FUNDAÇÃO GETULIO VARGAS
ESCOLA DE MATEMÁTICA APLICADA

CRISTHIAN GRUNDMANN

GEODESIC TRACING: UM SISTEMA DE ESPECIFICAÇÃO E VISUALIZAÇÃO DE CURVAS E SUPERFÍCIES ATRAVÉS DE GEODÉSICAS

Rio de Janeiro
2022

CRISTHIAN GRUNDMANN

GEODESIC TRACING: UM SISTEMA DE ESPECIFICAÇÃO E 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

Resumo

Curvas e superfícies costumam ser visualizados em um espaço ambiente 2D ou 3D. Esse projeto implementa essa visualização em 3D, e para superfícies, implementa também o *Geodesic Tracing*: uma visualização intrínseca à superfície, baseada em curvas geodésicas. Além de curvas e superfícies, o projeto permite visualizar pontos e vetores. Outros objetos auxiliares podem ser definidos, como parâmetros(controles deslizantes), funções e grades para instanciar objetos múltiplas vezes.

Para a especificação dos objetos, uma linguagem textual foi estabelecida, acompanhada de um compilador capaz de transformar o texto em estruturas de dados úteis para a renderização. A linguagem é descrita por uma gramática livre-de-contexto inambígua.

Para a interface gráfica, *OpenGL* é usado para a renderização, e *Dear ImGUI* é usado para construir os controles e janelas.

O resultado é um sistema de performance em tempo real, testado sem inconsistências. A estética dos gráficos, da interface e da linguagem não foram negligenciados, e se tornaram bastante agradáveis.

Palavras-chave: visualização. curvas. superfícies. compilador.

Abstract

Curves and surfaces are usually visualized in a 2D or 3D ambient space. This project implements this visualization in 3D, and for surfaces, also implements the *Geodesic Tracing*: an intrinsic visualization of the surface, based on geodesic curves. Points and vectors can also be visualized. Other auxiliary objects can be defined, like parameters(with slider controls), functions and grids to instantiate objects.

A textual language was designed for the specification of these objects, accompanied by a compiler capable of transforming the text into data structures that are useful for rendering. This language is described by an unambiguous context-free grammar.

For the graphical interface, *OpenGL* is used to do the rendering, and *Dear ImGUI* is used to build GUI components like buttons and windows.

The result is a system with real-time performance, tested without any inconsistencies. The aesthetic of the graphics, the interface and the language weren't overlooked, and became quite pleasing.

Keywords: visualization. curves. surfaces. compiler.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Sumário

1 Introdução

Existem diversos ambientes computacionais bem estabelecidos para a funcionalidade de desenho de curvas e superfícies a partir de descrições matemáticas, sejam paramétricas ou implícitas. Alguns exemplos são o software *Mathematica* (WOLFRAM, 2022), *Matlab* (MATHWORKS, 2022), seus similares em software livre(*Octave*, *Scilab*, linguagem *Julia*), e softwares de Geometria Dinâmica, como o *GeoGebra*, cuja descrição do objeto gráfico pode ser feito através da interface gráfica.

Tais softwares são compostos de três componentes cuja distinção nem sempre é transparente ao usuário, são elas: especificação de um objeto gráfico, renderização desse objeto, e sua interação com o usuário. Nesse projeto, curvas, superfícies, pontos ou vetores são especificados e então renderizados.

Tipicamente, a estratégia de renderização de superfícies costuma assumir um ponto de vista no espaço ambiente 3D como sendo o ponto de vista do observador. Uma forma muito comum de renderização é a discretização da curva ou superfície, formando segmentos no caso de uma curva, ou triângulos para superfícies.

O objetivo desse projeto é o desenvolvimento de um sistema de visualização de curvas e superfícies. Para tanto, esse projeto implementa um sistema que inclui a possibilidade de visualização de superfícies que não depende de um espaço ambiente. A visualização simula um observador posicionado sobre a superfície, que percorre o mundo restrito à essa dimensão. Para isso, simula-se raios de luz partindo da posição do observador, 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, estudados na geometria diferencial, e descritos no Capítulo 2. A visualização, chamada de *Geodesic Tracing*, renderiza a imagem sobre a superfície, e suas curvaturas podem ser notadas. Ao se mover, a imagem observada pode se distorcer, dependendo da curvatura.

A implementação desse 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, que então é processado. O texto deve seguir uma gramática livre-de-contexto, cuja in-ambiguidade foi provada. A teoria de compiladores é essencial para essa etapa, principalmente a análise léxica e a análise sintática (AHO, 1986). O compilador está descrito no Apêndice A. A linguagem de especificação dos objetos gráficos, com exemplos de programas, está descrita no Capítulo 3.

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 *Dear ImGUI* (CORNUT, 2022), 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. A interface está descrita no Capítulo 5.

O objetivo primário desse projeto é a visualização de curvas, superfícies, e o *Geodesic Tracing*. Porém, a estética dos gráficos e da linguagem descritiva, performance e robustez do sistema também são levados em consideração.

A Figura 1 demonstra a interface gráfica.

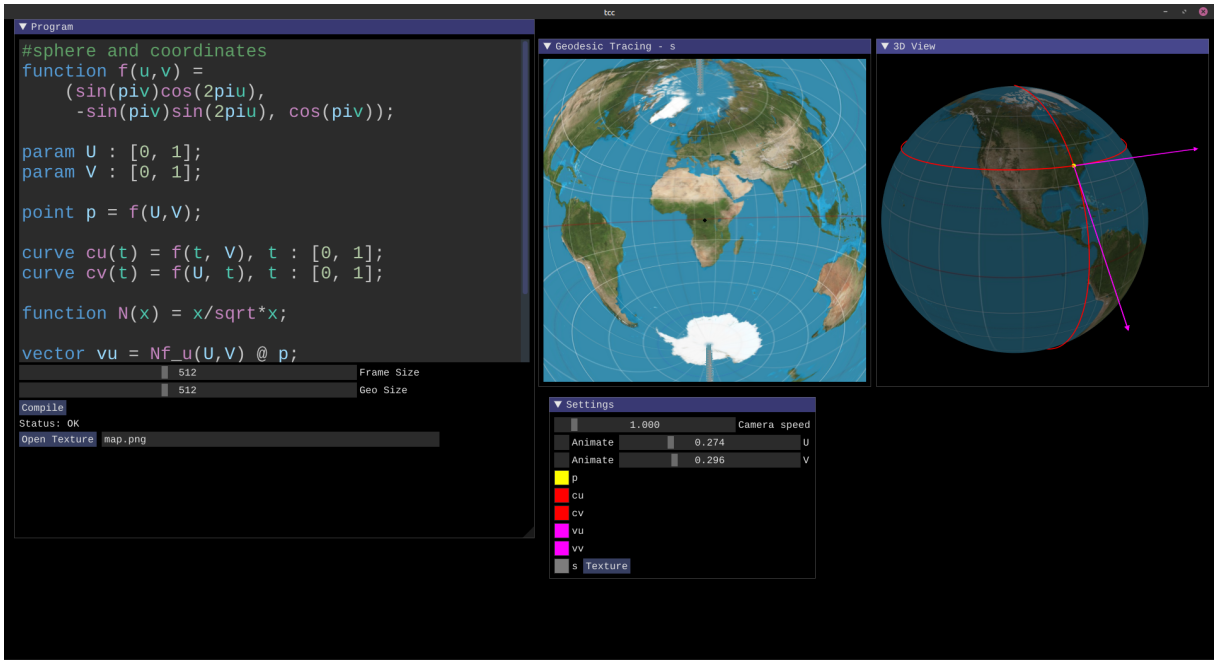


Figura 1 – Exemplo da visualização

A Figura 2 representa os estágios da aplicação.

Figura 2 – Estágios da aplicação

A entrada se refere ao texto que o usuário escreve para definir os objetos matemáticos. O texto é digitado na própria interface gráfica, numa caixa de texto multilinha e com *Syntax Highlighting*: cores associadas às palavras e símbolos.

A compilação se refere ao processamento da entrada. Eventualmente, o usuário pode cometer erros sintáticos ou semânticos. Esses erros são detectados no compilador, e uma mensagem de erro é emitida. Se o texto for válido, várias estruturas de dados são geradas, com todas as informações relevantes sobre os objetos. Além disso, *Shaders* são gerados e compilados. Cada *shader* faz a renderização de um objeto. Para superfícies, um outro *shader* é necessário para fazer o *Geodesic Tracing*.

A saída se refere ao produto final da compilação. Todos os objetos são desenhados na interface gráfica. Além disso, o usuário pode controlar vários aspectos da visualização, como a câmera, e controles deslizantes para os parâmetros.

Todos os arquivos necessários para a compilação dessa aplicação se encontram no repositório público (GRUNDMANN, 2022). Todos os códigos são abertos e livres.

2 Caminhos Geodésicos

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*.

O estudo da geometria diferencial de curvas e superfícies é essencial para o *Geodesic Tracing*, e a teoria foi baseada em (PRESSLEY, 2012).

2.1 Curvas

Há duas principais maneiras de se definir uma curva na geometria analítica: por equação paramétrica ou equação implícita. Esse trabalho apenas considera curvas paramétricas.

Definição 1. Uma curva paramétrica γ é uma função $\gamma : I \to \mathbb{R}^n$, onde I é um intervalo real.

Definição 2. Numa curva paramétrica γ, o vetor tangente pode ser facilmente calculado com $\gamma'(t)$. Uma curva γ é regular quando é diferenciável e $\gamma'(t) \neq 0$ para todo $t \in I$.

Nesse trabalho, o intervalo é fechado, e $n = 3$. Isso significa que todas as curvas devem ser definidas no espaço 3D. Para desenhar curvas planares, uma possibilidade é defini-las no plano xy, fazendo $z = 0$.

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. Essa escolha foi feita pois essa é a forma mais simples de se implementar. Porém, dependendo da curvatura de γ, o intervalo fixo entre os pontos pode ser demasiadamente pequeno, custando performance, ou demasiadamente grande, degradando a estética.

2.2 Superfícies

Assim como as curvas, apenas superfícies parametrizadas regulares serão consideradas nesse trabalho, e apenas definidas em um espaço ambiente 3D.

Definição 3. Uma superfície parametrizada regular σ é uma função suave $\sigma : I_1 \times I_2 \to \mathbb{R}^3$, onde I_1 e I_2 são intervalos fechados reais. Além disso, os vetores tangentes $\sigma_u(u, v)$ e $\sigma_v(u, v)$ devem ser linearmente independentes.

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. De forma similar às curvas, essa escolha é a mais simples. Há maneiras mais inteligentes de escolher as partições, considerando os vetores tangentes, seus tamanhos e o ângulo formado pelos vetores.

2.2.1 Primeira forma fundamental

Definição 4. Numa superfície parametrizada regular, 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.

2.2.2 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 $w = u\sigma_u + v\sigma_v$ o vetor a ser rotacionado e θ o ângulo da rotação. A base $\{\sigma_u, \sigma_v\}$ do espaço tangente pode ser ortogonalizada. Com uma base ortogonal, a rotação é feita com a matriz de rotação.

A nova base pode ser escrita como:

$$\sigma'_u = \frac{\sigma_u}{E}$$

$$\sigma'_v = \frac{\sigma_v - \frac{F\sigma_u}{E}}{R}$$

$$R = \sqrt{EG - F^2}$$

Note que os vetores são ortogonais e de mesmo comprimento. Apesar do comprimento não ser 1, a matriz de rotação funciona corretamente.

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

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

$$w' = (u'\cos\theta - v'\sin\theta)\,\sigma'_u + (u'\sin\theta + v'\cos\theta)\,\sigma'_v$$
$$w' = \left(u\cos\theta - \frac{uF + vG}{R}\sin\theta\right)\sigma_u + \left(\frac{uE + vF}{R}\sin\theta + v\cos\theta\right)\sigma_v$$

Definição 5. Seja $w = u\sigma_u + v\sigma_v$ um vetor tangente à superfície σ. O vetor w rotacionado pelo ângulo θ é

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

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

2.2.3 Transporte Paralelo

Definição 6. Seja γ uma curva sobre a superfície e v um campo vetorial tangente à σ definido sobre a curva γ. Diz-se que v é paralelo ao longo de γ quando v' é ortogonal à superfície σ para qualquer ponto de γ. Ou seja,

$$v'(t) \perp \{\sigma_u(\gamma(t)), \sigma_v(\gamma(t))\}$$

para todo ponto $t \in I$ da curva.

Nesse caso, um ser sobre a superfície não observaria mudanças em v ao traçar a curva γ. A variação de v se dá ortogonalmente à superfície, logo não seria percebida pelo ser. Diz-se que o vetor v está sendo transportado paralelamente ao longo de γ.

Sejam v e w vetores paralelos ao longo de γ(transportados paralelamente). Então o produto escalar $v \cdot w$ é constante, pois $(v \cdot w)' = v' \cdot w + v \cdot w' = 0$. Como o produto escalar pode definir as noções de comprimento e ângulo, vetores transportados paralelamente à uma curva mantém seus comprimentos e ângulos entre si.

2.2.4 Equação de um caminho geodésico

Definição 7. Uma curva regular γ sobre a superfície é dita geodésica quando γ' é paralelo ao longo de γ.

Nesse caso, um ser perceberia γ'(velocidade) como constante, e a curva γ pode ser considerada como reta nesse espaço.

Uma curva γ é uma curva geodésica quando satisfaz o sistema 2.1 (PRESSLEY, 2012).

$$\begin{cases} \frac{1}{2}\left(E_u(u')^2 + 2F_u u'v' + G_u(v')^2\right) &= \frac{d}{dt}(Eu' + Fv') \\ \frac{1}{2}\left(E_v(u')^2 + 2F_v u'v' + G_v(v')^2\right) &= \frac{d}{dt}(Fu' + Gv') \end{cases} \tag{2.1}$$

Esse sistema pode ser reescrito como o sistema 2.2.

$$2\begin{pmatrix} E & F \\ F & G \end{pmatrix}\begin{pmatrix} u'' \\ v'' \end{pmatrix} = \begin{pmatrix} v'^2(G_u - 2F_v) - u'^2E_u - 2u'v'E_v \\ u'^2(E_v - 2F_u) - v'^2G_v - 2u'v'G_u \end{pmatrix} \tag{2.2}$$

Como a superfície é regular, u'' e v'' são únicos, pois a primeira forma fundamental é invertível.

Note que a aceleração (u'', v'') pode ser obtida em função da posição (u, v) e velocidade (u', v'): $(u'', v'') = g(u, v, u', v')$.

Esse é um sistema de equações diferenciais ordinárias de primeira ordem. Sua solução depende de uma posição e uma velocidade inicial. Na maioria das superfícies interessantes, esse sistema é muito difícil, ou até impossível, de resolver analiticamente.

2.2.5 *Geodesic Tracing*

O *Geodesic Tracing* gera uma imagem a partir de duas informações:

- um ponto (u, v) sobre a superfície, representando a posição da câmera
- dois vetores tangentes X e Y no ponto (u, v). Os vetores X e Y são de mesmo comprimento(z), são ortogonais entre si e representam a orientação da câmera.

A imagem gerada é um quadrado $[-1, +1] \times [-1, +1]$. O primeiro eixo corresponde ao vetor X, e o segundo ao Y.

Cada ponto (x, y) é associado a um ponto (u_1, v_1) da superfície. O ponto (u_1, v_1) é determinado traçando-se uma curva geodésica γ. A posição inicial de γ é (u, v), e a velocidade inicial é $xX + yY$. Finalmente, o ponto (u_1, v_1) é definido como $\gamma(1)$. A

distância percorrida, ou comprimento de arco, é o próprio comprimento da velocidade inicial: $z\sqrt{x^2 + y^2}$.

A Figura 3 ilustra o *Geodesic Tracing*. Os vetores em rosa são os vetores X e Y, ancorados na câmera, o ponto verde. O vetor azul é uma combinação linear de X e Y, e portanto está no espaço tangente. O ponto da superfície observado pelo vetor azul é o final da curva vermelha. A curva é uma geodésica com velocidade inicial igual ao vetor azul, partindo da câmera.

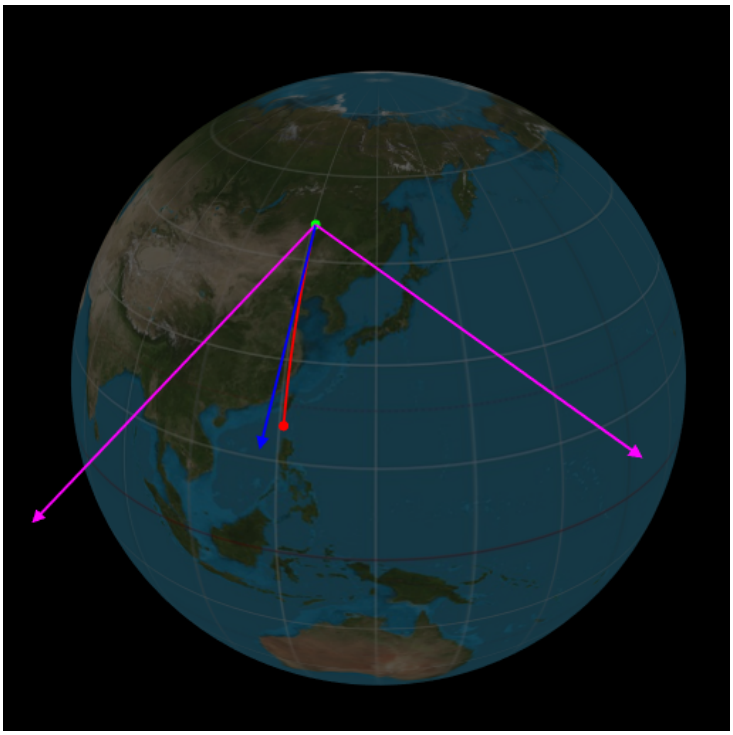


Figura 3 – Os vetores rosa são os vetores X e Y. O ponto verde é o centro da câmera. O vetor azul representa um ponto da imagem, e corresponde ao ponto vermelho. O ponto é obtido pela curva geodésica vermelha, que é determinada pelo vetor azul e pelo ponto verde.

A visualização do *Geodesic Tracing* requer uma imagem sobre a superfície. A cor observada pelo vetor azul, na Figura 3, é a cor da superfície no ponto vermelho, que é um ponto da imagem original.

2.2.6 Interação

Para compreender melhor o *Geodesic Tracing*, é necessário alterar as condições iniciais e observar as alterações na imagem.

A interação mais simples é a rotação. Os vetores X e Y são apenas rotacionados por um ângulo θ. A imagem resultante não se deforma, apenas é rotacionada pelo mesmo ângulo θ.

Outra interação simples é o *zoom*. Os vetores X e Y são multiplicados por um fator $\alpha > 0$. Para $\alpha > 1$, a imagem é ampliada, pois as curvas geodésicas são maiores. Para $\alpha < 1$, a imagem é contraída.

A interação mais interessante é o movimento. Para se mover na direção X, uma geodésica γ é traçada na direção X. O novo centro da imagem passa a ser $\gamma(t)$, onde t é o tamanho do passo. O novo vetor X é a velocidade final($\gamma'(t)$), que foi transportada paralelamente ao longo de γ. O comprimento de X foi preservado. O novo vetor Y também é transportado paralelamente. Como visto anteriormente, seu comprimento é preservado e seu ângulo com X também. Assim, o novo Y é apenas a rotação de X pelo ângulo de 90 graus. O comprimento real do passo é zt. De forma similar, a câmera pode se movimentar ao longo de Y.

As curvaturas da superfície causam distorções na imagem observada. Ao se mover, pode-se perceber curvatura pela forma com que a imagem se distorce. Curvatura gaussiana positiva faz os "objetos" expandirem ao se afastar. Curvatura negativa faz os "objetos" contraírem ao se afastar. Curvatura nula faz os "objetos" se moverem de forma rígida. Além disso, é possível fazer um "passeio paralelo": apenas movimentos geodésicos e sem rotacionar a câmera. Se um passeio paralelo cobre uma área e retorna à posição inicial, observa-se uma rotação da câmera em relação à direção inicial. Essa rotação depende da curvatura da superfície na área envolvida. Uma área de curvatura positiva causa rotações no mesmo sentido do caminho. O contrário pode ser observado em curvatura negativa, a câmera rotaciona no sentido oposto ao sentido do caminho: ao andar num círculo em sentido horário, a câmera é rotacionada do sentido anti-horário.

No caso de um cone, é possível observar uma geometria localmente plana, ou de curvatura nula. Ao se mover, a imagem se deforma(localmente) rigidamente. Um passeio paralelo não causa rotações quando a ponta do cone não está no interior do caminho. Nesse caso, a distância da ponta ao interior da região é completamente irrelevante. Porém, se a ponta do cone estiver no interior da região percorrida, observa-se uma rotação de ângulo fixo. Nesse caso, a distância da ponta ao exterior da região é irrelevante também.

O domínio de parametrização é desprezado no *Geodesic Tracing*. A fórmula da parametrização é usada no plano uv todo, supondo-se que a fórmula não faça operações indefinidas, como raiz quadrada de números negativos ou divisão por zero. A textura deve ser um arquivo de imagem, sempre retangular. No quadrado $[0, 1] \times [0, 1]$ se encontra a imagem/textura original. Se a textura não for quadrada, ela será distorcida. A textura é repetida para cobrir o plano, simplesmente transladando o quadrado, sem mudar a orientação da textura.

3 Linguagem de especificação dos objetos gráficos

O usuário se comunica com a interface através de um texto, chamado de programa, que contém os objetos de interesse. Por exemplo, em *GeoGebra*, o círculo unitário pode ser definido como `c = Curve(cos(t), sin(t), t, 0, 2pi)`. Na linguagem desse projeto, a definição seria `curve c(t) = (cos(t), sin(t), 0), t : [0, 2pi];`.

O Código 1 é um exemplo.

```
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];
```

Código 1 – Código para a Figura 4

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

O programa declara os seguintes objetos:

Tabela 1 – Objetos do Código 1

`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 4 demonstra os objetos declarados em perspectiva 3D.

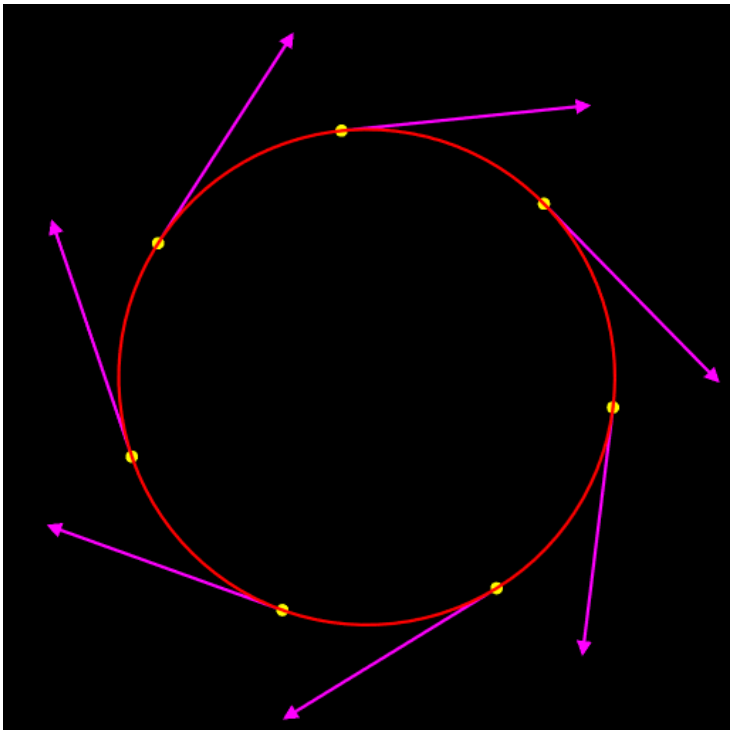


Figura 4 – Renderização do Código 1

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. Os parâmetros das curvas e superfícies podem ter qualquer nome disponível.
- 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;`.

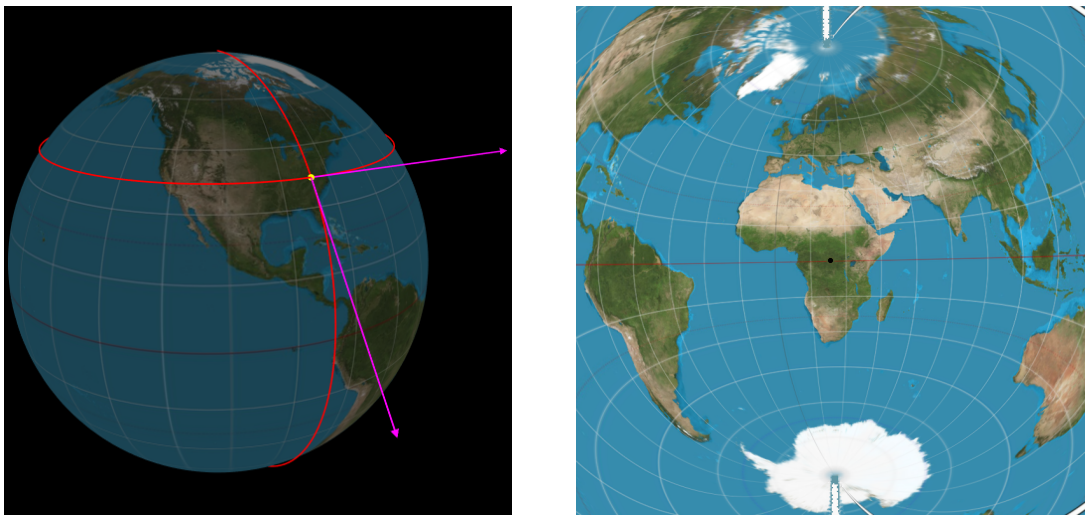
- Numa aplicação de função de uma variável, o argumento não precisa de parênteses: `sin x`. O argumento pode ter operadores unários e até expoentes: `sin -x^2 = sin(-x^2)`. Deve-se tomar cuidado com expoentes: `sin(x)^y = sin(x^y)`. Para a exponenciação de uma aplicação, deve-se usar a sintaxe: `sin^2 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.

O Código 2 é outro exemplo.

```
#sphere and coordinates
function f(u,v) =
     (sin(piv)cos(2piu), sin(piv)sin(2piu), cos(piv));

param U : [0, 1];
param V : [0, 1];
point p = f(U,V);

curve cu(t) = f(t, V), t : [0, 1];
curve cv(t) = f(U, t), t : [0, 1];

function N(x) = x/sqrt*x;
vector vu = Nf_u(U,V) @ p;
vector vv = Nf_v(U,V) @ p;

surface s(u,v) = f(u,v)*0.99, u : [0, 1], v : [0, 1];
```

Código 2 – Código para as Figuras 5a e 5b



(a) Perspectiva em 3D (b) *Geodesic Tracing*

Figura 5 – A Figura 5a à esquerda é a renderização em perspectiva 3D. A Figura 5b à direita é a renderização em *Geodesic Tracing*

4 Cálculo numérico de geodésicas

A equação 2.2 caracteriza as curvas geodésicas que devem ser computadas para a visualização do *Geodesic Tracing*. Nem sempre é possível resolver a equação de forma analítica, então uma aproximação deve ser computada no lugar.

O método numérico escolhido é o método de Runge-Kutta de ordem 4 (BURDEN, 2001). Seja $y' = f(t, y)$ uma equação diferencial para $y(t)$ com valores iniciais t_0 e y_0. O método consiste em aproximar $y_1 = y(t_0 + h)$, para um passo $h > 0$. A aproximação é feita pelas equações 4.1.

$$\begin{aligned} k_1 &= hf(t_0, y_0) \\ k_2 &= hf(t_0 + h/2, y_0 + k_1/2) \\ k_3 &= hf(t_0 + h/2, y_0 + k_2/2) \\ k_4 &= hf(t_0 + h, y_0 + k_3) \\ y_1 &= y_0 + (k_1 + 2k_2 + 2k_3 + k_4)/6 \\ t_1 &= t_0 + h \end{aligned} \tag{4.1}$$

Após um passo, o erro de y_1 estimado com o valor real é da ordem de $O(h^4)$. Como a aproximação é melhor para $h \to 0$, esse resultado garante uma precisão muito melhor que ao método de Euler, que possui erro na ordem de $O(h)$.

As equações devem ser adaptadas para o sistema 2.2, pois é de segunda ordem e possui mais de uma equação. O sistema pode ser escrito vetorialmente:

$$y' = (u, v, u', v')' = f(u, v, u', v') = f(y)$$

Ou ainda

$$y' = (\text{pos}, \text{vel})' = f(\text{pos}, \text{vel}) = f(y)$$

A função f computa $(u', v', u'', v'') = (\text{vel}, \text{acc})$, onde vel é igual ao argumento vel de f. Os valores u'' e v'' são calculados conforme a equação 2.2, usando pos e vel. Note que a função f não depende do parâmetro t da curva.

O sistema pode ser resolvido numericamente por Runge-Kutta, ignorando a variável

t, e fazendo y como (pos, vel).

$$\begin{aligned}
p_1 &= h\text{vel}_0 \\
v_1 &= hg(\text{pos}_0, \text{vel}_0) \\
p_2 &= h(\text{vel}_0 + v_1/2) \\
v_2 &= hg(\text{pos}_0 + p_1/2, \text{vel}_0 + v_1/2) \\
p_3 &= h(\text{vel}_0 + v_2/2) \\
v_3 &= hg(\text{pos}_0 + p_2/2, \text{vel}_0 + v_2/2) \\
p_4 &= h(\text{vel}_0 + v_3) \\
v_4 &= hg(\text{pos}_0 + p_3, \text{vel}_0 + v_3) \\
\text{pos}_1 &= (p_1 + 2p_2 + 2p_3 + p_4)/6 \\
\text{vel}_1 &= (v_1 + 2v_2 + 2v_3 + v_4)/6
\end{aligned} \tag{4.2}$$

A implementação do sistema 4.2 define um passo do método de Runge-Kutta, e é feita no Código 3

```
1  //Runge-Kutta 4 step on surfaces
2  void step(Obj &o, vec2 &pos, vec2 &vec, float h)
3  {
4      vec2 k1_pos = h*vec;
5      vec2 k1_vec = h*accel(o, pos, vec);
6      vec2 k2_pos = h*(vec+k1_vec/2.0f);
7      vec2 k2_vec = h*accel(o, pos+k1_pos/2.0f, vec+k1_vec/2.0f);
8      vec2 k3_pos = h*(vec+k2_vec/2.0f);
9      vec2 k3_vec = h*accel(o, pos+k2_pos/2.0f, vec+k2_vec/2.0f);
10     vec2 k4_pos = h*(vec+k3_vec);
11     vec2 k4_vec = h*accel(o, pos+k3_pos, vec+k3_vec);
12     pos += (k1_pos + 2.0f*k2_pos + 2.0f*k3_pos + k4_pos)/6.0f;
13     vec += (k1_vec + 2.0f*k2_vec + 2.0f*k3_vec + k4_vec)/6.0f;
14 }
```

Código 3 – Implementação do método de Runge-Kutta

No código, o argumento `o` representa o objeto da superfície dado pelo compilador. Os argumentos `pos`, `vel` e `h` são as variáveis correspondentes ao sistema 4.2. A função `accel` obtém a aceleração em função da posição e da velocidade (condições iniciais).

5 Interface gráfica

A interface gráfica é responsável pela interação com o usuário. Nela, o usuário escreve o programa, visualiza e interage com os objetos. A Figura 6 ilustra os componentes da interface gráfica.

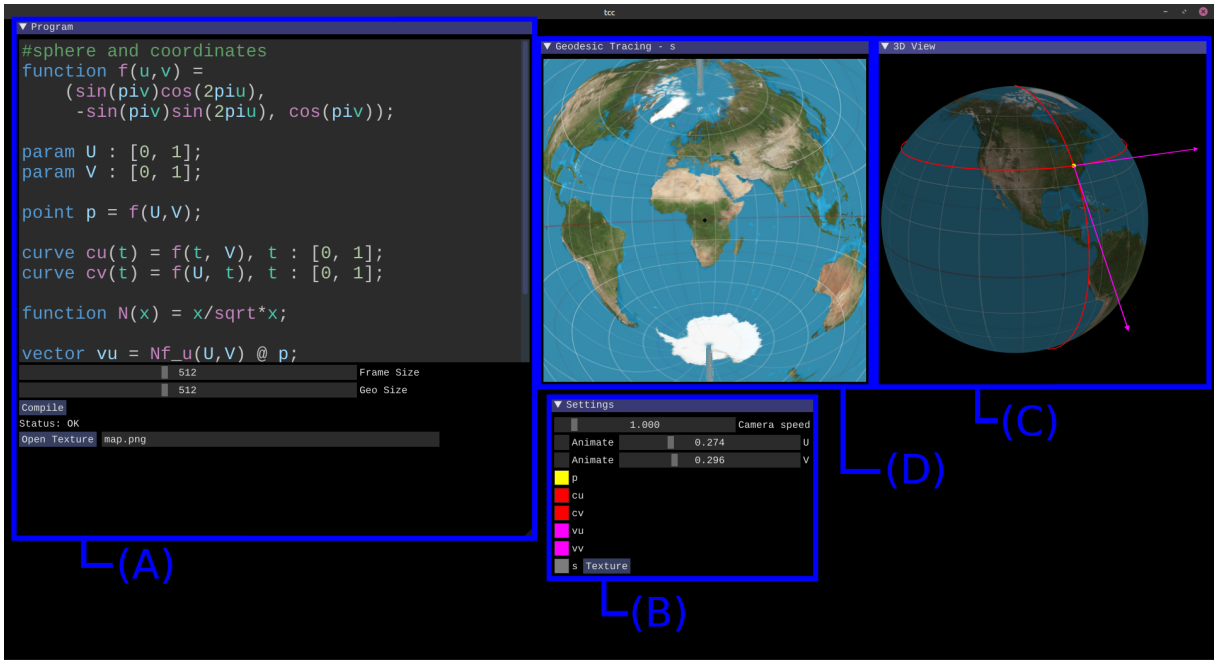


Figura 6 – Componentes da interface gráfica

A Janela (A) da Figura 6 ilustra a janela `"Program"`, que contém uma caixa de texto multilinha, onde o usuário deve escrever o programa. O botão `"Compile"` compila o texto escrito. A resposta do compilador será `"Status: OK"` para um programa compilado corretamente, ou `"Status: ..."` com uma mensagem de erro, indicando a linha e a coluna correspondentes. Dois controles deslizantes definem o tamanho, em *pixels*, da janela de visualização 3D(`"Frame size"`) e do *Geodesic Tracing*(`"Geo Size"`). O botão `"Open Texture"` serve para carregar uma imagem local, com o nome informado à caixa de texto justaposta. Quando o programa é compilado corretamente, duas janelas extras são exibidas.

A Janela (B) da Figura 6 ilustra a janela `"Settings"`, que controla algumas propriedades dos objetos. Para os parâmetros, um controle deslizante é criado, com a opção `"Animate"`. Quando ativado, o parâmetro é controlado pelo tempo, crescendo uma unidade por segundo. Quando o limite superior do parâmetro é atingido, o valor volta para o limite inferior.

Para cada objeto desenhável é possível escolher uma cor.

Para uma superfície, os componentes RGB da textura são multiplicadas pelos componentes RGB da cor escolhida. A cor branca deixa a textura inalterada, e preto deixa

a textura completamente preta. Além disso, é possível escolher uma textura previamente carregada para uma superfície. A textura padrão é a textura local de nome `"default.png"`.

A velocidade da câmera é controlada por `"Camera speed"`.

A Janela (C) da Figura 6 ilustra a janela `"3D View"`, que exibe os objetos desenháveis no espaço 3d a partir de uma câmera. A câmera pode ser controlada da seguinte forma:

Tabela 2 – Controles da câmera 3D

`W`	move a câmera para frente
`S`	move a câmera para trás
`A`	move a câmera para a esquerda
`D`	move a câmera para a direita
`Q`	move a câmera para cima (absoluto)
`E`	move a câmera para baixo (absoluto)
`Clicar e mover`	gira a câmera conforme o movimento do mouse

A renderização dos objetos utiliza um anti-serrilhado, melhorando sua estética. Além disso, o desenho das linhas considera uma grossura que leva em consideração a perspectiva, assim como os pontos e vetores.

Para uma superfície $\boldsymbol{x}$, a janela `"Geodesic Tracing - `$\boldsymbol{x}$`"` é exibida. A janela exibe a visualização do *Geodesic Tracing*, e está ilustrada na Janela (D) da Figura 6.

A câmera pode ser controlada da seguinte forma:

Tabela 3 – Controles do *Geodesic Tracing*

`W`	move a câmera para frente
`S`	move a câmera para trás
`A`	move a câmera para a esquerda
`D`	move a câmera para a direita
`Q`	gira a câmera no sentido anti-horário
`E`	gira a câmera no sentido horário
`Z`	zoom in
`X`	zoom out
`Clicar e mover`	move a câmera conforme o movimento do mouse

Dependendo da expansão ou contração da imagem gerada no *Geodesic Tracing*, o tamanho dos *pixels* pode ficar grandes ou pequenos demais. O primeiro caso deixa os *pixels* individuais evidentes, e o segundo gera ruído, pois *pixels* distantes de cores muito diferentes 'competem' para serem exibidos. A exibição da textura é feita com filtro de magnificação linear, fazendo as transições de *pixels* grandes mais suave, resolvendo o primeiro problema. O segundo problema foi resolvido com a técnica de *mipmapping* linear. Nessa técnica, a textura é copiada em resoluções progressivamente menores. Por exemplo, um tabuleiro

de xadrez possui casas brancas e pretas, e sua menor resolução é apenas um *pixel* cinza. Desse modo, quando a textura está muito contraída, uma versão de resolução menor é usada, resolvendo o ruído. Essas técnicas podem ser facilmente obtidas configurando o *OpenGL*, como pode ser observado em (VRIES, 2022).

6 Conclusão

A interface gráfica, a linguagem de especificação e o compilador funcionam sem irregularidades ou inconsistências, e possuem performance em tempo-real. Além disso, a estética da interface e da linguagem foi levada em consideração: a linguagem possui *Syntax Highlighting*, e a interface é construída usando uma biblioteca terceira. O desenho de objetos gráficos é feito com anti-serrilhado, melhorando a estética. Para o *Geodesic Tracing*, é possível observar os fenômenos de curvatura correspondentes à teoria da geometria diferencial.

Uma das limitações desse projeto é a linguagem de especificação. Curvas e superfícies devem ser definidas por equações paramétricas. Muitas curvas e superfícies são melhores definidas implicitamente. Outra limitação é o fato do programa ser estrito apenas em forma textual, sem poder utilizar notações matemáticas como por exemplo, subscritos e expoentes, frações e raízes.

Um possível trabalho futuro é aprimorar a discretização das curvas e superfícies. Assim, cada segmento ou triângulo se ajusta melhor à curva ou superfície. Para o desenho das superfícies em 3D, truques de textura podem ser usados. Com os vetores normais à superfície, é possível fazer um sombreado na superfície(sem projeção de sombras), aumentando a compreensão e o realismo da forma da superfície. Com isso, é possível reduzir a quantidade de segmentos ou triângulos da discretização sem degradar o realismo. O *Geodesic Tracing* desconsidera o domínio da parametrização da superfície. Isso significa que superfícies como a faixa de *Möbius* e a garrafa de *Klein* não são exibidas de forma totalmente correta, pois deveria ser possível inverter a orientação da câmera ao dar um passeio paralelo.

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.], 2022. Acessado em 2022-09-28. Disponível em: <http://smlweb.cpsc.ucalgary.ca/>.

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

GRUNDMANN, Cristhian. **TCC**. [S.l.: s.n.], 2022. Acessado em 2022-11-29. Disponível em: <https://github.com/cristhiangrundmann/tcc>.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. **The syntax of C in Backus-Naur Form**. [S.l.: s.n.], 2022. Acessado em 2022-09-28. Disponível em: <https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm>.

MATHWORKS. **Matlab**. [S.l.: s.n.], 2022. Acessado em 2022-12-1. Disponível em: <https://www.mathworks.com/products/matlab.html>.

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

VRIES, Joey de. **Learn OpenGL**. [S.l.: s.n.], 2022. Acessado em 2022-11-28. Disponível em: <https://learnopengl.com/>.

WOLFRAM. **Mathematica**. [S.l.: s.n.], 2022. Acessado em 2022-12-1. Disponível em: <https://www.wolfram.com/mathematica/>.

Apêndices

APÊNDICE A – Compilador

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

- **análise léxica**: reconhece 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**: reconhece a estrutura do programa, como as declarações dos objetos e as expressões matemáticas. A gramática 6 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.

A teoria de compiladores é essencial para esses estágios (AHO, 1986).

A.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:

Tabela 4 – Tipos de *token*

`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`(4) define o analisador léxico.

```
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 };
```

Código 4 – Estrutura do *lexer*

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

Tabela 5 – Estrutura do *token*

`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(5), 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 estende 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.

A.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`(5) define a 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 };
```

Código 5 – Tabela de símbolos

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 raiz. 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 raiz, o caractere é nulo(0). O membro `parent` é o pai do nó, ou nulo para a raiz. 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`.

Para o exemplo 1, a tabela é a trie ilustrada na Figura 7.

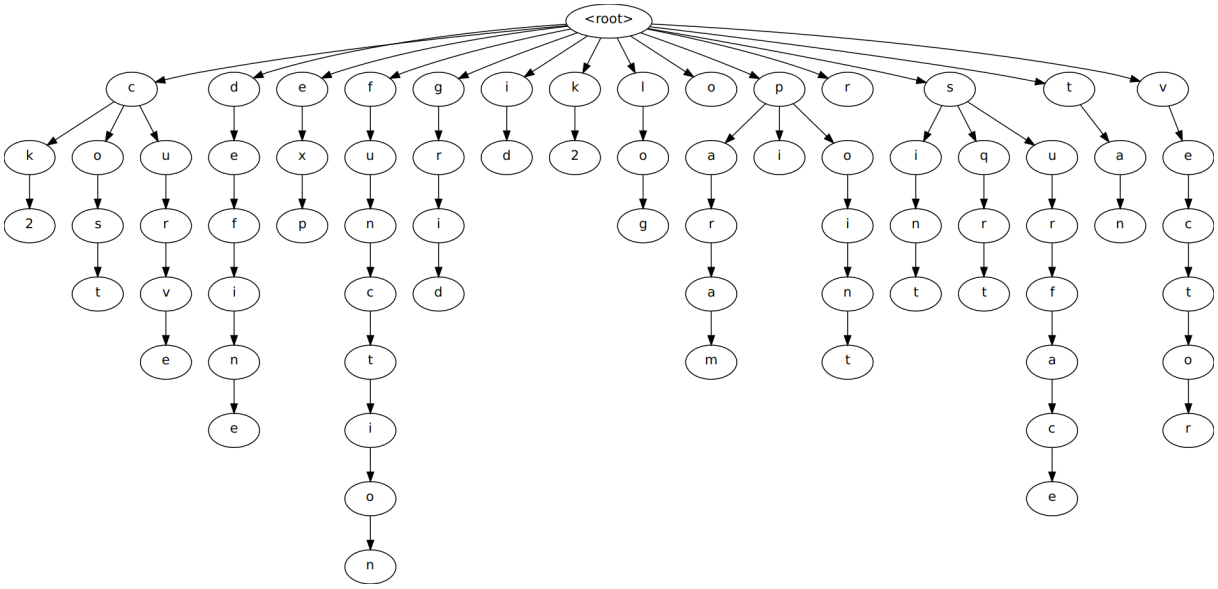


Figura 7 – Tabela de símbolos do Código 1

Nessa Figura, o nó correspondente ao termo `cos` indica `type=FUNCTION`, e para `k2`, `type=CONSTANT`.

A.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 (6) define as regras gramaticais da linguagem.

```
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 ;
```

Código 6 – Gramática livre-de-contexto

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 (6) é inambígua. A verificação foi feita em (CALGARY, 2022). Algumas transformações na gramática a fizeram ser uma gramática LL(1). Uma consequência disso é a in-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 sub-rotina. As sub-rotinas simulam a substituição de seu não-terminal por uma de suas formas sentenciais possíveis. Ou seja, uma sub-rotina simula uma produção de seu não-terminal. Para decidir qual produção aplicar, as sub-rotinas 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 sub-rotina 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 sub-rotinas.

Por exemplo, considere a produção `FUNC = FUNC2 ^UNARY`. Para simulá-la, deve-se derivar `FUNC2`, chamando sua sub-rotina. Após a sub-rotina terminar, o *token* atual é comparado com `^`, e caso seja igual, o *lexer* avança para o próximo *token*. Em seguida, a sub-rotina `UNARY` é chamada. No final de uma sub-rotina, 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 (KERNIGHAN; RITCHIE, 2022). 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 6 descreve as operações e suas ordens de precedência, com base na gramática.

Tabela 6 – 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
$\hat{}$	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` (7) define o *parser*.

```
1  struct Parser
2  {
3      Lexer lexer;
4      std::unique_ptr<Table> table = std::make_unique<Table>();
5      std::vector<std::vector<Table*>> argList;
6      Table *objType{};
7      Table *objName{};
8      Table *tag{};
9      int tupleSize = 0;
10
11     #define INIT(x, y) Table *x = table->initString(#x, TokenType::y);
12         INIT(param,     DECLARE)
13         INIT(pi,        CONSTANT)
14         INIT(sqrt,      FUNCTION)
15         /*.....*/
16     #undef INIT
17
18     Parser();
19     void advance(bool match = true);
20
21     typedef void Parse();
22
23     void parseProgram(const char *source);
24
25     Parse
26         parseFDecl, parseParam, parseGrid, parseDefine /*.....*/;
27
28     void parseInt(ExprType type);
29     void parseInts(ExprType type);
30
31     void parseMult(bool unary);
32
33     virtual void actAdvance();
34     virtual void actInt(ExprType type);
35     virtual void actOp(ExprType type);
36     virtual void actDecl();
37
38     virtual ~Parser() = 0;
39 };
```

Código 7 – Estrutura parcial do *parser*

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 sub-rotinas 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. O compilador não usa essa ação semântica. Essa ação semântica é usada no *syntax highlighting*, uma outro "compilador", que serve para atribuir cores ao texto do programa. As cores são determinadas conforme o tipo de *token* e conforme uma paleta pré-definida. Desse modo, um comentário pode ser colorido pois não é exatamente ignorado.

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

A.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` (8) define o 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 };
```

Código 8 – Estrutura parcial do compilador

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

`actInt` é a ação mais simples e apenas junta as informações para construir 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. `newExpr` e `op` auxiliam na criação de expressões matemáticas, e são usadas em vários outros métodos. `_comp` auxilia o acesso de um elemento de uma tupla, por exemplo: `(x, y, z)_1 = x`, e `p_1 = p_1` para uma constante `p`. `compute` é o método principal e invoca os outros métodos, além de descompactar algumas operações. O método `substitute` resolve funções, substituindo uma aplicação pelo corpo da função. O método `derivative` computa derivadas simbólicas. O método `calculate` calcula o valor numérico de uma expressão, quando possível, e é usado para cálculos feitos na *CPU*, pois as expressões são compiladas apenas para a *GPU*. O método `dependencies` detecta de quais grades um objeto depende.

Os métodos de `compile` a `declareFunction` geram o produto final. Juntos, geram os códigos-fonte na linguagem de *shader*(*GLSL*) do *OpenGL*. O método `header` auxilia a declaração dos parâmetros na linguagem *GLSL*. O método `compileFunction` apenas auxilia o retorno das funções no código *GLSL*. O método `declareFunction` auxilia na declaração das funções. O primeiro método `compile` gera o código-fonte para calcular o valor de uma expressão. 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. Ao finalizar, a interface está pronta para a interação com os objetos.

O compilador também verifica 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.