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 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 3.

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:

Objetos	Descrição
`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.

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

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: `sin(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.

2.1 Gramática formal

O programa deve seguir uma gramática formal, que especifica a sintaxe das declarações dos objetos e das expressões matemáticas. As expressões matemáticas podem seguir uma notação mais natural que as de várias linguagens de programação. Por exemplo, há multiplicação por justaposição: `3x = 3*x`; e a aplicação de funções de uma variável não exige parênteses: `sin-x = sin(-x)`.

A gramática livre de contexto é definida pelo código 2.2 (AHO, 1986). Uma parte da sintaxe das expressões matemáticas foi baseada na gramática da linguagem C (UNIVERSITY, s.d.).

Código 2.2 – 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 ;
```

Os termos em maiúsculo(não-terminais) representam variáveis gramaticais. O lado direito de uma igualdade especifica as possíveis formas sentenciais que um não-terminal pode assumir, separadas por uma barra vertical ou em diferentes equações. Uma forma sentencial(ou produção) é uma sequência de terminais e não terminais, possivelmente vazia. Por exemplo, `MULT` possui 3 formas: `MULT * UNARY`, `MULT / UNARY` e `UNARY`. Cada forma

tem um significado diferente. Uma forma pode ser vazia, como ocorre para `PROG`.

Os símbolos entre aspas representam textos literais, e os termos em minúsculo(terminais) representam uma classe de "palavras": Por exemplo, `num` representa um número e `var` o nome de uma variável.

O termo `PROG` representa um programa completo, que é uma sequência de declarações(`DECL`). O termo `EXPR` representa uma expressão matemática. Os símbolos abaixo de `EXPR` definem a sintaxe das operações, suas ordens de precedência e associatividades.

Para formar um programa gramaticalmente correto, inicia-se com o símbolo `PROG`. Cada não-terminal deve ser substituído no lugar por uma de suas formas sentenciais. O programa estará completo quando não houver mais não-terminais.

Para extraír o significado de um programa, o processo contrário deve ser feito. É necessário encontrar uma maneira de se obter o programa a partir de `PROG`. Para um programa coeso(gramaticalmente correto), sempre há uma maneira, que é única.

Algumas transformações nessa gramática a torna LL1, uma propriedade que garante que é possível fazer um *parsing* preditivo. Uma consequência desse fato é a gramática não ser ambígua. A verificação foi feita em (CALGARY, s.d.). 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 LL1, que falhou.

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
* /	Binário	Esquerda	`a*b`	Multiplicação e Divisão
+ - * /	Unário		`-x`, `*v`	Positivo, Negativo, Quadrado e Recíproco
aplicação	Binário	Esquerda	`sin x`	Aplicação de função
^	Binário	Direita	`a^b`	Potenciação
_	Unário		`(1, 2, 3)_2`	Elemento de tupla
' _	Unário		`sin'x + f_z(3)`	Derivada Total e Parcial

2.2 Compilador

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

- análise léxica: reconheçe as "palavras" que compõe um programa, ignorando comentários e espaços em branco. É capaz de identificar números, constantes, nomes de objetos, e pontuação.
- análise sintática: parser reconheçe a estrutura do programa: as declarações dos objetos e as expressões matemáticas. A gramática 2.2 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.

Os estágios se comunicam entre si, e compartilham uma tabela de símbolos. A tabela especifica quais identificadores estão definidos, e quais são seus tipos. Uma declaração de um objeto de nome `X` insere `X` na tabela, e o associa ao tipo do objeto. Para isso, `X` não pode estar definido antes de ser declarado. Identificadores não definidos são reconhecidos como `id` na gramática. Após a declaração, o mesmo identificador pode ser reconhecido como `const` ou `func`, dependendo do tipo de objeto. Isso significa que os estágios devem ser executados em conjunto, pois o nome de uma função é gramaticamente diferente do nome de uma constante, por exemplo. A tabela é inicializada com as constantes e funções pré-definidas e palavras-chave.

3 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*.

3.1 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 $\gamma'(t)$.

3.2 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.

3.2.1 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.

3.2.2 Equação geodésica

3.2.3 Solução Numérica

3.2.4 Geodesic Tracing

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.