Um compilador de BRDFs para geração de shaders em GLSL

Kellisson Felipe Silva Freire
Departamento de Computação
Universidade Federal de Sergipe
São Cristóvão, Sergipe
Email: kellisson@outlook.com

Beatriz Trinchão Andrade
Departamento de Computação
Universidade Federal de Sergipe
São Cristóvão, Sergipe
Email: beatriz@ufs.br

***Resumo*—Implementar *shaders* baseados em modelos de BRDF pode se tornar um processo complexo, dada a abrangência de sua definição. Da mesma forma, é difícil implementar um renderizador que aceite formulações genéricas de uma BRDF. Neste artigo é proposto um compilador capaz de gerar *shaders* baseados em BRDFs. Para isso utilizamos uma Gramática Livre de Contexto capaz de representar suas formulações. Com isso, é possível criar *shaders* baseados em BRDF específicas a partir de sua formulação matemática.**

***Abstract*—To implement shaders based on BRDF models can become a complex process, given the coverage of its definition. Similarly, it is hard to implement a renderer that accept such generic formulations of a BRDF. In this article, we propose a compiler that is able to generate BRDF-based shaders. For this, we use a Context-Free Grammar able to generate its formulations. This way, it is possible to create shaders based on a BRDF only giving its mathematic formulation.**

***Keywords*-BRDFs; compilers; parsing; shaders; GLSL;**

I. Introdução

Uma BRDF, do inglês "Bidirectional Reflectance Distribution Function", é uma função de distribuição. Esta função representa a proporção de luz refletida, em relação à luz incidente, e tem a seguinte formulação básica [1]:

$$f_r(\theta_i, \phi_i; \theta_r, \phi_r) \tag{1}$$

Onde θ_i e ϕ_i indicam a direção da luz incidente, e θ_r e ϕ_r indicam a direção da luz refletida. Essas representações são dadas em termos de coordenadas polares.

Com esta formulação, é possível representar as propriedades de refletância de uma superfície [1], podendo variar de fórmulas simples pouco realistas a modelos complexos. Seu uso estende-se desde sistemas de renderização 3D fotorrealistas [2] a visão computacional.

Porém, existe a dificuldade de criar um renderizador capaz de aceitar uma formulação tão genérica. Esta dificuldade existe pois toda uma gama de constantes e operações podem compor uma função de uma BRDF. Isso implicaria em implementar todas essas propriedades, um processo que pode tornar-se complexo à medida que mais BRDFs são implementadas.

Neste artigo, propomos a criação de um renderizador capaz de aceitar modelos genéricos de BRDFs, e assim representar suas propriedades de refletância visualmente. Para isso, fizemos uso de uma Gramática Livre de Contexto e um compilador [3], estes sendo capazes de representar modelos analíticos genéricos de BRDF. Dessa forma, torna-se possível transcrever uma BRDF em sua formulação matemática para uma linguagem-alvo.

Este trabalho está estruturado da seguinte maneira. Na Seção 2 discutimos os conceitos essenciais da nossa proposta. Em seguida na Seção 3 são descritos trabalhos relacionados sobre renderizadores que propõem outras soluções para este problema. Já nas Seções 4 e 5 o renderizador e a gramática desenvolvidas por este trabalho são respectivamente descritos. Dois exemplos de seu funcionamento em conjunto são apresentados na Seção 6, e, por fim, as conclusões deste trabalho estão na Seção 7.

II. Conceitos Essenciais

A. Renderizadores

A renderização é o processo de gerar uma imagem dada uma representação de uma cena [4]. Logo, um renderizador é uma aplicação capaz de realizar esse processo. Comumente, a representação de uma cena é dada por uma malha, que nada mais é do que um conjunto de vértices e arestas que dão forma aos objetos da cena.

Para implementar um renderizador APIs gráficas são utilizadas, pois permitem a troca de dados entre dispositivos especializados em renderização e a aplicação. Neste artigo a API utilizada foi o OpenGL [5].

Nas versões a partir da 2.0, o OpenGL utiliza dois arquivos separados para calcular como a luz reage na malha. Estes arquivos são chamados de *vertex shader* e *fragment shader*.

Citando Shreiner [4], o *vertex shader* é utilizado durante o processo de renderização no momento de calcular os vértices individualmente. Já o *fragment shader* é utilizado durante o processo de rasterização para definir a cor e profundidade de um fragmento da cena. Estes *shaders* são compilados durante a execução do OpenGL de maneira separada à compilação do renderizador.

Dessa forma, esses arquivos podem ser utilizados para representar computacionalmente as formulações de BRDFs.

B. Gramáticas Livres de Contexto

Citando [6], uma Gramática Livre de Contexto é um conjunto de regras que definem uma linguagem. Com ela é

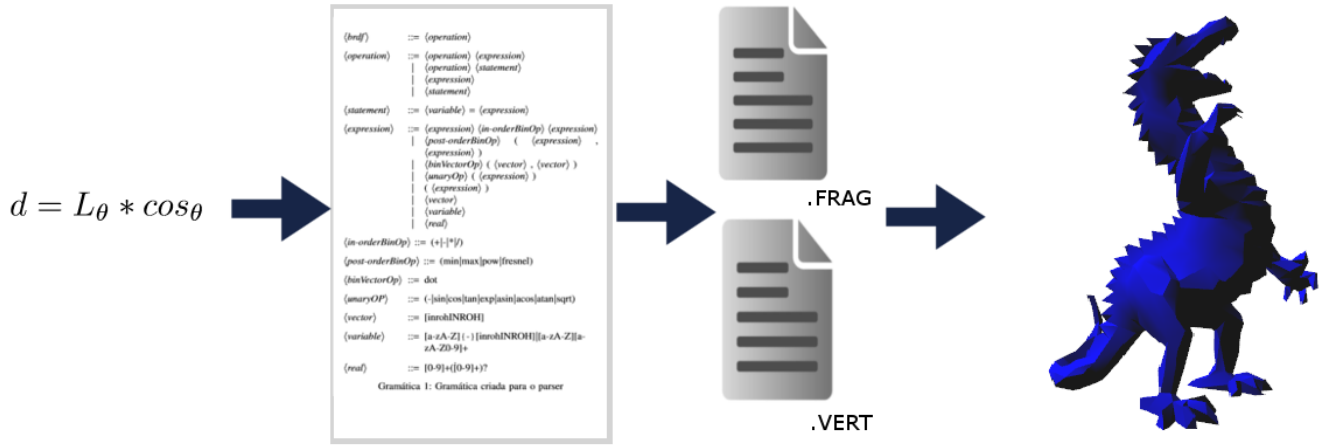


Figura 1: A Figura representa o processo da geração do *shader* dado a formulação da BRDF, onde a formulação é dada ao *parser*, representado pela gramática. Com isso serão gerados os *shaders* correspondentes. Por fim, a imagem resultante é apresentada.

possível dizer se um conjunto de caracteres pertence ou não a uma linguagem. A estrutura de uma gramática é definida por 3 componentes:

- Um conjunto de terminais, também chamados de "tokens". Estes são os símbolos minimais de uma gramática.
- Um conjunto de não-terminais, que definem uma cadeia de terminais.
- Um conjunto de produções, onde cada uma consiste em um não terminal, chamado de lado esquerdo da produção, uma seta, e um conjunto de terminais e não-terminais, chamado de corpo ou lado direito de uma produção.

1) Notação: A notação utilizada por este trabalho para definir uma Gramática Livre de Contexto é descrita abaixo:

- ⟨ ⟩ : Define um não terminal.
- [ ] : Define um conjunto de *tokens* dado por cada caractere entro das colchetes.
- ( ) : Define que todos os caracteres dentro dos parênteses são um *token* só.
- - : Define uma lista de caracteres que vai do caractere anterior ao hífen até o caractere posterior a ele.
- | : Define que a regra possui mais de uma alternativa de produção.
- ? : Define que a expressão regular pode aparecer ou não naquela regra.
- + : Define que a expressão regular aparecerá uma ou mais vezes naquela regra.

C. Compiladores

Um compilador é um programa que pode ler outro em uma linguagem (chamada linguagem fonte) e traduzi-lo em um programa equivalente em outra linguagem (chamada linguagem alvo) [6]. Este processo de tradução é dividido em duas partes em um compilador: análise e síntese.

O processo de análise é subdividido em análise léxica, análise sintática e análise semântica[3]. Na análise léxica os caracteres do código fonte são organizados em lexemas, e uma tabela de símbolos é criada. Estes lexemas são grupos mínimos de caracteres significativos numa linguagem. Com os lexemas, o analisador léxico entrega como saída os *tokens* correspondentes para o analisador sintático. Desta forma inicia-se a análise sintática.

Na análise sintática, ou *parsing*, um parser recebe os *tokens* da análise léxica como entrada. Esses *tokens* serão processador de acordo com as regras da gramática, gerando uma árvore sintática. Esta árvore será utilizada pelas fases seguintes para a geração do código-alvo.

Na análise semântica, a consistência do codigo fonte é conferida com o auxílio da árvore sintática e a tabela de símbolos. Com isso é possível verificar possíveis erros como erros de tipagem, por exemplo.

As fases subsequentes envolvem a geração de um código intermediário e otimização. Para este trabalho estes processos não são relevantes, já que a tradução da nossa linguagem-fonte para a linguagem-alvo é direta. Sendo assim, na Seção 6 serão apresentados apenas os processos de análise e geração de código-alvo.

III. O Renderizador

Para implementar o renderizador da forma pretendida, é necessário que ele possua uma interface entre o usuário e a malha renderizada. Esta interface deve permitir que o usuário movimente a malha e a luz pontual. Dessa forma usamos como referência as implementações do Shader Maker e o BRDF Explorer Tool. Ambos utilizam o OpenGL e o Qt para a renderização e entrada e saída de dados. O uso das duas ferramentas em conjunto permitem que estas funcionalidades sejam implementadas, motivo que levou este trabalho a utilizá-las para a construção do renderizador.

Como este trabalho pretende representar vários modelos diferentes de BRDF, foi necessária uma forma de implementar vários *shaders* diferentes sem modificar o código principal do

renderizador. E como foi visto na Seção III, as versões mais recentes do OpenGL permitem isso.

No OpenGL os *shaders* são os responsáveis por definir o comportamento da luz na malha renderizada. Dessa forma, é possível mudar as representações das BRDFs apenas modificando os arquivos que descrevem esses *shaders*.

Ainda assim, todos os valores das constantes da cena são entregues aos *shaders* pelo OpenGL. Isso implica que todas as constantes existentes nos *shaders* têm que estar mapeadas no renderizador. Este fator limita a possibilidade de um renderizador genérico, já que não é possível implementar este mapeamento sem saber como o *shader* está escrito.

IV. A Gramática

Dado o problema de representar computacionalmente as BRDFs implementadas nos *shaders* do renderizador, inicialmente este trabalho optou pelo estudo de Padrões de Projeto [7]. Porém, durante a pesquisa uma nova abordagem foi proposta, utilizando o conceito de compiladores e Gramáticas Livres de Contexto [3]. Esta nova abordagem mostrou-se mais flexível e por isso não utilizamos os conceitos de Padrões de Projeto no trabalho final.

Com base na gramática proposta pro Brady et al. [8] (vide Seção 2), uma solução mais sofisticada para nosso problema foi formulada. Utilizando a gramática proposta pelo artigo como base, tornou-se possível criar um parser [3] capaz de gerar o código fonte dos *shaders* utilizados. Para isto basta que a formulação matemática da BRDF seja dada ao parser, que avaliará se esta é admissível às regras da gramática. Se a formulação for admissível os *shaders* correspondentes àquela formulação serão criados.

Levando em consideração uma análise da gramática do artigo [8] e as necessidades da aplicação, uma nova Gramática 1 é proposta. Adicionamos o uso de operadores em-ordem, uma regra para a operação de atribuição e um *token* que simboliza o fator de Fresnel. Este fator foi implementado de acordo com a versão de Schlick [9].

Com isso, conseguimos o embasamento teórico necessário para criar um parser que aceitasse formulações de BRDFs. Para implementá-lo, utilizamos as ferramentas Flex e o GNU Bison. O Flex é um gerador de analisadores léxicos baseado no Lex, porém sobre a licença de software livre [10]. O GNU Bison é um gerador de parser que integra o GNU Project [10]. Por definição ele gera *parsers* do tipo LALR, sigla em inglês para *Look-Ahead Left-to-Right*.

V. Trabalhos Relacionados

Procuramos trabalhos onde foram feitos renderizadores que trouxessem as funcionalidades desejadas por esta pesquisa, que são:

- Representar visualmente diversos modelos analíticos de BRDFs.
- Permitir interação do usuário com a cena renderizada, movendo a câmera ou o objeto.
- Permitir que novas formulações de BRDFs possam ser adicionadas ao programa de maneira simples.

⟨*brdf*⟩ ::= ⟨*operation*⟩

⟨*operation*⟩ ::= ⟨*operation*⟩ ⟨*expression*⟩
| ⟨*operation*⟩ ⟨*statement*⟩
| ⟨*expression*⟩
| ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*variable*⟩ = ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*expression*⟩ ⟨*in-orderBinOp*⟩ ⟨*expression*⟩
| ⟨*post-orderBinOp*⟩ (⟨*expression*⟩ , ⟨*expression*⟩)
| ⟨*binVectorOp*⟩ (⟨*vector*⟩ , ⟨*vector*⟩)
| ⟨*unaryOp*⟩ (⟨*expression*⟩)
| (⟨*expression*⟩)
| ⟨*vector*⟩
| ⟨*variable*⟩
| ⟨*real*⟩

⟨*in-orderBinOp*⟩ ::= (+|-|*|/)

⟨*post-orderBinOp*⟩ ::= (min|max|pow|fresnel)

⟨*binVectorOp*⟩ ::= dot

⟨*unaryOP*⟩ ::= (-|sin|cos|tan|exp|asin|acos|atan|sqrt)

⟨*vector*⟩ ::= [inrohINROH]

⟨*variable*⟩ ::= [a-zA-Z]{-}[inrohINROH]|[a-zA-Z][a-zA-Z0-9]+

⟨*real*⟩ ::= [0-9]+(.[0-9]+)?

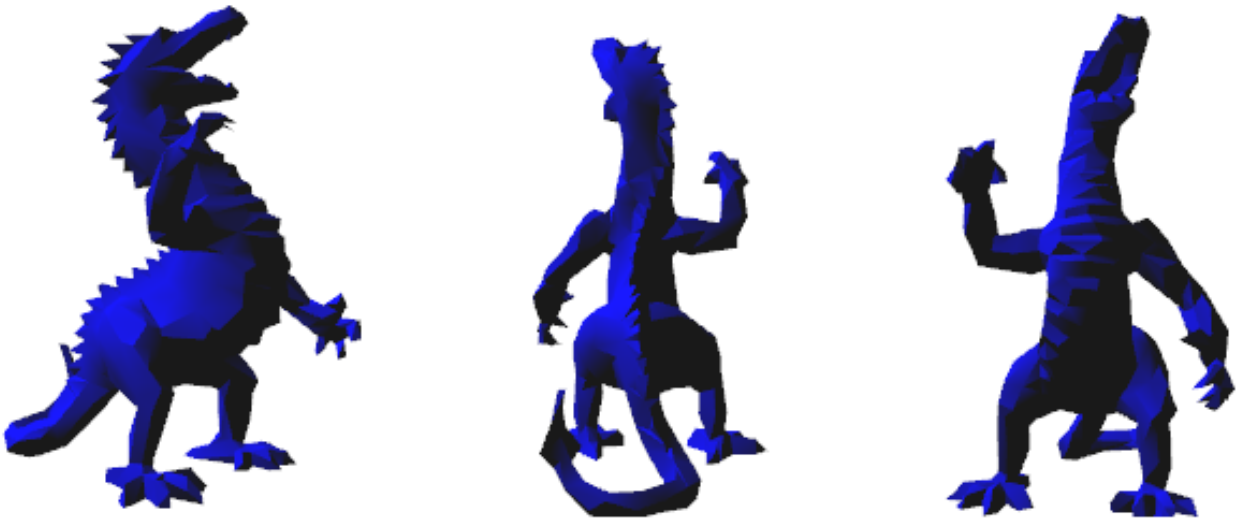
Gramática 1: Gramática criada para o parser

Os softwares encontrados neste artigo que melhor atendiam estes requisitos foram o BRDF Explorer Tool [11], e o Shader Maker [12]. Ambos dão ao usuário capacidade de modificar os valores das variáveis utilizadas pelo *shader*. Estas variáveis são utilizadas para descrever as propriedades do material da malha, como seu componente especular e sua cor. Como estas características são desejadas para nosso renderizador, eles foram usados como casos de estudo neste trabalho.

Durante a revisão bibliográfica foi estudado um novo artigo sobre a geração de formulações de BRDFs. Esta formulação era feita a partir de algoritmos genéticos, onde uma Gramática Livre de Contexto foi utilizada. A gramática tinha como função representar a formulação de constantes e operadores matemáticos encontrados nas BRDFs existentes [8]. Dessa forma, era possível validar as BRDFs geradas pelo algoritmo genético e estabelecer padrões de mutação para este algoritmo.

VI. Experimentos

Para testar o *parser*, propomos gerar com ele os *fragment* e *vertex shaders* seguindo a descrição da BRDF dos modelos de Blinn-Phong [13] e Lambertiano. O processo para ambos é descrito a seguir.



(a)

(b)

Figura 2: Imagens renderizadas utilizando o *shader* gerado pelo compilador: (a) modelo renderizado utilizando o *shader* do Lambertiano; (b) modelo renderizado utilizando o *shader* Blinn-Phong.

A. Lambertiano

O modelo Lambertiano é definido por um componente difuso, que segue a Lei de Lambert, definida na Equação 2.

$$d = L_\theta * cos_\theta \tag{2}$$

Sendo L_θ a constante difusa da superfície. O ângulo θ é dado pelo ângulo entre a normal N da superfície e o raio incidente I. Este valor pode ser obtido com a função dot. Sendo assim a função pode ser reescrita como a Equação 3.

$$d = L_\theta * dot(N, I) \tag{3}$$

Como é possível ver na Figura 2, o shader respeita a formulação dada do modelo Lambertiano.

B. Blinn-Phong

O modelo de Blinn-Phong é uma variante do modelo de Phong [13]. Seus termos difuso d e especular s são descritos nas As Equações 4 e 5.

$$d = max(0, N.I) \tag{4}$$

$$s = (N.H)^c \tag{5}$$

Onde:

- N é o vetor normal.
- I é o vetor da luz incidente.
- H é o halfway vector entre a luz incidente e o observador.
- c é a fator de brilho da superfície.

Para que este modelo possa ser aceito pela gramática criada algumas modificações devem ser feitas. O produto vetorial, na Equação descrito da forma $(vetor.vetor)$ deve ser alterado para $dot(vetor, vetor)$. Da mesma forma a potência entre fatores $(base^e)$ deve ser reescrita da forma $pow(base, e)$. Sendo assim as Equações 4 e 5 serão reescritas na forma das Equações 6 e 7.

$$d = max(0, dot(N, I)) \tag{6}$$

$$s = pow(dot(N, H), c) \tag{7}$$

O formato das Equações 6 e 7 é aceito pelo *parser*. Isso permite ao parser gerar o código do *shaders* de acordo com o casamento das regras da gramática. É importante frisar que os vetores utilizados pelas BRDFs já são calculados anteriormente pelos *shaders*, precisando apenas serem chamados quando for necessário.

Como é possível na Figura 2 ver os *shaders* respeitam a descrição do modelo de Blinn-Phong, apresentando seus componentes especular e difuso.

VII. Conclusão

O processo de implementar vários *shaders* baseados em BRDFs é um processo complexo. Neste trabalho apresentamos uma solução utilizando uma solução baseada em compiladores. Utilizando uma Gramática Livre de Contexto é possível reconhecer uma formulação matemática de BRDF e gerar um *shader* numa linguagem-algo específica.

Esta abordagem não exige do usuário conhecimento em linguagens de programação, podendo assim criar shaders baseados em BRDFs com pouco esforço. Para isso basta inserir a formulação matemática da BRDF com leves alterações, no caso de operadores como o exponencial.

Como próximos passos, testaremos mais exaustivamente as formulações de *shaders*, a fim de amadurecer a gramática. Também desenvolveremos uma interface gráfica capaz de dar ao usuário maior facilidade em alterar os dados da aplicação. Uma maneira de modificar os *shaders* em tempo de execução também será estudada.

Referências

[1] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis, "Radiometry," L. B. Wolff, S. A. Shafer, and G. Healey, Eds. USA: Jones and Bartlett Publishers, Inc., 1992, ch. Geometrical Considerations and Nomenclature for Reflectance, pp. 94–145. [Online]. Available: http://dl.acm.org/citation.cfm?id=136913.136929

[2] T. Weyrich, J. Lawrence, H. P. A. Lensch, S. Rusinkiewicz, and T. Zickler, "Principles of appearance acquisition and representation," *Found. Trends. Comput. Graph. Vis.*, vol. 4, no. 2, pp. 75–191, Feb. 2009.

[3] K. C. Louden, *Compiler Construction: Principles and Practice*. Boston, MA, USA: PWS Publishing Co., 1997.

[4] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th ed. Addison-Wesley Professional, 2013.

[5] K. Group, "Opengl," 2015. [Online]. Available: https://www.opengl.org/

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[8] A. Brady, J. Lawrence, P. Peers, and W. Weimer, "genbrdf: Discovering new analytic brdfs with genetic programming," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 114:1–114:11, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601193

[9] C. Schlick, "An inexpensive brdf model for physi-callybased rendering," *Computer Graphics Forum*, vol. 13, pp. 233–246, 1994.

[10] J. Levine, T. Mason, and D. Brown, *Lex & Yacc, 2Nd Edition*, 2nd ed. O'Reilly, 1992.

[11] Disney, "Brdf explorer," 2015. [Online]. Available: http://www.disneyanimation.com/technology/brdf.html

[12] U. Bremen, "Shader maker," 2015. [Online]. Available: http://cgvr.cs.uni-bremen.de/teaching/shader_maker/

[13] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, pp. 192–198, Jul. 1977.