

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

**Compilador de Funções de Distribuição de Reflexão
Bidirecional descritas em L^AT_EX para Linguagem de Shading**

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

**Compilador de Funções de Distribuição de Reflexão
Bidirecional descritas em L^AT_EX para Linguagem de Shading**

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 L^AT_EX 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, descritas em equações L^AT_EX, para o código GLSL utilizado na 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 de 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*.

Listas de ilustrações

Figura 1 – Visualização da radiância em uma direção específica do hemisfério.	19
Figura 2 – Ângulo sólido s do objeto B visto pelo ponto p .	20
Figura 3 – Reflexão especular. Em vermelho está o raio incidente, e em azul o raio de saída.	22
Figura 4 – Reflexão Difusa. Note que os raios refletidos não dependem do ângulo de entrada.	22
Figura 5 – Reflexão <i>glossy</i> .	23
Figura 6 – Reflexão retro-refletora.	23
Figura 7 – O <i>pipeline</i> .	24
Figura 8 – Diferença entre shading a nível de vértice e shading a nível de fragmento.	26
Figura 9 – Exemplo de circuito de produto interno entre vetores.	37
Figura 10 – Ferramenta de visualização de BRDFs da Disney.	45
Figura 11 – Estrutura de geral da arquitetura do compilador.	48
Figura 12 – Estrutura de pacotes do compilador.	48
Figura 13 – Erro ao usar símbolo não definido.	51
Figura 14 – Erro ao tentar símbolo não definido.	51
Figura 15 – Erro de tipos incompatíveis.	52
Figura 16 – Erro de balanceamento de parênteses.	52
Figura 17 – Erro de uso incorreto de palavras reservadas.	52
Figura 18 – Erro de <i>token</i> incapaz de produzir expressão.	52
Figura 19 – Erro de redefinição de símbolo.	53
Figura 20 – SVG da AST gerado para Equação 5.2.	70
Figura 21 – Erro reportado sobre dependencia circular.	77
Figura 22 – Erro gerado por uso incorreto de tipos na chamada de função.	82
Figura 23 – Equações da BRDF do experimento blinn-phong-Kay em documento <i>LATEX</i> .	95
Figura 24 – <i>Plots</i> da BRDF deste experimento.	95
Figura 25 – Objetos 3D renderizados por este experimento	96
Figura 26 – Equações da BRDF do experimento cook-torrance em documento <i>LATEX</i> .	99
Figura 27 – <i>Plots</i> da BRDF deste experimentos.	99
Figura 28 – Objetos 3D renderizados por este experimento	99
Figura 29 – Equações da BRDF do experimento ward em documento <i>LATEX</i> .	103
Figura 30 – <i>Plots</i> da BRDF deste experimento.	104
Figura 31 – Objetos 3D renderizados por este experimento	104
Figura 32 – Equações da BRDF do experimento ashikhmin-shirley-close-to-original-Kay em documento <i>LATEX</i> .	109
Figura 33 – <i>Plots</i> da BRDF deste experimento.	110
Figura 34 – Objetos 3D renderizados por este experimento	110
Figura 35 – Equações da BRDF do experimento oren-nayar em documento <i>LATEX</i> .	115

Figura 36 – <i>Plots</i> da BRDF deste experimento.	116
Figura 37 – Objetos 3D renderizados por este experimento	116
Figura 38 – Equações da BRDF do experimento ashikhmin-shirley-alternative em documento <i>L^AT_EX</i>	122
Figura 39 – <i>Plots</i> da BRDF deste experimento.	122
Figura 40 – Objetos 3D renderizados por este experimento	123
Figura 41 – Equações da BRDF do experimento cook-torrance-alternative em documento <i>L^AT_EX</i>	126
Figura 42 – <i>Plots</i> da BRDF deste experimento.	126
Figura 43 – Objetos 3D renderizados por este experimento	127
Figura 44 – Equações da BRDF do experimento dues em documento <i>L^AT_EX</i>	130
Figura 45 – <i>Plots</i> da BRDF deste experimento.	130
Figura 46 – Objetos 3D renderizados por este experimento	131
Figura 47 – Equações da BRDF do experimento edwards-2006 em documento <i>L^AT_EX</i>	134
Figura 48 – <i>Plots</i> da BRDF deste experimento.	134
Figura 49 – Objetos 3D renderizados por este experimento	135
Figura 50 – Equações da BRDF do experimento Kajiya-Kay em documento <i>L^AT_EX</i>	138
Figura 51 – <i>Plots</i> da BRDF deste experimento.	139
Figura 52 – Objetos 3D renderizado no experimento BRDF Anisotrópica: Kajiya-Kay (1989)	139
Figura 53 – Equações da BRDF do experimento minnaert em documento <i>L^AT_EX</i>	143
Figura 54 – <i>Plots</i> da BRDF deste experimento.	143
Figura 55 – Objetos 3D renderizados por este experimento	144

Lista de tabelas

Tabela 1 – Tabela de pesquisa.	34
Tabela 2 – Resultados das bases após aplicar os critérios.	35
Tabela 3 – bases	38
Tabela 4 – Tabela de especificação da linguagem.	41
Tabela 5 – Tabela de símbolos e suas descrições	42
Tabela 6 – Tabela de Precedência dos Tokens	64
Tabela 7 – Tabela de mapeamento de conveções para código GLSL	87
Tabela 8 – Tabela dos Experimentos	94

Listas de códigos

Código 1 – Exemplo GLSL de <i>shader</i> de vértice.	25
Código 2 – Exemplo de código escrito em C.	30
Código 3 – Cálculo vetorial em código-fonte L ^A T _E X.	31
Código 4 – Cálculo vetorial em código GLSL.	31
Código 5 – Código de interface em Slang.	36
Código 6 – Código-fonte de função quadrática.	40
Código 7 – Código GLSL da função quadrática g.	40
Código 8 – Entrada em L ^A T _E X (Cook-Torrance BRDF).	42
Código 9 – Saída em GLSL esperada (Cook-Torrance BRDF).	43
Código 10 – O código GLSL com sintaxe extra para definir parâmetros.	46
Código 11 – Função principal do Lexer.	50
Código 12 – Função de erro exposto pelo pacote lexer.	50
Código 13 – Estruturas do Lexer.	54
Código 14 – Mapa de identificadores especiais.	55
Código 15 – Gramática ilustrativa para tokens.	56
Código 16 – Enumeração dos tipos de tokens.	57
Código 17 – Regras tradicionais de precedência por gramática.	59
Código 18 – Parsing de expressão em código Odin.	60
Código 19 – Estrutura e função principal do pacote parser.	60
Código 20 – Parsing do nó Start.	61
Código 21 – Exemplo código escrito na linguagem EquationLang.	62
Código 22 – Gramática para EquationLang parte 1.	62
Código 23 – Gramática para EquationLang parte 2.	63
Código 24 – Parte do código de parsing de expressão para identificadores.	66
Código 25 – Estrutura polimórfica Visitor. O DataType é o parâmetro concreto dessa estrutura.	67
Código 27 – Validação de precedência por parentização de expressões.	68
Código 28 – Assinatura da função que extrai nós filhos de maneira uniforme para qualquer tipo de nó.	69
Código 29 – Estruturas que representam os tipos de expressões da AST.	73
Código 30 – Estrutura do Símbolo.	74
Código 31 – Código da estrutura de símbolos escrito em Odin.	74
Código 32 – Estrutura de grafo de dependências.	76
Código 33 – Entrada para o compilador que gera dependência circular.	76
Código 35 – Identificadores embutidos pela convenção deste trabalho.	79
Código 36 – Recorte da função check_expr.	80
Código 37 – Equação com uso incorreto de tipos na chamada de função.	82

Código 38 – Validação de parâmetros de uma função.	82
Código 26 – Função de percurso walk.	88
Código 34 – Parte do switch da inferencia de tipos.	89
Código 39 – Emissão de chamada de funções.	89
Código 40 – Exemplo de código de expressão gerado.	90
Código 41 – Emitir expressão.	91
Código 42 – Recorte da função BRDF one as variaveis built-ins são inicializadas	92
Código 43 – Código GLSL gerado pelo compilador para as funções de normalização e reflexão de vetores.	92
Código 44 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	96
Código 45 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	97
Código 46 – Código fonte da BRDF deste experimento (parte 1).	98
Código 47 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	100
Código 48 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	101
Código 49 – Código fonte da BRDF deste experimento.	102
Código 50 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	105
Código 51 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	106
Código 52 – Código fonte da BRDF deste experimento (parte 1).	107
Código 53 – Código fonte da BRDF deste experimento (parte 2).	108
Código 54 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	111
Código 55 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	112
Código 56 – Código fonte da BRDF deste experimento (parte 1).	113
Código 57 – Código fonte da BRDF deste experimento (parte 2).	114
Código 58 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	117
Código 59 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	118
Código 60 – Saída do compilador, código GLSL da BRDF deste experimento (parte 3).	119
Código 61 – Código fonte da BRDF deste experimento (parte 1).	120
Código 62 – Código fonte da BRDF deste experimento (parte 2).	121
Código 63 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	123
Código 64 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	124
Código 65 – Código fonte da BRDF deste experimento.	125
Código 66 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	127
Código 67 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	128
Código 68 – Código fonte da BRDF deste experimento.	129
Código 69 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	131
Código 70 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	132
Código 71 – Código fonte da BRDF deste experimento.	133
Código 72 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	135
Código 73 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	136

Código 74 – Código fonte da BRDF deste experimento.	137
Código 75 – Saída do compilador, código GLSL da BRDF do experimento baseado em Kajiya-Kay (parte 1).	140
Código 76 – Saída do compilador, código GLSL da BRDF do experimento baseado em Kajiya-Kay (parte 2).	141
Código 77 – Código fonte da BRDF do experimento Kajiya-Kay.	142
Código 78 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).	144
Código 79 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).	145
Código 80 – Código fonte da BRDF deste experimento (parte 1).	146

Lista de algoritmos

Sumário

1	Introdução	14
1.1	Motivação	14
1.2	Objetivo	15
1.3	Estrutura do Documento	15
2	Conceitos	17
2.1	Radiometria	17
2.1.1	Energia Radiante e Fluxo	18
2.1.2	Radiância e BRDF	19
2.2	Modelos de BRDFs	21
2.2.1	BRDF Pura Especular	21
2.2.2	BRDF Difusa Ideal	22
2.2.3	BRDF Brilhante	23
2.2.4	BRDF Retro-Refletora	23
2.3	Introdução ao Shading e ao <i>pipeline</i> de GPU	23
2.3.1	Shader de Vértice	25
2.3.2	Shader de Fragmento	25
2.4	Compiladores	26
2.4.1	Cadeia de Símbolos e Alfabeto	26
2.4.2	Definições de Linguagens	26
2.4.3	Compilador como uma Transformação	27
2.4.4	Gramática	27
2.4.4.1	Gramáticas Livres de Contexto (GLCs)	27
2.4.5	Análise Léxica	28
2.4.6	Análise Sintática ou <i>Parsing</i>	28
2.4.6.1	Pratt Parsing	29
2.4.6.2	Pseudo-código para Análise de Expressões	29
2.4.7	Análise Semântica	30
2.4.8	Geração da Linguagem Alvo	30
3	Revisão Bibliográfica	32
3.1	Mapeamento Sistemático	32
3.1.1	Seleção das Bases	32
3.1.2	Questões de Pesquisa	33
3.1.3	Termos de Busca	33
3.1.4	Critérios	34

3.1.4.1	Critérios de Inclusão	34
3.1.4.2	Critérios de Exclusão	34
3.1.5	Descrição dos Trabalhos Relacionados	35
3.1.5.1	genBRDF: Discovering New Analytic BRDFs with Genetic Programming	35
3.1.5.2	Slang: language mechanisms for extensible real-time shading systems	36
3.1.5.3	A Real-Time Configurable Shader Based on Lookup Tables	37
3.2	Pesquisa por Repositórios Online	38
4	Metodologia	39
4.1	Análise e Técnicas	39
4.2	Especificação da Linguagem	40
4.3	Design de Casos de Teste	41
4.4	Implementação do Compilador	43
4.5	Experimentos de Renderização	44
5	Desenvolvimento	47
5.1	Análise Léxica (<i>lexer</i>)	49
5.1.1	Reporte de Erros	50
5.1.2	Classificação e Extração de Tokens	53
5.2	Análise Sintática, (<i>parser</i>)	58
5.2.1	Sintaxe de Gramáticas	58
5.2.2	Parser	58
5.2.3	Gramática	61
5.2.3.1	Estrutura da Árvore de Sintaxe	64
5.3	Implementação do Padrão de Visitante (<i>walker</i>)	66
5.3.1	Estrutura <i>Visitor</i>	67
5.3.2	Função <i>walk</i>	67
5.3.3	Validação de Precedência	68
5.3.4	Visualização da AST por Imagem	69
5.4	Análise Semântica (<i>checker</i>)	71
5.4.1	Funções do Pacote <i>checker</i>	71
5.4.2	Tipos, Símbolos e Escopos	72
5.4.2.1	Tipos	72
5.4.2.2	Símbolos	73
5.4.3	Escopo e Tabela de Símbolos	74
5.4.4	Resolução de Símbolos	75
5.4.5	Inferência de Tipos	77
5.4.6	Validação de Equações	78

5.4.7	Validação de Funções	80
5.4.7.1	Definição de Funções	80
5.4.7.2	Chamada de Funções	81
5.5	Geração de Código (<code>emitter</code>)	82
5.5.1	Emissão de Equações	83
5.5.2	Unicidade de Variáveis	83
5.5.3	Geração de Expressões	84
5.5.4	Fases do Processo de Geração do Shader	85
5.5.4.1	Variáveis Built-in	86
5.5.5	Emissão de Definições de Função	87
6	Resultados	93
6.1	Experimento BRDF Blinn-Phong	95
6.1.1	Representação em documento L ^A T _E X	95
6.1.2	Visualização do Resultado	95
6.1.3	Código GLSL Gerado	96
6.1.4	Código Fonte em EquationLang	98
6.2	Experimento BRDF Cook-Torrance	98
6.2.1	Representação em documento L ^A T _E X	99
6.2.2	Visualização do Resultado	99
6.2.3	Código GLSL Gerado	100
6.2.4	Código Fonte em EquationLang	102
6.3	Experimento BRDF Ward	102
6.3.1	Representação em documento L ^A T _E X	103
6.3.2	Visualização do Resultado	104
6.3.3	Código GLSL Gerado	105
6.3.4	Código Fonte em EquationLang	107
6.4	Experimento BRDF Ashikhmin-Shirley	108
6.4.1	Representação em documento L ^A T _E X	109
6.4.2	Visualização do Resultado	110
6.4.3	Código GLSL Gerado	111
6.4.4	Código Fonte em EquationLang	113
6.5	Experimento BRDF Oren-Nayar	114
6.5.1	Representação em documento L ^A T _E X	115
6.5.2	Visualização do Resultado	116
6.5.3	Código GLSL Gerado	117
6.5.4	Código Fonte em EquationLang	120
6.6	Experimento BRDF Ashikhmin-Shirley Alternativa	122
6.6.1	Representação em documento L ^A T _E X	122
6.6.2	Visualização do Resultado	122

6.6.3	Código GLSL Gerado	123
6.6.4	Código Fonte em EquationLang	125
6.7	Experimento BRDF Cook-Torrance Alternativa	125
6.7.1	Representação em documento L ^A T _E X	126
6.7.2	Visualização do Resultado	126
6.7.3	Código GLSL Gerado	127
6.7.4	Código Fonte em EquationLang	129
6.8	Experimento BRDF Dür	130
6.8.1	Representação em documento L ^A T _E X	130
6.8.2	Visualização do Resultado	130
6.8.3	Código GLSL Gerado	131
6.8.4	Código Fonte em EquationLang	133
6.9	Experimento BRDF Edwards 2006	133
6.9.1	Representação em documento L ^A T _E X	134
6.9.2	Visualização do Resultado	134
6.9.3	Código GLSL Gerado	135
6.9.4	Código Fonte em EquationLang	137
6.10	Experimento BRDF Anisotrópica baseado em Kajiya-Kay (1989)	137
6.10.1	Representação em documento L ^A T _E X	138
6.10.2	Visualização do Resultado	139
6.10.3	Código GLSL Gerado	140
6.10.4	Código Fonte em EquationLang	142
6.11	Experimento BRDF Minnaert	143
6.11.1	Representação em documento L ^A T _E X	143
6.11.2	Visualização do Resultado	143
6.11.3	Código GLSL Gerado	144
6.11.4	Código Fonte em EquationLang	146
7	Conclusão	147
	Referências	149

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 criação e edição 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 [\$\text{\LaTeX}\$](#) ¹,

¹ \LaTeX foi desenvolvido por Leslie Lamport ([LESLIE, 1994](#)), baseado no sistema $\text{T}_{\text{E}}\text{X}$ de Donald Knuth, e permite a representação de equações matemáticas complexas.

que é um sistema de preparação de documentos para alta qualidade tipográfica, geralmente utilizado para a criação de artigos científicos. Desta forma, uma abordagem promissora para simplificar a criação de *shaders* é o desenvolvimento de um compilador capaz de traduzir BRDFs escritas em L^AT_EX 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 L^AT_EX, seja capaz de gerar código de sombreamento (*shading*)² 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 L^AT_EX 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 as convenções matemáticas e operações que o compilador deve implementar.

O Capítulo 5 descreve o desenvolvimento deste trabalho, que consiste na implementação de um analisador léxico, sintático e semântico, além da geração de código GLSL, todos implementados na linguagem de programação Odin³. O método de análise sintática utilizado é o de Pratt, que, juntamente com a inferência de tipos e a visualização da AST por meio de arquivos de imagem, compõe boa parte do projeto. A linguagem desenvolvida possui uma gramática

²O termo *shading* refere-se ao processo computacional realizado por um *shader*, ou seja, o cálculo de cor, iluminação e textura para cada ponto de uma superfície em uma cena gráfica.

³<<https://odin-lang.org/>>

restrita em comparação com L^AT_EX, priorizando as construções necessárias para descrever BRDFs, o que torna viável a construção deste compilador.

O [Capítulo 6](#) apresenta os experimentos elaborados para validar a implementação. Nele, são apresentadas as equações que descrevem as BRDFs, acompanhadas do código gerado pelo compilador desenvolvido. Além disso, são exibidas as renderizações de três objetos distintos para cada código de BRDF testado. Por último, no [Capítulo 7](#), são recapitulados os objetivos propostos e discutidas as potenciais direções futuras deste trabalho.

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 das etapas ordenadas de renderização na GPU, conhecidas nesse contexto como *pipeline*¹. 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](#),

¹Do inglês, *pipeline* representa uma sequência de estágios de processamento, onde a saída de cada etapa serve como entrada para a próxima.

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.

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

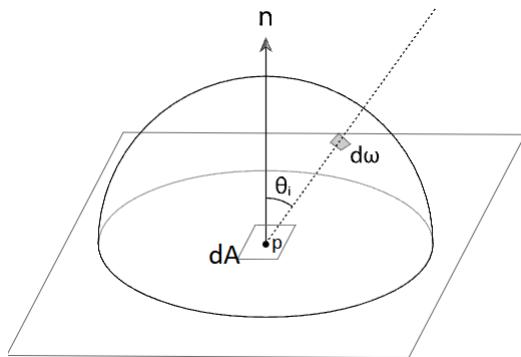
2.1.2 Radiância e BRDF

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).

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

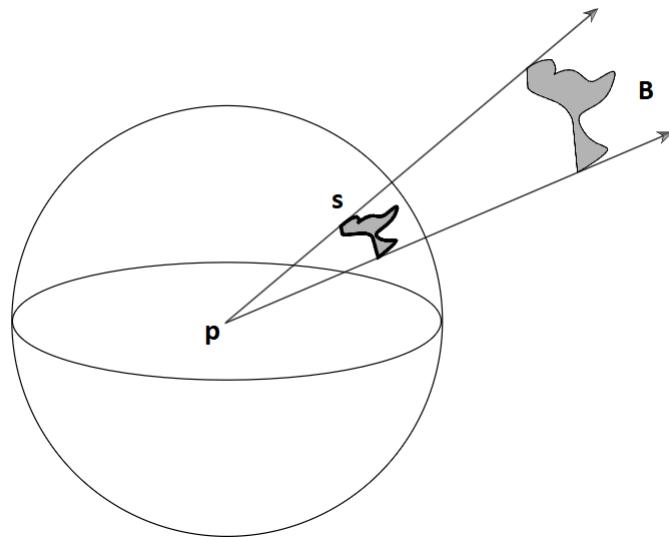
Figura 1 – Visualização da radiância em uma direção específica do hemisfério.



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

Equivalentemente, podemos definir radiância para diferentes orientações da superfície e direção do raio de luz incidente 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.

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

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

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

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 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.

$$\begin{aligned}
 L_o(p, \omega_o) &= L_e(p, \omega_o) + \int_{H^2} f(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos(\theta_i) d\omega_i \\
 L_o &\text{ é radiância de saída (*outgoing*)} \\
 L_e &\text{ é radiância emitida pela superfície (i.e. fonte de luz)} \\
 L_i &\text{ é radiância incidente na superfície} \\
 \omega_i &\text{ é a direção do raio incidente} \\
 \omega_o &\text{ é a direção do raio refletido} \\
 H^2 &\text{ são todas as direções no hemisfério no ponto } p \\
 \theta_i &\text{ ângulo entre direção incidente e a normal da superfície} \\
 f &\text{ função de refletância}
 \end{aligned} \tag{2.7}$$

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. Ele 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

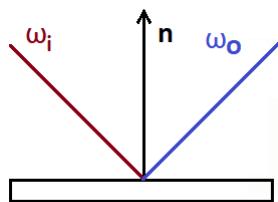
As próximas seções apresentam alguns modelos comuns de BRDFs na literatura (MONTES; UREÑA, 2012).

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.

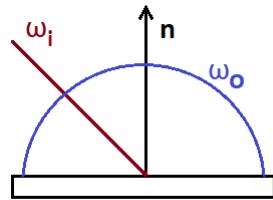
2.2.2 BRDF Difusa Ideal

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 superfície² e θ é o ângulo entre a direção da luz incidente e a normal da superfície. O termo coseno garante que a radiância refletida seja proporcional ao coseno 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.

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

²Albedo é a propriedade óptica que quantifica a fração da radiação incidente que é refletida por uma superfície. Varia de 0 (superfície totalmente absorvente) a 1 (superfície que reflete toda a luz incidente)

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

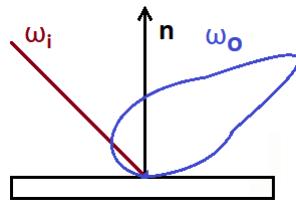


Fonte: Autor.

2.2.3 BRDF Brilhante

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*.

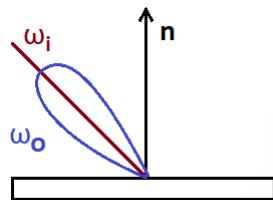


Fonte: Autor.

2.2.4 BRDF Retro-Refletora

A BRDF para uma superfície retro-refletora envolve, tipicamente, uma formulação matemática ou modelagem especializada que representa o redirecionamento da luz de volta para a fonte, como ilustrado na Figura 6.

Figura 6 – Reflexão retro-refletora.



Fonte: Autor.

2.3 Introdução ao Shading e ao *pipeline* de GPU

O termo *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, do inglês *Graphics Processing Unit* (GPU).

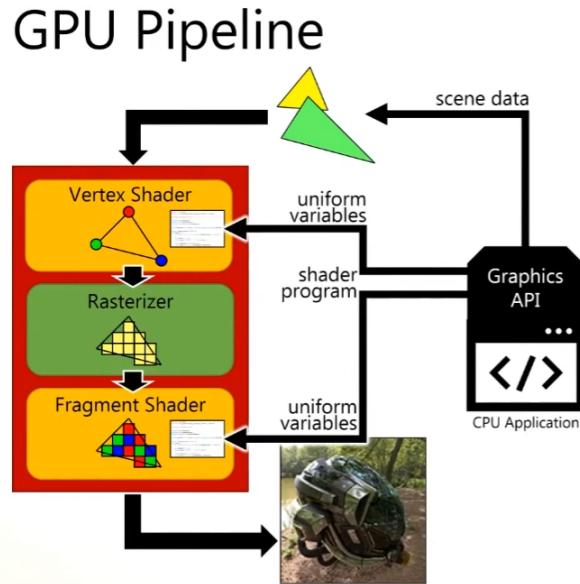
A interação com as GPUs é facilitada por meio de uma API, sendo o OpenGL uma das mais utilizadas para acessar funções da GPU ([OpenGL Architecture Review Board, 2017](#)). O *pipeline* de renderização do OpenGL é composto por várias etapas, incluindo a definição de dados de vértices e a aplicação de *shaders* de vértice e fragmento. Os vértices representam os pontos das primitivas geométricas, geralmente triângulos, que formam o objeto a ser renderizado, enquanto os fragmentos são os elementos gerados durante o processamento dessas primitivas, correspondendo a pontos discretos na tela onde a cor final será determinada.

Além desses *shaders*, o OpenGL oferece suporte a *shaders* de tesselação e geometria, que são etapas opcionais: a tesselação subdivide superfícies em triângulos menores para aumentar o nível de detalhe, e o shader de geometria permite manipular primitivas (como pontos e linhas) antes da rasterização. No entanto, estas etapas não serão utilizadas neste trabalho.

Essas etapas do *pipeline* 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 representada 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. Além disso, a CPU também pode enviar variáveis uniformes³ (*uniform variables*) para os *shaders*, que são essenciais para a etapa de renderização e contribuem para a geração da imagem final.

³Variáveis uniformes são parâmetros, como dados de luz e matrizes de transformação, que podem ser passados para um *shader* durante sua execução.

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). Para ilustrar como esse *shader* pode ser implementado, apresentamos um exemplo no [Código 1](#), onde uma matriz é utilizada para transformar a posição dos vértices de entrada (`inPosition`).

Neste exemplo, a posição de saída dos vértices (`gl_Position`) é a posição dos vértices de entrada manipulada por uma função seno baseada no identificador do vértice (`gl_VertexID`), adicionando uma oscilação à posição original. O *shader* também repassa adiante as normais dos vértices (`fragNormal`) para o *shader* de fragmentos. Além disso, a multiplicação pela matriz (`modelViewProjection`) é responsável por aplicar uma projeção de perspectiva.

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

```

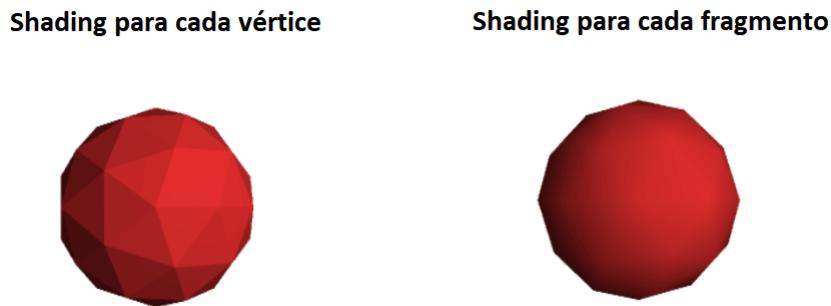
1 #version 330 core
2 layout(location = 0) in vec3 inPosition;
3 layout(location = 1) in vec3 inNormal;
4
5
6 uniform mat4 modelViewProjection;
7
8
9 out vec3 fragNormal;
10
11
12 void main() {
13     vec3 manipulatedPosition = inPosition + (sin(gl_VertexID * 0.1)
14         * 0.1);
15     fragNormal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



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

2.4 Compiladores

2.4.1 Cadeia de Símbolos e Alfabeto

Uma **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](#)).

2.4.3 Compilador como uma 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 ou conceitual. A estrutura semântica de uma linguagem pode ser entendida como o conjunto de regras e convenções que determinam o significado de seus elementos. Para uma linguagem como o conjunto de equações escritas em \LaTeX , a semântica está nas convenções matemáticas, que definem como expressões devem ser interpretadas.

Manter a semântica entre duas linguagens significa garantir que o programa traduzido de L_1 para L_2 preserve tanto os resultados produzidos, quanto as intenções originais. No caso do L^AT_EX, isso envolve traduzir expressões matemáticas para uma representação equivalente na linguagem de destino, respeitando propriedades algébricas, significado de funções e valor de constantes (como \cos e π), entre outros elementos semânticos da matemática.

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 , de modo que essa transformação assegure 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 através de uma 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 $G = (V, \Sigma, R, S)$, conforme apresentado por Michael Sipser (SIPSER, 2012, seção 2.2) em sua obra “Introduction to the Theory of Computation”.

- 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](#)). Os *tokens* gerados nessa etapa correspondem aos terminais das gramáticas livres de contexto.

2.4.6 Análise Sintática ou *Parsing*

A análise sintática, que é a segunda fase do processo de compilação, recebe os *tokens* gerados pela análise léxica, formando o alfabeto da gramática da linguagem. Nessa etapa, o analisador sintático aplica regras de produção gramatical para validar a conformidade do código-fonte com a sintaxe da linguagem, identificando e relatando possíveis erros. Além disso, essas regras são utilizadas para construir uma árvore sintática, ou uma estrutura equivalente, que representa a organização hierárquica das expressões e instruções do programa. Essa árvore sintática atua como uma representação intermediária crucial para a análise semântica e a geração de código.

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 infixo quanto prefixo, por exemplo “–” pode ser um *token* de subtração ou de negação. Assim, cada *token* tem uma função de prefixo e infixo associada.

Nesse 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. Uma demonstração em código desse algoritmo pode ser vista na subseção 5.2.2, onde casos práticos demonstram como o Pratt *parsing* pode ajudar a reduzir o número de regras de produção de uma gramática e simplificar a análise sintática.

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 fundamental no processo de compilação, responsável por garantir a corretude semântica das declarações e instruções do programa. Durante essa fase,

são realizadas verificações, com uso da árvore sintática, para garantir que as operações sejam executadas 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 , gerada pelo analisador sintático, 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 possuem estruturas semanticamente equivalentes a de 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*. Nossa 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.

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

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 3 – Cálculo vetorial em código-fonte L^AT_EX.

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

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 da literatura com o objetivo de explorar trabalhos relacionados ao desenvolvimento de compiladores para tradução de BRDFs expressas em L^AT_EX 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.ez1.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 servem como guia para identificar e selecionar trabalhos científicos capazes de fornecer estratégias para o desenvolvimento do nosso projeto. Essas questões orientam a busca por artigos que apresentem soluções para desafios como: transformar descrições matemáticas de BRDFs em sequências computáveis adequadas para execução por renderizadores; representar modelos de *shading* em estruturas hierárquicas; e desenvolver estratégias de compilação de BRDFs para linguagens de *shading*. A partir desse processo, foram identificados e selecionados os trabalhos que melhor atendem 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 L^AT_EX, para *shaders*?
2. Quais as técnicas de *parsing* têm sido aplicadas no desenvolvimento de compiladores para linguagens matemáticas?
3. O trabalho utiliza gramáticas ou árvores de sintaxe para gerar/representar BRDFs ?
4. De que maneira é possível transformar BRDFs em representações hierárquicas estruturadas?
5. Como operações matemáticas de BRDFs podem ser decompostas em cálculos modulares, permitindo sua implementação em sistemas computacionais?
6. Quais são os principais desafios enfrentados ao traduzir funções matemáticas complexas, como as BRDFs, em *shaders*?
7. 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" OR "shading") AND "BRDF" AND ("compiler" OR "parser" OR "grammar"). As adaptações para cada base e o número de trabalhos encontrados são listados 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 citem a criação de um compilador ou um *parser*;
3. Foram incluídos artigos que utilizam representações hierárquicas para BRDFs, pois contribuem para a compreensão de como decompor BRDFs em estruturas adequadas para o nosso compilador, que é baseado em árvores.
4. Foram incluídos artigos que apresentam linearização de BRDFs em sequências computáveis, relevantes para a geração de código a partir de árvores sintáticas.

3.1.4.2 Critérios de Exclusão

1. Foram excluídos artigos que dispunham de *links* incorretos e ou quebrados;

2. Foram excluídos artigos que não estão relacionados com as questões de pesquisa da subseção 3.1.2;
3. Foram excluídos artigos que não abordam nenhum dos seguintes aspectos: a geração de *shaders* como saída, a estruturação hierárquica de BRDFs ou a decomposição das operações matemáticas associadas a BRDFs;
4. Foram excluídos artigos que não citam BRDFs e compilador ou árvores em seu resumo;
5. Se, após a leitura completa, o artigo não concerne os interesses deste trabalho, esse foi excluído.

Bases	Filtrados
IEEE Xplore Digital Library	1
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 um *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 com a descrição analítica de uma BRDF existente⁶, e interativamente 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*. Esse tipo de função atua como o inverso de uma função de erro, sendo baseada em um *dataset* de materiais previamente medidos. Ela direciona as mutações ao avaliar a qualidade das expressões geradas, permitindo que o *framework* identifique as soluções viáveis após analisar milhares de variantes.

Os autores geraram uma gramática que inclui constantes e operadores matemáticos comuns encontrados em equações de BRDFs. A gramática é compilada, e a árvore sintática 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 de BRDF apresentados no documento incluem exemplares que superam os modelos existentes em termos de precisão e simplicidade.

⁶As BRDFs de entrada são escritas na linguagem definida pela gramática apresentada pelos autores, que permite operações matemáticas, como `sin`, `cos`, `tan`, `exp`, `asin`, `acos`, `dot`, `+`, `sqrt`. Como exemplo, os autores ilustram $brdf_{init} = \rho_s \cos(R \cdot V)^\alpha$, como uma BRDF válida.

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 aprimorar a modularidade e a extensibilidade (HE; FATAHALIAN; FOLEY, 2018). A abordagem de *design* da Slang é baseada em dois princípios fundamentais: garantir compatibilidade com o HLSL existente sempre que possível e introduzir recursos inspirados em linguagens de programação *mainstream*, visando facilitar a familiaridade e a intuição dos desenvolvedores. O compilador desenvolvido neste estudo é capaz de gerar código de *shading* em HLSL, que pode ser compilado por um compilador de HLSL existente.

O autor enfatiza que cada extensão da Slang foi projetada para permitir uma transição gradual a partir do código HLSL já desenvolvido, garantindo a compatibilidade com a maioria das construções sintáticas e semânticas da linguagem HLSL. Isso elimina a necessidade de uma migração completa. Algumas dessas extensões incluem: funções genéricas, estruturas genéricas e tipos que implementam interfaces específicas, semelhantes às interfaces em Java.

Um exemplo do uso de interfaces e funções genéricas em Slang é apresentado no [Código 5](#). Ao declarar a conformidade com a interface IFoo, a definição da estrutura MyType deve incluir um método chamado myMethod, com a assinatura correspondente àquela definida na interface IFoo. Já na função myGenericMethod, o parâmetro arg é genérico, permitindo maior flexibilidade na implementação.

Código 5 – Código de interface em Slang.

```

1 interface IFoo {
2     int myMethod(float arg);
3 }
4
5 struct MyType : IFoo {
6     int myMethod(float arg) {
7         return (int)arg + 1;
8     }
9 }
10
11 int myGenericMethod<T>(T arg) where T : IFoo {
12     return arg.myMethod(1.0);
13 }
```

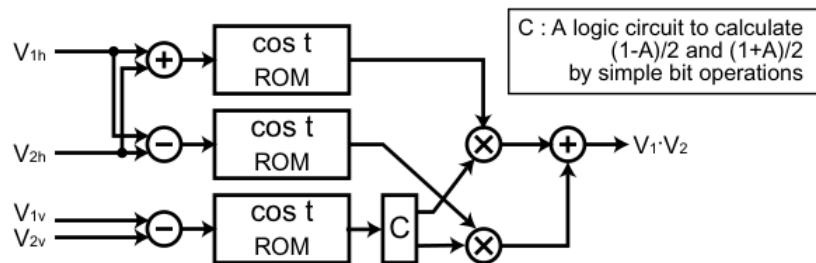
Enquanto o artigo tenta melhorar a flexibilidade e a extensibilidade dos sistemas de *shading* em tempo real, o nosso trabalho se concentra na compilação de equações BRDF em

linguagens de *shading*. Embora ambos os projetos envolvam o uso de *shaders* e a geração de código de *shading*, as abordagens e os focos são distintos

3.1.5.3 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, como o cálculo do produto interno entre dois vetores e a rotação de um vetor por um ângulo. Um exemplo desses diagramas é representado na Figura 9. 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 9 – 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.

Este trabalho foi aceito por incluir o processo de tradução estruturada de BRDFs para uma sequência computável através de circuitos. Assim, o artigo demonstra como modularizar o cálculo das operações matemáticas contidas em BRDFs. A abordagem é útil para o nosso projeto, pois o processo de geração de *shaders* a partir de uma árvore envolve linearizar as operações da BRDF em uma sequência correta, embora o propósito dessa tarefa em nosso projeto seja diferente do apresentado neste trabalho

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 o desenvolvimento do compilador proposto adota uma abordagem prática, estruturada em etapas principais: a análise de informações relevantes sobre BRDFs e a compilação de *shaders*; a exploração de técnicas existentes no domínio; a especificação da linguagem de entrada, um subconjunto de L^AT_EX; a implementação do compilador; e a avaliação de seu desempenho por meio de experimentos de renderização.

O método para realizar a análise e a exploração de técnicas é descrito na [seção 4.1](#). Em seguida, a especificação da linguagem de entrada é apresentada na [seção 4.2](#). É importante destacar que a definição precisa da gramática está consolidada no capítulo de desenvolvimento, na [seção 5.2](#).

Posteriormente, é discutida a elaboração dos casos de teste para validar a correção e a precisão do compilador, conforme detalhado na [seção 4.3](#). Embora a validação escolhida se baseie nesses testes iniciais, os resultados obtidos no [Capítulo 6](#) expandem a validação para um conjunto maior de BRDFs.

O método de implementação do compilador é detalhado na [seção 4.4](#), enquanto a avaliação dos experimentos de renderização, focada na qualidade visual dos *shaders* compilados, é abordada na [seção 4.5](#).

Seguindo essa metodologia, a ferramenta proposta busca compilar descrições de BRDFs em *shaders* GLSL, garantindo qualidade e precisão no resultado final.

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 a análise do código-fonte da ferramenta Disney Explorer ([Figura 10](#)) permitiram a familiarização com o desenvolvimento de BRDFs, fornecendo uma base sólida para compreender o mapeamento de equações para código, aspecto essencial para o desenvolvimento do compilador proposto neste trabalho.

Além disso, foram exploradas algumas técnicas de compilação, o que levou à escolha do método de Pratt *Parsing* para a construção do compilador. Também foram estudados conceitos de recursividade e caminhada em árvores, que foram aplicados na análise semântica e na 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 L^AT_EX, na qual as expressões matemáticas em ambientes `equation` são suficientes para descrever BRDFs. O L^AT_EX é um sistema de composição amplamente utilizado para documentos matemáticos e científicos. O ambiente `equation` é especificamente projetado para exibir equações individuais. O [Código 6](#) é um exemplo de código-fonte L^AT_EX usando o ambiente `equation`.

Código 6 – 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 7](#).

Código 7 – 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 do L^AT_EX oferece uma ampla gama de construções matemáticas, mas, para este projeto, é necessário restringir-se a um subconjunto essencial para representar BRDFs. Ao analisar as principais BRDFs, como as citadas na [seção 4.3](#), identificam-se construções indispensáveis que devem ser reconhecidas e interpretadas pelo compilador para gerar código GLSL, como as apresentadas na [Tabela 4](#).

Descrição	Exemplos	Comando L ^A T _E X
Funções trigonométricas	\tan, \sin, \cos e suas funções inversas	<code>\tan, \sin, \cos\arctan, \arcsin, ...</code>
Função raiz quadrada	$\sqrt{}$	<code>\sqrt</code>
Função exponencial	\exp	<code>\exp</code>
Funções utilitárias	\max, \min	<code>\max, \min</code>
Definições de equações	$f = x$	<code>f = x</code>
Definições de funções	$f(x, y) = x^y$	<code>f(x, y) = x^y</code>
Constantes comuns	π, ϵ	<code>\pi, \epsilon</code>
Constantes de radiometria	θ_i e outras detalhadas na Tabela 7	<code>\theta_i</code>
Indicadores de vetor	\vec{n}	<code>\vec{}</code>
Identificadores aninhados	f_{n_i}	<code>f_{n_i}</code>
Chamadas de funções	$f(x + y)$	<code>f(x + y)</code>
Produto vetorial	$x \times y$	<code>x \times y</code>
Soma e Subtração	$x + y, x - y$	<code>x + y, x - y</code>
Negação	$-y$	<code>-y</code>
Multiplicação	$x \cdot y$	<code>x \cdot y</code>
Frações	$\frac{x}{y}$	<code>\frac{x}{y}</code>
Divisão	x/y	<code>x / y</code>
Potenciação	x^y	<code>x^y</code>

Tabela 4: Tabela de especificação da linguagem.

A descrição completa dos símbolos reconhecidos em nível de código está na seção dedicada ao *lexer* ([seção 5.1](#)), enquanto a gramática completa reconhecida pelo compilador é apresentada na seção sobre o *parser* ([seção 5.2](#)).

Embora o *lexer* e o *parser* identifiquem os símbolos, o compilador também precisa atribuir significado a eles durante a análise semântica, que ocorre após o *parsing*. Por exemplo, ω_o representa o ângulo de saída da luz, enquanto f refere-se à função BRDF. Todas as convenções de símbolos suportadas pela linguagem estão detalhadas na [Tabela 5](#), juntamente com seus significados.

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 L^AT_EX 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

Símbolo	Comando L ^A T _E X	Descrição
θ_i	\theta_i	Ângulo de elevação da direção da luz incidente
θ_o	\theta_o	Ângulo de elevação da direção da luz refletida
ϕ_i	\phi_i	Ângulo azimutal da direção da luz incidente
ϕ_o	\phi_o	Ângulo azimutal da direção da luz refletida
ω_i	\omega_i	Direção da luz incidente
ω_o	\omega_o	Direção da luz refletida
f	f	BRDF de referência
\vec{n}	\vec{n}	Vetor normal à superfície
\vec{h}	\vec{h}	Vetor do meio entre ω_o e ω_i
θ_h	\theta_h	Ângulo entre \vec{n} e \vec{h}
θ_d	\theta_d	Ângulo entre ω_i e \vec{h}

Tabela 5: Tabela de símbolos e suas descrições

Código 8), 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 e F não tenham sido definidas explicitamente, é importante ressaltar que, caso essas funções sejam definidas na equação L^AT_EX, elas também devem aparecer no Código 9, o GLSL esperado de saída. Vale ressaltar que, nesta seção de metodologia, apresentamos uma versão simplificada do processo de *design* dos casos de teste para facilitar o entendimento. Na prática, tanto unidades, como ρ_d , quanto funções, como D , G e F , devem estar devidamente definidas. Casos de teste completos e detalhados estão disponíveis no Capítulo 6.

Além disso, variáveis como a normal \vec{n} são frequentemente fornecidas como entrada para o *shader* de fragmentos ou declaradas como variáveis uniformes. Por isso, elas não estão explicitamente definidas na função cook_torrance do Código 9, sendo tratadas como variáveis implícitas. Uma lista completa dessas variáveis pode ser encontrada no mapeamento de convenções para código GLSL na Tabela 7.

Inicialmente, os casos de teste priorizam a avaliação da geração de operações e precedências. Contudo, vale destacar que o compilador desenvolvido produz código GLSL com a definição completa da função BRDF, incluindo todas as variáveis de convenções necessárias. Esse código permite que a BRDF calculada seja utilizada para determinar a cor final e encaminhada às etapas subsequentes do *pipeline* gráfico, possibilitando sua renderização na ferramenta Disney BRDF Explorer

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

Código 8 – Entrada em L^AT_EX (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 9 – 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 de equações L^AT_EX que descrevem BRDFs em código de *shader* GLSL. As tarefas incluem: criar um *lexer* e *parser* para aceitar equações L^AT_EX; 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 e o padrão de projeto 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 Odin¹, 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, não foi utilizada nenhuma biblioteca externa, apenas as bibliotecas padrão básicas incluídas na instalação da linguagem.

Para a análise e construção da estrutura do compilador, foram adotadas técnicas de análise recursiva, com destaque para o método Pratt Parsing. Inicialmente, o lexer e o parser foram implementados para suportar o subconjunto da linguagem L^AT_EX descrito em seção 4.2. O objetivo inicial foi garantir que os fundamentos do compilador estivessem funcionais, com precedências devidamente testadas na geração da árvore sintática abstrata (AST).

A travessia e manipulação da AST é possível pelo pacote *walker*, que abstrai operações sobre a árvore. Este pacote é utilizado em três etapas principais: adicionar parênteses para

¹Mais informações sobre a linguagem Odin estão disponíveis no site oficial <<https://odin-lang.org>>, e o código-fonte do seu compilador pode ser acessado no GitHub em <<https://github.com/odin-lang/Odin>>.

explicitar a ordem das operações, facilitando a validação; inferir recursivamente os tipos de cada expressão (nós que representam valores); e realizar a travessia da árvore com discriminação dos tipos de nós para a geração de código.

Além disso, foi implementada uma etapa de análise semântica por meio do pacote chamado **checker**. Essa etapa consiste em validar todas as expressões, criar os escopos e a tabela de símbolos, além de anotar a AST com informações como os tipos inferidos de cada nó, incluindo funções com seus domínios e contradomínios, vetores e suas dimensões, e números reais.

Por fim, com a AST devidamente anotada, foi desenvolvido o pacote **emitter**, o qual realiza a geração do código GLSL que atenda aos requisitos do pipeline gráfico. Esse código é formatado para ser carregado diretamente e renderizado na ferramenta Disney descrita na (seção 4.5).

4.5 Experimentos de Renderização

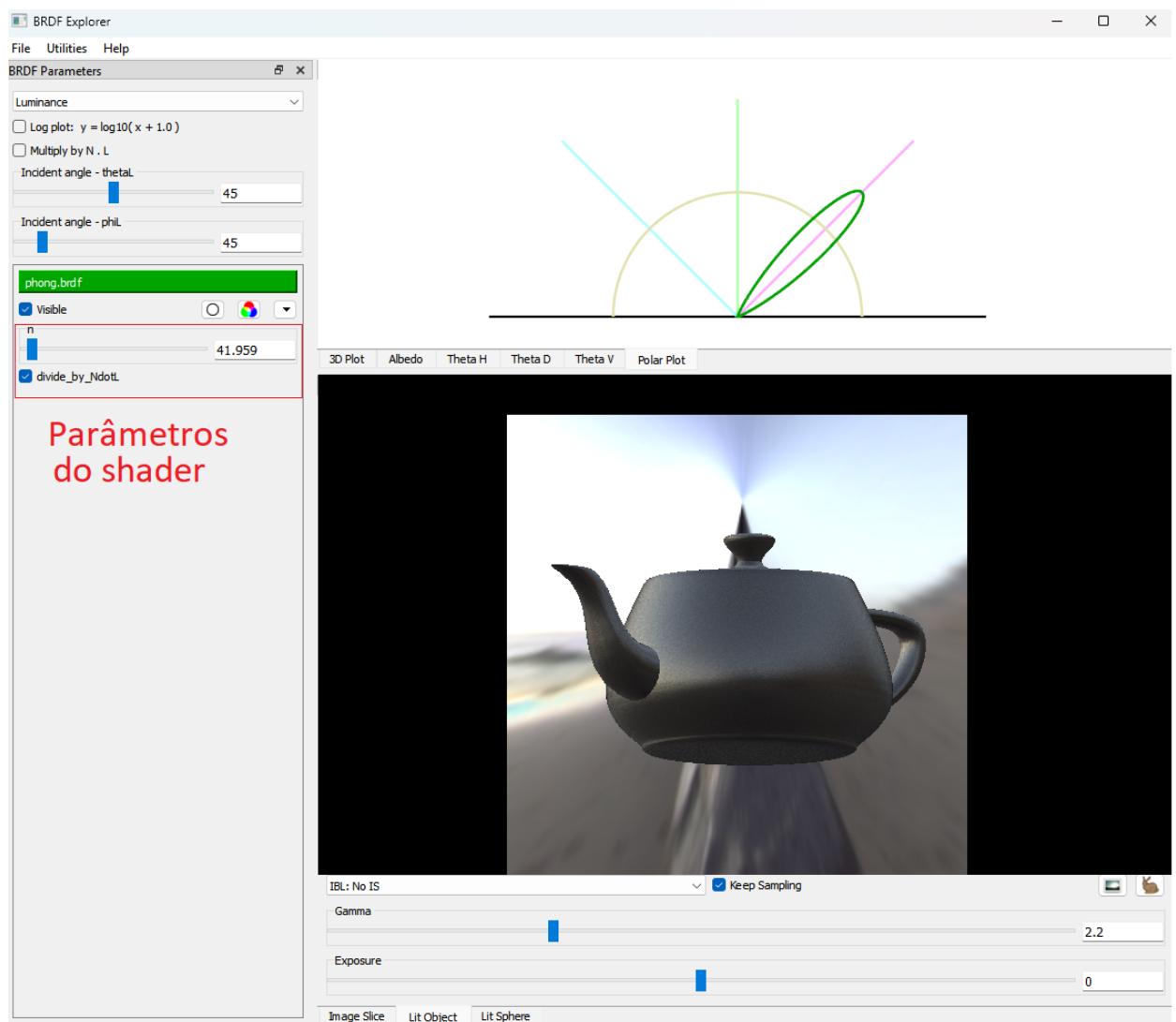
Experimentos de renderização são realizados usando os *shaders* gerados pelo compilador. Isso permite a avaliação da qualidade visual das imagens renderizadas produzidas pelos *shaders* compilados. A plataforma escolhida para os testes é a ferramenta Disney BRDF Explorer².

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 10.

O código que informa à ferramenta qual a BRDF a ser renderizada e seus possíveis parâmetros pode ser visto na Código 10. 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. Esses parâmetros se tornam elementos de interface, permitindo o ajuste dinâmico das constantes. O nosso compilador gera *shaders* nesse formato.

²<<https://github.com/wdas/brdf>>

Figura 10 – Ferramenta de visualização de BRDFs da Disney.



Fonte: autor.

Código 10 – O código GLSL com sintaxe extra para definir parâmetros.

```
1 analytic
2
3 # variables go here...
4 # only floats supported right now.
5 # [type] [name] [min val] [max val] [default val]
6
7 ::begin parameters
8 float n 1 1000 100
9 bool divide_by_NdotL 1
10 ::end parameters
11
12 # Then comes the shader. This should be GLSL code
13 # that defines a function called BRDF (although you can
14 # add whatever else you want too, like sqr() below).
15
16 ::begin shader
17
18 vec3 reflect(vec3 I, vec3 N)
19 {
20     return 2*dot(I,N)*N - I;
21 }
22
23 float sqr( float x )
24 {
25     return x*x;
26 }
27
28
29 // Phong BRDF
30 vec3 BRDF( vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y )
31 {
32     vec3 R = reflect(L,N);
33
34     // specular
35     float val = pow(max(0, dot(R,V)),n);
36     if (divide_by_NdotL)
37         val = val / dot(N,L);
38     return vec3(val);
39 }
40 ::end shader
```

5

Desenvolvimento

Este capítulo detalha o desenvolvimento do compilador escrito na linguagem Odin, que torna possível a transformação de equações em documentos L^AT_EX em código GLSL. Cada etapa do processo é encapsulada em um pacote distinto, estruturado em diretórios conforme a [Figura 12](#). O `lexer` corresponde à tokenização da linguagem, responsável por converter o texto em *tokens* identificáveis. O `parser` realiza a análise sintática, construindo a estrutura gramatical do documento. O `walker` contém funções essenciais para visualização da árvore de sintaxe abstrata (AST) e checagem de tipos feita pelo `checker`, executando a travessia da árvore de forma ordenada para geração de código feita pelo pacote `emitter`. A arquitetura completa do compilador pode ser vista na [Figura 11](#).

No módulo `lexer`, explicado na [seção 5.1](#), foi implementada uma análise léxica manual inspirada em máquina de estados para tokenização do documento L^AT_EX. Este processo converte a entrada textual em uma sequência de *tokens*, preparando o as estruturas de dados para as próximas etapas de processamento.

Na [seção 5.2](#), é discutido sobre o pacote `parser`, que utiliza gramática livre de contexto e a técnica de *Pratt Parsing* para construir a árvore de sintaxe abstrata (AST). Essa abordagem possibilita uma representação hierárquica precisa das expressões matemáticas de BRDFs, capturando as nuances sintáticas e estruturais do documento original. A especificação da linguagem, apresentada no [Código 22](#) (parte 1) e no [Código 23](#) (parte 2), é definida na seção de análise sintática, juntamente com a precedência dos operadores prefixos e infixos.

O componente `walker`, discutido na [seção 5.3](#), desempenha funções essenciais para a navegação e análise da AST. Suas funcionalidades incluem tanto a visualização da estrutura gerada quanto a preparação para verificações subsequentes por meio do pacote `checker`. O papel principal do `walker` é realizar a travessia dessa árvore de forma genérica, oferecendo suporte para decidir se a travessia deve continuar ou se é necessário retornar de um nó antes de alcançar os nós-folha. Além disso, o componente abstrai a forma de percorrer nós filhos de

Figura 11 – Estrutura de geral da arquitetura do compilador.

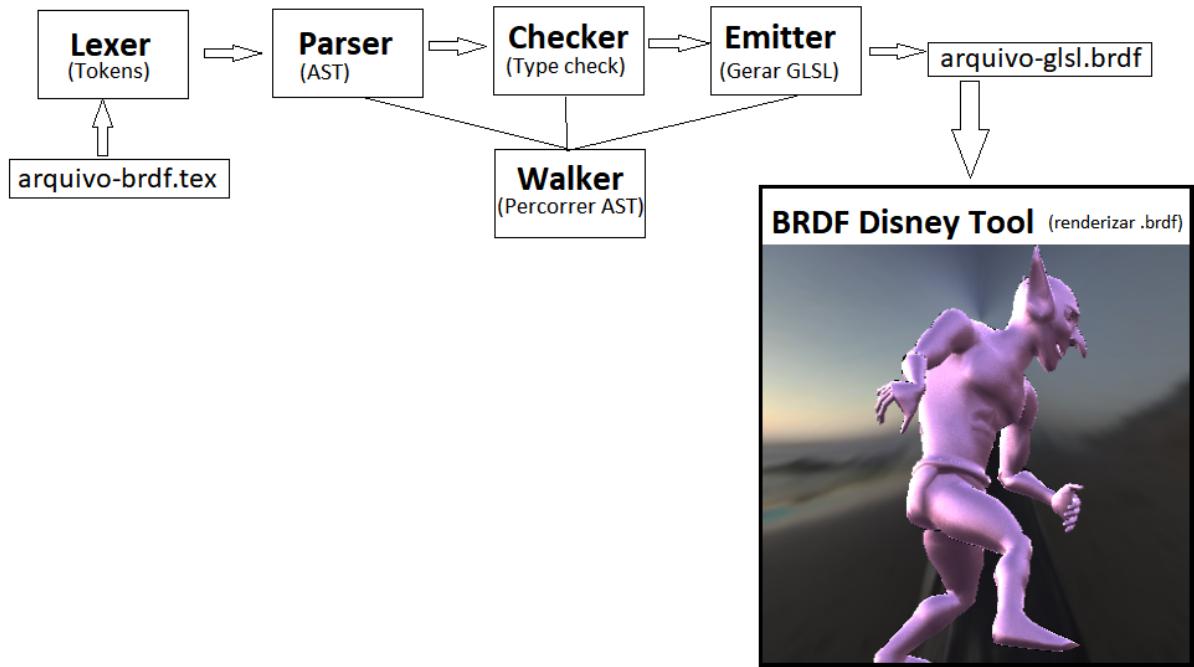
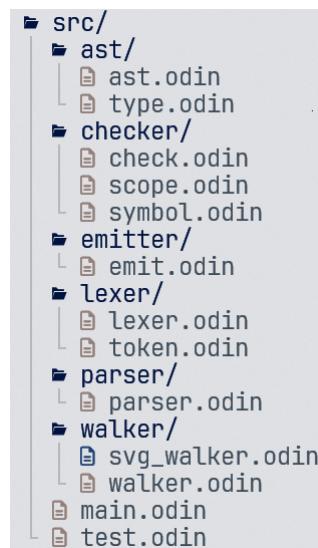


Figura 12 – Estrutura de pacotes do compilador.



maneira uniforme, independentemente do tipo do nó pai.

No módulo **checker** (seção 5.4), foram implementadas as inferências de tipos e validações semânticas. Este componente garante a consistência das expressões matemáticas antes da geração de código, com o auxílio da tabela de símbolos (subseção 5.4.2), eliminando potenciais erros de modelagem. A saída dessa etapa deve estar apta para emitir código GLSL, com os símbolos

organizados na ordem correta.

Por fim, o `emitter` utiliza a AST validada, a tabela de símbolos e os escopos para gerar código GLSL, transformando as expressões matemáticas de BRDFs contidas na AST em um *shader* com toda implementação necessária para ser carregado na ferramenta de visualização Disney BRDF Explorer. Os resultados detalhados e os experimentos com a aplicação do compilador em BRDFs usadas na literatura podem ser consultados no [Capítulo 6](#), onde é demonstrada a eficácia da ferramenta na tradução de diversos modelos de BRDFs. Esses experimentos também serviram como guia para a verificação da corretude da gramática durante seu desenvolvimento.

5.1 Análise Léxica (`lexer`)

Nesta etapa, é realizado o processo de tokenização de um subconjunto dos símbolos possíveis no ambiente de equação do `LATEX`, conforme comentado na [seção 4.2](#). A entrada desse processo são caracteres do arquivo fonte, enquanto a saída é uma sequência lógica desses caracteres, organizada em *tokens*. O código responsável por essa funcionalidade está contido no pacote `lexer`.

O processo de análise léxica realiza uma varredura completa no arquivo de entrada, caractere por caractere, para identificar e extrair os *tokens*. Antes de iniciar essa extração, verificamos se o trecho analisado pertence a um ambiente de equação. Essa verificação é feita ao identificar a `string1 \begin{equation}`, que marca o início da extração de *tokens*. Da mesma forma, a delimitação do ambiente se encerra com a string `\end{equation}`. Isso permite que o sistema ignore partes do arquivo que não pertencem ao ambiente de equação, como textos explicativos ou outros elementos presentes no mesmo arquivo `.tex`. Dessa forma, garantimos que a tokenização seja restrita às secções relevantes do código.

Na lista de expressões regulares ([Código 15](#)), definimos os tipos de *tokens*, onde o lado esquerdo do símbolo “=” corresponde ao tipo de *token*, e o lado direito descreve sua expressão regular. Palavras em letras maiúsculas representam categorias de caracteres, como `DIGIT`, que denota qualquer dígito de 0 a 9, e `LETTER`, que cobre letras de ‘a’ a ‘z’. Já palavras entre aspas simples correspondem a sequências literais de caracteres.

Além disso, utilizamos os seguintes símbolos na notação: “*” indica zero ou mais ocorrências do caractere especificado; “|” representa alternativas para a geração do mesmo tipo de *token*; e “;” marca o fim da definição do tipo de *token*.

O pacote inteiro de tokenização pode ser acessado por meio de uma única função, descrita no [Código 11](#), escrita na linguagem `Odin`. Essa função, chamada `lex`, aceita uma lista de caracteres como entrada e retorna uma lista de estruturas do tipo `Token` (detalhado no [Código 13](#)).

¹ `string` é uma cadeia de caracteres

A estrutura `Token` possui três campos principais:

- `kind`: identifica o tipo de *token*, mapeando-o para uma dos tipos definidos no [Código 15](#).
- `text`: contém a *string* correspondente ao *token* gerado.
- `position`: uma instância do tipo `Position`, que registra a posição exata do *token* no arquivo de origem.

Código 11 – Função principal do Lexer.

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

Durante a iteração sobre o `input`, o processo de tokenização mantém algumas variáveis de controle para monitorar o estado do fluxo de caracteres. Quebras de linha são contadas ao encontrar sequências como "`\n`" ou "`\n\r`". É mantida a coluna atual que rastreia a posição horizontal do caractere em uma linha. O cursor é o índice que aponta para o caractere atualmente em processamento. Essas informações são usadas para preencher o campo `position` de cada *token*. A estrutura `Position`, detalhada no [Código 13](#), é essencial para garantir a precisão na geração de relatórios e rastreamento de erros.

5.1.1 Reporte de Erros

O sistema informação de erros, implementado nesta etapa, é utilizado por todos os pacotes do projeto. Essa funcionalidade assegura que erros sejam associados a posições específicas no arquivo de entrada, facilitando a depuração e correção. A assinatura da função de tratamento de erros, bem como suas possíveis variações, está documentada no [Código 12](#).

Código 12 – 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);
```

Dada uma posição ou um `token`, é exibida uma mensagem (`msg`) diretamente no terminal, formatada para destacar visualmente o erro em vermelho. A formatação utiliza as informações do `token`, como o nome do arquivo, a linha, a coluna e o comprimento do `token` problemático, permitindo sublinhar precisamente onde o erro ocorreu. Isso proporciona maior clareza às mensagens de erro, como exemplificado no caso de erro semântico devido ao uso de identificadores não definidos ([Figura 14](#)).

Optou-se por exibir os erros de forma uniforme para ilustrar como o compilador os reporta, independentemente de serem léxicos, sintáticos ou semânticos. Isso oferece uma visão panorâmica dos erros, sem sobrecarregar as seções de análise sintática e semântica com detalhes adicionais de erros. Dessa forma, o entendimento das tarefas do compilador é facilitado nas etapas subsequentes. A seguir, são apresentados alguns tipos de erros possíveis:

1. **Erros léxicos:** uso de palavras reservadas ([Figura 17](#)).
2. **Erros sintáticos:** problemas de estrutura, como balanceamento incorreto de parênteses ([Figura 16](#)) e *tokens* que não formam uma expressão matemática válida ([Figura 18](#)).
3. **Erros semânticos:** envolvendo tipos incompatíveis ([Figura 15](#)), símbolos não definidos ([Figura 14](#)) e redefinição de símbolos ([Figura 19](#)).

Figura 13 – Erro ao usar símbolo não definido.

```
$ .bin/unnamed 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}' inside equation `f'
    7 | \begin{equation}
    8 |     f = 2^{\exp(\text{var})}
    |
    9 | \end{equation}
[excyber @ ~/code/tcc-lang]
```

Figura 14 – Erro ao tentar símbolo não definido.

```
$ .bin/unnamed 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}' inside equation `f'
    7 | \begin{equation}
    8 |     f = 2^{\exp(\text{var})}
    |
    9 | \end{equation}
[excyber @ ~/code/tcc-lang]
```

Figura 15 – Erro de tipos incompatíveis.

```
[excyber @ ~/code/tcc-lang]
$ .bin/unnamed 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 argument, but we got `R^3`.

    7 | \begin{equation}
    8 |     f = \exp(\vec{1,1,1})
        ^^^^
    9 | \end{equation}
[excyber @ ~/code/tcc-lang]
```

Figura 16 – Erro de balanceamento de parênteses.

```
[excyber @ ~/code/tcc-lang]
$ .bin/unnamed 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 17 – Erro de uso incorreto de palavras reservadas.

```
[excyber @ ~/code/tcc-lang]
$ .bin/unnamed 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 the \
he ` in front, as in `\'arccos`'

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

Figura 18 – Erro de *token* incapaz de produzir expressão.

```
$ .bin/unnamed 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 make a expresion got `\'end`

    8 | \begin{equation}
    9 |     f = ----- \end
        ^^^^
   10 | \end{equation}
```

Figura 19 – Erro de redefinição de símbolo.

```
test/should-error/redefinition.tex(12:7) error: Redefinition of symbol `f'
11 | \begin{equation}
12 |     f = 2
13 | \end{equation}
```

5.1.2 Classificação e Extração de Tokens

Tokens simples, como aqueles compostos por um ou dois caracteres, são extraídos lendo-se o número correspondente de caracteres do `input`. Após a leitura, o `token` é construído e o laço continua para o próximo. Quando um caractere `%` é encontrado, os caracteres subsequentes são ignorados até a próxima quebra de linha. Isso adiciona suporte a comentários no estilo `LATEX`.

Tokens mais complexos, como números, identificadores ou *tokens* especiais, são extraídos com base em suas características. Números podem opcionalmente conter um ponto decimal, como em `1.0`. Já identificadores consistem em uma ou mais letras, opcionalmente prefixadas pelo símbolo `\`.

A lista de expressões regulares de *tokens* fornecida é intrinsecamente ambígua. Por exemplo, uma sequência como `\frac` pode ser interpretada como um identificador comum ou como `token_frac`. Para resolver essa ambiguidade, criamos um dicionário que mapeia identificadores específicos para *tokens* especiais (Código 14). Assim, se um identificador começar com o caractere `\`, ele será verificado no dicionário e classificado como um *token* especial, se apropriado.

Identificadores não permitem números nem mesmo o caractere de sublinhado `(_)`. Isso ocorre porque, no analisador sintático, um nó do tipo identificador é modelado como um tipo recursivo, permitindo que identificadores sejam aninhados ao conter outros nós. Dessa forma, não é necessário permitir sublinhados diretamente no nível de token, o que possibilita a escrita de identificadores mais complexos, como `\pi{n_1}` (renderizado em `LATEX` como π_{n_1}). Nesse caso, `\pi` seria o primeiro *token* do nó identificador, e sua subexpressão seria `n_1` (renderizado como n_1), que, por sua vez, é o identificador `n` com a subexpressão `1`.

Adicionalmente, permitimos o uso da palavra-chave `\text`, como em `\text{id}`, para descrever identificadores. Essa palavra-chave é utilizada para incluir texto dentro do ambiente de equações em `LATEX`, oferecendo uma visualização mais clara de funções longas. Por exemplo, em vez de renderizar como `normalize(x)`, pode-se usar `\text{normalize}(x)`, que será renderizado como `normalize(x)`, tornando a expressão mais legível. Outro caso útil é o *token* `len`, que pode ser visualmente confundido com a multiplicação entre as variáveis `l`, `e` e `n`; usando `\text{len}`, fica mais claro que “`len`” é um único token, e não como uma multiplicação entre 3 outros *tokens*. A extração desse *token* segue um processo semelhante ao do suporte para

\vec{id}.

A enumeração que representa os tipos de *tokens* pode ser consultada no [Código 16](#). Cada entrada dessa enumeração corresponde diretamente aos tipos de *tokens* gerados pela lista de expressões regulares apresentada no [Código 15](#). Para facilitar a leitura, incluímos, à direita de cada entrada, o símbolo correspondente, indicado em comentários

Com a etapa de tokenização concluída, é possível avançar para a análise sintática, onde é explorada a forma como os *tokens* gerados são organizados em estruturas hierárquicas que formam a árvore sintática.

Código 13 – Estruturas do Lexer.

```
1 Token :: struct {
2     kind: Token_Kind,
3     val: union{i64,f64},
4     text: string,
5     pos: Position,
6 }
7
8 Position :: struct {
9     file: string,
10    offset: i64,    // starting at 0, buffer offeset in file
11    line: i64,      // starting at 1, starting
12    column: i64,   // starting at 1
13    length: int    // how much chars foward
14 }
15 }
```

Código 14 – Mapa de identificadores 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 }

```

Código 15 – Gramática 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        = '\sim';
token_underline   = '_'; --- Used for subexpressions
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 16 – 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, (parser)

O *parser* para a linguagem subconjunto do ambiente `equation` do L^AT_EX foi desenvolvido utilizando o método de Pratt *Parsing* na linguagem Odin. Neste contexto, esse subconjunto é denominado como `EquationLang`, que abrange todas as partes essenciais para a definição de BRDFs descritas na [seção 4.2](#), e sua gramática é documentada.

A implementação deste *parser* adota o método de descida recursiva, no qual cada regra de produção da gramática possui uma função de análise correspondente. Este método foi escolhido por priorizar a simplicidade do código e clareza da gramática. A entrada do *parser* consiste nos *tokens* gerados na etapa de análise léxica.

5.2.1 Sintaxe de Gramáticas

Na definição de gramáticas deste capítulo, utilizamos uma notação simplificada de sintaxe para representar as regras. Palavras compostas por letras minúsculas representam *não-terminais*, enquanto palavras entre aspas simples correspondem a *terminais*.

Além disso, os seguintes operadores são utilizados na notação:

- “*” indica zero ou mais ocorrências de um elemento, que pode ser tanto um *terminal* quanto um *não-terminal*;
- “()” define agrupamentos para aplicar um operador sobre os elementos agrupados;
- “|” introduz uma regra alternativa para o mesmo *não-terminal*. Caso esteja dentro de um agrupamento, como em “(a|b)”, significa que aceita *a* ou *b*;
- “=” representa uma produção.

Adicionalmente, sequências de três hífens (“---”) são usadas para comentários destinados ao leitor, sem impacto na definição gramatical.

No caso da gramática da `EquationLang`, apresentada na [subseção 5.2.3](#), todas as palavras em letras minúsculas que começam com `token_` são tratadas como *terminais*. Esses terminais correspondem diretamente aos *tokens* gerados pelo `lexer` e estão presentes na lista de expressões regulares descrita na [seção 5.1](#).

5.2.2 Parser

Diferentemente dos *parsers* tradicionais de descida recursiva, que frequentemente utilizam múltiplas chamadas de função indiretamente recursivas para tratar diferentes níveis de precedência, o *parser* aqui implementado organiza as funções de análise com base na tabela de precedência dos operadores. Essa abordagem é detalhada no [Código 18](#), que descreve a lógica central para o *parsing* de expressões.

O Pratt Parsing simplifica a análise sintática de expressões ao tratar precedência e associatividade de forma dinâmica, sem inflar a gramática. Na abordagem tradicional, as precedências são fixadas por meio de várias regras de produção, como demonstrado neste exemplo de expressões binárias no [Código 17](#). Regras intermediárias são adicionadas à gramática com o principal propósito de embutir a precedência na estrutura gramatical.

Código 17 – Regras tradicionais de precedência por gramática.

```

expr      = or_expr;
or_expr   = or_expr '||' and_expr | and_expr;

and_expr  = and_expr '&&' eq_expr| eq_expr;

--- Igualdade
eq_expr   = eq_expr '==' rel_expr | rel_expr;

--- Comparação
rel_expr  = rel_expr '<' add_expr | add_expr;

add_expr  = add_expr '+' mul_expr | mul_expr;

mul_expr  = mul_expr '*' primary_expr | primary_expr;

primary_expr = '(' expr ')' | NUMBER | IDENTIFIER;

```

Isso torna a gramática mais longa, exigindo mais regras que derivam outras antes de chegar a um terminal. Na descida recursiva, as regras atuam como funções, o que significa que métodos recursivos tradicionais com muitos níveis de precedência resultam em um maior número de chamadas recursivas, aumentando a complexidade da implementação e a rigidez na modificação da gramática. Em contraste, no Pratt Parsing, uma única regra é suficiente, juntamente com uma tabela de precedência: `expr = expr ('||' | '&&' | '==' | '<' | '+' | '*') expr;`.

O *parser* desenvolvido para EquantionLang consulta à tabela de precedência dinamicamente, conforme definida na [Tabela 6](#), para decidir a ordem de avaliação com base no *token* operador encontrado.

Foi utilizada uma notação semelhante à original de Pratt ([PRATT, 1973](#)), onde as funções `parse_null_denotations` e `parse_left_denotations` desempenham papéis equivalentes às funções `token.prefixo` e `token.infixo`, respectivamente, como demonstrado no [Algoritmo 1](#).

Além disso, a abordagem de descida recursiva (descendente) permite que cada regra de produção definida na gramática ([Código 22](#)) seja diretamente mapeada para um procedimento em código. Isso pode ser observado, por exemplo, na função de análise do nó `Start` da AST ([Código 20](#)), que reflete diretamente as regras de produção `start`, `decl` e `decl_equation_begin_end_block` especificadas na gramática apresentada no [Código 22](#).

Do ponto de vista da interface oferecida pelo pacote `parser`, todo o processo de análise sintática é abstraído em uma única chamada de função ([Código 19](#)). A função principal, `parse`, trabalha em conjunto com a estrutura `Parser` para realizar a análise.

Código 18 – 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
8             higher precedence
9         . than previous precedence then stay in the loop, effectively
10            creating a left leaning
11         . sub-tree, else, we recurse to create a right leaning sub-tree.
12     */
13     for precedence(peek()) > prec_prev + associativity(peek()) {
14         /* expressions that needs a left operand such as postfix,
15            mixfix, and infix operator */
16         left = parse_left_denotations(left)
17     }
18     return left
19 }
```

Código 19 – Estrutura e função principal do pacote `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 }
```

Código 20 – 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 }
```

5.2.3 Gramática

Para formalizar a gramática da linguagem de entrada (EquationLang), suas regras estão detalhadas nos [Código 22](#) e [Código 23](#). Todas as definições de equações ou funções que compõem a descrição de uma BRDF devem estar dentro do seu próprio bloco de ambiente `equation` em um arquivo `LATEX`. Um exemplo de código-fonte contendo três equações válidas nessa linguagem se encontra no [Código 21](#), e sua renderização correspondente em `LATEX` é ilustrada na [Equação 5.1](#).

No exemplo, a primeira equação (ρ_d) é gerada pela regra `expr_vector_literal`. A segunda (ρ_s) é uma expressão binária derivada da regra `expr_infix`. Já a última equação utiliza múltiplas regras, como `expr_grouped` e `expr_prefix`.

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

$$\rho_s = 0.0, \vec{0.2}, 1.0 * 20 \quad (5.1b)$$

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

Código 21 – Exemplo código escrito na linguagem EquationLang.

```

1 \begin{equation}
2     \rho_d = \vec{0.3,0.3,0.3}
3 \end{equation}
4
5 \begin{equation}
6     \rho_s = \vec{0.0,0.2,1.0}*20
7 \end{equation}
8
9 \begin{equation}
10 f = \frac{\rho_d}{\pi} + \frac{\rho_s}{8*\pi} *
11 \frac{(\vec{n}\cdot\vec{h})}{(\vec{\omega_o}\cdot\vec{h})} *
12 \max((\vec{n}\cdot\vec{\omega_i}), \\
13 (\vec{n}\cdot\vec{\omega_o})) \\
14 \end{equation}

```

Código 22 – 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
    --- Ressalto que 'function_call' e 'function_definition' tem a mesma
        construção.
    --- Apenas diferenciamos pela posição que aparecer, se à esquerda ou à
        direita de '=' da regra 'field'.
    --- por exemplo 'a = f(1)', 'f(1)' é uma chamada de função
    --- Já 'f(x) = 1' é uma definição de função
    | expr_function_call | expr_function_definition
    | token_opencurly expr token_closecurly
;

expr_identifier =
    --- Ex: '\text{id}'
    token_text token_opencurly expr_identifier token_closecurly
    --- Ex: '\vec{id}'
    token_vec token_opencurly expr_identifier token_closecurly
    --- Ex: 'id_n'
    | expr_identifier token_underline expr_identifier
    --- Ex: 'id_2'
    | expr_identifier token_underline token_number
    --- Ex: 'id_{n+1}'
    | expr_identifier token_underline token_opencurly expr token_closecurly
    --- Token especiais como \phi ou \alpha
    | token_identifier | token_omega | token_theta | token_phi
    | token_rho | token_alpha | token_beta | token_sigma
    | token_pi | token_epsilon | token_max | token_min
;

```

Código 23 – Gramática para EquantionLang 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_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 22 e Código 23), adotamos a notação sintática previamente estabelecida na [subseção 5.2.1](#). A gramática apresentada inclui regras para expressões, atribuições, agrupamento, literais numéricos e vetoriais, chamadas e definições de funções, além de operadores, como `expr_prefix` e `expr_infix`. Ela é projetada para suportar a sintaxe necessária à definição de BRDFs em L^AT_EX.

A tabela de operadores ([Tabela 6](#)), usada no Pratt Parsing, é implementada pela função `precedence_from_token`, que associa cada *token* a um valor inteiro indicando sua precedência: valores maiores indicam maior prioridade. Alguns *tokens* assumem diferentes funções conforme o contexto e podem alterar sua precedência. Por exemplo, C pode ser um prefixo em expressões

agrupadas, $(2 * 3)$, ou um infixo em chamadas de função, $f(x)$. De forma análoga, $-$ pode atuar como operador de negação (prefixo) ou como operador de subtração (infixo).

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
\times	Não	20
$*$	Não	20
$/$	Não	20
$^$	Não	30
$!$	Não	400

Tabela 6: Tabela de Precedência dos Tokens

5.2.3.1 Estrutura da Árvore de Sintaxe

Esta seção apresenta os tipos de nós que compõem a AST do compilador da linguagem `EquationLang`. Cada estrutura de nó captura diferentes regras de produção da sintaxe em nível de código. O nó `Expr`, o mais genérico, inclui o campo `ty_inferred`, que armazena o tipo inferido da expressão, preenchido durante a análise semântica e utilizado na geração de código.

- **Node**: estrutura base para todos os nós da AST. Representa um tipo genérico e é estendido por outros tipos mais específicos.
- **Expr**: representa expressões de forma geral, como literais, operadores, ou agrupamentos.
- **Decl**: representa declarações genéricas na linguagem, como definições de funções ou equações.
- **Start**: o nó raiz da AST, contendo todas as declarações e marcando o fim do arquivo.
- **Decl_Equation**: representa uma equação, conectando uma expressão ou valor a um identificador.
- **Field**: representa uma atribuição, associando um nome a um valor através do operador `=`.
- **Expr_Identifier**: representa identificadores, como variáveis ou constantes, podendo incluir subexpressões e anotações adicionais.

- **Expr_Number**: representa literais numéricos, como inteiros ou números de ponto flutuante.
- **Expr_Vector_Literal**: representa vetores literais, contendo uma sequência de valores numéricos.
- **Expr_Grouped**: representa expressões agrupadas por delimitadores, como parênteses ou chaves.
- **Expr_Prefix**: representa expressões prefixas, onde o operador precede o valor, como -3 .
- **Expr_Infix**: representa expressões infixas, onde o operador está entre dois valores, como $3 + 3$.
- **Expr_Function_Call**: representa chamadas de função, contendo o nome da função e seus argumentos.
- **Expr_Function_Definition**: representa definições de funções, incluindo o nome, parâmetros e o corpo da função.

Anteriormente, na [subseção 5.1.2](#), foi discutido que o *parser* suporta identificadores aninhados, como x_{i_1} (`x_{i_1}`). Esse suporte é demonstrado no [Código 24](#), que processa identificadores de forma recursiva. O trecho faz parte de uma função maior onde um `switch`² é aplicado sobre a enumeração descrita na [Código 16](#). Dentro desse `switch`, há um `case` específico que reconhece *tokens* de identificadores ou símbolos especiais, como $\omega, \theta, \phi, \rho, \alpha, \beta, \sigma, \pi, \epsilon$.

A recursividade, realizada por meio da função `parse_expr`, permite incluir subíndices numéricos, subexpressões de identificadores ou expressões mais complexas, como $n + 1$ em f_{n+1} . Isso amplia a flexibilidade na representação de equações de BRDFs, onde o uso de subíndices numéricos é bastante comum.

Essa mesma lógica exemplifica o processamento de todas as expressões recursivas, como operações binárias. Para identificar o *token* atual, utilizamos a função `peek()`, que permite visualizar um ou dois *tokens* à frente e decidir qual nó da AST será construído. Após identificar o *token*, calcula-se a variável `prec`, que determina a precedência do operador. Com base nessa precedência, são realizadas chamadas recursivas à função `parse_expr` para processar expressões aninhadas.

Uma vez preenchidos todos os campos necessários, a expressão completa é retornada. Esse processo se repete até que toda a subárvore de expressões de uma equação seja construída. A análise sintática é concluída quando todas as equações são adicionadas à AST.

Posteriormente, essa estrutura hierárquica é anotada com tipos e validada pelo pacote `checker`, conforme descrito na [seção 5.4](#). Antes da validação, é necessário implementar métodos para a travessia da árvore. Esse processo é detalhado na [seção 5.3](#), que apresenta técnicas para

²Os comandos `switch` e `case` na linguagem Odin são semelhantes aos da linguagem C.

Código 24 – 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_IDENTIFIER[1:])
31            }
32            node.sub_expression = sub_node
33        }
34    }
35 }
```

percorrer a árvore, acessar seus dados e facilitar as etapas subsequentes de análise semântica e geração de código.

5.3 Implementação do Padrão de Visitante (walker)

Nessa etapa, é desenvolvido o pacote `walker` para auxiliar em quatro tarefas-chave: inferência de tipos das expressões, validação da precedência da AST gerada pelo *parser*, visualização gráfica da AST, e geração de código. Para atingir esses objetivos, o padrão de projeto *visitor*³ é usado, o qual facilita a travessia e manipulação da AST. A estrutura principal do pacote

³<<https://refactoring.guru/pt-br/design-patterns/visitor>>

consiste em duas peças fundamentais: a estrutura `Visitor` e a função `walk`.

5.3.1 Estrutura `Visitor`

A estrutura `Visitor` ([Código 25](#)) encapsula uma função de visita polimórfica (`visit`) que pode ser invocada em cada nó da AST. A função `visit` é definida nessa estrutura para realizar operações de transformação de nós, como: modificar o campo `ty_inferred` do nó do tipo `Expr`; e remover ou adicionar nós.

A estrutura também permite que o visitante mantenha um estado interno (`data`), que pode ser alterado dinamicamente durante a travessia. Um exemplo de uso desse estado é o acompanhamento da profundidade atual para gerar um arquivo SVG da árvore ([subseção 5.3.4](#)), ou a manutenção de uma lista de identificadores usados para a resolução de símbolos pelo pacote `checker`.

Código 25 – Estrutura polimórfica `Visitor`. O `DataType` é o parâmetro concreto dessa estrutura.

```

1 // Estrutura polimórfica, aceita um tipo qualquer, chamado de
2 // DataType, como estrada para criar um tipo concreto.
3 Visitor :: struct (DataType: typeid) {
4     visit: proc(visitor: ^Visitor(DataType), node: ^ast.Node) ->
5         ^Visitor(DataType),
6     data: DataType,
7 }
```

5.3.2 Função `walk`

A função `walk` ([Código 26](#)) realiza a travessia em profundidade (*depth-first*) da AST usando a estrutura genérica de visita. Ela percorre todos os nós, como declarações, expressões, equações e definições de funções, aplicando a função `visit` antes e depois de explorar cada subárvore. Isso permite criar *visitors* personalizados para tarefas como:

- Checagem de tipos: verifica a consistência dos tipos nos nós.
- Parentização de expressões: usa os nós para testar a precedência correta de operadores.
- Geração de gráficos: cria uma representação visual da AST em formato SVG.
- Geração de código: converte a AST para a linguagem GLSL.

O controle de parada em `walk` é feito por duas verificações principais:

1. Verificação de nulidade: a função checa se o visitante (`v`) ou o nó (`node`) são nulos antes de prosseguir, evitando operações em dados inexistentes.

2. Interrupção de travessia: após cada chamada à função `visit`, se o retorno for nulo, a travessia é interrompida, permitindo ao `visitor` decidir dinamicamente se deseja continuar ou parar; adaptando-se ao cenário de uso.

5.3.3 Validação de Precedência

As funções e a estrutura de travessia foram utilizadas para validar a precedência dos operadores na AST gerada pelo parser. Uma função de parentização foi implementada para percorrer a AST e inserir parênteses, preservando a precedência original implícita. Assim, é possível testar se a representação textual reflete corretamente a ordem de avaliação das operações matemáticas e assegurar que a hierarquia das operações foram corretamente representadas na árvore gerada.

Os testes de precedência comparam o texto original de uma expressão com sua versão esperada, onde os parênteses indicam a ordem correta de avaliação ([Código 27](#)). Casos complexos, como operadores associativos à direita (por exemplo, exponenciação) combinados com operadores de diferentes precedências, bem como expressões aninhadas e funções, também são testados para abranger variados cenários possíveis.

Código 27 – Validação de precedência por parentização de expressões.

```

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 );

```

À medida que o compilador foi desenvolvido, esses testes foram úteis para evitar regressões. Sempre que uma nova funcionalidade era adicionada, os testes garantiam que as funcionalidades existentes não fossem comprometidas.

5.3.4 Visualização da AST por Imagem

Para validação visual, foi implementada uma função que gera uma imagem da AST no formato SVG. Cada nó da AST é representado por um retângulo com textos que fornecem informações como o tipo de operador, o tipo do nó e o identificador, quando aplicável. Anteriormente, era usada a função `print_ast`, que imprimia os nós e seus atributos com indentação correspondente à profundidade. No entanto, essa abordagem se tornou limitada conforme a complexidade da AST aumentou, exigindo uma solução mais robusta para depuração.

Na [Figura 20](#), é ilustrada a imagem gerada para a equação [Equação 5.2](#). Observa-se que o nó da operação binária `+` (`Expr_Infix Plus`), localizado mais próximo da raiz, é avaliado por último, enquanto os nós mais próximos das folhas, como `*` e `^` (`Expr_Infix Caret`), têm maior precedência e são resolvidos primeiro. Além disso, o SVG inclui informações adicionais, como o tipo das expressões. Por exemplo, o identificador `f` é anotado como `R`, determinado na validação semântica (`checker`).

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

Os nós da AST possuem diferentes formas de acesso aos seus filhos, pois os campos podem ter nomes ou posições variadas conforme o tipo do nó. Para lidar com essa heterogeneidade, o pacote `walker` oferece a função `children` ([Código 28](#)), que abstrai essas diferenças e retorna uma lista uniforme de filhos para qualquer nó.

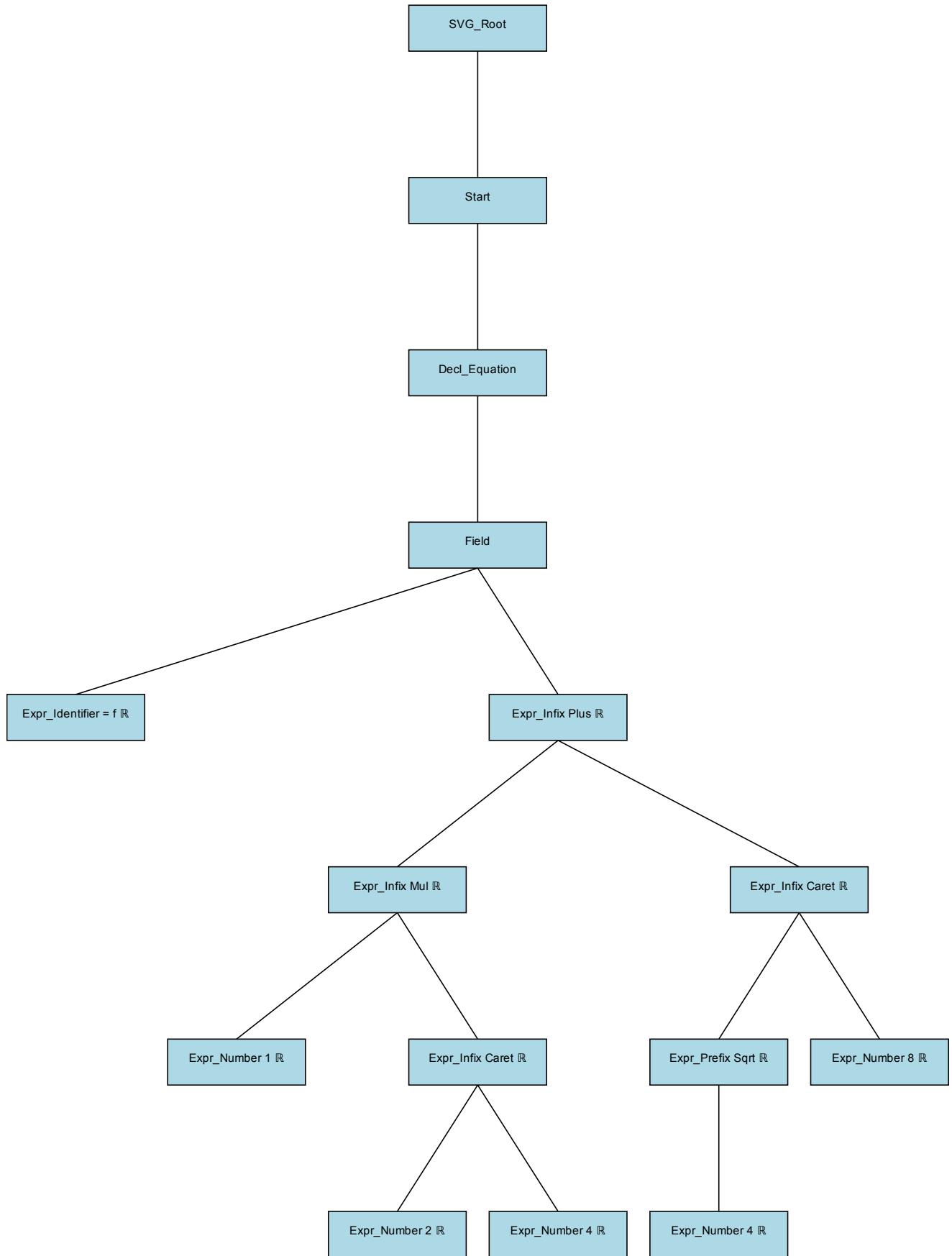
Código 28 – Assinatura da função que extrai nós filhos de maneira uniforme para qualquer tipo de nó.

```

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

```

Figura 20 – SVG da AST gerado para Equação 5.2.



5.4 Análise Semântica (checker)

O processo de validação semântica no compilador é crucial para garantir a corretude das equações, a consistência de tipos em operações e a resolução adequada de símbolos. Ele é estruturado em dois mecanismos principais: validação de definições de funções e validação de declarações. Esses mecanismos trabalham de forma integrada para garantir que a estrutura do programa esteja em conformidade com as regras semânticas da EquationLang.

Nesta seção, abordamos essas regras e discutimos como são validadas expressões envolvendo vetores, redefinições de equações e possíveis erros nas definições de BRDFs que inviabilizem a geração de código. A conclusão bem-sucedida dessa etapa de inferência e validação indica que o programa está apto a prosseguir para a fase de geração de código GLSL, conduzida pelo pacote `emitter`.

5.4.1 Funções do Pacote checker

O pacote `checker` é responsável por realizar diversas tarefas fundamentais para a validação semântica do compilador. Essas tarefas são descritas detalhadamente a seguir:

- Inferência e Anotação de Tipos:** Cada expressão na AST possui um campo `ty_inferred` que deve ser preenchido durante essa etapa. Para determinar esses tipos, é necessário realizar a inferência de tipos. Essa tarefa é detalhada na [subseção 5.4.5](#). Os tipos possíveis incluem números (`float`), vetores (`vector`) e funções. O tipo de uma função é representada por um domínio (tipos dos argumentos de entrada) e um contradomínio (tipo do valor de retorno). Exemplos incluem:

- Uma função $f(a, b) = a \cdot b \cdot 1, \vec{1}, 1$, que recebe dois valores reais e retorna um vetor tridimensional, possui a assinatura:

$$f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^3.$$

- O produto vetorial, que combina dois vetores tridimensionais, possui a assinatura:

$$\times : \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3.$$

Essas assinaturas são coletadas durante o processo de inferência para garantir a consistência em expressões de chamada de funções.

- Compatibilidade de Tipos:** O `checker` também verifica a compatibilidade entre tipos em diferentes contextos do programa. Essa validação assegura que:

- Não sejam realizadas operações não definidas por EquationLang, como a divisão entre dois vetores.

- Não sejam aplicados operadores em tipos incompatíveis, como elevar um número a um vetor ($2^{\vec{n}}$).
- Os argumentos de uma função devem pertencer ao domínio esperado pela função. Por exemplo, se a função `normalize(\vec{u})` retorna um vetor tridimensional (\mathbb{R}^3), utilizá-lo como argumento para a função seno, que espera um número escalar (\mathbb{R}), seria inválido. Assim, `sin(normalize(\vec{u}))` não é permitido.
- O tipo do valor de retorno de uma função deve ser compatível com o contexto em que é utilizado.

3. Validação de Definições de Símbolos: Outro papel fundamental do checker é garantir que todos os identificadores, abstraídos em símbolos na [subseção 5.4.2](#), utilizados no programa, estejam devidamente definidos antes de serem usados. Essa etapa inclui:

- Identificar declarações ausentes.
- Identificar dependência circular.
- Certificar-se de que funções obrigatórias, como a BRDF f , estejam presentes.

Essas validações são realizadas utilizando as funções do pacote `walker`, e os erros encontrados são reportados usando as funções de erro introduzidas na [seção 5.1](#).

5.4.2 Tipos, Símbolos e Escopos

Cada expressão na AST possui um tipo associado, modelado como uma união de estruturas na linguagem Odin. Essa união permite representar as seguintes categorias semânticas: tipos primitivos fundamentais, como números e vetores; assinaturas de funções, que capturam o domínio e o contradomínio. Por exemplo, o produto vetorial possui a assinatura $\mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$. Já o vetor normal (\vec{n}), possui tipo primitivo de número (\mathbb{R}). A modelagem clara de tipos e o conceito de escopos e símbolos é essencial para validar expressões em diferentes contextos, verificar compatibilidade de operações e garantir a consistência semântica do programa.

5.4.2.1 Tipos

No [Código 29](#), a representação dos tipos segue uma modelagem hierárquica, onde o **tipo base** (Type) contém metadados comuns, como a referência ao nó na árvore sintática e o identificador do tipo concreto. Os **tipos derivados**, que são subdivididos em categorias específicas, incluem: *tipo básico* (Type_Basic), que representa um tipo primitiva como um número; *tipo vetorial* (Type_Vector), que é caracterizado pela sua dimensionalidade e pelo tipo de elemento que o compõe; e *tipo função* (Type_Function), que define os parâmetros e os resultados da função.

Código 29 – Estruturas que representam os tipos de expressões da AST.

```
Type :: struct {
    node:      ^Node,
    size:      i64,
    derived:   Any_Type, // Ou Type_Vector, ou Type_Basic, ou
                  Type_Function.
    id:        typeid,
}

Type_Vector :: struct {
    using _:      Type,
    element_type: ^Type,
    dimensions:  int,
}

Type_Basic :: struct {
    using _: Type,
    basic_kind : Basic_Kind,
};

Type_Function :: struct {
    using _: Type,
    params, results :[]^Type,
};
```

5.4.2.2 Símbolos

Os símbolos representam entidades nomeadas em EquationLang. A estrutura `Symbol`, apresentada no [Código 30](#), encapsula as informações semânticas necessárias sobre os identificadores para validação da AST e geração de código. Essas informações incluem:

- **Escopo:** o escopo em que o símbolo foi definido.
- **Nó do identificador:** referência ao nó correspondente na AST para uso futuro.
- **Definição de função (opcional):** nó da definição de função, quando aplicável.
- **Estado de resolução:** indica se o símbolo já foi resolvido.
- **Tipo associado:** o tipo inferido ou declarado do símbolo.

O gerenciamento de símbolos é essencial para garantir que o estado de cada símbolo seja mantido durante a resolução, como discutido na [subseção 5.4.4](#). Os possíveis estados são listados a seguir:

1. **Não Resolvido:** Estado inicial
2. **Em Progresso:** Resolução em andamento
3. **Resolvido:** Completamente processado

Código 30 – Estrutura do Símbolo.

```
// An Symbol is a named entity in the language
Symbol :: struct {
    scope:      ^Scope,
    identifier: ^ast.Expr_Identifier, // Can be nullptr
    fn_defn:    ^ast.Expr_Function_Definition, // if the Symbol is a
                                                function

    state:      Symbol_State,
    flags:      bit_set[Symbol_Flag; u64],
    type:       ^Type,
    value:      Maybe(Value)
};
```

5.4.3 Escopo e Tabela de Símbolos

Nesta etapa, foi criada uma tabela de símbolos para análise semântica e geração de código GLSL. 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 31](#), a estrutura Scope representa um mapeamento de nomes para objetos de símbolo dentro de um **único escopo**, e a estrutura Scope_Table mantém uma **pilha de escopos**, permitindo aninhamento.

Código 31 – Código da estrutura de símbolos escrito em Odin.

```
1 Scope_Table :: [dynamic]^Scope
2
3 Scope :: struct {
4 /*
5 . ‘node’ Is a parent node that created that scope
6 . Ex: a block, a function block, a struct or namespace
7 . If null, then the scope is the global
8 */
9     parent:      ^Scope,
10    children:    [dynamic]^Scope,
11    elements:   map[string]^Symbol,
12    ordered_keys: []string, // bit of a HACK, yeah
13};
```

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_get`: recupera um símbolo da tabela de símbolos pelo seu identificador.
- `scope_add`: adiciona um novo símbolo ao escopo atual.

Essa tabela de símbolos armazena todas as informações necessárias para a fase de geração do *shader* GLSL.

5.4.4 Resolução de Símbolos

A resolução de símbolos é uma etapa fundamental no `checker`, garantindo que cada símbolo seja corretamente definido e tipado antes de seu uso. Esse processo é especialmente relevante em situações onde a ordem de definição não segue um fluxo linear, como no exemplo da [Equação 5.3](#)).

Um escopo global é criado contendo todos os símbolos definidos, e a validação verifica se cada identificador está presente no escopo atual. Caso contrário, é gerado um erro apropriado. É nesta etapa que verificamos se a função f , a BRDF por convenção, existe no escopo global. Identificadores embutidos, definidos pelas convenções deste trabalho como ω_i e θ_d , são adicionados automaticamente à tabela de símbolos juntamente com seus tipos. A lista completa de embutidos está disponível em [Código 35](#). Símbolos de parâmetros são resolvidos no escopo da função correspondente.

Nesse caso da [Equação 5.3](#), b é atribuído a a antes que a seja definido. O `checker` deve resolver essa dependência, analisando a antes de b para inferir corretamente o tipo de b . Isso é possível por conta da construção de um grafo de dependências entre símbolos (apresentado no [Código 32](#)). A partir de uma ordenação topológica desse grafo, o sistema determina uma ordem de avaliação válida, além de identificar dependências circulares que possam impedir a compilação.

$$b = a \tag{5.3a}$$

$$a = \vec{n} \tag{5.3b}$$

$$f = b \tag{5.3c}$$

Essa ordenação permite usar símbolos antes de suas equações serem declaradas, desde que definidos em alguma equação posterior. A resolução mantém em memória uma lista com a ordem correta de avaliação das declarações, o que é essencial para a geração de código GLSL, onde referências a variáveis antes de suas definições são proibidas.

Para implementar essas funcionalidades, o checker realiza múltiplas passadas na AST: duas são para resolução de símbolos e a última é para validação das equações, na seguinte ordem:

1. Coleta de Símbolos:

- Coletar todos identificadores nas declarações
- Registrar símbolos nos escopos apropriados
- Inicializar estruturas de rastreamento de dependências.

2. Análise de Dependências:

- Construir o grafo de dependências. Estruturas podem ser vistas no [Código 32](#), um simples grafo que associa um símbolo a outros símbolos dos quais depende
- Validar referências de símbolos, incluindo detectar uso de símbolos que nunca foram definidos
- Estabelecer a ordem de avaliação por meio de ordenação topológica
- Verificação do ponto de entrada

3. Validação Final:

- Inferência e verificação de tipos
- Validação de expressões
- Validação de definição de funções com uso de escopo

Código 32 – Estrutura de grafo de dependências.

```

1 Symbol_Dependency :: struct {
2     dependencies: [dynamic]^Symbol,
3 }
4 Symbol_Graph :: map[^Symbol] Symbol_Dependency

```

Código 33 – Entrada para o compilador que gera dependência circular.

```

\begin{equation}
    a = f
\end{equation}

\begin{equation}
    f = a
\end{equation}

```

Figura 21 – Erro reportado sobre dependencia circular.

```
test/should-error/cicle.tex(9:4) error: Circular dependency detected a
  8 | \begin{equation}
  9 |   a = f
  10| \end{equation}
```

5.4.5 Inferência de Tipos

A função `infer_type` determina o tipo de uma expressão (`expr`) na AST durante a análise semântica, aplicando regras baseadas em operações matemáticas, como multiplicação entre número real e vetor, produto vetorial entre vetores e assinaturas de funções.

Projetada para processar todas as expressões de `EquationLang` — como identificadores, operadores, chamadas de função e literais —, a função verifica inicialmente se o tipo já foi inferido (`expr.ty_inferred`) para evitar processamento redundante. Caso contrário, realiza a inferência com base em um `switch` que é capaz de avaliar todos tipos de expressões. Um trecho relevante desse processo está no [Código 34](#), que ilustra a discriminação e validação dos tipos.

Para **identificadores** (`Expr_Identifier`), verifica se o identificador corresponde a um vetor especial, como ω_i ou \vec{n} . Nesse caso, o tipo é inferido como \mathbb{R}^3 (vetor tridimensional). Identificadores específicos, como `\pi` ou `\epsilon`, são atribuídos ao tipo numérico real (\mathbb{R}). Para outros identificadores, a função consulta o escopo atual para determinar o tipo, usando as estruturas detalhado na [subseção 5.4.3](#).

Para **operações prefixadas** (`Expr_Prefix`), como raiz quadrada (`\sqrt`) ou funções trigonométricas (`\sin`, `\cos`), a função valida se o operando é numérico e atribui o tipo correspondente à expressão. Nas **operações binárias** (`Expr_Infix`), a função realiza a inferência dos tipos dos operandos esquerdo e direito. Se os tipos não forem compatíveis, aplica regras específicas. Alguma dessas regras são:

- A multiplicação de um número por um vetor ($2 * \vec{n}$) ou a divisão de um vetor por um número ($\frac{\vec{u}}{\sqrt{u \cdot u}}$) resultam no tipo vetor (\mathbb{R}^3).
- Operações entre dois números resultam em um número.
- Operações entre dois vetores resultam em um número ou um vetor a depender se a operação foi um produto vetorial ou produto interno.

Outros casos incluem **literais**, como números (`Expr_Number`) e vetores literais (`Expr_Vector_Literal`), cujos tipos são atribuídos diretamente como \mathbb{R} (números reais) e \mathbb{R}^3 (vetores tridimensionais), respectivamente.

Para chamadas de função (`Expr_Function_Call`), o tipo da expressão é determinado pelo tipo do primeiro valor retornado pela função. A validação dos argumentos é realizada em outra função, detalhada na [subseção 5.4.7.1](#).

5.4.6 Validação de Equações

As declarações de equações são validadas após a coleta e ordenação descritas na [subseção 5.4.4](#), portanto assume-se que o lado esquerdo das equações seja um identificador válido ou uma definição de função e que estão na ordem correta de avaliação.

Qualquer violação semântica, como incompatibilidades de tipos ou uso de escalares onde vetores são esperados, é reportada ao usuário, com detalhes sobre o contexto e a localização do erro, conforme descrito em [subseção 5.1.1](#).

A função `check_expr` realiza uma travessia semelhante à inferência de tipos e chama `infer_type`. Nela, a análise de expressões que não foram abordadas por `infer_type` é realizada para todos os tipos de expressões. Algumas dessas validações estão listadas após este parágrafo, e um trecho dessa travessia pode ser visto no [Código 36](#).

- `Expr_Function_Call`: Verifica se estamos fazendo a chamada corretamente com um identificador, evitando casos como tentar chamar uma função com um número, por exemplo, `123(x, y)`, o que é incorreto. Também checa se o número de argumentos corresponde ao número de parâmetros esperados.
- `Expr_Prefix`: Verifica operadores `(-, +)` e funções como `sqrt(x)` e `sin(x)`. Certifica-se de que os tipos sejam compatíveis, gerando erro em caso contrário, como quando um vetor é passado para a função seno.
- Literais de Vetor `Expr_Vector_Literal`: Garante que os vetores tenham exatamente 3 dimensões⁴.

⁴Exigir vetores de apenas 3 dimensões pode ser considerado uma limitação semântica imposta pelo compilador

Código 35 – Identificadores embutidos pela convenção deste trabalho.

```
BUILTIN_IDENTIFIERSS :: []string {
    '\pi',
    '\epsilon',
    '\theta{h}',
    '\vec{n}',
    '\vec{h}',
    '\vec{\omega{i}}',
    '\theta{i}',
    '\phi{i}',
    '\vec{\omega{o}}',
    '\theta{o}',
    '\phi{o}',
    '\theta{h}',
    '\theta{d}',
}
```

Código 36 – Recorte da função `check_expr`.

```

1  check_expr :: proc(expr: ^ast.Expr) {
2      // Primeiro inferimos o tipo
3      infer_type(expr)
4      // Código omitido de preambulo
5      case ^Expr_Identifier:
6          // Check for using undefined indetifiers
7          identifier_key := key_from_identifier(e)
8          if !is_defined(e, false) {
9              error(e.identifier, "Identifier '%v' is not defined in
10                 the current scope.", identifier_key)
11         }
12     case ^Expr_Function_Call:
13         check_expr(e.left)
14         fn_ident, fn_ident_ok :=
15             e.left.expr_derived.(^ast.Expr_Identifier)
16         if !fn_ident_ok {
17             error(e.open, "Tried to call an expression that is not an
18                 identifier")
19         }
20         fn_string := key_from_identifier(fn_ident)
21         fn_sym, fn_sym_ok := scope_get(fn_string)
22         fn_type, fn_type_ok :=
23             e.left.ty_inferred.derived.(^ast.Type_Function)
24         if !fn_type_ok {
25             error(e.open, "Tried to call '%v', which is not a
26                 function. Its type is '%v'", fn_string,
27                 format_type(e.left.ty_inferred))
28         }
29         if len(e.exprs) != len(fn_type.params) {
30             error(
31                 e.open, "Args number mismatch. '%v' function expects
32                     '%v' arguments but '%v' were given.",
33                 fn_string, len(fn_type.params), len(e.exprs)
34             )
35         }
36     // Outros casos omitidos
37   }
38 }
```

5.4.7 Validação de Funções

A validação semântica de definições e chamadas de funções segue um processo semelhante ao da análise de expressões, mas com o uso da pilha de escopos. Nas definições de funções, as variáveis no corpo da função são validadas para assegurar que todos os símbolos usados realmente estejam definidos no escopo da função. Já nas chamadas de funções, os argumentos fornecidos são comparados aos parâmetros esperados para validação.

5.4.7.1 Definição de Funções

O procedimento `check_function_definition` valida definições de funções, garantindo segurança de tipos e consistência através das seguintes tarefas:

1. Inferir os tipos dos parâmetros.
2. Inferir o tipo de retorno baseado na expressão final.
3. Construir o tipo completo da função, incluindo parâmetros e retorno.
4. Garantir que os identificadores usados estejam consistentes com seus tipos declarados.
5. Validar as expressões no corpo da função.
6. Criar um novo escopo para os parâmetros da função.

O processo começa com o processamento dos parâmetros e o gerenciamento do escopo, como mostrado no [Código 38](#). Quando uma função é definida, um novo escopo é criado para armazenar informações dos parâmetros e da função, que são adicionadas ao escopo correspondente.

Esse escopo é essencial para validar chamadas de função e expressões no corpo. Cada função tem seu próprio escopo, com o escopo pai sendo o global, evitando conflitos de identificadores.

O controle de visibilidade funciona como esperado; se um parâmetro x existe, ele é preferido ao x global, resultando no fenômeno de *shadowing* ou sombreamento, como exemplificado na [Equação 5.4](#), onde o valor de f é 3, não 2.

$$x = 2 \tag{5.4}$$

$$g(x) = x \tag{5.5}$$

$$f = g(3) \tag{5.6}$$

Durante a validação dos parâmetros, cada identificador passa por um processo de inferência de tipo. Parâmetros marcados explicitamente com o prefixo `\vec` recebem o tipo padrão \mathbb{R}^3 (vetor tridimensional), enquanto os demais são tratados como número real (\mathbb{R}).

Se um parâmetro \vec{x} é declarado como vetor, todas as operações envolvendo x no corpo da função devem respeitar as operações vetoriais; caso contrário, um erro será reportado.

5.4.7.2 Chamada de Funções

Chamadas de função passam por uma validação semelhante: os argumentos têm seus tipos inferidos e são comparados com a assinatura da função chamada (`Type_Function`).

No exemplo do [Código 37](#), a função g possui a assinatura $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. A expressão resultante da chamada de função terá o tipo do contradomínio da função. Também é necessário confirmar que g refere-se a um símbolo do tipo função.

Código 37 – Equação com uso incorreto de tipos na chamada de função.

```
\begin{equation}
    g(a, x) = a*x*x
\end{equation}

\begin{equation}
    f = g(1, \vec{1,1,1})
\end{equation}
```

A função g espera dois números reais como argumentos, mas na equação f , um vetor foi passado no lugar de um número, o que gera um erro. A Figura 22 ilustra esse erro, indicando a função com o argumento incompatível. Além disso, ao passar os argumentos, também validamos se a quantidade de argumentos corresponde ao número de parâmetros esperados.

Figura 22 – Erro gerado por uso incorreto de tipos na chamada de função.

```
test/should-error/redefinition.tex(12:8) error: Type mismatch in 2nd argument in function `g'. Expected `R', but got `R^3'
11 | \begin{equation}
12 |     f = g(1, \vec{1,1,1})
13 | \end{equation}
```

Código 38 – Validação de parâmetros de uma função.

```
// ...
parameter_types := [dynamic]^Type{}
scope_enter(fn_sym.scope) {
    for &parameter in fn.parameters {
        parameter_key := key_from_identifier(parameter)
        ty := infer_type(parameter, true, ty_number)
        parameter_sym.type = ty
        append(&parameter_types, ty)
    }
}
// ...
```

5.5 Geração de Código (`emitter`)

A etapa de geração de código é realizada pelo pacote `emitter`, que cria um *shader* compatível com a ferramenta Disney BRDF Explorer. Essa etapa é separada em três fases, descritas na subseção 5.5.4. O processo começa com o escopo global, que contém informações essenciais, como símbolos, assinaturas de funções, tipos e nós da AST completamente anotada.

A travessia recursiva dos símbolos e nós ocorre sem validações adicionais, uma vez que a etapa semântica já foi realizada.

A geração de código abrange o lado esquerdo da equação, que inclui identificadores e parâmetros de funções, e o lado direito, que é sempre uma expressão, cuja emissão é explorada em [subseção 5.5.3](#). Além disso, é fundamental garantir que cada variável tenha um nome único no código GLSL, conforme descrito em [subseção 5.5.2](#).

5.5.1 Emissão de Equações

A emissão de código começa pelo lado esquerdo da equação. Existem três aspectos principais nesse processo:

1. Mapeamento de Tipos: o tipo associado ao identificador é a primeira informação a ser resolvida. Caso seja um número real, ele é mapeado para `float` em GLSL. Para vetores (\mathbb{R}^3), o mapeamento é feito para `vec3`. Em definições de funções, a assinatura é construída com base nos tipos dos parâmetros e do retorno.
2. Identificador: o identificador associado ao símbolo é traduzido conforme regras específicas que garantem unicidade e conformidade com as restrições do GLSL. Este processo é detalhado em [subseção 5.5.2](#).
3. Expressões: no lado direito da equação, tanto funções quanto variáveis são expressões. Para gerar isso, realiza-se uma travessia da AST, mapeando nós de expressões (como somas e chamadas de funções trigonométricas) para seu equivalente em GLSL. Detalhes adicionais sobre a emissão de código para expressões podem ser encontrados na [subseção 5.5.3](#).

5.5.2 Unicidade de Variáveis

Para garantir unicidade e compatibilidade dos identificadores gerados no GLSL, é necessário lidar com algumas restrições sintáticas e evitar colisões entre identificadores.

1. **Restrição de Caracteres:** O GLSL não permite caracteres especiais, como { e }, em seus identificadores. Por exemplo, uma equação em `EquationLang f_{1} = 2` não pode ser diretamente transformada no identificador `f_{1}`. Para resolver isso, todos os caracteres inválidos são substituídos por sublinhados (_), considerando subexpressões, resultando em identificadores como `f__1_`.
2. **Prevenção de Colisões:** Mesmo após a substituição de caracteres, ainda podem ocorrer colisões. Para resolver isso, mapeia-se cada símbolo de identificador para um número inteiro único de 64 bits (ID). Esse ID é então concatenado ao começo do identificador com o prefixo `var`, resultando em `var1_f__1_`; uma cadeia de caracteres única para cada símbolo.

3. Remoção de Sequências Reservadas: O GLSL não aceita⁵ identificadores que contêm duas ou mais ocorrências consecutivas de `_`. Também é necessário remover sublinhados ao final, resultando em `var1_f_1`. Após a geração inicial, identificadores que contenham essas sequências são corrigidos para atender às essas restrições.

Essas etapas garantem que cada variável no código GLSL seja única e válida. Apesar da limitação de $2^{64} - 1$ identificadores distintos imposta pelo uso de inteiros de 64 bits, esse valor é mais do que suficiente para o propósito de criação de BRDFs. Caso necessário, a utilização de inteiros de tamanho arbitrário implementado por *software* pode ser aplicada para ampliar esse limite.

5.5.3 Geração de Expressões

A geração de expressões no pacote `emitter` converte as expressões do lado direito das equações em código GLSL válido, usando a AST anotada com tipos inferidos.

A função principal, `emit_expr`, processa cada nó de expressão e constrói a cadeira de caracteres correspondente em GLSL usando um `StringBuilder` da biblioteca padrão de Odin, evitando concatenações excessivas. Na travessia recursiva da AST, realizada pelo pacote `walker`, converte operações binárias, prefixas, vetoriais e chamadas de função em código *shading*.

Um trecho dessa função está no [Código 41](#), que demonstra como os tipos são discriminados (`case`) e `strings` para funções trigonométricas são geradas usando `sbprint`. A seguir, são enumeradas as principais categorias de expressões tratadas:

1. Expressões Prefixas (Expr_Prefix): Essas expressões incluem operações unárias e funções matemáticas básicas. A implementação:

- Emite funções trigonométricas utilizando as funções fornecidas pelo GLSL automaticamente, como `sin`, `cos` e `tan`.
- Lida com operadores unários, como negação `(-)` e raiz quadrada `(sqrt)`. Nesses casos é aplicar escrever o operador com parentesis ao redor. Por exemplo, `\sqrt{-2}` ($\sqrt{-2}$) emite `(sqrt(-(2)))`.
- Constrói vetores usando o construtor provido por OpenGL `vec3()`.
- Mantém a precedência de operadores adicionando parênteses apropriados.

2. Expressões Binárias (Expr_Infix): Operações binárias representam a maior parte das expressões matemáticas.

- Operações aritméticas básicas `(+, -, *, /)`: basta adicionar parênteses com o operador no meio e emitir recursivamente à esquerda e à direita.

⁵Um erro comum ocorre ao utilizar caracteres consecutivos de `_` em identificadores, resultando no erro C7528: OpenGL reserves names containing `__`.

- Produto interno: usa-se a função `dot` fornecida pelo GLSL.
- Multiplicação escalar-vetor é suportada pelo GLSL, portanto, fica semelhante ao original: `\vec{n} \cdot 2` é equivalente a `(vec3(n) * 2)`.
- Suporta o produto vetorial (`cross`). Nesse caso, o operador vem primeiro, no lugar de infixo: `\vec{n_1} \times \vec{n_2}` é equivalente a `cross(vec3(var_1_n_1), vec3(var_1_n_2))`.
- Processa exponenciação (`x^y`) chamando a função `pow()` do GLSL.

3. Expressões Literais e Agrupamentos:

- Constrói vetores literais utilizando `vec3`.
- Adiciona parênteses em expressões agrupadas.

4. Expressões de Chamadas de Função: Na geração de uma chamada de função, ocorre a chamada recursiva para emitir a expressão do identificador dessa função. Em seguida, abre-se um parêntese, emite-se uma expressão para cada argumento, separados por vírgula, e, por fim, fecha-se o parêntese. Isso é demonstrado pelo trecho no [Código 39](#).

Esse sistema permite traduzir expressões matemáticas, como mostrada na [Equação 5.7](#), gerando o código correspondente apresentado no [Código 40](#).

$$\rho_d = 0, \vec{1}, 1 \quad (5.7a)$$

$$\rho_s = 1, \vec{0}, 1 \quad (5.7b)$$

$$n = +2^8 \quad (5.7c)$$

$$f = \frac{\rho_d}{\pi} + \rho_s * \frac{n+2}{2 * \pi} * \cos \theta_h^n \quad (5.7d)$$

5.5.4 Fases do Processo de Geração do Shader

A função `emit` é o ponto de partida do processo de emissão de código, onde ocorre a transformação final da AST em um shader GLSL. Este processo é dividido em três fases principais, cada uma com responsabilidades bem definidas para garantir a geração correta do código.

1. Inicialização e Estruturação

Nesta etapa inicial, o foco está em estabelecer a base do shader GLSL. Isso inclui:

- **Parâmetros:** emissão de uma seção parametrizável, `::begin parameters ::end parameters`, dos shaders, requerida pela ferramenta Disney BRDF Explorer.
- **Marcadores de Seção:** adição de delimitadores como `::begin shader` e `::end shader`, que iniciam o código GLSL na ferramenta da Disney.
- **Built-ins:** emissão das declarações e definições de variáveis e funções *built-in* necessárias, para dar ao shader suporte às convenções da [Tabela 5](#).

2. Processamento de Símbolos

Após a estruturação inicial, a função processa uma tabela de símbolos (Scope) para organizar as declarações e definições necessárias. As operações nesta fase incluem:

- **Iteração sobre Símbolos:** processamento de variáveis escalares (Type_Basic), definições de funções (Type_Function) e variáveis vetoriais (Type_Vector).
- **Separação de Declarações e Definições:** uso de duas instâncias distintas de `StringBuilder`, uma para declarações, que devem vir antes para que as variáveis possam ser usadas globalmente, e outra para definições que podem fazer uso dessas declarações sejam built-ins ou definidos pelos usuários.

3. Emissão Final

A fase final concatena os elementos gerados anteriormente, garantindo que o código esteja na ordem correta para compilação. Essa fase gera a função principal BRDF, que contém todas as declarações concatenadas, somadas às declarações e preâmbulos. Por fim, é realizada a gravação em disco do shader completo.

5.5.4.1 Variáveis Built-in

A emissão correta de um shader de BRDF necessita da geração de variáveis *built-in* diretamente no `shader`, conforme ilustrado na [Código 42](#). Essas variáveis, presentes na tabela [Tabela 7](#), seguem as convenções estabelecidas e são inicializadas automaticamente dentro da função de entrada do `shader`, denominada BRDF, conforme o padrão da ferramenta Disney.

Essas variáveis fornecem a infraestrutura necessária para cálculos de reflectância bidirecional em shaders GLSL. A ferramenta Disney disponibiliza algumas delas como parâmetros da função de entrada BRDF, mas com nomenclaturas diferentes das adotadas neste trabalho. Essas parâmetros são: `N` que representa a normal da superfície; `L` que indica a direção da luz incidente (`L`); `V` que representa a direção de visualização. As demais variáveis são calculadas na emissão de código.

Para calcular variáveis como `omega_i`, `theta_d` e `phi_i`, foram desenvolvidas funções auxiliares que produzem *strings* válidas em GLSL. A entrada dessas funções é uma *string* representando o vetor (`v`), substituída diretamente no corpo do GLSL gerado:

- `phi(v)`: Calcula o ângulo azimutal (ϕ), gera `atan(sqrt(v.y*v.y + v.x*v.x), v.z)`.

- `theta(v)`: Calcula o ângulo polar (θ) com a expressão `atan(v.y, v.x)`.

A declaração dessas variáveis embutidas é realizada pela função `emit_builtin_globals_declaration`, que também define constantes matemáticas como `\pi` e `\epsilon`. É apresentado na Tabela 7 como cada convenção é mapeada para código GLSL. Para maior clareza, o prefixo das variáveis discutido na seção de unicidade (subseção 5.5.2) foi removido.

Símbolo	Código GLSL Gerado
θ_i	<code>atan(omega_i.y, omega_i.x)</code>
θ_o	<code>atan(omega_o.y, omega_o.x)</code>
ϕ_i	<code>atan(sqrt(omega_i.y*omega_i.y+omega_i.x*omega_i.x), omega_i.z)</code>
ϕ_o	<code>atan(sqrt(omega_o.y*omega_o.y+omega_o.x*omega_o.x), omega_o.z)</code>
ω_i	<code>L</code>
ω_o	<code>V</code>
\vec{n}	<code>normalize(N)</code>
\vec{h}	<code>normalize(L+V)</code>
θ_h	<code>acos(dot(vec_h, vec_n))</code>
θ_d	<code>acos(dot(vec_h, omega_i))</code>
π	<code>3.141592653589793</code>
ϵ	<code>1.192092896e-07</code>

Tabela 7: Tabela de mapeamento de convenções para código GLSL

5.5.5 Emissão de Definições de Função

A emissão de definições de função ocorre antes da inicialização das variáveis. A assinatura da função, que inclui o nome (baseado na estratégia da subseção 5.5.2), é gerada, juntamente com o tipo de retorno e os parâmetros, mapeados conforme a subseção 5.5.1. Os nomes dos parâmetros são determinados pelos símbolos presentes no escopo da função, enquanto o corpo da função é emitido chamando `emit_expr`, da mesma forma que a emissão de expressões.

O mapeamento entre tipos em EquationLang e GLSL é bijetivo, garantindo que qualquer função definida em EquationLang possa ser representada em GLSL. Um exemplo de definição gerada é mostrado na Código 43, que se refere às funções de reflexão e normalização da equação subseção 5.5.5. Os nomes gerados seguem a numeração discutida na subseção 5.5.2, e o retorno é sempre avaliado como uma única expressão.

$$\text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}} \quad (5.8a)$$

$$\text{reflect}(\vec{I}, \vec{N}) = 2 * (\vec{I} \cdot \vec{N}) * \vec{N} - \vec{I} \quad (5.8b)$$

Código 26 – Função de percurso walk.

```

1 // Por brevidade vamos omitir varios casos do 'switch' que seguem a
2 // mesma lógica
3 walk :: proc(v: ^Visitor($T), node: ^ast.Node) {
4     if v == nil || node == nil {
5         return
6     }
7     v := v->visit(node)
8     if v == nil {
9         return
10    }
11    using ast
12    switch n in &node.derived {
13        case ^Start:
14            for d in n.decls {
15                walk(v, d)
16            }
17        // ...
18        // casos OMITIDOS aqui Também
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        // casos OMITIDOS aqui Também
34
35        case ^Expr_Infix:
36            walk(v, n.left)
37            walk(v, n.right)
38
39        case ^Expr_Grouped:
40            walk(v, n.expr)
41
42        case ^Expr_Function_Call:
43            walk(v, n.left)
44            for e in n.exprs {
45                walk(v, e)
46            }
47        case:
48            assert(false, "Unhandled token on walk_print ")
49    }
50    v = v->visit(nil)
51 }
```

Código 34 – Parte do switch da inferencia de tipos.

```

1 infer_type :: proc(expr: ^ast.Expr, allow_invalid := false, default :
2   ^ast.Type = ast.ty_invalid ) -> ^Type {
3   /// Código omitido por brevidade /**
4   #partial switch e in expr.derived {
5     case ^Expr_Identifier:
6       // Alway set by the parser if identifier starts with '\vec{}'
7       if e.is_vector {
8         type = new_type_vector(ty_number, 3) // vector 3 of type
9           number (real)
10      } else if e.identifier.kind == .Pi || e.identifier.kind ==
11        .Epsilon {
12          type = ty_number
13        } else {
14          key := key_from_identifier(e)
15          sym, ok := scope_get(key)
16          if ok {
17            if sym.type != nil {
18              type = sym.type
19            }
20          }
21        /// Código omitido por brevidade /**
22      }
23
24      case ^Expr_Prefix:
25        right_type := infer_type(e.right, allow_invalid, default)
26        /// Código omitido, mas aqui fazemos a validação da subexpressão
27        // direita e atribuimos o tipo correto para Expr_Prefix /**
28
29      case ^Expr_Infix:
30        ty_left := infer_type(e.left, allow_invalid, default)
31        ty_right := infer_type(e.right, allow_invalid, default)
32        /// Código omitido
33        // Inferimos tipo da expressão esquerda e direto dessa operação
34        // binária
35        // Depois validamos compatibilidade considerando a operação sendo
36        // usada nessas duas expressões
37
38      /// Outros casos ... /**
39    }

```

Código 39 – Emissão de chamada de funções.

```

1
2   case ^Expr_Function_Call:
3     emit_expr(sb, e.left)
4     sbprint(sb, '(')
5     for arg, idx in e.exprs {
6       emit_expr(sb, arg)
7       if idx < (len(e.exprs) -1) {
8         sbprint(sb, ',')
9       }
10      }
11      sbprint(sb, ')')

```

Código 40 – Exemplo de código de expressão gerado.

```

1 var_12_rho_d = vec3(0.0, 1.0, 1.0);
2 var_13_n      = pow(2.0, 8.0);
3 var_14_rho_s = vec3(1.0, 0.0, 1.0);
4 var_15_f      = ((var_12_rho_d / var_1_pi) +
5                   ((var_14_rho_s * ((var_13_n + 2.0) / (2.0 * var_1_pi))) *
6                   pow(cos(var_10_theta_h), var_13_n)));

```

Código 41 – Emitir expressão.

```

1  case ^Expr_Prefix:
2      #partial switch e.op.kind {
3          case .ArcSin:
4              sbprint(sb, "asin(")
5              emit_expr(sb, e.right)
6              sbprint(sb, ")")
7              return
8
9      //... Outros casos omissos
10
11     case .Tan:
12         sbprint(sb, "tan(")
13         emit_expr(sb, e.right)
14         sbprint(sb, ")")
15         return
16     case .Exp:
17         sbprint(sb, "exp(")
18         emit_expr(sb, e.right)
19         sbprint(sb, ")")
20         return
21     case .Vec:
22         ty_vec, ok := e.ty_inferred.derived.(^ast.Type_Vector)
23         assert(ok && ty_vec.dimensions == 3)
24
25         sbprint(sb, "vec3(")
26         emit_expr(sb, e.right)
27         sbprint(sb, ")")
28         return
29     case ^Expr_Infix:
30         op: string
31         #partial switch e.op.kind {
32             case .Plus: op = "+"
33             case .Minus: op = "-"
34             case .Mul:
35                 // Check if both operands are vectors
36                 if is_vector(e.left.ty_inferred) && is_vector(e.right.ty_inferred)
37                 {
38                     sbprint(sb, "dot(")
39                     emit_expr(sb, e.left)
40                     sbprint(sb, ",")
41                     emit_expr(sb, e.right)
42                     sbprint(sb, ")")
43                     return
44                 } else {
45                     op = "*"
46                 }
47             case .Frac, .Div: op = "/"
48             // Especially handled because it's not infix in glsl
49             case .Caret: op = ""
50                 sbprint(sb, "pow(")
51                 emit_expr(sb, e.left)
52                 sbprint(sb, ",")
53                 emit_expr(sb, e.right)
54                 sbprint(sb, ")")
55                 return
56
57             //... Outros casos omissos
58             case .Cross: op = ""
59                 sbprint(sb, "cross(")
60                 emit_expr(sb, e.left)
61                 sbprint(sb, ",")
```

Código 42 – Recorte da função BRDF one as variaveis built-ins são inicializadas

```

1 ////////////// START OF BUILTINS INITIALIZATION ///////////
2 var_0_vec_h = normalize(L + V);
3 var_3_vec_n = normalize(N);
4 var_1_pi = 3.141592653589793;
5 var_2_epsilon = 1.192092896e-07;
6 var_4_vec_omega_i = L;
7 var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
8 var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
9                     var_4_vec_omega_i.x * var_4_vec_omega_i.x),
10                    var_4_vec_omega_i.z);
11 var_7_vec_omega_o = V;
12 var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
13 var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
14                     var_7_vec_omega_o.x * var_7_vec_omega_o.x),
15                     var_7_vec_omega_o.z);
16 var_10_theta_h = acos(dot(var_0_vec_h, N));
17 var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
18 ////////// END OF BUILTINS INITIALIZATION ///////////

```

Código 43 – Código GLSL gerado pelo compilador para as funções de normalização e reflexão de vetores.

```

1 ////////////// START FUNCTIONS DECLARATIONS ///////////
2 vec3 var_13_reflect(vec3 var_14_vec_I, vec3 var_15_vec_N) {
3     return
4         (((2.0*(dot(var_14_vec_I, var_15_vec_N)))*var_15_vec_N)-var_14_vec_I);
5
6     vec3 var_17_text_normalize(vec3 var_18_vec_u) {
7         return (var_18_vec_u/sqrt(dot(var_18_vec_u, var_18_vec_u)));
8     }
9 ////////// END FUNCTIONS DECLARATIONS ///////////

```

6

Resultados

Este capítulo apresenta os resultados dos experimentos com diferentes BRDFs, servindo como validação e visualização da capacidade do compilador desenvolvido. A escolha de cada BRDF foi direcionada para explorar expressões matemáticas com diversos níveis de complexidade, aspectos importantes para testar a capacidade do compilador desenvolvido neste projeto.

Os experimentos seguem uma metodologia padronizada. Inicialmente, apresenta-se a BRDF do experimento, incluindo sua referência, quando relevante, uma breve explicação conceitual. Em seguida, demonstra-se o código fonte que descreve a BRDF em `EquationLang`, acompanhado de sua representação em PDF `LATEX`. Utilizando o compilador desenvolvido, o código fonte é traduzido para linguagem de *shading GLSL* e carregado na ferramenta Disney BRDF Explorer.

A análise inclui gráficos 3D e 2D da distribuição de reflexão especular e difusa da BRDF. Para demonstrar a eficácia do código `GLSL` gerado, são renderizados três objetos tridimensionais utilizando técnicas de *raytracing* fornecido pela ferramenta Disney. Os objetos possuem ângulos em coordenadas polares fixas, com condições padronizadas de iluminação: ângulos de elevação (θ_i) e azimutal (ϕ_i) da luz incidente predefinidos em 33, 8941 e 145, 826 respectivamente; gamma fixado em 2, 112 e exposição em -1, 248.

O *plot*¹ 3D de BRDFs na ferramenta Disney Explorer oferece uma visualização que fixa uma direção de luz incidente (ω_i) e coleta valores da direções de visualização (ω_o) em um hemisfério como entrada para BRDF. Para cada direção de visualização, renderiza-se renderiza um primitivo proporcional ao valor da função BRDF.

O *plot* polar, por sua vez, representa um corte bidimensional, fixando a direção de luz incidente (ω_i) e o ângulo azimutal de saída (ϕ_o), variando apenas o ângulo polar de saída

¹O termo *plot* é comumente usado em ferramentas de visualização e análise para se referir a gráficos ou representações visuais de dados. Aqui, refere-se a representações 2D (polar) ou 3D usadas para ilustrar os componentes especulares e difusos de cada canal de cor da BRDF.

(θ_o), similar ao mostrado nas figuras da [seção 2.2](#). Cada ponto representa o valor médio das componentes da BRDF, visualizando o comportamento da reflectância em diferentes ângulos de observação. Em alguns casos, fatores logarítmicos são utilizados para melhor visualização do gráfico.

É importante observar que os gráficos polares e 3D representam simultaneamente os três canais de cores, como na [Figura 39](#), podendo haver sobreposição entre vermelho, azul e verde na visualização, já que a distribuição de cada canal pode ser idêntica em um dado experimento.

Embora os experimentos contenham uma explicação sobre a BRDF, o foco principal permanece na representação fidedigna das equações em GLSL provida pelo compilador. Recomenda-se observar rapidamente o código gerado para compreender sua estrutura, reconhecendo que o código GLSL gerado por computador não é tão legível quanto código *shading* escrito manualmente.

Alguns experimentos exploram múltiplas formas de expressar a mesma BRDF. Não apenas com parâmetros distintos, mas também com expressões matemáticas alternativas. Para facilitar a navegação, a tabela [Tabela 8](#) é disponibilizada para acesso rápido às imagens e códigos dos experimentos.

Experimento	Equações	Objetos 3D	Plots	GLSL
Blinn-Phong	Figura 23	Figura 25	Figura 24	Código 44
Cook-Torrance	Figura 26	Figura 28	Figura 27	Código 47
Ward	Figura 29	Figura 31	Figura 30	Código 50
Ashikhmin-Shirley	Figura 32	Figura 34	Figura 33	Código 54
Oren-Nayar	Figura 35	Figura 37	Figura 36	Código 58
Ashikhmin-Shirley ₂	Figura 38	Figura 40	Figura 39	Código 63
Cook-torrance ₂	Figura 41	Figura 43	Figura 42	Código 66
Dür	Figura 44	Figura 46	Figura 45	Código 69
Edwards-2006	Figura 47	Figura 49	Figura 48	Código 72
Kajiya-Kay-1989 _*	Figura 50	Figura 52	Figura 51	Código 75
Minnaert	Figura 53	Figura 55	Figura 54	Código 78

Tabela 8: Tabela dos Experimentos

Concluímos que os experimentos realizados apresentaram resultados satisfatórios. O compilador desenvolvido demonstra flexibilidade ao capturar as nuances das diferentes BRDFs, inclusive em materiais com estruturas complexas. O sistema permite diversas parametrizações e equações alternativas para representar os comportamentos da superfície.

Os resultados obtidos não apenas validam a abordagem metodológica adotada, mas também abrem perspectivas para futuras extensões e refinamentos da ferramenta. Após o último experimento, seguimos diretamente para o capítulo de conclusão ([Capítulo 7](#)), onde são discutidas as possíveis direções para a continuidade deste trabalho.

6.1 Experimento BRDF Blinn-Phong

Neste experimento usamos uma BRDF com um método simplificado de cálculo de reflexão especular, presente na [Figura 23](#), introduzido por Blinn-Phong ([BLINN, 1977](#)). O [Código 46](#), escrito `EquationLang`, é a entrada para o compilador. O [Código 44](#) e [Código 45](#) são a saída em GLSL. A renderização dos objetos 3D estão em [Figura 25](#). Por fim, os plots estão na [Figura 24](#).

6.1.1 Representação em documento L^AT_EX

Figura 23 – Equações da BRDF do experimento blinn-phong-Kay em documento L^AT_EX.

$$\rho_d = 0, \vec{1}, 1 \quad (1)$$

$$\rho_s = 1, \vec{0}, 1 \quad (2)$$

$$n = +2^8 \quad (3)$$

$$f = \frac{\rho_d}{\pi} + \rho_s * \frac{n+2}{2*\pi} * \cos \theta_h^n \quad (4)$$

6.1.2 Visualização do Resultado

Figura 24 – *Plots* da BRDF deste experimento.

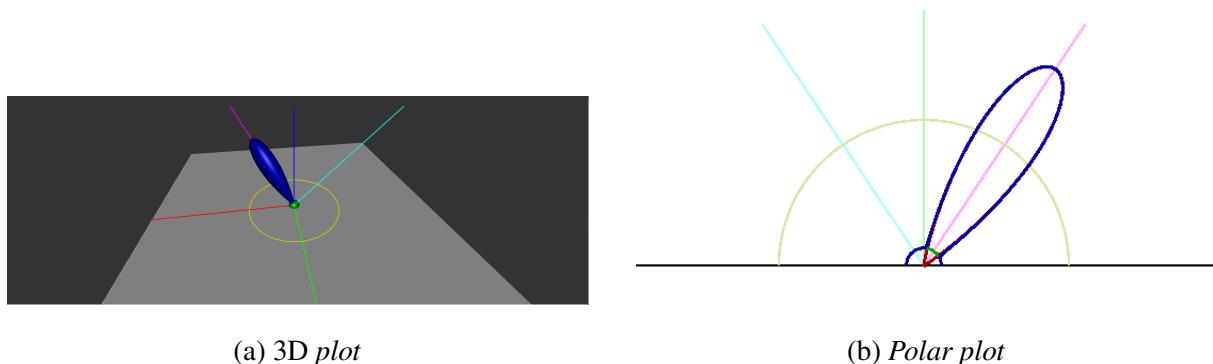


Figura 25 – Objetos 3D renderizados por este experimento

(a) *Teapot*

(b) Dragão de Stanford

(c) Goblin

6.1.3 Código GLSL Gerado

Código 44 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19
20 //////////// START OF USER DECLARED ///////////
21 vec3 var_12_rho_s;
22 float var_13_n;
23 vec3 var_14_rho_d;
24 vec3 var_15_f;
25 //////////// END OF USER DECLARED ///////////

```

Código 45 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
2
3     ////////////////// START OF BUILTINS INITIALIZATION //////////////////
4     var_0_vec_h = normalize(L + V);
5     var_3_vec_n = normalize(N);
6     var_1_pi = 3.141592653589793;
7     var_2_epsilon = 1.192092896e-07;
8     var_4_vec_omega_i = L;
9     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
10    var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
11                           var_4_vec_omega_i.x * var_4_vec_omega_i.x),
12                           var_4_vec_omega_i.z);
13    var_7_vec_omega_o = V;
14    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
15    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
16                           var_7_vec_omega_o.x * var_7_vec_omega_o.x),
17                           var_7_vec_omega_o.z);
18    var_10_theta_h = acos(dot(var_0_vec_h, N));
19    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
20    ////////////////// END OF BUILTINS INITIALIZATION //////////////////
21    var_12_rho_s = vec3(1.0, 0.0, 1.0);
22    var_13_n = pow(2.0, 8.0);
23    var_14_rho_d = vec3(0.0, 1.0, 1.0);
24    var_15_f = ((var_14_rho_d / var_1_pi) +
25                 ((var_12_rho_s * ((var_13_n + 2.0) / (2.0 *
26                               var_1_pi)))) *
27                 pow(cos(var_10_theta_h), var_13_n));
28
29     return vec3(var_15_f);
}

```

6.1.4 Código Fonte em EquationLang

Código 46 – Código fonte da BRDF deste experimento (parte 1).

```

1 \begin{document}
2
3 \begin{equation}
4     \rho_d = \vec{0,1,1}
5 \end{equation}
6
7 \begin{equation}
8     \rho_s = \vec{1,0,1}
9 \end{equation}
10
11 \begin{equation}
12     n = +2^8
13 \end{equation}
14
15 \begin{equation}
16 f = \frac{\rho_d}{\pi} + \rho_s * \frac{n+2}{2*\pi} *
17 \cos{\theta_h}^n
18 \end{equation}
```

6.2 Experimento BRDF Cook-Torrance

Este experimento é baseada no modelo microfacetado que descreve o comportamento de reflexão de superfícies metálicas descrito no trabalho de Cook-Torrance (COOK; TORRANCE, 1982). As equações e parametros escolhidos que descrevem esse modelo estão em [Figura 26](#). O código fonte em EquationLang para o compilador está em [Código 49](#). O código GLSL está dividido em duas partes, parte 1 está no [Código 47](#) e a segunda parte está em [Código 48](#). A renderização dos objetos 3D usando essa BRDF se encontra em [Figura 28](#). Usamos plot logaritmo para gerar os plots 3D e polar presentes na [Figura 27](#).

6.2.1 Representação em documento L^AT_EX

Figura 26 – Equações da BRDF do experimento cook-torrance em documento L^AT_EX.

$$m = 0.13 \quad (1)$$

$$\rho_d = 0.3, 0.05, 0.05 \quad (2)$$

$$\rho_s = 0.0, 0.2, 1.0 \quad (3)$$

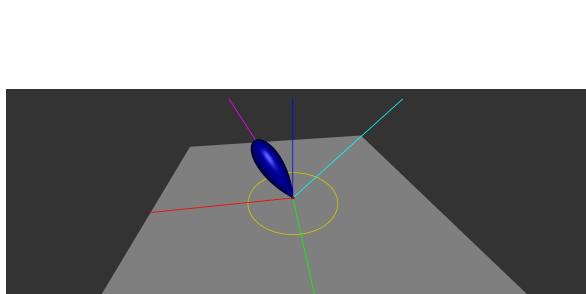
$$f = \frac{\rho_d}{\pi} + \frac{\rho_s}{\pi} * \frac{D * G}{(\vec{n} \cdot \vec{\omega}_i) * (\vec{n} \cdot \vec{\omega}_o)} \quad (4)$$

$$G = \min(1, \min(\frac{2 * (\vec{n} \cdot \vec{h}) * (\vec{n} \cdot \vec{\omega}_o)}{(\vec{h} \cdot \vec{\omega}_o)}, \frac{2 * (\vec{n} \cdot \vec{h}) * (\vec{n} \cdot \vec{\omega}_i)}{(\vec{h} \cdot \vec{\omega}_i)})) \quad (5)$$

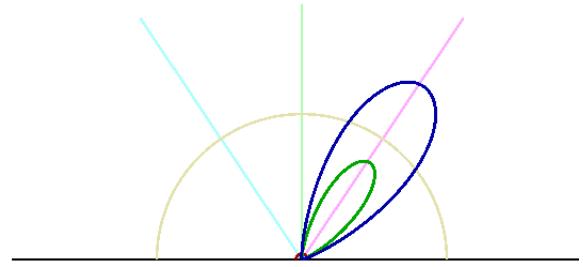
$$D = \frac{1}{(m^2) * (\cos \theta_h)^4} * \exp -((\tan \theta_h) / m)^2 \quad (6)$$

6.2.2 Visualização do Resultado

Figura 27 – Plots da BRDF deste experimentos.



(a) 3D plot



(b) Polar plot

Figura 28 – Objetos 3D renderizados por este experimento



(a) Teapot



(b) Dragão de Stanford



(c) Goblin

6.2.3 Código GLSL Gerado

Código 47 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```
1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 ////////////// START OF BUILTINS DECLARTION /////////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 ////////////// END OF BUILTINS DECLARTION /////////////
19 ////////////// START OF USER DECLARED /////////////
20 float var_12_G;
21 vec3 var_13_rho_s;
22 float var_14_m;
23 float var_15_D;
24 vec3 var_16_rho_d;
25 vec3 var_17_f;
26 ////////////// END OF USER DECLARED /////////////
27 ////////////// START FUNCTIONS DECLARATIONS /////////////
28 ////////////// END FUNCTIONS DECLARATIONS ///////////
```

Código 48 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
2
3     ////////////////// START OF BUILTINS INITIALIZATION //////////////////
4     var_0_vec_h = normalize(L + V);
5     var_3_vec_n = normalize(N);
6     var_1_pi = 3.141592653589793;
7     var_2_epsilon = 1.192092896e-07;
8     var_4_vec_omega_i = L;
9     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
10    var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
11                           var_4_vec_omega_i.x * var_4_vec_omega_i.x),
12                           var_4_vec_omega_i.z);
13    var_7_vec_omega_o = V;
14    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
15    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
16                           var_7_vec_omega_o.x * var_7_vec_omega_o.x),
17                           var_7_vec_omega_o.z);
18    var_10_theta_h = acos(dot(var_0_vec_h, N));
19    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
20    ////////////////// END OF BUILTINS INITIALIZATION //////////////////
21
22    var_12_G = min(1.0, min(((2.0 * (dot(var_3_vec_n, var_0_vec_h))) *
23                             (dot(var_3_vec_n, var_7_vec_omega_o))) /
24                             (dot(var_0_vec_h, var_7_vec_omega_o))),
25                             (((2.0 * (dot(var_3_vec_n, var_0_vec_h))) *
26                             (dot(var_3_vec_n, var_4_vec_omega_i))) /
27                             (dot(var_0_vec_h, var_7_vec_omega_o))));;
28    var_13_rho_s = vec3(0.0, 0.2, 1.0);
29    var_14_m = 0.13;
30    var_15_D = ((1.0 / ((pow(var_14_m, 2.0)) *
31                  pow((cos(var_10_theta_h)), 4.0))) *
32                  exp((-pow(((tan(var_10_theta_h)) / var_14_m),
33                             2.0))));;
34    var_16_rho_d = vec3(0.3, 0.05, 0.05);
35    var_17_f =
36        ((var_16_rho_d / var_1_pi) +
37         ((var_13_rho_s / var_1_pi) *
38          ((var_15_D * var_12_G) / ((dot(var_3_vec_n,
39                                      var_4_vec_omega_i)) *
40                                      (dot(var_3_vec_n,
41                                         var_7_vec_omega_o))))));
41
42    return vec3(var_17_f);
43 }

```

6.2.4 Código Fonte em EquationLang

Código 49 – Código fonte da BRDF deste experimento.

```

1 \begin{equation}
2 m = 0.13
3 \end{equation}
4
5 \begin{equation}
6     \rho_d = \vec{0.3, 0.05, 0.05}
7 \end{equation}
8
9 \begin{equation}
10    \rho_s = \vec{0.0, 0.2, 1.0}
11 \end{equation}
12
13 \begin{equation}
14 f = \frac{\rho_d}{\pi} +
15 \frac{\rho_s}{\pi} *
16 \frac{D^*G}{(\vec{n} \cdot \vec{\omega_i}) * (\vec{n} \cdot \vec{\omega_o})}
17 \end{equation}
18 \end{equation}
19
20 \begin{equation}
21 G = \min(1, \min(
22 \frac{2 * ((\vec{n} \cdot \vec{h}) * (\vec{n} \cdot \vec{\omega_o})) + (\vec{n} \cdot \vec{h}) * (\vec{n} \cdot \vec{\omega_i}))}{2 * ((\vec{n} \cdot \vec{h}) * (\vec{n} \cdot \vec{\omega_o}))})
23 \end{equation}
24 \begin{equation}
25 D = \frac{1}{(m^2) * (\cos{\theta_h})^4 * \exp{(-(\tan{\theta_h})/m)^2}}
26 \end{equation}
```

6.3 Experimento BRDF Ward

Este experimento é baseado nas BRDF após revisão usando as notas de Walter (WALTER, 2005). Nessas notas, são detalhados o modelo BRDF de Ward. Suas equações podem ser vistas na Figura 29, enquanto o código em EquationLang está disponível no Código 52 e Código 52.

O código gerado pelo compilador são formados pelo [Código 50](#) e [Código 51](#). A renderização de objetos usando este modelo é ilustrada na [Figura 31](#) e os plots da sua reflectancia está na [Figura 30](#).

6.3.1 Representação em documento L^AT_EX

Figura 29 – Equações da BRDF do experimento ward em documento L^AT_EX.

Equations representing the Ward BRDF:

$$\text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}} \quad (1)$$

1. Half vector:

$$\vec{H} = \text{normalize}(\vec{\omega}_i + \vec{\omega}_o) \quad (2)$$

2. Tangent vector:

$$\vec{X} = \text{normalize}(0, \vec{1}, 0 \times \vec{n}) \quad (3)$$

3. Bitangent vector:

$$\vec{Y} = \text{normalize}(\vec{n} \times \vec{X}) \quad (4)$$

4. Roughness parameters:

$$\alpha_x = 0.4 \quad (5)$$

$$\alpha_y = 0.2 \quad (6)$$

5. Exponent calculation:

$$\text{exponent} = -\frac{\frac{\vec{H} \cdot \vec{X}^2}{\alpha_x} + \frac{\vec{H} \cdot \vec{Y}^2}{\alpha_y}}{(\vec{H} \cdot \vec{n})^2} \quad (7)$$

6. Specular term: 7. And Exponent:

$$\text{spec} = \frac{1}{4 * \pi * \alpha_x * \alpha_y * \sqrt{(\vec{\omega}_i \cdot \vec{n}) * (\vec{\omega}_o \cdot \vec{n})}} \cdot \exp(\text{exponent}) \quad (8)$$

8. Color parameters

$$\vec{C}_s = 1, \vec{1}, 1 \quad (9)$$

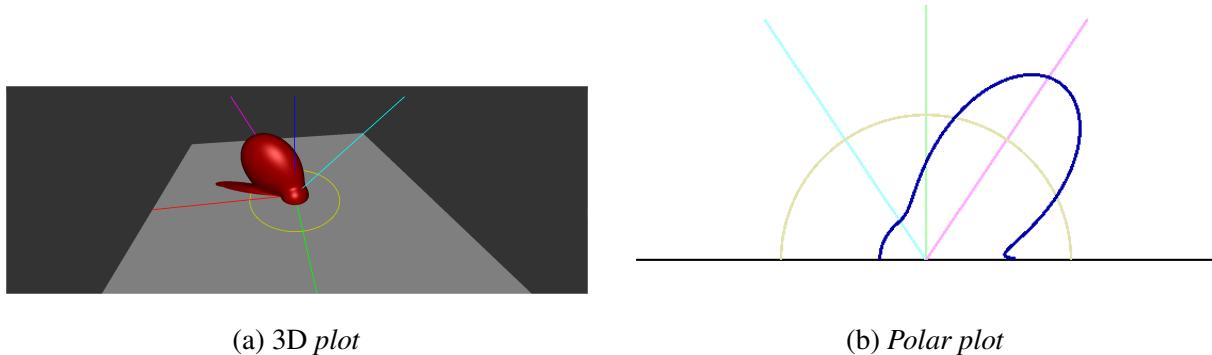
$$\vec{C}_d = 1, \vec{1}, 1 \quad (10)$$

9. Final BRDF:

$$f = \frac{\vec{C}_d}{\pi} + \vec{C}_s \cdot \text{spec} \quad (11)$$

6.3.2 Visualização do Resultado

Figura 30 – *Plots* da BRDF deste experimento.



(a) 3D *plot*

(b) *Polar plot*

Figura 31 – Objetos 3D renderizados por este experimento



(a) *Teapot*

(b) *Dragão de Stanford*

(c) *Goblin*

6.3.3 Código GLSL Gerado

Código 50 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```
1 analytic
2 ::begin parameters
3 #[type][name][min val][max val][default val]
4 ::end parameters
5 ::begin shader
6 //////////// START OF BUILTINS DECLARTION ///////////
7 vec3 var_0_vec_h;
8 vec3 var_3_vec_n;
9 float var_10_theta_h;
10 float var_11_theta_d;
11 float var_1_pi;
12 float var_2_epsilon;
13 vec3 var_4_vec_omega_i;
14 float var_5_theta_i;
15 float var_6_phi_i;
16 vec3 var_7_vec_omega_o;
17 float var_8_theta_o;
18 float var_9_phi_o;
19 //////////// END OF BUILTINS DECLARTION ///////////
20 //////////// START OF USER DECLARED ///////////
21 vec3 var_14_vec_X;
22 vec3 var_15_vec_Y;
23 float var_16_alpha_x;
24 float var_17_alpha_y;
25 vec3 var_18_vec_C_d;
26 vec3 var_19_vec_C_s;
27 vec3 var_20_vec_H;
28 float var_21_text_exponent;
29 float var_22_text_spec;
30 vec3 var_23_f;
31 //////////// END OF USER DECLARED ///////////
32 //////////// START FUNCTIONS DECLARATIONS ///////////
33 vec3 var_12_text_normalize(vec3 var_13_vec_u) {
34     return (var_13_vec_u / sqrt(dot(var_13_vec_u, var_13_vec_u)));
35 }
36 //////////// END FUNCTIONS DECLARATIONS ///////////
```

Código 51 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
2     ////////////////// START OF BUILTINS INITIALIZATION //////////////////
3     var_0_vec_h = normalize(L + V);
4     var_3_vec_n = normalize(N);
5     var_1_pi = 3.141592653589793;
6     var_2_epsilon = 1.192092896e-07;
7     var_4_vec_omega_i = L;
8     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
9     var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
10                         var_4_vec_omega_i.x * var_4_vec_omega_i.x),
11                         var_4_vec_omega_i.z);
12    var_7_vec_omega_o = V;
13    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
14    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
15                         var_7_vec_omega_o.x * var_7_vec_omega_o.x),
16                         var_7_vec_omega_o.z);
17    var_10_theta_h = acos(dot(var_0_vec_h, N));
18    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
19    ////////////////// END OF BUILTINS INITIALIZATION //////////////////
20    var_14_vec_X = var_12_text_normalize(cross(vec3(0.0, 1.0, 0.0),
21        var_3_vec_n));
22    var_15_vec_Y = var_12_text_normalize(cross(var_3_vec_n,
23        var_14_vec_X));
24    var_16_alpha_x = 0.4;
25    var_17_alpha_y = 0.2;
26    var_18_vec_C_d = vec3(1.0, 1.0, 1.0);
27    var_19_vec_C_s = vec3(1.0, 1.0, 1.0);
28    var_20_vec_H = var_12_text_normalize((var_4_vec_omega_i +
29        var_7_vec_omega_o));
30    var_21_text_exponent = (-((pow((dot(var_20_vec_H, var_14_vec_X) /
31        var_16_alpha_x), 2.0) +
32        pow((dot(var_20_vec_H, var_15_vec_Y) / var_17_alpha_y),
33        2.0)) /
34        pow((dot(var_20_vec_H, var_3_vec_n)), 2.0)));
35    var_22_text_spec = ((1.0 / (((4.0 * var_1_pi) * var_16_alpha_x) *
36        var_17_alpha_y) *
37            sqrt(((dot(var_4_vec_omega_i, var_3_vec_n)) *
38                (dot(var_7_vec_omega_o, var_3_vec_n))))))) *
39            exp((var_21_text_exponent)));
40    var_23_f = ((var_18_vec_C_d / var_1_pi) + (var_19_vec_C_s *
41        var_22_text_spec));
42
43    return vec3(var_23_f);
44}
```

6.3.4 Código Fonte em EquationLang

Código 52 – Código fonte da BRDF deste experimento (parte 1).

```
1 Equations representing the Ward BRDF:  
2   \begin{equation}  
3     \text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}}  
4   \end{equation}  
5 1. Half vector:  
6   \begin{equation}  
7     \vec{H} = \text{normalize}(\vec{\omega_i} + \vec{\omega_o})  
8   \end{equation}  
9 2. Tangent vector:  
10  \begin{equation}  
11    \vec{X} = \text{normalize}(\vec{0,1,0} \times \vec{n})  
12  \end{equation}  
13 3. Bitangent vector:  
14  \begin{equation}  
15    \vec{Y} = \text{normalize}(\vec{n} \times \vec{X})  
16  \end{equation}  
17 4. Roughness parameters:  
18  \begin{equation}  
19    \alpha_x = 0.4  
20  \end{equation}  
21  \begin{equation}  
22    \alpha_y = 0.2  
23  \end{equation}
```

Código 53 – Código fonte da BRDF deste experimento (parte 2).

```

1 5. Exponent calculation:
2   \begin{equation}
3     \text{exponent} = -\frac{
4       \frac{\vec{H} \cdot \vec{X}}{\alpha_x^2} +
5       \frac{\vec{H} \cdot \vec{Y}}{\alpha_y^2}
6     }{(\vec{H} \cdot \vec{n})^2}
7   \end{equation}
8 6. Specular term:
9 7. And Exponent:
10 \begin{equation}
11   \text{spec} = \frac{1}{4 * \pi} * \alpha_x * \alpha_y
12     * \sqrt{(\vec{\omega_i} \cdot \vec{n}) * (\vec{\omega_o} \cdot \vec{n})}
13     \cdot \exp{(-\text{exponent})}
14 \end{equation}
15 8. Color parameters
16 \begin{equation}
17   \vec{C_s} = \vec{1, 1, 1}
18 \end{equation}
19 \begin{equation}
20   \vec{C_d} = \vec{1, 1, 1}
21 \end{equation}
22 9. Final BRDF:
23 \begin{equation}
24   f = \frac{\vec{C_d}}{\pi} + \vec{C_s} \cdot \text{spec}
25 \end{equation}

```

6.4 Experimento BRDF Ashikhmin-Shirley

Neste experimento utilizamos uma BRDF anisotrópica desenvolvida por Ashikhmin-Shirley (ASHIKHMIN; SHIRLEY, 2000), que apresenta um modelo de reflexão não uniforme. A descrição matemática está presente na Figura 32, com o código fonte em EquationLang disponível no Código 56 e Código 57. Os códigos gerados em GLSL são apresentados nos Código 54 e Código 55. A renderização dos objetos 3D pode ser observada na Figura 34 e os plots correspondentes estão na Figura 33.

6.4.1 Representação em documento L^AT_EX

Figura 32 – Equações da BRDF do experimento ashikhmin-shirley-close-to-original-Kay em documento L^AT_EX.

Ashikhmin Shirley 2000 - Anisotropic phong reflectance model

- R_s : a color (RGB) that specifies the specular reflectance at normal incidence.
- R_d : a color (RGB) that specifies the diffuse reflectance
- n_u, n_v : two phong-like exponents that control the shape of the specular lobe

The model is a classical sum of a "specular" term and a "diffuse" term.

$$R_s = 0.4 \quad (1)$$

$$R_d = 0.9 \quad (2)$$

$$n_v = 1.5 \quad (3)$$

$$n_u = 300 \quad (4)$$

The specular component ρ_s of the BRDF is:

$$n = \vec{n} \quad (5)$$

$$h = \vec{h} \quad (6)$$

$$\text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}} \quad (7)$$

Tangent vector:

$$u = \text{normalize}(0, \vec{1}, 0 \times n) \quad (8)$$

Bitangent vector:

$$v = \text{normalize}(\vec{n} \times u) \quad (9)$$

$$k = \vec{\omega}_i \quad (10)$$

$$\rho_s(\vec{\omega}_i, \vec{\omega}_o) = \frac{\sqrt{(n_u + 1) * (n_v + 1)}}{8 * \pi} * \frac{(n \cdot h)^{\frac{(n_u * (h \cdot u)^2 + n_v * (h \cdot v)^2)}{1 - (h \cdot n)^2}} * F(k \cdot h)}{(h \cdot k) * \max((n \cdot \vec{\omega}_i), (n \cdot \vec{\omega}_o)))} \quad (11)$$

$$\rho_d(\vec{\omega}_i, \vec{\omega}_o) = \frac{28 * R_d}{23 * \pi} * (1 - R_s) * (1 - (1 - \frac{(n \cdot \vec{\omega}_i)}{2})^5) * (1 - (1 - \frac{(n \cdot \vec{\omega}_o)}{2})^5) \quad (12)$$

$$F(x) = R_s + (1 - R_s) * (1 - (k \cdot h))^5 \quad (13)$$

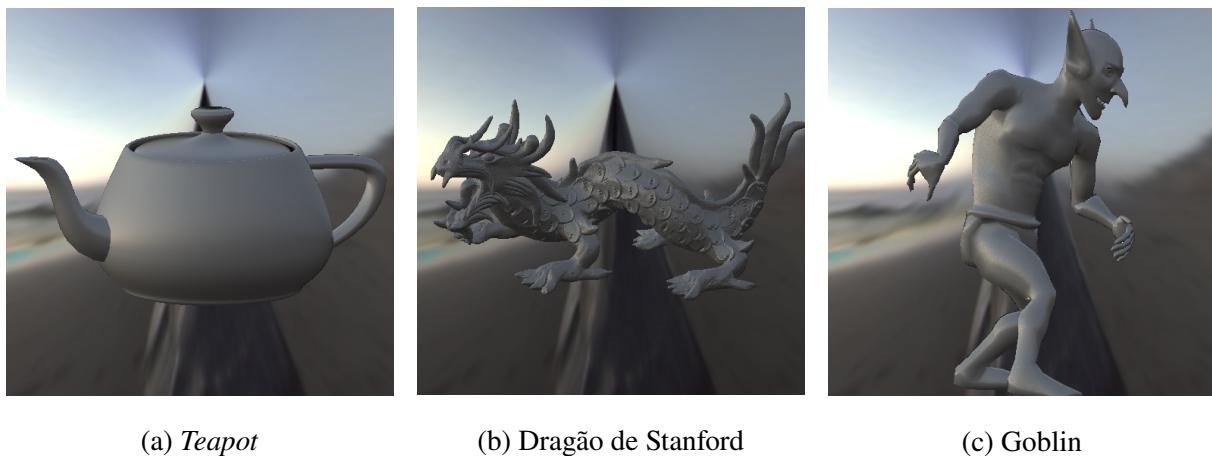
$$f = \rho_s(\vec{\omega}_i, \vec{\omega}_o) + \rho_d(\vec{\omega}_i, \vec{\omega}_o) \quad (14)$$

6.4.2 Visualização do Resultado

Figura 33 – *Plots* da BRDF deste experimento.



Figura 34 – Objetos 3D renderizados por este experimento



6.4.3 Código GLSL Gerado

Código 54 – Saida do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19 //////////// START OF USER DECLARED ///////////
20 vec3 var_12_k;
21 float var_13_n_v;
22 float var_14_n_u;
23 vec3 var_17_n;
24 vec3 var_18_u;
25 vec3 var_19_v;
26 float var_20_R_s;
27 vec3 var_21_h;
28 float var_24_R_d;
29 float var_27_f;
30 //////////// END OF USER DECLARED ///////////
31 //////////// START FUNCTIONS DECLARATIONS ///////////
32 vec3 var_15_text_normalize(vec3 var_16_vec_u) {
33     return (var_16_vec_u / sqrt(dot(var_16_vec_u, var_16_vec_u)));
34 }
35 float var_22_F(float var_23_x) {
36     return (var_20_R_s + (((1.0 - var_20_R_s)) *
37                         pow(((1.0 - (dot(var_12_k, var_21_h)))), 5.0)));
38 }
39 float var_25_rho_d(vec3 var_4_vec_omega_i, vec3 var_7_vec_omega_o) {
40     return (((((28.0 * var_24_R_d) / (23.0 * var_1_pi)) * ((1.0 -
41         var_20_R_s)) *
42         ((1.0 - pow(((1.0 - ((dot(var_17_n, var_4_vec_omega_i)) /
43             2.0))), 5.0)))) * ((1.0 - pow(((1.0 - ((dot(var_17_n, var_7_vec_omega_o)) /
44             2.0))), 5.0))));
```

Código 55 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 float var_26_rho_s(vec3 var_4_vec_omega_i, vec3 var_7_vec_omega_o) {
2     return ((sqrt(((var_14_n_u + 1.0) * (var_13_n_v + 1.0))) /
3             (8.0 * var_1_pi)) *
4             ((pow((dot(var_17_n, var_21_h)),
5                   (((var_14_n_u * pow((dot(var_21_h, var_18_u)), 2.0)) +
6                   (var_13_n_v * pow((dot(var_21_h, var_19_v)), 2.0))) /
7                   (1.0 - pow((dot(var_21_h, var_17_n)), 2.0)))) * *
8             var_22_F(dot(var_12_k, var_21_h))) /
9             ((dot(var_21_h, var_12_k)) * max((dot(var_17_n,
10            var_4_vec_omega_i)),
11            (dot(var_17_n,
12            var_7_vec_omega_o))))));
13 }
14 ////////////// END FUNCTIONS DECLARATIONS /////////////
15 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
16     ////////////////// START OF BUILTINS INITIALIZATION ///////////
17     var_0_vec_h = normalize(L + V);
18     var_3_vec_n = normalize(N);
19     var_1_pi = 3.141592653589793;
20     var_2_epsilon = 1.192092896e-07;
21     var_4_vec_omega_i = L;
22     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
23     var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
24                           var_4_vec_omega_i.x * var_4_vec_omega_i.x),
25                           var_4_vec_omega_i.z);
26     var_7_vec_omega_o = V;
27     var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
28     var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
29                           var_7_vec_omega_o.x * var_7_vec_omega_o.x),
30                           var_7_vec_omega_o.z);
31     var_10_theta_h = acos(dot(var_0_vec_h, N));
32     var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
33     ////////////////// END OF BUILTINS INITIALIZATION ///////////
34     var_12_k = var_4_vec_omega_i;
35     var_13_n_v = 1.5;
36     var_14_n_u = 300.0;
37     var_17_n = var_3_vec_n;
38     var_18_u = var_15_text_normalize(cross(vec3(0.0, 1.0, 0.0),
39             var_17_n));
40     var_19_v = var_15_text_normalize(cross(var_3_vec_n, var_18_u));
41     var_20_R_s = 0.4;
42     var_21_h = var_0_vec_h;
43     var_24_R_d = 0.9;
44     var_27_f = (var_26_rho_s(var_4_vec_omega_i, var_7_vec_omega_o) +
45                 var_25_rho_d(var_4_vec_omega_i, var_7_vec_omega_o));
46     return vec3(var_27_f);
47 }
```

6.4.4 Código Fonte em EquationLang

Código 56 – Código fonte da BRDF deste experimento (parte 1).

```

1 Ashikhmin Shirley 2000 - Anisotropic phong reflectance model
2
3 $R_s$ : a color (RGB) that specifies the specular reflectance
4 at normal incidence.
5
6 $R_d$ : a color (RGB) that specifies the diffuse reflectance
7
8 $n_u$, $n_v$ : two phong-like exponents that control the shape of the
9     specular lobe
10 The model is a classical sum of a "specular" term and a "diffuse"
11     term.
12 \begin{equation}
13     R_s = 0.4
14 \end{equation}
15
16 \begin{equation}
17     R_d = 0.9
18 \end{equation}
19
20 \begin{equation}
21     n_v = 1.5
22 \end{equation}
23
24 \begin{equation}
25     n_u = 300
26 \end{equation}
27
28 The specular component $\rho_s$ of the BRDF is:
29
30 \begin{equation}
31     n = \vec{n}
32 \end{equation}
33
34 \begin{equation}
35     h = \vec{h}
36 \end{equation}
37 \end{equation}
```

Código 57 – Código fonte da BRDF deste experimento (parte 2).

```

1 \begin{equation}
2   \text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}}
3 \end{equation}
4 Tangent vector:
5 \begin{equation}
6   u = \text{normalize}(\vec{0}, 1, 0) \times n
7 \end{equation}
8 Bitangent vector:
9 \begin{equation}
10  v = \text{normalize}(\vec{n} \times u)
11 \end{equation}
12 \begin{equation}
13  k = \vec{\omega}_i
14 \end{equation}
15
16 \begin{equation}
17  \rho_s(\vec{\omega}_i, \vec{\omega}_o) =
18    \frac{\sqrt{(n_u+1)(n_v+1)}}{8\pi} \\
19    * \frac{(n \cdot h)^{(\frac{n_u}{n} (h \cdot u)^2 + n_v (h \cdot v)^2)}{1 - (h \cdot n)^2}} \\
20    * F(k \cdot h) \cdot (h \cdot k) * \max((n \cdot \vec{\omega}_i), \\
21      (n \cdot \vec{\omega}_o))
22 \end{equation}
23 \begin{equation}
24  \rho_d(\vec{\omega}_i, \vec{\omega}_o) = \frac{28R_d}{23\pi} \\
25    * (1 - R_s) \\
26    * (1 - (1 - \frac{(n \cdot \vec{\omega}_i)^2}{2})^5) \\
27    * (1 - (1 - \frac{(n \cdot \vec{\omega}_o)^2}{2})^5)
28 \end{equation}
29
30 \begin{equation}
31  F(x) = R_s + (1 - R_s) * (1 - (k \cdot h))^5
32 \end{equation}
33
34 \begin{equation}
35  f = \rho_s(\vec{\omega}_i, \vec{\omega}_o) + \\
36    \rho_d(\vec{\omega}_i, \vec{\omega}_o)
37 \end{equation}

```

6.5 Experimento BRDF Oren-Nayar

Este experimento é baseado no trabalho de Oren e Nayar (OREN; NAYAR, 1994) o qual propuseram uma generalização do modelo de reflectância de Lambert, desenvolvendo uma abordagem matemática que modela com maior precisão a difusão de luz em superfícies rugosas naturais. As equações que descrevem esse experimento se encontram em [Figura 35](#). O código

fonte, em `EquationLang`, de entrada para o compilador está dividido em duas partes, a primeira está no [Código 61](#) e a segunda está no [Código 62](#). A renderização dos objetos 3D usando essa BRDF se encontra em [Figura 37](#) e seus plots no [Figura 36](#). A saída GLSL se encontram em: parte 1 ([Código 59](#)), parte 2 ([Código 59](#)) e parte 3 no [Código 60](#).

6.5.1 Representação em documento L^AT_EX

Figura 35 – Equações da BRDF do experimento oren-nayar em documento L^AT_EX.

$$\rho = 0.9 \quad (1)$$

$$\sigma = 30 * \pi / 180 \quad (2)$$

$$\theta_r = \arccos((V \cdot N)) \quad (3)$$

$$\phi_{\text{diff}} = (\text{normalize}(V - N * ((V \cdot N))) \cdot \text{normalize}(L - N * ((L \cdot N)))) \quad (4)$$

$$N = \vec{n} \quad (5)$$

$$L = \vec{\omega}_i \quad (6)$$

$$V = \vec{\omega}_o \quad (7)$$

$$\theta_i = \arccos((L \cdot N)) \quad (8)$$

$$\alpha = \max(\theta_i, \theta_r) \quad (9)$$

$$\beta = \min(\theta_i, \theta_r) \quad (10)$$

$$C_1 = 1 - 0.5 * (\sigma^2) / ((\sigma^2) + 0.33) \quad (11)$$

$$C_2 = \frac{0.45 * (\sigma^2)}{((\sigma^2) + 0.09)} * (\text{step}(\phi_{\text{diff}}) * (\sin(\alpha)) + (1 - \text{step}(\phi_{\text{diff}})) * (\sin(\alpha) - \frac{2 * \beta^3}{\pi})) \quad (12)$$

$$C_3 = 0.125 * (\sigma^2) / ((\sigma^2) + 0.09) * ((4 * \alpha * \beta) / (\pi * \pi))^2 \quad (13)$$

$$L_1 = \rho / \pi * (C_1 + \phi_{\text{diff}} * C_2 * \tan(\beta) + (1 - \text{abs}(\phi_{\text{diff}})) * C_3 * \tan((\alpha + \beta) / 2)) \quad (14)$$

$$L_2 = 0.17 * \rho * \rho / \pi * (\sigma^2) / ((\sigma^2) + 0.13) * (1 - \phi_{\text{diff}} * (4 * \beta * \beta) / (\pi * \pi)) \quad (15)$$

The *BRDF* output

$$f = L_1 + L_2 \quad (16)$$

Utility Functions

$$\text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}} \quad (17)$$

$$\text{abs}(v) = \max(v, -v) \quad (18)$$

Step function that returns 1 when $x \geq 0$ and 0 when $x < 0$

$$\text{step}(x) = \min(1, \max(0, x / (\text{abs}(x) + \epsilon))) \quad (19)$$

6.5.2 Visualização do Resultado

Figura 36 – *Plots* da BRDF deste experimento.

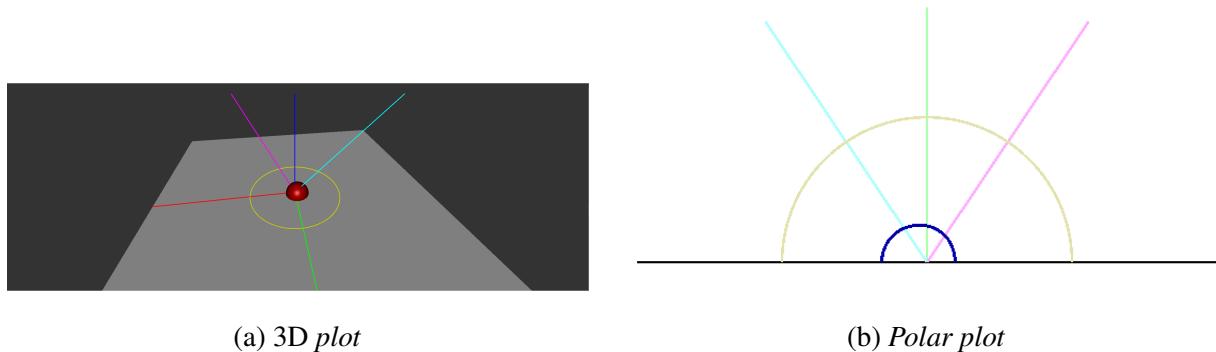
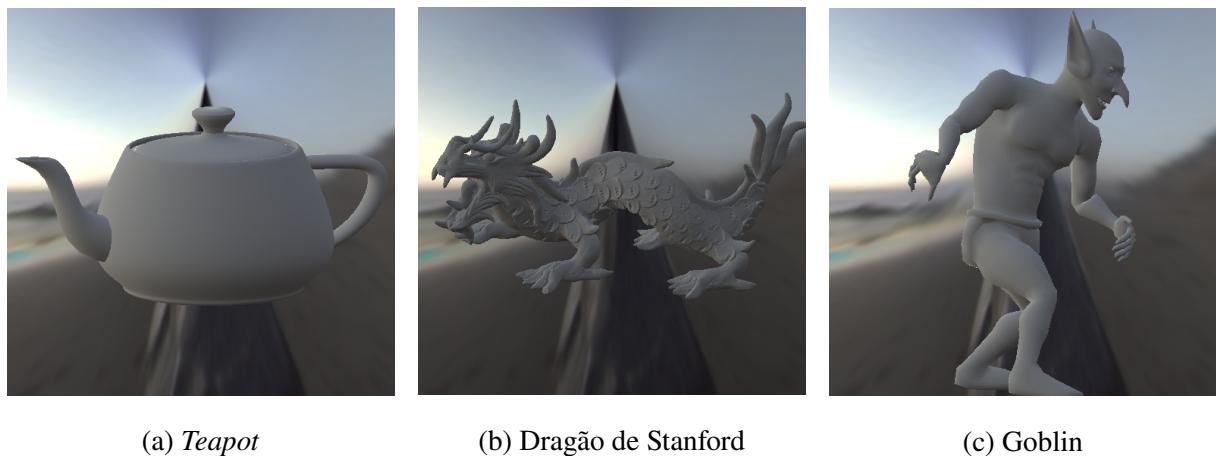


Figura 37 – Objetos 3D renderizados por este experimento



6.5.3 Código GLSL Gerado

Código 58 – Saida do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19 //////////// START OF USER DECLARED ///////////
20 float var_14_sigma;
21 vec3 var_19_V;
22 vec3 var_20_N;
23 vec3 var_21_L;
24 float var_22_phi_text_diff;
25 float var_23_theta_i;
26 float var_24_theta_r;
27 float var_25_alpha;
28 float var_26_beta;
29 float var_27_C_3;
30 float var_28_C_1;
31 float var_29_rho;
32 float var_30_C_2;
33 float var_31_L_1;
34 float var_32_L_2;
35 float var_33_f;
36 //////////// END OF USER DECLARED ///////////
37 //////////// START FUNCTIONS DECLARATIONS ///////////
38 float var_12_text_abs(float var_13_v) { return max(var_13_v,
   (-var_13_v)); }
39 float var_15_text_step(float var_16_x) {
40   return min(1.0, max(0.0, (var_16_x / ((var_12_text_abs(var_16_x) +
   var_2_epsilon)))));
41 }
42 vec3 var_17_text_normalize(vec3 var_18_vec_u) {
43   return (var_18_vec_u / sqrt(dot(var_18_vec_u, var_18_vec_u)));
44 }
45 //////////// END FUNCTIONS DECLARATIONS ///////////

```

Código 59 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
2     ////////////////// START OF BUILTINS INITIALIZATION //////////////////
3     var_0_vec_h = normalize(L + V);
4     var_3_vec_n = normalize(N);
5     var_1_pi = 3.141592653589793;
6     var_2_epsilon = 1.192092896e-07;
7     var_4_vec_omega_i = L;
8     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
9     var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
10                         var_4_vec_omega_i.x * var_4_vec_omega_i.x),
11                         var_4_vec_omega_i.z);
12    var_7_vec_omega_o = V;
13    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
14    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
15                         var_7_vec_omega_o.x * var_7_vec_omega_o.x),
16                         var_7_vec_omega_o.z);
17    var_10_theta_h = acos(dot(var_0_vec_h, N));
18    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
19    ////////////////// END OF BUILTINS INITIALIZATION //////////////////
20    var_14_sigma = ((30.0 * var_1_pi) / 180.0);
21    var_19_V = var_7_vec_omega_o;
22    var_20_N = var_3_vec_n;
23    var_21_L = var_4_vec_omega_i;
24    var_22_phi_text_diff = (dot(var_17_text_normalize(
25        (var_19_V - (var_20_N * ((dot(var_19_V,
26            var_20_N)))))),
27        var_17_text_normalize(
28            (var_21_L - (var_20_N * ((dot(var_21_L,
29                var_20_N)))))));
30    var_23_theta_i = acos(((dot(var_21_L, var_20_N))));
31    var_24_theta_r = acos(((dot(var_19_V, var_20_N))));
32    var_25_alpha = max(var_23_theta_i, var_24_theta_r);
33    var_26_beta = min(var_23_theta_i, var_24_theta_r);
34    var_27_C_3 = (((0.125 * (pow(var_14_sigma, 2.0))) /
35        (((pow(var_14_sigma, 2.0)) + 0.09))) *
36        pow((((((4.0 * var_25_alpha) * var_26_beta)) / ((var_1_pi *
37            var_1_pi))), 2.0));
38    var_28_C_1 = (1.0 - ((0.5 * (pow(var_14_sigma, 2.0))) /
39        (((pow(var_14_sigma, 2.0)) + 0.33))));
40    var_29_rho = 0.9;

```

Código 60 – Saída do compilador, código GLSL da BRDF deste experimento (parte 3).

```
1 var_30_C_2 = (((0.45 * (pow(var_14_sigma, 2.0))) /
2   (((pow(var_14_sigma, 2.0)) + 0.09))) *
3     (((var_15_text_step(var_22_phi_text_diff) *
4       (sin((var_25_alpha)))) +
5         (((1.0 - var_15_text_step(var_22_phi_text_diff))) *
6           ((sin((var_25_alpha)) -
7             pow(((2.0 * var_26_beta) / var_1_pi), 3.0))))));
6 var_31_L_1 = ((var_29_rho / var_1_pi) *
7   (((var_28_C_1 + ((var_22_phi_text_diff * var_30_C_2) *
8     tan((var_26_beta)))) +
9       (((((1.0 - var_12_text_abs(var_22_phi_text_diff)) *
10         var_27_C_3) *
11           tan((((var_25_alpha + var_26_beta) / 2.0))))));
10 var_32_L_2 = (((((0.17 * var_29_rho) * var_29_rho) / var_1_pi) *
11   (pow(var_14_sigma, 2.0))) /
12     (((pow(var_14_sigma, 2.0)) + 0.13))) *
13       ((1.0 - ((var_22_phi_text_diff * (((4.0 * var_26_beta) *
14         var_26_beta)) /
15           ((var_1_pi * var_1_pi))))));
15 var_33_f = (var_31_L_1 + var_32_L_2);
16 return vec3(var_33_f);
```

6.5.4 Código Fonte em EquationLang

Código 61 – Código fonte da BRDF deste experimento (parte 1).

```

1 \begin{equation}
2   \rho = 0.9
3 \end{equation}
4 \begin{equation}
5   \sigma = 30*\pi/180
6 \end{equation}
7 \begin{equation}
8   \theta_r = \arccos((\mathbf{V} \cdot \mathbf{N}))
9 \end{equation}
10 \begin{equation}
11   \phi_{\text{diff}} = (\text{normalize}(\mathbf{V} - \mathbf{N} * ((\mathbf{V} \cdot \mathbf{N}))) \cdot
12     \text{normalize}(\mathbf{L} - \mathbf{N} * ((\mathbf{L} \cdot \mathbf{N}))) )
13 \end{equation}
14 \begin{equation}
15   \mathbf{N} = \vec{n}
16 \end{equation}
17 \begin{equation}
18   \mathbf{L} = \vec{\omega_i}
19 \end{equation}
20 \begin{equation}
21   \mathbf{V} = \vec{\omega_o}
22 \end{equation}
23 \begin{equation}
24   \theta_i = \arccos((\mathbf{L} \cdot \mathbf{N}))
25 \end{equation}
26 \begin{equation}
27   \alpha = \max(\theta_i, \theta_r)
28 \end{equation}
29 \begin{equation}
30   \beta = \min(\theta_i, \theta_r)
31 \end{equation}
32 \begin{equation}
33   C_1 = 1 - 0.5 * (\sigma^2) / ((\sigma^2) + 0.33)
34 \end{equation}
35 \begin{equation}
36 \end{equation}
37 \begin{equation}
38 \end{equation}
39 \end{equation}
```

Código 62 – Código fonte da BRDF deste experimento (parte 2).

```

1 \begin{equation}
2     C_2 = \frac{0.45 * (\sigma^2)}{((\sigma^2) + 0.09)} *
3         (
4             \text{step}(\phi_{\text{diff}}) * (\sin(\alpha))
5             + (1 - \text{step}(\phi_{\text{diff}})) * (\sin(\alpha)
6                 - \frac{2 * \beta}{\pi})^3)
7 \end{equation}
8 \begin{equation}
9     C_3 = 0.125 * (\sigma^2) / ((\sigma^2) + 0.09) *
10        ((4 * \alpha * \beta) / (\pi * \pi))^2
11 \end{equation}
12 \begin{equation}
13     L_1 = \rho / \pi * (C_1 + \phi_{\text{diff}} * C_2 * \tan(\beta) +
14         (1 - \text{abs}(\phi_{\text{diff}})) * C_3 *
15             \tan((\alpha + \beta) / 2))
16 \end{equation}
17 \begin{equation}
18     L_2 = 0.17 * \rho * \rho / \pi * (\sigma^2) / ((\sigma^2) + 0.13) * (1 -
19             \phi_{\text{diff}} * (4 * \beta * \beta) / (\pi * \pi))
20 \end{equation}
21 \hspace{60px} The \emph{BRDF} output
22 \begin{equation}
23     f = L_1 + L_2
24 \end{equation}
25 \hspace{60px} Utility Functions
26 \begin{equation}
27     \text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}}
28 \end{equation}
29 \begin{equation}
30     \text{abs}(v) = \max(v, -v)
31 \end{equation}
32 \begin{equation}
33 \end{equation}
34 \hspace{60px} \small{Step function that returns 1 when x \verb">=" 0
35     and 0 when \verb"x < 0"}
36 \begin{equation}
37     \text{step}(x) = \min(1, \max(0, x / (\text{abs}(x) + \epsilon)))
38 \end{equation}

```

6.6 Experimento BRDF Ashikhmin-Shirley Alternativa

Nesse caso, foi realizado uma versão alternativa do experimento da [seção 6.4](#). As equações na [Figura 38](#) estão simplificadas e algumas equações foram comprimidas em uma só. Além disso, nessa versão, foram escolhidas diferentes constantes, como cor difusa, fatores de multiplicação, entre outros. Código fonte em [EquationLang](#) disponível no [Código 65](#). Os códigos gerados em GLSL estão no [Código 63](#) e [Código 64](#). Objetos 3D renderizados pode ser vistos na [Figura 40](#) e os plots estão na [Figura 39](#).

6.6.1 Representação em documento L^AT_EX

Figura 38 – Equações da BRDF do experimento ashikhmin-shirley-alternative em documento L^AT_EX.

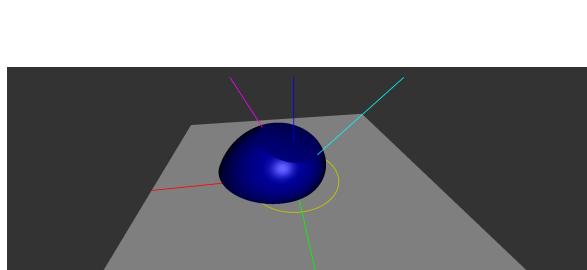
$$\rho_d = 0.3, \vec{0.3}, 0.3 \quad (1)$$

$$\rho_s = 0.0, \vec{0.2}, 1.0 * 20 \quad (2)$$

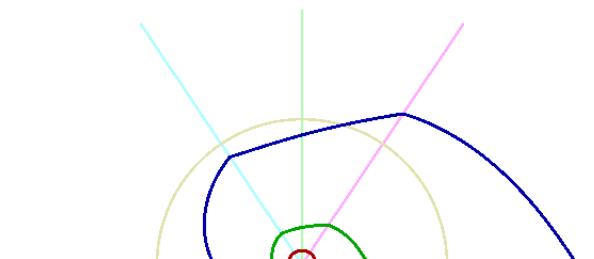
$$f = \frac{\rho_d}{\pi} + \frac{\rho_s}{8 * \pi} * \frac{(\vec{n} \cdot \vec{h})}{(\vec{\omega}_o \cdot \vec{h}) * \max((\vec{n} \cdot \vec{\omega}_i), (\vec{n} \cdot \vec{\omega}_o))} \quad (3)$$

6.6.2 Visualização do Resultado

Figura 39 – *Plots* da BRDF deste experimento.

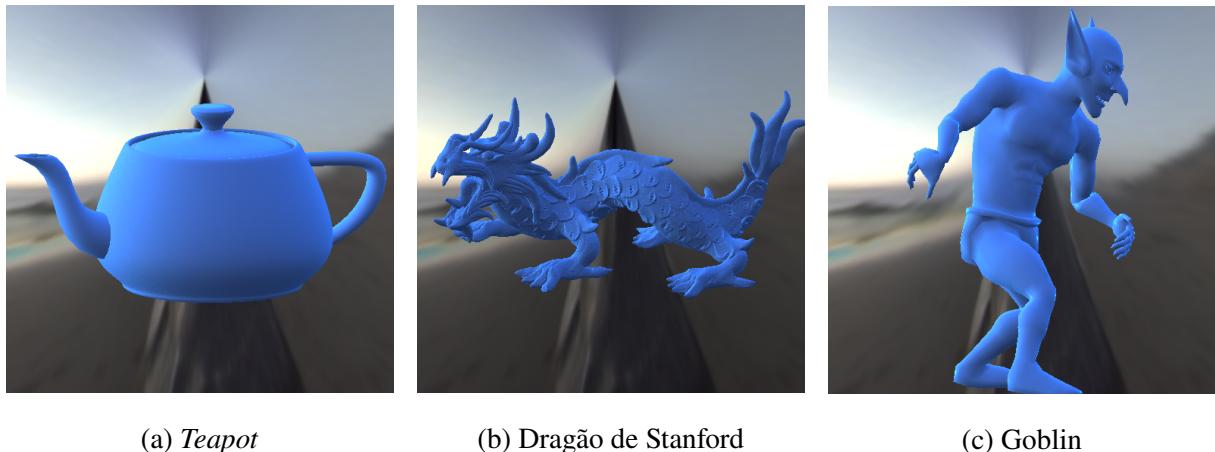


(a) 3D plot



(b) Polar plot

Figura 40 – Objetos 3D renderizados por este experimento

(a) *Teapot*

(b) Dragão de Stanford

(c) Goblin

6.6.3 Código GLSL Gerado

Código 63 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19
20 //////////// START OF USER DECLARED ///////////
21 vec3 var_12_rho_d;
22 vec3 var_13_rho_s;
23 vec3 var_14_f;
24 //////////// END OF USER DECLARED ///////////

```

Código 64 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 ////////////// START FUNCTIONS DECLARATIONS /////////////
2 ////////////// END FUNCTIONS DECLARATIONS /////////////
3
4 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
5
6     ////////////// START OF BUILTINS INITIALIZATION ///////////
7     var_0_vec_h = normalize(L + V);
8     var_3_vec_n = normalize(N);
9     var_1_pi = 3.141592653589793;
10    var_2_epsilon = 1.192092896e-07;
11    var_4_vec_omega_i = L;
12    var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
13    var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
14                           var_4_vec_omega_i.x * var_4_vec_omega_i.x),
15                           var_4_vec_omega_i.z);
16    var_7_vec_omega_o = V;
17    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
18    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
19                           var_7_vec_omega_o.x * var_7_vec_omega_o.x),
20                           var_7_vec_omega_o.z);
21    var_10_theta_h = acos(dot(var_0_vec_h, N));
22    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
23    ////////////// END OF BUILTINS INITIALIZATION ///////////
24
25    var_12_rho_d = vec3(0.3, 0.3, 0.3);
26    var_13_rho_s = (vec3(0.0, 0.2, 1.0) * 20.0);
27    var_14_f = ((var_12_rho_d / var_1_pi) +
28                 ((var_13_rho_s / (8.0 * var_1_pi)) *
29                  ((dot(var_3_vec_n, var_0_vec_h)) /
30                   ((dot(var_7_vec_omega_o, var_0_vec_h)) *
31                    max((dot(var_3_vec_n, var_4_vec_omega_i)),
32                         (dot(var_3_vec_n, var_7_vec_omega_o)))))));
33
34    return vec3(var_14_f);
35 }
```

6.6.4 Código Fonte em EquationLang

Código 65 – Código fonte da BRDF deste experimento.

```

1 \begin{document}
2
3
4 \begin{equation}
5     \rho_d = \vec{0.3, 0.3, 0.3}
6 \end{equation}
7
8 \begin{equation}
9     \rho_s = \vec{0.0, 0.2, 1.0}^2
10 \end{equation}
11
12 \begin{equation}
13 f = \frac{\rho_d}{\pi} + \frac{\rho_s}{8*\pi} *
14 \frac{(\vec{n} \cdot \vec{h})}{(\vec{n} \cdot \vec{o}) \cdot (\vec{n} \cdot \vec{h})} *
15 \max((\vec{n} \cdot \vec{o}), (\vec{n} \cdot \vec{o}))
16
17 \end{equation}
18 \end{document}
```

6.7 Experimento BRDF Cook-Torrance Alternativa

Este experimento é uma versão alternativa da feita na [Seção 6.2](#). As equações na [Figura 41](#) fazem uso da equação do Efeito Fresnel e foram escolhidas constantes diferentes. Código fonte em EquationLang pode ser visto no [Código 68](#). Os códigos gerados em GLSL estão no [Código 66](#) e [Código 67](#). Objetos 3D renderizados pode ser vistos na [Figura 43](#) e os plots estão na [Figura 42](#).

6.7.1 Representação em documento L^AT_EX

Figura 41 – Equações da BRDF do experimento cook-torrance-alternative em documento L^AT_EX.

$$m = 0.3 \quad (1)$$

$$f_0 = 0.4 \quad (2)$$

$$Beckmann(m, t) = \exp((t * t - 1)/(m * m * t * t)) / (m * m * t * t * t) \quad (3)$$

$$Fresnel(f_0, u) = f_0 + (1 - f_0) * ((1 - u)^5) \quad (4)$$

$$H = \vec{h} \quad (5)$$

$$V = \vec{\omega}_o \quad (6)$$

$$L = \vec{\omega}_i \quad (7)$$

$$N = \vec{n} \quad (8)$$

$$D = Beckmann(m, (N \cdot H)) \quad (9)$$

$$F = Fresnel(f_0, (V \cdot H)) \quad (10)$$

$$G = 1/(N \cdot V) \quad (11)$$

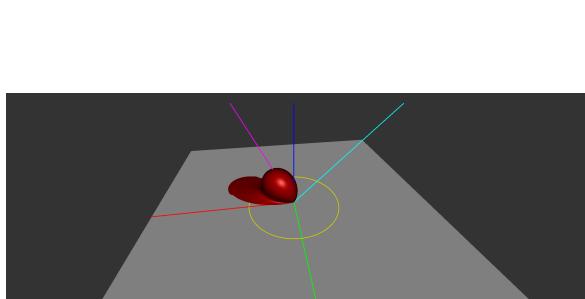
$$val = \max(D * G, 0.0) \cdot F \quad (12)$$

$$color = 1, 0.5, 1 \quad (13)$$

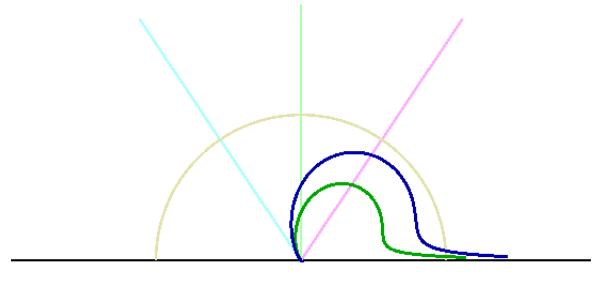
$$f = color * val / (N \cdot L) \quad (14)$$

6.7.2 Visualização do Resultado

Figura 42 – Plots da BRDF deste experimento.

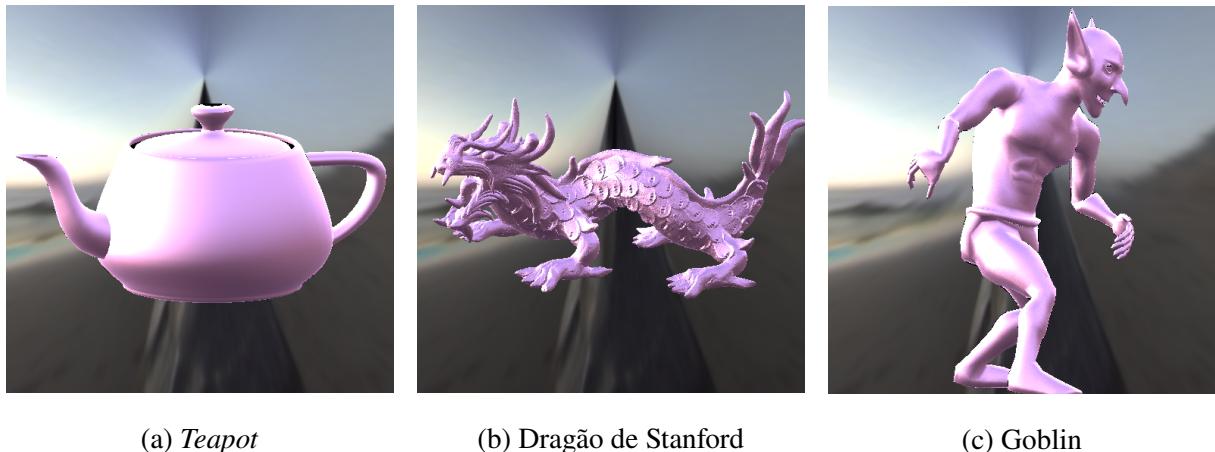


(a) 3D plot



(b) Polar plot

Figura 43 – Objetos 3D renderizados por este experimento

(a) *Teapot*

(b) Dragão de Stanford

(c) Goblin

6.7.3 Código GLSL Gerado

Código 66 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19 //////////// START OF USER DECLARED ///////////
20 vec3 var_12_V;
21 vec3 var_13_L;
22 float var_17_f_0;
23 float var_15_m;
24 vec3 var_20_H;
25 vec3 var_21_text_color;
26 vec3 var_22_N;
27 float var_23_D;
28 float var_24_G;
29 float var_25_F;
30 float var_26_val;
31 vec3 var_27_f;
```

Código 67 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 ////////////// END OF USER DECLARED /////////////
2 ////////////// START FUNCTIONS DECLARATIONS /////////////
3 float var_14_Beckmann(float var_15_m, float var_16_t) {
4     return (exp((((var_16_t * var_16_t) - 1.0)) /
5                 (((var_15_m * var_15_m) * var_16_t) * var_16_t)))) /
6         (((((var_15_m * var_15_m) * var_16_t) * var_16_t) *
7             var_16_t) *
8             var_16_t));
9 }
10 float var_18_Fresnel(float var_17_f_0, float var_19_u) {
11     return (var_17_f_0 + ((1.0 - var_17_f_0)) * (pow(((1.0 -
12         var_19_u)), 5.0)));
13 }
14 ////////////// END FUNCTIONS DECLARATIONS /////////////
15 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
16     ////////////// START OF BUILTINS INITIALIZATION ///////////
17     var_0_vec_h = normalize(L + V);
18     var_3_vec_n = normalize(N);
19     var_1_pi = 3.141592653589793;
20     var_2_epsilon = 1.192092896e-07;
21     var_4_vec_omega_i = L;
22     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
23     var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
24                           var_4_vec_omega_i.x * var_4_vec_omega_i.x),
25                           var_4_vec_omega_i.z);
26     var_7_vec_omega_o = V;
27     var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
28     var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
29                           var_7_vec_omega_o.x * var_7_vec_omega_o.x),
30                           var_7_vec_omega_o.z);
31     var_10_theta_h = acos(dot(var_0_vec_h, N));
32     var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
33     ////////////// END OF BUILTINS INITIALIZATION ///////////
34
35     var_12_V = var_7_vec_omega_o;
36     var_13_L = var_4_vec_omega_i;
37     var_17_f_0 = 0.4;
38     var_15_m = 0.3;
39     var_20_H = var_0_vec_h;
40     var_21_text_color = vec3(1.0, 0.5, 1.0);
41     var_22_N = var_3_vec_n;
42     var_23_D = var_14_Beckmann(var_15_m, (dot(var_22_N, var_20_H)));
43     var_24_G = (1.0 / (dot(var_22_N, var_12_V)));
44     var_25_F = var_18_Fresnel(var_17_f_0, (dot(var_12_V, var_20_H)));
45     var_26_val = (max((var_23_D * var_24_G), 0.0) * var_25_F);
46     var_27_f = ((var_21_text_color * var_26_val) / (dot(var_22_N,
47             var_13_L)));
48
49     return vec3(var_27_f);
50 }
```

6.7.4 Código Fonte em EquationLang

Código 68 – Código fonte da BRDF deste experimento.

```

1 \begin{equation}
2     m = 0.3
3 \end{equation}
4 \begin{equation}
5     f_0 = 0.4
6 \end{equation}
7 \begin{equation}
8 Beckmann(m, t) = \exp((t*t-1)/(m*m*t*t)) / (m*m*t*t*t*t)
9 \end{equation}
10 \begin{equation}
11     Fresnel(f_0, u) = f_0 + (1 - f_0) * ((1-u)^ 5)
12 \end{equation}
13 \begin{equation}
14     H = \vec{h}
15 \end{equation}
16 \begin{equation}
17     V = \vec{\omega_o}
18 \end{equation}
19 \begin{equation}
20     L = \vec{\omega_i}
21 \end{equation}
22 \begin{equation}
23     N = \vec{n}
24 \end{equation}
25 \begin{equation}
26     D = Beckmann(m, (N \cdot H) )
27 \end{equation}
28 \begin{equation}
29     F = Fresnel(f_0, (V \cdot H) )
30 \end{equation}
31 \begin{equation}
32     G = 1/(N \cdot V)
33 \end{equation}
34 \begin{equation}
35     val = \max(D * G, 0.0) \cdot F
36 \end{equation}
37 \begin{equation}
38     \text{color} = \vec{1,0.5,1}
39 \end{equation}
40 \begin{equation}
41     f = \text{color} * val / (N \cdot L)
42 \end{equation}
```

6.8 Experimento BRDF Dür

No artigo de Geisler-Moroder e Dür (GEISLER-MORODER; DÜR, 2010), é discutido sobre as limitações do modelo de reflexão de Ward, onde é proposto uma abordagem para restringir o albedo e garantir a conservação de energia. Este experimento é baseado nessa BRDF com albedo restrinido. As equações são apresentadas [Figura 44](#), com o código fonte em EquationLang disponível no [Código 71](#). Os códigos gerados em GLSL pode ser vistos no [Código 69](#) e [Código 70](#). A renderização dos objetos 3D pode ser observada na [Figura 46](#) e os *plots* na [Figura 45](#).

6.8.1 Representação em documento L^AT_EX

Figura 44 – Equações da BRDF do experimento duer em documento L^AT_EX.

Duer 2010 Bounding the Albedo of the Ward Reflectance Model

$$G = ((\vec{\omega}_i + \vec{\omega}_o) \cdot (\vec{\omega}_i + \vec{\omega}_o)) * ((\vec{\omega}_i + \vec{\omega}_o) \cdot \vec{n})^{-4} * (\vec{n} \cdot \vec{\omega}_i) * (\vec{n} \cdot \vec{\omega}_o) \quad (1)$$

$$f = G \quad (2)$$

6.8.2 Visualização do Resultado

Figura 45 – *Plots* da BRDF deste experimento.

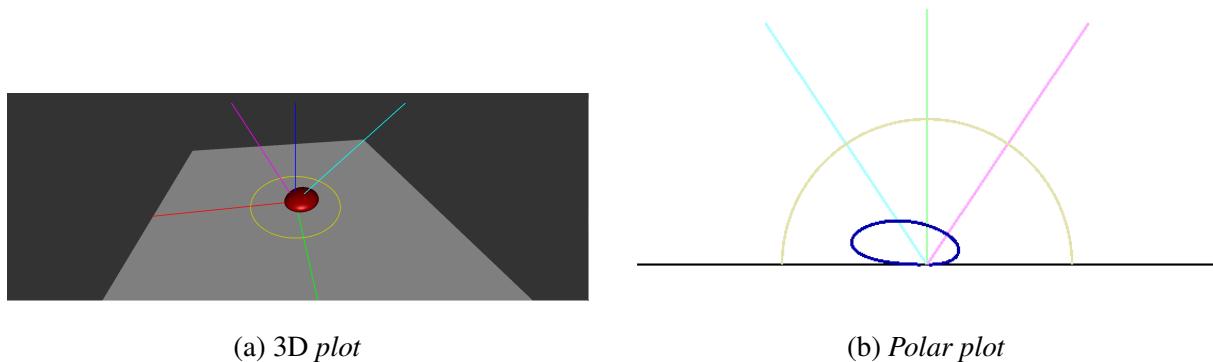
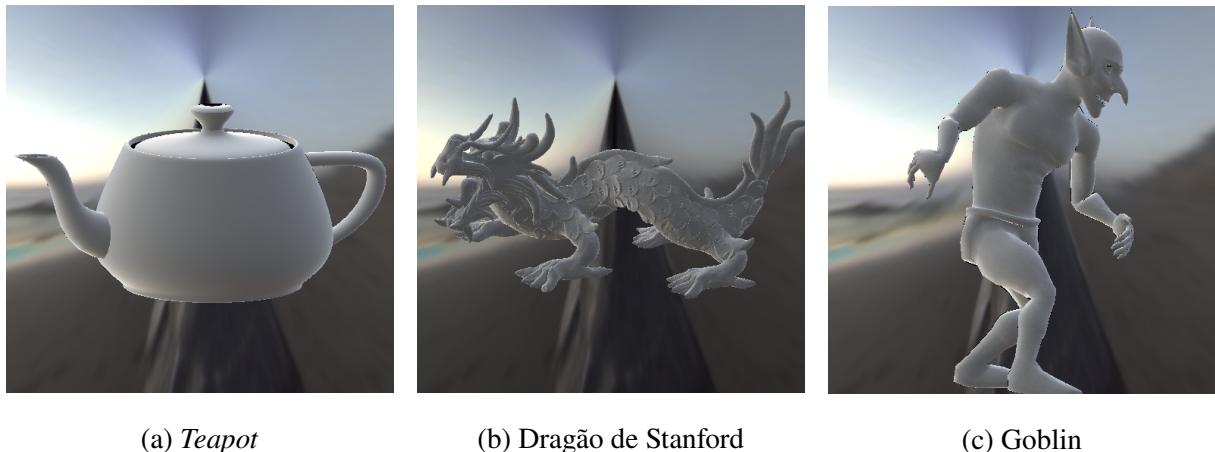


Figura 46 – Objetos 3D renderizados por este experimento

(a) *Teapot*

(b) Dragão de Stanford

(c) Goblin

6.8.3 Código GLSL Gerado

Código 69 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19 //////////// START OF USER DECLARED ///////////
20 float var_12_G;
21 float var_13_f;
22 //////////// END OF USER DECLARED ///////////
23 //////////// START FUNCTIONS DECLARATIONS ///////////
24 //////////// END FUNCTIONS DECLARATIONS ///////////

```

Código 70 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
2
3     ////////////////// START OF BUILTINS INITIALIZATION //////////////////
4     var_0_vec_h = normalize(L + V);
5     var_3_vec_n = normalize(N);
6     var_1_pi = 3.141592653589793;
7     var_2_epsilon = 1.192092896e-07;
8     var_4_vec_omega_i = L;
9     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
10    var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
11                           var_4_vec_omega_i.x * var_4_vec_omega_i.x),
12                           var_4_vec_omega_i.z);
13    var_7_vec_omega_o = V;
14    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
15    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
16                           var_7_vec_omega_o.x * var_7_vec_omega_o.x),
17                           var_7_vec_omega_o.z);
18    var_10_theta_h = acos(dot(var_0_vec_h, N));
19    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
20    ////////////////// END OF BUILTINS INITIALIZATION //////////////////
21
22    var_12_G =
23        (((dot(((var_4_vec_omega_i + var_7_vec_omega_o)),
24                  ((var_4_vec_omega_i + var_7_vec_omega_o)))) *
25         pow((dot(((var_4_vec_omega_i + var_7_vec_omega_o)),
26                  var_3_vec_n)),
27                  (-4.0))) *
28         (dot(var_3_vec_n, var_4_vec_omega_i))) *
29         (dot(var_3_vec_n, var_7_vec_omega_o)));
30    var_13_f = var_12_G;
31
32    return vec3(var_13_f);
33 }
```

6.8.4 Código Fonte em EquationLang

Código 71 – Código fonte da BRDF deste experimento.

```

1 Duer 2010 Bounding the Albedo of the Ward Reflectance Model
2
3 \begin{equation}
4 G = ((\vec{\omega}_i)+\vec{\omega}_o)
5 \cdot (\vec{\omega}_i+\vec{\omega}_o)) *
6 ((\vec{\omega}_i)+\vec{\omega}_o) \cdot \vec{n})^{-4}
7 * (\vec{n} \cdot \vec{\omega}_i) * (\vec{n} \cdot \vec{\omega}_o)
8 \end{equation}
9 f = G
10 \end{equation}
```

6.9 Experimento BRDF Edwards 2006

No artigo de Edwards et al. ([EDWARDS et al., 2006](#)), é apresentado o conceito do *Halfway Vector Disk* como uma extensão para modelagem de BRDFs. Este método, usado neste experimento, propõe uma abordagem geométrica que melhora a eficiência computacional. As equações principais são descritas na [Figura 47](#), com o código fonte em EquationLang disponível no [Código 74](#). Os códigos gerados em GLSL podem ser vistos no [Código 72](#) e [Código 73](#). A renderização de objetos 3D utilizando o método pode ser observada na [Figura 49](#), enquanto os *plots* estão ilustrados na [Figura 48](#).

6.9.1 Representação em documento L^AT_EX

Figura 47 – Equações da BRDF do experimento edwards-2006 em documento L^AT_EX.

Edwards halfway-vector disk (2006)

$$n = 10 \quad (1)$$

$$R = 1 \quad (2)$$

$$\text{lump}(\vec{h}, R, n) = (n + 1)/(\pi * R * R) * (1 - (\vec{h} \cdot \vec{h})/(R * R)^n) \quad (3)$$

Scaling projection

$$uH = \vec{\omega}_i + \vec{\omega}_o \quad (4)$$

$$h = (\vec{n} \cdot \vec{\omega}_o)/(\vec{n} \cdot uH) * uH \quad (5)$$

$$huv = h - (\vec{n} \cdot \vec{\omega}_o) * \vec{n} \quad (6)$$

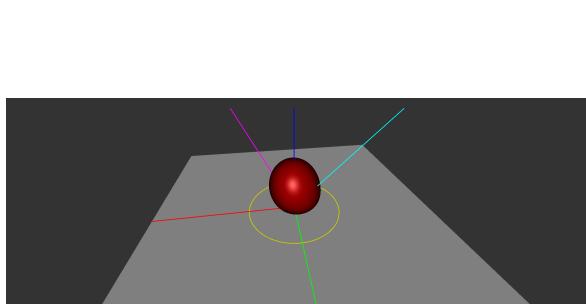
Specular term (D and G)

$$p = \text{lump}(huv, R, n) \quad (7)$$

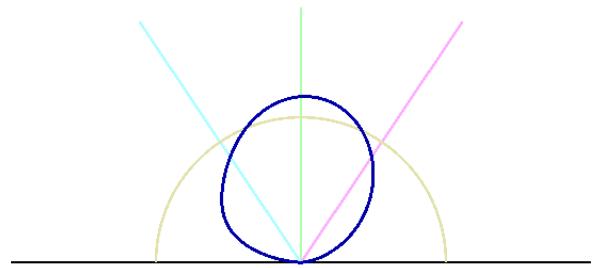
$$f = p * ((\vec{n} \cdot \vec{\omega}_o)^2)/(4 * (\vec{n} \cdot \vec{\omega}_i) * (\vec{\omega}_i \cdot \vec{h}) * ((\vec{n} \cdot \vec{h})^3)) \quad (8)$$

6.9.2 Visualização do Resultado

Figura 48 – Plots da BRDF deste experimento.

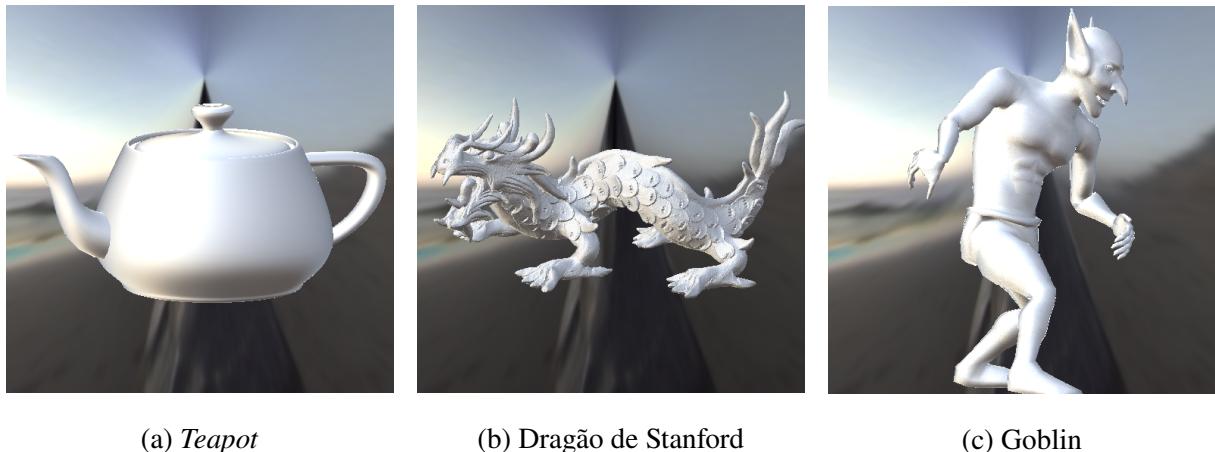


(a) 3D plot



(b) Polar plot

Figura 49 – Objetos 3D renderizados por este experimento

(a) *Teapot*

(b) Dragão de Stanford

(c) Goblin

6.9.3 Código GLSL Gerado

Código 72 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19
20 //////////// START OF USER DECLARED ///////////
21 vec3 var_15_uH;
22 vec3 var_16_h;
23 float var_14_n;
24 vec3 var_17_huv;
25 float var_13_R;
26 float var_18_p;
27 float var_19_f;
```

Código 73 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```

1 ////////////// END OF USER DECLARED /////////////
2
3 ////////////// START FUNCTIONS DECLARATIONS /////////////
4 float var_12_text_lump(vec3 var_0_vec_h, float var_13_R, float
5     var_14_n) {
6     return (((var_14_n + 1.0)) / (((var_1_pi * var_13_R) *
7         var_13_R))) *
8         ((1.0 - ((dot(var_0_vec_h, var_0_vec_h)) /
9             pow(((var_13_R * var_13_R)), var_14_n))))));
10 }
11 ////////////// END FUNCTIONS DECLARATIONS /////////////
12 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
13
14     ////////////// START OF BUILTINS INITIALIZATION /////////////
15     var_0_vec_h = normalize(L + V);
16     var_3_vec_n = normalize(N);
17     var_1_pi = 3.141592653589793;
18     var_2_epsilon = 1.192092896e-07;
19     var_4_vec_omega_i = L;
20     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
21     var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
22         var_4_vec_omega_i.x * var_4_vec_omega_i.x),
23         var_4_vec_omega_i.z);
24     var_7_vec_omega_o = V;
25     var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
26     var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
27         var_7_vec_omega_o.x * var_7_vec_omega_o.x),
28         var_7_vec_omega_o.z);
29     var_10_theta_h = acos(dot(var_0_vec_h, N));
30     var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
31     ////////////// END OF BUILTINS INITIALIZATION /////////////
32
33     var_15_uH = (var_4_vec_omega_i + var_7_vec_omega_o);
34     var_16_h =
35         (((dot(var_3_vec_n, var_7_vec_omega_o)) / (dot(var_3_vec_n,
36             var_15_uH))) *
37             var_15_uH);
38     var_14_n = 10.0;
39     var_17_huv =
40         (var_16_h - ((dot(var_3_vec_n, var_7_vec_omega_o)) *
41             var_3_vec_n));
42     var_13_R = 1.0;
43     var_18_p = var_12_text_lump(var_17_huv, var_13_R, var_14_n);
44     var_19_f = ((var_18_p * (pow((dot(var_3_vec_n,
45             var_7_vec_omega_o)), 2.0))) /
46         (((4.0 * (dot(var_3_vec_n, var_4_vec_omega_i))) *
47             (dot(var_4_vec_omega_i, var_0_vec_h))) *
48             (pow((dot(var_3_vec_n, var_0_vec_h)), 3.0))));
```

6.9.4 Código Fonte em EquationLang

Código 74 – Código fonte da BRDF deste experimento.

```

1 \begin{equation}
2 n = 10
3 \end{equation}
4
5 \begin{equation}
6 R = 1
7 \end{equation}
8
9 \begin{equation}
10 \text{lump}(\vec{h}, R, n) = (n+1)/(\pi * R * R) * (1 - (\vec{h} \cdot
    \vec{h})/(R * R)^n)
11 \end{equation}
12
13 Scaling projection
14 \begin{equation}
15 uH = \vec{\omega_i} + \vec{\omega_o} \% // unnormalized H
16 \end{equation}
17
18 \begin{equation}
19 h = (\vec{n} \cdot \vec{\omega_o}) / (\vec{n} \cdot uH) * uH
20 \end{equation}
21
22 \begin{equation}
23 huv = h - (\vec{n} \cdot \vec{\omega_o}) * \vec{n}
24 \end{equation}
25
26 Specular term (D and G)
27
28 \begin{equation}
29 p = \text{lump}(huv, R, n)
30 \end{equation}
31
32 \begin{equation}
33 f = p * ((\vec{n} \cdot \vec{\omega_o})^2)
34 / (4 * (\vec{n} \cdot \vec{\omega_i}) * (\vec{\omega_i} \cdot
    \vec{h}))
35 * ((\vec{n} \cdot \vec{h})^3))
36 \end{equation}
```

6.10 Experimento BRDF Anisotrópica baseado em Kajiya-Kay (1989)

Este experimento é baseado no modelo anisotrópico que descreve o comportamento de reflexão de superfícies rugosas simplificadas proposto no trabalho de Kajiya (KAJIYA,

1985). As equações e parâmetros escolhidos que descrevem esse modelo estão em [Figura 50](#). O código fonte em [EquationLang](#) para o compilador está em [Código 77](#). A saída gerada pelo compilador está dividido em duas partes: a parte 1 está no [Código 75](#), enquanto a parte 2 está em [Código 76](#). A renderização dos objetos 3D usando essa BRDF se encontra em [Figura 52](#). Utilizamos interpolação linear para aproximar valores calculados previamente na tabela de refletância, com plots logarítmicos e polares presentes na [Figura 51](#).

6.10.1 Representação em documento L^AT_EX

Figura 50 – Equações da BRDF do experimento Kajiya-Kay em documento L^AT_EX.

Based on Kajiya-Kay 1989

$$\text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}} \quad (1)$$

Tangent vector:

$$X = \text{normalize}(0, \vec{1}, 0 \times \vec{n}) \quad (2)$$

Bitangent vector:

$$Y = \text{normalize}(\vec{n} \times X) \quad (3)$$

$$T = Y \quad (4)$$

$$L = \vec{\omega}_i \quad (5)$$

$$\text{roughness} = 0.1 \quad (6)$$

$$\text{glossiness} = (1/\text{roughness}) \quad (7)$$

$$s_\alpha = \sqrt{(1 - ((\vec{\omega}_i \cdot T) * (\vec{\omega}_i \cdot T)))} \quad (8)$$

$$\text{spec} = ((s_\alpha \cdot \sqrt{(1 - ((\vec{\omega}_o \cdot T) \cdot (\vec{\omega}_o \cdot T))))}) - ((\vec{\omega}_i \cdot T) \cdot (\vec{\omega}_o \cdot T)))^{\text{glossiness}} \quad (9)$$

$$f = \text{spec} \quad (10)$$

6.10.2 Visualização do Resultado

Figura 51 – *Plots* da BRDF deste experimento.

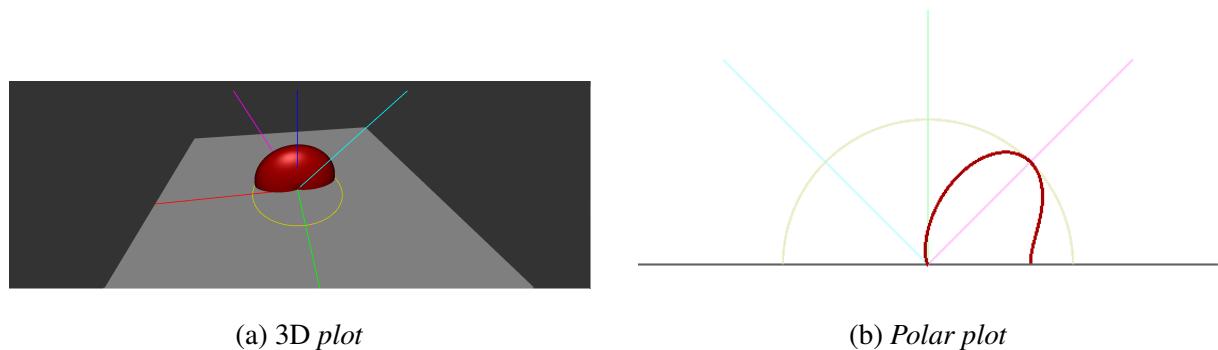


Figura 52 – Objetos 3D renderizado no experimento BRDF Anisotrópica: Kajiya-Kay (1989)



6.10.3 Código GLSL Gerado

Código 75 – Saída do compilador, código GLSL da BRDF do experimento baseado em Kajiya-Kay (parte 1).

```
1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 ////////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 ////////////// END OF BUILTINS DECLARTION ///////////
19
20 ////////////// START OF USER DECLARED ///////////
21 vec3 var_12_L;
22 vec3 var_15_X;
23 vec3 var_16_Y;
24 vec3 var_17_T;
25 float var_18_s_alpha;
26 float var_19_text_roughness;
27 float var_20_text_glossiness;
28 float var_21_text_spec;
29 float var_22_f;
30 ////////////// END OF USER DECLARED ///////////
31 ////////////// START FUNCTIONS DECLARATIONS ///////////
32 vec3 var_13_text_normalize(vec3 var_14_vec_u) {
33     return (var_14_vec_u / sqrt(dot(var_14_vec_u, var_14_vec_u)));
34 }
35 ////////////// END FUNCTIONS DECLARATIONS ///////////
```

Código 76 – Saída do compilador, código GLSL da BRDF do experimento baseado em Kajiya-Kay (parte 2).

```

1 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
2     ////////////// START OF BUILTINS INITIALIZATION ///////////
3     var_0_vec_h = normalize(L + V);
4     var_3_vec_n = normalize(N);
5     var_1_pi = 3.141592653589793;
6     var_2_epsilon = 1.192092896e-07;
7     var_4_vec_omega_i = L;
8     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
9     var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
10                         var_4_vec_omega_i.x * var_4_vec_omega_i.x),
11                         var_4_vec_omega_i.z);
12    var_7_vec_omega_o = V;
13    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
14    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
15                         var_7_vec_omega_o.x * var_7_vec_omega_o.x),
16                         var_7_vec_omega_o.z);
17    var_10_theta_h = acos(dot(var_0_vec_h, N));
18    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
19    ////////////// END OF BUILTINS INITIALIZATION ///////////
20    var_12_L = var_4_vec_omega_i;
21    var_15_X = var_13_text_normalize(cross(vec3(0.0, 1.0, 0.0),
22                                         var_3_vec_n));
23    var_16_Y = var_13_text_normalize(cross(var_3_vec_n, var_15_X));
24    var_17_T = var_16_Y;
25    var_18_s_alpha = sqrt(((1.0 - (((dot(var_4_vec_omega_i, var_17_T)) *
26                                         (dot(var_4_vec_omega_i,
27                                         var_17_T)))))));
28    var_19_text_roughness = 0.1;
29    var_20_text_glossiness = ((1.0 / var_19_text_roughness));
30    var_21_text_spec =
31        pow((((var_18_s_alpha *
32            sqrt(((1.0 - (((dot(var_7_vec_omega_o, var_17_T)) *
33                (dot(var_7_vec_omega_o,
34                var_17_T))))))) -
35            (((dot(var_4_vec_omega_i, var_17_T)) *
36                (dot(var_7_vec_omega_o, var_17_T)))))),
37        var_20_text_glossiness);
38    var_22_f = var_21_text_spec;
39
40    return vec3(var_22_f);
41 }
```

6.10.4 Código Fonte em EquationLang

Código 77 – Código fonte da BRDF do experimento Kajiya-Kay.

```

1 Based on Kajiya-Kay 1989
2
3 \begin{equation}
4   \text{normalize}(\vec{u}) = \frac{\vec{u}}{\sqrt{\vec{u} \cdot \vec{u}}}
5 \end{equation}
6
7 Tangent vector:
8 \begin{equation}
9   X = \text{normalize}(\vec{0}, 1, 0) \times \vec{n}
10 \end{equation}
11
12 Bitangent vector:
13 \begin{equation}
14   Y = \text{normalize}(\vec{n} \times X)
15 \end{equation}
16
17 \begin{equation}
18   T = Y
19 \end{equation}
20
21 \begin{equation}
22   L = \vec{\omega}_i
23 \end{equation}
24
25 \begin{equation}
26   \text{roughness} = 0.1
27 \end{equation}
28
29 \begin{equation}
30   \text{glossiness} = (1/\text{roughness})
31 \end{equation}
32
33 \begin{equation}
34   s_\alpha = \sqrt{(1 - ((\vec{\omega}_i \cdot T) * (\vec{\omega}_i \cdot T)))}
35 \end{equation}
36
37 \begin{equation}
38   \text{spec} = (s_\alpha \cdot \sqrt{1 - ((\vec{\omega}_o \cdot T) * (\vec{\omega}_o \cdot T))})
39     - ((\vec{\omega}_i \cdot T) * (\vec{\omega}_o \cdot T)))^\text{glossiness}
40 \end{equation}
41 \begin{equation}
42   f = \text{spec}
43 \end{equation}

```

6.11 Experimento BRDF Minnaert

Esse experimento foi realizado seguindo os princípios do artigo de Minnaert (THE..., 1941). Nele é apresentado um modelo de reflexão que introduz uma abordagem para descrever superfícies que exibem comportamentos encontrado em superfícies porosas, como a lua. As equações desse experimento estão em [Figura 53](#). O código fonte se encontra no [Código 80](#). O GLSL gerado pode ser encontrada no [Código 78](#) e [Código 79](#), enquanto os resultados de renderização podem ser observados na [Figura 55](#) e os *plots* na [Figura 54](#).

6.11.1 Representação em documento L^AT_EX

Figura 53 – Equações da BRDF do experimento minnaert em documento L^AT_EX.

[Min41] MINNAERT M.: The reciprocity principle in lunar photometry.
Astrophysical Journal, 3 (1941), 403– 410. 10

ω_o : This is the outgoing (view) direction vector (often normalized). $\cos \omega_i$ and $\cos \omega_o$: These are actually shorthand notations.

They don't mean the cosine of the entire vector, but rather: $\cos \omega_i$ actually means $\cos(\theta_i) = (\vec{\omega}_i, n)$ $\cos \omega_o$ actually means $\cos(\theta_o) = (\vec{\omega}_o, n)$

Where:

θ_i is the angle between ω_i and the surface normal n .

θ_o is the angle between ω_o and the surface normal n .

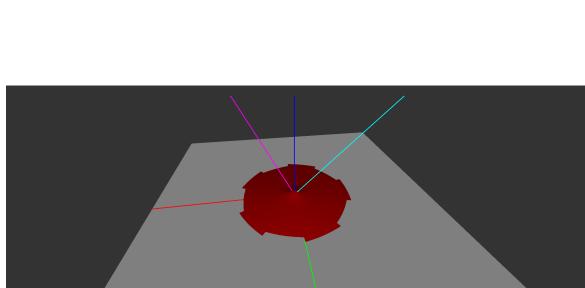
$$\rho_d = 0.3, 0.05, 0.05 \quad (1)$$

$$k = 0.5 \quad (2)$$

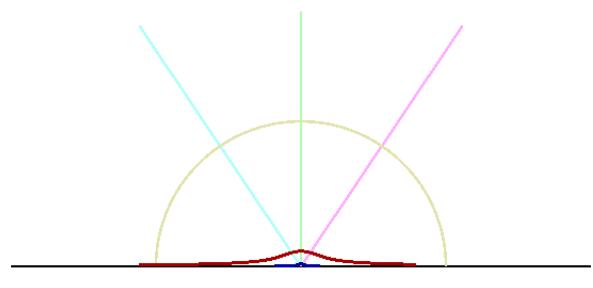
$$f = \frac{\rho_d}{\pi} * ((\vec{n} \cdot \vec{\omega}_i) * (\vec{n} \cdot \vec{\omega}_o))^{(k-1)} \quad (3)$$

6.11.2 Visualização do Resultado

Figura 54 – *Plots* da BRDF deste experimento.

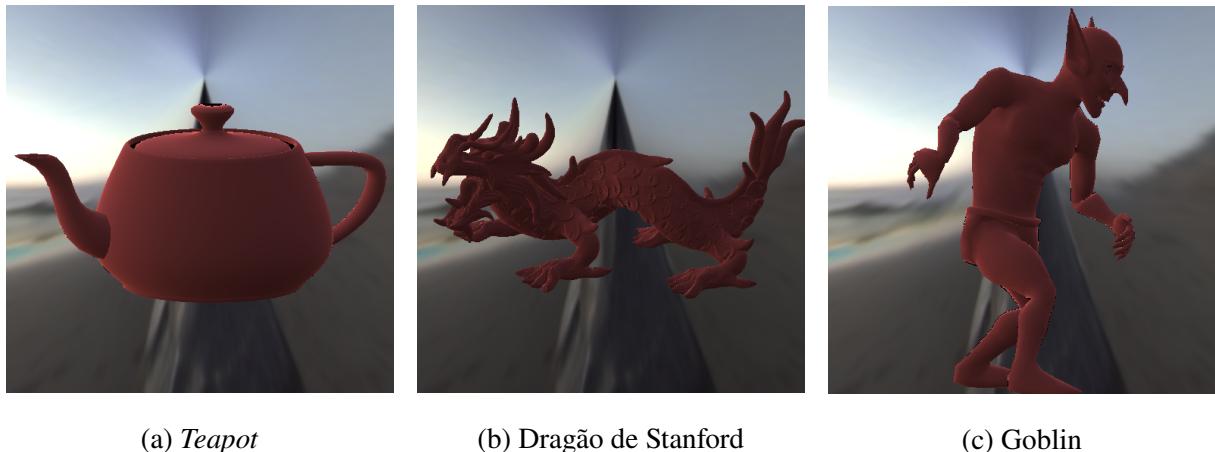


(a) 3D plot



(b) Polar plot

Figura 55 – Objetos 3D renderizados por este experimento

(a) *Teapot*

(b) Dragão de Stanford

(c) Goblin

6.11.3 Código GLSL Gerado

Código 78 – Saída do compilador, código GLSL da BRDF deste experimento (parte 1).

```

1 analytic ::begin parameters
2 #[type][name][min val][max val][default val]
3 ::end parameters
4 ::begin shader
5 //////////// START OF BUILTINS DECLARTION ///////////
6 vec3 var_0_vec_h;
7 vec3 var_3_vec_n;
8 float var_10_theta_h;
9 float var_11_theta_d;
10 float var_1_pi;
11 float var_2_epsilon;
12 vec3 var_4_vec_omega_i;
13 float var_5_theta_i;
14 float var_6_phi_i;
15 vec3 var_7_vec_omega_o;
16 float var_8_theta_o;
17 float var_9_phi_o;
18 //////////// END OF BUILTINS DECLARTION ///////////
19
20 //////////// START OF USER DECLARED ///////////
21 vec3 var_12_rho_d;
22 float var_13_k;
23 vec3 var_14_f;
24 //////////// END OF USER DECLARED ///////////
25
26 //////////// START FUNCTIONS DECLARATIONS ///////////
27 //////////// END FUNCTIONS DECLARATIONS ///////////

```

Código 79 – Saída do compilador, código GLSL da BRDF deste experimento (parte 2).

```
1 vec3 BRDF(vec3 L, vec3 V, vec3 N, vec3 X, vec3 Y) {
2
3     ////////////// START OF BUILTINS INITIALIZATION ///////////
4     var_0_vec_h = normalize(L + V);
5     var_3_vec_n = normalize(N);
6     var_1_pi = 3.141592653589793;
7     var_2_epsilon = 1.192092896e-07;
8     var_4_vec_omega_i = L;
9     var_5_theta_i = atan(var_4_vec_omega_i.y, var_4_vec_omega_i.x);
10    var_6_phi_i = atan(sqrt(var_4_vec_omega_i.y * var_4_vec_omega_i.y +
11                           var_4_vec_omega_i.x * var_4_vec_omega_i.x),
12                           var_4_vec_omega_i.z);
13    var_7_vec_omega_o = V;
14    var_8_theta_o = atan(var_7_vec_omega_o.y, var_7_vec_omega_o.x);
15    var_9_phi_o = atan(sqrt(var_7_vec_omega_o.y * var_7_vec_omega_o.y +
16                           var_7_vec_omega_o.x * var_7_vec_omega_o.x),
17                           var_7_vec_omega_o.z);
18    var_10_theta_h = acos(dot(var_0_vec_h, N));
19    var_11_theta_d = acos(dot(var_0_vec_h, var_4_vec_omega_i));
20    ////////////// END OF BUILTINS INITIALIZATION ///////////
21
22    var_12_rho_d = vec3(0.3, 0.05, 0.05);
23    var_13_k = 0.5;
24    var_14_f = ((var_12_rho_d / var_1_pi) *
25                 pow(((dot(var_3_vec_n, var_4_vec_omega_i)) *
26                      (dot(var_3_vec_n, var_7_vec_omega_o)))), +
27                 ((var_13_k - 1.0))));
28
29    return vec3(var_14_f);
30 }
```

6.11.4 Código Fonte em EquationLang

Código 80 – Código fonte da BRDF deste experimento (parte 1).

```

1 [Min41] MINNAERT M.: The reciprocity principle in lunar photometry.
          Astrophysical Journal, 3 (1941), 403- 410. 10
2
3 $\omega_o$: This is the outgoing (view) direction vector (often
   normalized).
4 $\cos\omega_i$ and $\cos\omega_o$: These are actually shorthand
   notations.
5
6 They don't mean the cosine of the entire vector, but rather:
7 $\cos\omega_i$ actually means $\cos(\theta_i) = \dot(\omega_i, n)$
8 $\cos\omega_o$ actually means $\cos(\theta_o) = \dot(\omega_o, n)$
9
10 Where:
11
12 $\theta_i$ is the angle between $\omega_i$ and the surface normal
   $n$.
13
14 $\theta_o$ is the angle between $\omega_o$ and the surface normal
   $n$.
15
16 \begin{equation}
17   \rho_d = \vec{0.3, 0.05, 0.05}
18 \end{equation}
19
20 \begin{equation}
21   k = 0.5
22 \end{equation}
23
24 \begin{equation}
25   f = \frac{\rho_d}{\pi} * ((\vec{n} \cdot \vec{\omega_i}) * (\vec{n} \cdot \vec{\omega_o}))^{(k-1)}
26 \end{equation}
```

7

Conclusão

Este trabalho alcançou os objetivos propostos de desenvolver um compilador capaz de traduzir funções de distribuição de refletância bidirecional para código de linguagem de *shading*. Para renderizar cenas de alta qualidade, são essenciais os *shaders*, componentes intrínsecos à pipeline gráfica, conforme discutido na [seção 2.3](#). Esses *shaders* podem conter a implementação de BRDFs que determinam a maneira como a luz interage com os materiais. Entretanto, a complexidade das BRDFs em suas equações, bem como a necessidade de conhecimentos na área de programação com *shaders*, tornam desafiadora a tradução de BRDFs para código.

A ferramenta reduz significativamente a barreira técnica que poderia restringir a exploração de efeitos visuais por profissionais fora da área de programação ao fornecer um sistema que transforma um documento \LaTeX contendo equações de BRDF diretamente em um arquivo GLSL, pronto para ser carregado utilizando o visualizador gratuito da Disney Explorer. Essa funcionalidade atende principalmente ao meio acadêmico, onde as BRDFs são frequentemente descritas por equações em \LaTeX , democratizando o acesso à criação de efeitos visuais complexos.

O projeto faz uso de conhecimentos técnicos de áreas multidisciplinares, como gramáticas livres de contexto, geração de código, algoritmos com árvores e grafos, renderização projetiva e por *raytracing*, programação com *shaders*, além de fundamentos teóricos sobre refletância e conceitos de radiometria para implementar os recursos da ferramenta. Esses recursos incluem: mensagens de erro bem estruturadas e informativas; suporte à definições de equação em qualquer ordem; geração de código para diferentes BRDFs diretamente a partir de equações \LaTeX ; integração com a ferramenta Disney para renderização visual das BRDFs; e visualização da árvore sintática gerada como arquivo SVG. Todas as etapas do compilador, apresentadas na arquitetura da [Figura 11](#), foram implementadas, incluindo análise léxica, sintática, semântica e emissão de código.

O compilador mantém consistência em ordem de operadores, permite uso de convenções, como símbolos matemáticos comuns (π, ϵ) e outros símbolos específicos da área, por exemplo,

ω_i e ω_o . Também suporta expressões matemáticas essenciais para simulações de iluminação realista em computação gráfica como produto interno, produto vetorial, funções trigonométricas, exponenciação, entre outras, permitindo cálculos com ângulos e vetores. Esses detalhes são cruciais para representar fielmente o comportamento da luz.

A partir dos experimentos da Capítulo 6, o sistema demonstrou que fornece uma base sólida para a implementação de BRDFs complexas a partir de suas equações. Isso confirma que usuários podem focar na lógica específica de modelagem de reflectância, sem precisar lidar com detalhes técnicos de baixo nível, como aspectos específicos da linguagem de *shading*. Embora o sistema seja funcional e os experimentos tenham sido bem-sucedidos, existem oportunidades para melhorias e expansão. Algumas direções promissoras são indicadas nos seguintes items:

- Ampliar o suporte para construções matemáticas adicionais, como somatórios (Σ) e produtos acumulados (Π);
- Adicionar funcionalidades para definição e cálculo de derivadas e integrais, utilizando algoritmos numéricos para avaliar essas expressões diretamente na linguagem de *shading*;
- Expandir as capacidades para suportar diferentes linguagens de *shading*, como as utilizadas nos motores gráficos Unity¹ e Unreal²;
- Desenvolver um editor de texto L^AT_EX integrado, que permita a compilação e visualização simultâneas da BRDFs.

Ademais, aprimoramentos no tratamento de erros podem proporcionar maior contextualização e clareza, auxiliando os usuários na resolução de problemas. Embora não tenham sido identificadas nas BRDFs exploradas certas construções matemáticas (como notação Π e Σ) ou integrais não analíticas - o que impulsionaria a necessidade de um algoritmo numérico para resolução -, a implementação dessas funcionalidades ampliaria significativamente o potencial do compilador.

As perspectivas futuras deste sistema apontam para uma ferramenta cada vez mais versátil e acessível, com potencial para transformar a forma como desenvolvedores e pesquisadores trabalham com BRDFs. A democratização do acesso a técnicas avançadas de computação gráfica representa uma oportunidade para facilitar e agilizar a modelagem dessas funções em simulações científicas.

¹<<https://unity.com/>>

²<<https://www.unrealengine.com/en-US>>

Referências

- ASHIKHMIN, M.; SHIRLEY, P. An anisotropic phong brdf model. *Journal of graphics tools*, Taylor & Francis, v. 5, n. 2, p. 25–32, 2000. Citado na página 108.
- BLINN, J. F. Models of light reflection for computer synthesized pictures. In: *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*. [S.l.: s.n.], 1977. p. 192–198. Citado na página 95.
- 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 35.
- CEM, Y. *Intro to Graphics 07 - GPU Pipeline*. 2020. Disponível em: <https://youtu.be/UzlnprHSbUw?si=Y0a0Tj7ia-lW_eGC>. Citado na página 24.
- COOK, R. L.; TORRANCE, K. E. A reflectance model for computer graphics. *ACM Transactions on Graphics (ToG)*, ACM New York, NY, USA, v. 1, n. 1, p. 7–24, 1982. Citado na página 98.
- DAVISONPRO. *Criando um jogo em JavaScript*. 2024. Disponível em: <<https://bulldogjob.pl/readme/tworzenie-gry-w-javascript>>. Citado na página 26.
- 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 18.
- EDWARDS, D. et al. The halfway vector disk for brdf modeling. *ACM Transactions on Graphics (TOG)*, ACM New York, NY, USA, v. 25, n. 1, p. 1–18, 2006. Citado na página 133.
- GEISLER-MORODER, D.; DÜR, A. Bounding the albedo of the ward reflectance model. *High Performance Graphics*, Citeseer, v. 4, 2010. Citado na página 130.
- 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 36.
- 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 26 e 27.
- JUDICE, S. F.; GIRALDI, G. A.; KARAM-FILHO, J. *Rendering Equation*. 2019. Citado na página 18.
- KAJIYA, J. T. Anisotropic reflectance models. In: *Computers Graphics, ACM Siggraph '85 Conference Proceedings*. [S.l.]: ACM, 1985. v. 4, p. 15–21. Citado na página 137.
- 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 20.

- LESLIE, L. *Latex: A Document Preparation System, User's Guide and Reference Manual*. [S.l.]: Addison-Wesley, 1994. Citado na página 14.
- MONTES, R.; UREÑA, C. An overview of BRDF models. *University of Grenada, Technical Report LSI-2012-001*, 2012. Citado na página 21.
- 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 na página 37.
- OpenGL Architecture Review Board. *OpenGL 4.6 Core Specification*. [S.l.], 2017. Https://www.khronos.org/registry/OpenGL/index_gl.php. Citado na página 23.
- OREN, M.; NAYAR, S. K. Generalization of lambert's reflectance model. In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. [S.l.: s.n.], 1994. p. 239–246. Citado na página 114.
- 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 14, 18, 19, 20, 21 e 40.
- 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 2 vezes nas páginas 29 e 59.
- PROKHOROV, A. V.; HANSEN, 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 18.
- 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 28.
- SIPSER, M. Introduction to the theory of computation. In: _____. 3rd. ed. [S.l.]: Cengage Learning, 2012. cap. 2. Citado na página 28.
- TAN, P. Phong reflectance model. *Computer Vision: A Reference Guide*, Springer, p. 1–3, 2020. Citado na página 23.
- The Khronos Group. *OpenGL Interpolation*. 2015. Disponível em: <<Https://www.khronos.org/opengl/wiki/Interpolation>>. Citado na página 26.
- THE reciprocity principle in lunar photometry. *Astrophysical Journal*, vol. 93, p. 403-410 (1941)., v. 93, p. 403–410, 1941. Citado na página 143.
- WALTER, B. Notes on the ward brdf. *Proceedings of Technical Report PCG-05–06, Cornell Program of Computer Graphics*, Citeseer, 2005. Citado na página 102.
- WEYRICH, T. et al. [S.l.: s.n.], 2009. Citado na página 19.
- WOLFE, W. L. *Introduction to radiometry*. [S.l.]: Spie press, 1998. v. 29. Citado na página 17.
- 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 21.