



UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Desenvolvimento de um Compilador de BRDFs em \LaTeX para 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 \LaTeX para
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

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 *shaders* de saída do compilador proposto será feita através da ferramenta Disney BRDF Explorer, que possibilita a visualização e análise de BRDFs.

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

Sumário

1	Introdução	6
1.1	Motivação	6
1.2	Objetivo	7
1.3	Estrutura do Documento	7
2	Conceitos	8
2.1	Radiometria	8
2.1.1	Energia Radiante e Fluxo	9
2.1.2	Radiância e BRDF	10
2.2	Modelos de BRDFs	12
2.2.1	BRDF Pura Especular	12
2.2.2	BRDF Difusa Ideal	13
2.2.3	BRDF <i>Glossy</i>	14
2.2.4	BRDF Retro-Refletora	14
2.3	Introdução ao Shading e ao <i>pipeline</i> de GPU	14
2.3.1	Shader de Vértice	15
2.3.2	Shader de Fragmento	16
2.4	Compiladores	17
2.4.1	Cadeia de Símbolos e Alfabeto	17
2.4.2	Definições de Linguagens	17
2.4.3	Compilador como um Transformação	17
2.4.4	Gramática	18
2.4.4.1	Gramáticas Livres de Contexto (GLCs)	18
2.4.5	Análise Léxica	19
2.4.6	Análise Sintática ou <i>Parsing</i>	19
2.4.6.1	Pratt Parsing	19
2.4.6.2	Pseudo-código para Análise de Expressões	19
2.4.7	Análise Semântica	20
2.4.8	Geração da Linguagem Alvo	21
3	Revisão Bibliográfica	22
3.1	Mapeamento Sistemático	22
3.1.1	Seleção das Bases	22
3.1.2	Questões de Pesquisa	23
3.1.3	Termos de Busca	23
3.1.4	Critérios	24

3.1.4.1	Critérios de Inclusão	24
3.1.4.2	Critérios de Exclusão	24
3.1.5	Descrição dos Trabalhos Relacionados	25
3.1.5.1	genBRDF: Discovering New Analytic BRDFs with Genetic Programming	25
3.1.5.2	Slang: language mechanisms for extensible real-time shading systems	25
3.1.5.3	Tree-Structured Shading Decomposition	26
3.1.5.4	A Real-Time Configurable Shader Based on Lookup Tables	27
3.2	Pesquisa por Repositórios Online	28
4	Metodologia	29
4.1	Análise e Técnicas	29
4.2	Especificação da Linguagem	30
4.3	Design de Casos de Teste	31
4.4	Implementação do Compilador	32
4.5	Experimentos de Renderização	33
5	Desenvolvimento	36
5.0.1	Desenvolvimento	37
5.0.2	Análise Semântica	39
5.0.2.1	Tabela de Symbolos	39
5.0.2.2	Inferência de Tipos	39
5.0.3	SVG da árvore abstrata com inferência de tipos	39
5.1	Análise Léxica	40
5.2	Análise Sintática	47
5.2.1	Parser	47
5.2.2	Gramática	48
5.2.2.1	Estrutura da Árvore de Sintaxe	51
5.3	Implementação do Padrão de Visitante (walker)	54
5.3.1	Validação de Precedência	54
5.3.2	Visualização da AST por Imagem	55
6	Resultados	59
6.1	Parser e Lexer em Odin	60
6.1.1	Parser	60
6.1.2	Gramática	61
6.1.3	Tabela de Símbolos	63
6.1.3.1	Estrutura de Símbolos	63
6.1.3.2	Gerenciamento de Escopo	64

6.1.3.3	Estrutura da Árvore de Sintaxe	64
6.1.4	Implementação do Padrão de Visitante	65
6.1.5	Testes	65
6.1.5.1	Geração de Árvore de Sintaxe	65
6.1.5.2	Interpretação Semântica	66
6.2	<i>Ray Tracing</i>	67
6.2.1	Implementação de Materiais	67
6.2.2	Mecanismo de Reflexão de Raios	67
Referências		71

1

Introdução

Na computação gráfica, a representação realista de cenas tridimensionais depende fortemente da modelagem da luz e dos materiais que compõem os objetos na cena. 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 (PHARR; JAKOB; HUMPHREYS, 2016). Na renderização, essas funções são implementadas por meio de programas especializados nas unidades de processamento gráfico (GPUs), chamados de *shaders*. Cada interface de programação, do inglês *Application Programming Interface* (API), disponibiliza etapas diferentes onde esses executáveis 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.

1.1 Motivação

Existem linguagens específicas para a programação de *shaders*, as quais permitem a modificação de procedimentos que representam uma BRDF. No entanto, essa aplicação requer conhecimento especializado em programação. 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 . Desta forma, uma abordagem promissora para simplificar a criação de *shaders* é o desenvolvimento de um compilador capaz de traduzir BRDFs escritas em \LaTeX para *shaders*. Isso permitiria uma

maior acessibilidade e democratização na criação de efeitos visuais complexos.

1.2 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. 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.

1.3 Estrutura do Documento

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

O [Capítulo 3](#) faz um mapeamento sistemático, utilizando termos de busca para identificar trabalhos relevantes sobre o desenvolvimento de compiladores para traduzir BRDFs de \LaTeX para *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. Esse capítulo também inclui o plano de continuação deste trabalho, que detalha as etapas futuras com datas previstas.

O [Capítulo 6](#) descreve os resultados preliminares deste trabalho, que consistem na implementação de um analisador léxico, sintático e interpretador na linguagem de programação Odin ¹, incluindo o método de análise sintática de Pratt. A linguagem desenvolvida possui uma gramática simplificada em comparação com \LaTeX , de forma a garantir o funcionamento dos algoritmos de análise léxica e sintática antes de avançar para uma linguagem mais complexa. O capítulo também descreve os testes elaborados para validar a implementação. Além disso, ele apresenta o desenvolvimento de um *ray tracer* em Odin, que modela raios e materiais para a renderização de imagens, utilizando a biblioteca Raylib ² para exibir as imagens renderizadas.

¹ [<https://odin-lang.org/>](https://odin-lang.org/)

² [<https://www.raylib.com/>](https://www.raylib.com/)

2

Conceitos

Neste capítulo, abordam-se os conceitos fundamentais da interação da luz com os materiais na computação gráfica. Destaca-se a importância da reflexão da luz, explorando as BRDFs e modelos comuns. Além disso, são discutidos elementos-chave na criação de compiladores e no processo de *shading* na GPU.

Especificamente na [seção 2.1](#), são apresentados os conceitos fundamentais relacionados à luz, como a capacidade de um material refletir raios de luz e sua importância na computação gráfica e renderização. Destaca-se 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](#), são destacados alguns modelos comuns de BRDFs.

A [seção 2.3](#) aborda o processo de *shading* e o funcionamento do *pipeline* de renderização na GPU. Nela, são introduzidos os processos de transformação de vértices e de determinação da cor dos fragmentos, mostrando exemplos de código.

Na [seção 2.4](#), é fornecida 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, é discutida a importância das gramáticas na definição de linguagens e é descrito o processo de compilação, incluindo a análise léxica, a análise sintática, o Pratt *Parsing* e análise semântica.

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 ([WOLFE, 1998](#)). No contexto da computação gráfica, a radiometria desempenha um papel crucial na compreensão do comportamento da luz em uma cena.

Na renderização, a intensidade de um pixel da imagem depende de vários fatores, como iluminação, orientação e refletância da superfície. A orientação da superfície é determinada pelo vetor normal em um dado ponto, 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.

Renderizar uma imagem envolve mais do que capturar cor (DISNEY; LEWIS; NORTH, 2000); requer conhecimento da intensidade de 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 um modelo simplificado no qual a luz é tratada como fótons que viajam em linha reta.

2.1.1 Energia Radiante e Fluxo

Vários processos físicos convertem energia em fótons, como radiação de corpo negro e fusão nuclear em estrelas (PROKHOROV; HANSEN; MEKHONTSEV, 2009). Quantificar a energia radiante total de uma cena é necessário para quantificar o brilho da imagem, que envolve entender a energia dos fótons colidindo com objetos (JUDICE; GIRALDI; KARAM-FILHO, 2019).

A Equação 2.1 expressa a energia radiante Q (PHARR; JAKOB; HUMPHREYS, 2016), que considera a energia total de todos os fótons atingindo a cena durante toda a duração, onde: $c \approx 3,00 \times 10^8$ m/s (metros por segundo) é a velocidade da luz; λ representa o comprimento de onda, uma variável que abrange o espectro visível, aproximadamente entre 389×10^{-3} m e 700×10^{-3} m; h denota a constante de Planck, aproximadamente $6,626 \times 10^{-34}$ J·s (joule-segundo).

$$Q = \frac{hc}{\lambda} \quad (2.1)$$

É interessante observar a evolução da energia radiante (Q) ao longo do tempo. Isso dá origem ao conceito de fluxo radiante ϕ , que é medido em impactos de cada fóton por segundo em uma superfície. Sua unidade é joules por segundo e está representada na Equação 2.2.

$$\phi = \frac{dQ}{dt} \text{ [J/s]} \quad (2.2)$$

A irradiância quantifica o número de impactos dos fótons em uma superfície por segundo por unidade de área. Mais precisamente, podemos definir a irradiância E ao considerar o limite do fluxo radiante ϕ diferencial por área A diferencial em um ponto p (PHARR; JAKOB; HUMPHREYS, 2016, 5.4.1). Assim, temos uma métrica mais específica para renderizar imagens com precisão. Sua fórmula é demonstrada na Equação 2.3.

$$E(p) = \frac{d\phi(p)}{dA} \left[\frac{\text{J}}{\text{s} \cdot \text{m}^2} \right] \quad (2.3)$$

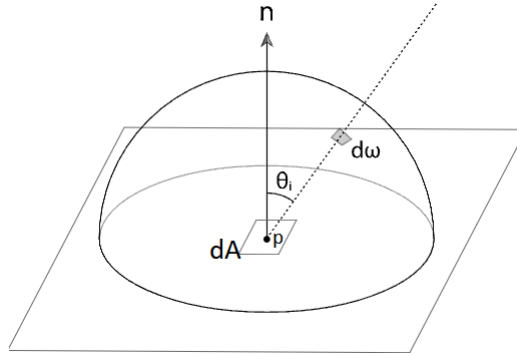
2.1.2 Radiância e BRDF

A radiância, denotada como L , caracteriza a densidade de fluxo por unidade de área A , por ângulo sólido ω (ver [Figura 1](#) para representação visual). Os ângulos sólidos representam a projeção da região no espaço sobre uma esfera unitária centrada em p , como ilustrado na [Figura 2](#). Ângulo sólido é a medida da área ocupada por uma região tridimensional conforme vista de um ponto específico p . Seu valor é expresso em esterradianos (sr), e são frequentemente representados pelo símbolo ω .

Assim, é possível definir radiância conforme a [Equação 2.4](#). Ao invés de usar diretamente a área A , a convenção estabelecida nessa definição é utilizar a projeção da área em um plano perpendicular à direção da câmera ([WEYRICH et al., 2009](#)).

$$L = \frac{d\Phi}{d\omega dA_{\perp}} \left[\text{W} \cdot \text{m}^{-2} \cdot \text{sr}^{-1} \right] \quad (2.4)$$

Figura 1 – Visualização da radiância em uma direção específica do hemisfério.

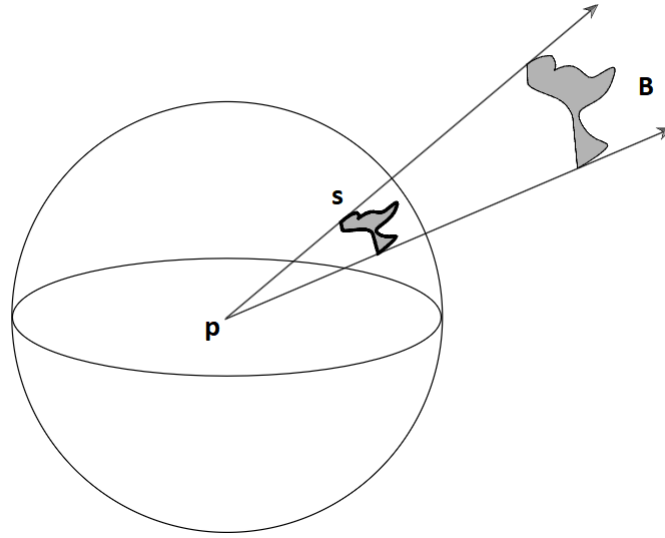


Fonte: ([PHARR; JAKOB; HUMPHREYS, 2016](#)). Adaptada.

Equivalentemente, podemos definir radiância para diferentes orientações da superfície e direção do raio ao introduzir o fator $\cos(\theta)$, tal que θ é o ângulo entre a normal da superfície e a direção ω ([PHARR; JAKOB; HUMPHREYS, 2016](#), 5.4.1). Essa definição é dada pela [Equação 2.5](#).

$$L(p, \omega) = \frac{d^2\phi(p)}{dA d\omega \cos(\theta)} = \frac{dE(p)}{d\omega \cos(\theta)} \quad (2.5)$$

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, mas também das propriedades de refletância da superfície. E, no contexto de renderização, a radiância de uma

Figura 2 – Ângulo sólido s do objeto B visto pelo ponto p .

Fonte: (PHARR; JAKOB; HUMPHREYS, 2016).

superfície na cena se correlaciona com a irradiância de um pixel em uma imagem pela [Equação 2.5](#). Isolando o termo $E(p)$, encontramos essa relação de maneira explícita na [Equação 2.6](#).

$$E(p) = \int_{H^2} L(p, \omega) \cos(\theta) d\omega \quad (2.6)$$

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

A principal funcionalidade de um renderizador fotorrealista é estimar a radiância em um ponto p numa dada direção ω_o . Essa radiância é dada pela [Equação 2.7](#), conhecida como equação de renderização apresentada por [Kajiya \(1986\)](#). Note que essa equação envolve um termo de radiância recursivo; o caso base ocorre quando não há mais o termo recursivo, isto é, a radiância é contribuída apenas por radiância emitida L_e , como ocorre com fontes de luz.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{H^2} f(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos(\theta_i) d\omega_i$$

L_o é radiância de saída (*outgoing*)

L_e é radiância emitida pela superfície (i.e. fonte de luz)

L_i é radiância incidente na superfície

ω_i é a direção incidente

ω_o é a direção de saída

H^2 são todas as direções no hemisfério no ponto p

θ_i ângulo entre direção incidente e a normal da superfície

f função de refletância

(2.7)

A Função de Distribuição Bidirecional de Reflectância (BRDF) descreve como a luz reflete de uma superfície em diferentes direções, afetando a radiância de saída (MONTES; UREÑA, 2012). Assim, BRDFs encapsulam as propriedades de reflexão de um material, considerando fatores como a rugosidade da superfície, o ângulo de incidência e o ângulo de reflexão. Formalmente uma BRDF pode ser definida por $f(\omega_i, \omega_o)$, onde ω_i é a direção incidente de luz e ω_o é a direção de saída. Para BRDFs fisicamente realistas, algumas propriedades devem ser respeitadas (PHARR; JAKOB; HUMPHREYS, 2016, 5.6):

- A positividade, $f(\omega_i, \omega_o) \geq 0$, que garante não existência de energia negativa.
- A reciprocidade de Helmholtz, $f(\omega_i, \omega_o) = f(\omega_o, \omega_i)$, é o princípio que indica que a função de refletância de uma superfície permanece inalterada quando os ângulos de incidência e reflexão da luz são trocados. Isso é utilizado na otimização do traçado de raios durante a renderização, permitindo traçar os raios da câmera para a fonte de luz. Essa abordagem evita o desperdício computacional em raios que não contribuem significativamente para a intensidade de um pixel na imagem final.
- A conservação de energia, expressa por $\forall \omega_i, \int_{H^2} f(\omega_i, \omega_o) \cos(\theta_o) d\omega_o \leq 1$, implica que parte da energia pode ser absorvida, transformando-se em outras formas de energia, como calor. Portanto, a soma infinitesimal pode atingir no máximo 1, mas nunca ultrapassá-la.

2.2 Modelos de BRDFs

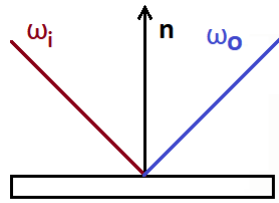
As próximas seções apresentam alguns modelos comuns de BRDFs na literatura (MONTES; UREÑA, 2012).

2.2.1 BRDF Pura Especular

Uma superfície puramente especular reflete a luz apenas em uma direção, seguindo a lei física da reflexão (ZEYU et al., 2017), ela produz reflexões nítidas, semelhantes a espelhos. A BRDF para essa superfície é frequentemente representada pela Equação 2.8, onde ω_i é a direção da luz incidente, ω_o é a direção refletida e δ é a função delta de Dirac que garante que toda a luz incidente seja refletida na direção perfeitamente espelhada como na Figura 3. Esse tipo de superfície é comum em materiais como metal polido ou vidro.

$$f(\omega_i, \omega_o) = k_s \cdot \delta(\omega_i - \omega_o) \quad (2.8)$$

Figura 3 – Reflexão especular. Em vermelho está o raio incidente, e em azul o raio de saída.



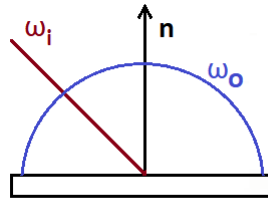
Fonte: Autor.

2.2.2 BRDF Difusa Ideal

Uma BRDF difusa ideal reflete a luz incidente uniformemente em todas as direções, sem preferência por ângulos específicos. É representada pela função f na Equação 2.9, onde ρ_d é o albedo da superfície e θ é o ângulo entre a direção da luz incidente e a normal da superfície. 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, como ilustrado na Figura 4. Esse modelo pode representar superfícies como tinta fosca ou papel.

$$f(\omega_i, \omega_o) = \frac{\rho_d}{\pi} \cdot \cos \theta \quad (2.9)$$

Figura 4 – Reflexão Difusa. Note que os raios refletidos não dependem do ângulo de entrada.

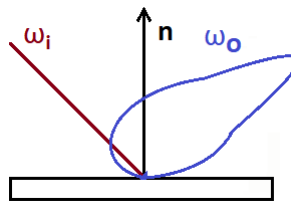


Fonte: Autor.

2.2.3 BRDF Glossy

Uma superfície pode exibir propriedades de reflexão tanto especulares quanto difusas, como na Figura 5. Uma BRDF para uma superfície brilhante é frequentemente representada por uma combinação de termos especulares e difusos, como o modelo de Blinn-Phong (TAN, 2020).

Figura 5 – Reflexão glossy.

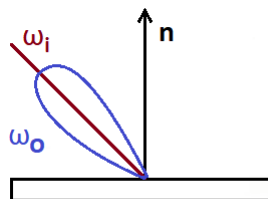


Fonte: Autor.

2.2.4 BRDF Retro-Refletora

Uma superfície retro-refletora reflete a luz incidente de volta na direção de onde veio, como na Figura 6. A BRDF para uma superfície retro-refletora envolve tipicamente geometria especializada ou revestimentos projetados para redirecionar a luz de volta para a fonte.

Figura 6 – Reflexão retro-refletora.



Fonte: Autor.

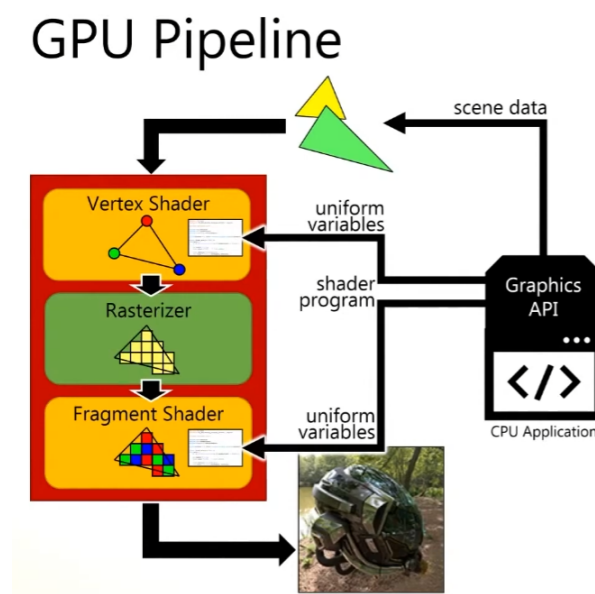
2.3 Introdução ao Shading e ao *pipeline* de GPU

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, sendo o OpenGL uma API padrão para o uso de funções na GPU (OpenGL Architecture Review Board, 2017). 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 e rasterização.

Essas etapas coordenam o fluxo de dados da CPU para a GPU e suas transformações, culminando na geração da imagem final. Uma representação visual desse processo pode ser observada na Figura 7. Nela, é representado a CPU enviando os dados da cena para a GPU, que utiliza essas informações nos *shaders* de vértice e fragmento. O *shader* de vértice manipula os vértices da cena, enquanto o *shader* de fragmento determina as cores dos pixels. Os fragmentos são elementos gerados durante o processamento das primitivas geométricas, como triângulos. Eles correspondem a pontos discretos na tela onde a cor final será determinada. Além disso, a CPU também pode enviar variáveis uniformes (*uniform variables*) para os *shaders*, que são essenciais para a etapa de renderização e contribuem para a geração da imagem final.

Figura 7 – O *pipeline*.



Fonte: (CEM, 2020).

2.3.1 Shader de Vértice

O *shader* de vértice opera em vértices individuais de primitivas geométricas antes de serem rasterizados em fragmentos. Sua principal tarefa é transformar vértices e passar os dados necessários para o *shader* de fragmentos. Esses *shaders* geralmente realizam várias transformações nos dados dos vértices, permitindo que objetos sejam posicionados, orientados e projetados em uma tela bidimensional (2D). Um exemplo desse *shader* está no [Código 1](#), que usa uma matriz para realizar essas transformações. Ao fim dessa etapa, os vértices são normalizados para coordenadas homogêneas. Essa normalização é essencial para realizar a projeção perspectiva e outros cálculos no *pipeline* de renderização.

Código 1 – Exemplo GLSL de *shader* de vértice.

```
1 #version 330 core
2 layout(location = 0) in vec3 inPosition;
3 layout(location = 1) in vec3 inNormal;
4
5
6 uniform mat4 modelViewProjection;
7
8
9 out vec3 fragNormal;
10
11
12 void main() {
13     vec3 manipulatedPosition = inPosition + (sin(gl_VertexID * 0.1)
14         * 0.1);
15     fragNormal = inNormal;
16     gl_Position = modelViewProjection * vec4(manipulatedPosition,
17         1.0);
18 }
```

2.3.2 Shader de Fragmento

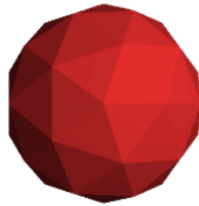
O *shader* de fragmento opera sobre os fragmentos produzidos pela etapa de rasterização. Sua principal responsabilidade é determinar a cor final de cada fragmento com base na iluminação, texturização e propriedades da superfície. Uma possível interpretação é que esse programa é repetido para todos os pixels da imagem paralelamente. Ele recebe dados interpolados, como vértices e normais, ou seja, cada instância desse programa possui entradas potencialmente diferentes uma das outras. Na API OpenGL, valores como normais e vértices são interpolados usando coordenadas baricêntricas ([The Khronos Group, 2015](#)).

As BRDFs podem ser implementadas nesse estágio do *pipeline* para atingir um nível de *shading* mais preciso, pois é possível ter mais dados do que os definidos na geometria devido à interpolação. Isso resulta em um nível de detalhamento potencialmente maior, considerando

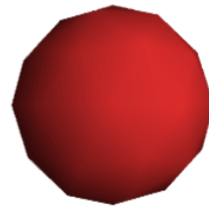
uma transição mais suave de um ponto para outro dentro de um triângulo, como representado na [Figura 8](#).

Figura 8 – Diferença entre shading a nível de vértice e shading a nível de fragmento.

Shading para cada vértice



Shading para cada fragmento



Fonte: ([DAVISONPRO, 2024](#)).

2.4 Compiladores

2.4.1 Cadeia de Símbolos e Alfabeto

Um **cadeia de símbolos** é uma sequência finita de símbolos retirados de um alfabeto Σ . Formalmente, uma cadeia w é representada como $[w_1, w_2, \dots, w_n]$, 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.

2.4.2 Definições de Linguagens

Na ciência da computação, as linguagens são sistemas formais compostos por símbolos e regras que são muito úteis para definir um significado algorítmico. 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 ([JÄGER; ROGERS, 2012](#)). A estrutura semântica de uma linguagem inclui seu alfabeto Σ , sintaxe e regras de gramática.

2.4.3 Compilador como um Transformação

Um compilador pode ser visto como uma transformação entre linguagens L_1 e L_2 que preserva a estrutura interna dos conjuntos, isto é, deve manter o mesmo significado algorítmico. 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.

2.4.4 Gramática

Durante a criação de um compilador, é necessário entender as regras que auxiliam na validação da linguagem de entrada, essas regras podem ser formalizadas 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 (JÄGER; ROGERS, 2012). Ela inclui terminais, não-terminais, regras de produção e um símbolo inicial.

- **Terminais:** são os símbolos básicos a partir dos quais as cadeias são formadas. Eles representam as unidades elementares da sintaxe da linguagem.
- **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:** definem a transformação ou substituição de não-terminais em sequências de terminais e/ou não-terminais.
- **Símbolo Inicial:** é um não-terminal especial a partir do qual a derivação de cadeias válidas na linguagem começa.

2.4.4.1 Gramáticas Livres de Contexto (GLCs)

Um tipo comum de gramática usado na definição de linguagens é a gramática livre de contexto (GLC). Uma GLC pode ser descrita formalmente como $G = (V, \Sigma, R, S)$:

- 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. As arestas representam a aplicação de regras de produção. Em código, essa árvore é gerada e usada como representação intermediária, auxiliando na geração da linguagem alvo L_2 .

2.4.5 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. Essa linguagem é, geralmente, reconhecível por máquinas de estado (RABIN, 1967).

2.4.6 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.

2.4.6.1 Pratt Parsing

O Pratt *Parsing*, introduzido por Vaughan Pratt, é uma técnica de análise sintática recursiva que permite analisar expressões com precedência de operadores de forma eficiente e sem ambiguidades (PRATT, 1973). Uma das suas características distintivas é determinar a ordem de avaliação das expressões. Ao contrário da análise descendente recursiva tradicional, na qual cada não-terminal possui uma função de *parsing*, a análise Pratt associa funções de manipulação (*handlers*) com *tokens*.

A precedência das expressões é definida por meio de uma tabela, na qual cada operador é associado a um valor que permita o *parser* decidir 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 gramática que encapsula a precedência em sua definição. Ela também evita a recursão profunda para lidar com diferentes níveis de precedência.

2.4.6.2 Pseudo-código para Análise de Expressões

O pseudo-código 1 demonstra o Pratt *parsing* para a construção de árvores de expressão. Esse algoritmo também é robusto mesmo quando um operador é tanto infixado quanto prefixado, por exemplo “-” pode ser um *token* de subtração ou de negação. Assim, cada *token* tem uma função de prefixo e infixado associada.

Nesse algoritmo, `proximo_token()` recupera o próximo elemento da lista de *tokens*, `token.precedencia()` retorna a precedência do token atual, `token.prefixo()` é a função associada

ao *token* que faz o *parsing* de uma expressão quando o *token* é o primeiro símbolo em uma subexpressão (e.g. o token “-” é o primeiro na expressão “-3”). Enquanto o **token.infixo(esquerda)** é a função associada ao *token* que utiliza outra subárvore já criada como entrada. Por exemplo a subárvore **esquerda** pode ser a expressão “-3”, o *token* atual ser “*” e 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 **expressão** no [Algoritmo 1](#). Por fim, **precedencia_anterior** representa a precedência do *token* anterior.

Algoritmo 1 – Função Pratt Parsing de Expressão.

```
1 function expressao(precedencia_anterior:=0):  
2     token := proximo_token()  
3     esquerda := token.prefixo()  
4     while precedencia_anterior < token.precedencia():  
5         token = proximo_token()  
6         esquerda = token.infixo(esquerda)  
7     return esquerda
```

2.4.7 Análise Semântica

A análise semântica é uma etapa essencial no processo de compilação, responsável por garantir a correte semântica das declarações e instruções do programa. Durante essa fase, são aplicadas verificações para garantir que as operações sejam realizadas com tipos compatíveis.

Um exemplo típico de verificação semântica é a inferência de tipos em expressões. Por exemplo, no [Código 2](#) o analisador semântico infere que o número inteiro 30 deve ser convertido para o tipo *float* antes da multiplicação, garantindo consistência de tipos. Além da verificação de tipos, o analisador semântico identifica e reporta outros erros comuns, como variáveis não declaradas e falhas no controle de fluxo do programa.

Código 2 – Exemplo de código escrito em C.

```
1 float x = 10.1;  
2 float y = x * 30;
```

No contexto deste trabalho, a análise semântica é importante para validar expressões e funções relacionadas à renderização de materiais no desenvolvimento de *shaders* no OpenGL. Por exemplo, ao escrever uma função BRDF em GLSL, o analisador deve verificar se os tipos de dados e operações são compatíveis tanto com a definição da função BRDF quanto com as dimensões de vetores e outras grandezas definidas.

2.4.8 Geração da Linguagem Alvo

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

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

Como exemplo, considere a tradução de um cálculo matemático de L_1 (\LaTeX), para L_2 (GLSL). O cálculo apresentado na [Equação 2.10](#) pertence a L_1 . O [Código 3](#) mostra o código-fonte desse cálculo.

$$\mathbf{v} = (\mathbf{a} + \mathbf{b}) \cdot \mathbf{c} - (\mathbf{d} \times \mathbf{e}) \quad (2.10)$$

Código 3 – Cálculo vetorial em código-fonte \LaTeX .

```
1 \mathbf{v} = (\mathbf{a} + \mathbf{b}) \cdot \mathbf{c} - (\mathbf{d} \times \mathbf{e})
2
```

Após a tradução da expressão matemática para L_2 , o cálculo pode ser convertido para o trecho de programa apresentado no [Código 4](#). Esse código é válido na linguagem GLSL.

Código 4 – Cálculo vetorial em código GLSL.

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

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 \LaTeX para a linguagem de *shading*, empregando técnicas de *parsing*. O processo de busca será conduzido em duas etapas distintas. Inicialmente, será realizado um levantamento dos trabalhos existentes nas bases de dados com relevantes periódicos, 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. Também foram criadas questões de pesquisa para guiar a seleção dos trabalhos.

3.1.1 Seleção das Bases

As bases escolhidas foram: ACM Digital Library ¹, IEEE Xplorer Digital Library ², Biblioteca Digital Brasileira de Teses e Dissertações (BDTD) ³, Portal de Periódicos da CAPES ⁴, Google Acadêmico ⁵. Essas foram escolhidas por serem acessíveis gratuitamente pela afiliação

¹ <<https://dl.acm.org/>>

² <<https://ieeexplore.ieee.org/>>

³ <<https://bdtd.ibict.br/>>

⁴ <<https://www.periodicos-capes.gov.br.ezl.periodicos.capes.gov.br/index.php?>>

⁵ <<https://scholar.google.com/>>

à Universidade Federal de Sergipe, já o Google Scholar foi escolhido por agregar pesquisas em outras bases que possam ter trabalhos relevantes.

3.1.2 Questões de Pesquisa

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 este estudo.

1. Quais são as abordagens mais comuns utilizadas na criação de compiladores para tradução de BRDFs expressas em alguma linguagem de texto, como \LaTeX , para *shaders*?
2. Quais as técnicas de *parsing* têm sido aplicadas no desenvolvimento de compiladores para linguagens matemáticas?
3. O trabalho utiliza árvores ou gramáticas livres 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 eles podem ser integrados ao processo de desenvolvimento?

3.1.3 Termos de Busca

As frases foram construídas 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. Os termos-chave escolhidos foram: ("shader"AND "BRDF"AND ("compiler" OR "parser" OR "grammar")), conforme demonstrado na [Tabela 1](#).

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 Periódico	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 Acadêmico	("BRDF" AND ("COMPILER" OR "COMPILING") AND ("PARSER" OR "PARSING") AND ("SHADER" OR "SHADING"))	69

Tabela 1: Tabela de pesquisa.

3.1.4 Critérios

Para garantir a relevância dos resultados obtidos, seguimos os critérios de inclusão e exclusão estabelecidos, de forma a filtrar os resultados. Ao fim desse procedimento, apenas os resultados com maior compatibilidade com este trabalho foram analisados e descritos de maneira detalhada. O resultados se encontram na [Tabela 2](#).

3.1.4.1 Critérios de Inclusão

1. Foram incluídos artigos relacionados às palavras-chaves;
2. Foram incluídos artigos que de alguma forma cite a criação de um compilador ou um *parser*;
3. Foram incluídos artigos que sintetizam uma árvore como representação de BRDFs.

3.1.4.2 Critérios de Exclusão

1. Foram excluídos artigos que dispunham de *links* incorretos e ou quebrados;
2. Foram excluídos artigos no quais os projetos são muito similares;
3. Foram excluídos artigos que não respondem as questões de pesquisa na [subseção 3.1.2](#);
4. Foram excluídos artigos que não têm como entrada uma BRDF no formato de equação, ou seja, utilizam a representação diretamente como código;

5. Foram excluídos artigos que não consideram a geração de *shaders* como saída ou estrutura da BRDF em árvore;
6. Foram excluídos artigos que não citam BRDFs e compilador ou árvores em seu resumo;
7. Se, após a leitura completa, o artigo não concerne os interesses deste trabalho, esse foi excluído.

Bases	Filtrados
IEEE Xplore Digital Library	2
BDTD	0
CAPES Periódico	0
ACM Digital Library	1
Google Acadêmico	1

Tabela 2: Resultados das bases 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 técnicas de programação genética para explorar e descobrir novas BRDFs de maneira analítica (BRADY et al., 2014). O processo inicia utilizando uma BRDF existente, e iterativamente aplica mutações e recombinações de partes das expressões matemáticas que compõem essas BRDFs à medida que novas gerações surgem. Essas mutações são guiadas por uma função *fitness*, que seria o inverso de uma função de erro, elas são baseadas em um *dataset* de materiais já medidos. Por meio da avaliação de milhares de expressões, a *framework* identifica as viáveis.

Os 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 em automaticamente encontrar novos modelos analíticos de BRDF, em vez de compilar diretamente equações BRDF em linguagens de *shading*. Embora a representação das expressões das BRDFs possa potencialmente inspirar o nosso trabalho, o principal objetivo do artigo difere do nosso tema.

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 (HE;

[FATAHALIAN; FOLEY, 2018](#)). 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 enfatiza que cada extensão da Slang busca oferecer uma progressão incremental para a adoção a partir do código HLSL existente, eliminando a necessidade de uma migração completa. Algumas dessas extensões incluem: funções genéricas, estruturas genéricas e tipos que implementam interfaces específicas, semelhantes ao funcionamento das interfaces em Java, mas aplicadas a estruturas. Um exemplo de função genérica escrita em Slang é:

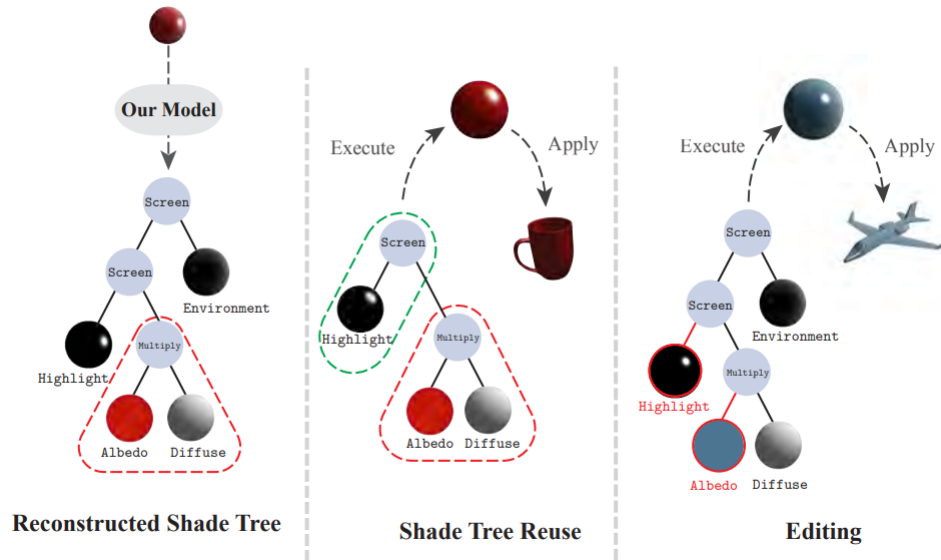
```
float3 integrateSingleRay<B:IBxDF>(B bxdf,  
SurfaceGeometry geom, float3 wi, float3 wo, float3 Li)  
{ return bxdf.eval(wo, wi) * Li * max(0, dot(wi, geom.n)); }
```

Enquanto o artigo tenta 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*. Embora ambos os projetos façam uso de *shaders* e compilação, as abordagens e focos são diferentes.

3.1.5.3 Tree-Structured Shading Decomposition

Esse trabalho propõe uma abordagem para inferir uma representação de BRDF estruturada em árvore a partir de uma única imagem para o sombreamento de objetos ([GENG et al., 2023](#)). Em vez de usar representações paramétricas, como é comum, é proposta uma abordagem que utiliza uma representação em árvore de *shading*, combinando nós básicos e métodos para decompor o *shading* da superfície do objeto, representado na [Figura 9](#).

Figura 9 – Exemplo de decomposição de BRDFs em nós de uma árvore.



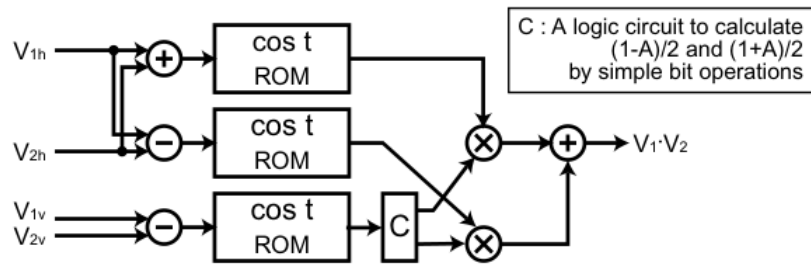
Fonte: Wolfe (1998).

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 *shading* sem exigir conhecimento avançado em programação. Esse artigo também emprega uma representação em árvore, embora para um propósito diferente.

3.1.5.4 A Real-Time Configurable Shader Based on Lookup Tables

Esse trabalho propõe uma arquitetura de *hardware* que permite cálculos de *shading* por pixel em tempo real, utilizando *lookup-tables* (OHBUCHI; UNNO, 2002). Para isso, são projetados circuitos configuráveis baseados nessas tabelas, memórias de acesso aleatório (RAMs) e memórias somente leitura (ROMs). Vários circuitos base foram projetados para as operações mais comuns. Por exemplo, circuitos para calcular o produto interno entre dois vetores e circuitos de rotação de um vetor por um ângulo, um exemplo desses diagramas é representado na Figura 10. Ademais, foi utilizada interpolação em um sistema de coordenadas polares em vez da interpolação vetorial convencional, com o objetivo de reduzir o tamanho dos circuitos e melhorar o desempenho.

Figura 10 – Exemplo de circuito de produto interno entre vetores.



Fonte: (OHBUCHI; UNNO, 2002).

Além disso, o circuito suporta diversas BRDFs, como Blinn-Phong, Cook-Torrance, Ward e modelos baseados em microfacetas, com tabelas 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. Esse trabalho foi aceito por incluir o processo de tradução estruturada de BRDFs para os circuitos. Assim como as árvores, eles são hierárquicos e são usados em composição para representar uma BRDF. Similar a este trabalho, a abordagem facilita a geração de *shaders* a partir da descrição de BRDFs, apesar da metodologia ser diferente.

3.2 Pesquisa por Repositórios Online

Também foram analisados repositórios no GitHub e SourceForge, cada um com uma *string* de busca específica. Os repositórios encontrados foram filtrados baseados em seus resumos. Caso não haja a menção da criação de um compilador ou não seja citada uma transformação de BRDF para outra estrutura, esse repositório foi excluído. O resultado se encontra na Tabela 3.

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: Resultados da pesquisa nos repositórios.

Após ler por completo os resumos dos repositórios do GitHub, é evidente que nenhum desses projetos é relacionado com o proposto neste trabalho. Apesar de comentarem 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 prática. As suas principais etapas são: uma análise das informações pertinentes a BRDFs e compilação de *shaders*; a exploração de técnicas existentes dentro do domínio; a especificação da linguagem subconjunto L^AT_EX de entrada; a implementação do compilador; a avaliação de seu desempenho por meio de experimentos de renderização.

Inicialmente, o método para realizar a análise e exploração das técnicas é descrito na [seção 4.1](#). Em seguida, a especificação da linguagem de entrada e saída é definida na [seção 4.2](#) @@@link the grammar and explain. Posteriormente, uma ideia de como o *design* dos casos de teste devem ser elaborados para validar a correção e precisão do compilador é apresentado na [seção 4.3](#). O método de implementação do compilador é detalhado na [seção 4.4](#). A [seção 4.5](#) planeja o método de avaliação dos experimentos de renderização quanto a qualidade visual dos *shaders* compilados. Por fim, um plano de continuação é delineado, abordando as próximas etapas para completar o desenvolvimento do compilador proposto (??). Seguindo essa metodologia, a ferramenta proposta visa compilar efetivamente descrições de BRDF em *shaders* GLSL.

4.1 Análise e Técnicas

O primeiro passo envolve a realização de uma análise detalhada das áreas relacionadas ao desenvolvimento da ferramenta proposta. Isso inclui a revisão da literatura ([Capítulo 3](#)) sobre BRDFs, linguagens de *shaders*, *design* de compiladores e técnicas de renderização gráfica. Além disso, envolve o estudo de ferramentas e bibliotecas pertinentes. Durante essa análise, foram estudados conceitos de radiometria para compreender tecnicamente as BRDFs. A principal fonte de informação sobre radiância e BRDFs foi o livro “Physically Based Rendering: From Theory To Implementation” ([PHARR; JAKOB; HUMPHREYS, 2016](#)). Esse livro foi importante para compreensão da equação de renderização ([Equação 2.7](#)). A leitura de exemplos práticos e leitura

das código fonte da ferramenta [Figura 11](#) permitiu a familiarização com o desenvolvimento de BRDFs, fornecendo uma base sólida para a compreensão do mapeamento da equação para código, aspecto fundamental para o desenvolvimento do compilador proposto neste trabalho. @

Ademais, foram exploradas diversas técnicas para compilação, como o método de Pratt *Parsing* para a construção de um compilador, conforme detalhado na [seção 6.1](#), somado ao uso do conhecimento de recursividade e caminhada em árvores para realizar a análise semântica e geração de código.

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 , na qual as expressões matemáticas nos ambientes `equation` são suficientes para descrever BRDFs. O \LaTeX é um sistema de composição amplamente utilizado para documentos matemáticos e científicos. O ambiente `equation` é especificamente projetado para exibir equações individuais. O [Código 5](#) é um exemplo de código-fonte \LaTeX usando o ambiente `equation`.

Código 5 – Código-fonte de função quadrática.

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

Este código representa a equação quadrática $g(x) = ax^2 + bx + c$, onde a , b e c são coeficientes. O código GLSL correspondente gerado a partir dessa equação pode ser o [Código 6](#).

Código 6 – Código GLSL da função quadrática g .

```
1 float g(float x, float a, float b, float c) {
2   return a * x * x + b * x + c;
3 }
```

@@@ O ambiente de equações \LaTeX é amplo demais para o projeto, entre todas as construções matemáticas representáveis por esse ambiente um subconjunto essencial para BRDFs deve ser escolhido. Ao analisar as principais BRDFs, como os ditos em [seção 4.3](#), nota-se algumas construções indispensáveis, essas devem ser reconhecidas e entendidas o suficiente para emitir código GLSL pelo nosso compilador, essas são enumeradas à seguir:

1. principais funções trigonométricas `\tan`, `\sin`, `\cos`, `\arctan`, `\arcsin`, `\arccos`;
2. função raiz quadrada `\sqrt{}`;

3. função exponencial \sqrt{x} (`\sqrt{x}`);
4. funções utilitárias como \max , \min , (`\max`, `\min`);
5. definição de equações, por exemplo $f = x$ (rederizado fica $f = x$).
6. denifição de funções, por exemplo $f(x, y) = x^y$ (rederizado fica $f(x, y) = x^y$) respectivamente;
7. constantes comuns como π (`\pi`), ϵ (`\epsilon`);
8. constantes especificar θ (`\theta`), entre outros detalhados na @ref capitulo@;
9. indicador de vetor como \vec{n} (`\vec{n}`);
10. identificadores aninhados como f_{n_i} (`f_{n_i}`);
11. chamada de funções $f(x+y)$;
12. operadores de produto vetorial ($x \times y$, `x \times y`), soma (+), multiplicação ($x * y$ ou $x \cdot y$, `x \cdot y`), fração ($\frac{x}{y}$, `\frac{x}{y}`), divisão ($\frac{x}{y}$, `x/y`), power x^y , (`x^y`);

Uma descrição completa dos simbolos reconhecidos são dados no @Desenvolvimento capitulo Lexer@. Construção completa da gramatica reconhecida pelo compilador é dado em @Capitulo Desenvolvimento Parser@. Note que do potno de vista do parser e lexer alguns simbolos são apenas reconhecidos, é citado que o compilador também precisa entenderlo, e para é preciso atribuir significado especifico à esses simbolos e construções, por exemplo ω_o , que o angulo de saída da luz @@@ ou f que é a brdf. Essa atribuição é feita em etapa de analise semantica, que vem após o parsing @ref@.

4.3 Design de Casos de Teste

Os casos de teste são essenciais para validar a precisão e correção do processo de tradução do compilador. Eles estabelecem uma correspondência entre as equações \LaTeX de entrada, que descrevem as BRDFs, e o código de *shader* GLSL esperado como saída. Um exemplo específico que demonstra a eficácia do compilador pode ser construído com a BRDF de Cook-Torrance. Sua função, `cook_torrance`, é representada pela [Equação 4.1](#) (seu código-fonte está definido no [Código 7](#)), onde D é a função de distribuição normal, G é a função de sombreamento geométrico e F é a função de Fresnel.

Embora as funções D , G , F não tenham sido definidas explicitamente, é importante ressaltar que, caso essas funções fossem definidas na equação \LaTeX , elas também devem ser definidas no [Código 8](#), GLSL esperado de saída. Vale resaltar que nessa sessão de metodologia estamos dando uma versão simplificada de como o design de casos de teste ocorre para auxiliar

entendimento. Na prática, unidades, como ρ_d , e funções, como D , G e F , devem estar definidas. Casos de teste completos estão disponíveis no ??.

Além disso, algumas variáveis, como a normal representada por n , seriam passadas como entrada no *shader* de fragmentos ou declaradas como variáveis uniformes, portanto não estão definidas explicitamente na função `cook_torrance` no [Código 8](#); elas são variáveis implícitas. Todas as variáveis implícitas se encontram na ??. Inicialmente, o foco é definir casos de teste para avaliar apenas a geração das operações e precedências. No entanto, é importante considerar que, posteriormente, o GLSL não deverá apenas gerar a função BRDF, mas sim o *shader* completo, incluindo as variáveis uniformes e a passagem da cor calculada para as próximas etapas do *pipeline* gráfico.

$$\text{cook_torrance}(\omega_i, \omega_o) = \frac{D(h)F(\omega_i, h)G(\omega_i, \omega_o, h)}{4(\omega_i \cdot n)(\omega_o \cdot n)} \quad (4.1)$$

Código 7 – Entrada em \LaTeX (Cook-Torrance BRDF).

```
1 \text{cook\_torrance}(\omega_i, \omega_o)
2 = \frac{D(h)F(\omega_i, h)G(\omega_i, \omega_o, h)}{4(\omega_i \cdot n)(\omega_o \cdot n)}
```

Código 8 – Saída em GLSL esperada (Cook-Torrance BRDF).

```
1 vec3 cook_torrance(vec3 wi, vec3 wo) {
2     float D_RESULT = D(h);
3     vec3 F_RESULT = F(wi, wo);
4     float G_RESULT = G(wi, wo, h);
5     float denominador = 4.0 * dot(n, wi) * dot(n, wo);
6     return D_RESULT * F_RESULT * G_RESULT / denominador;
7 }
```

4.4 Implementação do Compilador

Este trabalho envolve várias tarefas-chave destinadas a completar o desenvolvimento do compilador proposto para converter equações \LaTeX que descrevem BRDFs em código de *shader* GLSL. As tarefas incluem: Criar um *lexer* e *parser* para aceitar equações \LaTeX ; testar o *lexer* para garantir o reconhecimento correto dos *tokens*; testar o *parser* para garantir que a árvore sintática está com precedência correta; definir símbolos predefinidos e constantes matemáticas; implementar o processo de geração de código GLSL usando a árvore sintática com o padrão de *design* visitante (*Visitor*); definir os casos de teste para cobrir uma certa variedade de BRDFs; testar o código gerado quanto à correção, incluindo as visualizações das BRDFs em algumas cenas.

A implementação do compilador é realizada utilizando a linguagem de programação Odin, conhecida por ser uma linguagem de propósito geral com foco em programação orientada a dados. Sua escolha se deve à sua capacidade de oferecer controle de baixo nível e a sua adequação para o desenvolvimento de sistemas complexos. Além disso, nenhuma biblioteca externa foi utilizada, sendo usada apenas as bibliotecas padrões básicas que acompanham a instalação da linguagem.

Técnicas de análise recursiva são utilizadas, especificamente o Pratt *Parsing*. Inicialmente, o *lexer* e o *parser* foram implementados para o subconjunto (4.2) linguagem \LaTeX comentado em Código 9. Para garantir que os fundamentos do compilador estejam funcionais, considerando precedência totalmente testada para a árvore sintática, foram criados o pacote *walker*, que abstrai uma maneira de andar pela AST e valida algo, esse é usado para duas coisas, uma é para adicionar parênteses expoiciando a ordem de operação, outro é recursivamente inferir os tipos de cada expressão (nós que representam valores) presentes na AST. Também, é necessário criar um passagem de análise semântica, pacote chamado de "checker" onde iremos anotar a AST com todos os campos relevantes como tipos (função com seu domínio e contradomínio, vetor real e sua dimensão, ou número $\in \mathbb{R}$). Por último, já com o AST anotadas com outras informações, realizamos através do pacote "emitter" a geração de código glsl, pronto para ser carregado e renderizado pela ferramenta seção 4.5.

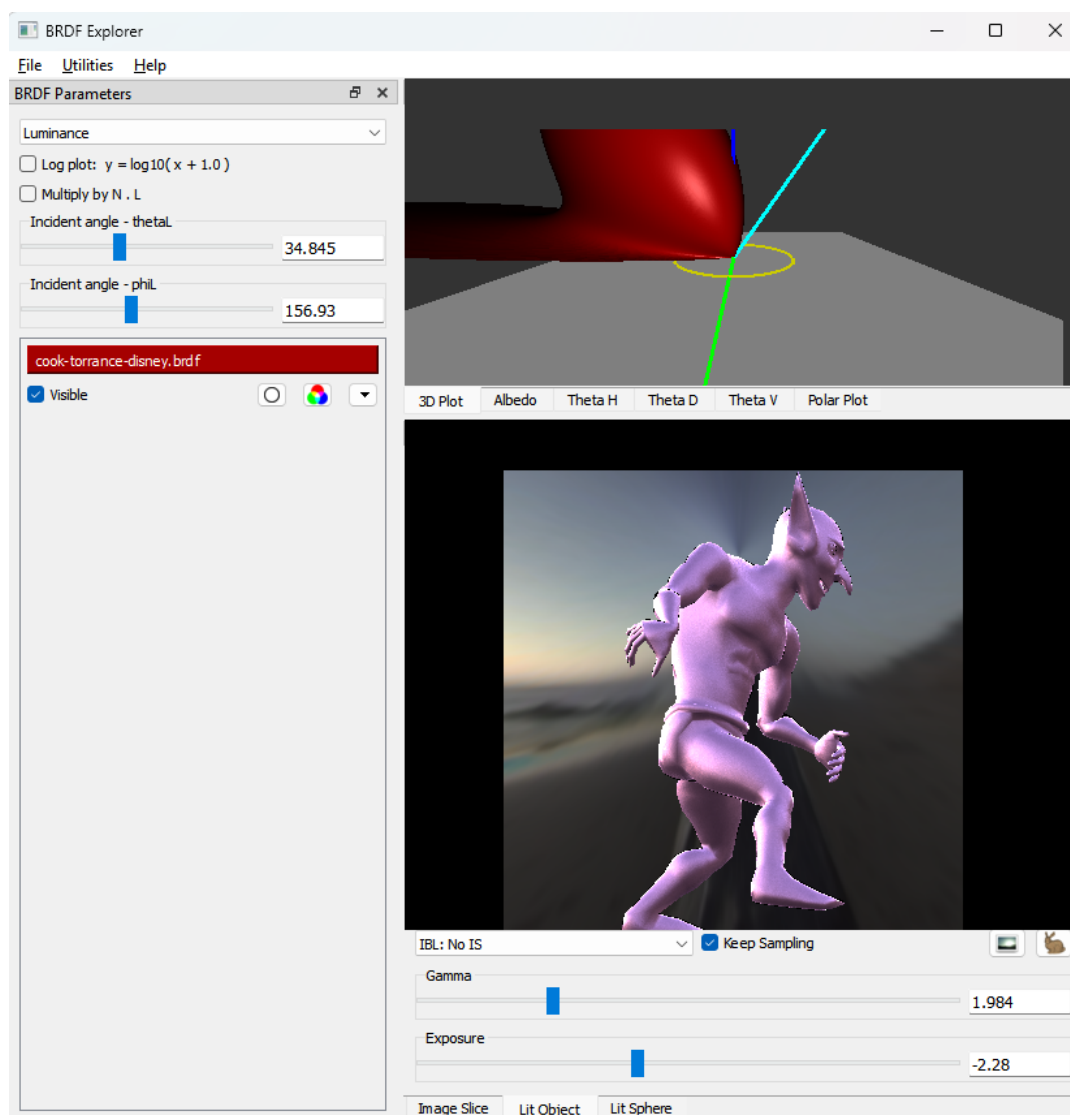
4.5 Experimentos de Renderização

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 BRDF¹, compilada localmente para modificar e adicionar outros *shaders*.

Essa ferramenta é composta por um renderizador e uma interface que permite ajustar parâmetros de BRDFs através de controles deslizantes em tempo real, fornecendo uma visualização interativa do efeito das mudanças nos parâmetros que afetam a aparência do objeto renderizado, como ilustrado na Figura 11. O código que informa à ferramenta qual a BRDF a ser renderizada e seus possíveis parâmetros pode ser visto na Figura 12. Esse código possui um formato específico, onde se encontram algumas seções. Existe a seção para código GLSL e outra seção delimitada por `::begin parameters` e `::end parameters`, na qual podemos definir os parâmetros que se tornam constantes dessa BRDF. O nosso compilador gera *shaders* nesse formato.

¹ <<https://github.com/wdas/brdf>>

Figura 11 – Ferramenta de visualização de BRDFs da Disney.



Fonte: autor.

Figura 12 – O código GLSL com sintaxe extra para definir parâmetros.

```
1 analytic
2 # Blinn Phong based on halfway-vector
3
4 # variables go here...
5 # only floats supported right now.
6 # [type] [name] [min val] [max val] [default val]
7
8 ::begin parameters
9 float n 1 1000 100
10 bool divide_by_NdotL 1
11 ::end parameters
12
13
14 # Then comes the shader. This should be GLSL code
15 # that defines a function called BRDF (although you can
16 # add whatever else you want too, like sqr() below).
17
18 ::begin shader
19
20 vec3 BRDF( vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y )
21 {
22     vec3 H = normalize(L+V);
23
24     float val = pow(max(0,dot(N,H)),n);
25     if (divide_by_NdotL)
26         val = val / dot(N,L);
27     return vec3(val);
28 }
29
30 ::end shader
```

5

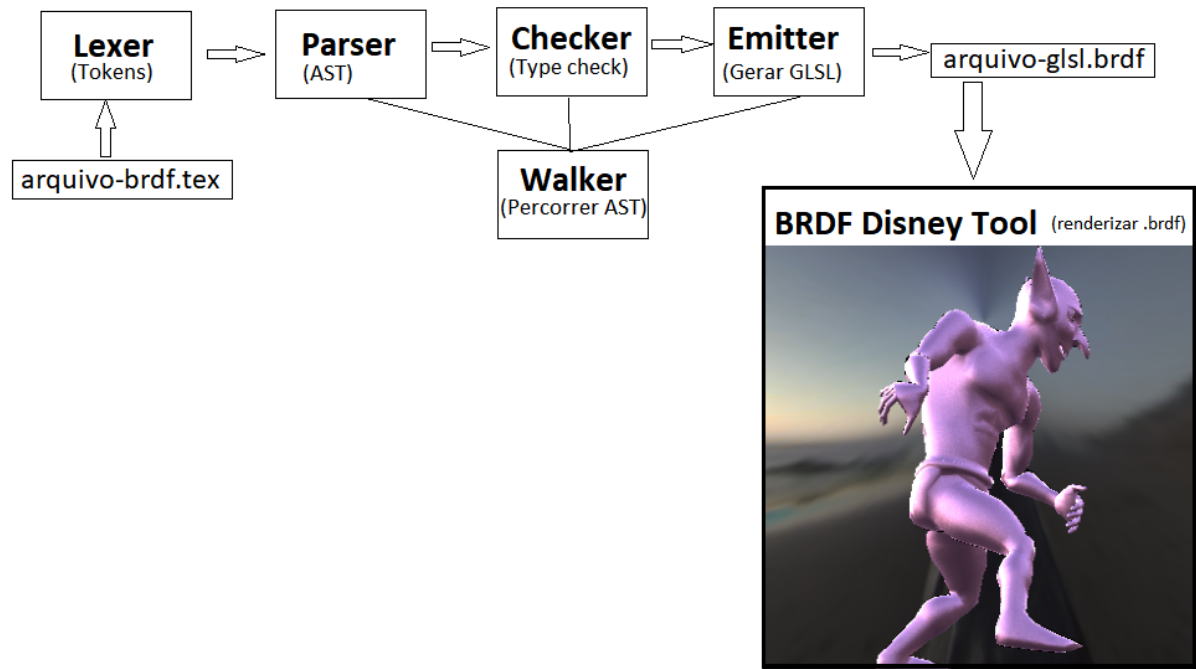
Desenvolvimento

Este capítulo aborda o processo de desenvolvimento do compilador proposto como um todo na linguagem Odin. Cada etapa é encapsulado em um pacote, representado em [Figura 13](#) diferente `lexer` corresponde à tokenização da linguagem, `parser` corresponde à análise sintática, `walker` contém funções que auxiliam tanto a visualizar o resultado da análise sintática, a AST, quando na checagem de tipos da análise sintática, pois ambas dependem de fazer a transver@@@ da árvore em ordem, . A arquitetura da pipeline para o compilador é delineado na [Figura 14](#). O repositório pode ser encontrado em [<https://github.com/evertorse/@@@>](https://github.com/evertorse/@@@>)

Figura 13 – Estrutura de Pacotes do Compilador.

```
src/
├── ast/
│   ├── ast.odin
│   └── type.odin
├── checker/
│   ├── check.odin
│   ├── scope.odin
│   └── symbol.odin
├── emitter/
│   └── emit.odin
├── lexer/
│   ├── lexer.odin
│   └── token.odin
├── parser/
│   └── parser.odin
├── walker/
│   ├── svg_walker.odin
│   └── walker.odin
├── main.odin
└── test.odin
```

Figura 14 – Estrutura de geral da arquitetura da pipeline do Compilador.



Os resultados do desenvolvimento desse compilador pode ser encontrado em ???. A especificação da linguagem pode ser encontrada no ???. Nesse apêndice temos a gramática @@@ para tokens e gramática que gera AST, a tabela de precedência que é necessário para desambiguar a linguagem encontra-se em ??. Os exemplos de BRDFs mostrados no ?? foram usados como base para verificação da correção da gramática durante seu desenvolvimento.

Nesta construção do compilador, foi feita análise léxica manualmente através de loops mudando o estado atual para separar a entrada, que seria um string do arquivo inteiro, para uma lista de tokens. Já a análise sintática usamos a gramática livre de contexto ?? para nos guiar, somado a tabela de precedência para aplicarmos o Pratt Parsing que resulta em uma AST.

@Add development preview of wahts to come

5.0.1 Desenvolvimento

Primeiro foi criado o analisador léxico, um pacote inteiro para esse analisador na linguagem `odin`. O trabalho desse analisador é transformar um array de caracteres que é a entrada e retornar uma sequência de tokens. Cada token tem um tipo (chamado de `kind` em código), um valor, reservado para números, texto, e posição, que é usado para reportar erros.

Cada tipo (*kind*) é dado pela enumeração `Token_Kind`, essa encode todos os possíveis tipos como dito @cite previous chapter talking about the entry language. Esses tokens podem

ser: comentários gerados por uma linha que comece com %, números, identificadores que são qualquer sequência de caracteres que não seja palavras especiais, símbolo de igual ('='), símbolos de operadores ('^', '*') .. bla, funções especiais (max, sin, arccos, etc ...)

Código 9 –

```
1 Token :: struct {
2     kind: Token\_Kind,
3     val: union{ i64, f64 },
4     text: string,
5     pos: Position,
6 }
```

O processo de lexing feito com um loop, simulado a uma maquina de estados, que decide qual token deve ser criado em sequencia ao olhar o caractere atual e o estado.

Estados estão relacionados ao processo de identificar estados pode estar relacionados a identificar palavras.

É adiante, por exemplo se encontrar um '1' sabemos que é um número, podendo ter um '.' para indicar decimal, então utilizamos uma subrotina para identificar esse continuar processando o "input" até o token de números ter sido totalmente coletado, se no meio de processar um número um caractere não esperado for encontrado, reportamos um erro léxico, exemplos pode ser visto na imagem @Mostre Imagem com Erro O mais simples são tokens de um caractere '^', '*', '/', '+', '-', '?', '=', ' ', '(', ')', ',', ':', ';', '"', '\'', '_'. Cada um tem um propósito específico na análise lexical. Na etapa lexical nos preparamos apenas em separar os tokens de maneira cega ao seu significado.

Todo identificador, especial ou não é processado da mesma maneira, é verificado se o caractere atual é um letra ou um ' ,

, isso indica o começo de um identificador. Depois de de

A gramática dos tokens é regular e será representada abaixo:

Vale ressaltar que nesse moment é criado uma tabela que mapeia cada numero de linha à um string dessa mesma linha, para reportar error, printando a linha do problema mais a linha anterior e posterior para. Tem um token que é especial que indica o começo de um ambiente ‘`beginequation`’, qualquer comentario antes de apaecer esse token é ignorado, isso é para poder dar como entrada ao compilador um documento inteir ocontendo `begin document` e ainda funciojnar

5.0.2 Analise Semantica

5.0.2.1 Tabela de Symbolos

Symbolos podem ser declarados fora de ordem, criamos um grafo de dependencias e fazemos um ordenação topologica de dependencia. Isso é pois, ao detectar analisa um certo simbolo queremos dizer se está usando simbolos não definidos, para isso precisamos definir todos os simbolos locais que estão no escopo visível a todos, isso inclui simbolos pre-definidos pela linguagem, (ver tabela @tabela de simbolos predefinidos, para isso precisamos primeiro coletar todos esses e analisar primeiros oq que dependem de ninguem, e medida que vão . Também pode ocorrer dependencia circular sem resolução e nesse caso reportamos um erro, nesse caso precisamos. @true? circular dependency?

5.0.2.2 Inferencia de Tipos

5.0.3 SVG da arvore abstrata com inferencia de tipos

Para identificar possíveis erros de ordenação algumas medidas foram feitas para auxiliar, como a geração de uma imagem da em SVG da arvore sintatica, já com inferencia de tipos

5.1 Análise Léxica

Esta etapa apresenta o desenvolvimento de tokenização do subconjunto do ambiente de equação do \LaTeX . A entrada para essa etapa são os caracteres do arquivo fonte, e a saída é uma organização lógica desses caracteres em sequencia que formam os `tokens`. O código dessa etapa se encontra no pacote `lexer` apresentado em [Figura 13](#).

Primeiro, realizamos um laço sobre o arquivo inteiro, passado caracter à caracter para extrair os tokens. Antes realizamos uma checagem de igualdade com a string `\begin{equation}` para decidir se já podemos começar a extrair os tokens. Dessa maneira permitimos que outros textos que não estão dentro da delimitação, a qual acaba com `\end{equation}`, possa existir, como textos explicatorios dentro de um mesmo arquivo de extensão `.tex`.

Por conveniencia, apresentamos uma gramatica para geração dos `tokens`, escrito apenas para fins de documentação, [Código 14](#), o alfabeto dessa gramatica são os caracteres. A geração de tokens internamente possui sua implementação similiar a simulação de uma máquina de estados.

Na definição da gramática (`??`), utilizamos uma notação leve de sintaxe para representá-la. Palavras com todas as letras minúsculas são não-terminais, enquanto palavras entre aspas simples representam literalmente *caracteres* com esse conteúdo. Palavras em letras maiúsculas representam um *caractere* que pode variar, mas mantém o mesmo significado semântico. Por exemplo, `DIGIT` pode ser um dígito de 0 à 9, mas nas regras de produção eles são tratados de maneira idêntica. `LETTER` é outro exemplo, que significa, uma letra a à z. O símbolo “`*`” indica zero ou mais ocorrências, “`()`” indica agrupamento para aplicar um operador a ele, “`|`” simboliza o início de uma regra alternativa para o mesmo não-terminal, ou se estiver dentro de um agrupamento dessa maneira “`(a|b)`” significa que aceita a ou b e “`=`” indica uma produção. Essa mesma definição de gramatica é utilizada para `??`, com a diferença que o alfabeto dela são formato pelo conjunto de tokens gerados nessa etapa.

O pacote inteiro pode ser chamada através de uma unica função, vista no [Código 10](#) em sintaxe `Odin`. Significa que temos um procedimento, chamado `lex` que aceita uma a lista de caracteres, e retorna uma lista do tipo `Token` ([Código 12](#)), esse tipo é uma estrutura que possui os campos: `kind`, que discrimina o tipo de token, que corresponde a uma das regras de produção na [Código 14](#); `text`, corresponde à string que o gerou; `position` instancia do tipo `Position`.

Código 10 – Função principal do Lexer.

```
1
2 lex :: proc(input: []u8) -> []Token
```

A medida que iteramos nesse `input`, também matemos algumas variaveis de controle, como contagem que quebra de linhas (“`\n`” ou “`\n\r`”), coluna atual e o cursor que repretenta `index` que aponta para o caractere sendo processado. A contagem de quebra de linha e coluna

é importante para preencher a estrutura o campo do token correspondente ao tipo `Position`, representado em [Código 12](#), que por sua vez é essencial para reportagem de errors. O reporte de erros que é implementada nessa etapa e utilizada por todos os pacotes do projeto, sua função possui possíveis assinatura vista no [Código 11](#): `function-errors`

Código 11 – Função de erro exposto pelo pacote `lexer`.

```
1 error_from_pos :: proc(pos: Position, msg: string, args: ..any)
2 error_from_token :: proc(token: Token, msg: string, args: ..any);
```

Dado um posição ou um `token` exibimos uma mensagem (`msg`) que é mostrado na tela do terminal com uma formatação que mostra exatamente onde está o erro, em vermelho. Extraindo as informações do token sabemos exatamente como sublinhar o erro, pois sabemos qual o nome do arquivo, linha, coluna, e comprimento do token que gerou o problema, possibilitando uma clara mensagem de erros exemplificado no erro sematnico de uso de indentificador não definido `??`, erro de `@@` outros erros `@@`

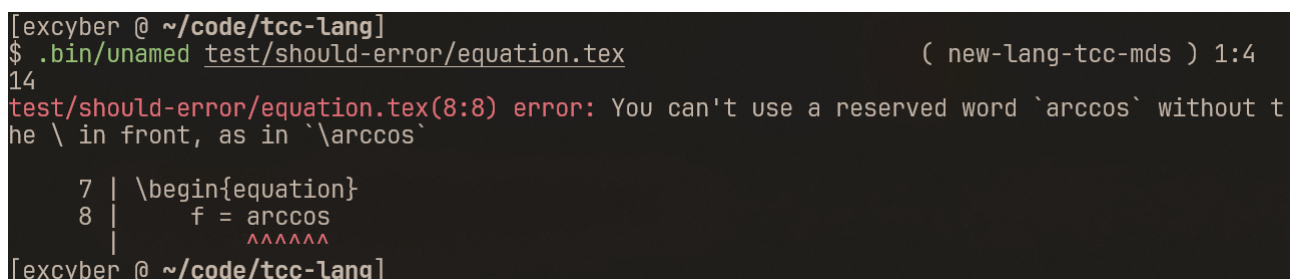
Figura 15 – Erro de balanceamento de parentesis.



```
[excyber @ ~/code/tcc-lang]
$ .bin/unamed test/should-error/balanceamento.tex ( new-lang-tcc-mds ) 2:17
`\\end`
`)`test/should-error/balanceamento.tex(9:1) error: Expected `)` but got `\\end` instead.

  8 |      f = ((1 * 2)
  9 |      \\end{equation}
    |      ^^^
 10 |
[excyber @ ~/code/tcc-lang]
```

Figura 16 – Erro de `@@@`.



```
[excyber @ ~/code/tcc-lang]
$ .bin/unamed test/should-error/equation.tex ( new-lang-tcc-mds ) 1:4
14
test/should-error/equation.tex(8:8) error: You can't use a reserved word `arccos` without t
he \\ in front, as in `\\arccos`

  7 | \\begin{equation}
  8 |      f = arccos
    |      ^^^^^
[excyber @ ~/code/tcc-lang]
```

Figura 17 – Erro de @@@.

```
[excyber @ ~/code/tcc-lang]
$ .bin/unamed test/should-error/math.tex ( new-lang-tcc-mds ) 1:52
parsed test/should-error/math.tex sucessfully
test/should-error/math.tex(8:9) error: Function `exp()` can only have a number type as argu
ment, but we got `R^3`.
    7 | \begin{equation}
    8 |     f = \exp(\vec{1,1,1})
        |           ^^^^
    9 | \end{equation}
[excyber @ ~/code/tcc-lang]
```

Figura 18 – Erro de @@@.

```
$ .bin/unamed test/should-error/token.tex ( new-lang-tcc-mds ) 1:54
test/should-error/token.tex(9:16) error: Expected an expression, but got token that can't m
ake a expresion got `\end`
    8 | \begin{equation}
    9 |     f = ----- \end
        |                   ^^^^
   10 | \end{equation}
```

Figura 19 – Erro de @@@.

```
$ .bin/unamed test/should-error/another.tex ( new-lang-tcc-mds ) 1:52
parsed test/should-error/another.tex sucessfully
test/should-error/another.tex(8:4) error: Trying to use undefined symbol `\text{var}` insi
de equation `f`
    7 | \begin{equation}
    8 |     f = 2^\exp(\text{var})
        |           ^
    9 | \end{equation}
[excyber @ ~/code/tcc-lang]
```

Tokens de um a dois caracteres são simples, basta ler um ou dois caracteres do input e contruir o token e continuar o laço até não poder mais. Se for ecnontrado o caractere %, então o restante dos caracteres são ignorates até encontrar uma quebra de linha, isso é feito para dar suporte à comentarios \LaTeX . Se for encontrado um dígito ou letra então são classificados como indentificadores, números ou tokens especiais.

Numbers podem opcionalmente ter (ex: 1.0). Indentificadores são formados por uma ou mais letras com o simbolo \backslash opcionalmente prefixo ao mesmo. Note que a grammatica de tokens é ambigua, uma sequencia de caracteres como $\backslash\text{frac}$ pode ser interpretada como indetificar, para

desambiguar, criamos um dicionário que mapeia um string à um token considerado especial. Assim se o identificador começar com o caractere `\`, mapeamos ele para um token especial através do dicionário exposto em [Código 13](#).

Com laços representamos a extração de.

Note que identificadores não permite numeros nem mesmo @underline “_”, pois no analisador sintático, um nó do tipo identificador é modelado como tipo recursivo, os identificadores podem ser aninhados, ao conter outro nó. Sendo assim, não é necessário que ao nível de token seja permitido o underline em identificador, isso permite identificadores mais complexos a serem escritos como `\pi_{n_1}` (renderizado em π_{n_1}). `\pi` seria o primeiro token do nó identificador e sua subexpressão seria `n_1` (`n_1`) que por sua vez é o identificador `n` com subexpressão `1`.

Código 12 – Estruturas do Lexer.

```

1  Token :: struct {
2      kind: Token_Kind,
3      val: union{i64,f64},
4      text: string,
5      pos: Position,
6  }
7
8
9  /*
10 . Line and colum in the source string,
11 . we only store the end line and col position for simplicity
12 */
13 Position :: struct {
14     file: string,
15     offset: i64,    // starting at 0, buffer offset in file
16     line: i64,     // starting at 1, starting
17     column: i64,   // starting at 1
18     length: int    // how much chars foward
19 }
```

Um enumeração que representa o tipo de token é mostrada no [Código 15](#), cada entrada nessa enumeração é correspondente as regras de produção na gramatica apresentada em [Código 14](#). Ao lado direito de cada entrada aprendemos o simbolo que o representa em comentarios.

Código 13 – Mapa de indentificadores especiais.

```

1
2 SPECIAL_WORDS := map[string]Token{
3     "text" = Token{text = "\\text",    kind = .Text},
4
5     // Special
6     "frac" = Token{text = "\\frac",    kind = .Frac},
7     "vec"  = Token{text = "\\vec",     kind = .Vec},
8     "cdot" = Token{text = "\\cdot",    kind = .Mul},
9     "begin" = Token{text = "\\begin",  kind = .Begin},
10    "end"   = Token{text = "\\end",     kind = .End},
11    "rho"   = Token{text = "\\rho",     kind = .Rho},
12    "sqrt"  = Token{text = "\\sqrt",    kind = .Sqrt},
13    "omega" = Token{text = "\\omega",   kind = .Omega},
14
15    // Cross product
16    "times" = Token{text = "\\times",   kind = .Cross},
17
18    "max"    = Token{text = "\\max",     kind = .Max},
19    "min"    = Token{text = "\\min",     kind = .Min},
20    "exp"    = Token{text = "\\exp",     kind = .Exp},
21
22    "cos"    = Token{text = "\\cos",     kind = .Cos},
23    "sin"    = Token{text = "\\sin",     kind = .Sin},
24    "tan"    = Token{text = "\\tan",     kind = .Tan},
25
26    "arccos" = Token{text = "\\arccos",  kind = .ArcCos},
27    "arcsin" = Token{text = "\\arcsin",  kind = .ArcSin},
28    "arctan" = Token{text = "\\arctan",  kind = .ArcTan},
29
30    "theta"  = Token{text = "\\theta",   kind = .Theta},
31    "phi"    = Token{text = "\\phi",     kind = .Phi},
32
33    "alpha"  = Token{text = "\\alpha",    kind = .Alpha},
34    "beta"   = Token{text = "\\beta",    kind = .Beta},
35    "sigma"  = Token{text = "\\sigma",    kind = .Sigma},
36    "pi"     = Token{text = "\\pi",      kind = .Pi},
37    "epsilon" = Token{text = "\\epsilon", kind = .Epsilon},
38
39 }

```

Código 14 – Gramatica ilustrativa para tokens.

```

token_number      = DIGIT DIGIT* '.' DIGIT DIGIT* | DIGIT DIGIT*;
token_identifier  = '\' LETTER LETTER* | LETTER LETTER*;
token_cmpgreater  = '>';
token_cmpless     = '<';
token_cmpequal    = '==';
token_equal       = '=';
token_mul         = '*' | '\cdot';
token_cross       = '\times';
token_div         = '/';
token_plus        = '+';
token_minus       = '-';
token_caret       = '^';
token_semicolon   = ';';
token_comma       = ',';
token_colon       = ':';
token_question    = '?';
token_bang        = '!';
token_openparen   = '(';
token_closeparen  = ')';
token_opencurly   = '{';
token_closecurly  = '}';
token_tilde       = '~';
token_underline   = '_'; --- @@@ used for subexpresions
token_arrow       = '->';
token_begin       = '\begin';
token_end         = '\end';
token_frac        = '\frac';
token_vec         = '\vec';
token_omega       = '\omega';
token_theta       = '\theta';
token_phi         = '\phi';
token_rho         = '\rho';
token_alpha       = '\alpha';
token_beta        = '\beta';
token_sigma       = '\sigma';
token_pi          = '\pi';
token_epsilon     = '\epsilon';
token_max         = '\max';
token_min         = '\min';
token_exp         = '\exp';
token_tan         = '\tan';
token_sin         = '\sin';
token_cos         = '\cos';
token_arctan      = '\arctan';
token_arcsin      = '\arcsin';
token_arccos      = '\arccos';
token_sqrt        = '\sqrt';
token_text        = '\text';
token_eof         = EOF;

```

Código 15 – Enumeração dos tipos de tokens.

```

1 Token_Kind :: enum {
2   EOF           = 0,
3   Number,
4   Identifier,
5
6   Equal,        // =
7   Mul,          // * ou \cdot
8   Cross,        // X
9   Div,          // /
10  Plus,         // +
11  Minus,        // -
12  Caret,        // ^
13  Comma,        // ,
14  Colon,        // :
15  Question,     // ?
16  Bang,         // !
17  OpenParen,    // (
18  CloseParen,   // )
19  OpenCurly,   // {
20  CloseCurly,  // }
21  Tilde,        // ~
22  Underline,    // _
23  Arrow,        // ->
24
25  Begin = 256,   // \begin
26  End,          // \end
27
28  Frac,         // \frac
29  Vec,          // \vec
30
31  Omega,        // \omega
32  Theta,        // \theta
33  Phi,          // \phi
34  Rho,          // \rho
35  Pi,           // \pi
36  Epsilon,      // \epsilon
37  Alpha,        // \alpha
38  Beta,         // \beta
39  Sigma,        // \sigma
40
41  Max,          // \max
42  Min,          // \min
43  Exp,          // \exp
44  Tan,          // \tan
45  ArcTan,       // \arctan
46  Sin,          // \sin
47  ArcSin,       // \arcsin
48  Cos,          // \cos
49  ArcCos,       // \arccos
50  Sqrt,         // \sqrt
51
52  Text,         // \text
53  Invalid

```

5.2 Análise Sintática

Desenvolvemos o *parser* para a linguagem subconjunto do ambiente `equation` do \LaTeX utilizando o Pratt *Parsing* na linguagem de programação Odin. Nesta seção, vamos chamar esse subconjunto de `EquationLang`, o qual inclui todas as partes essenciais para BRDFs citada em [seção 4.2](#), e também documentamos a sua gramática.

Esse *parser* é implementado por descida recursiva, o que significa que cada regra de produção tem uma função de análise associada. A implementação prioriza a simplicidade de código e a clareza de ideias, com extensos comentários para auxiliar na compreensão. Essa etapa aceita os *tokens* da etapa anterior.

5.2.1 Parser

Diferente dos *parser* de descida recursiva tradicionais, que muitas vezes exigem várias chamadas de função aninhadas para cada nível de precedência, o nosso *parser* organiza as funções de análise hierarquicamente com base na precedência do operador, como demonstrado no [Código 27](#). Esse código é a parte principal do *parsing* de expressões. Nessa implementação usamos a notação original de Pratt ([PRATT, 1973](#)), as funções `parse_null_denotations` e `parse_left_denotations` são equivalentes as funções `token.prefixo` e `token.infixo` declaradas no [Algoritmo 1](#), respectivamente. Além disso, pela característica de descida recursiva (top-down), cada regra de produção especificada em [Código 20](#) é mapeada similiarmente para um procedimento em código. Podemos notar a semelhança entre a definição da função de parsing do nó `Start` da AST, no [Código 18](#), e as regras de produção `start`, `decl`, `decl_equation_begin_end_block` presente na gramática do [Código 20](#).

Do ponto de vista da interface que o pacote `parser` oferece, o trabalho inteiro de análise sintática, pode ser resumido a uma chamada de função e uma estrutura de controle ([Código 17](#)): `parse` e `Parser`, respectivamente.

Código 16 – Parsing de expressão em código Odin.

```

1
2
3 parse_expr :: proc(prec_prev: i64) -> ^Expr {
4     /* expressions that takes nothing (null) as left operand */
5     left := parse_null_denotations()
6     /*
7     . if current token is left associative or current token has
        higher precedence
8     . than previous precedence then stay in the loop, effectively
        creating a left leaning
9     . sub-tree, else, we recurse to create a right leaning sub-tree.
10    */
11    for precedence(peek()) > prec_prev + associativity(peek()) {
12        /* expressions that needs a left operand such as postfix,
            mixfix, and infix operator */
13        left = parse_left_denotations(left)
14    }
15    return left
16 }

```

Código 17 – Estruturas e Funções do Parser.

```

1 Parser :: struct {
2     tokens:      []Token,
3     cursor:      i64,
4     error_count: int,
5 }
6
7 parse :: proc(using p: ^Parser) -> ^ast.Start {
8     return parse_start(p)
9 }

```

5.2.2 Gramática

Para formalizar a gramática da linguagem de entrada (EquationLang) deste compilador, definimos suas regras no [Código 20](#) e [Código 21](#). Um exemplo de código-fonte válido em EquationLang é apresentado no [Código 28](#), sua renderização em L^AT_EX é dado em [subseção 5.2.2](#).

$$\rho_d = 0.3, 0.\vec{3}, 0.3 \quad (5.1a)$$

$$\rho_s = 0.0, 0.\vec{2}, 1.0 * 20 \quad (5.1b)$$

$$f = \frac{\rho_d}{\pi} + \frac{\rho_s}{8 * \pi} * \frac{(\vec{n} \cdot \vec{h})}{(\vec{\omega}_o \cdot \vec{h}) * \max((\vec{n} \cdot \vec{\omega}_i), (\vec{n} \cdot \vec{\omega}_o))} \quad (5.1c)$$

Código 18 – Parsing do nó Start.

```

1 parse_start :: proc(using p: ^Parser) -> ^ast.Start {
2     node := ast.new(ast.Start)
3     decls := [dynamic]^ast.Decl{}
4
5     for peek(p).kind != .EOF {
6         decl := parse_equation_begin_end_block(p)
7         append(&decls, decl)
8     }
9     node.eof = next(p, Token_Kind.EOF)
10    node.decls = decls[:]
11    return node
12 }

```

Código 19 – Exemplo código escrito na linguagem EquationLang.

```

1 \begin{equation}
2     \rho_{\text{d}} = \text{vec}\{0.3, 0.3, 0.3\}
3 \end{equation}
4
5 \begin{equation}
6     \rho_{\text{s}} = \text{vec}\{0.0, 0.2, 1.0\} * 20
7 \end{equation}
8
9 \begin{equation}
10    f = \frac{\rho_{\text{d}}}{\pi} + \frac{\rho_{\text{s}}}{8\pi} *
11    \frac{(\text{vec}\{n\} \cdot \text{vec}\{h\})}{
12    \{(\text{vec}\{\omega_{\text{o}}\} \cdot \text{vec}\{h\}) *
13    \max((\text{vec}\{n\} \cdot \text{vec}\{\omega_{\text{i}}\}),
14    (\text{vec}\{n\} \cdot \text{vec}\{\omega_{\text{o}}\}))\}}
15 \end{equation}

```

Código 20 – Gramática para EquationLang parte 1.

```

start = decl* token_eof;

decl = decl_equation_begin_end_block;

decl_equation_begin_end_block =
    token_begin token_opencurly 'equation' token_closecurly
    decl_equation
    token_end token_opencurly 'equation' token_closecurly;

decl_equation = field;

field = expr token_equal expr;

expr = expr_identifier
    | expr_number
    | expr_vector_literal
    | expr_grouped
    | expr_prefix
    | expr_infix
    | expr_suffix

```

Código 21 – Gramática para EquationLang parte 2.

```

expr_number = token_number;

expr_vector_literal = token_vec
    --- Ex: '\vec{1, 1, 1}'
    token_opencurly
    (expr_number token_comma)* expr_number
    token_closecurly
;

expr_grouped = token_openparen expr token_closeparen;

expr_prefix =
    (token_sqrt | token_exp | token_tan | token_cos | token_sin |
     token_arctan | token_arccos | token_arcsin | token_minus |
     token_plus) expr
;

expr_infix = token_frac
    token_opencurly expr token_closecurly
    token_opencurly expr token_closecurly
    | expr token_plus      expr
    | expr token_minus     expr
    | expr token_mul       expr
    | expr token_cross     expr
    | expr token_cmpequal  expr
    | expr token_div       expr
    | expr token_caret     expr
;

expr_postfix = expr token_bang;

expr_function_call = expr token_openparen
    (expr token_comma)* expr
    token_closeparen
;

--- Mesmo que expr_function_call, em etapas posteriores é decidido
    qual tipo realmente é.
expr_function_definition = expr token_openparen
    (expr token_comma)* expr
    token_closeparen
;

```

Na definição da gramática ([Código 20](#)), utilizamos a mesma notação de sintaxe definida no [Código 9](#) para representá-la, exceto que uma sequência de ---, três hifens, significa um comentário para o leitor, ela não afeta a definição da gramática.

Essa gramática define regras para expressões, atribuições, agrupamento, literais de

números e vetores, chamadas de função, definições de funções, e vários operadores, como `expr_prefix` e `expr_infix`, com o intuito de criar uma vasta coleção de operadores com diferentes precedências que atinge o objetivo de entender a sintaxe necessário para definições de BRDFs em \LaTeX . A tabela de operadores (Tabela 4) usadas no Pratt Parsing é representá-la por uma função chamada `precedence_from_token` que implementa esse mapeamento. Dado um token, ela retorna um inteiro que representa sua precedência; quanto maior o número, maior a precedência. Note que os mesmos tokens podem ser prefixo ou infixo, por exemplo `'(` é o token do prefixo do agrupamento (ex: $(2 * 3)$) mas ao mesmo tempo é infixo para chamada de função $f(x)$; o mesmo ocorre com `'-'`.

Tipo de Token	Prefixo	Precedência
+	Sim	25
-	Sim	25
(Sim	100
:	Sim	100
*	Sim	100
!	Sim	300
(Não	500
>	Não	5
<	Não	5
+	Não	10
-	Não	10
×	Não	20
*	Não	20
/	Não	20
^	Não	30
!	Não	400

Tabela 4: Tabela de Precedência dos Tokens

5.2.2.1 Estrutura da Árvore de Sintaxe

Nesta seção, apresentamos os tipos de nós que compõem a árvore de sintaxe abstrata (AST), utilizada no compilador da linguagem `EquationLang`. A estrutura da AST é definida com vários tipos de nós para capturar diferentes elementos da sintaxe. Diferente da gramática definida no Código 29, aqui os nós são representados em nível de código. Note que a `Expr` mais genérica possui um campo `ty_inferred` do tipo `Type`, esse campo será preenchido pela etapa de análise semântica, e usado na geração de código. A seguir, listamos a representação semântica de cada nó, e citamos os campos que cada nó contém:

@@Fix this

- **Node:** estrutura base para todos os nós da AST. **Campos:** `kind (typeid)`, guarda um número que indica qual o tipo do nó.

- **Expr**: representa expressões de forma geral. **Campos**: `expr_derived ty_inferred` (Type)
- **Decl**: representa genericamente declarações.
- **Start**: o nó raiz da AST. **Campos**: `decls` (lista de Decl), `eof` (Token).
- **Decl_Equation**: representa uma equação. **Campos**: `field` (Field).
- **Field**: **Campos**: `name` (Expr), `equals` (Token), `value` (Expr).
- **Expr_Identifier**: representa identificadores. **Campos**: `identifier` (Token), `is_vector` (bool), `sub_expression` (Expr), `var` (Maybe(string)).
- **Expr_Number**: representa literais numéricos. **Campos**: `number` (Token).
- **Expr_Vector_Literal**: representa vetores literais. **Campos**: `vec` (Token), `numbers` ([] Expr_Number).
- **Expr_Grouped**: representa expressões agrupadas, geralmente por `,` mas é permitido agrupar por `{ }`. **Campos**: `open` (Token), `expr` (Expr), `close` (Token).
- **Expr_Prefix**: representa expressão com operador prefixo (ex: `-3`). **Campos**: `op` (Token), `right` (Expr).
- **Expr_Infix**: representa expressões para operador infixo, isto é entre expressões, (ex: `3+3`). **Campos**: `left` (ponteiro de Expr), `op` (Token), `right` (ponteiro de Expr).
- **Expr_Function_Call**: representa chamadas de função. **Campos**: `left` (ponteiro de Expr), `open` (Token), `exprs` (lista de ponteiros de Expr), `close` (Token).
- **Expr_Function_Definition**: representa definições de funções. **Campos**: `name` (Expr_Identifier), `open` (Token), `parameters` (lista de Expr_Identifier), `close` (Token).

No [Figura 5.1](#) comentamos que *parser* é capaz de lidar com identificadores aninhados, como por exemplo x_{i_1} ($x_{\{i_1\}}$). No [Código 22](#), apresentamos como são criados esses identificadores recursivamente. Primeiramente, esse código está inserido em um função bem maior, especificamente é um recorte de um `switch`¹ da enumeração [Código 15](#). Temos um case, que reconhece token de identificador ou símbolos especiais ($\omega, \theta, \phi, \rho, \alpha, \beta, \sigma, \pi, \epsilon$) ou simplesmente token de identificador e, ao fazer uma chamada recursiva a `parse_expr`, permite subíndices numéricos, identificadores, ou até expressões binárias como $n + 1$ em f_{n+1} . Isso oferece maior flexibilidade na hora de expressar funções e equações para descrever as BRDFs, é muito comum usar subíndices numéricos. Na etapa de geração de código isso é usado para diferenciar um símbolo de outro apesar de ter o mesmo token inicial, por exemplo, o primeiro token é f , mas f_1 é diferente semanticamente de f_2 .

¹ `switch` e `case` em Odin, funciona da mesma maneira que na linguagem de programação C

Código 22 – Parte do código de *parsing* de expressão para identificadores.

```

1  case .Identifier,
2      .Omega,      // \omega
3      .Theta,      // \theta
4      .Phi,        // \phi
5      .Rho,        // \rho
6      .Alpha,      // \alpha
7      .Beta,       // \beta
8      .Sigma,      // \sigma
9      .Pi,         // \pi
10     .Epsilon,    // \epsilon
11     node := ast.new(ast.Expr_Identifier)
12     node.identifier = next(p)
13     if peek(p).kind == Token_Kind('_') {
14         next(p)
15         if peek(p).kind == Token_Kind('{') {
16             next(p, '{')
17             node.sub_expression = parse_expr(p, prec)
18             next(p, '}')
19         } else {
20             //
21             // If we're not using 'identifier_{ }' then, we only
22             // allow simple number or identifier
23             //
24             sub_node := ast.new(ast.Expr_Identifier)
25             if peek(p).kind == .Number {
26                 // We only allow number as sub expressions
27                 sub_node.identifier = next_expects_kind(p, .Number)
28             } else {
29                 sub_node.identifier = next_expects_kind(p,
30                     .Identifier, ..SPECIAL_IDENTIFIERS[1:])
31             }
32             node.sub_expression = sub_node
33         }
34     }
35 }

```

Esse código serve de exemplos para outras expressões recursivas, como uma expressão infixa (operação binária). Sempre identificamos o token atual através de `peek()`, que vê 1 ou dois token adiante para decidir qual nó da AST deve ser construído. Em seguida, é calculado a variável `prec` que indica precedência do token atual, enfim 1 ou mais chamadas recursivas (`parse_expr`) são feitas para os campos que precisam de uma expressão aninhadas. Depois dos campos serem preenchidos a expressão é retornada.

5.3 Implementação do Padrão de Visitante (walker)

Desenvolvemos o pacote `walker` para auxiliar em 3 tarefas chaves: validação de precedência da AST gerada pelo *parser*; visualização da AST gráficamente; geração de código. O padrão visitante foi empregado para percorrer e operar em uma AST. Uma estrutura e uma função implementam esse padrão e agem em cima da AST. procedimentos implementam esse padrão e manipulam a AST: `Visitor` e `walk`, respectivamente.

O padrão visitor implementado na função `walk` representa um mecanismo genérico e recursivo para travessia da AST, permitindo a aplicação de transformações ou análises personalizadas em cada nó da árvore.

A estrutura `Visitor` (Código 23) encapsula uma função de visita polimórfica que pode ser chamada para cada tipo de nó, possibilitando um processamento flexível e extensível, onde o visitante pode modificar seu próprio estado durante a travessia, decidir continuar ou interromper o caminhamento, e realizar operações arbitrárias como transformação, análise semântica, geração de código ou depuração.

A função, no Código 24 implementa uma travessia profunda (*depth-first*) recursiva, que automaticamente percorre todos os nós da AST, incluindo declarações, expressões, statements e estruturas aninhadas, invocando a função de visita antes e depois da exploração de cada subárvore. Isso é útil para criação de visitors personalizados para diferentes propósitos como checagem de tipos, parentização de expressões, geração de gráficos para árvore.

Código 23 – Estrutura polimórfica `Visitor`

```
1
2 // Estrutura polimórfica, aceita um tipo qualquer, chamado de
   DataType, como estrada para criar um tipo concreto.
3 Visitor :: struct (DataType: typeid) {
4     visit: proc(visitor: ^Visitor(DataType), node: ^ast.Node) ->
         ^Visitor(DataType),
5     data:  DataType,
6 }
```

5.3.1 Validação de Precedencia

Utilizamos o pacote `walker` para validão precedencia de operadores na AST gerada pelo `parser`. A função de parentização implementa inserção automática de parênteses que captura a precedência original das operações na AST, garantindo que a representação textual preserve a ordem de avaliação das expressões matemáticas. Através de uma travessia disponível pelo pacote usado, o algoritmo cria uma cadeia de caracteres com parênteses adicionais em expressões com prefixo, expressões binárias, chamada de funções. Isso é feita todas as os tipos de expressões.

Essa reprodução explícita da hierarquia de operações permite verificar automaticamente se a construção da AST durante o parsing manteve corretamente as regras de precedência.

Cada teste de precedência consiste em um texto original e um texto com parênteses esperados, como demonstrado na listagem [Código 25](#). Tentamos testar os casos mais complexos de expressões matemáticas, operações como exponenciação, que é associativo pela direita, combinado com operadores associativo pela esquerda com diferentes precedências

À medida que o compilador foi sendo desenvolvido esses testes se mostraram úteis em prevenir quebra de casos anteriores, pois ao dar suporte a nova funcionalidade, é possível quebrar funcionalidade já estabelecida anteriormente. ,

- **walker_interp**: interpreta a AST, calculando o valor numérico das expressões.
- **walker_paren**:
- **walker_print**: imprime os nós da AST e seus atributos, facilitando a depuração e compreensão da estrutura da AST.

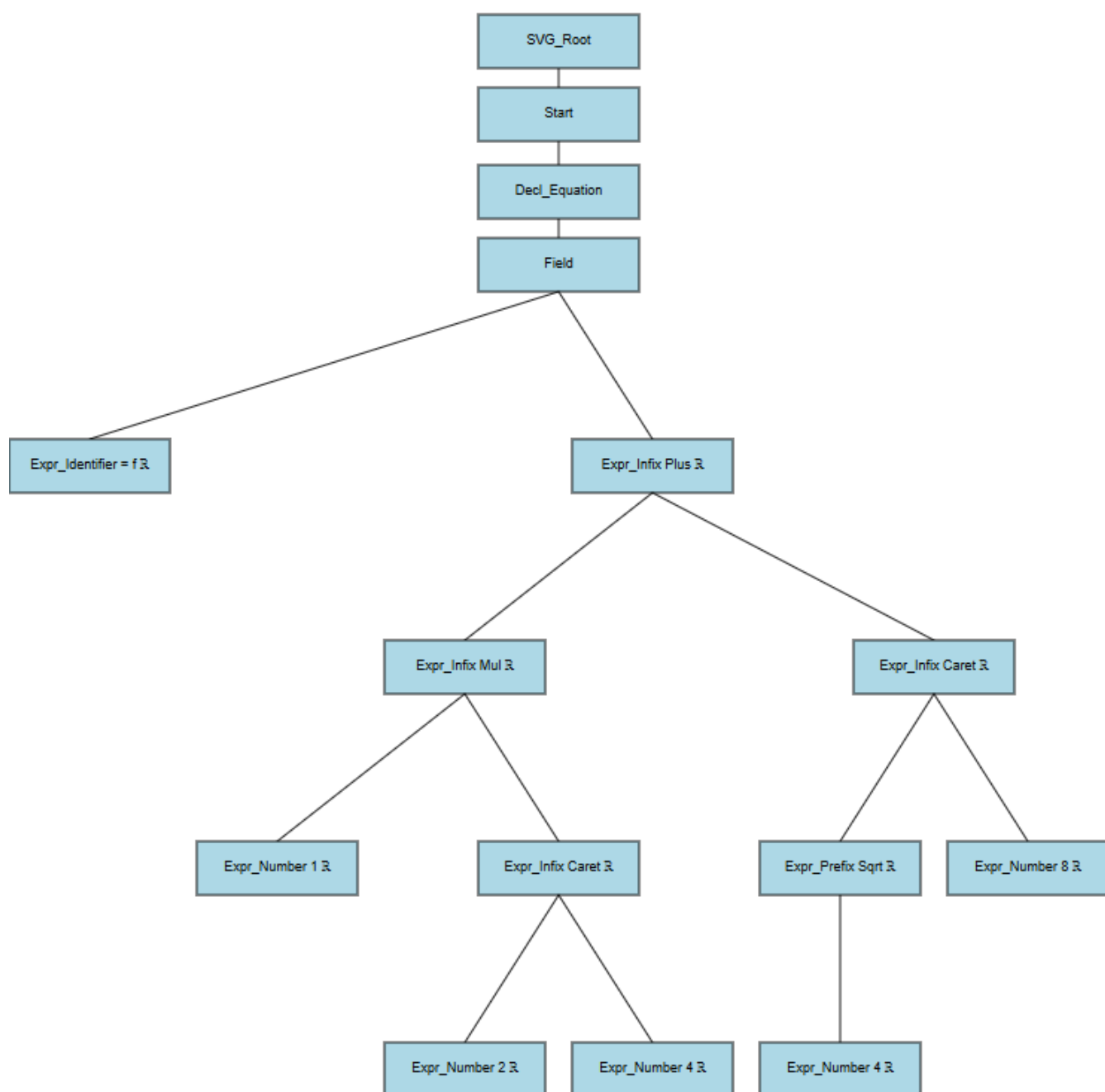
5.3.2 Visualização da AST por Imagem

Para validação visual, foi implementado uma função que gera uma imagem no formato “SVG”, que é um formato textual, da árvore contendo informações circulares, representados os nós da AST, juntamente com textos subscritos informados metadados sobre os nós, como o tipo de operador, o tipo de nó, a string do identificador no caso de ser @etc.

Como exemplo, na [Figura 20](#) temos a visualização do SVG gerado pela equação [Equação 5.2](#). Notamos que os nós de expressões binárias + e - próximo da raiz seriam avaliados depois, já os nós mais próximos das folhas devem ser resolvidos primeiro indicando uma precedência maior, como expressões binárias * e ^. Esse SVG também anota o tipo da expressão, (note que “f” é do tipo \mathbb{R}), é feito na etapa de checker, veremos mais a frente como isso é feito.

@@@ Os nós são heterogêneos, a maneira de acessar um filho de cada nó depende do tipo, pois o campo varia de nome ou posição na estrutura, o pacote walker também permite extrair os filhos de maneira uniforme para qualquer tipo de nó através de uma função chamada children ([Código 26](#)). (funções como “children” que dado um nó abstrato, ele resolve qual o tipo resolvido e cria um array de nós, como extrair os filhos). Ela é usada para simplificar o código de geração do SVG ao agir em cima de um nó de maneira uniforme, sem se preocupar com o tipo do nó.

$$f = 1 * 2^4 + \sqrt{4}^8 \quad (5.2)$$

Figura 20 – SVG da AST gerado para [Equação 5.2](#).

Código 24 – Estrutura Visitor e função de percurso walk.

```

1  // Por brevidade vamos omitir varios casos do 'switch' que seguem a
   mesma lógica
2  walk :: proc(v: ^Visitor($T), node: ^ast.Node) {
3      if v == nil || node == nil {
4          return
5      }
6      v := v->visit(node)
7      if v == nil {
8          return
9      }
10     using ast
11     switch n in &node.derived {
12         case ^Start:
13             for d in n.decls {
14                 walk(v, d)
15             }
16
17         case ^Decl_Equation:
18             walk(v, n.field)
19
20         case ^Field:
21             walk(v, n.name)
22             walk(v, n.value)
23
24         case ^Expr_Number:      // Caso base
25
26         case ^Expr_Vector_Literal:
27             for number in n.numbers {
28                 walk(v, number)
29             }
30         case ^Expr_Identifier:
31             walk(v, n.sub_expression)
32
33         // ...
34         // casos OMITIDOS aqui Também
35         // ...
36         case ^Expr_Infix:
37             walk(v, n.left)
38             walk(v, n.right)
39
40         case ^Expr_Grouped:
41             walk(v, n.expr)
42
43         case ^Expr_Function_Call:
44             walk(v, n.left)
45             for e in n.exprs {
46                 walk(v, e)
47             }
48         case:
49             assert(false, "Unhandled token on walk_print ")
50     }
51     v = v->visit(nil)
52 }

```

Código 25 – Teste de precedência usando por parentização.

```
1 test_paren(  
2     'a = 1+2', // Entrada  
3     'a=(1+2)' // Saída Esperada  
4 )  
5  
6 test_paren(  
7     'a = \exp 1 + 2^3', // Entrada  
8     'a=(\exp(1)+(2^3))' // Saída Esperada  
9 )  
10  
11 // ...  
12 // Outros Testes  
13 // ...  
14  
15 test_paren(  
16     'a = a(1*2 ^ 4 + \sqrt 4^8 , 2)', // Entrada  
17     'a=a(((1*(2^4))+(\sqrt(4)^8)),2)' // Saída Esperada  
18 )
```

Código 26 – Assinatura da função que extrai nós filhos de maneira uniforme para qualquer tipo de nó.

```
1 // Aceita um ponteiro de nós abstrato e return uma lista de nós  
   filhos  
2 children :: proc(node: ^Node) -> (array :[dynamic]^Node);
```

6

Resultados

Este capítulo apresenta os resultados iniciais de dois experimentos distintos, cada um focado em explorar aspectos importantes para este trabalho. Inicialmente, abordaremos o desenvolvimento de um compilador, seguido por uma investigação prática sobre técnicas de *ray tracing*.

O experimento com o compilador visa proporcionar uma compreensão mais profunda do desenvolvimento de compiladores. Isso foi realizado através da escolha de uma linguagem simples para implementação, que será chamada `SimpleLang`. Por ser uma linguagem com menos regras de sintaxe que o `LATEX`, é possível a exploração do processo de compilação incluindo a tokenização, criação da árvore sintática e interpretação do código-fonte por meio dessa árvore sintática. Esse compilador foi desenvolvido sem o uso de bibliotecas externas, o que amplia o entendimento sobre os fundamentos do desenvolvimento de compiladores.

Por outro lado, o experimento com o *ray tracing* representa um estudo prático sobre BRDFs e a linguagem Odin. Embora o objetivo principal seja compreender melhor o funcionamento das funções de distribuição de reflectância bidirecional (BRDFs), há uma conexão futura com o compilador, pois há a possibilidade de integração do *ray tracer* implementado como um pré-visualizador das BRDFs. No entanto, o foco principal permanece no compilador, enquanto o *ray tracer* serve como uma oportunidade para explorar as capacidades da linguagem Odin e aplicar os conceitos de radiometria.

Esses experimentos se complementam, proporcionando uma abordagem que explora tanto os aspectos teóricos quanto práticos relacionados ao desenvolvimento de compiladores e à aplicação de conceitos como BRDFs

6.1 Parser e Lexer em Odin

Desenvolvemos um *lexer*, *parser* e interpretador para uma linguagem simples chamada SimpleLang, juntamente com sua gramática, utilizando o Pratt *Parsing* na linguagem de programação Odin. O repositório pode ser encontrado em <https://github.com/evertonse/pratt-parser>. Esse *parser* é implementado por descida recursiva, o que significa que cada regra de produção tem uma função de análise associada. A implementação prioriza a simplicidade de código e a clareza de ideias, com extensos comentários para auxiliar na compreensão. Isso é importante, pois esse *parser* será modificado para aceitar *tokens* e sintaxe de \LaTeX .

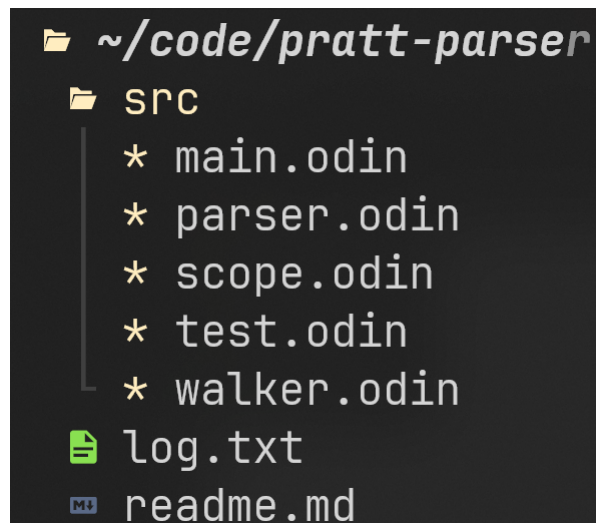
6.1.1 Parser

Ao contrário dos *parser* de descida recursiva tradicionais, que muitas vezes exigem várias chamadas de função aninhadas para cada nível de precedência, o nosso *parser* organiza as funções de análise hierarquicamente com base na precedência do operador, como demonstrado no Código 27. Esse código é a parte principal do *parsing* de expressões. Nessa implementação usamos a notação original de Pratt (PRATT, 1973), as funções `null_denotations` e `left_denotations` são equivalentes as funções `token.prefixo` e `token.infixo` declaradas no Algoritmo 1, respectivamente. Os pacotes desse projeto estão definidos na Figura 21.

Código 27 – Parsing de expressão em código Odin.

```
1
2
3 parse_expr :: proc(prec_prev: i64) -> ^Expr {
4     /* expressions that takes nothing (null) as left operand */
5     left := parse_null_denotations()
6     /*
7     . if current token is left associative or current token has
        higher precedence
8     . than previous precedence then stay in the loop, effectively
        creating a left leaning
9     . sub-tree, else, we recurse to create a right leaning sub-tree.
10    */
11    for precedence(peek()) > prec_prev + associativity(peek()) {
12        /* expressions that needs a left operand such as postfix,
13        mixfix, and infix operator */
14        left = parse_left_denotations(left)
15    }
16    return left
17 }
```

Figura 21 – Estrutura de pacotes do compilador.



6.1.2 Gramática

Para formalizar a gramática da linguagem de entrada (SimpleLang) deste compilador, definimos suas regras no [Código 29](#). Um exemplo de código-fonte válido em SimpleLang é apresentado no [Código 28](#).

Código 28 – Exemplo código escrito na linguagem SimpleLang.

```
1  epsilon = 0.001; # proximidade em float
2
3
4  abs = fn(a) a lt 0 or a lt -0 ? -a : a;
5
6
7  float_close = fn(a, b)
8      abs(a - b) lt epsilon ?
9      1 : 0;
10
11
12  # Classical fibonacci
13  fib = fn(n)
14      float_close(n, 0) ?
15      0
16      : float_close(n, 1) ?
17      1
18      : fib(n-1) + fib(n-2);
19
20
21  fib(10) # Ultima expressao significa retorno do main
```

Código 29 – Gramática para SimpleLang.

```

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'
        ;

```

Na definição da gramática (Código 29), utilizamos uma notação leve de sintaxe para representá-la. Palavras com todas as letras minúsculas são não-terminais, enquanto palavras entre aspas simples representam literalmente um *token* com esse conteúdo. Palavras em letras maiúsculas representam um *token* que pode variar, mas mantém o mesmo significado semântico. Por exemplo, NUMBER pode ser 2.0 ou 1.0, mas nas regras de produção eles são tratados de maneira idêntica. O símbolo “*” indica zero ou mais ocorrências, “()” indica agrupamento para aplicar um operador a ele, “|” simboliza o início de uma regra alternativa para o mesmo não-terminal e “=” indica uma produção.

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 precedências para facilitar a transição da sintaxe de SimpleLang para a sintaxe \LaTeX futuramente.

6.1.3 Tabela de Símbolos

Nesse projeto, foi desenvolvido também uma tabela de símbolos simples, cuja implementação será reaproveitada na análise semântica e na geração de código GLSL futuramente. 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. No [Código 30](#) temos a estrutura `Scope`, que representa um mapeamento de nomes para objetos de símbolo dentro de um **único escopo**, e também a estrutura `Scope_Table`, que mantém uma **pilha de escopos**, permitindo aninhamento.

6.1.3.1 Estrutura de Símbolos

Cada objeto 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.
- `body`: um ponteiro para a expressão que representa o corpo da função (se aplicável).

Código 30 – Código da estrutura de símbolos escrito em Odin.

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


6.1.3.2 Gerenciamento de Escopo

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

- `scope_enter`: entrar em um novo escopo, anexando-o à pilha de escopos.
- `scope_exit`: sai do escopo atual, removendo-o da pilha de escopos e o retornando.
- `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 adaptada para a fase de geração de código e tradução adequada do código-fonte em *shaders* GLSL.

6.1.3.3 Estrutura da Árvore de Sintaxe

Nesta seção, apresentamos os tipos de nós que compõem a árvore de sintaxe abstrata (AST), utilizada no compilador da linguagem SimpleLang. A estrutura da AST é definida com vários tipos de nós para capturar diferentes elementos da sintaxe. Diferente da gramática definida no [Código 29](#), aqui os nós são representados em nível de código. A seguir, listamos a representação semântica de cada nó:

- **Ast**: a estrutura base para todos os nós da AST.
- **Start**: representa o ponto de partida do programa, contendo uma sequência de atribuições seguidas 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 de maneira abstrata, servindo como a estrutura base tipos concretos 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 (prefixo).
- **Expr_Infix**: representa operações binárias (infixo).

- **Expr_Postfix**: representa operações unárias (sufixo).
- **Expr_Mixfix**: representa operações ternárias.
- **Expr_Function_Call**: representa chamadas de função com argumentos.

6.1.4 Implementação do Padrão de Visitante

O padrão visitante foi empregado para percorrer e operar em uma AST. Três procedimentos implementam esse padrão e manipulam 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.

6.1.5 Testes

Foi desenvolvida uma série de testes que abrangem vários aspectos da funcionalidade do *parser*, incluindo geração de árvore de sintaxe, precedência de operadores e interpretação semântica.

6.1.5.1 Geração de Árvore de Sintaxe

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

- 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 confirmar regras de precedência.

6.1.5.2 Interpretação Semântica

Além da geração da árvore de sintaxe e da definição de precedência de operadores, foram realizados testes para garantir a interpretação semântica das expressões. Isso envolve avaliar as entradas e comparar a saída com os valores esperados. Quanto à verificação de tipos, é simples: cada variável pode ser uma função ou um número. Portanto, permitimos apenas operadores entre números e, além disso, booleanos também são considerados números, sendo 0 interpretado como falso e 1 como verdadeiro. Os testes de interpretação semântica realizados para SimpleLang abrangem:

- Avaliar expressões aritméticas envolvendo constantes, variáveis e chamadas de função recursivas.
- Verificar o comportamento de expressões condicionais (por exemplo, operador ternário) sob diferentes condições.

Ao testar geração de árvore de sintaxe, precedência de operadores e interpretação semântica, a implementação do Pratt *Parsing* foi validada quanto à correção e confiabilidade, pois obteve um desempenho robusto em vários cenários de entrada.

6.2 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 ¹, usada na renderização de imagens em uma janela. Isso foi feito para começar a entender melhor as BRDFs e a equação de renderização (Equação 2.7).

O *ray tracer*, que foi construído baseado no livro “Ray Tracing in One Weekend” ², opera inteiramente na unidade de processamento central (CPU). Sua funcionalidade principal envolve a modelagem de raios e a reflexão da cena para os *pixels* da imagem. A cena consiste exclusivamente em esferas, empregando cálculos de colisão padrão entre um raio e uma esfera.

6.2.1 Implementação de Materiais

O *ray tracer* inclui vários materiais que ditam o comportamento dos raios ao interagir com superfícies, os quais não são garantidos de serem fisicamente realistas em relação as propriedades de reflexão discutidas na subseção 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. Como demonstrado no Código 31, os seguintes materiais foram 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 índices de refração e reflexão.

6.2.2 Mecanismo de Reflexão de Raios

O mecanismo central do *ray tracer* envolve traçar raios pela cena para determinar suas interações com superfícies e calcular os valores de cor resultantes, o resultado pode ser encontrado na Figura 22. Esse processo foi implementado considerando os seguintes passos:

1. **Geração de Raios:** raios são gerados a partir do ponto de vista da câmera e projetados na cena.
2. **Deteção de Colisão:** cada raio é testado quanto à interseção com objetos na cena.

¹ <<https://www.raylib.com/>>

² <<https://raytracing.github.io/books/RayTracingInOneWeekend.html>>

Código 31 – Materiais.

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

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

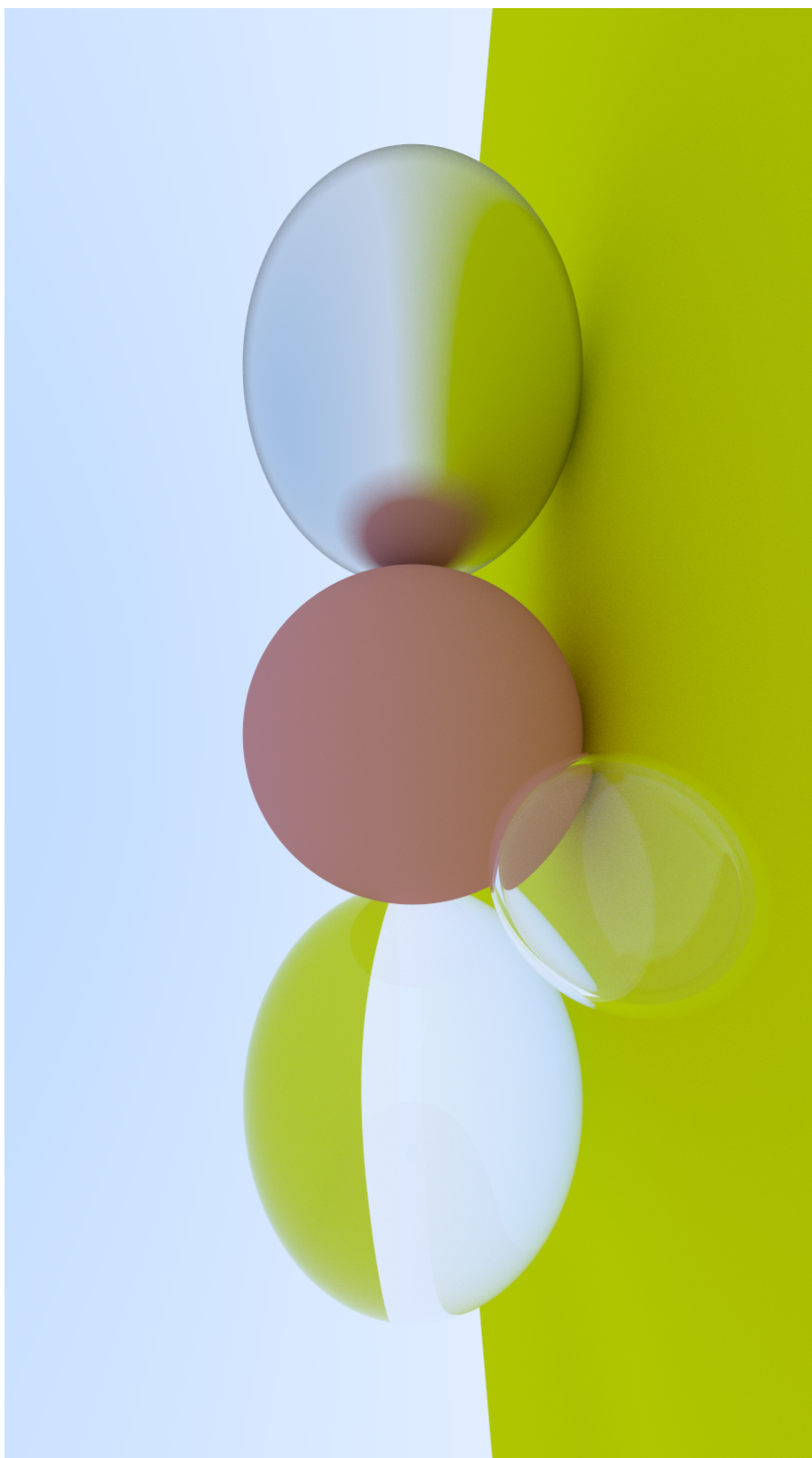
Lambertian_Material :: struct {
    using _ : Material,
    albedo: Color,
}

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

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

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.

Figura 22 – Imagem gerada por ray tracing conforme a implementação em Odin.



Referências

- BRADY, A. et al. genbrdf: Discovering new analytic brdfs with genetic programming. *ACM Transactions on Graphics (TOG)*, ACM New York, NY, USA, v. 33, n. 4, p. 1–11, 2014. Citado na página 25.
- CEM, Y. *Intro to Graphics 07 - GPU Pipeline*. 2020. Disponível em: <https://youtu.be/UzlnprHSbUw?si=Y0a0Tj7ia-lW_eGC>. Citado na página 15.
- DAVISONPRO. *Criando um jogo em JavaScript*. 2024. Disponível em: <<https://bulldogjob.pl/readme/tworzenie-gry-w-javascript>>. Citado na página 17.
- DISNEY, M.; LEWIS, P.; NORTH, P. Monte carlo ray tracing in optical canopy reflectance modelling. *Remote Sensing Reviews*, Taylor & Francis, v. 18, n. 2-4, p. 163–196, 2000. Citado na página 9.
- GENG, C. et al. Tree-structured shading decomposition. In: *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. [S.l.: s.n.], 2023. p. 488–498. Citado na página 26.
- HE, Y.; FATAHALIAN, K.; FOLEY, T. Slang: language mechanisms for extensible real-time shading systems. *ACM Transactions on Graphics (TOG)*, ACM New York, NY, USA, v. 37, n. 4, p. 1–13, 2018. Citado na página 26.
- JÄGER, G.; ROGERS, J. Formal language theory: refining the chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, The Royal Society, v. 367, n. 1598, p. 1956–1970, 2012. Citado 2 vezes nas páginas 17 e 18.
- JUDICE, S. F.; GIRALDI, G. A.; KARAM-FILHO, J. *Rendering Equation*. 2019. Citado na página 9.
- KAJIYA, J. T. The rendering equation. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. [S.l.: s.n.], 1986. p. 143–150. Citado na página 11.
- MONTES, R.; UREÑA, C. An overview of BRDF models. *University of Grenada, Technical Report LSI-2012-001*, 2012. Citado na página 12.
- OHBUCHI, E.; UNNO, H. A real-time configurable shader based on lookup tables. In: *IEEE. First International Symposium on Cyber Worlds, 2002. Proceedings*. [S.l.], 2002. p. 507–514. Citado 2 vezes nas páginas 27 e 28.
- OpenGL Architecture Review Board. *OpenGL 4.6 Core Specification*. [S.l.], 2017. https://www.khronos.org/registry/OpenGL/index_gl.php. Citado na página 14.
- PHARR, M.; JAKOB, W.; HUMPHREYS, G. *Physically Based Rendering: From Theory to Implementation (3rd ed.)*. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. 1266 p. ISBN 9780128006450. Citado 6 vezes nas páginas 6, 9, 10, 11, 12 e 29.
- PRATT, V. R. Top down operator precedence. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. [S.l.: s.n.], 1973. p. 41–51. Citado 3 vezes nas páginas 19, 47 e 60.

- PROKHOROV, A. V.; HANSSEN, L. M.; MEKHONTSEV, S. N. Calculation of the radiation characteristics of blackbody radiation sources. *Experimental Methods in the Physical Sciences*, Elsevier, v. 42, p. 181–240, 2009. Citado na página 9.
- RABIN, M. O. Mathematical theory of automata. In: *Proc. Sympos. Appl. Math.* [S.l.: s.n.], 1967. v. 19, p. 153–175. Citado na página 19.
- TAN, P. Phong reflectance model. *Computer Vision: A Reference Guide*, Springer, p. 1–3, 2020. Citado na página 14.
- The Khronos Group. *OpenGL Interpolation*. 2015. Disponível em: <<https://www.khronos.org/opengl/wiki/Interpolation>>. Citado na página 16.
- WEYRICH, T. et al. [S.l.: s.n.], 2009. Citado na página 10.
- WOLFE, W. L. *Introduction to radiometry*. [S.l.]: Spie press, 1998. v. 29. Citado 2 vezes nas páginas 8 e 27.
- ZEYU, Z. et al. The generalized laws of refraction and reflection. *Opto-Electronic Engineering*, Opto-Electronic Engineering, v. 44, n. 2, p. 129–139, 2017. Citado na página 12.