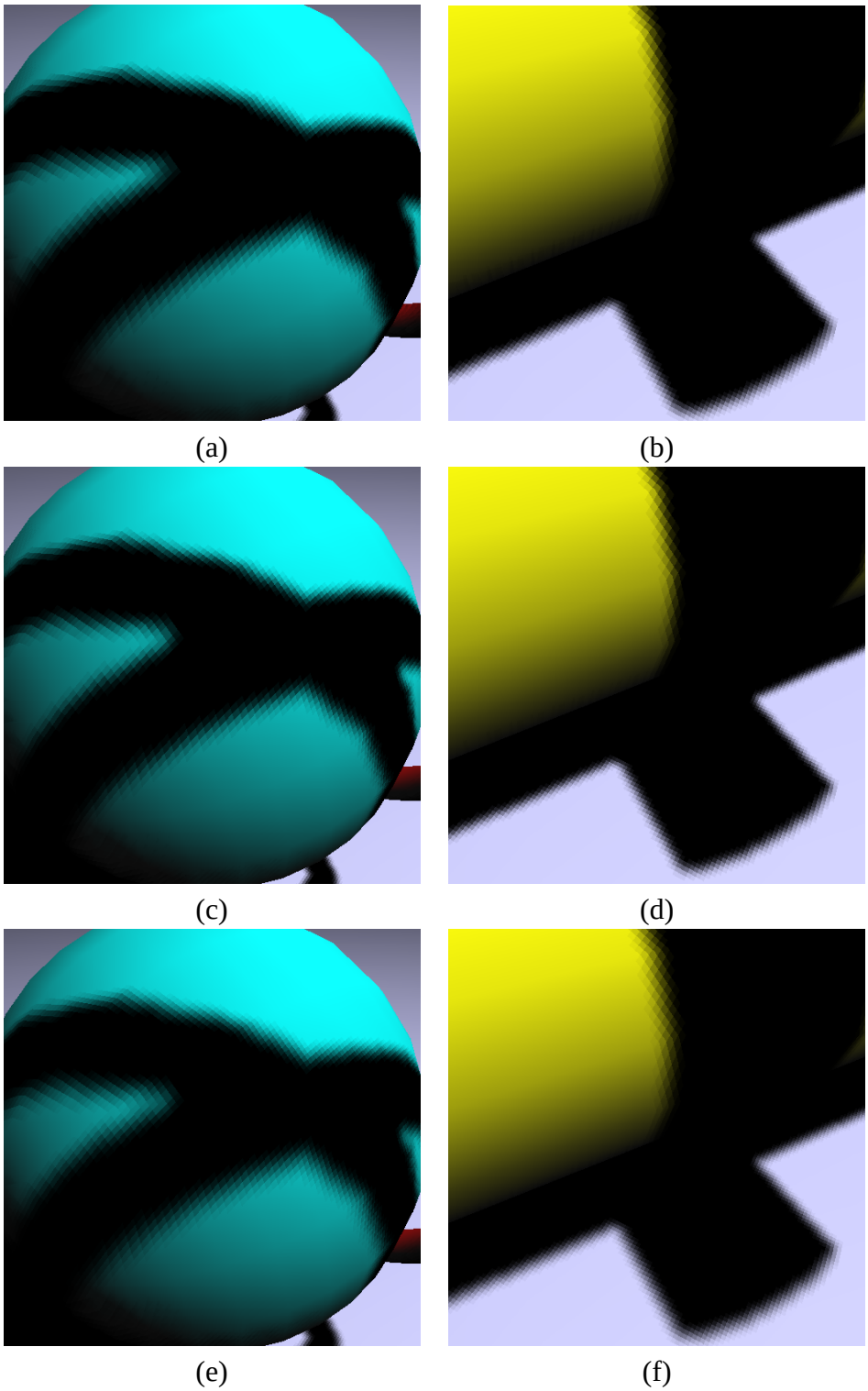


Figura 5.4: Comparação entre o *Shadow Mapping* tradicional e SMMVPs para o caso de sombras com bordas curvas. (a)(b) *Shadow Mapping* tradicional com *PCF* de 16 amostras (4x4). (c)(d) 2-SM com *PCF* de 9 amostras (3x3). (e)(f) 3-SM com *PCF* de 9 amostras (3x3).

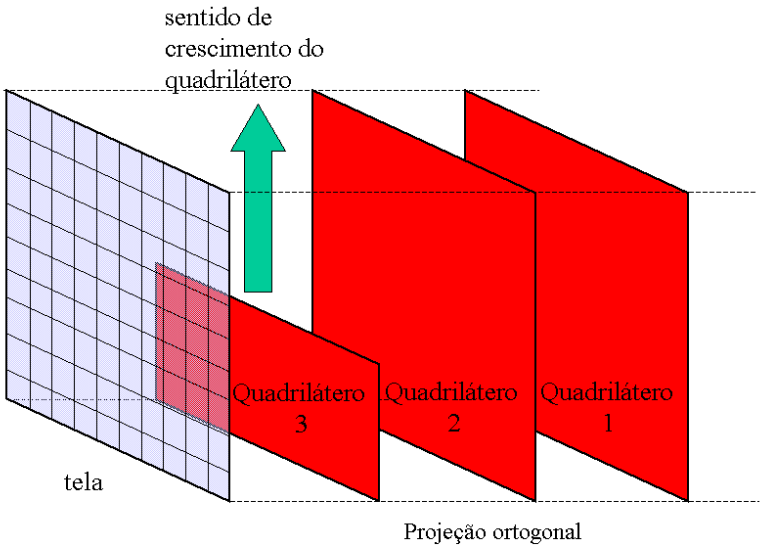


Figura 5.5: A figura acima apresenta a estrutura da cena utilizada nos testes de desempenho do algoritmo. Quadriláteros são projetados ortogonalmente sobre tela, de forma a gerarem um número elevado de fragmentos sem sobrecarregarem a unidade de vértices. A cena acima mostra o momento em que dois quadriláteros (1 e 2) já ocupam toda a área da tela enquanto que um terceiro (quadrilátero 3) é adicionado. Os quadriláteros são desenhados sempre de trás para a frente, de forma que o teste precoce de z não elimine os seus fragmentos do *pipeline*.

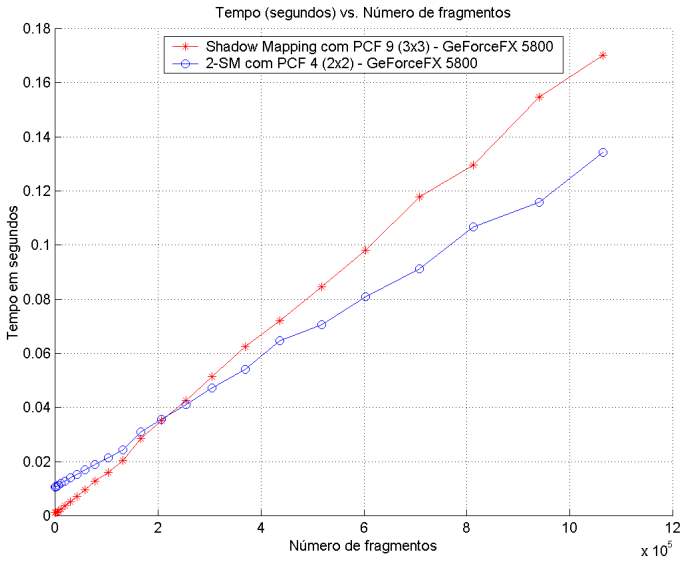


Figura 5.6: A figura apresenta a comparação entre os custos de geração de uma cena através dos algoritmos de *Shadow Mapping* tradicional, com um *PCF* de 9 elementos, e o 2-SM, com *PCF* de 4 elementos, em função do número de fragmentos da cena. A resolução da imagem final, do *Shadow Map* e do 2-SM é de 512^2 *pixels*. Medições feitas com uma placa *GeForceFX 5800*. Pode-se observar o ponto (cruzamento entre as linhas) a partir do qual o algoritmo de SMMVP torna-se mais eficiente.

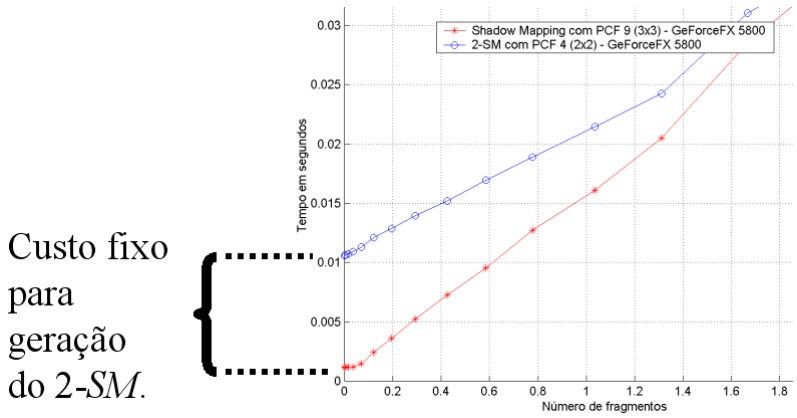


Figura 5.7: Detalhe (obtido a partir da ampliação da figura 5.6) do custo inicial gerado pelo *Shadow Mapping* e pelo 2-SM. O *Shadow Mapping* tem um custo inicial praticamente nulo, pois este é dependente do número de fragmentos gerados pela cena. O custo inicial do 2-SM parte de um valor maior, pois conta com o custo fixo relacionado ao pré-processamento responsável pela geração dos valores adicionais de suas células.

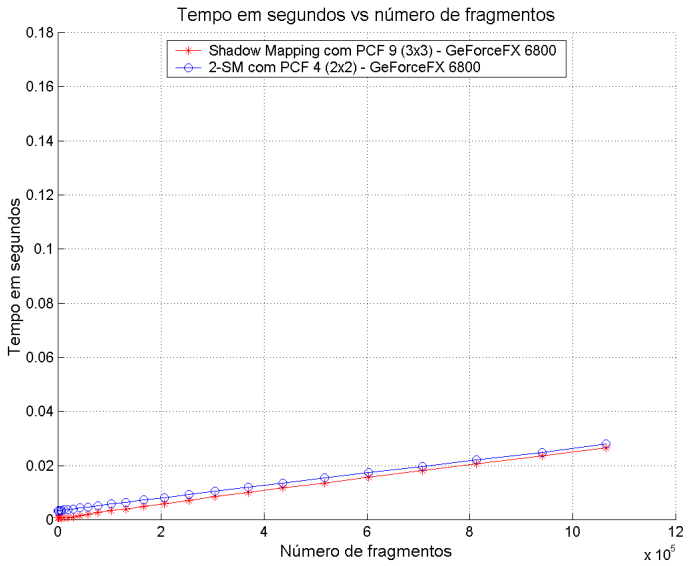


Figura 5.8: A figura apresenta a comparação entre os custos de geração de uma cena através dos algoritmos de *Shadow Mapping* tradicional, com um *PCF* de 9 elementos, e o 2-SM, com *PCF* de 4 elementos, em função do número de fragmentos da cena. A resolução da imagem final, do *Shadow Map* e do 2-SM é de 512^2 *pixels*. Medições feitas com uma placa *GeForceGT 6800*. Neste caso, não ocorre cruzamento dos gráficos.

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou o conceito de SMMVP, capaz de armazenar múltiplos valores de profundidade em cada célula, como uma generalização do *Shadow Map* tradicional. Também foi introduzido o conceito de TSC, como uma generalização do teste de sombra tradicional. Quando combinado com o *PCF*, estas novas estruturas produzem um bom número de valores distintos de atenuação, garantindo transições suaves entre regiões de sombra e regiões iluminadas. Como pode ser observado, os resultados visuais obtidos com o SMMVP, através do uso de *PCFs* menores, é comparável ao obtido com o *Shadow Mapping* tradicional em conjunto de *PCF*s de dimensões maiores.

Embora o SMMVP conte com um custo adicional relacionado à geração dos valores adicionais para suas células, este é constante, e independe da complexidade da cena, estando vinculado somente à resolução do SMMVP. Foi demonstrado analiticamente e experimentalmente que o método proposto apresenta melhor performance à medida que o número de fragmentos gerados pela cena aumenta.

O SMMVP, por se tratar de uma extensão ao algoritmo de *Shadow Mapping*, conta também com todas as suas vantagens. É um algoritmo robusto, que pode ser utilizado em conjunto com qualquer objeto que possa ser discretizado em um *depth-buffer*. Todas as suas etapas são facilmente mapeáveis para o *hardware* gráfico atual, permitindo assim a geração de sombras a uma taxa interativa. Este método também pode ser utilizado em sistemas de rendering *offline*.

Como trabalhos futuros, implementações com diferentes critérios para a seleção dos valores adicionais de profundidade das células do SMMVP serão feitas. Também se pretente fazer uma análise mais detalhada da performance do algoritmo, principalmente em *hardware* com políticas mais elaboradas de *cache* e *prefetching*.

REFERÊNCIAS

AILA, T.; LAINE, S. Alias-Free Shadow Maps. In: EUROGRAPHICS SYMPOSIUM ON RENDERING, 2004. **Proceedings...** [S.l.:s.n.], 2004. p.161-166.

ATi. **HyperZ, HyperZII, HyperZIII+, HyperZ HD**. Disponível em: <http://www.ati.com/companyinfo/glossary/includes/list.html#hzhd>. Acesso em: 4 fev. 2005.

ATi. **SMARTSHADER 2.0**. Disponível em: <http://www.ati.com/companyinfo/glossary/includes/list.html#smartshader20>. Acesso em: 6 mar. 2005.

BLINN, J. Me and My (Fake) Shadow. **IEEE Computer Graphics and Applications**, [S.l.], v.8, n.1, p.82-86, Jan. 1988.

BRABEC, S.; ANNEN, T.; SEIDELL H. Practical shadow mapping. **Journal of Graphics Tools**, [S.l.], v.7, n.4, p.9-18, 2002.

BRABEC, S.; SEIDEL, H. Hardware-accelerated Rendering of Antialiased Shadows with Shadow Maps. In: COMPUTER GRAPHICS INTERNATIONAL, 2001. **Proceedings...** [S.l.:s.n.], 2001.

CATMULL, E. **A Subdivision Algorithm for Computer Display of Curved Surfaces**. 1974. Tese (Doutorado) - Utah University, [S.l.].

CHONG, H.; GORTLER, S. A Lixel for Every Pixel. In: EUROGRAPHICS SYMPOSIUM ON RENDERING, 2004. **Proceedings...** [S.l.:s.n.], 2004.

COHEN M.F.; WALLACE, J.R. **Radiosity and realistic image synthesis**. Boston:Academic Press Professional, 1993.

COOK, R. Shade Trees. **Computer Graphics**, New York, v.18, n.3, July 1984. Trabalho apresentado na 11. SIGGRAPH, 1984.

COOK, R.; TORRANCE, K. A Reflection Model for Computer Graphics. **ACM Transactions on Graphics**, [S.l.], v.1, n.1, p.7-24, 1982.

CROW, F. Shadow Algorithms for Computer Graphics. **Computer Graphics**, New York, v.11, n.2, 1977. Trabalho apresentado na SIGGRAPH, 1977.

DEERING, M. et al. The triangle processor and normal vector shader: a VLSI system for high performance graphics. **Computer Graphics**, New York, v.22, n.4, 1988. Trabalho apresentado na SIGGRAPH, 1988.

DIEFENBACH, P. J. **Pipeline Rendering: Interaction and Realism through Hardware-based Multi-pass Rendering**. 1996. Tese (Doutorado) - Pennsylvania University, [S.l.].

EVERITT, C. **Interactive Order-Independent Transparency**. Disponível em: <http://developer.nvidia.com/object/Interactive_Order_Transparency.html>. Acesso em: 29 mar. 2005.

EVERITT, C. **Shadow Mapping**. Disponível em: <http://developer.nvidia.com/attach/6392>. Acesso em: 20 mar. 2005.

FERNANDO, R. **GPU Gems:** Programming Techniques, Tips and Tricks for Real-Time Graphics. Boston:Addison-Wesley, 2004. 765p.

FERNANDO, R. et al. Adaptive shadow maps. In: PROCEEDINGS OF ACM SIGGRAPH, 2001. **Proceedings...** [S.l.:s.n.], 2001. p.387-390.

FERNANDO, R. et al. Programming Graphics Hardware. In: EUROGRAPHICS, 2004. **Proceedings...** [S.l.:s.n.], 2004.

FERNANDO, R.; KILGARD, M. J. **The Cg Tutorial**: The Defintive Guide to Programmable Real-Time Graphics. [S.l.]:Addison-Wesley, 2003. 336p.

FUCHS, H. et al. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. **Computer Graphics**, [S.l.], v.19, n.3, 1985. Trabalho apresentado na SIGGRAPH, 1985.

FUCHS, H.; POULTON, J. Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine. **VLSI Design**, [S.l.], v.2, n.3, p.20-28, 1982.

GORAL, C. et al. Modeling the Interactions of Light between Diffuse Surfaces. **Computer Graphics**, [S.l.], v.18, n.3, 1984. Trabalho apresentado na SIGGRAPH, 1984.

GOURAUD, H. Continuous shading of curved surfaces. **IEEE Transactions on Computers**, [S.l.], p.623-629, June 1971.

GREENE, N.; KASS, M.; MILLER, G. Hierarchical Z-buffer visibility. **Computer Graphics**, [S.l.], v.27, Annual Conference Series, 1993. Trabalho apresentado na 20. SIGGRAPH, 1993.

HAEBERLI, P.; AKELEY K. The Accumulation Buffer: Hardware Support for High-Quality Rendering. **Computer Graphics**, [S.l.], v.24, n.4, p.309-318, 1990.

HASENFRATZ, J. et al. A survey of Real-Time Soft Shadows Algorithms. In: EUROGRAPHICS, 2003. **Proceedings...** [S.l.:s.n.], 2003.

HALL, R. **Illumination and color in computer generated imagery**. [S.l.]:Springer-Verlag, 1988.

HEIDMANN, T. Real Shadows, Real Time. In: IRIS UNIVERSE, 1991. **Proceedings...** [S.l.:s.n.], 1991. p.23-31.

HEIDRICH, W. **High-quality Shading and Lighting for Hardware-accelerated Rendering**. 1999. Tese (Doutorado) - University of Erlangen, [S.l.].

HILLIS, D. **The Connection Machine.** Tese (Doutorado) - Massachusetts Insitute of Technology, Cambridge.

HUBONA, G.S. et al. The role of object shadows in promoting 3D visualization. **ACM Transactions on Computer-Human Interaction**, [S.l.], v.6, n.3, p.214-242, 1999.

JOHNSON, G.; MARK, W.; BURNS, C. **The Irregular Z-Buffer and its Applications to Shadow Mapping**. [S.l.]:The University of Texas at Austin, Department of Computer Sciences, 2004. (Technical Report TR-04-09).

LOKOVIC, T.; VEACH, E. Deep Shadow Maps. In: SIGGRAPH, 2000. **Proceedings...** [S.l.:s.n.], 2000. p. 385-392.

MAMASSIAN, P.; KNILL, D.C.; KERSTEN D. The perception of cast shadows. **Trends in Cognitive Sciences**, [S.l.], v.2, n.8, p.288-295, 1998.

MARTIN, T.; TAN, T. Anti-aliasing and Continuity with Trapezoidal Shadow Maps. In: EUROGRAPHICS SYMPOSIUM ON RENDERING, 2004. **Proceedings...** [S.l.:s.n.], 2004.

MÖLLER, T.A.; HAINES E. **Real-Time Rendering.** 2nd. ed. Natick, MA:A. K. Peters, 2002.

MOORE, G. Cramming more components onto integrated circuits. **Electronics**, [S.l.], v.38, n.8, 1965.

NVIDIA. **Vertex Blending**. 1999. Disponível em: <http://developer.nvidia.com/object/IO_20010903_4822.html>. Acesso em: 6 fev. 2005.

NVIDIA. **Using Vertex Textures**. 2004. Disponível em: <http://developer.nvidia.com/object/using_vertex_textures.html>. Acesso em: 6 fev. 2005.

NVIDIA. **GeForce 6 Series**. 2004. Disponível em: <http://www.nvidia.com/object/IO_16823.html>. Acesso em: 6 fev. 2005.

NVIDIA. **NVIDIA GPU Programming Guide version 2.1.0**. 2004. Disponível em: <http://developer.nvidia.com/object/gpu_programming_guide.html>. Acesso em: 6 fev. 2005.

OLANO, M.; LASTRA A. A Shading Language on Graphics Hardware: The Pixel-Flow Shading System. In: PROCEEDINGS OF ACM SIGGRAPH, 1998. **Proceedings...** [S.l.:s.n.], 1998. p. 159-168.

OpenGL Specification. **OpenGl Specification**. 2004. Disponível em: <http://www.opengl.org/documentation/spec.html>. Acesso em: 7 fev. 2005.

PAGOT, C.; COMBA, J.; OLIVEIRA M. Multiple-depth shadow maps. In: BRAZILIAN SYMPOSIUM ON COMPUTER GRAPHICS AND IMAGE PROCESSING, 17., 2004, Curitiba. **Proceedings...** Los Alamitos: IEEE, 2004. p.308-315.

PHONG, B.T. Illumination for computer generated pictures. **Comunications of the ACM**, [S.l.], v.18, n.8, p.311-317, 1975.

PROUDFOOT, K. et al. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In: PROCEEDINGS OF ACM SIGGRAPH, 2001. **Proceedings...** [S.l.:s.n.], 2001. p. 161-166.

REEVES, W.T.; SALESIN, D.H.; COOK, R.L. Rendering antialiased shadows with depth maps. **Computer Graphics**, [S.l.], v.21, n.4, 1987. Trabalho apresentado na SIGGRAPH, 1987.

SEGAL, M. et al. Fast Shadows and Lighting Effects using Texture Mapping. In: PROCEEDINGS OF ACM SIGGRAPH, 1992. **Proceedings...** [S.l.:s.n.], 1992. p. 249-252.

STAMMINGER, M.; DRETAKKIS, G. Perspective Shadow Maps. **Computer Graphics**, [S.l.], v.21, n.3, 2002. Trabalho apresentado na SIGGRAPH, 2002.

TOMSHARDWARE. **Rivoluzione nelle Prestazioni: NV40 alias GeForce 6800 Ultra di NVIDIA**. 2004. Disponível em: <http://www.tomshw.it/graphic.php?guide=20040414&page=feforce_6800-09>. Acesso em: 29 mar. 2005.

UPSTILL, S. **The RenderMan Companion:** A programmer's Guide to Realistic Computer Graphics. Reading Massachusetts: Addison-Wesley, 1990.

WANGER, L. The effect of shadow quality on the perception of spatial relationships in computer generated imagery. **Computer Graphics**, [S.l.], v.25, n.2, p.39-42, 1992.

WARNOCK, J.E. **A hidden surface algorithm for halftone picture representation**. 1969. Tese (Doutorado) - University of Utah, [S.l.].

WHITTED, T. An Improved Illumination Model for Shaded Display. **Communications of the ACM**, [S.l.], v.23, n.6, p.343-349. Jun. 1980.

WILLIAMS, L. Casting Curved Shadows on Curved Surfaces. In: PROCEEDINGS OF ACM SIGGRAPH, 1978. **Proceedings...** [S.l.:s.n.], 1978. p. 270-274.

WOO, A.; POULIN, P.; FOURNIER, A. A Survey of Shadow Algorithms. **IEEE Computer Graphics and Applications**, [S.l.], v.10, n.6, p.13-32, 1990.