Renderização de cenas tridimensionais não-fotorealistas explorando hardware programável

Bruno Evangelista[1], Alessandro Silva[1], Marcelo Nery[1] (Orientador), Rosilane Mota[1] (Orientadora)
[1]PUC-MG - Pontifícia Universidade Católica de Minas Gerais
bpevangelista@yahoo.com.br, spdoido@yahoo.com.br, {msnery, rosilane}@pucminas.br

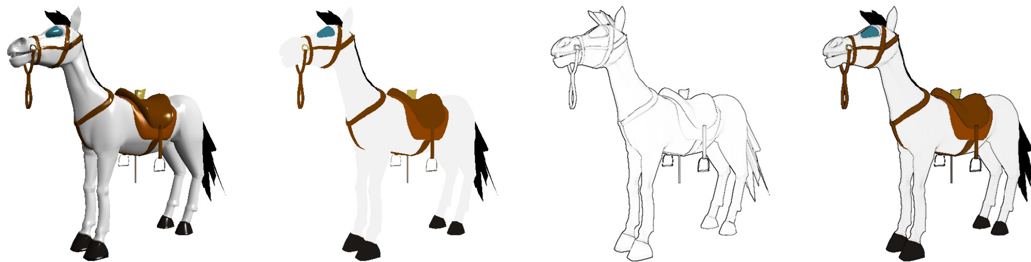


Figura 1. Técnica de três passos para renderização de cartoons.

Resumo

Renderização não-foto realista (NPR) é uma área que estuda a renderização ou caracterização de desenhos técnicos e artísticos. Renderização de cartoons é uma das áreas presentes na NPR, que estuda a obtenção de imagens com características de desenhos feitos à mão, a partir de geometrias tridimensionais. A renderização de cartoons utilizando hardwares gráficos programáveis é uma técnica recente que vem ganhando destaque, devido ao desenvolvimento de hardwares gráficos de alto desempenho e baixo custo. Nesse trabalho, são apresentadas duas técnicas para a renderização de imagens que simulam desenhos no estilo cartoons, em tempo real. Essas técnicas utilizam hardware gráfico programável e a linguagem de shaders do OpenGL, chamada GLSL.

1 Introdução

Renderização não-fotorealista ou NPR é uma área que estuda a renderização ou caracterização de desenhos técnicos e artísticos [11]. Renderização de *cartoons* é uma das áreas presentes na NPR, que estuda a obtenção de imagens com características de desenhos feitos à mão, a partir de geometrias tridimensionais. Enquanto o foto-realismo se concentra em gerar imagens que não possam ser distinguidas de fotografias do mundo real, a renderização de *cartoons* se baseia em gerar imagens que não possam ser distinguidas de desenhos feitos à mão. Quanto maior a semelhança com os desenhos, melhor é considerada a qualidade da imagem.

Modelos de NPR para *cartoons* constituem uma área antiga em computação gráfica, sendo utilizada há quase duas décadas [15]. Os primeiros trabalhos na área não podiam ser aplicados em tempo real, necessitando de um grande tempo de processamento. Nos últimos anos, devido à popularização dos *hardwares* gráficos programáveis, tornou-se possível aplicar a técnica em tempo real. Dessa forma, é possível modificar o *pipeline* de processamento de cenas realísticas, inserindo um programa *shader* capaz de modificar a renderização de toda a cena. Nos últimos anos, essa técnica vem sendo utilizada em jogos [2], desenhos animados japoneses [1], dentre vários outros.

Nesse trabalho, são propostas duas técnicas para a renderização de *cartoons* em tempo real, constituindo a principal contribuição dos autores. Essas técnicas utilizam *hardware* gráfico programável e a linguagem de *shaders* do OpenGL, chamada GLSL [14]. A Figura 1 ilustra o resultado obtido com uma das técnicas propostas.

2 Trabalhos relacionados

Em 1990, Saito e Takahashi [15] propõem uma nova técnica para renderização de objetos tridimensionais, que

[1]*Apple Seed* é uma animação japonesa, que mistura renderizações foto-realistas e renderizações de *cartoons*. Página: http://www.a-seed.jp/index2.html. Último acesso: 10/07/2005.

produz imagens com um realce visual mais compreensível, ao invés de simular com precisão os fenômenos ópticos. Essa técnica utiliza recursos como desenho do contorno dos objetos e mudança no tipo de sombreamento das faces, comuns nos desenhos feitos à mão. Essas técnicas são comuns nos desenhos feitos à mão. Saito e Takahashi também introduzem o conceito de G-Buffer (Geometric Buffer), uma estrutura de dados que armazena propriedades das geometrias tridimensionais, como mapa de profundidade, mapa de normais e mapa de identificação de objetos.

Decaudin [4] foi o primeiro a olhar especificamente para a renderização de *cartoons*. Ele descreve uma seqüência de passos para a renderização de *cartoons*, baseada em técnicas de processamento de imagens, descritas por Saito [15]. Decaudin usa uma versão simplificada do G-Buffer, onde somente os mapas de profundidade e normais são utilizados. Entretanto, assim como Saito, sua abordagem não pode ser utilizada em tempo real, pois cada cena demora cerca de seis minutos para ser gerada. A Figura 2 apresenta o resultado final da técnica proposta por Decaudin.

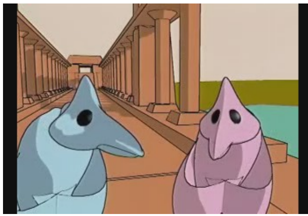


Figura 2. Rendez-vous. Renderização proposta por Decaudin. Detecção de bordas e sombreamento uniforme com dois tons. [4]

Mitchell [10, 3] apresenta uma solução para desenho de *cartoons* em tempo real, utilizando *shaders*. Ele propõe várias técnicas para detecção de bordas, duas delas aplicadas ao domínio da imagem. Uma dessas técnicas utiliza o mapa de normais e identificador de objetos, a outra utiliza mapa de profundidade e normais. As duas abordagens conseguem detectar grande parte das bordas com precisão. Segundo Mitchell, sua solução pode ser utilizada em tempo real, mas não são fornecidos dados sobre os testes realizados ou sobre o desempenho obtido.

Gooch [8] faz um trabalho comparando várias técnicas para detecção de bordas. Ele divide os algoritmos em dois grupos, de acordo com o domínio de trabalho: domínio do espaço e domínio da imagem.

Outras técnicas para adicionar maior qualidade no desenho de *cartoons* podem ser utilizadas, como *hatching* [13, 3], *halftoning* [6] e *strokes* [9]. Essas técnicas podem ser adicionadas de acordo com o estilo do desenho final desejado. A aplicação dessas técnicas pode ser vista na Figura 3.

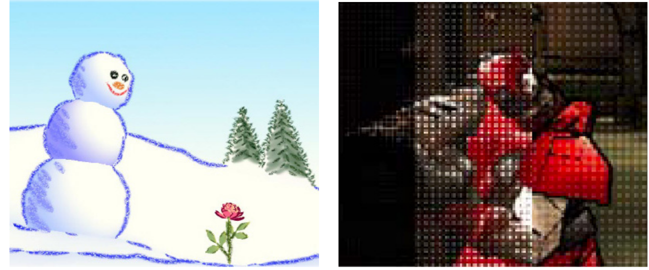


Figura 3. Exemplos de renderização. Hatching e strokes (esquerda), simulando desenhos com giz de cera [9]. Halftoning (direita), simulando impressões antigas [6].

2.1 Shaders e GLSL

De maneira geral, os *shaders* podem ser definidos como pequenos programas, que funcionam nos *hardwares* gráficos programáveis. Eles proporcionam uma maneira fácil de processar os vértices e fragmentos da cena.

Neste trabalho, utilizou-se a linguagem de *shaders* do OpenGL, chamada GLSL. Na GLSL, cada *shader* é composto por um programa de processamento de vértices e um programa de processamento de fragmentos. Um fragmento é um conjunto de dados gerado pela rasterização, como coordenadas, profundidade, cor e coordenada de textura.

3 Renderização de cartoons

3.1 Visão geral

Na renderização de *cartoons*, os objetos são desenhados com linhas sólidas que separam áreas de cores diferentes ou objetos diferentes. As cores são uniformes, não existindo suavização na troca entre cores adjacentes. Além disso é utilizado um número limitado de tons, para simular sombras próprias (self-shadows) e as componentes de luz ambiente, difusa e especular. As características utilizadas para renderização de *cartoons* devem ser extraídas da geometria tridimensional da cena, sendo as mais utilizadas (i) bordas, silhueta e bordas internas; (ii) sombreamento de cores uniforme; (iii) número limitado de tons e (iv) *hatching*.

O uso dessas características depende do efeito final desejado, do suporte proporcionado pelo *hardware* gráfico e do tempo de resposta disponível. Adicionar apenas algumas características à imagem não implica no resultado final possuir uma qualidade visual baixa. Um exemplo disso é o jogo *The Legend of Zelda: The Wind Waker* [2]. O jogo não faz uso da detecção de bordas, economizando tempo de processamento. O sombreamento utilizado é uniforme, com

Figura 4. ‘The Legend of Zelda: The Wind Waker’ [2]. Renderização de cartoon sem borda, sombreamento uniforme com dois tons.

dois tons: ambiente e especular. Apesar de não possuir bordas, as imagens geradas são de alta qualidade, como pode ser visto na Figura 4.

3.2 Sombreamento de cores e tonalização

A técnica de sombreamento utilizada na renderização de *cartoons* é responsável pela aparência das superfícies da cena. Nos *cartoons*, o sombreamento das superfícies é uniforme, não existindo suavização entre as cores das faces ou dos vértices. Os tons de iluminação também não possuem suavização, sendo brusca a passagem entre uma área muito iluminada e outra pouco iluminada.

O cálculo da cor de sombreamento é realizado utilizando a equação de iluminação de Phong [5]. No entanto, para fins de otimização, o fator de atenuação da componente especular não é considerado.

A equação de Phong é aplicada a todos os fragmentos da cena. O resultado da equação é calculado separadamente para cada componente da luz. Para cada uma das componentes, os valores de saída são normalizados e variam entre 0.0 e 1.0. São criadas faixas de valores para cada tom da cena, de acordo com o efeito de luz desejado. O resultado é utilizado para escolher o tom que deve ser utilizado em cada fragmento. Esse resultado também pode ser utilizado para acessar texturas unidimensionais [9].

A Figura 5 mostra a comparação entre renderizações de um cavalo, considerando o mesmo ponto de vista. São utilizados os sombreamentos de Phong e uniforme, utilizando um, dois e três tons respectivamente. Os dois primeiros tons representam a luz difusa e o terceiro tom, mais claro, a luz especular. As renderizações foram realizadas sem o uso de bordas.

3.3 Bordas

Nos *cartoons*, as bordas são utilizadas para desenhar as linhas que contornam os objetos, interna e externamente.

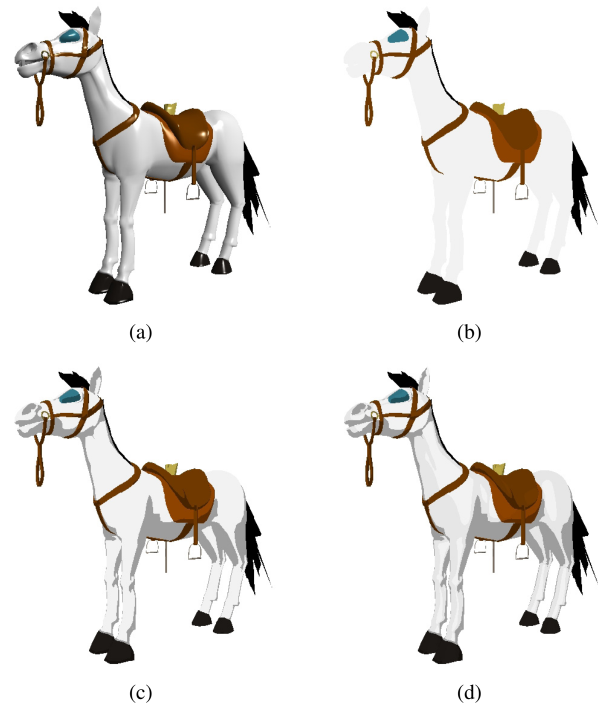


Figura 5. Comparação de renderização, considerando o mesmo ponto de vista. Sombreamento de Phong (a), sombreamento uniforme com um tom (b), sombreamento uniforme com dois tons (c) e sombreamento uniforme com três tons (d).

Existem vários tipos de borda [11], sendo que, para o efeito desejado neste trabalho, foram usados dois tipos:

- Silhueta ou *outline border*;
- Bordas internas ou *crease border*.

A silhueta, ou borda externa, varia de acordo com o ponto de vista pelo qual o objeto é visualizado. As bordas internas geralmente são formadas por faces adjacentes, cujo ângulo entre suas normais é superior a um valor pré-definido. Na Figura 6 são exibidos os dois tipos de bordas. A borda externa é representada com a cor vermelha e a borda interna é representada com a cor azul.

A borda dos objetos pode ser obtida a partir de várias técnicas que são aplicadas no domínio do espaço ou da imagem [8]. Nesse trabalho, foi utilizada uma técnica aplicada ao domínio da imagem. Trabalhar com o domínio do espaço geralmente requer um pré-processamento de toda a cena e uma maior quantidade de memória. Além disso, grande parte das técnicas aplicadas ao domínio do espaço são difíceis de implementar e não fazem uso dos *hardwares* programáveis. Dentre as técnicas que trabalham no domínio

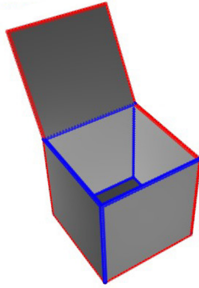


Figura 6. Uma caixa aberta. Bordas externas(vermelho), bordas internas(azul).

da imagem, as mais utilizadas são: método de um passo, método de dois passos e detecção de descontinuidade, [8].

Na Seção 4 serão apresentados os algoritmos desenvolvidos para a renderização de *cartoon*.

4 Metodologia para renderização de cartoon

Nas Seções 3.2 e 3.3, foram apresentados os conceitos básicos utilizados nas técnicas para a renderização de *cartoons*. A princípio, são necessários três passos para se renderizar uma cena: (i) obtenção dos mapas de profundidade e normais, (ii) renderização da cena utilizando sombreamento uniforme e tonalização desejada e (iii) detecção de bordas e mistura com a cena final.

Os passos não precisam ser necessariamente aplicados nessa ordem. Após uma observação nos passos necessários para a renderização, é possível perceber que alguns deles poderiam ser agrupados. Os dois últimos passos necessários poderiam ser agrupados em um único passo, onde as bordas são detectadas e misturadas com a cena. Também é possível agrupar os dois primeiros passos, gerando os mapas para detecção de bordas e a renderização com sombreamento uniforme e tonalização em um único passo. No entanto, para se agrupar os dois primeiros passos, seria necessário um suporte maior do *hardware* gráfico. Alguns *hardwares* gráficos possuem o recurso de múltiplas saídas de renderização, o que permitiria agrupar os dois primeiros passos.

Na Seção 4.1 é apresentado um algoritmo para detecção de bordas no domínio da imagem. Na Seção 4.2 é apresentada uma otimização na seqüência de passos para realizar a renderização final de *cartoons*, com dois passos. A Figura 7 apresenta o diagrama com a seqüências de passos, para renderização com dois e três passos.

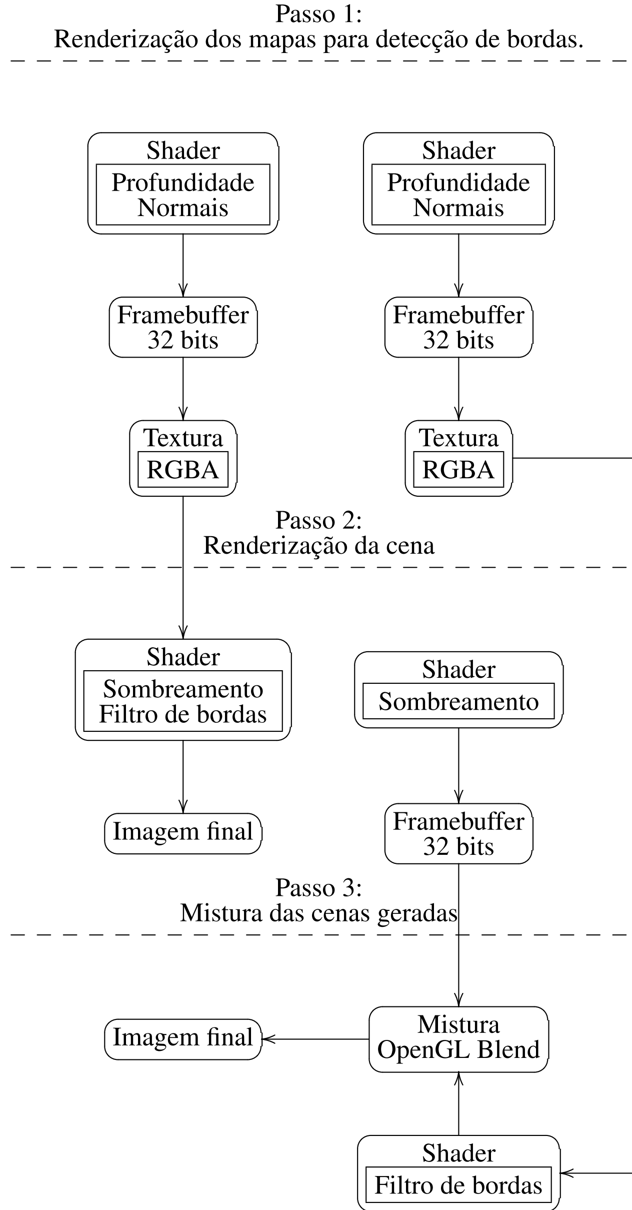


Figura 7. Diagrama de passos para renderização de cartoons. Seqüência de dois passos (esquerda), seqüência de três passos (direita).

4.1 Algoritmo para detecção de bordas

O algoritmo para detecção de bordas utilizado é aplicado ao domínio da imagem e utiliza a técnica de detecção de descontinuidade na cor dos fragmentos da cena. Essa técnica permite detectar com precisão as bordas internas e externas das geometrias, incluindo bordas curvas [4, 8, 10, 15]. O algoritmo utiliza o mapa de profundidades para de-

tectar as bordas externas das geometrias. O mapa de normais é utilizado para detectar as bordas internas. O mapa de profundidades é quantizado e representado usando o canal *Alpha* de uma textura que possui quatro componentes, RGBA. Os valores do mapa de profundidades são normalizados entre 0.0 e 1.0. O mapa de normais é representado utilizando as componentes RGB da textura. Isso restringe a precisão dos elementos do mapa de profundidade, não podendo ultrapassar 8 bits.

A Figura 8 ilustra um modelo de macacos renderizados utilizando o sombreamento de Phong. Na Figura 9, podem ser vistos separadamente os mapas de profundidade e normais do mesmo modelo.

Sem o suporte dos *hardwares* programáveis modernos, seria necessário renderizar toda a cena duas vezes e armazenar os resultados. O *hardware* gráfico programável é explorado para extrair todas as informações necessárias em apenas um passo. Para isso, é utilizada uma das abordagens propostas por Mitchell [3]. Todas as informações são renderizadas no *framebuffer* e posteriormente copiadas para uma textura RGBA.

A textura será utilizada como entrada para um novo *shader* somente após ser extraída da cena, obtendo nesse processo os mapas de profundidade e normais. Esse *shader* é responsável em aplicar um filtro de detecção de bordas à textura. Diversos filtros podem ser utilizados para essa tarefa como o filtro de Sobel, Laplaciano, Gaussiano e Roberts [7].

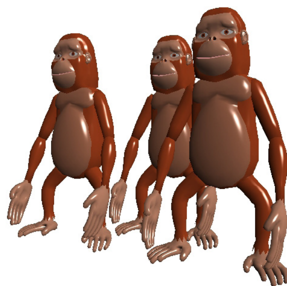


Figura 8. Os macacos Groley. Renderização utilizando sombreamento de Phong. Cada modelo possui 6.310 triângulos.

Saito, Decaudin e outros autores [4, 15] utilizam como filtro inicial um filtro de detecção de bordas 3×3, que exige nove acessos à memória de textura. Em seguida são aplicados novos filtros sobre o resultado do primeiro filtro. Marc Nienhaus [12] utiliza o filtro Laplaciano em seu trabalho, que pode ser visto na Figura 10. Apesar de exigir o mesmo número de acessos à memória de textura, esse filtro realiza um número menor de operações matemáticas.

Mitchell [3] utiliza um filtro 3×3, que exige cinco acessos à memória de textura. Entretanto, devido às limitações da linguagem de *shader* utilizada por Mitchell, os cinco acessos à memória de textura precisam ser realizados duas vezes, em dois programas de *shaders* diferentes. A Figura 10 mostra o filtro utilizado por Mitchell.

Figura 9. Mapa de normais (esquerda), mapa de profundidade (direita). No mapa de profundidade, as cores escuras representam maior proximidade.

-1	-1	-1
-1	8	-1
-1	-1	-1

1	0	1
0	1	0
1	0	1

Figura 10. Filtro para detecção de bordas. Filtro Laplaciano (esquerda), utilizado por Nienhaus [12]. Filtro utilizado por Mitchell [3] (direita).

Caracteristicamente, a precisão obtida com o filtro Laplaciano é muito grande, detectando assim uma grande quantidade de bordas, o que não é desejado para o estilo *cartoon* proposto. Além disso, a quantidade de acessos à memória de textura, realizadas dentro do *shader*, aumenta o tempo total para renderização da cena. De modo a otimizar o processo de detecção de bordas, foi utilizada uma versão simplificada do filtro Laplaciano. Esse filtro também realiza cinco acessos à memória de textura, como visto na Figura 10. O filtro utilizado é baseado no filtro Laplaciano e detecta bordas na horizontal e vertical.

À medida que as bordas são detectadas, elas precisam ser

0	-1	0
-1	4	-1
0	-1	0

Figura 11. Filtro de detecção de bordas utilizado.

misturadas à imagem final. Para isso, é necessário habilitar o estado de mistura de cores, presente no *pipeline* padrão do OpenGL denominado *blend*, possibilitando misturar a renderização atual com a última renderização realizada.

De modo a permitir que a mistura seja feita em todos os fragmentos da cena, é necessário renderizar um quadrado com as dimensões da tela. Uma projeção paralela é então utilizada, renderizando o quadrado no plano mais próximo à câmera da cena. A renderização desse quadrado utiliza o *shader* de fragmentos, responsável por detectar as bordas, e o *blend*, responsável por misturar as bordas detectadas com a última cena renderizada.

Na Figura 12 pode ser vista a renderização separada das bordas do modelo, geradas a partir dos mapas da Figura 9.

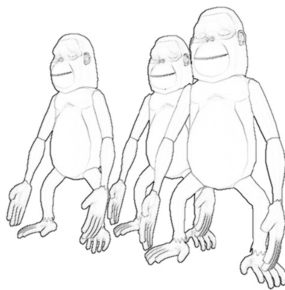


Figura 12. Renderização da borda dos modelos da cena. Resultado da convolução do filtro de detecção de bordas, aplicado ao mapa de profundidade e normais da cena.

4.2 Algoritmo de dois passos

A otimização na seqüência de passos para renderização de *cartoons* é possível agrupando os dois últimos passos em um único passo. Para isso, é necessário agrupar todo o processamento necessário dentro do *shader* de fragmentos. Antes, esse processamento era dividido entre o *shader* e o *pipeline* padrão do OpenGL, por causa do estado *blend*. Após a modificação, o *shader* de fragmentos fica responsável em realizar as seguintes tarefas:

- Detectar bordas;
- Calcular cor do fragmento, utilizando sombreamento uniforme;
- Misturar os itens um e dois, gerando o fragmento de saída.

Na seqüência de três passos, essas tarefas eram realizadas pelo *shader* de detecção de bordas, *shader* de sombreamento e *pipeline* padrão do OpenGL.

5 Resultados

Os testes realizados objetivaram verificar a qualidade das imagens geradas, bem como o desempenho em relação ao número de quadros por segundo renderizados, comparando as metodologias de dois e três passos propostas.

Para que o uso das técnicas desenvolvidas no trabalho não fosse limitado às placas gráficas de última geração, foi utilizada uma arquitetura simples no seu desenvolvimento. Isso permite que as técnicas apresentadas possam ser aplicadas em um maior número de computadores. A única restrição para o desenvolvimento deste trabalho foi o uso de um *hardware* gráfico com suporte ao GLSL. A configuração do *hardware* utilizado para os testes foi de um processador com 1GHz, 512 MB de memória e placa de vídeo GeForce FX 5200 com 128 MB de memória. Todas as cenas foram renderizadas na resolução de 512×512 *pixels* com os modelos disponibilizados livremente pela 3D Café [1].

Em relação ao filtro de detecção de bordas proposto, o novo filtro utilizado proporcionou um aumento no desempenho de aproximadamente 48%, comparado ao Laplaciano convencional. Se a extração dos mapas para detecção de bordas tivesse sido realizada com mais de um passo, poderia ser necessário um maior processamento na detecção de bordas, exigindo acessos à memória de outras texturas.

Em todas as renderizações foi desenhado um quadrado no fundo, com a cor de fundo da cena. A renderização desse quadrado foi feita devido à uma restrição da técnica de dois passos que somente processava todos os fragmentos se houvesse um fundo no modelo tridimensional da cena, e não apenas modelos visíveis na tela.

A comparação de desempenho, por número de triângulos renderizados, entre as técnicas de dois e três passos, foi feita como segue. Optou-se por utilizar a renderização das cenas com 1, 2 e 3 tons de sombreamento, avaliando o número de quadros por segundo que cada uma das duas técnicas propostas gastava para gerar a imagem final. Foram renderizadas seis cenas nos três testes de sombreamentos de tons. Estas cenas foram montadas inicialmente com 3.039 triângulos, acrescentando novos modelos tridimensionais até ser atingido 122.646 triângulos. A Figura 13 ilustra os resultados obtidos para este experimento.

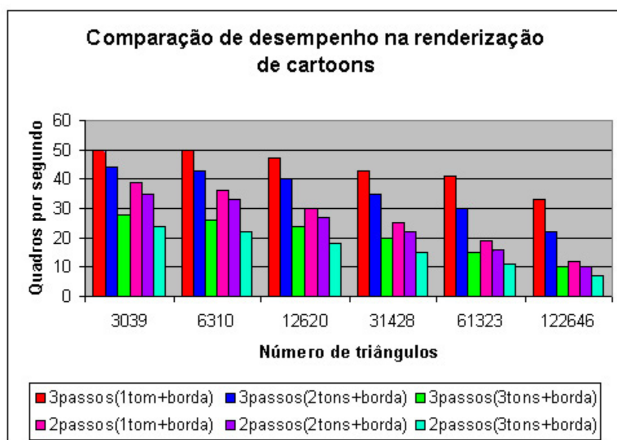
Dos dados analisados, pode-se perceber que o algoritmo de dois passos sempre obteve um desempenho inferior ao algoritmo de três passos. Acredita-se que devido à técnica de três passos dividir as tarefas necessárias para a renderização entre três *shaders* e o *pipeline* padrão do OpenGL, ela consegue um maior paralelismo das operações, e com isso um maior desempenho. A técnica de dois passos concentra a maior parte do processamento em um único programa de *shader*, o que pode tornar o paralelismo mais difícil. Com isso, o *shader* de dois passos não consegue obter um *throughput* adequado, comparado com a

técnica de três passos.

Entretanto, testes realizados com o algoritmo de dois passos, sem a renderização de um objeto de fundo, obtiveram um desempenho até 100% superior ao algoritmo de três passos. Esse valor sofria grandes variações em relação ao modo que a cena era observada. Em casos onde os objetos estavam com faces ocultas, o desempenho do algoritmo de duas faces era melhor, o mesmo ocorrendo quando os objetos estavam distantes. Obviamente, os casos opostos à estes, como um efeito de zoom muito alto, comprometeram o desempenho do método de dois passos.

Os *shaders* apresentados foram avaliados utilizando uma ferramenta de medição de desempenho. A única ferramenta disponível para esse propósito, que suporta a linguagem GLSL, é o NVShaderPerf da nVidia. Entretanto, essa ferramenta avalia apenas o desempenho dos *shader* de fragmentos utilizados. O resultado da avaliação é o *throughput* respectivo a cada *shader*, medido em mega *pixels* por segundo. No *shader* de três passos os valores são respectivamente 400.00, 36.36 e 28.57, enquanto que no *shader* de dois passos são de 400.00 e 15.69. Realizando uma comparação, abstraindo-se os detalhes de implementação e outros fatores constantes, pode-se percerber que o *shader* de três passos é ligeiramente superior ao *shader* de dois passos.

Assim sendo, animações em tempo real, baseados no desempenho dos dois algoritmos, cada um em seus casos particulares, permitem a renderização de *cartoons* sem comprometer a visualização da animação. A Figura 14 ilustra algumas imagens finais obtidas pela aplicação do algoritmo de dois passos.



(a)

Figura 13. Comparação de desempenho na renderização de cartoons, utilizando modelo de sombreamento de 1, 2 e 3 tons.

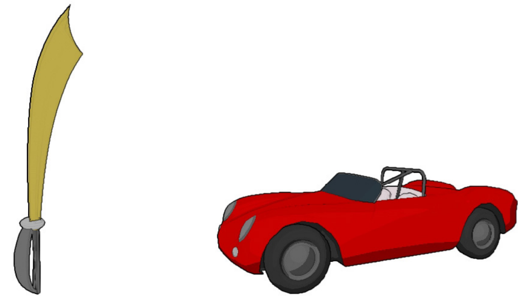


Figura 14. Sombreamento de um tom e bordas (esquerda). Sombreamento de três tons e bordas (direita).

6 Conclusão e trabalhos futuros

Nesse artigo foram apresentadas duas técnicas para a renderização de *cartoons* que exploram *hardware* gráfico programável. As duas técnicas apresentadas foram implementadas em *shaders* utilizando a linguagem GLSL. Também foram apresentadas comparações de desempenho entre ambas as técnicas.

Trabalhos futuros na área deverão explorar os recursos das placas gráficas mais recentes, como múltiplas saídas de renderização, além de utilizar extensões disponíveis para o OpenGL, visando eliminar as restrições impostas ao algoritmo, como precisão limitada no mapa de profundidade. Outros trabalhos futuros incluem a comparação de desempenho entre diferentes *hardwares* gráficos, e outros algoritmos propostos.

Referências

[1] 3d cafe. página: http://www.3dcafe.com. Último acesso: 10/07/2005.

[2] The legend of zelda: The wind waker. página: http://www.zelda.com/gcn. Último acesso: 10/07/2005.

[3] D. Card and J. L. Mitchell. *Non-Photorealistic Rendering with Pixel and Vertex Shaders. in ShaderX: Vertex and Pixel Shaders Tips and Tricks*. Wordware, 2002.

[4] P. Decaudin. *Modeling using Fusion of 3D Shapes for Computer Graphics – Cartoon-Looking Rendering of 3D Scenes*. PhD thesis, Universite de Technologie de Compiegne, France, dec 1996.

[5] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison-Wesley, Reading, second edition, 1995.

[6] O. Giroux and A. Denault. Image-based processing of game method streams and depth-buffered video for non-photorealistic rendering, 2004.

[7] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] B. Gooch, M. Hartner, and N. Beddes. Silhouette algorithims, siggraph 2003 course, 2003.

[9] R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, and A. Finkelstein. Wysiwyg npr: Drawing strokes directly on 3d models. *ACM Transactions on Graphics*, 21(3):755–762, July 2002.

[10] J. Mitchell, C. Brennan, and D. Card. Real-time image-space outlining for nonphotorealistic rendering, 2002.

[11] T. Moller, E. Haines, and T. Akenine-Moller. *Real-Time Rendering (2nd Edition)*. AK Peters, Ltd, 2002.

[12] M. Nienhaus and J. Döllner. Edge-enhancement - an algorithm for real-time non-photorealistic rendering. In *WSCG*, 2003.

[13] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581, New York, NY, USA, 2001. ACM Press.

[14] R. J. Rost. *OpenGL(R) Shading Language*. Addison-Wesley Professional, 2004.

[15] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, New York, NY, USA, 1990. ACM Press.

Apêndice 1: Shader de 3 passos

```
void main() {
  // Define a cor do vértice como a cor de sua normal
  gl_FrontColor.xyz=gl_NormalMatrix*gl_Normal;
  gl_Position=ftransform();
}
void main() {
  // Cor relativa aos mapas de normal e profundidade
  gl_FragColor=vec4(gl_Color.xyz,gl_FragCoord.z);
}
------------------------------------------------------------
varying vec3 reflectVec;
varying vec3 viewVec;
varying float diffuse;
// Pode ser trocado para uniform
const vec3 lightPos = vec3(1.0f,1.0f,1.0f);
void main() {
  vec3 normal=normalize(gl_NormalMatrix*gl_Normal);
  vec3 vertexPos=vec3(gl_ModelViewMatrix*gl_Vertex);
  vec3 lightDir=normalize(lightPos-vertexPos);
  // Luz difusa
  diffuse=max((dot(lightDir,normal)+1.0f)*0.5f,0.0f);
  // Luz especular
  reflectVec=normalize(reflect(-lightDir,normal));
  viewVec=normalize(-vertexPos);
  gl_FrontColor=gl_Color;
  gl_Position=ftransform();
}

varying vec3 reflectVec;
varying vec3 viewVec;
varying float diffuse;
void main() {
  // Luz especular
  reflectVec=normalize(reflectVec);
  viewVec=normalize(viewVec);
  float specular=max(dot(reflectVec,viewVec),0.0f);
  // Faixas de tons
  float lightIntensity=0.65f;
  if (specular>0.65f) lightIntensity=1.0f;
  else if (diffuse>=0.8f) lightIntensity=0.95f;
  gl_FragColor=vec4(gl_Color.xyz*lightIntensity,1.0f);
}
------------------------------------------------------------
void main() {
  // Projeção paralela utilizada
  gl_Position=gl_Vertex;
}

sampler2D borderMap;
// Utilizado para cálculo da coordenada da textura
#define TEXTURE_UNIT 0.001953125f
#define texPosX(trans) ((gl_FragCoord.x+trans)*TEXTURE_UNIT)
#define texPosY(trans) ((gl_FragCoord.y+trans)*TEXTURE_UNIT)
#define DETECTION_FACTOR 0.2f // Fator de detecção de bordas
void main() {
  // Filtro
  vec4 B=texture2D(borderMap,vec2(texPosX(0),texPosY(-1)));
  vec4 D=texture2D(borderMap,vec2(texPosX(-1),texPosY(0)));
  vec4 E=texture2D(borderMap,vec2(texPosX(0),texPosY(0)));
  vec4 F=texture2D(borderMap,vec2(texPosX(1),texPosY(0)));
  vec4 H=texture2D(borderMap,vec2(texPosX(0),texPosY(1)));
  vec4 filter=abs((-B-D+4*E-F-H)*DETECTION_FACTOR);
  float border=filter.r+filter.g+filter.b+filter.a;
  gl_FragColor=vec4(vec3(0.0f),border);
}
```

Apêndice 2: Segundo passo do shader de 2 passos

```
varying float diffuse;
varying vec3 reflectVec;
varying vec3 viewVec;
// Pode ser trocado para uniform
const vec3 lightPos = vec3(1.0f, 1.0f, 1.0f);
void main() {
  vec3 normal=normalize(gl_NormalMatrix*gl_Normal);
  vec3 vertexPos=vec3(gl_ModelViewMatrix*gl_Vertex);
  vec3 lightDir=normalize(lightPos-vertexPos);
  // Luz difusa
  diffuse=max((dot(lightDir,normal)+1.0f)*0.5f,0.0f);
  // Luz especular
  reflectVec=normalize(reflect(-lightDir,normal));
  viewVec=normalize(-vertexPos);
  gl_FrontColor=gl_Color;
  gl_Position=ftransform();
}

// Utilizado para cálculo da coordenada da textura
#define TEXTURE_UNIT 0.001953125f
#define texPosX(trans) ((gl_FragCoord.x+trans)*TEXTURE_UNIT)
#define texPosY(trans) ((gl_FragCoord.y+trans)*TEXTURE_UNIT)
#define DETECTION_FACTOR 0.2f // Fator de detecção de bordas
varying float diffuse;
varying vec3 reflectVec;
varying vec3 viewVec;
// Textura contendo os mapas de normal e profundidade
uniform sampler2D borderMap;
void main() {
  // Luz especular
  reflectVec=normalize(reflectVec);
  viewVec=normalize(viewVec);
  float specular=max(dot(reflectVec,viewVec),0.0f);
  // Filtro
  vec4 B=texture2D(borderMap,vec2(texPosX(0),texPosY(-1)));
  vec4 D=texture2D(borderMap,vec2(texPosX(-1),texPosY(0)));
  vec4 E=texture2D(borderMap,vec2(texPosX(0),texPosY(0)));
  vec4 F=texture2D(borderMap,vec2(texPosX(1),texPosY(0)));
  vec4 H=texture2D(borderMap,vec2(texPosX(0),texPosY(1)));
  vec4 filter=abs((-B-D+4*E-F-H)*DETECTION_FACTOR);
  float border=1.0f-(filter.r+filter.g+filter.b+filter.a);
  // Faixas de tons
  float light=0.65f;
  if (specular>0.65f) light=1.0f;
  else if (diffuse>=0.8f) light=0.95f;
  gl_FragColor=vec4(gl_Color.xyz*light*border,1.0f);
}
```