FUNDAÇÃO GETULIO VARGAS
ESCOLA DE MATEMÁTICA APLICADA

CRISTHIAN GRUNDMANN

GEODESIC TRACING: VISUALIZAÇÃO DE CURVAS E SUPERFÍCIES ATRAVÉS DE GEODÉSICAS

Rio de Janeiro
2022

CRISTHIAN GRUNDMANN

GEODESIC TRACING: VISUALIZAÇÃO DE CURVAS E SUPERFÍCIES ATRAVÉS DE GEODÉSICAS

Trabalho de conclusão de curso apresentada para a Escola de Matemática Aplicada (FGV/EMAp) como requisito para o grau de bacharel em Matemática Aplicada.

Área de estudo: curvas e superfícies.

Orientador: Asla Medeiros e Sá

Rio de Janeiro

2022

Lista de códigos

1 Introdução

Desenhos de superfícies costumam ser feitos a partir de um ponto de vista do espaço ambiente 3D. Esse projeto implementa uma visualização de superfícies que não depende de um espaço ambiente.

A visualização pode ser comparada ao que um ser bidimensional interno à superfície observaria: simula-se raios de luz partindo da posição do ser, e os pontos iluminados são observados. Os raios de luz devem seguir caminhos em 'linha reta', que minimizam distância. Para uma superfície qualquer, esses caminhos são chamados de geodésicos, e são estudados na geometria diferencial. A visualização, chamada de *geodesic tracing*, obtém uma transformação de uma imagem original sobre a superfície.

A implementação feita nesse projeto é feita em três partes: compilador, método numérico e interface gráfica.

O compilador fornece uma maneira do usuário definir as superfícies e outros objetos. O usuário escreve um texto, seguindo algumas regras gramaticais, que então é processado. A teoria de compiladores é essencial para essa etapa, principalmente a análise léxica e a análise sintática (AHO, 1986). O compilador está descrito no capítulo 3. A linguagem, com exemplos de programas, está descrita no capítulo 2.

O método numérico se refere à simulação dos raios de luz na superfície. Um raio de luz é determinado pela posição e direção inicial, que são as condições iniciais. Um sistema de equações diferenciais ordinárias(equação geodésica (PRESSLEY, 2012)) determina a curva que a luz traça. Uma solução aproximada da equação é calculada pelo método de Runge-Kutta de ordem 4 (BURDEN, 2001). O método está descrito no capítulo 4.

A interface gráfica é simples e é construída usando *ImGui* (CORNUT, s.d.), uma ferramenta de interface gráfica fácil de usar. A linguagem de programação escolhida para a implementação desse projeto é *C++*, e para desenhar a interface e os objetos, *OpenGL* é usado.

2 Linguagem

O usuário se comunica com a interface através de um texto, chamado de programa, que contém os objetos de interesse. O código 2.1 é um exemplo.

Código 2.1 – Exemplo de objetos

```
1  #circle and tangents
2  param r : [/2, 1];
3  param o : [0, 2pi];
4  curve c(t) = r(cost, sint, 0), t : [0, 2pi];
5  grid k : [0, 2pi, 8];
6  define k2 = k + o;
7  point p = ck2;
8  vector v = c'k2 @ p;
9
10 #function and surface
11 #function f(x, y) = x^2+y^2;
12 #surface s(u,v) = (u,v,f(u,v)), u : [-1, 1], v : [-1, 1];
```

A linguagem permite comentários no estilo da linguagem Python, usando `#`.

O programa declara os seguintes objetos:

`r` e `o`	parâmetros que podem ser alterados na interface. Seus valores devem estar nos intervalos indicados.
`c`	uma curva parametrizada por `t`. O domínio da parametrização é o intervalo indicado. A curva depende do parâmetro `r`, que foi definido anteriormente.
`k`	uma grade de 8 pontos igualmente espaçados no intervalo indicado. Uma grade é tratada como uma constante, assim como um parâmetro. Se um objeto desenhável depende de uma grade, uma instância é desenhada para cada valor da grade. Um objeto pode depender de mais de uma grade.
`k2`	uma constante, e não pode ser alterada na interface como os parâmetros. Esse tipo de objeto pode ser usado para deixar o programa mais legível.
`p`	o ponto da curva `c` de parâmetro `t = k2`. Esse objeto depende indiretamente de `k`, então é instanciado 8 vezes.
`v`	o vetor tangente da curva `c` no ponto `p` e desenhado a partir do mesmo ponto. O vetor também depende indiretamente de `k`, então é desenhado 8 vezes.

A figura 1 demonstra os objetos declarados em perspectiva 3D.

Os objetos `f` e `s` estão comentados, então não são considerados. Estão presentes apenas para o exemplo ter todos os tipos de objeto.

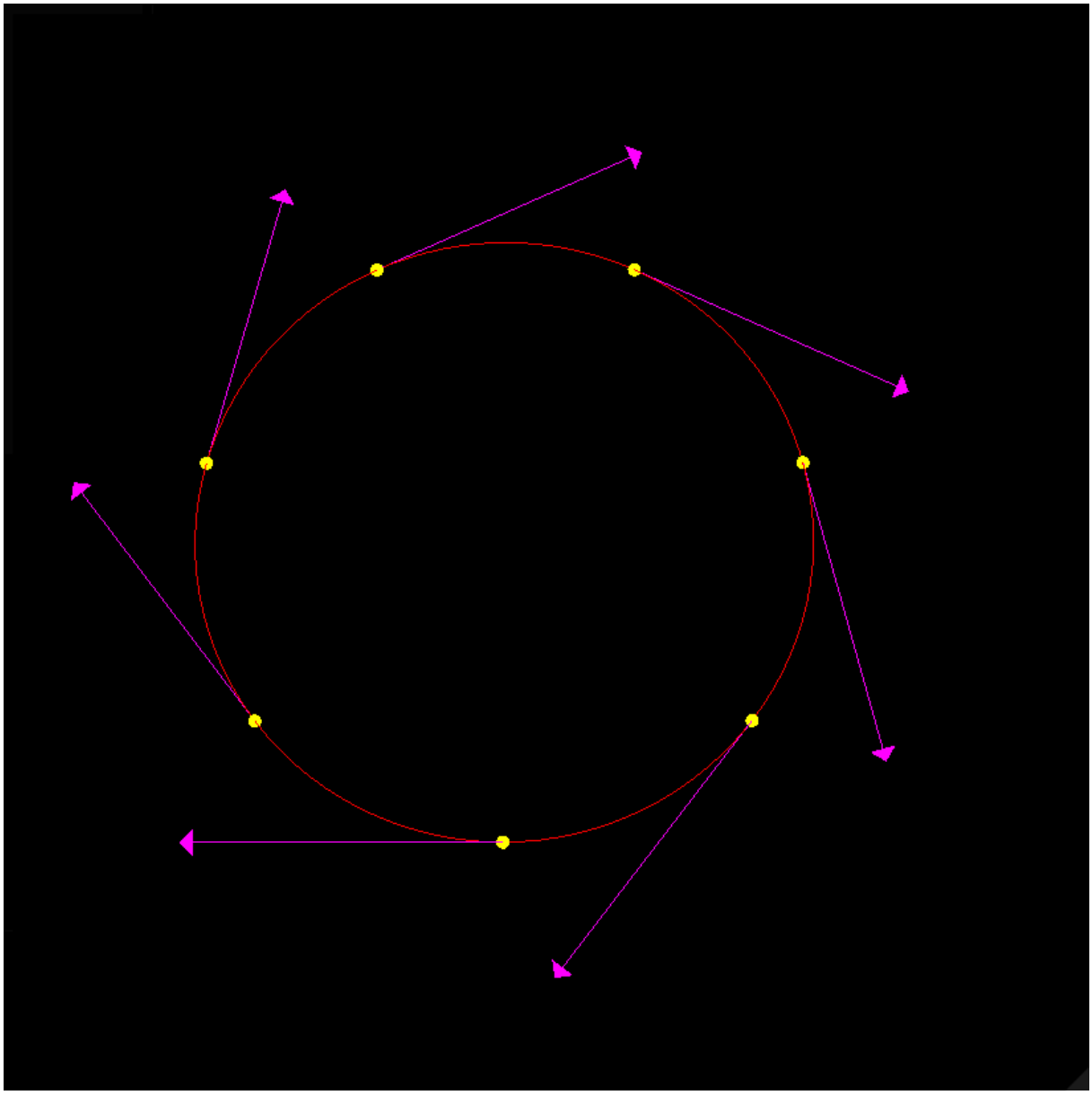

Figura 1 – Exemplo 1

A linguagem de descrição de objetos não é trivial, nem sua sintaxe matemática, que possui elementos inventados para esse projeto. A seguir, uma breve lista de observações:

- Os objetos desenháveis são pontos, vetores, curvas e superfícies. Pontos e vetores podem ser usados em outros objetos, sendo tratados como tuplas. Por exemplo, `v` usa o ponto `p`. Curvas e superfícies podem ser usadas como funções, mas sem a restrição no domínio. Por exemplo, `p` usa `c` como função. Um objeto só pode se referir aos objetos definidos anteriormente.
- Há duas constantes pré-definidas: `pi` e `e`; e diversas funções pré-definidas: `sin`, `cos`, `tan`, `exp`, `log`, `sqrt` e `id`. A função `id` é a identidade é útil apenas no funcionamento interno do sistema.

- Parâmetros e grades podem ser multidimensionais: `param T : [0, 1], [0, 1];`. Assim, o objeto `T` é uma tupla, e seus elementos podem ser obtidos com `T_1` e `T_2`.
- As grades das curvas e superfícies são por padrão 100 e 100x100, respectivamente. É possível alterar esse valor informando um intervalo do tipo grade: `[0, 2pi, 250]`.
- Há 4 operadores unários. Os operadores `+` e `-` são os usuais. A operação `*x` representa `xx`, e `/x` é igual a `1/x`. Para números reais, multiplicação com `*` e por justaposição são equivalentes. Porém, para tuplas, `a*b` representa o produto vetorial e `ab` representa o produto escalar. Assim, `*x` calcula o quadrado do módulo do vetor `x`. Uma função que normaliza vetores pode ser definida assim: `function N(x) = x/sqrt*x;`.
- Numa aplicação de função de uma variável, o argumento não precisa de parênteses: `sin x`. O argumento pode ter operadores unários e até expoentes: `sin -x^2 = sin(-x^2)`. Deve-se tomar cuidado com expoentes:
  textttsin(x)^y = sin(x^y). Para a exponenciação de uma aplicação, deve-se usar a sintaxe: `sin^2 x`.
- Não é sempre necessário uma separação entre identificadores. Por exemplo, considere `sinx`. Caso haja um termo chamado `sinx` definido, esse seria o identificador reconhecido. Caso contrário, `sin x` será reconhecido, mesmo que `sinx` seja definido posteriormente (`sinx` seria reconhecido apenas depois de sua definição). Em geral, o maior identificador definido será reconhecido.

O código 2.2 é outro exemplo.

Código 2.2 – Exemplo de objetos

```
1  #sphere and coordinates
2  function f(u,v) =
3      (sin(piv)cos(2piu), sin(piv)sin(2piu), cos(piv));
4
5  param U : [0, 1];
6  param V : [0, 1];
7
8  point p = f(U,V);
9
10 curve cu(t) = f(t, V), t : [0, 1];
11 curve cv(t) = f(U, t), t : [0, 1];
12
13 function N(x) = x/sqrt*x;
14
15 vector vu = Nf_u(U,V) @ p;
16 vector vv = Nf_v(U,V) @ p;
17
18 surface s(u,v) = f(u,v)*0.99,
```

```
19 u : [0, 1], v : [0, 1];
```

A figura 2 demonstra o programa em perspectiva 3D.

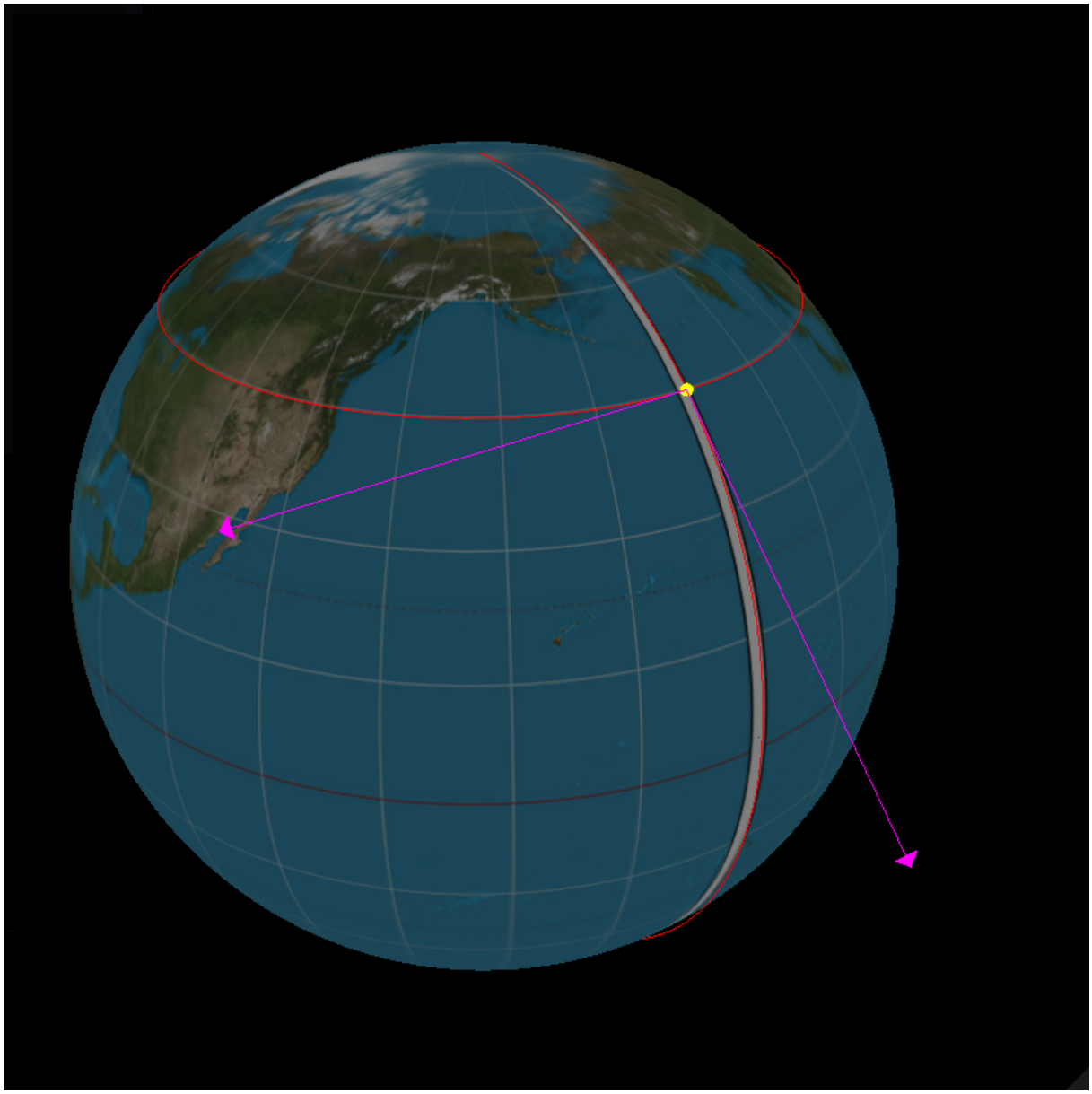

Figura 2 – Exemplo 2 - Perspectiva em 3D

A figura 3 demonstra o geodesic tracing.

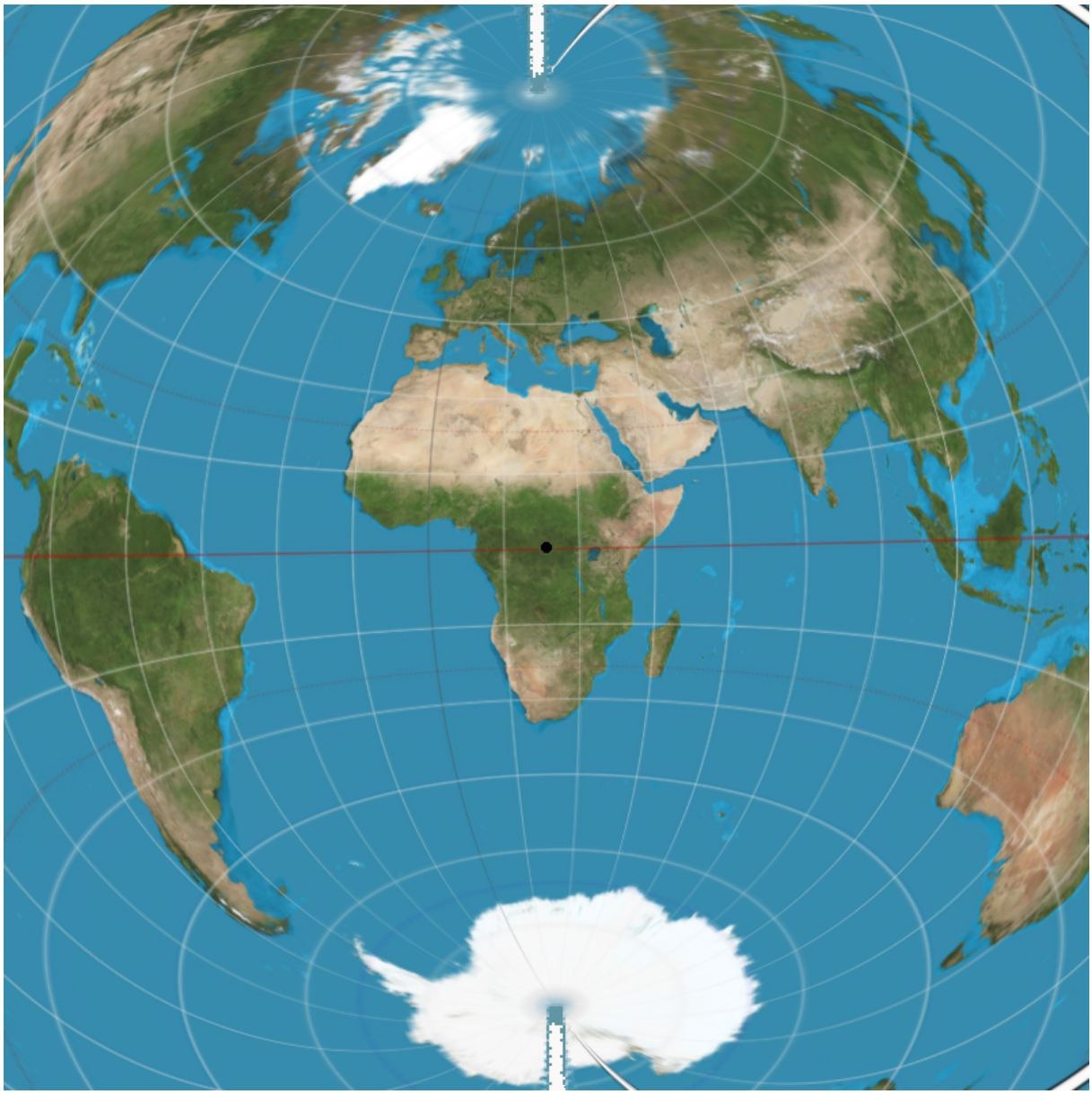

Figura 3 – Exemplo 2 - geodesic tracing

3 Compilador

O processo de compilação não é trivial, e é dividido em 3 estágios:

- análise léxica: reconheçe as “palavras” que compõe um programa, ignorando espaços em branco. É capaz de identificar números, constantes, nomes de objetos e pontuação. Os termos são usados no estágio seguinte.
- análise sintática: reconheçe a estrutura do programa, como as declarações dos objetos e as expressões matemáticas. A gramática 3.3 é usada como base.
- análise semântica e síntese: gera todas as estruturas de dados necessárias para a visualização dos objetos. Verifica também a semântica do programa, detectando erros que não podem ser verificados com noções gramaticais.

3.1 Análise léxica

A análise léxica tem a função de ler o código-fonte que descreve um programa e abstrair as palavras e símbolos presentes. Dessa forma, os estágios seguintes se beneficiam dessa abstração. O analisador léxico é chamado de lexer.

As palavras-chave, números, constantes, símbolos, etc. são chamados de lexemas. Todo lexema deve pertencer a uma classe gramatical. Por exemplo, o texto `"1024"` forma um lexema de 4 caracteres e sua classe gramatical é `NUMBER`. Conforme a classe, um lexema pode ter atributos. No caso de `NUMBER`, o próprio número em forma de ponto flutuante é um atributo. No caso de um símbolo como `";"`, não há atributos.

Um lexema, sua classe gramatical e seus atributos juntos formam um **token**.

Os tipos de token são:

`COMMENT`	texto livre, começando com `"#"` e terminando com uma quebra de linha ou o fim do código-fonte.
`FUNCTION`	identificador de função definida ou pré-definida.
`NUMBER`	número de ponto flutuante.
`VARIABLE`	identificador de variável de função.
`CONSTANT`	identificador de constante definida ou pré-definida.
`DECLARE`	identificador de tipo de objeto.
`UNDEFINED`	identificador não definido ou a ser definido.
`EOI`	fim do código-fonte, caractere nulo(0).
Caso final	o lexema é um símbolo e seu tipo é o próprio símbolo.

A estrutura `Lexer`(3.1) define o analisador léxico.

Código 3.1 – Extrutura do Lexer

```
1  struct Lexer
2  {
3      const char *source{};
4      const char *lexeme{};
5      int length = 0;
6      int lineno = 0;
7      int column = 0;
8      TokenType type = TokenType::UNDEFINED;
9      float number{};
10     Table *node{};
11     Table *table{};
12
13     void advance(bool match = true);
14 };
```

O lexer lê os tokens um de cada vez, da esquerda para a direita. Essa estrutura guarda as seguintes informações sobre o token atual:

`source`	string do código-fonte inteiro.
`lexeme`	aponta para o primeiro caractere do lexema atual(dentro da string `source`)
`length`	comprimento do token.
`lineno` e `column`	o número da linha e coluna do lexema.
`type`	tipo do token.
`number` e `node`	atributos do token. No caso de um número, `number` é o atributo. No caso de um identificador, `node` é sua posição na tabela de símbolos(3.2), contendo mais informações sobre o token.
`table`	tabela de símbolos compartilhada pelos estágios da compilação.

O lexer tem apenas o método `advance`, que serve para avançar para o próximo token. O método também é usado para inicializar o lexer e obter o primeiro token do código-fonte, invocando-o com `lexeme=source` e `length=0`.

O método começa avançando a posição do `lexeme` a quantidade de `length` caracteres à direita. Em seguida, espaços em branco são ignorados: espaços, tabulações e quebras de linha.

Se o caractere em `lexeme` for `"#"`, então o token é um comentário(tipo `COMMENT`), que se extende até uma quebra de linha ou até o `EOI`, sem incluí-los.

Se o caractere for nulo(0), então o tipo do token é `EOI` e `length=0`. Isso faz com que o lexer trave nesse token e nunca mais avance.

Se o caractere for um dígito ou `"."`, então o token é um número(tipo `NUMBER`), e é lido pela função `sscanf` da linguagem C.

Se o caractere pertencer ao alfabeto dos identificadores, então o lexer o procura na tabela de símbolos obtendo o `node`. Com esse nó, o tipo de token e comprimento são obtidos.

Caso contrário, o token é um símbolo, seu tipo é o próprio símbolo e seu comprimento é 1.

3.2 Tabela de símbolos

Os estágios da compilação compartilham uma tabela de símbolos, que é inicializada com palavras-chave, funções e constantes pré-definidas. A tabela de símbolos define os atributos dos identificadores, que são o tipo do token, os argumentos da função e o índice do objeto.

A estrutura `Table`(3.2) define a tabela de símbolos.

Código 3.2 – Tabela de símbolos

```
1  struct Table
2  {
3      Table *parent{};
4      std::unique_ptr<Table> children[62];
5      int argIndex = -1;
6      int objIndex = -1;
7      int length = 0;
8      TokenType type = TokenType::UNDEFINED;
9      char character{};
10     std::string str{};
11
12     Table *next(char c);
13     Table *procString(const char *str, bool match);
14     Table *initString(const char *str, TokenType type);
15 };
```

Essa estrutura é uma árvore de prefixos(trie): cada nó representa um identificador. Para encontrar um nó a partir de um identificador, basta traçar um caminho a partir da raíz. Os filhos de um nó correspondem a um caractere do alfabeto `a-zA-Z0-9`. Os 26 primeiros filhos são `a-z`, os próximos 26 são `A-Z`, e os 10 últimos são `0-9`. Assim, cada letra do alfabeto indica qual filho seguir.

O membro `character` representa a letra conforme o pai do nó. Por exemplo, se o nó é o primeiro filho, então a letra é `a`. Para a raíz, o caractere é nulo(0). O membro `parent` é o pai do nó, ou nulo para a raíz. Os nós `children[62]` são os $26 + 26 + 10$ filhos.

O tamanho do identificador é `length`, e seus atributos são `type`, `argIndex` e `objIndex`. O atributo `argIndex` representa os argumentos de uma função definida. `type` sempre indica o tipo do token. `objType` representa o índice de um objeto.

O método `next` encontra o filho correspondente ao caractere `c`, e caso seja nulo, um novo filho é criado. O método `procString` busca o identificador em `str` na árvore, usando `next` para traçar o caminho correto. O ponteiro `str` indica o início do identificador(dentro do código-fonte). O método avança no máximo até o primeiro caractere fora do alfabeto dos identificadores. Se o indicador `match` estiver ativo, o método buscará o maior identificador definido, se existir, ou o identificador todo, caso contrário. Por exemplo(`match=true`) para `"sinx"`, o método encontra o identificador `"sin"`, que é uma função. Ou seja, os identificadores não precisam estar separados por um espaço em branco, caso não haja ambiguidade. Se um objeto de nome `"sinx"` estivesse definido, o método encontraria o identificador `"sinx"`, pois é maior. Nesse caso, um espaço em branco faz diferença.

O método `initString` usa `procString` para criar o identificador em `str`, inicializando seu tipo de token com `type`.

3.3 Análise sintática

A análise sintática tem a função de identificar as estruturas sintáticas presentes nos tokens gerados pelo lexer. O gerador, no estágio seguinte, atribui um significado para as estruturas sintáticas reconhecidas, gerando as estruturas de dados desejadas. O analisador sintático é chamado de parser.

A gramática livre de contexto (3.3) define as regras gramaticais da linguagem.

Código 3.3 – Gramática livre de contexto

```
1  PROG    = DECL PROG | ;
2
3  DECL    = "param"     id ":" INTS ";" ;
4  DECL    = "grid"      id ":" GRIDS ";" ;
5  DECL    = "define"    id "=" EXPR ";" ;
6  DECL    = "curve"     FDECL "," TINTS ";" ;
7  DECL    = "surface"   FDECL "," TINTS ";" ;
8  DECL    = "function"  FDECL ";" ;
9  DECL    = "point"     id "=" EXPR ";" ;
10 DECL    = "vector"    id "=" EXPR "@" EXPR ";" ;
11
12 FDECL   = id "(" IDS ")" "=" EXPR ;
13 IDS     = IDS "," id | id ;
14 INT     = "[" EXPR "," EXPR "]" ;
15 GRID    = "[" EXPR "," EXPR "," EXPR "]" ;
16 TINT    = id ":" INT | id ":" GRID ;
17 INTS    = INTS "," INT | INT ;
18 TINTS   = TINTS "," TINT | TINT ;
19 GRIDS   = GRIDS "," GRID | GRID ;
20
21 EXPR    = ADD ;
22 ADD     = ADD "+" JUX | ADD "-" JUX | JUX ;
23 JUX     = JUX MULT2 | MULT ;
24 MULT    = MULT "*" UNARY | MULT "/" UNARY | UNARY ;
25 MULT2   = MULT2 "*" UNARY | MULT2 "/" UNARY | APP ;
26 UNARY   = "+" UNARY | "-" UNARY | "*" UNARY | "/" UNARY | APP;
27 APP     = FUNC UNARY | POW ;
28 FUNC    = FUNC2 "^" UNARY | FUNC2 ;
29 FUNC2   = FUNC2 "_" var | FUNC2 "'" | func ;
30
31 POW     = COMP "^" UNARY | COMP ;
32 COMP    = COMP "_" num | FACT ;
33 FACT    = const | num | var
34         | "(" TUPLE ")" | "[" TUPLE "]" | "{" TUPLE "}" ;
35 TUPLE   = ADD "," TUPLE | ADD ;
```

A gramática consiste em diversas igualdades. Os termos que aparecem no lado esquerdo de alguma igualdade são chamados de não-terminais, e representam um conjunto de sentenças(uma sentença é uma sequência de terminais). Os outros termos, como `";"` e `id`, são terminais, e correspondem a tokens. Os termos `id`, `var`, `const`, `num` e `func` representam qualquer token do tipo indicado: `UNDEFINED`, `VARIABLE`, `CONSTANT`, `NUMBER` e `FUNCTION` respectivamente. Os termos `"param"`, `"grid"`, `"define"`, etc. representam os tokens do tipo `DECLARE`, que são os tipos de objeto.

Uma igualdade na gramática é dita uma produção para o não-terminal à esquerda. O símbolo `"|"` abrevia uma produção alternativa. Por exemplo: `ADD = ADD + JUX | ADD - JUX | JUX` é uma abreviação de `ADD = ADD + JUX`, `ADD = ADD - JUX` e `ADD = JUX`. Uma produção pode ser a string vazia, por exemplo: `PROG = DECL PROG | ;` (o ponto e vírgula no final das igualdades pertence à meta-linguagem).

Uma produção significa que o não-terminal à esquerda pode ser substituído pela forma sentencial à direita. Uma forma sentencial é uma sequência de terminais e não-terminais. Por exemplo, `ADD` pode ser substituído por `ADD + JUX`. Nesse caso, `ADD` deriva `ADD + JUX`. Para se obter um programa gramaticalmente válido, o não-terminal inicial `PROG` deve ser derivado até se obter somente terminais. Uma gramática é dita ambígua quando existe uma sentença com mais de uma forma de obtê-la a partir do não-terminal inicial.

A gramática (3.3) não é ambígua. A verificação foi feita em (CALGARY, s.d.). Algumas transformações na gramática a fizeram ser uma gramática LL(1). Uma consequência disso é a não ambiguidade. Em uma iteração anterior da gramática, a potenciação de funções era associativa à esquerda, enquanto a potenciação de números era à direita. Isso causou uma ambiguidade que não foi detectada no momento. Ela só foi descoberta ao tentar verificar a propriedade LL(1), que falhou.

O trabalho do parser, então, é achar uma forma de derivar uma sentença a partir de `PROG`. O método mais simples de parsing se aplica a gramáticas LL(1).

Num parser LL(1), cada não-terminal possui sua própria subrotina. As subrotinas simulam a substituição de seu não-terminal por uma de suas formas sentenciais possíveis. Ou seja, uma subrotina simula uma produção de seu não-terminal. Para decidir qual produção aplicar, as subrotinas devem consultar o token atual. O fato da gramática ser LL(1) garante que o token atual fornece informação suficiente para determinar qual é a produção correta e, na falta de produção adequada, detectar um erro gramatical. Após decidir a produção, a subrotina começa sua simulação. Os termos da forma sentencial da produção são tratados da esquerda para a direita. Terminais são comparados com o token atual e um erro é detectado quando diferem. Quando são iguais, o lexer avança para o próximo token. Os não-terminais são substituídos imediatamente, através de suas subrotinas.

Por exemplo, considere a produção `FUNC = FUNC2 ^UNARY`. Para simulá-la, deve-se derivar `FUNC2`, chamando sua subrotina. Após a subrotina terminar, o token atual é comparado com `^`, e caso seja igual, o lexer avança para o próximo token. Em seguida, a subrotina `UNARY` é chamada. No final de uma subrotina, seu não-terminal derivou uma sub-sentença, e o token atual ficou imediatamente à direita dessa sub-sentença. Assim, indutivamente, a rotina para `FUNC2` avançou o token para `^` na produção examinada.

O termo `EXPR` define como funcionam as expressões matemáticas, definindo opera-

ções, ordens de precedência e associatividades. A gramática para as expressões matemáticas foi baseada na linguagem C (UNIVERSITY, s.d.). Para a estética das expressões ser mais agradável, a multiplicação pode ser por justaposição, por exemplo: `3*x = 3x`. Em notação matemática comum, isso deixa as fórmulas muito mais simples de ler. Vários ambientes computacionais não possuem essa facilidade, como o Matlab, Scilab, e linguagens de programação geral. Além disso, a aplicação de função não precisa necessariamente de parênteses: `f(x) = fx`. Entretanto, deve-se tomar cuidado para entender quando que parênteses são necessários. O fato dessa linguagem ser de domínio bem específico facilita essas decisões.

A tabela 1 descreve as operações e suas ordens de precedência, com base na gramática.

Tabela 1 – Ordem das operações

Operações	Aridade	Associatividade	Exemplo	Descrição
`() [] {}`	Unário		`(expr)`	Isola a expressão interna
,	Binário	Esquerda	`(a,b,c)`	Adiciona uma elemento à tupla(dentro de parênteses)
+ -	Binário	Esquerda	`a+b`	Soma e subtração
justaposição	Binário	Esquerda	`ab`	Multiplicação justaposta
* /	Binário	Esquerda	`a*b`	Multiplicação e Divisão
+ - * /	Unário		`-x`, `*v`	Positivo, Negativo, Quadrado e Recíproco
aplicação	Binário	Esquerda	`sin x`	Aplicação de função
$\widehat{}$	Binário	Direita	`a^b`	Potenciação
_	Unário		`(1, 2, 3)_2`	Elemento de tupla
’ _	Unário		`sin’x + f_z(3)`	Derivada Total e Parcial

A estrutura `Parser` (3.4) define o parser.

Código 3.4 – Estrutura parcial do parser

```
struct Parser
{
    Lexer lexer;
    std::unique_ptr<Table> table = std::make_unique<Table>();
    std::vector<std::vector<Table*>> argList;
    Table *objType{};
    Table *objName{};
    Table *tag{};
    int tupleSize = 0;

    #define INIT(x, y) Table *x = table->initString(#x, TokenType::y);
        INIT(param,     DECLARE)
        INIT(pi,        CONSTANT)
        INIT(sqrt,      FUNCTION)
        /*.....*/
    #undef INIT

    Parser();
    void advance(bool match = true);

    typedef void Parse();

    void parseProgram(const char *source);

    Parse
        parseFDecl, parseParam, parseGrid, parseDefine /*.....*/;

    void parseInt(ExprType type);
    void parseInts(ExprType type);

    void parseMult(bool unary);

    virtual void actSyntaxError(TokenType type);
    virtual void actAdvance();
    virtual void actInt(ExprType type);
    virtual void actOp(ExprType type);
    virtual void actDecl();

    virtual ~Parser() = 0;
};
```

O membro `lexer` é o analisador léxico. O parser controla o avanço dos tokens diretamente. O membro `table` é a tabela de símbolos compartilhada pelos estágios da compilação. O membro `argList` é uma lista de listas identificadores. O atributo `argIndex`

de um token de função é um índice dessa lista, indicando a lista de parâmetros da função(exceto para as funções pré-definidas). Os membros `objType` e `objName` auxiliam o estágio da geração, e correspondem ao tipo de objeto e seu nome. O membro `tag` é o nome do argumento marcado em um intervalo do tipo tag. Os membros `param`, `pi`, `sqrt`, etc. são as palavras-chave, funções e constantes pré-definidas.

Os métodos com prefixo `parse` são as subrotinas dos não-terminais. A lógica do código foi simplificada, então não há uma correspondência exata. Os métodos com prefixo `act`, marcados com `virtual`, são implementados no estágio seguinte. Esses métodos são chamados de ações semânticas, e são invocados pelo parser quando uma estrutura sintática é detectada.

O método `advance` chama `actAdvance` e avança o token. Isso possibilita uma reação a um comentário ou a um token qualquer.

Quando uma função está sendo definida, os identificadores de seus argumentos passam a ser do tipo `VARIABLE`. Após a definição, são redefinidos para `UNDEFINED`, pelo método `removeArgs`.

3.4 Análise semântica e síntese

A análise semântica e síntese é responsável pela geração das estruturas de dados adequadas para a visualização dos objetos. A síntese é o estágio mais complexo do projeto.

Para os parâmetros, o compilador cria um controle deslizante na interface. Para os objetos desenháveis, cria as funções para a renderização.

A estrutura `Compiler` (3.5) define o compilador.

Código 3.5 – Estrutura parcial do compilador

```
1  struct Compiler : public Parser
2  {
3      std::vector<std::unique_ptr<SymbExpr>> symbExprs;
4      std::vector<std::unique_ptr<CompExpr>> compExprs;
5      std::vector<SymbExpr*> expStack;
6      std::vector<Interval> intStack;
7      std::vector<Obj> objects;
8      Size frameSize = {512, 512};
9
10     Buffer block{};
11     uint blockSize{};
12     /*.....*/
13
14     bool compiled = false;
15
16     void actInt(ExprType type);
17     void actOp(ExprType type);
18     void actDecl();
19
20     SymbExpr *newExpr(SymbExpr &e);
21     CompExpr *newExpr(CompExpr &e);
22     SymbExpr op(Parser::ExprType type, SymbExpr *a = nullptr, SymbExpr *
           b = nullptr, float number = 0, Table *name = nullptr);
23     CompExpr *op(CompExpr::ExprType type, CompExpr *a = nullptr,
           CompExpr *b = nullptr, float number = 0, Table *name = nullptr,
           int nTuple = 1);
24     CompExpr *_comp(CompExpr *e, unsigned int index, std::vector<Subst>
           &subs);
25     CompExpr *compute(SymbExpr *e, std::vector<Subst> &subs);
26     CompExpr *substitute(CompExpr *e, std::vector<Subst> &subs);
27     CompExpr *derivative(CompExpr *e, Table *var);
28     float calculate(CompExpr *e, std::vector<Subst> &subs);
29     void dependencies(CompExpr *e, std::vector<int> &grids, bool allow =
           false);
30
31     void compile(CompExpr *e, std::stringstream &str, int &v);
32     void compile(const char *source);
33     void header(std::stringstream &str);
34     void compileFunction(CompExpr *exp, int argIndex, std::stringstream
           &str, std::string name);
35     void declareFunction(int N, int argIndex, std::stringstream &str,
           std::string name, bool declareOnly = false);
36 };
```

Os métodos de prefixo `act` implementam as ações semânticas invocadas pelo parser.

`actInt` é a ação mais simples e apenas junta as informações para construír um intervalo. `actOp` gera as árvores de expressões matemáticas. `actDecl` junta as informações para contruír um objeto declarado.

Os métodos de `newExpr` a `dependencies` processam as expressões matemáticas para uma forma mais tratável. Para isso, derivadas são computadas simbolicamente, expressões com tuplas são descompactadas, e funções são aplicadas. O segundo método `op` é capaz de realizar otimizações simples nas expressões matemáticas.

Os métodos de `compile` a `declareFunction` geram o produto final. Juntos, geram os códigos-fonte na linguagem de *shader*(GLSL) do OpenGL. Por exemplo, a expressão matemática `x*3+1` é compilada para(aproximadamente) o seguinte código:

```
1 float v0 = x;
2 float v1 = 3;
3 float v2 = v0*v1;
4 float v3 = 1;
5 float v4 = v2+v3;
```

O código é gerado por um algoritmo na árvore da expressão, então o código-fonte gerado pode ser bem verboso/grande.

O segundo método `compile` é o método principal. Ele inicializa e invoca o parser. Após o parser finalizar seu trabalho, os objetos estão descritos numa estrutura de dados fácil de manipular. O método então compila as expressões matemáticas, e finalmente gera o produto final. Para os objetos desenháveis, gera as funções para desenhá-los. Para superfícies, compila também o geodesic tracing. Para os parâmetros, gera controles deslizantes na interface.

O compilador também verfica a validade semântica dos objetos. Por exemplo, intervalos não podem depender de parâmetros ou grades. Curvas e superfícies devem ter o número correto de parâmetros, e devem estar definidos em 3 dimensões.

4 Curvas e Superfícies

O objetivo primário do projeto é visualizar curvas e superfícies. As curvas são visualizadas apenas em espaço 3D. As superfícies são visualizadas em espaço 3D e em *geodesic tracing*.

4.1 Curvas

Há duas principais maneiras de se definir uma curva na geometria analítica: por parametrização e por equação. Esse trabalho apenas consideras curvas paramétricas.

Uma curva pode ser parametrizada por um número real. Formalmente, uma parametrização é uma função $\gamma : I \to \mathbb{R}^n$, onde I é um intervalo real. Nesse trabalho, o intervalo é fechado, e $n = 3$.

As curvas são desenhadas através de vários segmentos. Dada uma partição de I de k pontos, pode-se aproximar a curva pelos segmentos de extremidade $\gamma(t_i)$ e $\gamma(t_{i-1})$ para $i < k$, onde t_i é o i-ésimo ponto da partição. Nesse trabalho, a partição depende apenas de k e é uniforme.

O vetor tangente pode ser calculado com $\gamma'(t)$.

4.2 Superfícies

Assim como as curvas, apenas superfícies parametrizadas serão consideradas nesse trabalho: $\sigma : I_1 \times I_2 \to \mathbb{R}^3$, onde I_1 e I_2 são intervalos fechados reais.

As superfícies são desenhadas através de vários triângulos, a partir de partições dos intervalos I_1 e I_2. Juntas, as partições formam uma grade de retângulos, e cada retângulo pode ser dividido em 2 triângulos. Esses são os triângulos desenhados.

Os vetores tangentes nas direções coordenadas são as derivadas parciais $\sigma_u(u, v)$ e $\sigma_v(u, v)$, onde os parâmetros são u e v. Nesse projeto, os parâmetros podem ter nomes quaisquer.

4.2.1 Primeira forma fundamental

Supondo que a superfície seja diferenciável e com vetores tangentes linearmente independentes, a primeira forma fundamental no ponto paramétrico (u, v) é definida como

$$\begin{bmatrix} \sigma_u \cdot \sigma_u & \sigma_u \cdot \sigma_v \\ \sigma_v \cdot \sigma_u & \sigma_v \cdot \sigma_v \end{bmatrix} = \begin{bmatrix} E & F \\ F & G \end{bmatrix}$$

onde as funções são todas aplicadas no ponto (u, v).

Os vetores σ_u e σ_v formam uma base do espaço tangente. O produto escalar de dois vetores tangentes $x = x_1\sigma_u + x_2\sigma_v$ e $y = y_1\sigma_u + y_2\sigma_v$ pode ser calculado da seguinte forma:

$$x \cdot y = (x_1\sigma_u + x_2\sigma_v) \cdot (y_1\sigma_u + y_2\sigma_v)$$

$$x \cdot y = x_1y_1E + x_1y_2F + x_2y_1F + x_2y_2G$$

O produto depende apenas dos coeficientes e da primeira forma fundamental. Isso significa que distâncias e ângulos podem ser calculados sem se referir ao espaço ambiente da parametrização, ou seja, de forma intrínseca.

4.2.2 Rotação

Para a aplicação, é necessário rotacionar vetores no espaço tangente. É possível rotacionar um vetor apenas usando seus componentes e a primeira forma fundamental.

Seja $w = u\sigma_u + v\sigma_v$, e γ o ângulo da rotação. A base $\{\sigma_u, \sigma_v\}$ do espaço tangente pode ser ortogonalizada e normalizada. Assim, basta aplicar uma matriz de rotação.

A nova base pode ser escrita como:

$$\sigma'_u = \frac{\sigma_u}{E}$$
$$\sigma'_v = \frac{\sigma_u - \frac{F\sigma_v}{E}}{R}$$
$$R = \sqrt{EG - F^2}$$

Para achar os componentes de w na nova base, basta observar: $w = u'\sigma'_u + v'\sigma'_v = u'\frac{\sigma_u}{E} + v'\frac{\sigma_v}{R} - v'\frac{\sigma_u F}{RE}$

Então

$$u' = uE + vF$$
$$v' = vR$$

O vetor rotacionado é

$$w' = (u'\cos\gamma - v'\sin\gamma)\sigma'_u + (u'\sin\gamma + v'\cos\gamma)\sigma'_v$$

$$w' = (u\cos\gamma - \frac{uF + vG}{R}\sin\gamma)\sigma_u + (\frac{uE + vF}{R}\sin\gamma + v\cos\gamma)\sigma_v$$

Como esperado, esse vetor só depende da primeira forma fundamental, dos componentes originais e do ângulo de rotação.

4.2.3 Equação geodésica

...Apenas definir o sistema de EDOs e citar...

4.2.4 Solução Numérica

...Definir o método de Runge-Kutta de ordem 4...

4.2.5 Geodesic Tracing

...Definir o geodesic tracing...

5 Interface gráfica

...DEFINIR A INTERFACE GRÁFICA...

Referências

AHO, Alfred V. **Compilers: Principles, Techniques, & Tools**. [S.l.]: Pearson, 1986. Syntax Analysis.

BURDEN, Richard L. **Numerical Analysis**. [S.l.]: Cengage, 2001. Initial-Value Problems for Ordinary Differential Equations.

CALGARY, University of. **The Context Free Grammar Checker**. [S.l.: s.n.]. http://smlweb.cpsc.ucalgary.ca/. Acessado em 2022-09-28.

CORNUT, Omar. **ImGui**. [S.l.: s.n.]. https://github.com/ocornut/imgui. Acessado em 2022-10-04.

PRESSLEY, Andrew. **Elementary Differential Geometry**. [S.l.]: Springer, 2012. Geodesics.

UNIVERSITY, Western Michigan. **The syntax of C in Backus-Naur Form**. [S.l.: s.n.]. https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm. Acessado em 2022-09-28.