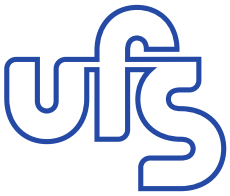


UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Desenvolvimento de um Compilador de BRDFs em LaTeXpara linguagem de shading GLSL, através da técnica Pratt Parsing

Trabalho de Conclusão de Curso

Everton Santos de Andrade Júnior

São Cristóvão – Sergipe

2024

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Everton Santos de Andrade Júnior

Desenvolvimento de um Compilador de BRDFs em LaTeXpara linguagem de shading GLSL, através da técnica Pratt Parsing

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Dra. Beatriz Trinchão Andrade
Coorientador(a): Dr. Gastao Florencio Miranda Junior

São Cristóvão – Sergipe

2024

Resumo

O presente trabalho propõe o desenvolvimento de um compilador de funções de distribuição de reflexão bidirecional (BRDFs) expressas em LaTeX para a linguagem de shading GLSL, utilizando a técnica de Pratt *Parsing* e linguagem de programação Odin. O objetivo é automatizar o processo de tradução de funções complexas de materiais, frequentemente descritas em equações LaTeX, para o código GLSL utilizado em programação de *shaders* para OpenGL. Ao fornecer essa ferramenta, pretende-se não apenas simplificar o trabalho dos desenvolvedores e pesquisadores na área de computação gráfica, mas também democratizar o acesso e compreensão de modelos de materiais complexos. Além disso, ao permitir que as BRDFs sejam expressas em uma forma mais familiar e acessível, como a notação matemática, o compilador reduz a barreira de entrada para aqueles que não estão familiarizados com linguagens programação, de modo a facilitar a colaboração interdisciplinar entre profissionais de diferentes áreas. A validação dos shader de saída do compilador proposto será feita através da ferramenta Disney BRDF Explorer, que auxilia na visualização de BRDF.

Palavras-chave: Compilador, BRDFs, LaTeX, GLSL, Shading, Pratt *Parsing*.

Sumário

1 Introdução 4
1.1 Contexto 4
1.2 Motivação 4
1.3 Objetivo 5
1.4 Estrutura do Documento 5

2 Conceitos 7
2.1 Radiometria 7
2.1.1 Energia Radiante e Fluxo 7
2.1.2 Radiância e BRDF 8
2.2 BRDF Models 10
2.2.1 Superfície Pura Especular 10
2.2.2 BRDF Difusa Ideal 10
2.2.2.1 BRDF Brilhante (*Glossy*) 10
2.2.2.2 BRDF Retro-Refletora 10
2.3 Compiladores 11
2.3.1 Cadeia de Símbolos e Alfabeto 11
2.3.2 Definições de Linguagens 11
2.3.3 Compilador como um Transformação 11
2.3.4 Gramática 11
2.3.4.1 Gramáticas Livres de Contexto (GLCs) 12
2.3.5 Análise Léxica 12
2.3.6 Análise Sintática ou *Parsing* 12
2.3.7 Pratt Parsing 13
2.3.7.1 Precedência de Expressões 13
2.3.7.2 Árvores Inclinadas 13
2.3.7.3 Pseudo-código para Análise de Expressões 14
2.3.8 Geração da Linguagem Alvo 15
2.4 Introdução ao Shading e ao Pipeline de GPU 15
2.4.1 Vertex Shader 16
2.4.2 Fragment Shader 17

3 Revisão Bibliográfica 18
3.1 Mapeamento Sistemático 18
3.1.1 Seleção das Bases 18
3.1.2 Questões de Pequisa 19

3.1.3 Termos de Busca . . . 19
3.1.4 Critérios . . . 20
3.1.4.1 Critérios de Inclusão . . . 20
3.1.4.2 Critérios de Exclusão . . . 20
3.1.5 Descrição dos Trabalhos Relacionados . . . 21
3.1.5.1 genBRDF: Discovering New Analytic BRDFs with Genetic Programming . . . 21
3.1.5.2 Slang: language mechanisms for extensible real-time shading systems . . . 22
3.1.5.3 Tree-Structured Shading Decomposition . . . 22
3.1.5.4 A Real-Time Configurable Shader Based on Lookup Tables . . . 23
3.2 Pesquisa por Repositórios online . . . 24

4 Metodologia . . . 25
4.1 Análise Abrangente . . . 25
4.2 Especificação da Linguagem . . . 25
4.3 Design de Casos de Teste . . . 26
4.4 Implementação do Compilador . . . 27
4.5 Experimentos de Renderização . . . 27
4.6 Plano de Continuação . . . 28

5 Resultados Iniciais . . . 30
5.1 Parser and Lexer in Odin . . . 30
5.1.1 Parser . . . 30
5.1.2 Gramática . . . 31
5.1.3 Tabela de Símbolos . . . 33
5.1.3.1 Estrutura de Símbolos . . . 33
5.1.3.2 Gerenciamento de Escopo . . . 34
5.1.3.3 Estrutura da Árvore de Sintaxe . . . 34
5.1.4 Implementação do Padrão de Visitante . . . 35
5.1.5 Testing . . . 35
5.1.6 Testes . . . 35
5.1.6.1 Geração de Árvore de Sintaxe . . . 36
5.1.6.2 Precedência de Operadores e Associatividade . . . 36
5.1.6.3 Interpretação Semântica . . . 36
5.2 *Ray Tracing* . . . 38
5.2.1 Implementação de Materiais . . . 38
5.2.2 Mecanismo de Reflexão de Raios . . . 39

1
Introdução

1.1 Contexto

Na computação gráfica, a representação realista de cenas tridimensionais depende fortemente da modelagem da luz e dos materiais que compõem os objetos na cena. [referencia] A interação da luminosidade incidente com esses materiais é crucial para a geração de imagens fiéis à realidade. Uma abordagem fundamental para modelar essa interação é por meio das funções de distribuição de refletância bidirecional, conhecidas como BRDFs (do inglês, *Bidirectional Reflectance Distribution Functions*).

As BRDFs, essencialmente, calculam a proporção entre a energia luminosa que atinge um ponto na superfície e como essa energia é refletida, transmitida ou absorvida [referencia]. Na renderização, essas funções são implementadas por meio programas especializados nas unidades de processamento gráfico (GPUs), esses programmas são chamados de *shaders*, e cada interface de programação, do inglês *Application Programming Interface* (API), de rederização disponibiliza etapas diferentes onde esses executaveis podem ser programados durante o processo de renderização. Esses *shaders* concedem a capacidade de cada objeto renderizado ter sua aparência configurada por meio de um código que implementa uma BRDF.

1.2 Motivação

Apesar da disponibilidade de linguagens específicas para a programação de shaders, que possibilitam a modificação procedimentos que representam uma BRDF, a aplicação de BRDFs na geração de *shaders* requer conhecimento especializado em programação [referencia?]. Essa barreira técnica pode restringir a exploração dos efeitos visuais por profissionais de áreas não relacionadas à programação. Diante disso, surge a necessidade de ferramentas mais acessíveis para a criação de *shaders*.

No meio acadêmico, as BRDFs são comumente descritas por fórmulas escritas em LaTeX, uma abordagem promissora para atender a essa necessidade é o desenvolvimento de um compilador capaz de traduzir BRDFs em LaTeXpara *shaders*, assim democratizando a visualização dessas BRDFs. Dado que as fórmulas são equações matemáticas, precisamos retrigir representação da linguagem de entrada para o compilador afim de garatir um projeto útil em tempo hábil.

1.3 Objetivo

Este trabalho visa projetar e implementar um compilador que, a partir de BRDFs escritas como equações em LaTeX, seja capaz de gerar código de *shading* na linguagem alvo da API OpenGL, considerando a precedência de operadores. O resultado será um *shader* capaz de reproduzir as características de reflexão da BRDF original ou, ao menos, alcançar uma aproximação satisfatória dessas características, levando em conta as limitações da linguagem de *shading* da API, principalmente as representações de dados de forma discreta.

1.4 Estrutura do Documento

No Capítulo 2, descrevemos todos os conhecimentos necessários para entender BRDFs, incluindo quantificação de luminosidade, radiação e conceitos de compiladores, como tokenização e construção da árvore sintática.

Especificamente na seção 2.1, tratamos dos conceitos fundamentais relacionados à luz, como a capacidade de um material refletir raios de luz e sua importância na computação gráfica e renderização. Destacamos a relação entre a intensidade de um pixel de imagem, a iluminação, a orientação da superfície e a definição de funções de refletância, as BRDFs. Já na seção 2.2, destacamos alguns modelos comuns de BRDFs.

A seção 2.3 fornece uma visão abrangente dos elementos essenciais na criação de compiladores. Ela começa com a definição de conceitos fundamentais, como cadeias de símbolos e alfabetos, necessários para entender linguagens formais. Além disso, a seção discute a importância das gramáticas na definição de linguagens e descreve o processo de compilação, incluindo a análise léxica, a análise sintática e o Pratt *Parsing*.

Na seção 2.4, é abordado o processo de *shading* e o funcionamento do *pipeline* de renderização da GPU. Nele, é descrito a transformação de vértices e a determinação da cor dos fragmentos, mostrando exemplos de código.

O Capítulo 3 tem como parte principal um mapeamento sistemático , utilizando termos de busca para identificar trabalhos relevantes sobre o desenvolvimento de compiladores para traduzir BRDFs de LaTeXpara *shaders*. Os critérios de inclusão e exclusão são definidos para filtrar os resultados. Além disso, são descritos os resultados encontrados em diversas bases de

dados, como IEEE Xplore, BDTD, CAPES, ACM Digital Library e Google Scholar, bem como a análise de repositórios online como GitHub e SourceForge.

No Capítulo 4 é descrito o método para desenvolver o compilador proposto, são definidas etapas para alcançar os objetivos especificados neste trabalho e casos de teste são projetados para validação. O plano de continuação, na seção 4.6, detalha as etapas futuras com datas previstas.

O Capítulo 5 descreve a implementação de um analisador léxico, sintático e interpretador na linguagem de programação Odin, includindo o *parsing* de Pratt. É priorizado simplicidade e clareza, com uma gramática simples. O capítulo também detalha testes desenvolvidos para validar a implementação. A seção 5.2 apresenta o desenvolvimento de um *ray tracer* em Odin com a biblioteca RayLib, modelando raios e materiais para renderização de imagens.

2
Conceitos

2.1 Radiometria

A radiometria trata de conceitos fundamentais relacionados à luz. Ela abrange a capacidade de um material de superfície receber raios de luz de uma direção e refleti-los em outra. No contexto da computação gráfica e renderização, a radiometria desempenha um papel crucial na compreensão do comportamento da luz em uma cena.

A intensidade de um pixel de imagem depende de vários fatores, como iluminação, orientação da superfície e refletância da superfície. A orientação da superfície é determinada pelo vetor normal em um ponto dado, enquanto a refletância da superfície diz respeito às propriedades materiais da mesma.

Para compreender e interpretar a intensidade de um pixel em uma imagem, é essencial compreender os conceitos radiométricos. A radiometria quantifica o brilho de uma fonte de luz, a iluminação de uma superfície, a radiância de uma cena e a refletância da superfície.

Renderização Além da Cor

Renderizar uma imagem envolve mais do que apenas capturar cor. Isso requer conhecimento da intensidade da luz em cada ponto da imagem, isto é, a quantidade de luz incidente na cena que alcança a câmera. A radiometria ajuda na criação de sistemas e unidades para quantificar a radiação eletromagnética, considerando a luz como fótons viajando em linha reta em um modelo óptico geométrico. Esse modelo simplifica considerações de difração e interferência, focando nos caminhos em linha reta dos fótons.

2.1.1 Energia Radiante e Fluxo

Vários processos físicos convertem energia em fótons, como radiação de corpo negro e fusão nuclear em estrelas. Quantificar a energia radiante total envolve entender a energia dos

fótons colidindo com um objeto, equivalente ao brilho da imagem. A energia radiante Q considera a energia total de todos os fótons atingindo a cena durante toda a duração.

$$
\begin{aligned}
Q &= \frac{hc}{\lambda} \\
h &\approx 6,626 \times 10^{-34} J \cdot s (\text{Joules por segundo}) \\
c &\approx 3,00 \times 10^{8} m \cdot s (\text{metros por segundo}) \\
\lambda &\approx 390 - 700 \times 10^{-3} m (\text{metros})
\end{aligned}
$$

É interessante observar a evolução da energia radiante Q ao longo do tempo, isso da origem ao fluxo radiantea ϕ, medida em impactos de cada fóton por segundo em uma superficie.

$$\phi = \frac{dQ}{dt} [\text{J/s}]$$

A irradiância E quantifica o número de impactos dos photons em uma superficie por segundo por unidade de área. Assim, tempos uma métrica mais específica e essencial para renderizar imagens com precisão.

$$E(p) = \frac{d\phi(p)}{dA} [\text{J/s} \cdot m^2]$$

2.1.2 Radiância e BRDF

A radiância, denotada como L, caracteriza a distribuição da luz em um ambiente ao longo de um raio definido por um ponto de origem e uma direção. A radiância desempenha um papel fundamental no cálculo do fluxo por unidade de área em uma superfície considerando toda a luz incidente de todas as direções possiveis em um dado ponto p.

$$L(p, w) = \frac{dE_w(p)}{dw} \qquad \left[\frac{J}{s \cdot m^2 \cdot \text{sr}}\right]$$

E_w é função de a irradiancia numa direção w

Para acomodar diferentes orientações da superfície e direção do raio, aplicamos o fator $\cos(\theta)$, tal que θ é o angulo entre a normal da superficie e a direção do w para obter a formula:

$$L(p, w) = \frac{dE(p)}{dw \cos(\theta)} = \frac{d^2\phi(p)}{dA dw \cos(\theta)}$$

A radiância pode fornecer informação sobre o quanto um ponto específico está iluminado na direção da câmera. Ela depende não apenas da direção do raio que incide câmera, mas também

das propriedades de refletância da superfície. E, no contexto de renderização, a radiância de uma superfície na cena se correlaciona com a irradiância de um pixel em uma imagem da sequinte forma:

$$E(p) = \int_{H^2} L(p, w) \cos(\theta) dw$$

H^2 é o hemisfério no plano tangente à superficie no ponto p

A principal funcionalidade de uma renderizador fotorealista é estimar a radiancia em um ponto p numa dada direção w_o. Essa radiancia é dada pela equação de renderização apresentada por @ref Kajiya. Note que essa euquação envolve um termo de radiancia recursiva, o caso base ocorre quando não há mais o termo recursivo, isto é, um fonte de luz na qual sua radiancia é contribuida apenas por randianca emitida L_e.

$$L_o(p, w_o) = L_e(p, w_o) + \int_{H^2} F(p, w_i, w_o) L_i(p, w_i) \cos(\theta) dw_i$$

L_o é radiancia de saída (*outgoing*) ou observada

L_e é radiancia emitida (i.e. fonte de luz)

L_i é radiancia incidente

w_i é a direção incidente

w_o é a direção de saída

H^2 são todas no hemisfério

θ angulo entre direção incidente e a normal da superficie

F função de refletancia

A Função de Distribuição Bidirecional de Reflectância (BRDF) descreve como a luz reflete de uma superficie em diferentes direções afetando a radiância de saída. Reflexão é o processo no qual a luz iterage com a superficie sem alterar a sua frequencia. Assim, BRDFs encapsulam as propriedades de reflexão de um material levando em conta vários fatores, como rugosidade da superfície, ângulo de incidência, angulo de reflexão Formalmente uma BRDF pode ser definida por $F(w_i, w_o)$, onde w_i é a direção indicente de luz e w_o é a direção de saída.

Para BRDFs fisicamente realistas algumas propriedades devem ser respeitadas.

- A propriedade de positividade, $F(\omega_i, \omega_o) \geq 0$, que garante não existência de energia negativa.
- Também, deve-se obedecer a reciprocidade de Helmhotz, $F(\omega_i, \omega_o) = F(\omega_o, \omega_i)$. Essa reciprocidade é usado na renderização, pois no lugar de traçar os raios da fonte de luz até a

camera, podemos traçar os raios da camera até a fonte de luz otimizando a maior parte dos raios traçados diretamente da fonte de luz que não iriam antigir a lente da camera, exitando desperdicio de poder computacional em raios que não contribuem para intensidade de um dado pixel.

- A BRDF deve, também, respeitar a conversação de energia, $\forall\omega_i, \int_{H^2} F(\omega_i, \omega_o)cos(\theta_o)d\omega_o \leq 1$. Nesse caso parte da energia pode ser absorvida, ou seja, transformado em outra forma de energia como calor, nesse caso esse somatório infintesimal pode no máximo chegar a 1, mas nunca ultrapassar.

2.2 BRDF Models

2.2.1 Superfície Pura Especular

Uma superfície puramente especular reflete a luz apenas em uma direção, seguindo a lei da reflexão. Ela produz reflexões nítidas, semelhantes a espelhos. A BRDF para uma superfície puramente especular é frequentemente representada pela função delta de Dirac $\delta(\omega_i - \omega_o)$, onde ω_i é a direção da luz incidente e ω_o é a direção refletida. $f(\omega_i, \omega_o) = k_s \cdot \delta(\omega_i - \omega_o)$ A função delta de Dirac garante que toda a luz incidente seja refletida na direção perfeitamente espelhada, resultando em uma reflexão altamente focada e intensa. Esse tipo de superfície é comum em materiais como metal polido ou vidro.

2.2.2 BRDF Difusa Ideal

Uma BRDFs difusa ideal reflete a luz incidente uniformemente em todas as direções, sem preferência por ângulos específicos. é representada por um termo cosseno lambertiano $\frac{\rho_d}{\pi} \cdot \cos\theta$, onde ρ_d é o albedo da superfície e θ é o ângulo entre a direção da luz incidente e a normal da superfície. $f(\omega_i, \omega_o) = \frac{\rho_d}{\pi} \cdot \cos\theta$ O termo cosseno garante que a radiância refletida seja proporcional ao cosseno do ângulo entre a direção da luz incidente e a normal da superfície. Esse modelo pode representar superfícies como tinta fosca ou papel.

2.2.2.1 BRDF Brilhante (*Glossy*)

Uma superfície brilhante exibe propriedades de reflexão tanto especulares quanto difusas. BRDF para uma superfície brilhante é frequentemente representada por uma combinação de termos especulares e difusos, como o modelo de Blinn-Phong @ref

2.2.2.2 BRDF Retro-Refletora

Uma superfície retro-refletora reflete a luz incidente de volta na direção de onde veio, independentemente do ângulo de incidência. A BRDF para uma superfície retro-refletora envolve

tipicamente geometria especializada ou revestimentos projetados para redirecionar a luz de volta para a fonte.

2.3 Compiladores

2.3.1 Cadeia de Símbolos e Alfabeto

Um **cadeia de símbolos** é uma sequência finita de símbolos retirados de um alfabeto Σ. Formalmente, um cadeia w é representado como $[w_1, w_2, ..., w_n]$, onde cada w_i pertence ao alfabeto Σ. O **alfabeto** Σ é um conjunto finito de símbolos distintos usados para construir cadeias em uma linguagem. Ele define os blocos de construção a partir dos quais cadeias válidas na linguagem são formadas.

2.3.2 Definições de Linguagens

Na ciência da computação, as linguagens são sistemas formais compostos por símbolos e regras que são muito uteis para definir um significado algoritimico. Uma **linguagem** L é definida como um conjunto de cadeias sobre um alfabeto finito Σ, $L \subseteq \Sigma^*$, onde Σ^* denota o conjunto de todas as cadeias possíveis sobre Σ. A estrutura e semântica de uma linguagem inclue seu alfabeto Σ, sintaxe e regras de gramática.

2.3.3 Compilador como um Transformação

Um compilador pode ser visto como um transformação entre linguagens L_1 e L_2 que preserva a estrutura interna dos cojuntos, isto é, deve manter o mesmo significado algoritmico. Assim, o compilador $C : L_1 \rightarrow L_2$ mapeia programas escritos na linguagem de origem L_1 para programas equivalentes na linguagem de destino L_2. Essa transformação garante a preservação semântica, mantendo o comportamento pretendido do programa original durante a tradução.

2.3.4 Gramática

Para auxiliar na criação de um compilador é necessario entender as regras que auxiliam na validação e geração da linguagem de interesse, esse entedimento pode ser alcaçando pela gramática. Uma gramática G é um sistema formal composto por um conjunto de regras de produção que especificam como cadeias válidas na linguagem podem ser geradas. Ela inclui terminais, não-terminais, regras de produção e um símbolo inicial.

- Terminais: Terminais são os símbolos básicos a partir dos quais as cadeias são formadas. Eles representam as unidades elementares da sintaxe da linguagem.
- Não-terminais: Não-terminais são espaços reservados que podem ser substituídos por terminais ou outros não-terminais de acordo com as regras de produção.

- Regras de Produção: As regras de produção definem a transformação ou substituição de não-terminais em sequências de terminais e/ou não-terminais.
- Símbolo Inicial: O símbolo inicial é um não-terminal especial a partir do qual a derivação de cadeias válidas na linguagem começa.

2.3.4.1 Gramáticas Livres de Contexto (GLCs)

Um tipo comum de gramática usado na definição de linguagens é a gramática livre de contexto (GLC). Em uma GLC $G = (V, \Sigma, R, S)$:

- V é um conjunto finito de símbolos não-terminais.
- Σ é um conjunto finito de símbolos terminais disjunto de V.
- R é um conjunto finito de regras de produção, cada regra no formato $A \rightarrow \beta$, onde A é um não-terminal e β é uma cadeia de terminais e não-terminais.
- S é o símbolo inicial, que pertence a V.

O processo de gerar uma cadeia na linguagem definida por uma gramática é chamado de derivação. Isso envolve aplicar regras de produção sucessivamente, começando pelo símbolo inicial S até restarem apenas símbolos terminais.

Uma árvore sintática é uma representação gráfica do processo de derivação, onde cada nó representa um símbolo na cadeia e cada aresta representa a aplicação de uma regra de produção, nos processos seguintes como análise sintatica,em código, essa árvore é gerada e usada como representação intermediária que auxilia na geração na linguagem alvo L_2.

2.3.5 Análise Léxica

A análise léxica, também conhecida como *lexing* ou *tokenization*, é a primeira etapa do processo de compilação, na qual a entrada textual é dividida em unidades léxicas significativas chamadas de *tokens*. Esses tokens representam os componentes básicos da linguagem, como palavras-chave, identificadores, operadores e literais. O analisador léxico percorre o código fonte caractere por caractere, agrupando-os em tokens conforme regras pré-definidas pela gramática da linguagem. No caso a linguagem do entrada analisador léxico são os formados por caracteres e, geralmente, são reconhecivel por maquinas de estado @ref, já a linguagem de saída é composta por tokens.

2.3.6 Análise Sintática ou *Parsing*

A análise sintática é a segunda fase do processo de compilação, na qual os tokens gerados pela análise léxica são organizados e verificados quanto à conformidade com a gramática da

linguagem. Essa etapa envolve a construção de uma árvore sintática ou estrutura de dados equivalente que representa a estrutura hierárquica das expressões e instruções do programa. O analisador sintático utiliza regras de produção gramatical para validar a sintaxe do código fonte e identificar possíveis erros.

2.3.7 Pratt Parsing

O Pratt Parsing, introduzida por Vaughan Pratt, é uma técnica de análise sintática recursiva que permite analisar expressões com precedência de operadores de forma eficiente e sem ambiguidades. Uma das características distintivas do Pratt Parsing é a maneira como lida com a precedência dos operadores, que é determinada pela ordem de avaliação das expressões. Ao contrário da análise descendente recursiva tradicional, onde cada não-terminal possui uma função de análise, a análise Pratt associa funções de manipulação (handlers) com tokens. Essas funções de manipulação são responsáveis por analisar expressões envolvendo seus respectivos tokens.

2.3.7.1 Precedência de Expressões

Na implementação do Pratt Parser, a precedência das expressões é definida por meio de uma tabela de precedência, na qual cada operador é associado a um nível de precedência. Isso permite que o parser decida dinamicamente a ordem de avaliação das expressões com base nos operadores encontrados durante a análise.

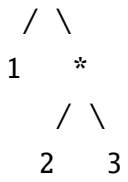
Essa abordagem simplifica significativamente a implementação do parser e elimina a necessidade de criar uma grammática que encapsula a precendencia em sua definição, também evita recursão profunda para lidar com diferentes níveis de precedência, tornando o Pratt Parsing uma técnica eficiente para análise sintática.

2.3.7.2 Árvores Inclinadas

No Pratt *Parsing*, a estrutura da árvore de expressão pode ser influenciada pela ordem de avaliação dos operadores. Essa distinção leva a dois tipos de árvores de expressão: árvores inclinadas à direita e árvores inclinadas à esquerda.

Árvore inclinada à Direita: Em uma árvore inclinada à direita, operadores com maior precedência são resolvidos primeiro, mesmo que apareçam mais tarde (para a direita) na expressão. Isso resulta em uma árvore onde os operadores com maior precedência estão mais próximos da raiz, indicando que eles são avaliados primeiro. Considere a expressão ‘1 + 2 * 3‘. Apesar de ‘*‘ aparecer após ‘+‘, ele tem uma precedência mais alta e, portanto, forma uma subárvore que é resolvida antes da adição. A árvore resultante é:

+



Árvore inclinada à Esquerda: Por outro lado, em uma árvore inclinada à esquerda, operadores com maior precedência são resolvidos por último, seguindo uma ordem de avaliação da esquerda para a direita. Isso significa que operadores com maior precedência formam subárvores que são resolvidas mais profundamente na árvore. As árvores inclinadas à esquerda estão tipicamente associadas a chamadas recursivas na análise.

Para alcançar a estrutura desejada da árvore, o Pratt parsing utiliza as estratégias de recursão e iteração com base na precedência dos operadores para saber o momento de gerar uma subarvores inclinada para esquerda ou direita. Operadores com precedência maior que a do operador atual formam a estrutura inclinada à direita, enquanto operadores com precedência menor formam a estrutura inclinada à esquerda.

2.3.7.3 Pseudo-código para Análise de Expressões

O pseudo-código 3, demonstra o Pratt *parsing* para a construção de árvores de expressão, considerando tanto estruturas inclinadas à direita quanto à esquerda. Esse algoritmo também é robusto mesmo quando um operador é tanto infixo quanto prefixo, por exemplo "−"pode ser um *token* de subtração ou de negação. Assim cada token tem uma função de prefixo e infixo associada.

Nesse algoritimo, **proximo_token()** recupera o próximo elemento da lista de tokens, **token.precedencia**() retorna a precedencia do token atual, **token.prefixo()** é a função associada ao token que realisa o parsing de uma expressão quando o token é o primeiro em uma subexpressão (e.g. o token "−"é o primeiro na expressão "−3"). Já o **token.infixo**(esquerda) é a função associada ao token uma função que cria um nó subárvore utilizando outra subarvore já criada como entrada para gerar expressão com operadores infixos, por exemplo a subarvore esquerda pode ser a expressão "−3", o token atual ser "∗"e o o retorno gera a expressão completa "−3 ∗ 1"

Tanto **token.infixo** quanto **token.prefixo** podem ser indiretamente recursivas, isto é, ambas podem chamar a função **expressao** no algortimo 3. Por fim, **precedencia_anterior** representa a precedência do token anterior, garantindo que os operadores sejam resolvidos na ordem correta.

Algoritmo 1 – Função Pratt Parsing de Expressão

```
function expressao(precedencia_anterior:=0):
    token := proximo_token()
    esquerda := token.prefixo()
    while precedencia_anterior < token.precedencia():
        token    = proximo_token()
        esquerda = token.infixo(esquerda)
    return esquerda
```

2.3.8 Geração da Linguagem Alvo

Nesta fase, fazemos a transição da representação intermediária da linguagem origem L_1 para a linguagem de destino L_2. O processo envolve traduzir construções da linguagem de origem L_1 em suas representações equivalentes na linguagem de destino L_2. Podemos realizar essa tradução ao percorrer recursivamente os nós da arvóre sintatica usando as informações contidas nesses nós para gerar partes do programa final em L_2.

Dado um programa $a \in L_1$ existem varios programas $b_{i=1,2,3,...} \in L_2$ que possui estrutura semanticamente equivalentes à a. Ao explorar esse conjunto, é possivel escolher um $b_j \in L_2$ tal que esse programa seja otimizado em algum sentido, como uso eficiente de memoria, ou executar menos instruções de *hardware*. Nosso foco neste trabalho está na tradução semanticamente correta, sem envolver explorão das saídas equivalentes.

Como exemplo, considere a tradução de um cálculo matemático de L_1 para L_2:

$$\mathbf{v} = (\mathbf{a} + \mathbf{b}) \cdot \mathbf{c} - (\mathbf{d} \times \mathbf{e})$$

Após a tradução da expressão matematica para L_2, o cálculo pode ser convertido para o trecho de programa abaixo. Esse código é válido na linguagem GLSL.

```
vec3 v = dot(a + b, c) - cross(d, e);
```

2.4 Introdução ao Shading e ao Pipeline de GPU

Shading refere-se ao processo de determinar a cor e o brilho dos pixels em uma imagem renderizada. Isso envolve simular a interação da luz com as superfícies, levando em consideração as propriedades dos materiais, condições de iluminação e orientação da superfície. Isso é alcançado por meio de pequenos programas chamados shaders, que são compilados e executados na unidade de processamento gráfico (GPU).

A interação com as GPUs é facilitada por meio de uma (API), em portugues, Interface de Programação de Aplicação, sendo o OpenGL uma API padrão para o uso de funções na GPU.

O OpenGL foi criado como um padrão de programação aberto e multiplataforma para GPUs. O pipeline de renderização do OpenGL é composto por várias etapas, incluindo definição de dados de vértices, shaders de vértice e fragmento, shaders de tesselação e geometria opcionais, configuração de primitivas, recorte, rasterização e shader de fragmento.

Essas etapas coordenam o fluxo de dados da CPU para a GPU e suas transformações, culminando na geração da imagem final. As etapas mais importantes para o nosso trabalho são o shading de fragmento e de vértice, os quais executam a manipulação dos vértices e determinar cores de pixels, respectivamente.

@image put in the overview image

2.4.1 Vertex Shader

O vertex shader opera em vértices individuais de primitivas geométricas antes de serem rasterizados em fragmentos. Sua principal tarefa é transformar vértices do espaço de coordenadas local do objeto para o espaço de coordenadas global da cena e passar os dados necessários para o fragment shader. Esses shaders geralmente realizam várias transformações nos dados dos vértices, permitindo que objetos sejam posicionados, orientados e projetados em uma tela 2D. Nesta seção, discutimos algumas das transformações comuns realizadas por shaders de vértice.

Ao fim dessa estapa, os vértices são normalizados para garantir coordenadas homogêneas. Coordenadas homogêneas são necessárias para realizar a projeção perspectiva e outros cálculos no pipeline de renderização.

```
// Exemplo de Shader de Vértice
#version 330 core

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;

uniform mat4 modelViewProjection;

out vec3 fragNormal;

void main() {
    vec3 manipulatedPosition = inPosition + (sin(gl_VertexID * 0.1) * 0.1);
    fragNormal = inNormal;
    gl_Position = modelViewProjection * vec4(manipulatedPosition, 1.0);
}
```

2.4.2 Fragment Shader

O Fragment Shader opera em fragmentos rasterizados produzidos pela etapa de rasterização. Sua principal responsabilidade é determinar a cor final de cada fragmento com base na iluminação, texturização e propriedades da superfície. O modelo mental é interpretar esse programa sendo repetido para todos os pixeis da imagem paralelamente, e ainda, esse programa recebe dados interpolados, como vértices e normais, ou seja, cada instancia desse programa possue entradas diferentes uma das outras. Esses valores interpolados são calculados usando interpolação bariêntrica em toda a superfície da primitiva.

Neste shader é onde as BRDFs devem ser implementadas para atingir um nível de shading mais preciso, pois temos mais dados do que os passados, devido à interpolação. Isso resulta em um nível de granularidade maior, considerando uma transição mais suave de um ponto para outro dentro de um triângulo.

@image put an image about verter shading and fragment shading

```
// Exemplo de Shader de Fragmento
#version 330 core

in vec3 fragNormal;
out vec4 fragColor;

void main() {
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

3
Revisão Bibliográfica

Para esta seção, será conduzida uma revisão literária abrangente com o objetivo de explorar trabalhos relacionados ao desenvolvimento de compiladores para tradução de BRDFs expressas em LaTeXpara a linguagem de shading, empregando, técnicas de parsing. O processo de busca será conduzido em duas etapas distintas. Primeiramente, será realizado um levantamento dos trabalhos existentes nas bases de dados com relevantes periodicos, anais de eventos, artigos e trabalhos.

Por fim, será realizada uma busca por produtos ou ferramentas similares no mercado, utilizando strings de busca específicas em repositórios digitais, especificamente GitHub, e SourceForge. Esses processos de busca permitirão identificar referências relevantes e estabelecer um panorama do estado da arte no campo dos compiladores de BRDFs para shaders, contribuindo para a compreensão do contexto acadêmico e prático no qual este trabalho se insere.

3.1 Mapeamento Sistemático

Com o intuito de obter resultados relevantes para a pesquisa, foram elaboradas frases de busca com base nos termos-chave relacionados ao tema deste trabalho. Assim como, foram criadas questões de pesquisa para guiar a seleção dos trabalhos.

3.1.1 Seleção das Bases

As bases escolhidas foram: ACM Digital Library, IEEE Xplorer Digital Library, Biblioteca Digital Brasileira de Teses e Dissertações (BDTD), Portal de Periódicos da CAPES, Google Acadêmico, esse foram escolhidos por serem acessíveis gratuitamente pela afiliação à Universidade Federal de Sergipe, já o google scholar foi escolhido para agregar pesquisas em outras bases que possam ter trabalhos relevantes.

<https://bdtd.ibict.br/> <https://ieeexplore.ieee.org/> <https://www-periodicos-capes-gov-br.ezl.periodicos.capes.gov.br/>

3.1.2 Questões de Pequisa

Foram elaboradas questões de pesquisa específicas, que guiam as frases-chave que refletem os principais aspectos do tema em questão. A partir desse processo, foram identificados e selecionados os trabalhos que melhor atendiam às questões propostas, garantindo maior relevância para o estudo em questão.

1. Quais são as abordagens mais comuns utilizadas na criação de compiladores para tradução de BRDFs expressas em alguma linguagem de texto, com LaTeX, para shaders?
2. Quais as técnicas de parsing que têm sido aplicadas no desenvolvimento de compiladores para linguagens matemáticas como LaTeX?
3. O trabalho utiliza arvores, ou gramáticas livre de contexto para representar uma BRDF?
4. Quais são os principais desafios enfrentados ao traduzir funções matemáticas complexas, como as BRDFs, em shaders?
5. Quais são as ferramentas e recursos disponíveis para auxiliar no desenvolvimento de compiladores para BRDFs e shaders, e como elas podem ser integradas ao processo de desenvolvimento?

3.1.3 Termos de Busca

As frases foram contruidas considerando suas variações equivalentes através de operadores lógicos. Posteriormente, as frases de pesquisa foram adaptadas de acordo com as características individuais de cada base de dados utilizada nas pesquisas. Os termos-chave escolhidos foram: "shader", "BRDF", "compiler", "parser"e "grammar".

Bases	**Termos de Pesquisa**	**Resultados**
IEEE Xplore Digital Library	("Full Text & Metadata":brdf) AND (("Full Text & Metadata":shader) OR ("Full Text & Metadata":shading)) AND (("Full Text & Metadata":compiler) OR ("Full Text & Metadata":parsing) OR ("Full Text & Metadata":parser) OR ("Full Text & Metadata":grammar))	36
BDTD	(Todos os campos:compiler OU Todos os campos:parsing OU Todos os campos:parser OU Todos os campos:compilador) E (Todos os campos:shader OU Todos os campos:shading) E (Todos os campos:brdf)	0
CAPES Periodico	Qualquer campo contém brdf E Qualquer campo contém compi* E shad*	0
ACM Digital Library	AllField:((shader OR shading) AND brdf AND (compiler OR compiling) AND (parser OR grammar OR parsing))	46
Google Academico	("BRDF"AND ("COMPILER"OR "COMPILING") AND("PARSER"OR "PARSING") AND ("SHADER"OR "SHADING"))	69

Tabela 1:

3.1.4 Critérios

Para garantir relevancia dos resultados obtivos, seguimos os critérios de inclusão e exclusão estabelecidos, de forma que os resultados serão filtrados. Ao fim desse procedimento, apenas os resultados com maior compatibilidade com este trabalho seram análisado e descritos de maneira mais detalhada.

3.1.4.1 Critérios de Inclusão

1. Foram incluídos artigos relacionados às palavras-chaves;
2. Foram incluídos artigos que de alguma forma inclua a criação de um compilador ou um parser;
3. Foram incluídos artigos que sintetize uma arvóre como representação de BRDFs

3.1.4.2 Critérios de Exclusão

1. Foram excluídos artigos dos quais dispunham de links incorretos e ou quebrados;
2. Foram excluídos artigos que dispunham de aplicacões muito similares/repetitivas;
3. Foram excluídos artigos que não respodem as questões de pesquisa **??**;
4. Artigo que não tem como entrada a BRDFs no formato de equação, ou seja, está utilizando a representação diretamente como código, também foi excluído.

5. Foram excluídos artigos que não consideram a geração de shaders como saída ou estrutura da BRDF em arvore.

6. Foram excluídos artigos que não citam BRDFs e compilador em seu resumo;

7. Se após a leitura completa, o artigo não concerne os interesse deste trabalho, esse foi excluído.

Bases	**Filtrados**
IEEE Xplore Digital Library	2
BDTD	0
CAPES Periodico	0
ACM Digital Library	1
Google Academico	1

Tabela 2: Resultados da Base após aplicar os critérios

3.1.5 Descrição dos Trabalhos Relacionados

3.1.5.1 genBRDF: Discovering New Analytic BRDFs with Genetic Programming

Neste artigo é introduzido uma framework chamada genBRDF, a qual aplica tecnicas de programação genética para explorar e descobrir novas BRDFs de maneira analitica. O processo inicia utilizando uma BRDF existente, e iterativamente aplica mutações e recombinações de partes das expressões matematicas que compões essas BRDFs a medida que novas gerações surgem. Essas mutações são guiadas por uma função fitness, que seria o inverso de uma função de erro, essa é baseada em um dataset de materiais já medidos. Por meio da avaliação de milhares de expressões, a framework identificas as viávies, que estão na Fronteira de Pareto.

A representação das BRDFs de entrada para o GA, autores geraram uma gramática que inclui constantes e operadores matemáticos comuns encontrados em equações BRDF. A gramática é compilada, e a árvore de sintaxe abstrata resultante passa por modificações realizadas pelo algoritmo genético. Nós na árvore podem ser trocados, substituídos, removidos e novos nós podem ser adicionados. Esse processo, após refinamento e análise, resulta em novas BRDFs.

Alguns dos novos modelos BRDF apresentados no documento incluem aqueles que superam os modelos existentes em termos de precisão e simplicidade.

Esse artigo se concentra principalmente em utilizar programação genética para descobrir automaticamente novos modelos analíticos de BRDF, em vez de compilar diretamente equações BRDF em linguagens de shading. Embora a representação das expressões das BRDFs possam potencialmente inspirar o nosso trabalho, o principal objetivo do artigo difere do objetivo de compilar equações BRDF para linguagem de shading.

3.1.5.2 Slang: language mechanisms for extensible real-time shading systems

O artigo descreve a linguagem Slang, uma extensão da amplamente utilizada linguagem de shading HLSL, projetada para melhorar o suporte à modularidade e extensibilidade. A abordagem de design da Slang é baseada em dois princípios fundamentais: manter a compatibilidade com o HLSL existente sempre que possível e introduzir recursos com precedentes em linguagens de programação mainstream para facilitar a familiaridade e intuição dos desenvolvedores.

O autor destaca que cada extensão da Slang visa fornecer uma trajetória incremental para a adoção a partir do código HLSL existente, evitando a necessidade de uma migração completa. Algumas dessas extensões são: funções generica, struct genericas, tipos que implementam uma dada interface assim como interfaces funcionam em Java nas para struct. Exemplo de funçao generica escrita em Slang:

```
float3 integrateSingleRay<B:IBxDF>(B bxdf,
SurfaceGeometry geom, float3 wi, float3 wo, float3 Li)
{ return bxdf.eval(wo, wi) * Li * max(0, dot(wi, geom.n)); }
```

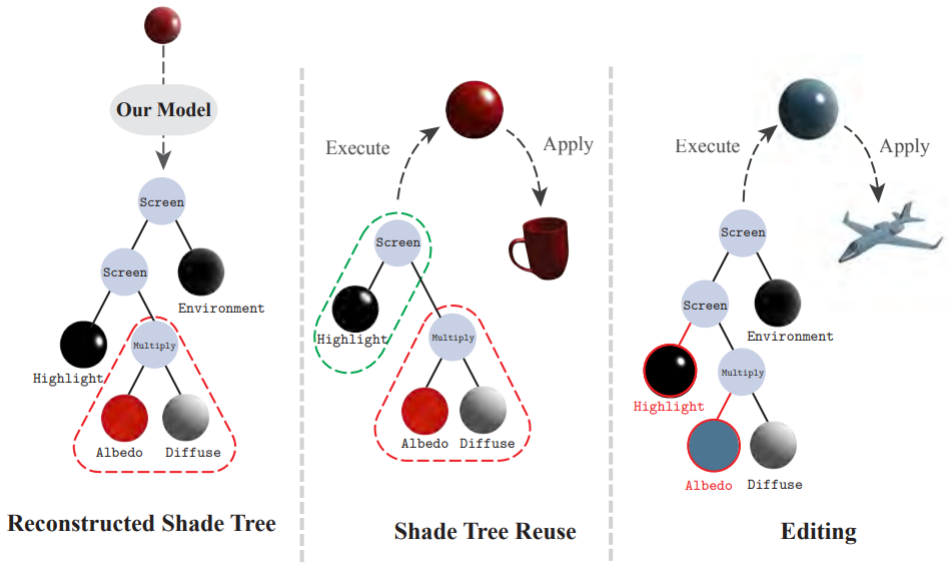
Enquanto o artigo se concentra na extensão de linguagens de shading existentes para melhorar a eficiência e a extensibilidade dos sistemas de shading em tempo real, o nosso trabalho se concentra na compilação de equações BRDF em linguagens de shading para explorar e descobrir novos modelos analíticos, mesmo para pessoas que não tem o conhecimento ténico da linguagem de shading espeficica. Embora ambos os projetos façam uso de shading e compilação, as abordagens e focos são diferentes.

3.1.5.3 Tree-Structured Shading Decomposition

O artigo propõe uma abordagem para inferir uma representação de BRDF estruturada em árvore a partir de uma única imagem para a sombreamento de objetos. Em vez de usar representações paramétricas ou medidas para modelar o sombreamento, como é comum, é proposta uma abordagem que utiliza uma representação em árvore de shading, combinando nós de sombreamento básicos e métodos de composição para decompor o sombreamento da superfície do objeto.

Essa representação permite que usuários inexperientes editem o sombreamento do objeto de maneira eficiente e intuitiva. Para abordar o desafio de inferir a árvore de sombreamento, é proposta uma abordagem híbrida que combina um modelo de inferência auto-regressivo para gerar uma estimativa aproximada da estrutura da árvore com um algoritmo de otimização para ajustar a árvore inferida. Experimentos são realizados em diversas imagens para demonstrar a eficácia da abordagem proposta.

Figura 1 – Exemplo de decomposição de BRDFs em nós de uma arvóre



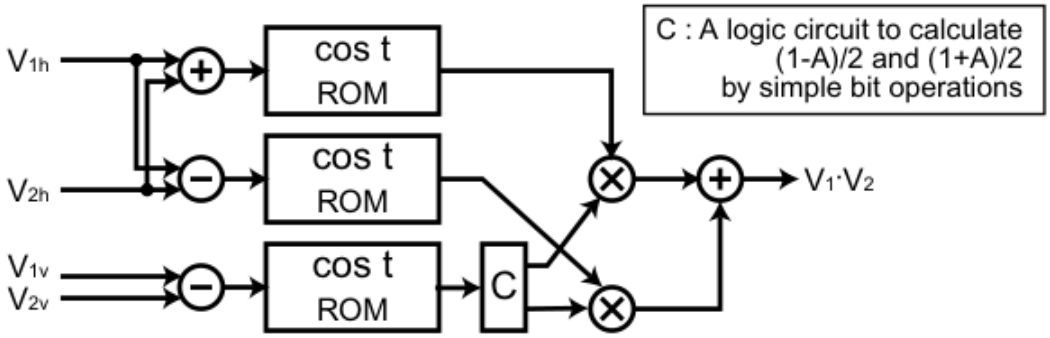
Fonte: **??**, p. 24)

Assim como o nosso trabalho, esse artigo se concentra em facilitar o processo para usuários inexperientes, pois ambos visam fornecer ferramentas acessíveis para manipular representações de sombreamento sem exigir conhecimento avançado em programação de shading. Esse artigo também emprega uma representação em árvore, embora para um propósito diferente. Enquanto o nosso trabalho utiliza árvores para representar expressões matemáticas de BRDFs, esse artigo utiliza a decomposição em nós de árvores para representar o shading parcial de objetos.

3.1.5.4 A Real-Time Configurable Shader Based on Lookup Tables

Este trabalho propõe uma arquitetura de hardware que permite cálculos de shading por pixel em tempo real, utilizando lookup-tables. Para isso, são projetados circuitos configurável baseado em lookup-tables, memórias de acesso aleatório (RAMs) e memórias somente leitura (ROMs). Varios circuitos base foram projetados visando realizar cálculos de shading, considerando as operações mais comuns, por exemplo, circuitos para calcular produto interno entre dois vetores, circuitos de rotação de um vetor por um angulo. Ademais, é usado interpolação em um sistema de coordenadas polares em vez da interpolação vetorial convencional com normalização, com o objetivo de reduzir o tamanho dos circuitos e melhorar o desempenho.

Figura 2 – Exemplo de circuito de produto interno entre vetores



Fonte: **??**, p. 24)

Além disso, o circuito suporta diversas BRDFs, como Blinn-Phong, Cook-Torrance, Ward e modelos baseados em microfacetes, com tabelas de lookup específicas para cada modelo. O uso de tabelas de pesquisa permite a representação organizada da parametrização das BRDFs, tornando o processo de transformação de BRDF para shaders mais acessível. Assim como este trabalho, a abordagem facilita a geração de shaders a partir da descrição de BRDFs, apesar da metologia ser diferente.

3.2 Pesquisa por Repositórios online

Também foram analisados repositórios no github e SourceForge, cada um com uma string de busca especifica. Os repositórios encontrados foram filtrados baseados em seus resumos, caso não haja a menção da criação de um compilador, ou não citar uma transformação de BRDF para outra estrutura, esse repositório será excluido.

Plataformas	**Termos de Pesquisa**	**Resultados**
GitHub	in:readme (GLSL AND BRDF AND (compiler OR compilation) AND (shader OR shading))	15
SourceForge	compiler bdrf	0

Tabela 3:

Após ler por completo os resumos dos repostiórios do GitHub, é evidente que nenhum desses projetos é relacionado com o proposto neste trabalhos, apesar de comentar sobre BRDFs, esses projetos não implementam compiladores, não fazem parsing de equações de BRDFs e nem mesmo geram shaders a partir de BRDFs.

4
Metodologia

A metodologia para desenvolver o compilador proposto envolve uma abordagem sistemática composta por etapas adaptadas para alcançar os objetivos de pesquisa especificados. Essas etapas incluem uma análise aprofundada das informações relacionadas pertinentes a BRDFs e compilação de shaders, exploração de técnicas existentes dentro do domínio, formulação de especificações de linguagem precisas, design e execução meticulosos de casos de teste abrangentes, a implementação real do compilador e a avaliação de seu desempenho por meio de experimentos de renderização.

4.1 Análise Abrangente

O primeiro passo envolve a realização de uma análise detalhada das áreas relacionadas ao desenvolvimento da ferramenta proposta. Isso inclui a revisão da literatura sobre BRDFs, linguagens de shaders, design de compiladores e técnicas de renderização gráfica. Isso envolve o estudo de artigos de pesquisa, ferramentas existentes e bibliotecas de software relevantes para identificar técnicas existentes e áreas potenciais para melhoria.

4.2 Especificação da Linguagem

As especificações da linguagem de entrada e saída para o compilador são definidas. A linguagem de entrada é uma versão simplificada do LaTeX, onde expressões matemáticas nos ambientes `equation` ou `align` são suficientes para os propósitos deste trabalho. O LaTeXé um sistema de composição amplamente utilizado para documentos matemáticos e científicos. Para descrever BRDFs usando equações LaTeX, enquanto a linguagem de saída especifica a linguagem de shader GLSL para renderização.

Para a linguagem de entrada, optamos por usar o ambiente `equation` devido à sua

simplicidade e uso generalizado na representação de expressões matemáticas. No LaTeX, os ambientes fornecem uma maneira de agrupar e formatar texto ou equações. O ambiente `equation` é especificamente projetado para exibir equações individuais. Abaixo está um exemplo de código LaTeXusando o ambiente `equation`:

```
\begin{equation}
    f(x) = ax^2 + bx + c
\end{equation}
```

Este código LaTeXrepresenta uma equação quadrática $f(x) = ax^2 + bx + c$, onde a, b e c são coeficientes.

Quanto à linguagem de saída, ela especifica a linguagem de shader GLSL para renderização. Um exemplo de código de shader GLSL correspondente gerado a partir da equação LaTeXacima pode ser:

```
float quadratic(float x, float a, float b, float c) {
    return a * x * x + b * x + c;
}
```

4.3 Design de Casos de Teste

Os casos de teste devem ser projetados para validar a correção e precisão do processo de tradução do compilador. Esses casos de teste emparelham equações LaTeXde entrada descrevendo BRDFs com o código de shader GLSL de saída esperado. Um caso específico que demonstra a capacidade do compilador pode ser criado com o BRDF de Cook-Torrance. A definição LaTeXe a saída esperada podem ser fornecidas como exemplo

LaTeX Input (Cook-Torrance BRDF):

```
\begin{equation}
f_{\text{CT}}(\omega_i, \omega_o) = \frac{D(h)F(\omega_i, h)G(\omega_i, \omega_o,
\end{equation}
```

GLSL Output esperado:

```
float CookTorrance(vec3 normal, vec3 viewDir, vec3 lightDir, float roughness, floa
    // halfway vector
    vec3 halfwayDir = normalize(viewDir + lightDir);

    // microfacet distribution function
```

```
    float D = ...; // D(h)

    // Calculate Fresnel term
    float F = ...; //F(wi, h)

    // Calculate geometric attenuation factor
    float G = ...; //G(wi, wo, h)

    float denominator = 4.0 * max(dot(viewDir, normal), 0.0) * max(dot(lightDir, n

    // Final BRDF value
    return (D * F * G) / denominator;
}
```

4.4 Implementação do Compilador

A implementação do compilador é feita usando a linguagem de programação Odin, escolhida por suas capacidades de linguagem de baixo nível com herança de composição e uma biblioteca padrão satisfatória. Técnicas de análise recursiva são utilizadas, especificamente o Parsing de Pratt, para processar as equações LaTeXde entrada e gerar o código de shader GLSL correspondente. Inicialmente, um lexer e um parser foram implementados para uma linguagem mais simples do que o LaTeXpara garantir que os fundamentos do compilador sejam funcionais, com precedência totalmente testada para a árvore de sintaxe e também a execução disso.

4.5 Experimentos de Renderização

Por fim, experimentos de renderização são realizados usando os shaders gerados pelo compilador. Isso permite a avaliação do desempenho e da qualidade visual das imagens renderizadas produzidas pelos shaders compilados. A plataforma escolhida para os testes é a ferramenta Disney BRDF, compilada localmente para modificar e adicionar shaders adicionais. Ele fornece uma interface para definir parâmetros que servem como controles deslizantes para os hiperparâmetros do BRDF.

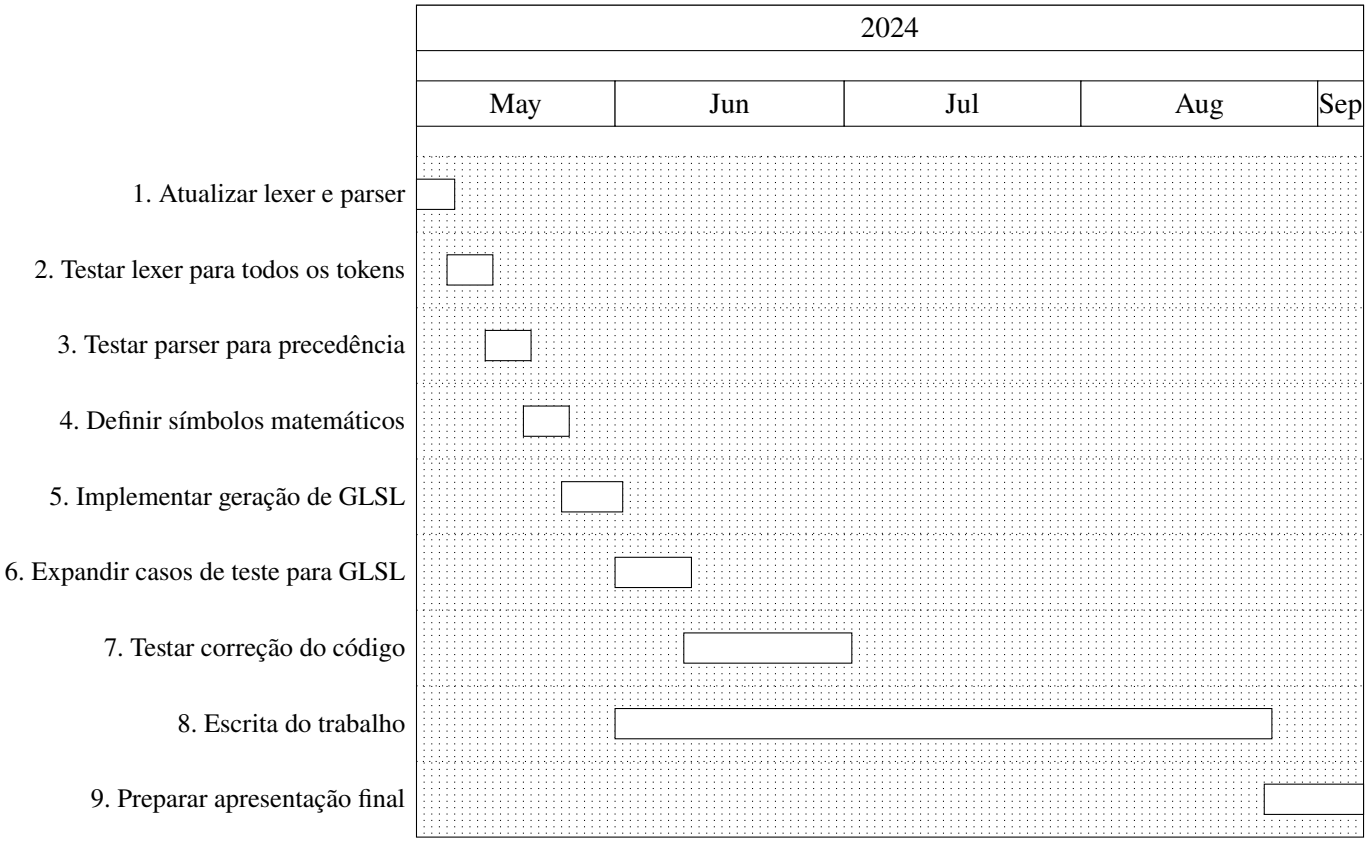
@ Imagem para a ferramenta @ Código para o superconjunto GLSL

Seguindo esta metodologia, a ferramenta proposta visa compilar efetivamente descrições de BRDF em shaders GLSL, facilitando a geração de gráficos realistas em aplicativos de gráficos por computador.

4.6 Plano de Continuação

A continuação desta tese envolve várias tarefas-chave destinadas a completar o desenvolvimento do compilador proposto para converter equações LaTeXque descrevem BRDFs em código de shader GLSL. As tarefas incluem atualizar o lexer e o parser escritos em Odin para aceitar equações LaTeX, testar o lexer e o parser para garantir o reconhecimento correto dos tokens LaTeXMath, testar o parser para garantir a árvore de sintaxe com a precedência correta, definir símbolos predefinidos como constantes matemáticas e outras quantidades, implementar o processo de geração de código GLSL usando a árvore de sintaxe usando o padrão de visitante @ref, expandir os casos de teste para cobrir uma melhor variedade de BRDFs, testar minuciosamente o código gerado quanto à correção e eficácia, incluindo as visualizações dos mencionados BRDFs em algumas cenas, preparar a apresentação final e, finalmente, escrever o documento do trabalho (TCC).

1. 06/05/2024 - 20/05/2024: Atualizar o lexer e o parser para aceitar equações LaTeX.
2. 06/05/2024 - 27/05/2024: Testar o lexer para garantir o reconhecimento correto de todos os tokens LaTeXMath.
3. 06/05/2024 - 27/05/2024: Testar o parser para garantir a árvore de sintaxe com a precedência correta.
4. 13/05/2024 - 03/06/2024: Definir símbolos predefinidos como constantes matemáticas e outras quantidades.
5. 20/05/2024 - 10/06/2024: Implementar o processo de geração de código GLSL.
6. 27/05/2024 - 17/06/2024: Expandir os casos de teste para cobrir uma ampla gama de BRDFs.
7. 03/06/2024 - 24/06/2024: Testar minuciosamente o código gerado quanto à correção e eficácia, tanto em código quanto em visualização.
8. 17/06/2024 - 06/09/2024: Escrever o documento da tese.
9. 10/06/2024 - 01/07/2024: Preparar a apresentação final.


2024
May
Jun
Jul
Aug
Sep
1. Atualizar lexer e parser
2. Testar lexer para todos os tokens
3. Testar parser para precedência
4. Definir símbolos matemáticos
5. Implementar geração de GLSL
6. Expandir casos de teste para GLSL
7. Testar correção do código
8. Escrita do trabalho
9. Preparar apresentação final

5

Resultados Iniciais

5.1 Parser and Lexer in Odin

We built a lexer, parser and interpreter for a simple language along with its grammar using Pratt parsing in Odin programming language, the repository can be found here <https://github.com/evertonse/pratt-parser>. The Pratt parser, is a recursive descente parser that means that each production rule as a associated parsing function. The implementation prioritizes simplicity and clarity, with extensive comments to aid understanding, specailly because it is meant to validate the parser for a ramp up, when we'll need to change to LaTeXtokens and de BRDF semantic meaning.

5.1.1 Parser

The Pratt parser, implemented in the ‘parser.odin‘ file, employs precedence climbing to efficiently handle arbitrary operator precedence levels. Unlike traditional recursive descent parsers, which often require multiple nested function calls for each precedence level, the Pratt parser organizes parsing functions hierarchically based on operator precedence.

Algoritmo 2 – Parte principal do parsing de expressão em código Odin, nessa implementação usamos a notação original de @ref(pratt) null_denotations and left_denotations que o mesmo que funções de parsing para operadore prefix e infixo respectivamente

```
1
2  parse_expr :: proc(prec_prev: i64) -> ^Expr {
3      /* expressions that takes nothing (null) as left operand */
4      left := parse_null_denotations()
5      /*
6      . if current token is left associative or current token has
           higher precedence
7      . than previous precedence then stay in the loop, effectively
           creating a left leaning
8      . sub-tree, else, we recurse to create a right leaning sub-tree.
9      */
10     for precedence(peek()) > prec_prev + associativity(peek())  {
11         /* expressions that needs a left operand such as postfix,
               mixfix, and infix operator */
12         left = parse_left_denotations(left)
13     }
14     return left
15 }
```

Functions such as ‘associativity‘ and ‘precedence‘ provide the necessary information for parsing operators correctly, ensuring the desired behavior in expressions.

@img(folder struture)

5.1.2 Gramática

A gramática é definida abaixo. Utilizamos uma notação leve de sintaxe para descrevê-la, todas as palavras em minúsculas são regras, o símbolo $*$ significa zero ou mais ocorrências, () indica agrupamento para aplicar um operador a ele, | é o operador ou, palavras entre aspas simples ″ significam literalmente um token com esse conteúdo como caractere, e todas as letras maiúsculas representam um token que pode ter conteúdo diferente, mas possui o mesmo significado semântico. Por exemplo, NUMBER pode ser 2.0 ou 1.0, mas nas regras de produção eles são tratados da mesma forma. O símbolo = indica uma produção.

```
start = assign* expr ;

expr = prefix expr
       | expr postfix
       | expr infix expr
       | expr '?' expr ':' expr
       | call
```

```
        | NUMBER
        | IDENTIFIER
        ;

assign = IDENTIFIER '=' expr ';'
         | IDENTIFIER '=' 'fn' (' params ')' expr ';'
         ;

call = expr '(' args ')';
args = expr
        | expr ',' args
        |
        ;
params = IDENTIFIER
         | IDENTIFIER ',' params
         |
        ;

postfix = '+' | '-' | '~' | '!';
prefix  = '+' | '-' | '~' | '!';
infix   = '+' | '-' | '*' | '/' | '^'
            | 'eq' | 'lt' | 'gt' | 'or' | 'and'
            ;
```

Essa gramática define regras para expressões, atribuições, chamadas de função e vários operadores, como postfix, prefix e infix com o intuito de criar uma vasta coleção de operadores com diferentes precendecias para facilitar quando for transicionado para sintaxe LaTeX.

Algoritmo 3 – Exemplo código da linguagem implementada

```
epsilon = 0.001; # proximidade em float

abs = fn(a) a lt 0 or a lt -0 ? -a : a;

float_close = fn(a, b)
    abs(a - b) lt epsilon ?
        1 : 0;

# Classical fibonacci
fib = fn(n)
    float_close(n, 0) ?
        0
    :float_close(n, 1) ?
        1
    :fib(n-1)  + fib(n-2);

fib(10)  # Ultima expressao significa retorno do main
```

```
1
cos = fn(a) a!; # Postfix '!' significa cosseno
sin = fn(a) !a; # Prefixo '!' significa seno
tan = fn(a) sin(a)/cos(a); # Temos acesso a procedimentos no escopo
   externo
val = tan(98);
val
```

5.1.3 Tabela de Símbolos

Nesse projeto também foi desenvolvido uma tabela de símbolos simples, que será reaproveitado na análise semântica e na geração de código GLSL futuramente. A

A implementação da tabela de símbolos fornecida aqui é baseada em uma estrutura de escopo hierárquico, onde cada escopo mantém um mapeamento entre os nomes dos símbolos e seus atributos correspondentes. A estrutura `Scope` representa um mapeamento de nomes de símbolos para objetos de símbolo dentro de um **único** escopo, enquanto a `Scope_Table` mantém uma pilha de escopos, permitindo aninhamento.

5.1.3.1 Estrutura de Símbolos

Cada símbolo na tabela de símbolos é representado pela estrutura `Symbol`, que contém os seguintes atributos:

- `name`: O nome do símbolo.

- `val`: O valor associado ao símbolo (para variáveis).
- `is_function`: Um sinalizador booleano indicando se o símbolo é uma função.
- `params`: Uma lista de tokens representando os parâmetros da função (se aplicável).
- `body`: Um ponteiro para a expressão que representa o corpo da função (se aplicável).

```
1
2  Scope :: #type map[string]Symbol
3  Scope\_Table :: [dynamic]Scope
4
5  Symbol :: struct  {
6      name : string,
7      val: f64,
8      is\_function: bool,
9      params: []Token,
10     body: ^Expr,
11 }
```

5.1.3.2 Gerenciamento de Escopo

A tabela de símbolos fornece funções para gerenciar escopos, incluindo:

- `scope_enter`: Entra em um novo escopo, anexando-o à pilha de escopos.
- `scope_exit`: Sai do escopo atual, removendo-o da pilha de escopos e retornando-o.
- `scope_reset`: Redefine a tabela de símbolos limpando todos os escopos.
- `scope_get`: Recupera um símbolo da tabela de símbolos pelo seu identificador.
- `scope_add`: Adiciona um novo símbolo ao escopo atual.

Essa tabela de símbolos será apdatada para a fase de geração de código e tradução adequada do código-fonte em shaders GLSL.

5.1.3.3 Estrutura da Árvore de Sintaxe

- **Ast**: A estrutura base para todos os nós da AST.
- **Start**: Representa o ponto de partida do programa, contendo uma sequência de atribuições seguida de uma expressão.
- **Assign**: Representa atribuições de variáveis, incluindo um identificador, operador de atribuição e expressão.

- **Assign_Function**: Estende Assign e representa definições de funções, incluindo parâmetros.
- **Expr**: Representa expressões, servindo como a estrutura base para diferentes tipos de expressões.
- **Expr_Identifier**: Representa identificadores dentro de expressões.
- **Expr_Number**: Representa literais numéricos dentro de expressões.
- **Expr_Grouped**: Representa expressões agrupadas dentro de parênteses.
- **Expr_Prefix**: Representa operações unárias de prefixo.
- **Expr_Infix**: Representa operações binárias de infixo.
- **Expr_Postfix**: Representa operações unárias de sufixo.
- **Expr_Mixfix**: Representa operações ternárias de mixfix.
- **Expr_Function_Call**: Representa chamadas de função com argumentos.

5.1.4 Implementação do Padrão de Visitante

O padrão visitante @ref foi empregado para percorrer e operar em uma árvore de sintaxe abstrata (AST). A estrutura da AST é definida com vários tipos de nó para capturar diferentes elementos da sintaxe. Três procedimentos incorporam o padrão de visitante para percorrer e manipular a AST:

- **walker_interp**: Interpreta a AST, calculando o valor numérico das expressões.
- **walker_paren**: Gera uma representação de string entre parênteses da AST, auxiliando na legibilidade e garantindo a ordem correta de avaliação.
- **walker_print**: Imprime os nós da AST e seus atributos, facilitando a depuração e compreensão da estrutura da AST.

5.1.5 Testing

5.1.6 Testes

Para garantir a correção e robustez da implementação do parser de Pratt, foi desenvolvida uma ampla suíte de testes. Esses testes abrangem vários aspectos da funcionalidade do parser, incluindo geração de árvore de sintaxe, precedência de operadores e interpretação semântica.

5.1.6.1 Geração de Árvore de Sintaxe

Um aspecto crucial dos testes envolve verificar a correta geração de árvores de sintaxe a partir de expressões de entrada. Os testes são projetados para cobrir diferentes cenários, incluindo operações aritméticas simples, expressões complexas com subexpressões aninhadas e chamadas de funções.

Por exemplo, a suíte de testes de geração de árvore de sintaxe inclui casos para validar:

- O manuseio correto de operadores unários e binários, garantindo a precedência e associatividade adequadas.
- A representação precisa de chamadas de função e seus argumentos dentro da árvore de sintaxe.
- O agrupamento adequado de expressões dentro de parênteses para impor regras de precedência.

5.1.6.2 Precedência de Operadores e Associatividade

Testes são conduzidos para validar a precedência e associatividade de operadores na gramática da linguagem. Isso inclui garantir que operadores com maior precedência sejam avaliados antes daqueles com menor precedência e que operadores associativos à esquerda ou à direita sejam analisados corretamente.

Exemplos de testes para precedência de operadores e associatividade incluem:

- Verificar a correta parantezização de expressões envolvendo múltiplos operadores com níveis de precedência variados.
- Garantir que operadores unários sejam aplicados antes de operadores binários, de acordo com suas regras de precedência.
- Validar o agrupamento adequado de expressões para impor associatividade à esquerda ou à direita conforme especificado na gramática.

5.1.6.3 Interpretação Semântica

Além da geração de árvore de sintaxe e precedência de operadores, testes são realizados para garantir a interpretação semântica de expressões produza os resultados esperados. Isso envolve avaliar expressões e comparar a saída com os valores antecipados.

Por exemplo, os testes de interpretação semântica abrangem:

- Avaliar expressões aritméticas envolvendo constantes, variáveis e chamadas de função para verificar resultados corretos.

- Verificar o comportamento de expressões condicionais (por exemplo, operador ternário) sob diferentes condições.
- Validar a correção das implementações de funções e seu comportamento dentro do ambiente de execução do parser.

Ao testar meticulosamente a geração de árvore de sintaxe, precedência de operadores e interpretação semântica, a implementação do parser de Pratt pode ser validada quanto à correção e confiabilidade, garantindo um desempenho robusto em vários cenários de entrada. 1. **Teste de Recursão**: Verifica a geração correta da sequência de Fibonacci até o 9º número. Por exemplo, verifica se o número de Fibonacci no índice 7 é calculado corretamente.

```
fib = fn(n)
    float_close(n, 0) ?
        0
    :float_close(n, 1) ?
        1
    :fib(n-1)  + fib(n-2);

fib(7)
```

2. **Teste de Atribuição e Interpretação**: Avalia uma expressão envolvendo atribuições de variáveis e chamadas de função no mesmo escopo. Por exemplo, verifica se a função ‘val‘ calcula corretamente a subtração de seus argumentos.

```
a = 1;
b = 2;
c = 3;
d = 4;

val = fn(a,b,c,d) d-c-b-a;
val(a,b,c,d, 2, 4 ,4)
```

3. **Teste de Operador Relacional**: Verifica a precedência e associatividade dos operadores relacionais (‘lt‘, ‘gt‘, ‘eq‘) e lógicos (‘or‘, ‘and‘). Por exemplo, verifica se a expressão ‘1 or 2 and 3 or 4‘ é corretamente parantezada.

```
1 or 2 and 3 or 4
```

4. **Teste de Precedência de Operadores Unários e Binários**: Examina a precedência dos operadores unários e binários. Por exemplo, verifica se a expressão ‘!a + b‘ é corretamente interpretada com o operador unário ‘¡ aplicado antes do binário ‘+‘.

```
!a + b
```

5. **Teste de Agrupamento**: Garante que as expressões entre parênteses sejam corretamente agrupadas. Por exemplo, verifica se a expressão ‘a + (b + c) + d‘ é parantezada conforme o esperado.

```
a + (b + c) + d
```

Esses testes, junto com outros na suíte, ajudam a validar a correção da implementação do parser de Pratt, cobrindo várias características da linguagem e regras de precedência de operadores.

5.2 *Ray Tracing*

Este capítulo apresenta o desenvolvimento e implementação de um simples *ray tracer* usando métodos estocásticos de colisão de raios na linguagem de programação Odin com a biblioteca RayLib (<https://www.raylib.com/>) para renderização de imagens em uma janela. Isso foi feito para começar a entender melhor BRDFs e a equação de renderização.

O *ray tracer* foi construído com instruções do livro <https://raytracing.github.io/books/RayTracingInOneWeekend.html>. Ele opera inteiramente na CPU, e a funcionalidade principal do *ray tracer* envolve a modelagem de raios e sua reflexão dos pixels da imagem para a cena. A cena consiste exclusivamente de esferas, empregando cálculos de colisão padrão entre um raio e uma esfera.

5.2.1 Implementação de Materiais

O *ray tracer* inclui vários materiais que ditam o comportamento dos raios ao interagir com superfícies, embora não garantidos de estar fisicamente vinculados, considerando a conservação de energia discutida no capítulo 2.1.2. Cada material é implementado como uma estrutura contendo um ponteiro de função de dispersão responsável por calcular a atenuação e o raio disperso após a interação com uma superfície. Os seguintes materiais são implementados:

- **Material Difuso**: Representa um material básico com refletância lambertiana.
- **Material Lambertiano**: Uma variante do material difuso com albedo personalizável.

- **Material Metálico**: Modela uma superfície metálica com reflexão especular, permitindo controle sobre a difusão.
- **Material Dielétrico**: Simula materiais transparentes com refração e reflexão com base no índice de refração.

```
Material :: struct {
    scatter: #type proc(self: ^Material, ray: Ray, hit: Hit) -> (attenuation: Colo
}

Shit_Diffuse_Material  :: struct {
    using _ : Material,
    albedo: Color,
}

Lambertian_Material :: struct {
    using _ : Material,
    albedo: Color,
}

Metal_Material :: struct {
    using _ : Material,
    albedo: Color,
    fuzz: f32,
}

Dielectric_Material :: struct {
    using _ : Material,
    ir: f32, // índice de refração
};
```

5.2.2 Mecanismo de Reflexão de Raios

O mecanismo central do *ray tracer* envolve traçar raios pela cena, determinar suas interações com superfícies e calcular os valores de cor resultantes. O processo de reflexão de raios consiste nos seguintes passos:

1. **Geração de Raios**: Raios são gerados a partir do ponto de vista da câmera e projetados na cena.

2. **Detecção de Colisão**: Cada raio é testado quanto à interseção com objetos na cena.

3. **Interação de Material**: Após a colisão, os raios interagem com o material da superfície, determinando atenuação e raios dispersos com base nas propriedades do material.

4. **Traçado Recursivo**: Se um raio se dispersa, o processo se repete, traçando o caminho do raio disperso até que uma profundidade máxima de recursão seja atingida ou o raio escape da cena.

5. **Acúmulo de Cor**: Os valores de cor são acumulados ao longo do caminho do raio, essa acumulação simula a irradiância de um certo ponto da superfície.

Referências