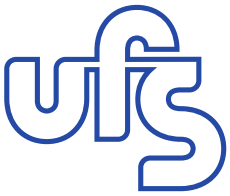


UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

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

Trabalho de Conclusão de Curso

Everton Santos de Andrade Júnior

São Cristóvão – Sergipe

2024

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Everton Santos de Andrade Júnior

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

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

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

São Cristóvão – Sergipe

2024

Resumo

O presente trabalho propõe o desenvolvimento de um compilador de funções de distribuição de reflexão bidirecional (BRDFs) expressas em LaTeX para a linguagem de shading GLSL, utilizando a técnica de parsing de Pratt. O objetivo é automatizar o processo de tradução de funções complexas de materiais, frequentemente descritas em LaTeX, para o código GLSL utilizado em programação de shaders para OpenGL. Ao fornecer essa ferramenta, pretende-se não apenas simplificar o trabalho dos desenvolvedores e pesquisadores na área de computação gráfica, mas também democratizar o acesso e compreensão de modelos de materiais complexos. Além disso, ao permitir que as BRDFs sejam expressas em uma forma mais familiar e acessível, como a notação matemática, o compilador reduz a barreira de entrada para aqueles que não estão familiarizados com linguagens programação. Isso pode facilitar a colaboração interdisciplinar entre profissionais de diferentes áreas, como artistas visuais, designers e cientistas de materiais, que desejam explorar e entender o comportamento visual de materiais em suas aplicações.

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

Lista de ilustrações

Lista de quadros

Lista de tabelas

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
abnTeX	ABsurdas Normas para TeX
DCOMP	Departamento de Computação
UFS	Universidade Federal de Sergipe

Sumário

1 Introdução

1.1 Contexto

Na computação gráfica, a representação realista de cenas tridimensionais depende fortemente da modelagem da luz. A interação da luminosidade incidente no objeto, bem como os materiais que compõem esses objetos, são aspectos críticos a serem considerados na geração dessas cenas [referencia]. Na prática, essa interação é frequentemente modelada por meio de funções de distribuição de refletância bidirecional, conhecidas como BRDFs.

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

1.2 Motivação

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

No meio acadêmico, as BRDFs são, comumente, descritas por uma fórmula escrita em LaTeX, uma abordagem promissora para atender a essa necessidade é o desenvolvimento

de um compilador capaz de traduzir BRDFs em LaTex para shaders, assim democratizando a visualização dessas BRDFs. Dado que as fórmulas são equações matemáticas, precisamos retrigir repretsentação da linguagem de entrada para o compilador afim de garatir um projeto útil em tempo ábil.

1.3 Objetivo

Este trabalho visa projetar e implementar um compilador que, a partir de funções de distribuição de refletância bidirecional escrita como equações em LaTeX, seja capaz de gerar código de shading na linguagem alvo da API OpenGL (referencia). A saída será um shader capaz de reproduzir as características de reflexão da função de refletância original, considerando a precedencia de operadores, em uma superfície tridimensional, ou, ao menos, alcançar uma aproximação satisfatória dessas características, considerando as limitações da linguagem de shading da API principalmente as representações de dados de forma discreta.

1.4 Metodologia

Para alcançar o objetivo, a sequencia das etapas adotadas serão as seguintes.

1. Realizar uma análise abrangente das áreas relacionadas ao desenvolvimento da ferramenta proposta;
2. Investigar o estado da arte no campo da compilação de BRDFs em linguagens de shading;
3. Definir a linguagem de entrada e a linguagem de saída do compilador;
4. Elaborar testes com equações LaTeX de entrada pareado com a saída em shader GLSL esperado;
5. Implementar o compilador utilizando uma linguagem de programação e tecnicas recursivas de parsing
6. Realizar a renderização de cenas utilizando o shader gerado pelo compilador.

2

Revisão Bibliográfica

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

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

2.1 Mapeamento Sistemático

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

2.1.1 Seleção das Bases

As bases escolhidas foram: ACM Digital Library, IEEE Xplorer Digital Library, Biblioteca Digital Brasileira de Teses e Dissertações (BDTD), Portal de Periódicos da CAPES, Google Acadêmico, esse foram escolhidos por serem acessíveis gratuitamente pela afiliação à Universidade Federal de Sergipe, já o google scholar foi escolhido para agregar pesquisas em outras bases que possam ter trabalhos relevantes. <https://bdtd.ibict.br/> <https://ieeexplore.ieee.org/> <https://www-periodicos-capes-gov-br.ezl.periodicos.capes.gov.br/>

2.1.2 Questões de Pequisa

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

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

2.1.3 Termos de Busca

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

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

Tabela 1:

2.1.4 Critérios

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

2.1.4.1 Critérios de Inclusão

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

2.1.4.2 Critérios de Exclusão

1. Foram excluídos artigos dos quais dispunham de links incorretos e ou quebrados;
2. Foram excluídos artigos que dispunham de aplicacões muito similares/repetitivas;
3. Foram excluídos artigos que não respodem as questões de pesquisa 2.1.2;
4. Artigo que não tem como entrada a BRDFs no formato de equação, ou seja, está utilizando a representação diretamente como código, também foi excluído.

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

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

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

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

Tabela 2: Resultados da Base após aplicar os critérios

2.1.5 Descrição dos Trabalhos Relacionados

2.1.5.1 genBRDF: Discovering New Analytic BRDFs with Genetic Programming

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

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

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

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

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

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

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

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

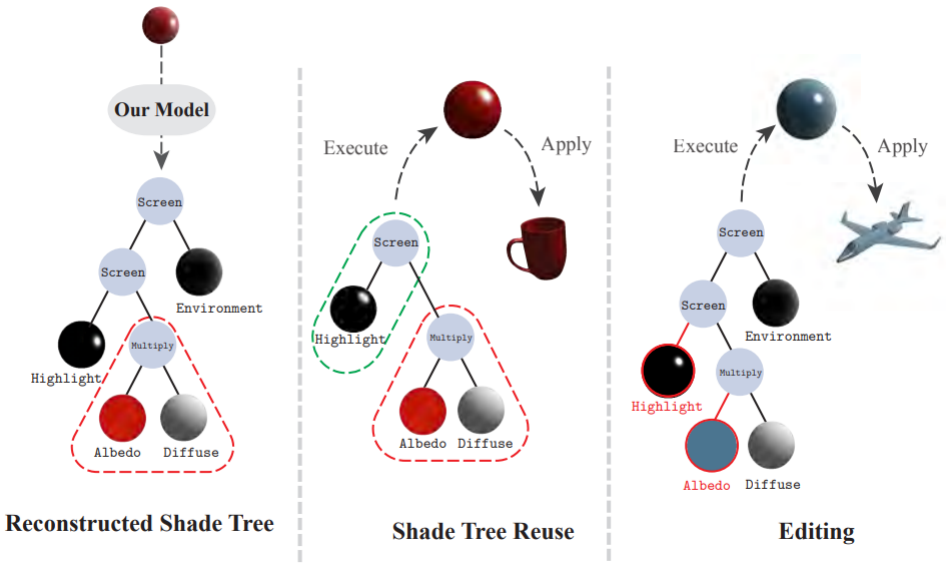
Enquanto o artigo se concentra na extensão de linguagens de shading existentes para melhorar a eficiência e a extensibilidade dos sistemas de shading em tempo real, o nosso trabalho se concentra na compilação de equações BRDF em linguagens de shading para explorar e descobrir novos modelos analíticos,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

2.1.5.3 Tree-Structured Shading Decomposition

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

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

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



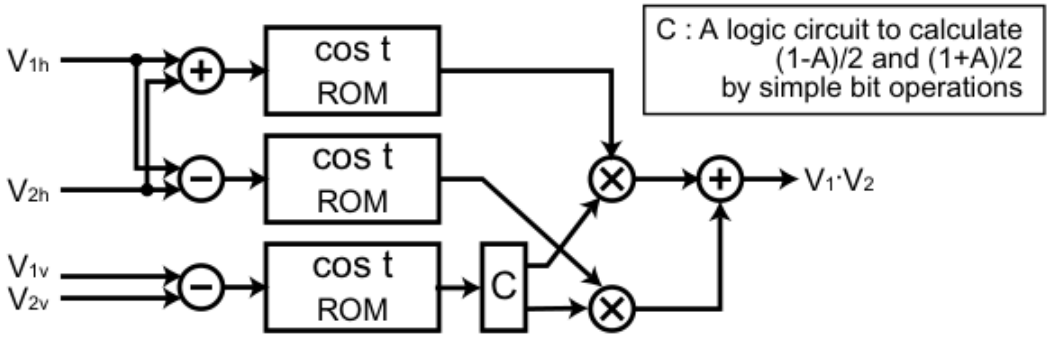
Fonte: **??**, p. 24)

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

2.1.5.4 A Real-Time Configurable Shader Based on Lookup Tables

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

Figura 2 – Exemplo de circuito de produto interno entre vetores



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

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

2.2 Pesquisa por Repositórios online

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

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

Tabela 3:

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

3
Conceitos

3.1 BRDFs

<https://www.youtube.com/watch?v=kPIqO929pIc&list=PL2zRqk16wsdpyQNZ6WFlGQtDICpzzQ9
index=3>

3.2 Radiometria

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

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

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

Renderização Além da Cor

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

focando nos caminhos em linha reta dos fótons.

3.2.1 Energia Radiante e Fluxo

Vários processos físicos convertem energia em fótons, como radiação de corpo negro e fusão nuclear em estrelas. Quantificar a energia radiante total envolve entender a energia dos fótons colidindo com um objeto, equivalente ao brilho da imagem. A energia radiante Q considera a energia total de todos os fótons atingindo a cena durante toda a duração.

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

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

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

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

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

3.2.2 Radiância e BRDF

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

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

E_w é função de a irradiancia numa direção w

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

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

A radiância pode fornecer informação sobre o quanto um ponto específico está iluminado na direção da câmera. Ela depende não apenas da direção do raio que incide câmera, mas também das propriedades de refletância da superfície. E, no contexto de renderização, a radiância de uma superfície na cena se correlaciona com a irradiância de um pixel em uma imagem da sequinte forma:

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

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

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

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

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

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

L_i é radiancia incidente

w_i é a direção incidente

w_o é a direção de saída

H^2 são todas no hemisfério

θ angulo entre direção incidente e a normal da superficie

F função de refletancia

A Função de Distribuição Bidirecional de Reflectância (BRDF) descreve como a luz reflete de uma superficie em diferentes direções afetando a radiância de saída. Reflexão é o processo no qual a luz iterage com a superficie sem alterar a sua frequencia. Assim, BRDFs encapsulam as propriedades de reflexão de um material levando em conta vários fatores, como

rugosidade da superfície, ângulo de incidência, angulo de reflexão Formalmente uma BRDF pode ser definida por $F(w_i, w_o)$, onde w_i é a direção indicente de luz e w_o é a direção de saída.

Para BRDFs fisicamente realistas algumas propriedades devem ser respeitadas.

- A propriedade de positividade, $F(\omega_i, \omega_o) \geq 0$, que garante não existência de energia negativa.
- Também, deve-se obedecer a reciprocidade de Helmhotz, $F(\omega_i, \omega_o) = F(\omega_o, \omega_i)$. Essa reciprocidade é usado na renderização, pois no lugar de traçar os raios da fonte de luz até a camera, podemos traçar os raios da camera até a fonte de luz otimizando a maior parte dos raios traçados diretamente da fonte de luz que não iriam antigir a lente da camera, exitando desperdicio de poder computacional em raios que não contribuem para intensidade de um dado pixel.
- A BRDF deve, também, respeitar a conversação de energia, $\forall\omega_i, \int_{H^2} F(\omega_i, \omega_o)cos(\theta_o)d\omega_o \leq 1$. Nesse caso parte da energia pode ser absorvida, ou seja, transformado em outra forma de energia como calor, nesse caso esse somatório infintesimal pode no máximo chegar a 1, mas nunca ultrapassar.

3.3 BRDF Models

3.3.1 Superfície Pura Especular

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

3.3.2 BRDF Difusa Ideal

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

3.3.2.1 BRDF Brilhante (*Glossy*)

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

3.3.2.2 BRDF Retro-Refletora

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

3.4 Compiladores

3.4.1 Cadeia de Símbolos e Alfabeto

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

3.4.2 Definições de Linguagens

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

3.4.3 Compilador como um Transformação

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

3.4.4 Gramática

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

- Terminais: Terminais são os símbolos básicos a partir dos quais as cadeias são formadas. Eles representam as unidades elementares da sintaxe da linguagem.
- Não-terminais: Não-terminais são espaços reservados que podem ser substituídos por terminais ou outros não-terminais de acordo com as regras de produção.
- Regras de Produção: As regras de produção definem a transformação ou substituição de não-terminais em sequências de terminais e/ou não-terminais.
- Símbolo Inicial: O símbolo inicial é um não-terminal especial a partir do qual a derivação de cadeias válidas na linguagem começa.

3.4.4.1 Gramáticas Livres de Contexto (GLCs)

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

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

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

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

3.4.5 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. No caso a linguagem do entrada analisador léxico são os formados por caracteres e, geralmente, são reconhecivel por maquinas de estado @ref, já a linguagem de saída é composta por tokens.

3.4.6 Análise Sintática ou *Parsing*

A análise sintática é a segunda fase do processo de compilação, na qual os tokens gerados pela análise léxica são organizados e verificados quanto à conformidade com a gramática da linguagem. Essa etapa envolve a construção de uma árvore sintática ou estrutura de dados equivalente que representa a estrutura hierárquica das expressões e instruções do programa. O analisador sintático utiliza regras de produção gramatical para validar a sintaxe do código fonte e identificar possíveis erros.

3.4.7 Pratt Parsing

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

3.4.7.1 Precedência de Expressões

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

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

3.4.7.2 Árvores Inclinadas

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

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

```
  +
 / \
1   *
   / \
  2   3
```

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

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

3.4.7.3 Pseudo-código para Análise de Expressões

O pseudo-código 1, demonstra o Pratt *parsing* para a construção de árvores de expressão, considerando tanto estruturas inclinadas à direita quanto à esquerda. Esse algoritmo também é robusto mesmo quando um operador é tanto infixo quanto prefixo, por exemplo "−"pode ser um *token* de subtração ou de negação. Assim cada token tem uma função de prefixo e infixo associada.

Nesse algoritimo, **proximo_token()** recupera o próximo elemento da lista de tokens, **token.precedencia**() retorna a precedencia do token atual, **token.prefixo()** é a função associada ao token que realisa o parsing de uma expressão quando o token é o primeiro em uma subexpressão (e.g. o token "−"é o primeiro na expressão "−3"). Já o **token.infixo**(esquerda) é a função associada

ao token uma função que cria um nó subárvore utilizando outra subarvore já criada como entrada para gerar expressão com operadores infixos, por exemplo a subarvore esquerda pode ser a expressão "-3", o token atual ser "$*$"e o o retorno gera a expressão completa "$-3 * 1$"

Tanto **token.infixo** quanto **token.prefixo** podem ser indiretamente recursivas, isto é, ambas podem chamar a função **expressao** no algortimo 1. Por fim, **precedencia_anterior** representa a precedência do token anterior, garantindo que os operadores sejam resolvidos na ordem correta.

Algoritmo 1 – Função Pratt Parsing de Expressão

```
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
```

3.4.8 Geração da Linguagem Alvo

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

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

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

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

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

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

3.5 Introdução ao Shading e ao Pipeline de GPU

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

A interação com as GPUs é facilitada por meio de uma *Application Programming Interface*(API), em portugues, Interface de Programação de Aplicação, sendo o OpenGL uma API padrão para o uso de funções na GPU. O OpenGL foi criado como um padrão de programação aberto e multiplataforma para GPUs. O pipeline de renderização do OpenGL é composto por várias etapas, incluindo definição de dados de vértices, shaders de vértice e fragmento, shaders de tesselação e geometria opcionais, configuração de primitivas, recorte, rasterização e shader de fragmento.

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

@image put in the overview image

3.6 Vertex Shader

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

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

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

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

```
uniform mat4 modelViewProjection;

out vec3 fragNormal;

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

3.7 Fragment Shader

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

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

@image put an image about verter shading and fragment shading

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

in vec3 fragNormal;
out vec4 fragColor;

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

4

Methodology

The methodology for developing the proposed compiler involves a systematic approach comprising steps tailored to accomplish the specified research objectives. These steps entail in-depth analysis of related information pertinent to BRDFs and shader compilation, exploration of existing techniques within the domain, formulation of precise language specifications, meticulous design and execution of comprehensive test cases, the actual implementation of the compiler, and evaluation of its performance via rendering experiments.

4.1 Comprehensive Analysis

The first step involves conducting a thorough analysis of areas related to the development of the proposed tool. This includes reviewing literature on BRDFs, shader languages, compiler design, and graphics rendering techniques. This involves studying research papers, existing tools, and relevant software libraries to identify existing techniques and potential areas for improvement.

4.2 Language Specification

The input and output language specifications for the compiler are defined. The input language is a simplified version of LaTeX, where mathematical expressions in the `equation` or `align` environments suffice for the purposes of this work. LaTeX is a typesetting system commonly used for mathematical and scientific documents. For describing BRDFs using LaTeX equations, while the output language specifies the GLSL shader language for rendering.

For the input language, we opted to use the `equation` environment due to its simplicity and widespread use in representing mathematical expressions. In LaTeX, environments provide a way to group and format text or equations. The `equation` environment is specifically designed for displaying single equations. Below is an example of LaTeX code using the `equation`

environment:

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

This LaTeX code represents a quadratic equation $f(x) = ax^2 + bx + c$, where a, b, and c are coefficients.

As for the output language, it specifies the GLSL shader language for rendering. An example of a corresponding GLSL shader code generated from the LaTeX equation above might be:

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

4.3 Test Case Design

Test cases must be designed to validate the correctness and accuracy of the compiler's translation process. These test cases pair input LaTeX equations describing BRDFs with the expected output GLSL shader code. A specific case demonstrating the compiler's capability can be created with the Cook-Torrance BRDF. The LaTeX definition and expected output can be provided as an example:

LaTeX Input (Cook-Torrance BRDF):

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

Expected GLSL Output:

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

    // microfacet distribution function
    float D = ...; // D(h)

    // Calculate Fresnel term
```

```
    float F = ...; //F(wi, h)

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

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

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

This example showcases how the input LaTeX equation representing the Cook-Torrance BRDF is translated into the corresponding GLSL shader function.

4.4 Compiler Implementation

The compiler is implemented using the Odin programming language, chosen for its low-level language capabilities with composition inheritance and a satisfactory standard library. Recursive parsing techniques are utilized, specifically Pratt Parsing, to process the input LaTeX equations and generate the corresponding GLSL shader code. Initially, a lexer and parser were implemented for a simpler language than LaTeX to ensure the compiler's foundations is functional, with fully tested precedence for the synax tree, and also the execution of this.

4.5 Rendering Experiments

Finally, rendering experiments are performed using the shaders generated by the compiler. This allows for the evaluation of the performance and visual quality of the rendered images produced by the compiled shaders. The chosen platform for testing is the Disney BRDF tool, compiled locally to modify and add additional shaders. It provides an interface for defining parameters that serve as sliders for the BRDF's hyperparameters.

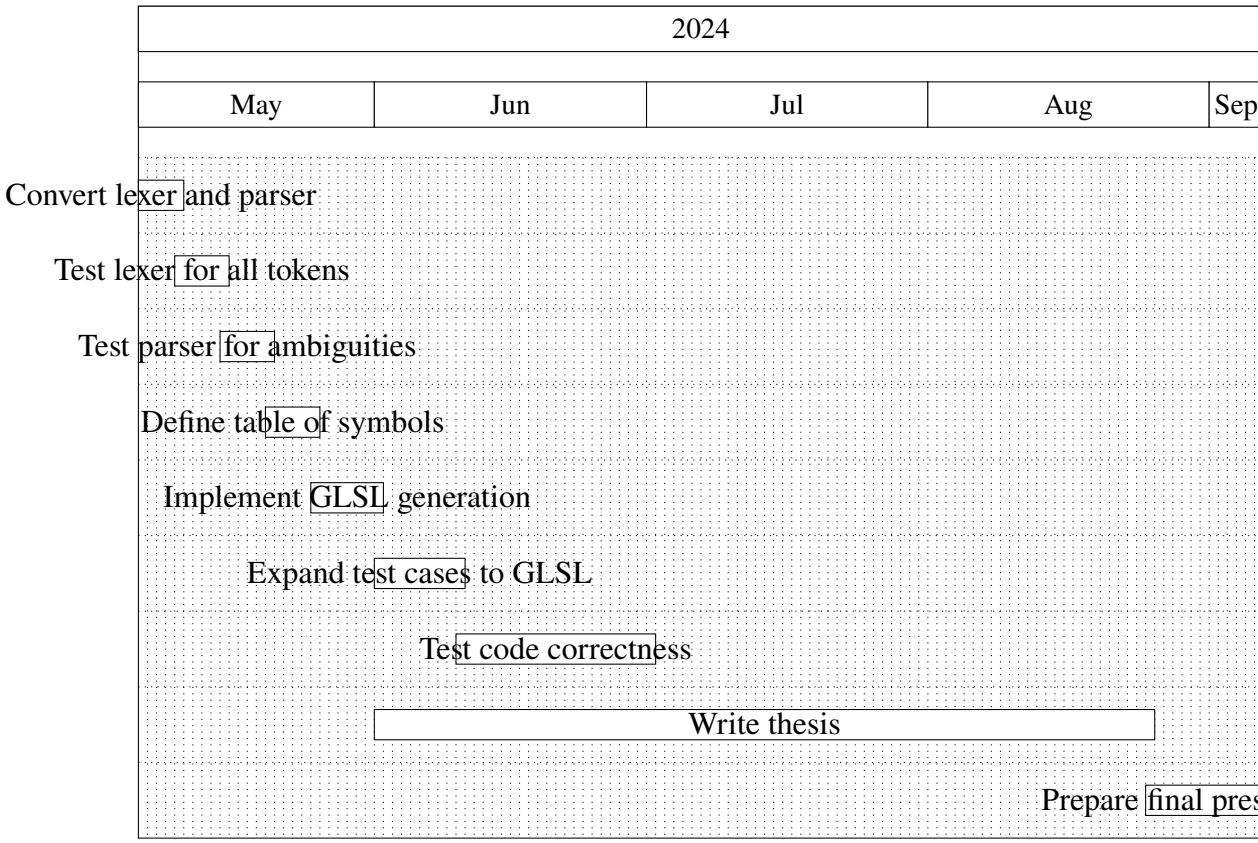
@Image for the tool @code for glsl super set

By following this methodology, the proposed tool aims to effectively compile BRDF descriptions into GLSL shaders, facilitating the generation of realistic graphics in computer graphics applications.

4.6 Plan of Continuation

The continuation of this thesis involves several key tasks aimed at completing the development of the proposed compiler for converting LaTeX equations describing Bidirectional Reflectance Distribution Functions (BRDFs) into GLSL shader code. The tasks include updating the lexer and parser written in Odin to accept LaTeX equations, testing the lexer and parser to ensure correct token recognition and parsing of LaTeX syntax, defining predefined symbols such as mathematical constants and other quantities, implementing the GLSL code generation process, expanding test cases to cover a wide range of BRDFs, thoroughly testing the generated code for correctness and effectiveness, preparing the final presentation, and finally, writing the thesis document.

1. 06/05/2024 - 20/05/2024: Update the lexer and parser to accept LaTeX equations.
2. 06/05/2024 - 27/05/2024: Test the lexer and parser to ensure correct token recognition and parsing of LaTeX syntax.
3. 13/05/2024 - 03/06/2024: Define predefined symbols such as mathematical constants and other quantities.
4. 20/05/2024 - 10/06/2024: Implement the GLSL code generation process.
5. 27/05/2024 - 17/06/2024: Expand test cases to cover a wide range of BRDFs.
6. 03/06/2024 - 24/06/2024: Thoroughly test the generated code for correctness and effectiveness.
7. 10/06/2024 - 01/07/2024: Prepare the final presentation.
8. 17/06/2024 - 06/09/2024: Write the thesis document.


2024
May
Jun
Jul
Aug
Sep
Convert lexer and parser
Test lexer for all tokens
Test parser for ambiguities
Define table of symbols
Implement GLSL generation
Expand test cases to GLSL
Test code correctness
Write thesis
Prepare final pres

Referências