UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS VII – GOVERNADOR ANTÔNIO MARIZ
CENTRO DE CIÊNCIAS EXATAS E SOCIAIS APLICADAS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARINALDO VIANA DA SILVA JUNIOR

BLENDING FUNCTIONS OU *FORWARD DIFFERENCES* NA GERAÇÃO DE CURVAS PARAMÉTRICAS EM COMPUTAÇÃO GRÁFICA

PATOS/PB
2019

MARINALDO VIANA DA SILVA JUNIOR

BLENDING FUNCTIONS OU *FORWARD DIFFERENCES* NA GERAÇÃO DE CURVAS PARAMÉTRICAS EM COMPUTAÇÃO GRÁFICA

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual da Paraíba, em cumprimento à exigência para obtenção do título de Bacharel em Ciência da Computação.

Área de concentração: Computação Gráfica e Análise de Algoritmos

.

Orientador: Prof. Esp. Fábio Júnior Francisco da Silva

PATOS/PB
2019

S586b Silva Junior, Marinaldo Viana da.

Blending functions ou forward differences na geração de curvas paramétricas em computação gráfica [manuscrito] / Marinaldo Viana da Silva Junior. - 2019.

46 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências Exatas e Sociais Aplicadas , 2019.

"Orientação : Prof. Esp. Fábio Júnior Francisco da Silva , Coordenação do Curso de Computação - CCEA."

1. Computação gráfica. 2. Curvas paramétricas. 3. Analise de algoritmos. I. Título

21. ed. CDD 006.6

Elaborada por José E. da S. Eugênio - CRB - 15/591 BSC7/UEPB

Marinaldo Viana da Silva Júnior

BLENDING FUNCTIONS OU FORWARD DIFFERENCES NA GERAÇÃO DE CURVAS PARAMÉTRICAS EM COMPUTAÇÃO GRÁFICA

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciências da Computação da Universidade Estadual da Paraíba, em cumprimento à exigência para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 26/11/2019

BANCA EXAMINADORA

Fábio Junior Francisco da Silva

Prof. Esp. Fábio Júnior F. da Silva
(Orientador)

Jannayna Domingues Barros Filgueira

Prof. Dra. Jannayna Domingues Barros Filgueira
(Examinadora)

Alexandre Faustino Leite

Prof. Esp. Alexandre Faustino Leite
(Examinador)

Dedico este trabalho a minha família, em especial aos meus pais Marinaldo e Maria Aparecida, aos meus avós Rita, José Genésio (in memorian), Iracema e Benedito (in memorian), e a minha irmã Maysa. Dedico também a minha namorada Edvania e ao meu amigo Geovane (in memorian).

AGRADECIMENTOS

Agradeço primeiramente a Deus, por me dar saúde e força para superar todas as dificuldades.

Ao meu orientador Prof. Esp. Fábio Júnior Francisco da Silva, que teve papel fundamental para a realização deste trabalho, através da sua dedicação e contribuição ao longo das orientações.

Aos meus pais, Marinaldo e Maria Aparecida, e a minha irmã Maysa, que sempre estiveram comigo me apoiando nos momentos difíceis e felizes, e me ajudando a enfrentar as dificuldades ao longo desses anos.

A Edvania Barbosa, namorada querida, que muitas vezes abriu mão dos seus compromissos para ficar ao meu lado. Te amo!

Aos meus amigos que me ajudaram em alguns momentos do desenvolvimento deste trabalho, ajudando de acordo com o alcance de cada um.

Aos professores e colegas do curso, agradeço o apoio e obrigado por tudo. Serão sempre lembrados com carinho.

“Possuo um entendimento realista das minhas forças e fraquezas. A mente é a minha arma. [...] e uma mente necessita de livros da mesma forma que uma espada necessita de uma pedra de amolar se quisermos que se mantenha afiada.”

(Tyrion Lannister)

RESUMO

A Computação Gráfica (CG) é a área da computação que lida com técnicas e algoritmos para criação de imagens através do computador. Inserida na mesma estão as curvas paramétricas, que são usadas para se trabalhar desde formas simples, como círculos e elipses, até formas complexas, como modelos de carros e outros. Essas curvas podem ser feitas por métodos matemáticos, dentre eles: *blending functions* (funções de suavização) e *forward differences* (diferenças adiante). Desenhar curvas em CG tem alto custo computacional para o hardware. Assim, viu-se a necessidade de analisar o custo computacional de cada um. Para tal, criou-se dois algoritmos, um para cada método citado, com base na função de Bézier. Depois foi realizada a análise sobre os algoritmos. O *software* utilizado para a escrita dos algoritmos foi o *Processing*. Para a análise foram utilizadas as técnicas de análise formal de algoritmos, através da contagem de instruções para determinar a complexidade do pior caso de cada algoritmo. Os resultados indicam que *forward differences* tem o mesmo custo computacional que *blending functions* para gerar curvas paramétricas em CG, porém é mais complexo de entender e implementar.

Palavras-chave: Computação Gráfica. Análise de Algoritmos. Curvas Paramétricas. *Blending functions*. *Forward differences.*

ABSTRACT

Computer Graphics (CG) is the area of computing that deals with techniques and algorithms for computer imaging. Inserted in it are parametric curves, which are used to work from simple shapes, like circles and ellipses, even complex shapes like car models and others. These curves can be made by mathematical methods, among them: blending functions and forward differences. Drawing curves in CG has a high computational cost for the hardware. Thus, there was the need to analyze the computational cost of each. For this, two algorithms were created, one for each method mentioned, based on function the Bezier. Then the analysis of the algorithms was performed. The software used for writing the algorithms was Processing. For the analysis were used the techniques of formal analysis of algorithms, by counting instructions to determine the worst-case complexity of each algorithm. The results indicate that forward differences has the same computational cost as blending functions to generate parametric curves in CG, but it is more complex to understand and implement.

Keywords: Computer Graphics. Algorithm Analysis. Parametric Curves. Blending functions. Forward differences.

LISTA DE ILUSTRAÇÕES

Figura 1 – Áreas da computação gráfica 14
Figura 2 – Curva de Hermite 17
Figura 3 – Curva de Bézier 19
Figura 4 – Curva *B-Spline* 21
Figura 5 – Níveis de continuidade em curvas paramétricas 23
Figura 6 – Superfície paramétrica 24
Figura 7 – Dominação assintótica de $g(n)$ sobre $f(n)$ 28
Figura 8 – Notação O 29
Figura 9 – Notação Ω 30
Figura 10 – Notação θ 30
Figura 11 – Curva de Bézier utilizando *blending functions* 36
Figura 12 – Curva de Bézier utilizando *forward differences* 37
Figura 13 – representação matricial da superfície de Bézier 38

LISTA DE ABREVIATURAS E SIGLAS

B0	Ponto inicial de Bézier.
B1	Ponto de controle de Bézier.
B2	Ponto de controle de Bézier.
B3	Ponto final de Bézier.
CAD	*Computer-Aided Design*.
CG	Computação Gráfica.
ISO	*International Organization for Standardization*.
NURBS	*Non-Uniform Rational Basis Spline*.
P1	Ponto inicial de Hermite.
P2	Ponto final de Hermite.
T1	Tangente do ponto inicial de Hermite.
T2	Tangente do ponto final de Hermite.
s	Segundos
min	Minutos
Sécs.	Séculos
max	Máximo

LISTA DE SÍMBOLOS

$\leq$	Menor ou igual que
<	Menor que
$\leftarrow$	Recebe
$\boldsymbol{O}$	Grande O
$\boldsymbol{\Omega}$	Ômega
$\boldsymbol{\theta}$	Teta
=	Igual
+	Adição
-	Subtração
()	Parênteses
[]	Colchetes

SUMÁRIO

1 INTRODUÇÃO 12
2 REFERÊNCIAL TEÓRICO 14
2.1 Curvas paramétricas 15
2.1.1 Curvas de Hermite 16
2.1.2 Curvas de Bézier 18
2.1.3 Curvas B-Spline 20
2.1.4 Superfícies paramétricas 23
2.2 Forward difference 24
2.3 Técnicas e análise de algoritmos 27
2.3.1 Notação O 29
2.3.2 Notação Ω 29
2.3.3 Notação θ 30
2.3.4 Análise Agregada 30
3 MATERIAIS E MÉTODOS 33
4 RESULTADOS E DISCUSSÕES 34
5 CONSIDERAÇÕES FINAIS 39
REFERÊNCIAS 40
APÊNDICE A – ALGORITMO DE BÉZIER COM *BLENDING FUNCTIONS* 42
APÊNDICE B – ALGORITMO DE BÉZIER COM *FORWARD DIFFERENCES* 44
ANEXO A – TRANSFORMAÇÃO DA FUNÇÃO EM MATRIZ 46

1 INTRODUÇÃO

A Computação Gráfica (CG) é a área da computação que lida com técnicas e algoritmos para a criação de imagens através do computador. Basicamente tudo que é visto pelo usuário no computador é gerado por CG (MANSSOUR; COHEN, 2006).

Dentro da CG está a modelagem geométrica, que consiste em modelar um objeto computacionalmente da mesma maneira que é visto e dentro dessa modelagem aborda-se a geração de curvas paramétricas, que são usadas para se trabalhar desde formas simples, como círculos ou elipses, até as mais complexas, como modelos de carros e outros. As curvas são feitas a partir de funções quadradas (de forma não tão suave) ou funções cúbicas (com mais suavidade) e podem ser feitas em 2D (duas dimensões) ou 3D (três dimensões) (AZEVEDO; CONCI, 2003, BARBARINI, 2007). Este trabalho se baseia em curvas em 2D.

A geração de curvas em CG usa modelos matemáticos criados, exclusivamente, para tal finalidade. Por se tratar de curvas interativas, ou seja, o usuário de sistemas como CAD (*Computer-Aided Design*), que são *softwares* para criação de desenhos feitos no computador, interagem diretamente com as curvas e estas devem se ajustar às necessidades do mesmo, então necessita-se de muitos cálculos em curto espaço de tempo. Isso explica porque os sistemas gráficos consomem mais recursos computacionais que os demais.

Entre as abordagens matemáticas que podem ser utilizadas para a criação de curvas paramétricas existem: *blending functions* e *forward differences*. Ambas fazem uso dos recursos computacionais no desenho das curvas. Elas são diferentes apenas no método de cálculo. A literatura é limitada quanto a essa questão e não apresenta resposta definitiva, apenas suposição sobre qual técnica é a mais adequada (custo computacional menor) para que as curvas sejam desenhadas. Dessa forma, através da análise formal de algoritmos, pode-se avaliar o custo computacional de cada um destes métodos.

A forma de analisar um algoritmo concentra-se no custo computacional para executá-lo (custo = memória + tempo). A determinação desse custo pode ser feita de duas formas: por análise empírica, que depende fortemente do programa e da máquina que foi usada para criar o algoritmo; e a análise formal matemática, que é independente da implementação (TOSCANI; VELOSO, 2012).

Existem três principais medidas de custo computacional em um algoritmo: Pior Caso, Caso Médio e Melhor caso. No pior caso, leva-se em consideração o número máximo de instruções que o algoritmo executará. No melhor caso, ao contrário do pior caso, considera-se

o número mínimo de instruções que o algoritmo executará. E o caso médio é uma constante entre o melhor caso e o pior caso (DE CASTRO BARBOSA; TOSCANI; RIBEIRO, 2001).

É relevante analisar as duas abordagens mencionadas, pois descobrir o custo computacional de cada uma pode trazer benefícios aos programas de computadores que trabalham com desenhos de curvas. Esta pesquisa, portanto, através da análise formal de algoritmos, visa descobrir a seguinte questão: qual é o custo computacional dos métodos *blending functions* e *forward differences* para geração de curvas paramétricas em CG?

Para o desenvolvimento da pesquisa formulou-se como objetivo geral:

- Analisar formalmente o custo computacional de algoritmos para a geração de curvas paramétricas em Computação Gráfica utilizando os métodos *blending functions* e *forward differences*.

Para atingir o objetivo da pesquisa definiu-se os seguintes objetivos específicos:

- Realizar revisão bibliográfica para formar embasamento sobre o tema;
- Implementar no *Processing* o algoritmo de Bézier para geração das curvas utilizando *blending functions*;
- Implementar no *Processing* o algoritmo de Bézier para geração das curvas utilizando f*orward differences*;
- Realizar análise formal (matemática) de cada algoritmo para determinar a função de custo computacional;

Dessa forma, considera-se como hipótese que *forward differences* seria menos custoso computacionalmente que *blending functions* para a geração de curvas paramétricas, pois esse método trabalha com valores constantes que são obtidos através de derivadas de um polinômio genérico de grau 3, tornando o processo mais rápido.

O resultado mostrou que ambas apresentam o mesmo custo computacional em termos de análise de algoritmos, sendo *forward differences* mais difícil de implementar do que *blending functions* tendo como base a fase de implementação de cada um. Como esses métodos de geração de curvas paramétricas podem ser utilizados para as curvas de Hermite e *B-Spline* e como as curvas, no geral são a base para a geração de superfícies paramétricas, então pode-se afirmar que a conclusão desta pesquisa pode ser estendida às superfícies também.

O desenho de curvas paramétricas com *forward diffenrences* tem custo computacional igual a *blending functions*, mas *blending functions* é mais fácil de ser implementado.

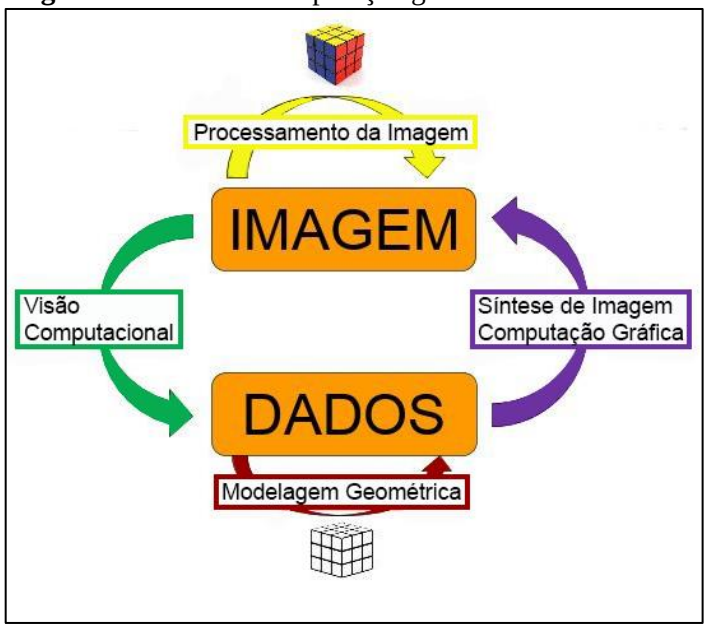
2 REFERÊNCIAL TEÓRICO

A Computação Gráfica (CG) é a área da Ciência da Computação que está ligada ao estudo e desenvolvimento de técnicas e algoritmos para a geração de imagens através do computador. A CG está presente em praticamente todas as áreas do conhecimento, indo da engenharia que utiliza as tradicionais ferramentas CAD, até a medicina, que trabalha com técnicas de visualização para auxiliar o diagnóstico por imagens (MANSSOUR; COHEN, 2006).

A ISO (*International Organization for Standardization*) define a CG como conjunto de ferramentas e técnicas que permitem converter dados para ou de um dispositivo gráfico através do computador. Ela é ferramenta não convencional que permite ao artista ultrapassar as técnicas tradicionais de desenho ou modelagem (AZEVEDO; CONCI, 2003). A CG tem a capacidade de levar a arte para um novo patamar, uma vez que se pode criar desenhos e se desprender de papel e lápis para usar a tecnologia para esta finalidade.

A Figura 1 mostra como a CG está organizada em áreas. O processamento de imagens tem como entrada uma imagem que sofre alguma transformação por processamento e gera a saída de outra imagem. Na síntese de imagens, tem-se a transformação de um conjunto de dados em uma imagem renderizada. A visão computacional tem a capacidade de identificar características em imagens. Já a modelagem geométrica lida somente com dados.

Figura 1 – Áreas da computação gráfica.



Fonte: Elaborada pelo autor, 2019.

O termo Computação Gráfica, está relacionado a qualquer técnica envolvida na criação ou manipulação de imagens no computador, incluindo imagens animadas (tradução feita pelo autor) (ECK, 2018). Com isso, vincula-se a CG com tudo visto pelo usuário por meio de recursos computacionais, como interfaces gráficas em *softwares*, jogos, vídeos, ambientes de realidade virtual, diagnósticos por imagens auxiliados por computador, projetos de engenharia civil, entre outras aplicações.

Em CG, modelos são usados para representar no computador, entidades e fenômenos do mundo físico. É dito que existem várias categorias ou métodos de construção de modelos bidimensionais e tridimensionais, onde cada um tem suas vantagens e desvantagens, adaptando-se melhor para determinadas aplicações. Pode-se modelar quando se consegue descrever um objeto ou cena, de forma que se possa desenhá-lo (MANSSOUR; COHEN, 2006).

Lopes (2008) disse que a modelagem geométrica baseada nas formas geométricas tem grande força de representação, pois com formas simples, onde cor e textura são atribuídas, pode-se representar objetos mais ou menos complexos com grandes detalhes. Curvas paramétricas estão inseridas dentro da modelagem geométrica e são de grande auxílio na modelagem de objetos.

2.1 Curvas paramétricas

Segundo Barbarini (2007) uma curva paramétrica é o mapeamento contínuo de um espaço unidimensional para um espaço n-dimensional. Elas são muito úteis em processos de modelagem na CG. Curvas de segunda ordem, que são curvas feitas a partir de funções quadradas, apresentam-se de forma restrita, devido à continuidade (curvas em sequência). Já as de terceira ordem (curvas feitas a partir de funções cúbicas), no entanto, são as curvas mais populares, pois apresentam mais suavidade.

"As curvas e superfícies paramétricas apresentam-se como pilar importante em várias áreas da Computação Gráfica e, em particular, no domínio da modelagem geométrica" (DOMINGOS, 2009, p. 9). Essa área consiste em modelar um objeto computacionalmente da mesma forma que é visto pelo observador. Com a ajuda de ferramentas CAD e utilizando curvas paramétricas pode-se criar modelos mais facilmente manipuláveis e de forma mais suave.

> "Na modelagem geométrica em Computação Gráfica, as curvas são a base, tanto da geração de formas simples, como círculos e elipses, quanto na criação de projetos complexos como automóveis, navios ou aeronaves (onde são referidas como formas livres)" (AZEVEDO; CONCI, 2003, p. 75).

É possível desenhar curvas de maneira diferente, dependendo da aplicação elas podem ser feitas através de uma sucessão de linhas, o que é suficiente para determinadas aplicações, ou podem ser feitas através de um conjunto de pontos interligados, dando suavidade a curva. Na forma paramétrica, as curvas podem ser criadas em função de um parâmetro t que é utilizado para plotar cada ponto da função. No plano cartesiano, cada coordenada é calculada em função do parâmetro t, conforme ilustra a função (1):

$$P(t) = (\, x\,(t)\,,\ y\,(t)) \tag{1}$$

Utilizando o conceito de derivadas de funções, pode-se obter o vetor tangente desse ponto pela derivada em relação ao parâmetro, mostrado na função (2), pois através dele pode-se movimentar a curva:

$$P'(t) = (\, x'\,(t)\,, y'(t)) \tag{2}$$

O comprimento da curva define-se pela variação do parâmetro t, tal que $0 \leq t \leq 1$ (AZEVEDO; CONCI, 2003). Para curvas em terceira dimensão (3D), precisa-se acrescentar a coordenada $z\ (t)$.

2.1.1 Curvas de Hermite

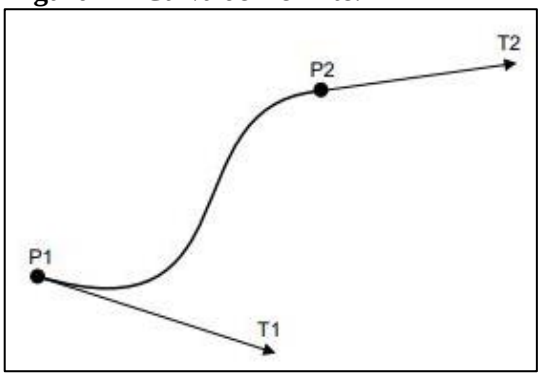
Autores afirmaram que a formulação da curva de Charles Hermite, matemático francês nascido em 1822, surgiu da necessidade de solucionar problemas de definição de uma curva, através de seus pontos extremos e suas derivadas nesses pontos. Em geral, esses pontos são usados para interpolar, por exemplo, animações ou controle de câmera. Entender como a matemática funciona nessa técnica de geração de curvas é o primeiro passo para a compreensão de curvas desse tipo e demais polinômios de ajuste de curvas (LUCCI, 2009, BARBARINI, 2007).

As curvas de Hermite são curvas paramétricas que possuem sua estrutura na forma de dois pontos de controle e duas derivadas que estão especificadas em cada um dos pontos (vetores tangentes). Dessa maneira, a função fica perfeitamente definida por quatro pontos: o valor da função nos pontos de controle e o valor da derivada da função nestes mesmos pontos (ROSA, 1991). Pode-se chamar esses pontos de P_1 (ponto inicial) e P_2 (ponto final) e suas

derivadas (vetores) T_1 (tangente que está ligada a P_1) e T_2 (que está ligada a P_2). É importante destacar que para gerar uma curva (Hermite, Bézier e *B-Spline*) precisa-se de, no mínimo, quatro pontos.

Assim, os dois pontos e os dois vetores tangentes participam da composição da curva de Hermite, como mostra a Figura 2. Os quatro pontos têm valores conhecidos e informados no momento de geração da curva, logo são condições iniciais para a curva de Hermite.

Figura 2 – Curva de Hermite.



Fonte: Azevedo e Conci (2003, p. 81).

Utilizando operações aritméticas que ficam mais facilmente representadas em operações matriciais, pode-se gerar com perfeição cada ponto da curva, ou seja, as coordenadas x, y e z (caso se trabalhe com curvas em terceira dimensão) a partir dos pontos P1 e P2 e suas tangentes T1 e T2. Assim, modifica-se a função (1) de acordo com o Anexo A (que também vale para as demais curvas), para se obter a função (3):

$$P(t) = T\ H\ G_h \tag{3}$$

Onde:

T é a matriz dos parâmetros em função de t:

$$[t^3 \quad t^2 \quad t \quad 1]$$

H é a matriz transposta dos coeficientes de Hermite:

$$\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Vale ressaltar que as matrizes transpostas de Hermite, Bézier e *B-Spline*, têm valores distintos. G_h é o vetor de geometria de Hermite:

$$\begin{bmatrix} P_1 \\ P_2 \\ T_1 \\ T_2 \end{bmatrix}$$

Assim, a função (3) pode ser escrita em sua forma matricial representada na função (4).

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ T_1 \\ T_2 \end{bmatrix} \quad (4)$$

A partir da operação de multiplicação sobre a função (4), pode-se obter a função (5) que representa cada ponto da curva de Hermite. É importante lembrar que esta função precisa ser aplicada, individualmente, as componentes x, y ou z quando se tratar de geração de curva no espaço tridimensional.

$$P(t) = \big(P_1(2t^3 - 3t^2 + 1) + P_2(-2t^3 + 3t^2) + T_1(t^3 - 2t^2 + t) + T_2(t^3 - t^2)\big) \quad (5)$$

(AZEVEDO; CONCI, 2003, p. 86).

2.1.2 Curvas de Bézier

As curvas de Bézier foram criadas aproximadamente na década de 60, por Pierre Bézier e Paul de Faget de Casteljau, mas de formas diferentes. Bézier criou sua definição usando funções de mesclagem (*blending functions*), enquanto Casteljau desenvolveu o algoritmo de interpolação recursiva. Bézier trabalhou na Renault e Casteljau, na Citröen, onde se desenvolvia os primeiros sistemas de *design* de automóveis assistido por computador (LUCCI, 2009, RAMAKRISHNAN, 2002).

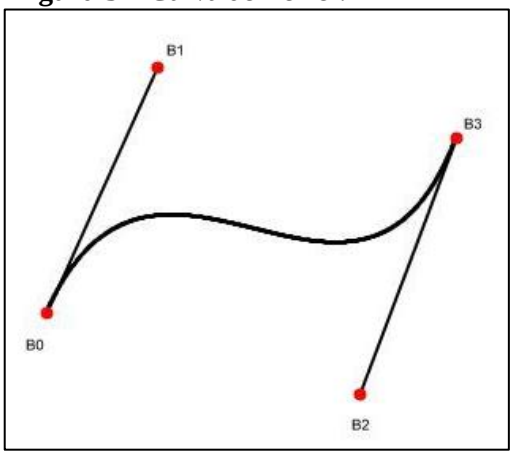
> "[...] A contribuição de Bézier para a Computação Gráfica abriu caminho para *softwares* CAD (Desenho assistido por computador) como o Maya, o Blender

> e o 3D Max. Seus desenvolvimentos servem como porta de entrada para aprender sobre computação gráfica moderna, que gerou um objeto matemático relativamente novo conhecido como *Spline*, ou uma curva suave especificada em termos de alguns pontos" (traduzido pelo autor) (BERTKA, 2008, p. 1).

Segundo Almeida (2015), o processo de construção da curva de Bézier é bastante intuitivo, levando o usuário a realizar uma construção sem grandes conhecimentos.

Na curva de Bézier, diferente da curva de Hermite, tem-se quatro pontos (B0, B1, B2, B3), onde todos os pontos são pontos de controle. Tem-se um ponto inicial B0, um final B3 e dois tangentes a esses, um tangente ao ponto inicial (B1), formando segmento de reta e o outro tangente ao ponto final (B2), formando outro segmento de reta e a curva é desenhada entre os pontos inicial e final e controlada por esses segmentos. Um exemplo da curva de Bézier, está ilustrado na Figura 3.

Figura 3 – Curva de Bézier.



Fonte: Adaptado de: Azevedo e Conci (2003).

A função (1), que é utilizada na geração de curvas em CG, pode ser representada para as curvas de Bézier da seguinte forma, mostrada na função (6):

$$P(t) = T\ M_B\ G_B \tag{6}$$

Onde:

T é a matriz dos parâmetros em função de t:

$$[t^3 \quad t^2 \quad t \quad 1];$$

M_B é a matriz transposta dos coeficientes de Bézier:

$$\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix};$$

G_B é o vetor de geometria de Bézier:

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}.$$

A função (6) pode ser representada matricialmente tal como está descrito na função (7).

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \quad (7)$$

Similar a curva de Hermite, tem-se que, ao realizar a multiplicação das matrizes na função (7), chega-se à função (8) que pode ser usada para representar cada ponto da curva nas componentes x, y ou z das coordenadas:

$$P(t) = (1-t)^3 B_0 + 3t(1-t)^2 B_1 + 3t^2(1-t)B_2 + t^3 B_3 \quad (8)$$

(AZEVEDO; CONCI, 2003).

2.1.3 Curvas B-Spline

Scalco (2005) afirma que as curvas *B-Spline* foram propostas em 1946 por Jacob Schoenberg (1903-1990) como solução de problemas de aproximação de curvas.

Segundo Barbarini (2007), o termo *Spline* foi definido na matemática como curvas especiais formadas por seguimentos de polinômios. As *B-Splines* vêm do modelo de *Spline* Natural, que faz alusão ao termo da língua inglesa para denominar régua flexível usada em desenhos para gerar curvas livres suaves com curvaturas contínuas. Porém, de forma diferente

da *Spline* Natural, nas *B-Splines* os pontos de controle não são interpolados pela curva. Dessa forma, apenas uma pequena parte da curva altera-se.

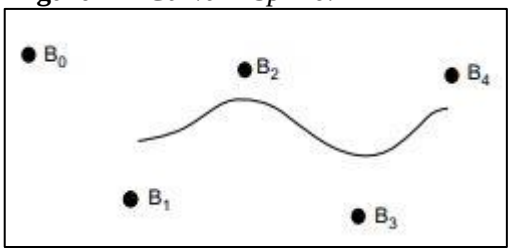
"*B-Splines* são como as curvas de Bézier porque elas usam um polígono de controle para definir a curva e são úteis devido ao controle local dos pontos de controle da forma resultante" (traduzido pelo autor) (BERTKA, 2008, p. 9).

> "A família de curvas e superfícies *B-Spline* pode ser considerada como uma analogia entre a *Spline* clássica para com métodos da curva de Bézier. O objetivo desta comunhão foi encontrar uma aplicabilidade do método de Bézier que usa um polígono de controle para produzir uma forma tão bem controlada de um objeto e das formas de *Spline* de grau suave" (JUNIOR et al., 2012, p.1).

Existem várias maneiras de expressar uma curva *B-Spline*, mas depende das condições de contorno nas extremidades e nos nós de fronteira entre cada segmento.

A curva *B-Spline* foi baseada na *Spline* Natural, onde a curva passa pelos pontos de controle e uma alteração em qualquer um desses pontos provoca alteração na curva. A *B-Spline* tem pontos de controle, mas a curva não toca nesses pontos, como mostra a Figura 4.

Figura 4 – Curva *B-Spline*.



Fonte: Azevedo e Conci (2003, p. 92).

Da mesma maneira que Hermite e Bézier, a função (1) pode ser escrita para *B-Spline*:

$$P(t) = T\ M_{Bs}\ G_{Bs} \tag{9}$$

Onde:

T é a matriz dos parâmetros t:

$$[t^3 \quad t^2 \quad t \quad 1];$$

M_{Bs} é a matriz de coeficientes da *B-Spline*:

$$\frac{1}{6} * \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix};$$

G_{Bs} é o vetor de geometria da *B-Spline*:

$$\begin{bmatrix} B_{i-1} \\ B_i \\ B_{i+1} \\ B_{i+2} \end{bmatrix}$$

sendo i igual ao grau do polinômio usado.
Assim, obtém-se a representação matricial da função (9) da seguinte forma:

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} 1/6 \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} B_{i-1} \\ B_i \\ B_{i+1} \\ B_{i+2} \end{bmatrix} \quad (10)$$

De modo geral, a função *B-Spline* para cada componente, x, y ou z, das coordenadas de cada ponto na curva pode ser representada na função (11) da seguinte maneira:

$$P(t) = \frac{1}{6}((-t^3 + 3t^2 - 3t + 1)B_{i-1} + (3t^3 - 6t^2 + 4)B_i + (-3t^3 + 3t^2 + 3t + 1)B_{i+1} + (t^3)B_{i+2}) \quad (11)$$

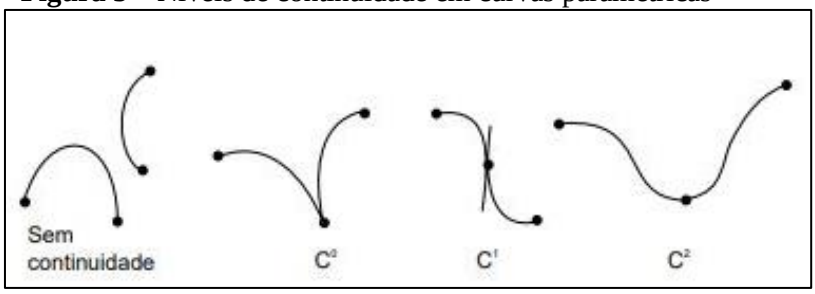
(AZEVEDO; CONCI, 2003, p. 97).

Dessa forma, foram definidas as funções de Hermite, Bézier e *B-Spline* usando *blending functions*.

Em geral, uma forma complexa pode ser modelada por várias curvas conectando seus pontos extremos. Essa conexão é feita quando se deseja controlar a continuidade nos pontos de junção. Continuidade de ordem 0 significa que as duas curvas se encontram, mas não há suavidade neste encontro; continuidade de primeira ordem exige que as curvas tenham tangentes comuns no ponto de junção (ponto onde uma curva termina e outra começa) e

continuidade de segunda ordem exige que as curvaturas sejam as mesmas. A notação usada para representar essas continuidades é C^0, C^1, C^2, etc. A continuidade C^0 assegura que uma curva ou a união de curvas não terá descontinuidade. A continuidade C^1 indica que a inclinação ou sua derivada primeira da curva é constante em todos os pontos. A continuidade C^2 implica em continuidade na derivada segunda da curva e assim por diante. As curvas de Hermite e Bézier apresentam continuidade do tipo C^1, enquanto *B-Spline* tem continuidade C^2. A Figura 5 abaixo mostra exemplos dos níveis de continuidade em curvas.

Figura 5 – Níveis de continuidade em curvas paramétricas



Fonte: Azevedo e Conci (2003, p. 90).

2.1.4 Superfícies paramétricas

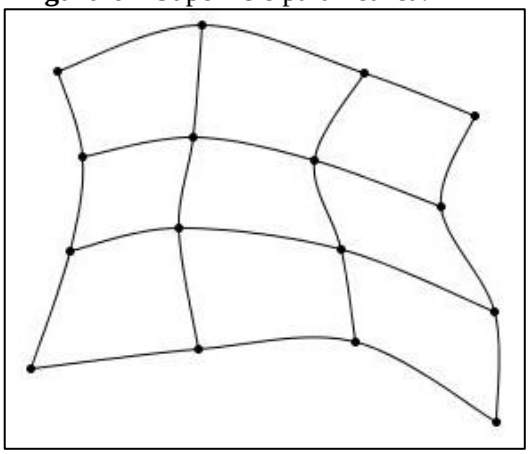
Chiarella (2004) afirma que a incorporação de superfícies paramétricas em sistemas computacionais de modelagem de formas, levou à criação de uma nova ferramenta gráfica que não apenas cobre a lacuna deixada pela geometria clássica, mas também permite a geração rápida de formas complexas com uma quantidade mínima de dados (traduzido pelo autor).

É dito que superfícies têm papel muito importante na computação gráfica. De maneira geral, as superfícies são uma generalização das curvas. Uma superfície pode ser gerada por famílias de conjuntos de pontos: ter representação analítica; explícita ou implícita; paramétrica ou não-paramétrica. Podendo-se ainda interpolar, ajustar ou aproximar superfícies a partir de pontos (AZEVEDO; CONCI, 2003).

Superfícies paramétricas foram usadas quando precisou-se trabalhar com superfícies suaves na modelagem de objetos de forma livre (*Free Form Objects*). Para tal, uma representação muito utilizada são os *patches* (retalhos) paramétricos bicúbicos, que calculam as coordenadas de todos os pontos que formam uma superfície curva através da definição de 16 pontos de controle e da utilização de três funções, uma para x, uma para y e uma para z. Cada função possui duas variáveis (ou parâmetros) e termos para todo domínio dos parâmetros até o

seu cubo (daí as expressões bi e cúbico) (MANSSOUR; COHEN, 2006). A Figura 6 mostra um exemplo de superfície paramétrica.

Figura 6 – Superfície paramétrica.



Fonte: Elaborada pelo autor, 2019.

O *patch* é uma superfície curva na qual cada um dos pontos que a formam deve ser processado. Quatro destes pontos que determinam a forma do *patch* pertencem aos seus cantos. Através de parâmetros passados para as funções, é possível determinar a quantidade de valores intermediários calculados. Além disso, sempre que um ponto de controle é alterado, os pontos que formam a superfície devem ser gerados novamente (MANSSOUR; COHEN, 2006).

"Superfícies Bézier, *B-spline* e Hermite fazem parte do conjunto de *patches* ou superfícies bicúbicas, sendo, em conjunto com NURBS (*Non-Uniform Rational Basis Spline*), os tipos de superfícies paramétricas mais utilizadas em modelagem CAD e animação para cinema" (SANTOS, 2012, p. 5).

O termo NURBS é a abreviatura de *Non-Uniform Rational Basis Spline*, ou seja, é uma *B-Spline* racional (originária da razão de polinômios). *Non-Uniform* significa que a extensão de um controle de vértice pode variar com relação ao parâmetro t (o que é muito bom na modelagem de superfícies irregulares). *Rational* significa que a equação usada para representar a curva ou superfície é expressa pela razão de dois polinômios, o que fornece um modelo melhor de superfícies (AZEVEDO; CONCI, 2003).

2.2 Forward difference

É dito que é possível, normalmente, distinguir dois tipos de funções em aplicações. Nas funções $f(x)$ onde a variável x pode tomar um valor qualquer num dado intervalo, tem-se uma variável contínua. Estas funções pertencem ao domínio do Cálculo Infinitesimal. Já as funções

para as quais a variável independente x apenas pode tomar determinados valores $x_0, x_1, \dots x_n$, a variável é discreta. Para estas funções aplica-se o Cálculo das Diferenças Finitas, o qual pode ser aplicado a ambas as categorias de funções (CARVALHO, 1996).

Carvalho (1996) afirma que a teoria do Cálculo das Diferenças Finitas foi desenvolvida para o caso de funções $\{ f(x)\}$ definidas para valores discretos de x, $\{x_0, x_1, x_2, \dots, x_n\}$, quando estes valores são equidistantes, isto é, quando se tem:

$$x_{i+1} - x_i = h \tag{12}$$

onde h é o passo constante adicionado à função e é independente de i e de x.

Neste caso, a teoria pode ser aplicada também a funções de variável contínua, e a 1ª diferença ou derivada de $f(x)$, representada por $\Delta f(x)$, é definida por:

$$\Delta f(x) = f(x + h) - f(x) \tag{13}$$

Essa função $\Delta f(x)$ foi chamada de *first forward difference* (primeira diferença adiante).

As fórmulas do Cálculo das Diferenças Finitas são extremamente simplificadas quando o incremento $h = 1$. Assim, pode-se definir diferenças de ordem superior:

$$\Delta^2 f(x) = \Delta\big(\Delta f(x)\big) = \Delta f(x + h) - \Delta f(x) \tag{14a}$$
$$\Delta^2 f(t) = \Delta\big(\Delta f(x)\big) = \Delta f(x + 1) - \Delta f(x) \tag{14b}$$

ou

$$\Delta^2 f(x) = f(x + 2) - 2f(x + 1) + f(x) \tag{14c}$$

como segunda diferença. E:

$$\Delta^n f(x) = \Delta[\Delta^{n-1} f(x)] \tag{15}$$

como enésima diferença (CARVALHO, 1996).

A diferenciação direta é um esquema eficiente para avaliar um polinômio em muitas etapas uniformes. Para avaliar um polinômio cúbico de t em várias etapas uniformes deriva-se uma função de diferença que expressa a diferença entre $p(t)$, o valor do polinômio avaliado em t e $p(t+h)$, o polinômio avaliado em $t+h$, onde h é o tamanho do passo uniforme. Para um polinômio cúbico e um tamanho de passo constante, essa diferença é um polinômio quadrático de t (traduzido pelo autor) (BARTLEY, 1997).

Dessa forma pode-se avaliar um polinômio quadrático mais eficientemente por passos do que um cúbico. No entanto, se esse esquema funcionou para uma equação cúbica, ele também deve funcionar para uma equação quadrática. Assim, aplicando a técnica acima a um quadrático, produz-se uma função de diferença que é uma função linear de t. Finalmente, o que funcionou para o quadrático deve funcionar para o linear, portanto, aplicar a técnica a uma função linear produz uma função de diferença que é uma constante (traduzido pelo autor) (BARTLEY, 1997).

Ao invés de avaliar um polinômio cúbico em cada etapa, pode-se calculá-lo no valor inicial de t e, em seguida, calcular cada etapa subsequente fazendo apenas algumas adições. Essa é a técnica de diferenciação avançada (*forward differencing technique*) (traduzido pelo autor) (BARTLEY, 1997).

A critério de exemplo, utilizou-se um polinômio de 3º grau:

$$f(t) = at^3 + bt^2 + ct + d \tag{16}$$

Dessa forma, tem-se a *forward difference*:

$$\Delta f(t) = a(t+h)^3 + b(t+h)^2 + c(t+h) + d - (at^3 + bt^2 + ct + d)$$

Aplicando os devidos cálculos tem-se:

$$\Delta f(t) = 3at^2h + t(3ah^2 + 2bh) + ah^3 + bh^2 + ch \tag{17}$$

A função (17) ainda não foi satisfatória, pois trata-se de um polinômio de 2º grau em relação a t, e ainda precisa-se calcular $\Delta f(x)$ e uma adição (BARTLEY, 1997). Aplicou-se *forward difference* a $\Delta f(x)$, na função (17), com base na função (14a), o que fez com que seu cálculo fosse simplificado.

Assim, obteve-se:

$$\Delta^2 f(x) = 6ah^2t + 6ah^3 + 2bh^2 \tag{18}$$

Para não depender mais de t, esse processo repetiu-se mais uma vez sobre a função (18):

$$\Delta^3 f(x) = \Delta\left(\Delta^2 f(t)\right) = \Delta^2 f(t+h) - h^2 f(t) = 6ah^3 \tag{19}$$

Como a terceira *forward difference* é uma constante, não foi necessário levar este processo adiante, pois a função não é diferençável além da terceira derivada.

As curvas supramencionadas podem ser geradas a partir, tanto das *blending functions* como de *forward differences*.

2.3 Técnicas e análise de algoritmos

É dito que um algoritmo nada mais é do que uma sequência de passos bem definida e não ambígua para executar uma determinada tarefa (TOSCANI; VELOSO, 2012). Mas o simples fato de um algoritmo resolver uma tarefa, não significa que ele seja o mais eficiente na prática. É preciso analisar seu custo computacional para decidir sobre sua eficiência.

A complexidade de um algoritmo está relacionada ao esforço computacional para executá-lo (custo = memória + tempo). Esse esforço mede o tempo de execução e a quantidade de memória usada pelo algoritmo. A determinação dessa complexidade pode ser feita de duas formas: por análise empírica, que depende fortemente do programa e da máquina que foi usada para criar o algoritmo; e a análise formal matemática, que é independente da implementação (TOSCANI; VELOSO, 2012). Ainda segundo os autores, o esforço computacional de um determinado algoritmo não pode ser relatado apenas por um número, pois a quantidade de trabalho depende da entrada. Dessa forma, o desempenho do algoritmo (custo computacional) concede o esforço computacional de execução baseado na entrada dada e na saída obtida. O custo desse algoritmo pondera o esforço para que ele seja executado em cima de um conjunto de entradas.

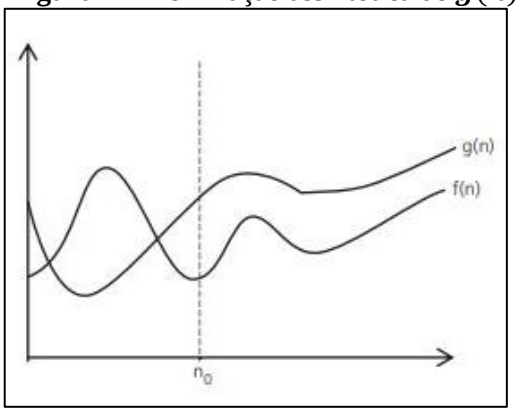
Existem três principais medidas de custo computacional em um algoritmo: Pior Caso, Caso Médio e Melhor caso. O custo no pior caso fixa um tamanho de entrada, e a análise é feita em relação ao número máximo de operações fundamentais necessárias para que o problema seja resolvido, de forma a se tornar o limite máximo que não será excedido. Assim, tem-se que

é uma garantia mínima de qualidade do algoritmo (DE CASTRO BARBOSA; TOSCANI; RIBEIRO, 2001). No custo de caso médio, as entradas foram frequentemente supostas como sendo iguais. É dito que essa suposição pode ser violada na prática, assim pode-se usar da análise probabilística para melhores resultados (CORMEN et al, 2002). Já o custo de melhor caso ocorre ao contrário do pior caso, ou seja, quando o número mínimo de operações fundamentais necessárias satisfaz o problema e ele seja resolvido.

Quando o crescimento desse custo se torna demasiadamente grande para determinadas entradas, adota-se a análise assintótica, que é usada para descrever os limites de um algoritmo. Em funções do tipo $n^2,\ 3n^2, 1000n^2, n^2/500$, com valores enormes para n, nota-se que crescem na mesma velocidade, assim, para a matemática, essas funções são equivalentes. Essa análise assintótica foca apenas na variável, deixando de lado as constantes (TOSCANI; VELOSO, 2012). “Em geral, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas” (CORMEN et al, 2002).

Com relação a análise assintótica, tem-se o limite assintótico superior, que é uma função que cresce mais rápido que outra e que permanece acima dela a partir de determinado ponto, como mostra a Figura 7, para n com um valor muito grande, $g(n)$ domina $f(n)$, ou seja, cresce mais rápido que $f(n)$ a partir de certo ponto.

Figura 7 – Dominação assintótica de $\boldsymbol{g(n)}$ sobre $\boldsymbol{f(n)}$.



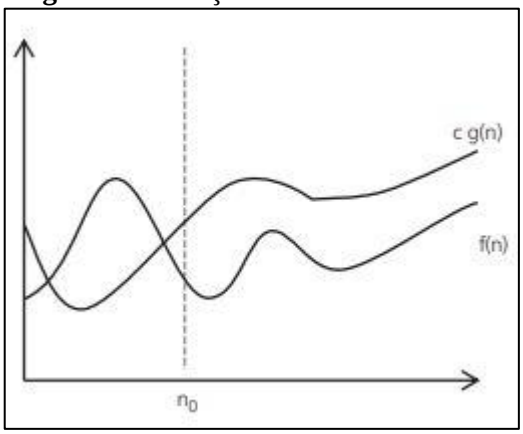
Fonte: Toscani e Veloso (2012, p. 25).

Esse limite assintótico nos leva para três tipos de notações: a notação O (“big O” ou “O grande”), a notaçãoω (Ômega) e a notação θ (Teta) (CORMEN et al, 2002).

2.3.1 Notação O

Dadas funções assintoticamente não-negativas f e g, essa notação diz que há um limite superior para funções f, onde essa função é menor ou igual a esse limite (g). Desta forma, diz-se que $f = O(g)$ ou $f \in O(g)$ se $f(n) \leq c.g(n)$, onde existe um número c e um número n_0, ambos positivos, e $n \geq n_0$. Assim, $g(n)$ é o limite assintótico superior de $f(n)$. A Figura 8 mostra graficamente a notação O (CORMEN et al, 2002).

Figura 8 – Notação O.

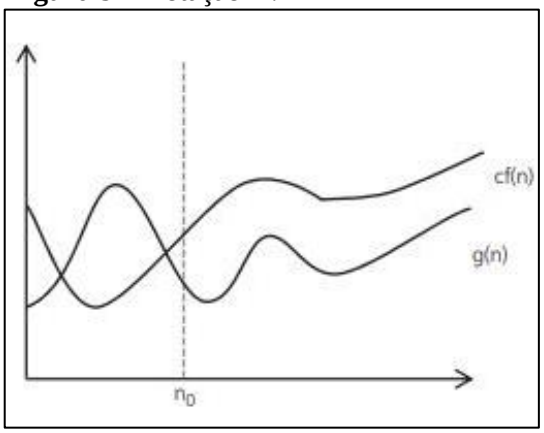


Fonte: Toscani e Veloso (2012, p. 26).

2.3.2 Notação Ω

As funções assintoticamente não-negativas f e g, a notação Ômega diz que há um limite inferior para funções f, onde essa função é maior ou igual a esse limite (g). Desta maneira, é dito que $f = \Omega(g)$ ou $f \in \Omega(g)$ se $g(n) \leq c.f(n)$, onde existe um número c e um número n_0, ambos positivos, e $n \geq n_0$. Assim, $g(n)$ é o limite assintótico inferior de $f(n)$. A Figura 9 mostra graficamente a notação Ω (TOSCANI; VELOSO, 2012).

Figura 9 – Notação Ω.

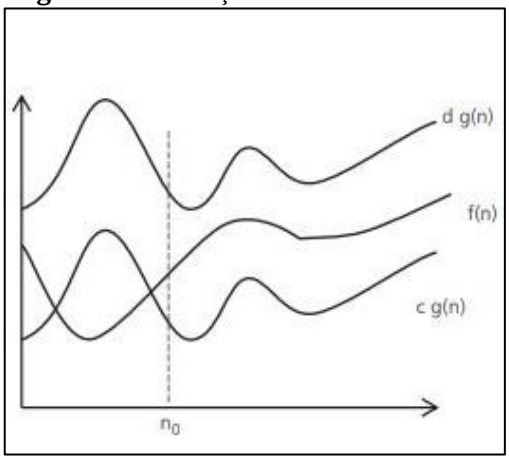


Fonte: Toscani e Veloso (2012, p. 28).

2.3.3 Notação θ

Diz-se que as funções f e g são de ordem Teta e assim, que $f = \theta(g)$ ou $f \in \theta(g)$ se $f = O(g)$ e $f = \Omega(g)$, onde existe um número c, um número d e um número n_0, todos positivos, com $n \geq n_0$, de forma que $c.g(n) \leq f(n) \leq d.g(n)$. Por fim, diz-se que f encontra-se entre seus limites inferior e superior. A Figura 10 mostra graficamente a notação θ (CORMEN et al, 2002).

Figura 10 – Notação θ.



Fonte: Toscani e Veloso (2012, p. 30).

2.3.4 Análise Agregada

Este tipo de análise leva em consideração a análise do pior caso $O(n)$, o qual um algoritmo que contém uma sequência de n operações demora o seu tempo máximo $T(n)$ para

ser executado. Segundo Cormen et al (2002), todas as linhas do algoritmo são contadas a partir de alguns princípios:

- Para comandos de atribuição, seja de entrada ou de saída, tem-se $O(1)$, ou seja, tem valor 1;
- Para sequência de comandos o tempo de execução é o maior da sequência. Por exemplo: um trecho de um programa com dois tempos de execução $O(n)$ e $O(n^2)$. Seu tempo de execução é $O(\max(n, n^2))$, que é $O(n^2)$;
- Para um comando de decisão simples o tempo de execução é dado pela soma do tempo para avaliar a condição com o tempo dos comandos executados dentro da condição. Em comandos *if / else* tem-se duas sequências que podem ser executadas. Se a sequência 1 tem $O(1)$ e a sequência 2 tem $O(n^3)$, o pior caso para esse *if / else* será $O(n^3)$;
- Para um comando de repetição o tempo de execução é dado pela soma do tempo de execução do laço com o tempo de avaliar a condição de parada do laço multiplicado pelo número de iterações do laço. Por exemplo, um laço *for* (para) tem sua atribuição que recebe zero (conta-se 1), sua condição de parada é até que se chegue em algum valor (n) e seu incremento é de um em um. Dessa forma, soma-se a atribuição (que vale 1) com seu incremento (que também vale 1) e multiplica-se pelo total de vezes que o laço será repetido (n). Assim pode-se chegar a uma função $2 * n + 1$. Então tem-se que $O(n)$. Um outro exemplo de laço é o *for* aninhado, ou seja, um *for* dentro de outro *for*. Nesse caso, cada *for* executa n vezes, então multiplica-se suas funções encontradas separadamente de cada um. O resultado pode ser, por exemplo, $(4 * n + 5) * (2 * n + 3) = 8 * n^2 + 22 * n + 15$. Assim, tem-se $O(n^2)$. O mesmo acontece com um laço do tipo *while* (enquanto).

Cormen et al (2002), afirma ainda que se pode classificar os algoritmos baseando-se nas suas ordens de complexidade. As principais classes de comportamento assintótico de algoritmos são:

- $O(1)$ - As instruções do algoritmo para esse caso são executadas um número fixo de vezes (constante), não dependendo do valor de n;
- $O(logn)$ - Muito bom, ocorre tipicamente em algoritmos que dividem um problema em problemas menores;

- $O(n)$ - É ideal para um algoritmo que processa n elementos de entrada ou saída. Esse tipo de ordem de complexidade é linear;
- $O(nlogn)$ - Ocorre tipicamente em algoritmos que dividem um problema em problemas menores, resolvendo cada uma dessas partes independentemente e depois juntando as soluções;
- $O(n^2)$ - Este tipo de algoritmo é útil para resolver problemas relativamente pequenos que são processados aos pares em forma de um *loop* (anel) dentro do outro;
- $O(n^k)$ - Polinomial. É aceitável para um k pequeno;
- $O(k^n), O(n!), O(n^n)$ – Estes são tipos exponenciais e deve-se evitá-los, pois não são úteis sob o ponto de vista prático. São usados quando se usa força bruta para resolver um problema.

A Tabela 1 a seguir ilustra a diferença entre as classes de comportamento assintótico, onde pode-se notar a razão de crescimento de várias funções com complexidade para tamanhos diferentes de n, em que cada função expressa o tempo de execução em microssegundos, onde um algoritmo linear executa em um segundo um milhão de operações.

Tabela 1 – Diferença entre algumas classes de comportamento assintótico

Função de custo	**n = 10**	**n = 20**	**n = 30**	**n = 40**	**n = 50**	**n = 60**
$\boldsymbol{n}$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
$\boldsymbol{n^2}$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0025 s	0,0036 s
$\boldsymbol{n^3}$	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,316 s
$\boldsymbol{n^5}$	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
$\boldsymbol{2^n}$	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 sécs.
$\boldsymbol{3^n}$	0,59 s	58 min	6,5 anos	3855 sécs.	10^8 sécs.	10^{13} sécs.

Fonte: Adaptado de: Cormen et al (2002).

Ao analisar um algoritmo e determinar sua complexidade, pode-se dizer se é um algoritmo eficiente ou não, tendo em vista o que ele busca resolver. Todavia, algoritmos com complexidade exponencial (com exceção de expoentes com valores baixos) devem ser evitados, pois podem não resolver problemas com a eficiência necessária, dependendo do caso (CORMEN et al, 2002).

3 MATERIAIS E MÉTODOS

Este trabalho formulou-se através de pesquisa exploratória, para que se pudesse ter familiaridade e entendimento do tema. Para isso, foi realizada revisão bibliográfica em diversas bases de dados para encontrar publicações com o intuito facilitar e formar o embasamento teórico do mesmo. Dentre várias bases de dados, destacaram-se três principais: o Google Acadêmico, o Portal de Periódicos da CAPES e o portal IEEE.

O recurso de *software* que foi utilizado para criar os algoritmos de desenho das curvas foi o *Processing*, criado por Bem Fry e Casey Reans em 2001, que foi criado inicialmente como rascunho de *software* e para ensinar fundamentos de programação dentro de um contexto visual, hoje é uma ferramenta de desenvolvimento utilizado por profissionais. Ele é de código aberto e é executado em Mac, Windows e Linux (BEN; CASEY, 2001).

Fazendo uso do *software* em questão, foi criado o algoritmo com base na Função (8), para desenhar na tela a curva paramétrica de Bézier. Vale ressaltar que esse algoritmo foi criado, inicialmente, usando a técnica *blending functions.*

Aplicando o mesmo *software* (*Processing*), mas dessa vez empregando a técnica *forward differences*, foi criado um algoritmo para desenhar a mesma curva paramétrica (Bézier).

Os algoritmos criados foram submetidos às técnicas formais de análise de algoritmos para encontrar a função do custo computacional (custo = memória + tempo). Essa análise matemática determinou o custo dos algoritmos de geração de curvas em Computação Gráfica, *blending functions* e *forward differences*.

4 RESULTADOS E DISCUSSÕES

A implementação dos algoritmos tomou como base uma matriz de valores que foram usados para gerar os pontos da curva, então esses valores foram aplicados nas respectivas formas de gerar uma curva. Após ser implementado o algoritmo da curva de Bézier utilizando *blending functions* e utilizando *forward differences* para gerar curvas paramétricas, realizou-se a análise agregada da complexidade de algoritmos, que leva em consideração o pior caso na execução do mesmo.

O **Algoritmo 1** mostra o pseudocódigo do trecho principal da curva de Bézier utilizando *blending functions*. Para não haver poluição visual, apenas o trecho principal dos dois algoritmos foi inserido nos resultados. O algoritmo completo implementado no *Processing* está no Apêndice A.

Algoritmo 1 – Pseudocódigo da curva de Bézier com *blending functions*

```
CURVA-BLENDINGFUNCTIONS(M[L][8])
1  para i de 0 até i < M.totalLinhas
2     para t de 0 até t ≤ 1 faça
3        n ← 1 / número de iterações
4        x ← (((1-t)^3) * M[i][j]) + ((3 * ((1-t)^2)) * t *
         M[i][j+2]) + (3 * (1-t) * (t^2) * M[i][j+4]) + ((t^3)
         * M[i][j+6])
5        y ← (((1-t)^3) * M[i][j+1]) + (3 * ((1-t)^2)) * t *
         M[i][j+3]) + (3 * (1-t) * (t^2) * M[i][j+5]) + ((t^3)
         * M[i][j+7])
6        t ← t + n
7        desenhaPonto(x, y)
8     fim-para
9     i ← i + 1
10    fim-para
```

Fonte: Elaborado pelo autor, 2019.

O `para` (comando de repetição dentro do algoritmo) que executa todo o trecho do código, das linhas 1-10, inicia com i igual a zero e continua até que tenha percorrido todas as linhas da matriz. Dessa forma, tem-se a função (20):

$$f(n) = 2 * n + 1 \tag{20}$$

onde n é o total de vezes que o `para` será executado, multiplicado pela soma das vezes que ocorrerá a comparação (`i < M.totalLinhas`) e da atribuição de incremento de `i` (1 + 1 = 2 vezes), e somado a atribuição de (`i ← 0`), uma vez.

Já o `para` que executa as linhas 2-8 inicia com `t` igual a zero e vai até 1, de forma que esse intervalo é incrementado de acordo com o número de iterações que se deseja para formar a curva (caso deseje-se mil iterações, o `n` corresponderá a 0.001).

Logo tem-se a função (21):

$$f(n) = 2 * (n + 1) + 1 \tag{21}$$

onde $n + 1$ corresponde ao total de vezes que o `para` será executado, multiplicado pela soma de vezes que ocorrerá a comparação (`t <= 1`) e da atribuição de incremento de `t` (1 + 1 = 2 vezes), e somado a atribuição (`t ← 0`) uma vez. Dessa forma, pode-se reescrever a função (21) da seguinte forma:

$$\mathrm{f(n)} = 2 * \mathrm{n} + 3 \tag{22}$$

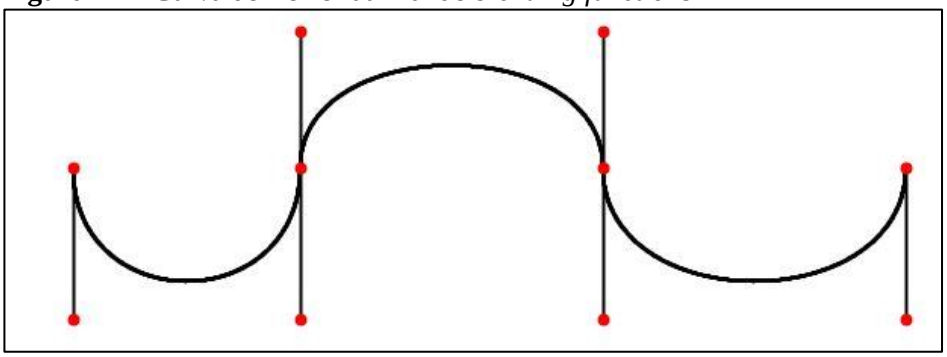
É notório que o pseudocódigo do **Algoritmo 1** tem um laço aninhado (*for* aninhado), assim, para se obter a função (23), multiplica-se a função (20) pela função (22):

$$f(n) = 4 * n^2 + 8 * n + 3 \tag{23}$$

Dessa forma, pode-se afirmar que o custo computacional de pior caso do pseudocódigo do **Algoritmo 1** é $O(n^2)$, ou seja, um algoritmo exponencial.

A Figura 11 mostra a ilustração da curva de Bézier fazendo uso de *blending functions* e produzida no Processing, a qual contém dez pontos e três seguimentos de curva, onde os quatro pontos centrais são o início e fim de cada seguimento e os mais externos são pontos de controle. Os dados de entrada para gerar esta curva estão no Apêndice A.

Figura 11 – Curva de Bézier utilizando *blending functions*



Fonte: Elaborado pelo autor, 2019.

O **Algoritmo 2**, ilustra o pseudocódigo do trecho principal da curva de Bézier fazendo uso de *forward differrences*. O algoritmo completo implementado no *Processing* está no Apêndice B.

Algoritmo 2 – Pseudocódigo da curva com *forward differences*

```
CURVA-FORWARDDIFFERENCES(M[L][8])
1  para i de 0 até i < M.totalLinhas faça
2     n ← número de iterações
3     x ← M[i][j];
4     y ← M[i][j];
5     para k de 0 até n faça
6           k ← k + 1;
7           x ← x + D1x;
8           D1x ← D1x + D2x;
9           D2x ← D2x + D3x;
10          y ← y + D1y;
11          D1y ← D1y + D2y;
12          D2y ← D2y + D3y;
13          desenhaPonto(x, y);
14    fim-para
15    i ← i + 1
16 fim-para
```

Fonte: Elaborado pelo autor, 2019.

O `para` que executa as linhas 1-16 inicia com `i` igual a zero, indo até varrer todas as linhas da matriz. Assim obteve-se a função (24):

$$f(n) = 2 * n + 1 \qquad (24)$$

onde n é o número de vezes que o laço será executado, multiplicado pela soma da comparação (`i < M.totalLinhas`) com o incremento de `i` (1 + 1 = 2 vezes) e somado ainda com sua atribuição (`i ← 0`).

O `para` que é executado nas linhas 5-14 faz a comparação de uma variável `k`, que inicia em zero, até que se chegue no total de iterações desejadas (`n`). As demais variáveis (`D1x, D1y, D2x, D2y, D3x, D3y`) são derivadas que, no decorrer da execução, terão seus valores alterados. Dessa maneira, obteve-se a função (25):

$$f(n) = 2 * n + 1 \tag{25}$$

onde n é o número de vezes que o laço será executado, multiplicado pela soma da comparação (`k < n`) com o incremento de `k` (1 + 1 = 2 vezes) e somado ainda com sua atribuição (`k ← 0`).

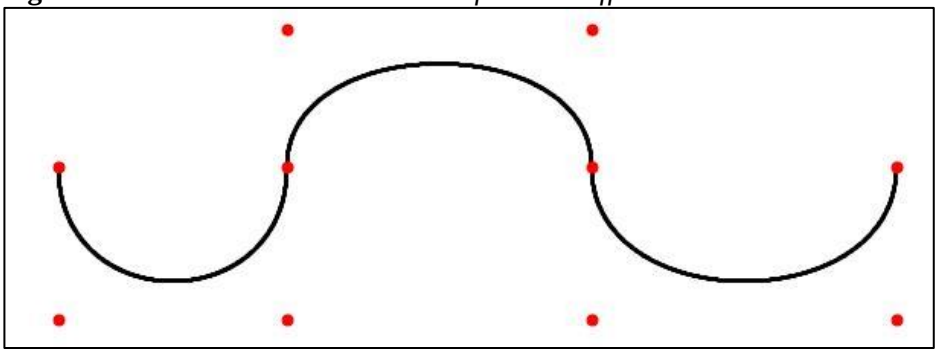
Para obter a função (26), multiplica-se a função (24) com a função (25):

$$f(n) = 4 * n^2 + 4 * n + 1 \tag{26}$$

Assim, pode-se afirmar que o custo computacional de pior caso do pseudocódigo do **Algoritmo 2** é $O(n^2)$, ou seja, um algoritmo exponencial.

A Figura 12 abaixo mostra a curva de Bézier utilizando *forward differences*, o qual apresenta dez pontos e três seguimentos de curva, onde os pontos que tocam a curva são os pontos iniciais e finais das mesmas, e os pontos que não tocam a curva são pontos de controle. Os dados de entrada para gerar esta curva estão no Apêndice B.

Figura 12 – Curva de Bézier utilizando *forward differences*



Fonte: Elaborado pelo autor, 2019.

Analisando o pior caso dos dois algoritmos a partir das funções (23) e (26), tem-se $O(\max(n^2, n^2))$, que resulta no custo computacional de pior caso $O(n^2)$.

É possível afirmar que os métodos são iguais quanto a seu custo algorítmico, diferenciando-se apenas sobre o modo de implementação, o qual *blending functions* se torna mais fácil de compreender e implementar. Dessa maneira, o algoritmo de Bézier com *forward differences* é igual o algoritmo de Bézier com *blending functions* para gerar curvas paramétricas, pois ambos crescem de forma exponencial (n^2).

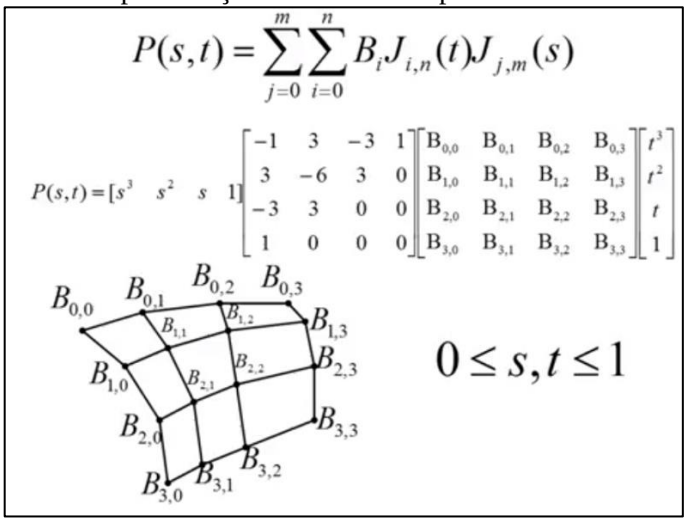
Essa comprovação estende-se também para as curvas de Hermite e *B-Spline*, pois as funções (3) de Hermite e (9) de *B-Spline* são similares a função (6) de Bézier.

Pode-se ainda estender essa comprovação e afirmar que essa igualdade permanece na geração de superfícies paramétricas, pois elas são geradas com base em curvas interligadas entre si e como modo exemplo, a função (27) ilustra o modelo geral da superfície de Bézier:

$$P(s,t) = [s^3\ s^2\ s\ 1]\ M_B\ G_B\ M_B^T \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \tag{27}$$

A Figura 13 representa a equação matricial completa para a superfície de Bézier onde cada par ordenado $P(s,t)$ é formado pelo somatório de linhas e colunas.

Figura 13 – representação matricial da superfície de Bézier



Fonte: adaptado de: Azevedo e Conci (2003).

Portanto a complexidade de pior caso das curvas também é igual para superfícies paramétricas.

5 CONSIDERAÇÕES FINAIS

O desenvolvimento do presente estudo possibilitou uma análise formal de algoritmos sobre os métodos *blending functions* e *forward differences*, que foram aplicados na curva de Bézier para determinar qual é o mais eficiente para geração de curvas paramétricas em CG.

Houve dificuldade para encontrar trabalhos sobre este tema, pois notou-se certa escassez de material nas bases de dados que foram pesquisadas, remetendo a pesquisas similares, fragmentando o tema por partes, de forma que a união delas pudesse formar tudo que foi necessário para que fosse alcançado o objetivo deste estudo.

Neste estudo foi implementado o algoritmo de Bézier utilizando *blending functions* e utilizando *forward differences* para determinar o custo computacional de cada um. Para isso, foi realizada uma análise formal de algoritmos sobre o trecho principal de cada algoritmo, trecho esse que é onde realmente é gerada a curva e onde está sua complexidade para o pior caso de cada um dos mesmos.

Os resultados apontaram que o algoritmo de Bézier que faz uso de *forward differences* tem custo de pior caso igual ao algoritmo de Bézier que faz uso de *blending functions*, já que os dois algoritmos crescem exponencialmente (n^2), havendo diferença apenas na forma de compreensão dos dois, pois *blending functions* é mais simples de se compreender do que *forward differences* com base na etapa de criação dos mesmos, assim sendo mais fácil para se implementar.

Fica como sugestão para trabalhos futuros realizar uma análise empírica sobre os algoritmos para determinar se um se sobressai sobre o outro em termos de eficiência, já que a análise matemática mostrou que são iguais quanto ao seu custo computacional. Mas vale ressaltar mais uma vez que a análise empírica depende da máquina que será utilizada e a eficiência pode variar se for feita em máquinas distintas.

REFERÊNCIAS

ALMEIDA, Evert Elvis Batista de. ***Curvas de Bézier***. 2015. 68f. Dissertação de Mestrado – Universidade Federal da Paraíba, João Pessoa, 2015.

AZEVEDO, Eduardo; CONCI, Aura. **Computação gráfica: teoria e prática**. 1ª edição. Campus: Rio de Janeiro, 2003.

BARBARINI, Luiz Henrique Maiorino. ***Síntese de cascos de embarcações através de métodos de otimização aplicados a curvas B-Spline***. 2007. 142f. Tese de Doutorado. Universidade de São Paulo, São Paulo, 2007.

BARTLEY, Curtis. **Forward difference calculation of Bezier curves**. C/C++ Users Journal, v. 15, n. 11, p. 19-26, 1997.

BEN, Fry; CASEY, Reas. **Processing**, 2001. Página inicial. Disponível em: https://processing.org//. Acesso em: 02 de jun. de 2019.

BERTKA, Benjamin T. **An introduction to Bézier curves, B-splines, and tensor product surfaces with history and applications**. University of California Santa Cruz, May 30th, 2008.

CARVALHO, Cesar Roberto Guimarães de. ***Aplicação dos polinômios ortogonais discretos à modelagem e simulação de processos de separação por estágios***. 1996. 97f. Dissertação de Mestrado. Universidade Estadual de Campinas, Campinas, 1996.

CHIARELLA, Mauro. **Superficies paramétricas y arquitectura: Conceptos, ideación y desarrollo**. In: CONGRESSO DA SOCIEDADE IBEROAMERICANA DE GRÁFICA DIGITAL. 2004. p. 393-395.

CORMEN, Thomas H. et al. **Algoritmos: teoria e prática**. Editora Campus, v. 2, p. 2, 2002.

DE CASTRO BARBOSA, Marco Antonio; TOSCANI, Laira Vieira; RIBEIRO, Leila. **ANAC–Uma ferramenta para análise automática da Complexidade de algoritmos**. Revista do CCEI, v. 5, n. 8, p. 57, 2001.

DOMINGOS, Vasco Alexandre dos Santos Dionísio. ***Tecnologia Java na representação de curvas e superfícies paramétricas***. 2009. 105f. Dissertação de Mestrado. FCT-UNL, Lisboa, 2009.

ECK, David. **Introduction to Computer Graphics**. 2008.

JUNIOR, Elizeu Martins de Oliveira et al. **Suavização de feições por B-Spline**. São Paulo. 2012.

LOPES, João Manuel Brisson. **Modelação geométrica**. Instituto Superior Técnico, p. 1-34, 2008.

LUCCI, Paulo Cesar de Alvarenga. ***Descrição matemática de geometrias curvas por interpolação transfinita***. 2009. 76f. Dissertação de Mestrado. Universidade Estadual de Campinas, Campinas, 2009.

MANSSOUR, Isabel Harb; COHEN, Marcelo. **Introdução à computação gráfica**. RITA, v. 13, n. 2, p. 43-68, 2006.

RAMAKRISHNAN, C. **An introduction to NURBS and OpenGL**. Lecture Notes. Duke University, 2002.

ROSA, Edison da. ***O uso de polinômios cúbicos de Hermite no planejamento de trajetórias de manipuladores***. 1991. 150f. Tese de Doutorado. Universidade Federal de Santa Catarina, Florianópolis, 1991.

SANTOS, Artur Lira dos. ***Elevando o nível gráfico de aplicativos interativos com o uso de Ray Tracing em tempo real de superfícies paramétricas***. 2012. Projeto de Pesquisa de Doutorado. Universidade Federal de Pernambuco, 2012.

SCALCO, Roberto. **Introdução a computação gráfica**. Mauá, 2005.

TOSCANI, Laira Vieira; VELOSO, Paulo A. S. **Complexidade de algoritmos**. 3ª edição. Porto Alegre: Bookman, 2012.

APÊNDICE A – ALGORITMO DE BÉZIER COM *BLENDING FUNCTIONS*

```
1  // DECLARAÇÃO DO TAMANHO DA TELA
2
3  void setup() {
4    size(800, 600);
5  }
6
7  // DECLARAÇÃO DA MATRIZ DOS PONTOS PARA A CURVA DE BÉZIER
8
9  int[][] matriz = {{50, 100, 50, 200, 200, 200, 200, 100},
10   {200, 100, 200, 10, 400, 10, 400, 100},
11   {400, 100, 400, 200, 600, 200, 600, 100}};
12
13 // MÉTODO QUE VAI DESENHAR A CURVA DE BÉZIER
14
15 void curvaBlendingFunction () {
16
17   stroke(#000000); // FORMATAÇÃO DE COR DAS LINHAS
18   strokeWeight(2); // FORMATAÇÃO DE ESPESSURA DA LINHA ENTRE
   OS PONTOS
19
20   line(matriz[0][0], matriz[0][1], matriz[0][2],
   matriz[0][3]); // DESENHO DA LINHA
21   line(matriz[0][6], matriz[0][7], matriz[0][4],
   matriz[0][5]); // DESENHO DA LINHA
22   line(matriz[2][0], matriz[2][1], matriz[2][2],
   matriz[2][3]); // DESENHO DA LINHA
23   line(matriz[2][6], matriz[2][7], matriz[2][4],
   matriz[2][5]); // DESENHO DA LINHA
24   line(matriz[1][0], matriz[1][1], matriz[1][2],
   matriz[1][3]); // DESENHO DA LINHA
25   line(matriz[1][6], matriz[1][7], matriz[1][4],
   matriz[1][5]); // DESENHO DA LINHA
26
27   // DESENHO DA CURVA DE BÉZIER COM BLENDING FUNCTIONS
28
29   for (int i = 0; i < matriz.length; i++) {
30
31     int j = 0;
32     for (float t = 0; t <=1; t = t + 0.001) {
33
34       float x = (pow((1 - t), 3) * matriz[i][j]) + (3 *
   pow((1 - t), 2) * t * matriz[i][j+2]) + (3 * (1-t) * pow(t,
   2) * matriz[i][j+4]) + (pow(t, 3) * matriz[i][j+6]);
35       float y = (pow((1 - t), 3) * matriz[i][j+1]) + (3 *
   pow((1 - t), 2) * t * matriz[i][j+3]) + (3 * (1-t) * pow(t,
   2) * matriz[i][j+5]) + (pow(t, 3) * matriz[i][j+7]);
36
37       stroke(#000000); // COR DA CURVA
```

```
38        strokeWeight(2); // ESPESSURA DA CURVA
39        point(int(x), (y)); // DESENHO DOS PONTOS QUE FORMARÃO
   A CURVA
40      }
41    }
42
43    stroke(#FF0000); // COR DOS PONTOS
44    strokeWeight(8); // ESPESSURA DOS PONTOS
45
46    // DESENHO DOS PONTOS
47
48    point(matriz[0][0], matriz[0][1]); // PONTO 1
49    point(matriz[0][2], matriz[0][3]); // PONTO 2
50    point(matriz[0][4], matriz[0][5]); // PONTO 3
51    point(matriz[0][6], matriz[0][7]); // PONTO 4
52    point(matriz[1][6], matriz[1][7]); // PONTO 5
53    point(matriz[2][2], matriz[2][3]); // PONTO 6
54    point(matriz[2][4], matriz[2][5]); // PONTO 7
55    point(matriz[2][6], matriz[2][7]); // PONTO 8
56    point(matriz[1][2], matriz[1][3]); // PONTO 9
57    point(matriz[1][4], matriz[1][5]); // PONTO 10
58  }
59
60  void draw() {
61    clear(); // LIMPAR A TELA
62    background(#FFFFFF); // COR DE FUNDO DA TELA
63    curvaBlendingFunction (); // DESENHAR A CURVA
64  }
```

APÊNDICE B – ALGORITMO DE BÉZIER COM *FORWARD DIFFERENCES*

```
1  void setup() {
2    size(800, 600);
3  }
4
5
6  int[][] M = {{50, 100, 50, 200, 200, 200, 200, 100},
7    {200, 100, 200, 10, 400, 10, 400, 100},
8    {400, 100, 400, 200, 600, 200, 600, 100}};
9
10 void curvaForwardDifference() {
11   for (int i = 0; i < M.length; i++) {
12
13     int j = 0;
14
15     float x = M[i][j];
16     float y = M[i][j+1];
17
18     float cx = 3 * (M[i][j+2] - M[i][j]);
19     float cy = 3 * (M[i][j+3] - M[1][j+1]);
20
21     float bx = (3 * (M[i][j+4] - M[i][j+2])) - cx;
22     float by = (3 * (M[i][j+5] - M[i][j+3])) - cy;
23
24     float ax = M[i][j+6] - M[i][j] - cx - bx;
25     float ay = M[i][j+7] - M[i][j+1] - cy - by;
26
27     int n = 1000;
28     float delta = 1/float(n);
29
30     float D1x = (ax*pow(delta, 3)) + (bx*pow(delta, 2)) +
   (cx*delta);
31     float D1y = (ay*pow(delta, 3)) + (by*pow(delta, 2)) +
   (cy*delta);
32
33     float D2x = (6 * ax*pow(delta, 3)) + (2 * bx*pow(delta,
   2));
34     float D2y = (6 * ay*pow(delta, 3)) + (2 * by*pow(delta,
   2));
35
36     float D3x = 6 * ax*pow(delta, 3);
37     float D3y = 6 * ay*pow(delta, 3);
38
39     // DESENHO DA CURVA DE BÉZIER COM FORWARD DIFFERENCES
40     for (int k =0; k < n; k++) {
41       x = x + D1x;
42       D1x = D1x + D2x;
43       D2x = D2x + D3x;
44       y = y + D1y;
```

```
45        D1y = D1y + D2y;
46        D2y = D2y + D3y;
47        point(int(x), int(y));
48      }
49    }
50
51    stroke(#FF0000);
52    fill(#FF0000);
53    strokeWeight(8);
54    point(M[0][0], M[0][1]); // PONTO 1
55    point(M[0][2], M[0][3]); // PONTO 2
56    point(M[0][4], M[0][5]); // PONTO 3
57    point(M[0][6], M[0][7]); // PONTO 4
58    point(M[1][6], M[1][7]); // PONTO 5
59    point(M[2][2], M[2][3]); // PONTO 6
60    point(M[2][4], M[2][5]); // PONTO 7
61    point(M[2][6], M[2][7]); // PONTO 8
62    point(M[1][2], M[1][3]); // PONTO 9
63    point(M[1][4], M[1][5]); // PONTO 10
64  }
65
66  void draw() {
67    clear();
68    background(#FFFFFF);
69    strokeWeight(2);
70    stroke(#000000);
71    curvaForwardDifference();
72  }
```

ANEXO A – TRANSFORMAÇÃO DA FUNÇÃO EM MATRIZ

Fórmula geral para a função (1) com base no modelo de Bézier:

$$x(t) = P_X = a_x t^3 + b_x t^2 + c_x t + d_x$$
$$y(t) = P_y = a_y t^3 + b_y t^2 + c_y t + d_y$$

Tomando o ponto em x como exemplo, sua forma matricial é:

$$x(t) = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}_x$$

lembrando que existe uma matriz para cada projeção (x, y, z).

Considerando a matriz linha do parâmetro t como T, e a matriz dos coeficientes a, b, c, d como C_x, tem-se:

$$x(t) = TC_x$$

Como C_x é desconhecido, adota-se uma matriz transposta de constantes multiplicada pelo vetor de geometria que são os pontos:

$$C_x = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

que pode ser reescrita como:

$$C_x = M_B G_B$$

Logo, pode-se reescrever $P(t)$ da seguinte forma:

$$P(t) = TM_B G_B$$