FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Safko

Tessellation of trimmed NURBS surfaces

Mathematical Institute of Charles Univesity

Supervisor of the master thesis: doc. RNDr. Zbyněk Šír, Ph.D.
Study programme: Computer Science
Study branch: IPGVPH

Prague 2022

I would like to thank my parents for supporting me during my studies. I would also like to thank Antonín Míšek for providing an endless supply of tessellation errors.

Title: Tessellation of trimmed NURBS surfaces

Author: Martin Safko

Institute: Mathematical Institute of Charles Univesity

Supervisor: doc. RNDr. Zbyněk Šír, Ph.D., Mathematical Institute of Charles Univesity

Abstract: Tessellation of trimmed NURBS surfaces is classical problem in CAD/CAM, with long history and huge amount of research developed so far. We present and describe a tessellation algorithm suitable for visualization purposes in either offline or online setting and present our results. We also provide pointers to literature and to tessellation algorithms for simulation. We discuss relevant definitions and procedures necessary to work with CAD data and try to familiarize it for people outside the industry.

Keywords: CAD/CAM CAGD NURBS tessellation

Contents

Intorduction **2**

1 Geometry of Curves **3**
1.1 Parametric Curves . . . 3
1.2 Bézier Curves . . . 4
1.3 B-spline Curves . . . 7
1.4 Rational Bézier Curves . . . 12
1.5 NURBS Curves . . . 13

2 Geometry of Surfaces **15**
2.1 Parametric Surfaces . . . 15
2.2 Bézier Surfaces . . . 16
2.3 B-spline Surfaces . . . 16
2.4 NURBS Surfaces . . . 17
2.5 Bézier Triangles . . . 18
2.6 Trimmed Surfaces . . . 19

3 Algorithms **20**
3.1 Bézier . . . 20
3.2 B-spline . . . 21
3.3 NURBS . . . 25

4 Tessellation **27**
4.1 Use Cases . . . 27
4.2 Approximation Error . . . 28
4.3 Tessellation Algorithm . . . 30
4.4 Results . . . 36

5 Implementation Details **38**
5.1 Algorithm Details . . . 38
5.2 Software Details . . . 39
5.3 Data formats . . . 41

Conclusion **44**

Bibliography **45**

A B-spline examples **47**

B More results **49**

Introduction

Tessellation is the process of covering a surface with simple geometric tiles without any gaps or overlap. Specifically, in computer graphics it refers to generating polygonal meshes to simplify further processing. Our work focuses on tessellating trimmed NURBS surfaces used predominantly for representing the shape of models in CAD/CAM.

Computer-Aided Design(CAD) and Computer-Aided Manufacturing(CAM) are long-established fields in both industry and academia, that lately also started to appear in consumer market with the introduction of relatively cheap 3D printers. Particularly in academia, the term Computer-Aided Geometric Design(CAGD) is being used as a study of accurate representation and efficient processing of geometric objects, such as curves, surfaces and volumes, on a computer. Traditionally, reviewing and judging new designs required a manufacturing of prototypes. Although scaled mockups and clay models still have its place in the production process, designers increasingly switch to using computer graphics visualizations, especially since the introduction of virtual reality devices. This introduces additional constraints and demands on tessellation algorithms.

In this thesis we design a procedure that outputs a watertight triangular mesh from a collection of trimmed NURBS surfaces, and describe the implementation of associated algorithms in VRUT. VRUT is a software application for producing real-time visualizations, driving simulations and virtual trainings, to name a few, developed at Škoda Auto a.s. by the Virtual Techniques department, which focuses on presenting CAD data in virtual reality(VR) and creating photorealistic renderings for assessing future vehicle designs. Our goal is to improve the original implementation, based in part on the works of [1], [2] and [3], in regards to performance and visual quality.

Overview

In this text we start with relevant mathematical definitions of curves and surfaces necessary for understanding NURBS type in chapters 1 and 2. Next in chapter 3, we present standard algorithms to evaluate and manipulate the defined objects, and is used as a reference for subsequent chapters. Following in chapter 4, we describe our tessellation algorithm and show the results. Finally in chapter 5, we finish with discussion on implementation details necessary to develop software for visualizing CAD data.

Chapter 1

Geometry of Curves

In this chapter we present basic definitions and concepts used throughout the thesis. Readers familiar with the concept of approximation curves are free to skip this chapter. This chapter is intended to provide a gentle introduction. More thorough treatment of this subject can be found in [4], [5] or [6].

Notation

We distinguish vector quantities from scalars using bold letters, e.g. d is a scalar while $\boldsymbol{d}$ is a vector. Furthermore, we use bold capital letters for points. Functions can be defined using both, uppercase and lowercase letters.

1.1 Parametric Curves

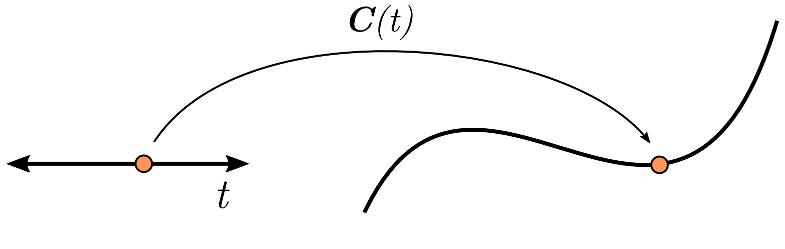
Parametric curve is defined as a smooth function from some real interval $I \subseteq \mathbb{R}$ to $\mathbb{R}^n$.

$$\boldsymbol{C}(t) : t \in I \longmapsto \boldsymbol{x} \in \mathbb{R}^n$$

We usually think of t as a parameter of time. Therefore $\boldsymbol{C}(t)$ is the position, $\boldsymbol{C}'(t) := \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{C}(t)$ velocity and $\boldsymbol{C}''(t) := \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{C}'(t)$ acceleration at time t.

We can define tangent and normal unit vectors at point t as

$$\boldsymbol{T}(t) = \frac{\boldsymbol{C}'(t)}{\|\boldsymbol{C}'(t)\|} \qquad \boldsymbol{N}(t) = \frac{\boldsymbol{T}'(t)}{\|\boldsymbol{T}'(t)\|}$$



Curvature can be computed as (see [7] for more details)

$$k(t) = \frac{\|\boldsymbol{C}'(t) \times \boldsymbol{C}''(t)\|}{\|\boldsymbol{C}'(t)\|^3}$$

Let's focus our attention on how we might go about defining a parametric curve suitable for computer-aided design. Ideally, we would like the curve to have several important properties:

1. Ease of computation:
    - It should be algorithmically easy to compute values on the curve.
    - It should be easy to evaluate derivatives.
    - All the computations mentioned above should be numerically stable in the context of floating point arithmetic.
2. Intuitive manipulation:
    - We would like the curve to be geometrically intuitive to manipulate and do small, local adjustments.
3. Geometrical expressiveness
    - Considering a class of all reasonable curve shapes, one should be able to represent them exactly or at least with suitable accuracy.
4. Smoothness
    - Generally, smooth curves are required with the ability to create sharp corners.

Looking at the first property, the obvious choice would be to use a polynomial

$$\boldsymbol{C}(t) = \sum_{i=0}^{n} t^i\, \mathbf{P}_i = \mathbf{P}_0 + t\mathbf{P}_1 + t^2\mathbf{P}_2 + \cdots + t^n\mathbf{P}_n \tag{1.1}$$

where $t \in \mathbb{R}$ is the parameter of the curve and $\mathbf{P}_i$ are the control points. This type of curve is easy and fast to evaluate and compute all derivatives. On the other hand, it is difficult to see how the control points correspond to the shape of the curve. If we consider t to be in range $[0, 1]$, then $\mathbf{P}_0$ is the starting point and $\mathbf{P}_1$ gives the tangent at the start. However, the geometric meaning of the control points gets more and more obscure with each point.

1.2 Bézier Curves

Given a sequence of $n+1$ control points $\mathbf{P}_0, \ldots, \mathbf{P}_n$, a Bézier[1] curve is defined as

$$\boldsymbol{C}(t) = \sum_{i=0}^{n} B_{i,n}(t)\, \mathbf{P}_i = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i}\, \mathbf{P}_i \qquad 0 \le t \le 1 \tag{1.2}$$

The $B_{i,n}(t)$ are called Bernstein[2] polynomials. The first sum emphasizes that the curve is a linear combination of control points weighted by functions $B_{i,n}(t)$, therefore, the curve inherits all of the basis function properties.

[1]Named after the French engineer Pierre Bézier, who developed and used functionally equivalent curves at Renault in the 1960s.

[2]Soviet mathematician Sergei Bernstein utilized the polynomials in a proof of Weierstrass approximation theorem in 1912.

Basis function properties

1. **Non-negativity:** $B_{i,n}(t) \geq 0 \quad \forall i, n \quad 0 \leq t \leq 1$
2. **Partition of unity:** $\sum_{i=0}^{n} B_{i,n}(t) = 1 \quad 0 \leq t \leq 1$
3. **Endpoints:** $B_{0,n}(0) = B_{n,n}(1) = 1$
4. **Recursion:** $B_{i,n}(t) = (1-t)B_{i,n-1}(t) + tB_{i-1,n-1}(t)$
5. **Derivative:** $\frac{\mathrm{d}}{\mathrm{d}t} B_{i,n}(t) = B'_{i,n}(t) = n\Big(B_{i-1,n-1}(t) - B_{i,n-1}(t)\Big)$

Using these facts we can show important properties of Bézier curves.

Curve properties

- $\boldsymbol{C}(0) = \mathbf{P}_0$ and $\boldsymbol{C}(1) = \mathbf{P}_n$ (follows from the endpoints property)
- Bézier curve with $n+1$ control points has degree n
- The entire curve lies in the convex hull of its control points. Since basis functions are non-negative and sum to one, every point on the curve $\boldsymbol{C}(t)$ is a convex combination of its control points.
- Curve shape is invariant under affine transformations(translations, rotations, scaling). The significance is that instead of transforming the whole curve, we can just apply the transformation to control points and construct the curve in the transformed location. This follows from the partition of unity property. Assume $T(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ is an affine transformation. Then

$$\begin{aligned} T\Big(\boldsymbol{C}(t)\Big) &= A\Big(\sum_{i=0}^{n} B_{i,n}\,\mathbf{P}_i\Big) + \boldsymbol{b} = \sum_{i=0}^{n} B_{i,n}\,A\mathbf{P}_i + \sum_{i=0}^{n} B_{i,n}\boldsymbol{b} \\ &= \sum_{i=0}^{n} B_{i,n}\,(A\mathbf{P}_i + \boldsymbol{b}) = \sum_{i=0}^{n} B_{i,n}\,T(\mathbf{P}_i) \end{aligned}$$

- Control points $\mathbf{P}_0, \ldots, \mathbf{P}_n$ form a piecewise linear approximation to the curve.
- Derivative of $\boldsymbol{C}(t)$ is a Bézier curve of degree $n-1$. Using the derivative property of Bernstein polynomials

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{C}(t) = \boldsymbol{C}'(t) &= \sum_{i=0}^{n} \frac{\mathrm{d}}{\mathrm{d}t} B_{i,n}(t)\,\mathbf{P}_i \\ &= \sum_{i=0}^{n-1} B_{i,n-1}(t)\,n\Big(\mathbf{P}_{i+1} - \mathbf{P}_i\Big) = \sum_{i=0}^{n-1} B_{i,n-1}(t)\,\mathbf{Q}_i \end{aligned}$$

  where $\mathbf{Q}_i := n\Big(\mathbf{P}_{i+1} - \mathbf{P}_i\Big)$ are the new control points. We can use the same process to find higher order derivatives.

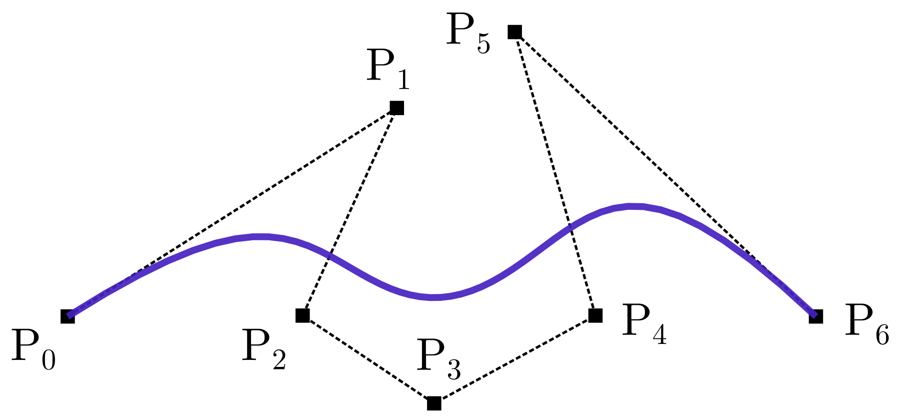


Figure 1.1: Bézier curve defined on the control points $\mathbf{P}_0, \ldots, \mathbf{P}_6$.

Cubic Bézier Curve

One of the most used form in computer graphics is the cubic Bézier curve

$$\boldsymbol{C}(t) = (1-t)^3\,\mathbf{P}_0 + 3(1-t)^2 t\,\mathbf{P}_1 + 3(1-t)t^2\,\mathbf{P}_2 + t^3\,\mathbf{P}_3$$

Notice that every control point has exceptionally intuitive geometric meaning. $\mathbf{P}_0$ and $\mathbf{P}_3$ are start and end points, while $\mathbf{P}_1$ and $\mathbf{P}_2$ define tangent direction at the endpoints respectively. These properties make it ideal for graphical user interfaces(GUI) where smooth curve is required such as in animation or color curve controls. Another useful application is in vector graphics and fonts, especially for its compact representation.

Unfortunately, there are a few limitations with Bézier curves. Let $\boldsymbol{C}(t)$ be a Bézier curve with control points $\mathbf{P}_0, \ldots, \mathbf{P}_n$. Consider moving a control point $\mathbf{P}_k$ in the direction $\boldsymbol{d}$. The resulting curve can be written as

$$\begin{aligned}
\boldsymbol{C}_{\boldsymbol{d}}(t) &= \sum_{i \neq k}^{n} B_{i,n}(t)\,\mathbf{P}_i + B_{k,n}(t)\Big(\mathbf{P}_k + \boldsymbol{d}\Big) \\
&= \sum_{i=0}^{n} B_{i,n}(t)\,\mathbf{P}_i + B_{k,n}(t)\boldsymbol{d} \\
&= \boldsymbol{C}(t) + B_{k,n}(t)\boldsymbol{d}
\end{aligned}$$

Since $B_{k,n}(t)$ is non-zero on the interval $(0,1)$, the whole curve is moved except for the endpoints.

This property makes it impossible to do a local change to a small part of a curve. Another problem is the dependence of curve degree on the number of control points, since n-th degree Bézier curve has $n+1$ points. Complex shapes require large number of control points, which in turn makes the curve unnecessarily smooth with high degree and causes computation to be numerically unstable. Moreover, we often need to include sharp corners in our shape design. To solve these issues, we can join several low degree Bézier curves into a spline. However, this solution is not completely satisfactory. Notice, that we need to make sure the endpoints between neighboring curves are the same to achieve C^0 continuity. We might also need to make sure tangents are the same to achieve C^1

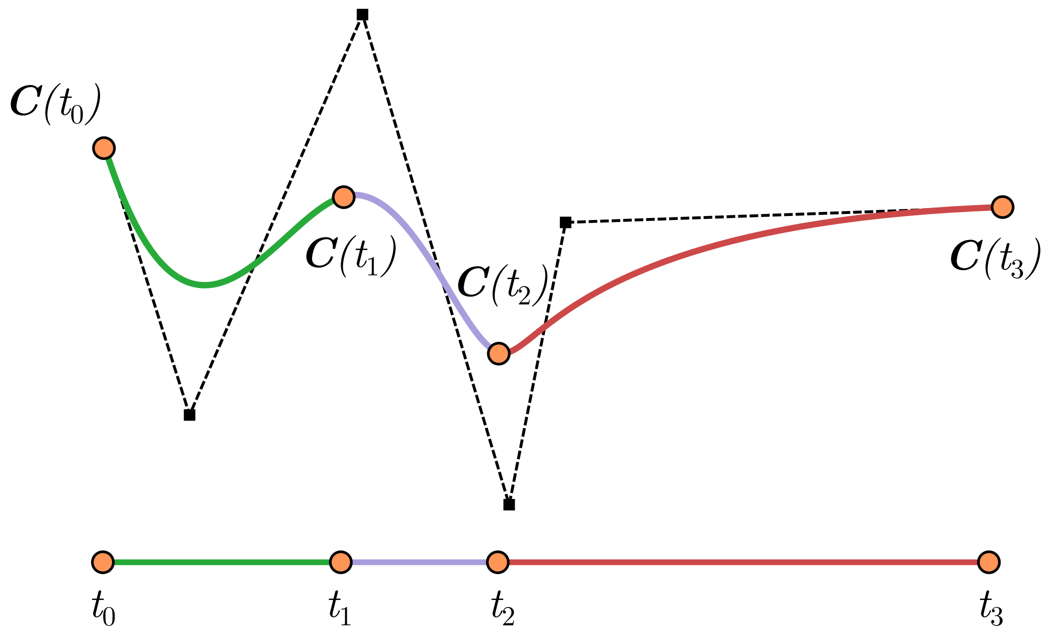


Figure 1.2: B-spline curve with knot values $t_0 \leq t_1 \leq t_2 \leq t_3$ showing the separate segments in color.

and so on for higher C^n continuity. Although it is possible to use such a system, maintaining constraints between curves for desired continuity becomes tedious, clunky and prone to errors. There is a better way.

1.3 B-spline Curves

Given a sequence of $n + 1$ control points $\mathbf{P}_0, \dots, \mathbf{P}_n$ and $m + 1$ knot values $t_0 \leq \cdots \leq t_m$, a B-spline curve of degree $p := m - n - 1$ is defined as

$$\boldsymbol{C}(t) = \sum_{i=0}^{n} N_{i,p}(t)\,\mathbf{P}_i \qquad t_0 \leq t \leq t_m \tag{1.3}$$

where $N_{i,p}(t)$ are B-spline basis functions of degree p defined by the *Cox-de Boor* recursion formula

$$N_{i,0}(t) := \begin{cases} 1 & \text{if} \quad t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,p}(t) := \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t)$$

with fractions $0/0$ defined to be 0.

Knot Vector

The *knots* $t_0 \leq \cdots \leq t_m$ are usually referred to as a *knot vector* $\{t_0, \dots, t_m\}$. The same knot t_i can be repeated several times. The number of repetitions is called

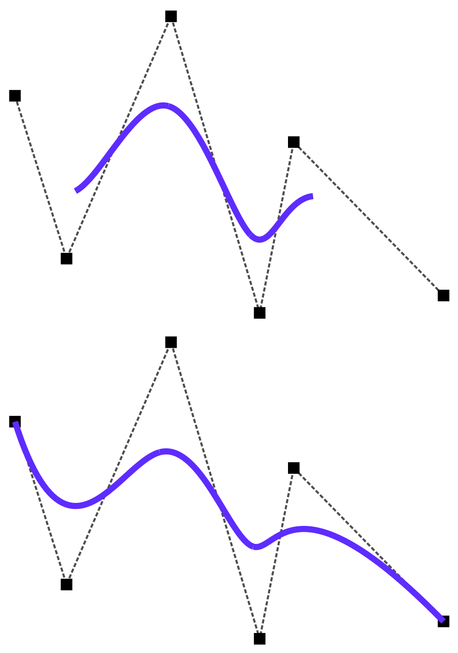


Figure 1.3: Two B-splines with the same control points are shown. Knot vector is the only difference, open(top), clamped(bottom).

multiplicity. The half-open interval $[t_i, t_{i+1})$ is the *i-th knot span.* Without any additional constraints a knot vector will generate an *open* B-spline curve, where start and end do not coincide with control points, see figure 1.3. To solve this issue the curve is *clamped*, which is achieved by increasing multiplicity of the first and last knot. The resulting knot vector has the form

$$\{\underbrace{a, \dots, a}_{p+1}, t_{p+1}, \dots, t_{m-p-1}, \underbrace{b, \dots, b}_{p+1}\}$$

Basis functions for open and clamped curves share the same properties, however for open curves, some of them are valid only for $t_p \leq t < t_{m-p}$. Appendix A shows manually computed basis functions and their graph with an open knot vector. We will only use clamped B-splines in this text unless stated otherwise. Knot vector is said to be *uniform* if all interior knots are equally spaced, i.e., every knot span has the same length. Although it can simplify basis function computation, it is rarely used.

Basis function properties

To better appreciate the B-spline basis function properties it is helpful to look at the definition as a pyramid/triangular scheme shown in figure 1.5.

1. **Degree:** $N_{i,p}(t)$ is a polynomial of degree p
   - Notice that $N_{i,p}$ is computed as a sum of $N_{i,p-1}$ and $N_{i+1,p-1}$, both multiplied by a linear function in t, therefore the degree is increased at each level of the pyramid.
2. **Non-negativity:** $N_{i,p}(t) \geq 0 \quad \forall t \in \mathbb{R}$

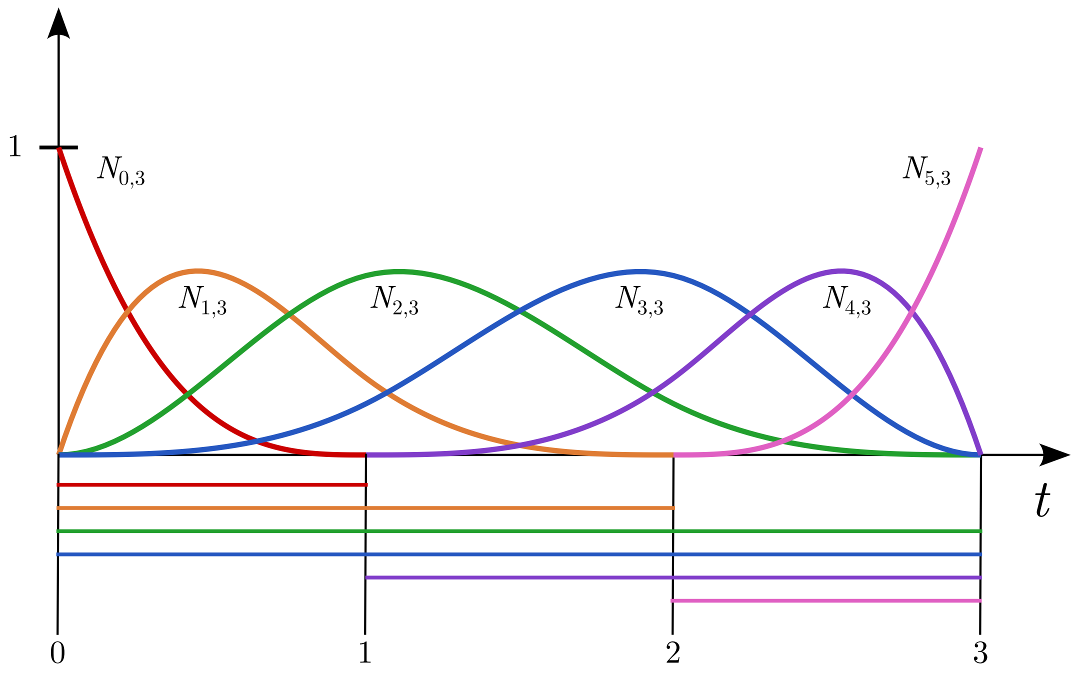


Figure 1.4: B-spline basis functions of degree 3 on the knot vector $\{0, 0, 0, 0, 1, 2, 3, 3, 3, 3\}$, and corresponding knot spans mapped below.

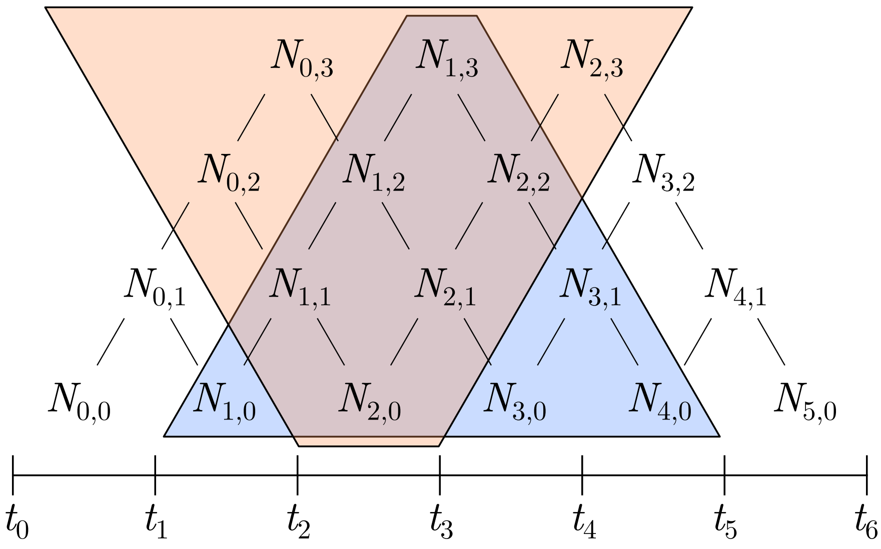


Figure 1.5: This diagram shows a pyramid of B-spline basis functions and their computational dependence. At the bottom are the 0-th degree basis functions with their associated knot spans. The blue trapezoid shows the support of $N_{1,3}$, while the orange one shows the local computation property of $[t_2, t_3)$.

3. **Local support:** $N_{i,p}(t) = 0$ outside the interval $[t_i, t_{i+p+1})$
   - Another important property is the *support*, i.e. interval where the function is non-zero. Bottom level basis functions $N_{i,0}$ have support on $[t_i, t_{i+1})$. Therefore, $N_{i,1}$ is non-zero on $[t_i, t_{i+2})$ since it is a linear combination of non-negative functions $N_{i,0}$ and $N_{i+1,0}$. We can follow this argument by induction to conclude that, in general, the B-spline basis function $N_{i,p}(t)$ is non-zero on $[t_i, t_{i+p+1})$, which can be easily seen in the diagram below.
4. **Local computation:** On the knot span $[t_i, t_{i+1})$, only $N_{i-p,p}(t), \ldots, N_{i,p}(t)$ are non-zero
   - It would be nice to have the opposite property to local support. We would like to know which basis functions are non-zero given the knot span $[t_i, t_{i+1})$. Borrowing the argument from above, $N_{i,0}$ is used by $N_{i-1,1}$ and $N_{i,1}$. Proceeding to the next degree, $N_{i,0}$ is transitively used by $N_{i-2,2}$, $N_{i-1,2}$ and $N_{i,2}$. Following this scheme until the degree p, $N_{i,0}$ is used in the recursive definition of $p+1$ basis functions $N_{i-p,p}, \ldots, N_{i,p}$. Note that for some $i < p$, not all of those functions exist, so generally, there are at most $p+1$ non-zero B-spline basis functions of degree p on $[t_i, t_{i+1})$. Again, this scheme is easy to see in the diagram below.
5. **Partition of unity:** $\displaystyle\sum_{j=i-p}^{i} N_{j,p}(t) = 1 \quad \forall t \in [t_i, t_{i+1})$, for all knot spans
6. **Knot multiplicity:**
   - At knot value t_i with multiplicity k, $N_{i,p}(t)$ is C^{p-k} continuous.
7. **Derivative:** $\displaystyle\frac{\mathrm{d}}{\mathrm{d}t} N_{i,p}(t) = N'_{i,p}(t) = p\left(\frac{N_{i,p-1}(t)}{t_{i+p} - t_i} - \frac{N_{i+1,p-1}(t)}{t_{i+p+1} - t_{i+1}}\right)$

Curve properties

- Clamped B-spline curve satisfies $\boldsymbol{C}(t_0) = \mathbf{P}_0$ and $\boldsymbol{C}(t_m) = \mathbf{P}_n$
- B-spline curve is a piecewise polynomial of degree p, where number of control points, $n+1$, and number of knots, $m+1$, satisfy $m = n + p + 1$.
- The entire curve lies in the convex hull of its control points. Looking at a knot span $[t_i, t_{i+1})$, only $N_{i-p,p}(t), \ldots, N_{i,p}(t)$ are non-zero. Therefore, only $\mathbf{P}_{i-p}, \ldots, \mathbf{P}_i$ contribute to the sum. Since these basis functions are non-negative and sum up to one, the curve is a convex combination of the corresponding control points.
- Modifying a control point $\mathbf{P}_i$ influences the curve only on the interval $[t_i, t_{i+p+1})$. Assume that we move $\mathbf{P}_k$ in the direction $\boldsymbol{d}$. Then the new

curve

$$\begin{aligned}\boldsymbol{C_d}(t) &= \sum_{i\neq k}^{n} N_{i,p}(t)\,\mathbf{P}_i + N_{k,p}(t)\Big(\mathbf{P}_k + \boldsymbol{d}\Big)\\ &= \sum_{i=0}^{n} N_{i,p}(t)\,\mathbf{P}_i + N_{k,p}(t)\boldsymbol{d}\\ &= \boldsymbol{C}(t) + N_{k,p}(t)\boldsymbol{d}\end{aligned}$$

Since $N_{k,p}(t)$ is non-zero only on the interval $[t_k, t_{k+p+1})$, the curve stays unchanged everywhere else. Therefore, we achieved local control over small part of the curve, which was missing in Bézier curves.

- Knot multiplicity controls continuity between knot spans. B-spline curve is infinitely differentiable inside the intervals. At knot values t_i with multiplicity k, $\boldsymbol{C}(t_i)$ is C^{p-k} continuous.
- B-spline curve with knot vector $\{0,\dots,0,1,\dots,1\}$ is a Bézier curve
- Curve shape is invariant under affine transformations. The argument is identical to the one for Bézier cuves, see section 1.2.
- Control points $\mathbf{P}_0,\dots,\mathbf{P}_n$ form a piecewise linear approximation to the curve, which can be improved by knot insertion or degree elevation, see chapter 3.
- Derivative of a clamped B-spline curve is another B-spline curve of degree $p-1$ with n new control points and $m-1$ knot values.

  $$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{C}(t) = \boldsymbol{C}'(t) &= \sum_{i=0}^{n} \frac{\mathrm{d}}{\mathrm{d}t} N_{i,p}(t)\,\mathbf{P}_i\\ &= \sum_{i=0}^{n-1} p\,\frac{N_{i+1,p-1}(t)}{t_{i+p+1}-t_{i+1}}\Big(\mathbf{P}_{i+1}-\mathbf{P}_i\Big) = \sum_{i=0}^{n-1} N_{i+1,p-1}(t)\,\mathbf{Q}_i\end{aligned}$$

  where $\mathbf{Q}_i := p\,\dfrac{\mathbf{P}_{i+1}-\mathbf{P}_i}{t_{i+p+1}-t_{i+1}}$ are the new control points.

  By removing first and last knot we obtain a B-spline curve

  $$\boldsymbol{C}'(t) = \sum_{i=0}^{n-1} N_{i,p-1}(t)\,\mathbf{Q}_i$$

  Note that this only works for a clamped knot vector. More detailed derivation can be found in Appendix A

So far, we have looked at smoothness, ease of design, manipulation and ease of computation properties. However, we have overlooked if these curves can represent geometric objects we might find useful. One of those are conic sections - circles, ellipses, hyperbolas. Let us look at a circle. It is a fact[] that one cannot describe every conic section, including circles, using simple polynomials. Since all curves we have described so far are polynomial, none of them can represent circles exactly. This might sound as a serious drawback given that circle is one of the simplest and most useful objects. Nevertheless, in the world of CAGD tolerance is always present and it is common to see approximations of a circle within some small epsilon. In spite of that, we would like to be able to represent conic sections exactly, and for that we need rational functions. For example, circle can be parametrized using stereographic projection as

$$\left(\frac{t^2-1}{1+t^2}, \frac{2t}{1+t^2}\right) \qquad -\infty \leq t \leq \infty$$

1.4 Rational Bézier Curves

Given a sequence of $n+1$ control points $\mathbf{P}_0, \dots, \mathbf{P}_n$ and $n+1$ weights $w_0, \dots, w_n$, a rational Bézier curve is defined as

$$\boldsymbol{C}(t) = \frac{\sum\limits_{i=0}^{n} B_{i,n}(t) w_i \mathbf{P}_i}{\sum\limits_{i=0}^{n} B_{i,n}(t) w_i} \qquad 0 \leq t \leq 1 \tag{1.4}$$

where $B_{i,n}(t)$ are Bernstein polynomials. To make it easier to see that this is a rational function, assume $\mathbf{P}_i = \big(x_i, y_i\big) \in \mathbb{R}^2$ and define polynomials

$$X(t) = \sum_{i=0}^{n} B_{i,n}(t) w_i x_i \qquad Y(t) = \sum_{i=0}^{n} B_{i,n}(t) w_i y_i \qquad W(t) = \sum_{i=0}^{n} B_{i,n}(t) w_i$$

then we can see that

$$\left(\frac{X(t)}{W(t)}, \frac{Y(t)}{W(t)}\right) = \frac{\sum\limits_{i=0}^{n} B_{i,n}(t) w_i \big(x_i, y_i\big)}{\sum\limits_{i=0}^{n} B_{i,n}(t) w_i} = \boldsymbol{C}(t)$$

Curve properties

Most of the properties are identical to non-rational Bézier curves. Here we will only list new and more general properties.

- If all the weights are set to constant c, i.e., $w_i = c \quad \forall i$ we have a non-rational Bézier curve.
- Setting $w_k = 0$ will "disable" the control point $\mathbf{P}_k$, i.e., $\mathbf{P}_k$ will have no influence on the curve shape. Furthermore, increasing w_k will move the curve closer to $\mathbf{P}_k$, while decreasing w_k will move it further away.
- Rational Bézier curves are projective invariant. Applying projective transformation to a curve is equivalent to applying it to the points only and then evaluating the curve with the transformed points.

Homogeneous Coordinates

Rational curves in d-dimensional Euclidean space can be embedded into d-dimensional projective space as a polynomial curve by using *homogeneous coordinates.* This allows us to replace point $\mathbf{P} = \Big(x, y, z\Big)$ and weight w with homogeneous point $\mathbf{P}^w = \Big(wx, wy, wz, w\Big)$. The original point $\mathbf{P}$ can be obtained from $\mathbf{P}^w$ by perspective division

$$\mathbf{P} = \mathbb{H}\{\mathbf{P}^w\} = \mathbb{H}\Big\{\Big(a, b, c, w\Big)\Big\} = \left(\frac{a}{w}, \frac{b}{w}, \frac{c}{w}\right)$$

Now we can represent rational curve in $\mathbb{R}^d$ with polynomial curve in $\mathbb{R}^{d+1}$ as

$$\boldsymbol{C}^w(t) = \sum_{i=0}^{n} B_{i,n}(t)\, \mathbf{P}_i^w \qquad 0 \le t \le 1$$

This allows us to work with rational curves the same way we do with polynomials ones. To see that the curves are identical consider applying perspective division to homogenous curve

$$\mathbb{H}\{\boldsymbol{C}^w(t)\} = \mathbb{H}\left\{\left(\sum_{i=0}^{n} B_{i,n}(t) w_i x_i,\ \sum_{i=0}^{n} B_{i,n}(t) w_i y_i,\ \sum_{i=0}^{n} B_{i,n}(t) w_i\right)\right\}$$

$$= \left(\frac{\sum_{i=0}^{n} B_{i,n}(t) w_i x_i}{\sum_{i=0}^{n} B_{i,n}(t) w_i}, \frac{\sum_{i=0}^{n} B_{i,n}(t) w_i y_i}{\sum_{i=0}^{n} B_{i,n}(t) w_i}\right) = \frac{\sum_{i=0}^{n} B_{i,n}(t) w_i \Big(x_i, y_i\Big)}{\sum_{i=0}^{n} B_{i,n}(t) w_i} = \boldsymbol{C}(t)$$

When we combine all the tricks from above (spline, knot vector and weights), we end up with the definition of **N**on-**U**niform, **R**ational **B**-**S**pline curve.

1.5 NURBS Curves

Given a sequence of $n+1$ control points $\mathbf{P}_0, \ldots, \mathbf{P}_n$, $n+1$ weights $w_0, \ldots, w_n$ and $m+1$ knot values $t_0 \le \cdots \le t_m$, a NURBS curve of degree $p := m - n - 1$ is defined as

$$\boldsymbol{C}(t) = \frac{\sum_{i=0}^{n} N_{i,p}(t) w_i\, \mathbf{P}_i}{\sum_{i=0}^{n} N_{i,p}(t) w_i} \qquad t_0 \le t \le t_m \tag{1.5}$$

where $N_{i,p}(t)$ are the B-spline basis functions of degree p defined in section 1.3. Using the rational basis function

$$R_{i,p}(t) := \frac{N_{i,p}(t) w_i}{\sum_{j=0}^{n} N_{j,p}(t) w_j}$$

allows us to write

$$\boldsymbol{C}(t) = \sum_{i=0}^{n} R_{i,p}(t)\, \mathbf{P}_i \qquad t_0 \le t \le t_m$$

Curve properties

Generally we assume weights are positive, however, setting a weight to zero or a negative number can have interesting effects. NURBS properties are, not surprisingly, similar to those of B-spline curves. We will only list some that are different.

- setting all the weights to a constant c, i.e., $w_i = c \quad \forall i$ produces a non-rational B-spline curve.
- Setting $w_k = 0$ will "disable" the control point $\mathbf{P}_k$, i.e., $\mathbf{P}_k$ will have no influence on the curve shape. Furthermore, increasing w_k will move the curve closer to $\mathbf{P}_k$, while decreasing w_k will move it further away.
- NURBS curves are projective invariant. Applying projective transformation to a curve is equivalent to constructing it from the transformed points.
- Local modification and convex hull properties are only valid for non-negative weights.

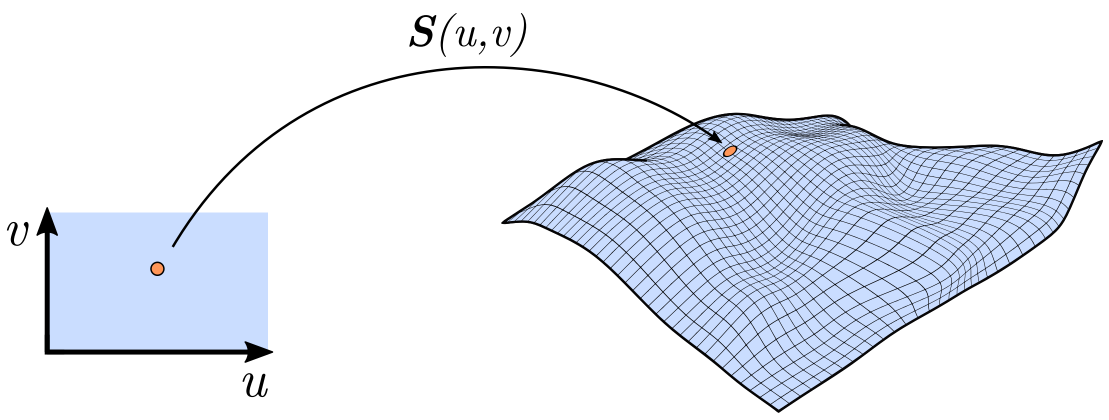
Homogeneous representation

$$\boldsymbol{C}^w(t) = \sum_{i=0}^{n} N_{i,p}(t)\,\mathbf{P}_i^w \qquad t_0 \leq t \leq t_m$$

Homogeneous coordinates offer compact storage and more efficient computation on a computer. Additionally, instead of creating new formulas and algorithms specifically for NURBS curves, it is easier to reuse B-spline ones and applying perspective divide at the end. For the rest of this thesis, we will use homogeneous form when referring to a NURBS curve, unless stated otherwise.

Chapter 2

Geometry of Surfaces



2.1 Parametric Surfaces

Parametric surface is a smooth function of the form

$$\boldsymbol{S}(u,v) : u,v \in \mathbb{R}^2 \longmapsto \boldsymbol{x} \in \mathbb{R}^3$$

We will denote partial derivatives of the surface as

$$\boldsymbol{S}_u(u,v) := \frac{\partial}{\partial u}\boldsymbol{S}(u,v) \qquad \boldsymbol{S}_v(u,v) := \frac{\partial}{\partial v}\boldsymbol{S}(u,v)$$

Normal vector at parameters u, v is defined by

$$\boldsymbol{N} = \frac{\boldsymbol{S}_u \times \boldsymbol{S}_v}{\|\boldsymbol{S}_u \times \boldsymbol{S}_v\|}$$

Tensor Product Surfaces

Tensor product surface is one of the simplest type of parametric surfaces. Imagine we have a curve

$$\boldsymbol{S}(u) = \sum_{i=0}^{n} F_i(u)\,\boldsymbol{P}_i$$

defined on some basis functions $F_i(u)$ and control points $\boldsymbol{P}_i$. Now, consider the control points $\boldsymbol{P}_i$ to be functions of independent parameter v, and define $\boldsymbol{P}_{i,j}$ to be $(n+1) \times (m+1)$ net of control points. Then for some $G_j(v)$ we have

$$\boldsymbol{P}_i(v) = \sum_{j=0}^{m} G_j(v)\, \boldsymbol{P}_{i,j}$$

and putting it all together we get the tensor product surface

$$\boldsymbol{S}(u,v) = \sum_{i=0}^{n} \sum_{i=0}^{n} F_i(u) G_j(v)\, \boldsymbol{P}_{i,j} \tag{2.1}$$

The name comes from function space point of view. The equation 2.1 represents a space of functions denoted $\boldsymbol{F} \otimes \boldsymbol{G}$, i.e. the tensor product of basis function spaces $\boldsymbol{F}$ and $\boldsymbol{G}$, spanned by $\{F_i(u)G_j(v)\}$.

Tensor product surfaces inherit properties of the univariate basis functions, making it easy to examine their behavior. Additionally, most operations performed on these surfaces are extensions of the curve algorithms applied to each row or column of the control points.

2.2 Bézier Surfaces

Given a $(n+1) \times (m+1)$ net of control points $\mathbf{P}_{i,j}$, a Bézier surface is defined as

$$\boldsymbol{S}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,n}(u) B_{j,m}(v)\, \mathbf{P}_{i,j} \qquad 0 \le u, v \le 1 \tag{2.2}$$

where $B_{i,n}(u)$ and $B_{j,m}(v)$ are the Bernstein polynomials defined in section 1.2.

Properties

- $\mathbf{P}_{0,0}$, $\mathbf{P}_{n,0}$, $\mathbf{P}_{0,m}$ and $\mathbf{P}_{n,m}$ are the corner points of the surface, i.e.

$$\begin{aligned} \boldsymbol{S}(0,0) &= \mathbf{P}_{0,0} & \qquad \boldsymbol{S}(1,0) &= \mathbf{P}_{n,0} \\ \boldsymbol{S}(0,1) &= \mathbf{P}_{0,m} & \qquad \boldsymbol{S}(1,1) &= \mathbf{P}_{n,m} \end{aligned}$$

- $\boldsymbol{S}(u,v)$ lies in the convex hull of its control points.
- The net of control points forms a piecewise linear approximation to the surface.
- Bézier surface is invariant under affine transformations.

2.3 B-spline Surfaces

Given a $(n+1) \times (m+1)$ net of control points $\mathbf{P}_{i,j}$, $h+1$ knot values in u-direction $u_0 \le \cdots \le u_h$ and $k+1$ knot values in v-direction $v_0 \le \cdots \le v_k$, a B-spline surface of degree $p \times q$ is defined as

$$\boldsymbol{S}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v)\, \mathbf{P}_{i,j} \qquad \begin{matrix} u_0 \le u \le u_h \\ v_0 \le v \le v_k \end{matrix} \tag{2.3}$$

where $p := h - n - 1$, $q := k - m - 1$ and $N_{i,p}(u)$, $N_{j,q}(v)$ are the B-spline basis functions defined in section 1.3. As before, we assume both knot vectors are clamped, that is

$$u_0 = u_1 = \cdots = u_p \quad \text{and} \quad u_{h-p} = \cdots = u_{h-1} = u_h$$

$$v_0 = v_1 = \cdots = v_q \quad \text{and} \quad v_{k-q} = \cdots = v_{k-1} = v_k$$

Properties

Following are just a generalization of the curve and B-spline basis function properties.

- $\mathbf{P}_{0,0}$, $\mathbf{P}_{n,0}$, $\mathbf{P}_{0,m}$ and $\mathbf{P}_{n,m}$ are the corner points of the surface, i.e.

$$\begin{aligned} \boldsymbol{S}(u_0, v_0) &= \mathbf{P}_{0,0} & \qquad \boldsymbol{S}(u_h, v_0) &= \mathbf{P}_{n,0} \\ \boldsymbol{S}(u_0, v_k) &= \mathbf{P}_{0,m} & \qquad \boldsymbol{S}(u_h, v_k) &= \mathbf{P}_{n,m} \end{aligned}$$

- $\boldsymbol{S}(u, v)$ lies in the convex hull of its control points.
- The net of control points forms a piecewise linear approximation to the surface.
- B-spline surface is invariant under affine transformations.
- Modifying a control point $\mathbf{P}_{i,j}$ only affects the surface locally in the rectangle $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$.
- B-spline surface with both knot vectors of the form $\{0, \ldots 0, 1, \ldots, 1\}$ is a Bézier surface.

2.4 NURBS Surfaces

Given a $(n + 1) \times (m + 1)$ net of control points $\mathbf{P}_{i,j}$, $(n + 1) \times (m + 1)$ net of weights $w_{i,j}$ for each control point, $h + 1$ knot values in u-direction $u_0 \leq \cdots \leq u_h$ and $k + 1$ knot values in v-direction $v_0 \leq \cdots \leq v_k$, a NURBS surface of degree $p \times q$ is defined as

$$\boldsymbol{S}(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v) w_{i,j} \, \mathbf{P}_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v) w_{i,j}} \qquad \begin{array}{l} u_0 \leq u \leq u_h \\ v_0 \leq v \leq v_k \end{array} \tag{2.4}$$

where $p := h - n - 1$, $q := k - m - 1$ and $N_{i,p}(u)$, $N_{j,q}(v)$ are the B-spline basis functions defined in section 1.3. We assume both knot vectors are clamped. By defining the rational surface basis functions as

$$R_{i,j}(u, v) := \frac{N_{i,p}(u) N_{j,q}(v) w_{i,j}}{\sum_{r=0}^{n} \sum_{s=0}^{m} N_{r,p}(u) N_{s,q}(v) w_{r,s}}$$

allows us to write

$$\boldsymbol{S}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} R_{i,j}(u,v)\,\mathbf{P}_{i,j}$$

Up until this section, every surface we defined was a tensor product surface. However, hold your horses, because rewriting the definition with rational surface basis functions reveals that NURBS surface is not a tensor product surface, since it cannot be written as a product of two univariate basis functions. Nevertheless, we can use homogeneous coordinates (see section 1.4) to transform it to a B-spline surface defined on a net homogeneous control points $\mathbf{P}^w_{i,j}$ as

$$\boldsymbol{S}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)\,\mathbf{P}^w_{i,j}$$

which is clearly a tensor product surface. We shall use this definition for a NURBS surface, unless stated otherwise.

2.5 Bézier Triangles

Tensor product surfaces are inherently defined on a rectangular domain, which forces a surface to have a square topology. Therefore, all such surfaces look like a wavy sheet of paper. One way to solve this issue is to deform an edge of the control points net into a single point, e.g. setting $\mathbf{P}_{0,0} = \mathbf{P}_{1,0} = \cdots = \mathbf{P}_{n,0}$. Topologically, it is still a square but visually this forms a triangle. The problem with this approach is that some algorithms might not work with such a degeneracy in mind, producing wrong results. Instead, let's look at how a triangular domain surface can be constructed.

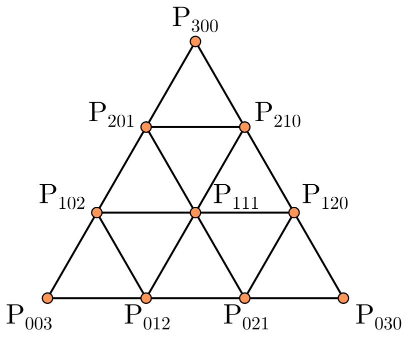


Figure 2.1: Arrangement of Bézier triangle control points.

Bézier triangle of degree n with $\binom{n+2}{2}$ (triangular number) control points $P_{i,j,k}$ arranged in a triangle as shown in a figure 2.1 is defined as

$$\boldsymbol{S}(u,v) = \sum_{\substack{i+j+k=n\\ i,j,k\geq 0}} B^n_{i,j,k}(u,v,1-u-v)\,\mathbf{P}_{i,j,k} \qquad \begin{matrix} 0\leq u,v\leq 1\\ u+v\leq 1\end{matrix} \tag{2.5}$$

where $B^n_{i,j,k}$ are triangular Bernstein polynomials defined on barycentric coordinates $\alpha+\beta+\gamma=1$ as

$$B^n_{i,j,k}(\alpha,\beta,\gamma) = \binom{n}{ijk}\alpha^i\beta^j\gamma^k = \frac{n!}{i!j!k!}\alpha^i\beta^j\gamma^k \qquad \begin{matrix} i+j+k=n\\ i,j,k\geq 0\end{matrix}$$

This concept can be generalized with B-splines and weights. For more details see [8], [9], [10].

2.6 Trimmed Surfaces

Another important requirement in CAD/CAM is the ability to model holes and cuts. Since the number of holes in a surface change its genus, it would be difficult to define such a surface mathematically. Instead, an engineering solution is used, where curves are used to delineate parts that are to be ”trimmed” away. There are two types of simple closed curves used. Outer curves specify the boundary of the surface, while inner curves specify holes. Each surface can have at most one outer curve and any amount of inner curves. In addition, inner curves must lie completely inside the outer boundary and have mutually disjoint interiors. Figure 2.2 shows this idea. Every data format defines its own semantics for interior and exterior of closed curves. Often, interior is to the left side of the direction of curve travel.

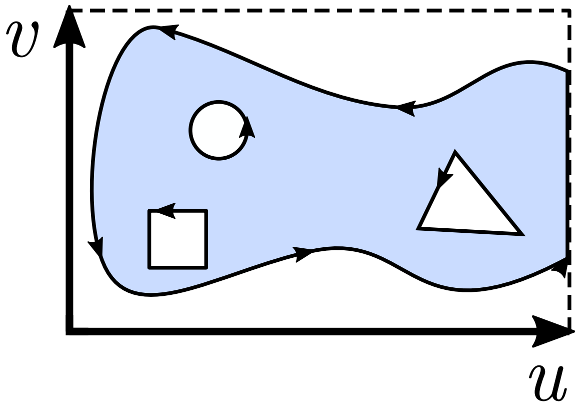


Figure 2.2: An example of a parametric domain with 1 outer curve and 3 inner curves.

Trimmed surface is a NURBS surface $\boldsymbol{S}(u, v)$ with an outer curve $\boldsymbol{C}_o(t)$ and $k \geq 0$ inner curves $\boldsymbol{C}_i(t)$. If no outer curve is specified, we implicitly construct one from the four boundary edges of the parametric rectangle. The curves produce values in the surface domain as

$$\boldsymbol{C} : \mathbb{R} \longmapsto \mathbb{R}^2$$
$$\boldsymbol{C}(t) = (u, v)$$

Chapter 3

Algorithms

This chapter will focus on algorithms to work with curves and surfaces defined in the previous chapters. One could do computations directly from definition, however, properties of basis functions allow us to reduce the amount of arithmetic operations that are needed and provide better numerical stability. We will show standard algorithms mainly for computing points, although, derivatives will also be discussed.

3.1 Bézier

Let's compute a point on Bézier curve $\boldsymbol{C}(t)$ defined by $n+1$ control points $\mathbf{P}_0, \ldots, \mathbf{P}_n$. By using the recursion property 4 of Bernstein polynomials we can write it as a linear interpolation of lower degree Bézier curves as

$$\boldsymbol{C}(t) = (1-t)\boldsymbol{C}_a(t) + t\boldsymbol{C}_b(t)$$

where $\boldsymbol{C}_a(t)$ is defined on $\mathbf{P}_0, \ldots, \mathbf{P}_{n-1}$ and $\boldsymbol{C}_b(t)$ is defined on $\mathbf{P}_1, \ldots, \mathbf{P}_n$. Repeating this process until the last point and collecting the terms bottom-up to reuse already computed values yields the *deCasteljau*[1] algorithm.

```
deCasteljau(P: Array, n: Int, t: Real) -> Point
{
    let Q = P;

    for (j = 0; j < n; ++j)
        for (i = 0; i < n - j; ++i)
            Q[i] = lerp(Q[i], Q[i+1], t);

    return Q[0];
}
```

This algorithm has a beautiful geometric interpretation shown below.

Similarly, for a Bézier surface we compute the intermediary points using the deCasteljau's algorithm in the u direction and then once again on the new points in the v direction.

[1]Paul de Casteljau independently formalized and developed Bézier curves at Citroën in 1959.

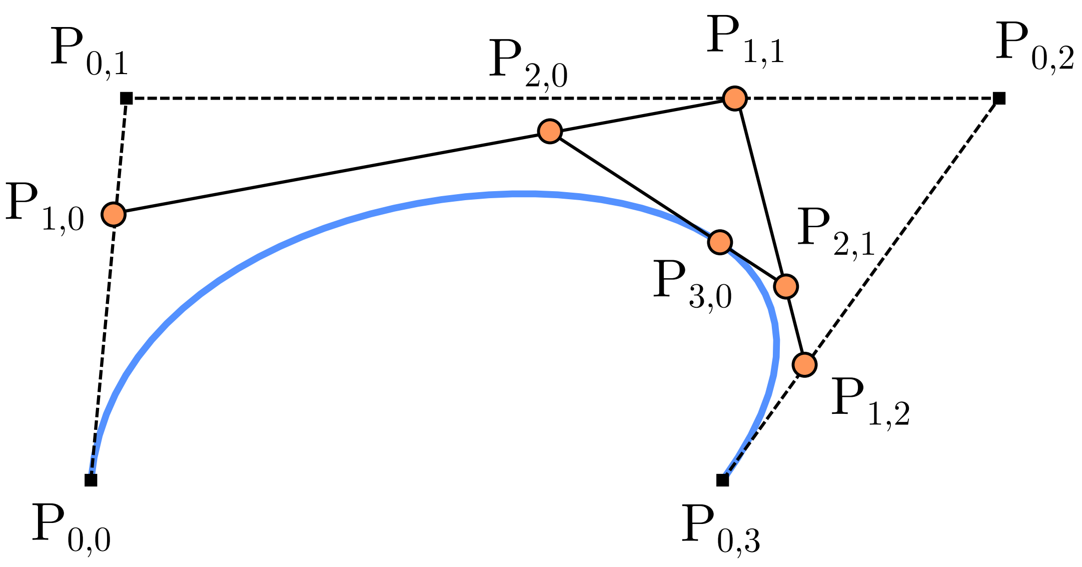


Figure 3.1: de Casteljau's algorithm on a 3rd degree Bézier curve, where $\mathbf{P}_{j,i}$ is the i-th control point on j-th iteration, i.e. $\mathbf{P}_{j,i} = lerp(\mathbf{P}_{j-1,i}, \mathbf{P}_{j-1,i+1}, 0.7)$.

```
deCasteljauSurface(
    P: Array, n: Int, m: Int, u: Real, v: Real
) -> Point
{
    let Q = Array(m + 1);

    for (j = 0; j <= m; ++j)
        Q[j] = deCasteljau(P[][j], n, u);
    deCasteljau(Q, m, v);

    return Q[0];
}
```

3.2 B-spline

Let $\boldsymbol{C}(t)$ be a B-spline curve of degree p with control points $\mathbf{P}_0, \dots, \mathbf{P}_n$ and clamped knot vector $T = \{t_0, \dots, t_m\}$. Evaluating a point at some parameter t, it would be wasteful to follow the definition, since most of the sum factors would be zero as a result of local support property. Better approach is to find all non-zero basis functions and only use those. To that end, we first need to find the knot span $[t_i, t_{i+1})$, given the parameter t. Generally, there are two options, linear or binary search. Although binary search might seem like an obvious choice, knot vectors are quite small in practice and linear search can be faster in some circumstances.

```
findKnotSpanIndex(T: Array, m: Int, p: Int, t: Real) -> Int
{
    if (t == T[m - p])
        return m - p - 1;

    let low = p;
    let high = m + 1;
    let mid = (low + high) / 2;

    while (t < T[mid] || t >= T[mid + 1])
    {
        if (t < T[mid])
            high = mid;
        else
            low = mid;

        mid = (low + high) / 2;
    }

    return mid;
}
```

The check at the start is for handling the case when $t = t_m$, since we are using half-open intervals.

Next, we present an algorithm to compute all non-zero basis functions on interval $[t_i, t_{i+1})$.

```
bSplineBasisFunctions(
    T: Array, m: Int, p: Int, t: Real, i: Int
) -> Array
{
    let N = Array(p + 1);
    let L = Array(p + 1);
    let R = Array(p + 1);

    N[0] = 1.0;
    for (j = 1; j <= p; ++j)
    {
        L[j] = t - T[i + 1 - j];
        R[j] = T[i + j] - t;

        let result = 0.0;
        for (k = 0; k < j; ++k)
        {
            let w = N[k] / (R[k + 1] + L[j - k]);
            N[k] = result + w * R[k + 1];
            result = w * L[j - k];
        }

        N[j] = result;
    }

    return N;
}
```

Finally, evaluating point is just a matter of calling the functions above.

```
evaluateBsplineCurve(
    P: Array, n: Int, T: Array, m: Int, p: Int, t: Real
) -> Point
{
    let i = findKnotSpanIndex(T, m, p, t);
    let N = bSplineBasisFunctions(T, m, p, t, i);

    let C = 0.0;
    for (j = 0; j <= p; ++j)
        C += N[j] * P[i - p + j];

    return C;
}
```

For completeness, we also provide code for evaluating point on a B-spline surface with $(n + 1) \times (m + 1)$ control points $P_{i,j}$ U r s

```
evaluateBsplineSurface(
    P: Array, n: Int, T: Array, m: Int, p: Int, t: Real
) -> Point
{
    let i = findKnotSpanIndex(T, m, p, t);
    let N = bSplineBasisFunctions(T, m, p, t, i);

    let C = 0.0;
    for (j = 0; j <= p; ++j)
    C += N[j] * P[i - p + j];

    return C;
}
```

One of the most important algorithm and a basic block for other procedures is knot insertion. The result is a new curve with knot t inserted into knot vector r times. We will provide an implementation taken from [4].

```
knotInsertion(
    P: Array, n: Int, T: Array, m: Int, p: Int, t: Real, r: Int
) -> { Array, Int, Array, Int }
{
    let s = p - r;
    let newT = Array(n + p + 1 + r);

    // Fill in the new knot vector
    for (i = 0; i <= k; ++i)
        newT[i] = T[i];
    for (i = 1; i <= r; ++i)
        newT[k + i] = t;
    for (i = k + 1; i <= n + p + 1; ++i)
        newT[i + r] = T[i];

    for (i = 0; i <= k - p; ++i)
        newP[i] = P[i];
    for (i = k - s; i <= n; ++i)
        newP[i + r] = P[i];

    let R = Array(p + 1);
    for (i = 0; i <= p - s; ++i)
        Rw[i] = P[k - p + i];

    for (j = 1; j <= r; ++j)
    {
        let L = k - p + j;
        for (i = 0; i <= p - j - s; ++i)
        {
            let a = (t - T[L + i]) / (T[i + k + 1] - T[L + i]);
            R[i] = lerp(R[i], R[i + 1], a);
        }
        newP[L] = R[0];
        newP[k + r - j - s] = R[p - j - s];
    }

    for (i = L + 1; i < k - s; ++i)
        newP[i] = R[i - L];

    return {newP, n+r, newT, m+r};
}
```

3.3 NURBS

We have already mentioned that we use homogeneous representation for NURBS curves and surfaces, so we will only provide couple of examples on how to deal with projective division with derivatives.

```
evaluateNURBSCurve(
    Pw: Array, n: Int, T: Array, m: Int, p: Int, t: Real
) -> Point
{
    let Cw = evaluateBsplineCurve(Pw, n, T, m, p, t);
    return Cw.xyz / Cw.w;
}
```

The same idea holds for evaluating a point on a NURBS surface. It is similar for derivatives, however, we have to remember to use chain rule.

$$\boldsymbol{S}(u,v) = \frac{w(u,v)\boldsymbol{S}(u,v)}{w(u,v)} = \frac{\boldsymbol{S}^w(u,v)}{w(u,v)}$$

where $w(u,v)$ is the weight of the homogeneous point computed at (u,v). Computing derivative we obtain

$$\boldsymbol{S}_u(u,v) = \frac{\boldsymbol{S}_u^w(u,v) - w_u(u,v)\boldsymbol{S}(u,v)}{w(u,v)} \tag{3.1}$$

Chapter 4

Tessellation

This chapter contains discussion about different tessellation strategies and requirements found in the CAD/CAM industry. We mention use cases and provide relevant references for each specific domain. Next, we describe our tessellation algorithm and motivation behind certain architectural choices. Finally, at the end of the chapter we present the results of our algorithm.

4.1 Use Cases

The first step in designing any kind of algorithm is to define a set of design goals and requirements put on the computation and the results. This is especially true for tessellation, since a little change in design goals might lead to completely different runtime performance and quality of the resulting polygonal mesh. The design generally revolves around what further computations are to be performed on the mesh. Some application require very uniform tessellation, while others are better off with large triangle size disparities. In the following list, we describe the two most common applications and their requirements.

Simulation

Simulation is undoubtedly one of the backbones of the CAD/CAM industry. Producing and machining complex parts is very expensive and time consuming process. Additionally, designing and testing a new product might require tens or hundreds of prototypes to optimize required parameters, giving an economic incentive for simulation. By modeling the part on a computer we can compute pretty much all of the physical properties, ranging from static ones like volume, surface area and mass to dynamic quantities like tensile stress, load distribution or thermal conductivity. Recently, advances in computer performance allowed to directly optimize the shape for certain constraints and properties.

Most of these properties can be calculated by solving partial differential equations(PDEs) over the surface or volume of a model. Solving these equations requires a geometrical description simple enough to facilitate analytical solutions. The most common technique used is the Finite Elements Method(FEM). It works by building the approximate solution from individual triangles on a tessellated mesh. Denser tessellations produce better solutions, however, equilateral triangles of uniform size are required for the best results. More detailed explanation of

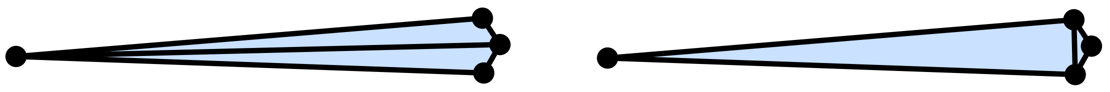


Figure 4.1: Difference between skinny triangles (left) and better triangulation (right)

FEM can be found in [11]. There is also research on grid-free methods for solving PDEs inspired by Monte Carlo techniques from photorealistic rendering, see [12].

Visualization

The other big use case for tessellation is visualization. From the beginning of CAD/CAM, users working on a computer needed to see the data to be able to meaningfully interact with it. This was facilitated by custom hardware and rendering subroutines in the early days. However, the introduction of GPUs to consumer market gave rise to tessellation approach we use today. The idea is to compute the slow tessellation offline and use the mesh for fast real-time rendering. In recent years, virtual reality devices further necessitated performance optimizations to make the actual rendering as fast as possible. On the other hands, there are still use cases for offline rendering. Although we can and often do ray-tracing on the tessellated mesh, there is also a possibility to do ray-NURBS itersections directly. The problem with ray-tracing parametric surfaces is inability to find an exact analytical solution. This means that we need to use numerical methods to find the precise intersection point. Most used approach is the Newton's method with a good initial starting point. See [13] and [14].

Unlike in simulation, visualization does not necessitate uniform meshes with approximately equilateral triangles. Using adaptive triangle sizes based on local geometry features can significantly reduce the number of triangles with little to no loss of quality. However, we still have to keep an eye out for skinny triangles(figure 4.1), which can cause problems such as numerical interpolation inaccuracies or pixel shader overdraw.

4.2 Approximation Error

Approximating C^2 curves and surfaces with piecewise linear structures requires a definition of approximation error measure. Simplest of the bunch is parametric distance. Given a curve $\boldsymbol{C}(t)$ and its piecewise linear approximation $\boldsymbol{L}(t)$ we define the error as

$$d_P(\boldsymbol{C}, \boldsymbol{L}) = \sup_t \|\boldsymbol{C}(t) - \boldsymbol{L}(t)\|$$

This of course depends on the actual parametrizations. Generally in tessellation we do not care about parametric distance but rather want to know if a point $\boldsymbol{L}(t)$ is within tolerance of some other point on the curve $\boldsymbol{C}(t)$. For this purpose we can use Hausdorff distance (sometimes called geometric distance) defined as

$$d_H(\boldsymbol{C}, \boldsymbol{L}) = \max\left\{ \sup_{P \in \boldsymbol{C}(t)} \inf_{Q \in \boldsymbol{L}(t)} \|P - Q\|, \sup_{Q \in \boldsymbol{L}(t)} \inf_{P \in \boldsymbol{C}(t)} \|P - Q\| \right\}$$

Both definitions can be trivially generalized to surfaces. Following the discussion in [15], it can be shown that $d_H(C, L) \leq d_P(C, L)$, which means we can use the parametric distance with the added benefit of being easier to compute.

Standard result from [15] shows that the parametric distance between a smooth curve and its linear approximation can be bounded from above by the second derivative, i.e.

$$\sup_{a \leq t \leq b} \|\boldsymbol{C}(t) - \boldsymbol{L}(t)\| \leq \frac{1}{8}(b-a)^2 \sup_{a \leq t \leq b} \|\boldsymbol{C}''(t)\|$$

By uniformly subdividing the curve into segments of equal length and requiring the error tolerance to be $\varepsilon > 0$ we arrive at the lower bound for the number of segments n needed

$$n \geq \sqrt{\frac{(b-a)^2 \sup_{a \leq t \leq b} \|\boldsymbol{C}''(t)\|}{8\varepsilon}}$$

This number, however, can be too large even for tight second derivative estimates.

Analogous error upper bounds can be devised for surfaces. Following [16], given a smooth parametric surface $\boldsymbol{S}(u, v)$, triangle T and a linear parametrization of this triangle $\boldsymbol{L}(u, v)$, we can show

$$\sup_{(u,v) \in T} \|\boldsymbol{S}(u, v) - \boldsymbol{L}(u, v)\| \leq \frac{2}{9}\Omega^2(M_1 + 2M_2 + M_3)$$

where Ω is the maximal edge length of the triangle and

$$M_1 = \sup_{(u,v) \in T} \left\| \frac{\partial^2 \boldsymbol{S}(u, v)}{\partial u^2} \right\| \quad M_2 = \sup_{(u,v) \in T} \left\| \frac{\partial^2 \boldsymbol{S}(u, v)}{\partial u \partial v} \right\| \quad M_3 = \sup_{(u,v) \in T} \left\| \frac{\partial^2 \boldsymbol{S}(u, v)}{\partial v^2} \right\|$$

Again, by uniform subdivision we can determine the required number of segments n in u direction and the number of segments m in v direction, forming a grid, such that the error measure is bounded by a given tolerance $\varepsilon > 0$.

Curve and Surface Sampling

With a suitable error measure, the next step is to sample points such that the resulting discrete approximation is within tolerance of the continuous input spline. We can improve uniform subdivision by observing that most curves and surfaces have areas of low curvature, where we only need a few samples, and areas of high curvature, where we need more samples. Such a method for sampling curves and surfaces based on curvature is explored in [17]. Similar optimization based algorithms can produce excellent results with a limited number of points in exchange for efficiency. Therefore, we opted for a recursive algorithm described in a later section.

Delaunay Triangulation

The next step after sampling the surface and all the outer and inner boundary curves is the triangulation where the sampled points are connected to form a triangle mesh. This process can severely influence the properties and quality of the individual triangles of the mesh. Delaunay triangulation is the standard algorithm for this task. It is defined by the property that no sampled point lies inside a circumcircle of any triangle. An important property of this triangulation in 2D is that it maximizes the minimum angle of triangles. There exists a zoo of papers constructing the Delaunay triangulation with different algorithmic properties, however, a simple edge flip procedure(see [18]) is often sufficient enough. Additionally, an extension is required for outer and inner boundary edges, since they cannot be flipped. This is known as a constrained Delaunay triangulation where it also keeps track of fixed edges that are forbidden to be altered or removed.

Most of the relevant literature performs triangulation in a parametric domain due to its relative simplicity, however some authors, e.g. [19], [20], also explored a three-dimensional triangulation approach, which benefits from using the final sampled points in 3D but suffers from high complexity and difficult implementation.

4.3 Tessellation Algorithm

This section contains a detailed description of the tessellation algorithm we designed and implemented, together with motivations and constraints that influenced out choices. To judge the final results we first outline a few objectives in order of priority:

1. **Correctness:** no empty triangles and no non-manifold vertices or edges are allowed

2. **Runtime performance:** even though tessellation is a preprocessing step, we do not want to keep users waiting on data import for ages

3. **Output size:** limit the amount of triangles produced

Our goal was to implement an algorithm suitable for visualization purposes, targeting ray-tracing and real-time rendering. The requirements on the triangular mesh are similar for both, minimize the amount of triangles and create detail only where necessary. One important concept in tessellation is the always present ε tolerance. This is one of the inputs of our algorithm and specifies the maximum allowed distance between the parametric surface and the resulting mesh. We proceed with the description of tessellating NURBS surfaces, next, we show how trimming curves come into play and will finish this section discussing removal of unwanted gaps.

Surface Tessellation

Uniform grid sampling of the surface is the easiest method of tessellation, where we increase the grid resolution until the tolerance constrained is satisfied. Clearly, this will produce unnecessarily large amount of samples on flat parts of a surface. Ideally, we would sample points according to the local curvature, producing more samples in highly curved areas, while creating relatively few in flat areas, as can be seen in [17]. Nevertheless, computing curvature or ideal parametrization can be costly. Another common approach is to compute bounds on the second partial derivatives, see [16], to approximate flatness of the given region and get an upper bound of triangle edge length. We have followed [2] and utilized a quadtree structure. At each level we check for approximate flatness and decide if we need to subdivide further. One minor issue with quadtrees is the requirement to split both parametric dimensions, even if the surface is completely flat in one direction. However, we observed that this improved uniformity of the triangles especially on the boundaries, while also benefiting from a simpler implementation. Flatness testing can be accomplished with the bound from the section 4.1 or approximated with additional samples. Subdivision is performed with knot insertion, similarly to boundary curves as described in the next subsection.

```
TessellateAll
Input: A set of NURBS surface patches
Output: Triangle mesh

  1. For each surface patch $\boldsymbol{S}$:
         Tessellate($\boldsymbol{S}$)
         RepairMesh        (optional)

  2. Sew all meshes together
```

```
Tessellate($\boldsymbol{S}$)
Input: NURBS surface patch $\boldsymbol{S}$, tolerance $\varepsilon$
Output: Triangle mesh

  1. Sample boundary curves of $\boldsymbol{S}$

  2. Sample surface $\boldsymbol{S}$

  3. Triangulate sampled points:
         constrain boundary edges
         incrementally add sample points

  4. For each sample point $(u, v)$ in parametric space
         evaluate point $\boldsymbol{S}(u, v)$
         evaluate normal
```

Trimming Curves

The introduction of trimming curves complicates things a bit. At first, we need to tessellate individual trimming curves into polygons to simplify inside/outside queries. Then we can remove surface samples that are trimmed by the curves. And finally, we can connect curve polygons with the quad grid.

In the original implementation(see [3]) authors converted each NURBS curve into several Bézier curves to simplify this process. However, conversions introduce additional error. Instead, we opted for direct tessellation of NURBS curves. We use a modification of the knot insertion algorithm to produce a piecewise linear

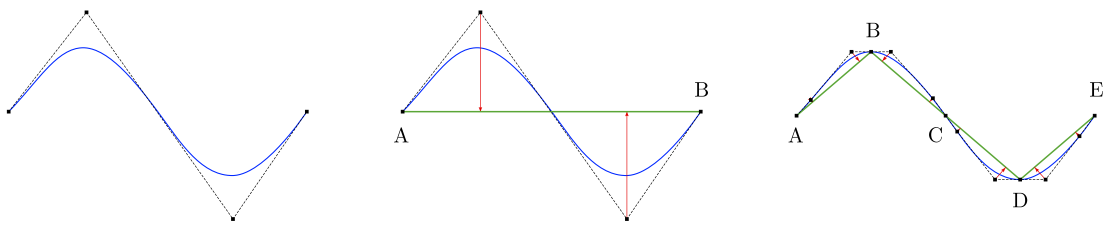


Figure 4.2: Linear approximation of a curve. Left: Original curve and its control points. Middle: Tolerance test in the first iteration. Right: Tolerance test in the third iteration.

approximation of curves. Diagram of the process is shown in figure 4.2. It works by recursively splitting the curve and checking the distance from control points to the polyline. Distance is computed in $\mathbb{R}^3$ by evaluating surface that the curve lies on, described in section 3.3. Control polygon is always further away, therefore, we use it as an upper bound for curve tolerance. Each time we split the curve its control polygon gets closer, tightening this bound.

In some applications, it might be useful to specify a standalone tolerance just for tessellating boundary curves to give users ability to choose the required quality of small details like logos and text in relation to the overall shape of the model.

After removing all samples lying outside we join the polygon boundary with the grid. This can be done by iteratively adding points by finding the triangle it lies in and splitting it into 3 more triangles while keeping the constraints on the boundary edges. Unfortunately, this will result in very narrow triangles creating complications further in line. This is a classic problem in computational geometry that can be solved with the help of Constrained Delaunay Triangulation(CDT). We opted for the algorithm presented in [21]. We try to avoid unnecessary computation of Delaunay triangulation in the whole interior and try to limit edge swapping to boundary areas, since we do not need perfect mesh for visualization. On the other hand, some authors, e.g. [22], go the other way and optimize the mesh in $\mathbb{R}^3$ to create very regular tessellation, mainly for FEM applications.

```
Sample surface
Input: NURBS surface $\boldsymbol{S}$, tolerance $\varepsilon$
Output: Sampled points

1. If isFlatEnough($\boldsymbol{S}$,$\varepsilon$)
       return corner points
2. Else
       Split $\boldsymbol{S}$ into four square regions
       Recursively call this procedure on all four regions
```

Sewing

One annoying characteristic of trimmed NURBS surfaces is the lack of precision at the boundaries, since surface intersections require insanely high degree curves to be exact, and are therefore only approximated with NURBS curves. This

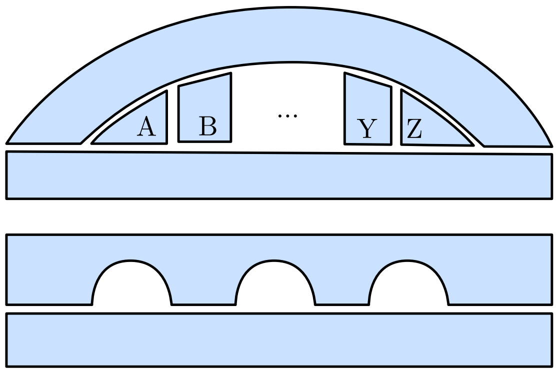


Figure 4.3: Non-trivial neighboring surfaces. Top: Unbounded number of surfaces between two neighbors. Bottom: Neighboring surfaces coming in and out of contact.

problem is further magnified by creating a piecewise linear approximation of surface boundaries during the tessellation process. This results in undesirable cracks and holes between neighboring patches.

At first, we thought about devising a scheme to create the same linear approximation of boundary curves between neighboring surfaces. However, this naive attempt has several big problems:

- Many CAD data formats do not provide connectivity information between patches.
- Boundary curves (as described in 2.6) are defined on surfaces they lie on, therefore, can have vastly different parametrizations in $\mathbb{R}^3$.
- There can be many places where surfaces come in and out of contact, with unbounded number of other surfaces in between, as show in figure 4.3.

The only viable approach to remove these gaps is to do it in a post-processing step after tessellation. We have followed in the steps of [1] and implemented our own triangle mesh merging algorithm described as follows:

SewAll
Input: N surface meshes
Output: N surface meshes with no holes (see figure 4.5)

```
1. For each neighboring surface patches $A$ and $B$:
       Sew($A$,$B$)     (sew $A$ onto $B$)
       Sew($B$,$A$)     (sew $B$ onto $A$)
```

Sew(A,B)
Input: Neighboring surface meshes A and B
Output: Modified mesh A sown onto B

```
1. Get boundary vertices and edges of $A$ and $B$.
2. For each $E$ boundary edge of $A$:
```

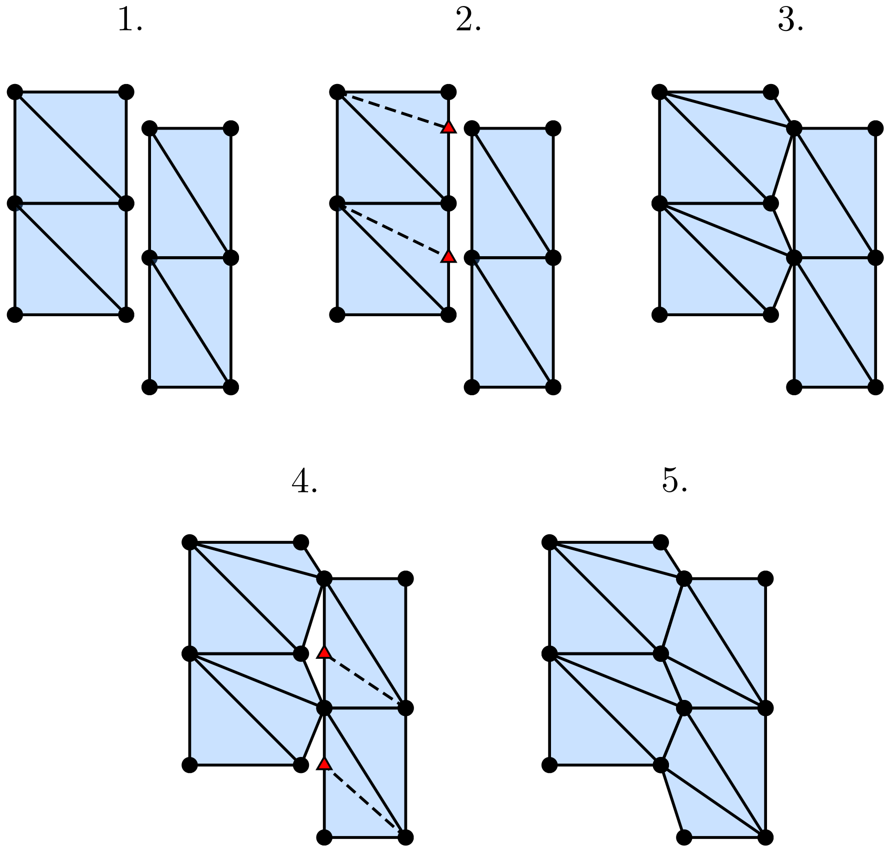


Figure 4.4: Visual example of each step of the sewing algorithm.

```
    Find a boundary vertex $V$ within tolerance to the edge
    Project $V$ onto $E$
    Split the edge at the projected position
```

For every patch we find the boundary edges and vertices by counting. Non-boundary edges are contained in exactly two faces, so by iterating all faces and keeping the counts, we can quickly get our result. Then by utilizing boundary boxes, we filter out every pair of patches that are too far apart and only apply the sewing algorithm to the rest.

The sewing algorithm can be seen in figure 4.4. It works asymmetrically by first gluing A to B and then B to A to make this algorithm easier to parallelize by limiting modification to only one mesh. We iterate through every boundary edge and search for boundary vertices on the other mesh that are within sewing tolerance. For each such vertex we project it on the edge and split the triangle as shown in step 2. If the projected vertex is too close to one of the edge vertices we do not create a new triangle and instead merge the vertices. Then we move those points to close the gap, see step 3. Finally, we repeat the same process the other way around, steps 4 and 5. This ensures that we reliably stitch arbitrarily complex boundaries coming in and out of contact. Figure 4.5 show the same process on the boundary of a circle. We can clearly see gaps and overlaps on the left and a clean mesh at the right after sewing.

Since the algorithm takes place in $\mathbb{R}^3$, one has to be careful to avoid merging boundaries with sharp angle to prevent normal smoothing along that edge.

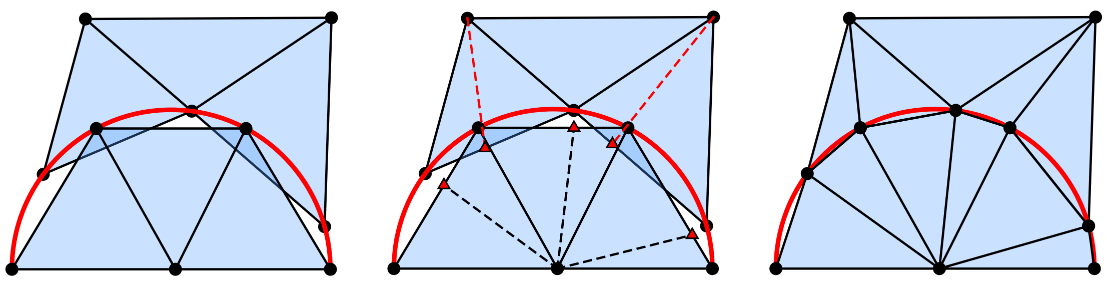


Figure 4.5: Example of the sewing algorithm on a circular boundary.

Instead, the vertices are duplicated and kept at the same position. Depending on the quality of input data it might be necessary to perform a healing/repair step right after tessellation to fix broken normal vectors or non-manifold mesh degeneracies. We do not need to perform this step, but your mileage may vary.

Mesh Decimation

When trying to tessellate an approximate mesh with small number of triangles it might be tempting to just increase the tolerance. However, sewing these low quality patches together will create an unsatisfactory result. Much better option is to use standard tolerance value that will preserve necessary detail in the model and then run a mesh decimation algorithm which can selectively target low curvature areas and produce high quality low poly result. With enough runtime budget, it might be a good idea to perform this decimation step even in normal operation.

4.4 Results

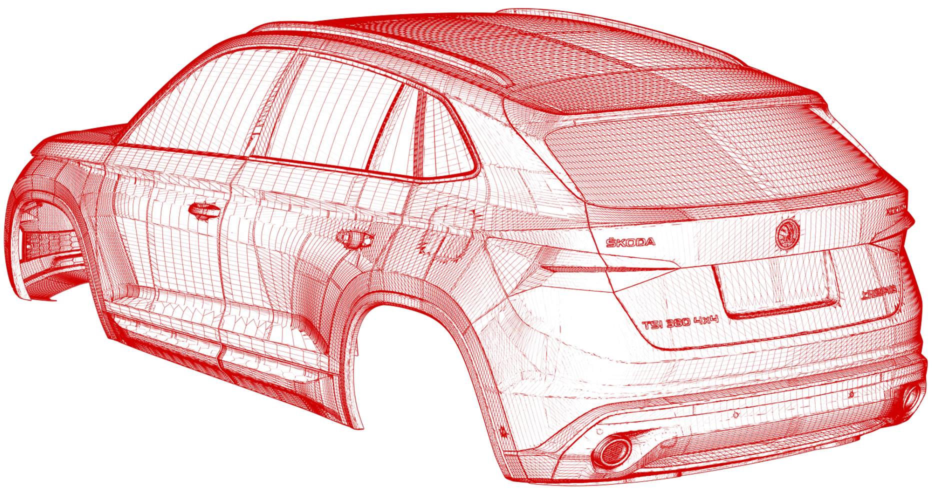


Figure 4.6: Tessellation of Škoda Kodiaq.

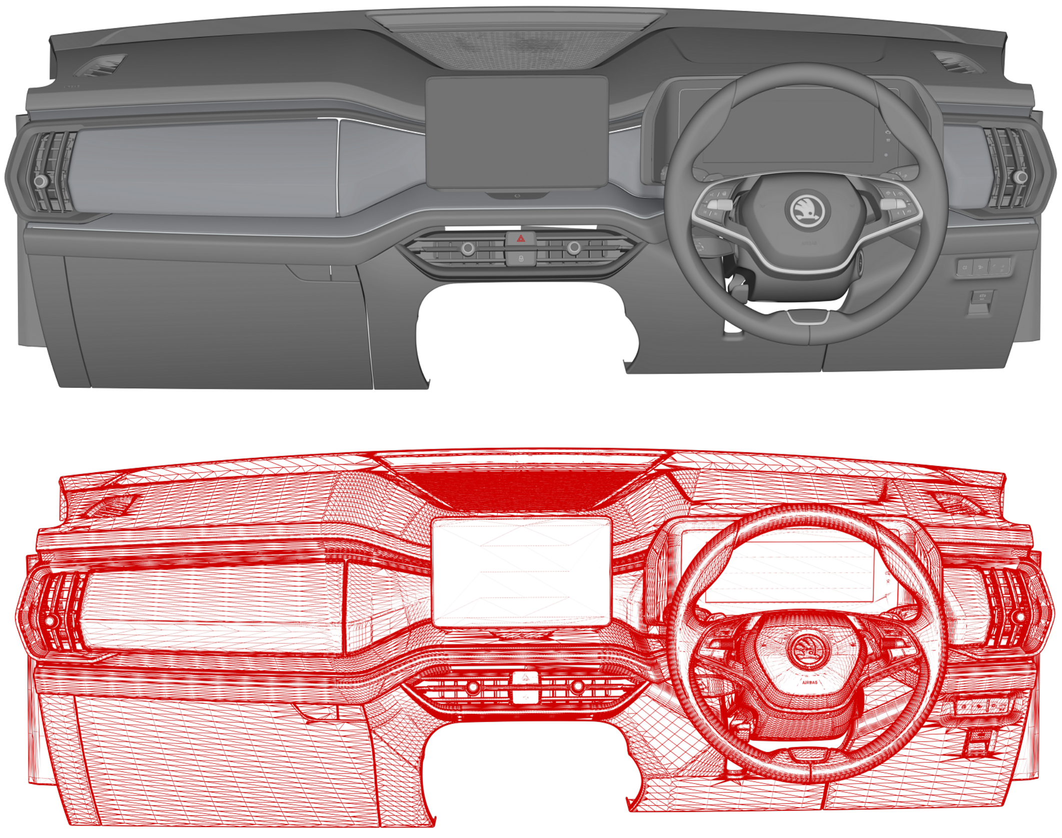


Figure 4.7: Interior design of a car; diffuse lighting is applied at the top and tessellated wireframe is shown at the bottom.

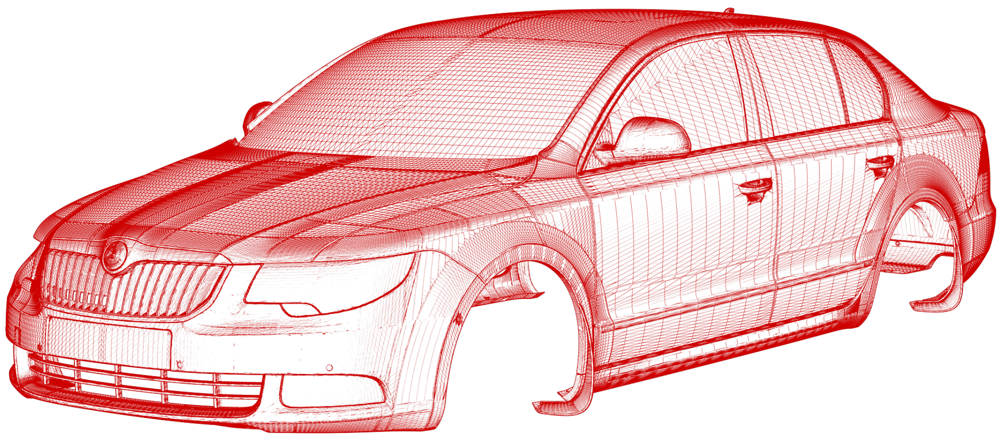


Figure 4.8: Exterior tessellation of Škoda Superb model.

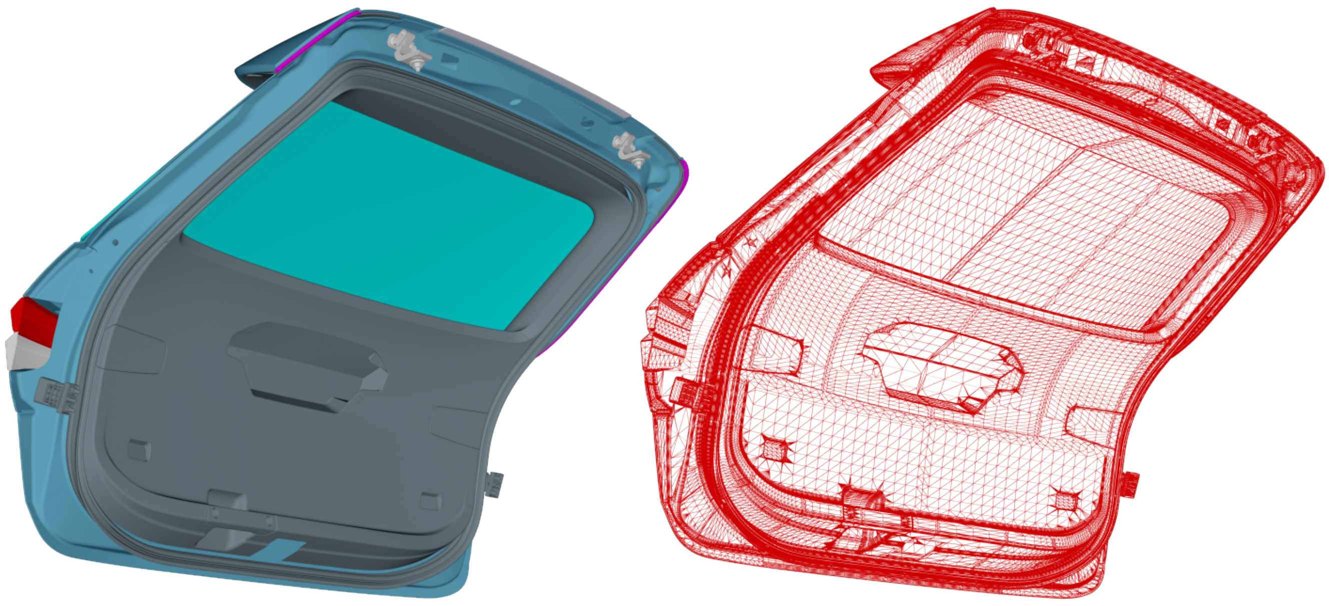


Figure 4.9: A random trunk door; left: diffuse color, right: tessellated wireframe.

Chapter 5

Implementation Details

In this chapter we will discuss and illustrate challenges associated with developing software utilizing CAD data and tools for visualization, optimization, simulation or other computational tasks. It is provided mainly as a means of familiarizing with the world of CAD/CAM for people outside the industry.

5.1 Algorithm Details

Trimming Curves

On several occasions we observed very long boundary curves with thousands of knots, geometrically consisting of several segments with C^0 continuity between them, as in example shown in figure 5.1.

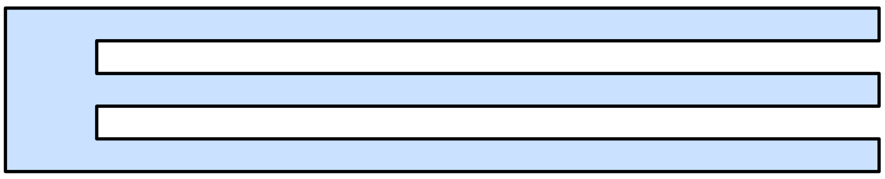


Figure 5.1: A single, long trimming curve with C^0 continuity between segments.

These breakpoints can be easily determined with the help of knot multiplicity property, mentioned in 1.3, and by utilizing knot insertion algorithm(see section 3.2) allows them to be split into individual curves and tessellated separately. This small adjustment substantially improves performance and tessellation quality for this type of boundary curves.

Sewing

Finding the boundary of triangular mesh is implemented by sorting all triangle edges and keeping the unique ones, since every non-boundary edge is incident to exactly two triangles. Boundary edges and vertices are then kept in separate arrays to simplify next steps.

```
struct Boundary
{
```

```
std::vector<edge_t> edges;
std::vector<index_t> vertices;
};
```

Theoretically, we could obtain this data directly from tessellation algorithm, however, CAD data might contain already triangulated meshes, so we used this unified approach.

In the implementation of the sewing algorithm it is important to make sure that the individual steps are parallelizable, i.e. thread safe. The easiest way is to keep the original patches read-only and write the modifications in a separate array for each patch. After merging all patches we can write all the modifications to original data in one step.

Degenerate Surfaces

A robust tessellation algorithm must keep in mind that not all data are well-behaved and satisfy all necessary assumptions. In this section we will briefly mention some problems we have observed and suggest a fix or a possible workaround.

We have already discussed the problem with rectangular domains in section 2.5. To quickly recap, sometimes it is necessary to create a triangular surface, commonly used in the corners of bevels, chamfers and fillets. Typically, this is done by collapsing a boundary edge into a single point, thus creating an illusion of a triangle. However, the topology is still rectangular and this creates a problem when computing a normal vector at the degenerate edge. Since the difference between any two points on that edge is a zero vector, one of the tangent vectors will also be zero. To combat this problem, we check for this condition and choose a different pair of isoparametric curves to compute the tangents from if needed.

In the section 2.6, we have mentioned ordering of the trimming curves. Each simple closed curve is often composed of several smaller curves connected in order. On rare occasions, this order would be completely wrong, breaking the closed curve assumption. Similarly, the assumption of a simple curve, i.e non-intersecting curve, would occasionally not be met either. This problem is difficult, if not impossible, to solve and so we elected to choose the strategy: garbage in, garbage out.

5.2 Software Details

Hot Reload

In our own implementation used for testing new ideas we quickly observed a bottleneck in productivity coming from the long edit-compile-run cycle of typical standalone C++ programs. Making a small tweak in code and judging the new results required to compile the code, reload the program, import the same data, run the algorithm and observe the results. Although compiling even moderately sized C++ code is slow, the biggest time consumer was manual rerunning and waiting for the import of large data. To solve this problem, we developed a *hot reload* feature that dramatically reduced the edit-compile-run latency. It works by separating the *frontend* of our application responsible for opening a window and handling user inputs, from *backend*, which performs imports and runs computations. The backend is compiled into a `.dll` and is loaded by the hosting frontend window application, which sets up a filesystem watcher, that keeps track of any changes in the executable folder. Once a new `.dll` is placed into that folder, the watcher runs a callback that ultimately unloads the old and loads in the new `.dll`. A large memory chunk is allocated by the hosting window and is passed as a pointer to the backend. Any data that is supposed to be persistent is placed in this block. This way, we only need to import a large CAD file once and place this data into the persistent memory block.

A problem we run into with this approach is that `msvc`[1] also generates a `.pdb` file used for debugging that is kept locked by Visual Studio even after unloading the `.dll`, preventing deletion. To circumvent this issue, we generate a unique file name for each new `.dll` and `.pdb` files and occasionally clean up the folder after some time.

Floating point representation

There are several factors to consider when choosing between IEEE 754 32-bit `float` and 64-bit `double` types. One has to balance accuracy and precision with performance and memory requirements to suit the application specifications.

It can be difficult to judge performance differences on modern CPUs and GPUs. Therefore, we conducted a series of benchmarks comparing accuracy, performance and memory characteristics of `float` and `double` data types on common NURBS operations. The result were different for each computation model and each machine, and we suggest to do your own experiments if performance matters. We have observed that most of the CAD systems use `double` as their default type and as a result, exported data files, sometimes only implicitly, depend on this fact. Nevertheless, using `float` type might still be a viable option for certain scenarios. Working with many different data files showed it is mostly ok to use for visualization, however, we encountered some occasional problems:

- Control points are specified too far from the origin, i.e. large values, resulting in loss of accuracy. This problem is hard to diagnose, since such a surface might collapse to a single point or a line without any obvious errors and will not appear on the screen at all.
- Control points that are too close together will be parsed as the same `float` value, despite having distinct `double` representations. This might be fine for evaluating points, however, problems arise with tangent and normal vector computations, since the difference between these two points will be zero, producing incorrect results.

C++20 Modules

Right around the same time we started implementing our software provided as attachment for this thesis, first experimental versions of C++20 modules started to appear. We quickly jumped on the opportunity to test out and benefit from the new features this addition provides.

[1]Microsoft Visual C++ compiler

The new syntax[2] introduces `import` and `export` keywords, where the former includes declarations and definitions from a specified module into the current translation unit, while the latter is mainly used to mark declarations that will be visible in other translation units. Modules are intended to replace header files and resolve two long-standing issues. First, header files need to be recompiled every[3] time they are included, which slows down compilation, especially for template metaprogramming heavy code, and second, is the unfortunate leaking of macro definitions, infamous example being `#include <windows.h>`. Modules promise to eliminate these issues resulting in faster compilation times and easier decomposition of code into separate files.

We observed that the compilation of our prototyping code was relatively fast, although, we do not have a header file version for performance comparison. However, we can express our subjective opinion that we felt like we had more options for separating concerns into different files, after some time familiarizing with modules.

5.3 Data formats

Considering large economic worth of the CAD/CAM industry, it is not surprising to see fierce competition in the market with every company selling proprietary hardware and software. This leads to everyone and their grandma developing new data formats, despite some effort to standardize. Furthermore, libraries to work with these formats are slow and archaic[4], and even the paid ones are exhibiting ignorance or incompetence. On the other hand, it is very difficult, if not impossible, to design a format suitable for every part of the industry, ranging from 2D electrical designs and 3D component packages, through railway construction and car design, all the way to huge turbines and tanker propellers. Here we provide a list of data formats that we encountered the most.

IGES (.igs, .iges)

The oldest format in this list, first published in 1980 with the latest version 5.3 dating to 1996. Despite its age it is still widely used and supported by most if not all CAD software, partly because of its simplicity. The file consists of lines, 80 ASCII character long, with the last 8 columns on every line specifying the section and line number, making it easily human-readable. There are 5 sections in total. The first two, Start and Global, provide information about the file. Followed by the Directory Entry and Parameter Data sections, where the actual CAD data is stored. Lastly, the Terminate section provides a primitive sum check. The data is divided into simple logical and geometric entities, e.g. Line, CurveOnSurface or TransformationMatrix, that are referencing each other to construct more complex objects.

[2]`https://en.cppreference.com/w/cpp/language/modules`

[3]Precompiled headers can be used to solve this problem, however, every compiler has its own implementation and requirements.

[4]Some libraries we encountered do not support multithreading and any attempt at parallelism causes dismay.

STEP (.stp, .step)

STEP started as a successor to IGES trying to solve some of the drawbacks of its predecessor. One of the main advantages of STEP over IGES is the ability to represent solid objects, as opposed to the boundary representation in IGES that can leave gaps in the model due to approximation errors. STEP defines many Application Protocols(AP) having a wide range of functionalities to suit the needs of the whole CAD industry. Each AP has an associated schema written in EXPRESS data modeling language. This makes it possible to create a tool[5] that reads a schema and generates source code for working with the data. Some industry domains have settled on two most common APs in CAD, AP214 used in automotive and AP203 in aerospace industry. Lately, there has been an incentive to merge some APs into one.

`https://www.iso.org/standard/63141.html`

JT (.jt)

JT is an open binary format for CAD data exchange. Models are described using Logical Scene Graph(LSG) containing shapes, components and metadata in hierarchical structure, forming a directed acyclic graph. JT extensively employs compression on multiple levels, and combined with its binary form leads to remarkably small file sizes compared to text files, such as IGES.

Link to JT specification

CATIA (.CATPart, .CATProduct)

Proprietary binary format. It is commonly separated into a product file(.CATProduct) functioning as a root, and several part(.CATPart) files. Third party CAD software applications supporting this format generally require that the user has Catia with the appropriate license installed.

STL (.stl)

Undoubtedly the most simple file format on this list, nowadays utilized frequently in consumer 3D printing market. Each object is a polygonal surface mesh described by a sequence triangles consisting of 3 vertices and 1 normal vector without any additional structure. File can be specified either in an ASCII text form or in a binary form to reduce file size. Tessellation is required when exporting freeform surfaces into this format.

Wavefront (.obj)

Relatively simple text format for specifying geometry of one or more objects used mainly in computer graphics. Stores a list of vertices and optionally a list of normals and texture coordinates. Topology information is stored as a list of faces, where each face is a polygon(usually triangle or quad) specified by indices from vertex data segment. This format can also provide simple material information stored in an accompanying MTL file. Additionally, Wavefront OBJ file format supports freeform structures, including Bézier, B-spline and NURBS

[5]`https://github.com/stepcode/stepcode`

curves/surfaces with corresponding connectivity details. Note that this format is most commonly used for polygonal meshes, whereas freeform surfaces are rarely observed.

`http://fegemo.github.io/cefet-cg/attachments/obj-spec.pdf`

openNURBS (.3dm)

openNURBS is an open source toolkit for reading and writing .3dm files native to Rhino 3D, a commercial CAD software. Making the toolkit open source is meant to make it easier to transfer CAD geometry data between different pieces of software. To directly quote the authors: ”... 3D market is stifled because of the inability to reliably transfer 3D geometry between applications.”.

`https://github.com/mcneel/opennurbs`

Conclusion

In this thesis, we presented fundamental definitions of approximation curves and surfaces necessary for understanding and working with CAD/CAM data. We carefully curated a collection of basic algorithms to serve as an easily digestible reference for future implementations of CAD related software. We also discussed implementation challenges in the industry and provided references for more details. The tessellation algorithm presented in chapter 4 was successfully implemented in a commercial application, being used in production on real data.

Bibliography

[1] Ferenc Kahlesz, Ákos Balázs, and Reinhard Klein. Multiresolution rendering by sewing trimmed nurbs surfaces. In *Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications*, SMA '02, page 281–288, New York, NY, USA, 2002. Association for Computing Machinery.

[2] Ákos Balázs, Michael Guthe, and R. Klein. Efficient trimmed nurbs tessellation. In *WSCG*, 2004.

[3] Ondřej Staňkovič. Triangulace ořezaných nurbs ploch. *Vysoká škola báňská - Technická univerzita Ostrava*, 2012.

[4] Les Piegl and Wayne Tiller. *The NURBS Book (2nd Ed.)*. Springer-Verlag, Berlin, Heidelberg, 1997.

[5] Dr. C.-K. Shene. CS3621 Introduction to Computing with Geometry Notes, 1997.

[6] Carl d. Boor. *A Practical Guide to Splines.* Springer Verlag, New York, 1978.

[7] Manfredo P. do Carmo. *Differential geometry of curves and surfaces.* Prentice Hall, 1976.

[8] Gerald Farin. Triangular bernstein-bézier patches. *Computer Aided Geometric Design*, 3(2):83–127, 1986.

[9] Wolfgang Dahmen, Charles A. Micchelli, and Hans-Peter Seidel. Blossoming begets b-spline bases built better by b-patches. *Mathematics of Computation*, 59(199):97–115, 1992.

[10] Hong Qin and Demetri Terzopoulos. Triangular nurbs and their dynamic generalizations. *Computer Aided Geometric Design*, 14(4):325–347, 1997.

[11] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method.* Dover Books on Mathematics Series. Dover Publications, Incorporated, 2012.

[12] Rohan Sawhney and Keenan Crane. Monte carlo geometry processing: A grid-free approach to pde-based methods on volumetric domains. *ACM Trans. Graph.*, 39(4), 2020.

[13] Daniel L. Toth. On ray tracing parametric surfaces. *SIGGRAPH Comput. Graph.*, 19(3):171–179, jul 1985.

[14] Oliver Abert, Markus Geimer, and Stefan Muller. Direct and fast ray tracing of nurbs surfaces. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 161–168, 2006.

[15] Daniel Filip, Robert Magedson, and Robert Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3(4):295–311, 1986.

[16] X. Sheng and B.E. Hirsch. Triangulation of trimmed surfaces in parametric space. *Computer-Aided Design*, 24(8):437–444, 1992.

[17] Luca Pagani and Paul J. Scott. Curvature based sampling of curves and surfaces. *Computer Aided Geometric Design*, 59:32–48, 2018.

[18] Charles L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3(4):365–372, 1972.

[19] R. Aubry, B.K. Karamete, E.L. Mestreau, and S. Dey. A three-dimensional parametric mesher with surface boundary-layer capability. *Journal of Computational Physics*, 270:161–181, 2014.

[20] R. Aubry, S. Dey, E.L. Mestreau, B.K. Karamete, and D. Gayman. A robust conforming nurbs tessellation for industrial applications based on a mesh generation approach. *Computer-Aided Design*, 63:26–38, 2015.

[21] Marc Vigo Anglada and Núria Pla Garcia. Computing directional constrained delaunay triangulations. *Comput. Graph.*, 24:181–190, 2000.

[22] Jianwei Guo, Fan Ding, Xiaohong Jia, and Dong-Ming Yan. Automatic and high-quality surface mesh generation for cad models. *Computer-Aided Design*, 109:49–59, 2019.

Appendix A

B-spline examples

B-spline basis functions example

Let $\{0, 1, 2, 3, 4, 5\}$ be a knot vector. Corresponding B-spline basis functions are manually computed below.

$$N_{0,0}(t) = \begin{cases} 1 & 0 \leq t < 1 \\ 0 & \text{otherwise} \end{cases} \quad \dots \quad N_{4,0}(t) = \begin{cases} 1 & 4 \leq t < 5 \\ 0 & \text{otherwise} \end{cases}$$

$$N_{0,1}(t) = \frac{t-0}{1-0} N_{0,0}(t) + \frac{2-t}{2-1} N_{1,0}(t) = \begin{cases} t & 0 \leq t < 1 \\ 2-t & 1 \leq t < 2 \end{cases}$$

$$N_{1,1}(t) = \frac{t-1}{2-1} N_{1,0}(t) + \frac{3-t}{3-2} N_{2,0}(t) = \begin{cases} t-1 & 1 \leq t < 2 \\ 3-t & 2 \leq t < 3 \end{cases}$$

$$N_{2,1}(t) = \frac{t-2}{3-2} N_{2,0}(t) + \frac{4-t}{4-3} N_{3,0}(t) = \begin{cases} t-2 & 2 \leq t < 3 \\ 4-t & 3 \leq t < 4 \end{cases}$$

$$N_{3,1}(t) = \frac{t-3}{4-3} N_{3,0}(t) + \frac{5-t}{5-4} N_{4,0}(t) = \begin{cases} t-3 & 3 \leq t < 4 \\ 5-t & 4 \leq t < 5 \end{cases}$$

$$N_{0,2}(t) = \frac{t-0}{2-0} N_{0,1}(t) + \frac{3-t}{3-1} N_{1,1}(t) = \begin{cases} \frac{1}{2}t^2 & 0 \leq t < 1 \\ \frac{1}{2}(t-0)(2-t) + \frac{1}{2}(3-t)(t-1) & 1 \leq t < 2 \\ \frac{1}{2}(3-t)^2 & 2 \leq t < 3 \end{cases}$$

$$N_{1,2}(t) = \frac{t-1}{3-1} N_{1,1}(t) + \frac{4-t}{4-2} N_{2,1}(t) = \begin{cases} \frac{1}{2}(t-1)^2 & 1 \leq t < 2 \\ \frac{1}{2}(t-1)(3-t) + \frac{1}{2}(4-t)(t-2) & 2 \leq t < 3 \\ \frac{1}{2}(4-t)^2 & 3 \leq t < 4 \end{cases}$$

$$N_{2,2}(t) = \frac{t-2}{4-2} N_{2,1}(t) + \frac{5-t}{5-3} N_{3,1}(t) = \begin{cases} \frac{1}{2}(t-2)^2 & 2 \leq t < 3 \\ \frac{1}{2}(t-2)(4-t) + \frac{1}{2}(5-t)(t-3) & 3 \leq t < 4 \\ \frac{1}{2}(5-t)^2 & 4 \leq t < 5 \end{cases}$$

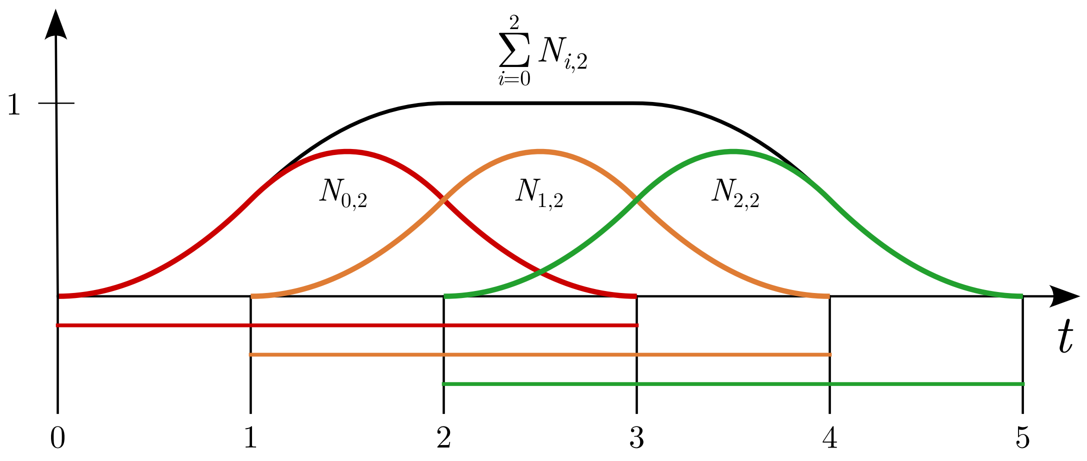


Figure A.1: B-spline basis functions of degree 2 on the open knot vector $\{0, 1, 2, 3, 4, 5\}$
and their sum.

Derivative of a B-spline curve

Let $\boldsymbol{C}(t)$ be a B-spline curve of degree p with $n+1$ control points $\mathbf{P}_0 \ldots, \mathbf{P}_n$ and $m+1$ **clamped** knot values $\{t_0, \ldots, t_m\}$. Then the derivative

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{C}(t) = \boldsymbol{C}'(t) &= \sum_{i=0}^{n} \frac{\mathrm{d}}{\mathrm{d}t} N_{i,p}(t)\,\mathbf{P}_i \\
&= \sum_{i=0}^{n} p\left(\frac{N_{i,p-1}(t)}{t_{i+p}-t_i} - \frac{N_{i+1,p-1}(t)}{t_{i+p+1}-t_{i+1}}\right)\mathbf{P}_i \\
&= \sum_{i=-1}^{n-1} p\,\frac{N_{i+1,p-1}(t)}{t_{i+p+1}-t_{i+1}}\,\mathbf{P}_{i+1} - \sum_{i=0}^{n} p\,\frac{N_{i+1,p-1}(t)}{t_{i+p+1}-t_{i+1}}\,\mathbf{P}_i \\
&= \underbrace{p\,\frac{N_{0,p-1}(t)}{t_p - t_0}\,\mathbf{P}_0}_{=0 \text{ on clamped knots}} + \sum_{i=0}^{n-1} p\,\frac{N_{i+1,p-1}(t)}{t_{i+p+1}-t_{i+1}}\left(\mathbf{P}_{i+1}-\mathbf{P}_i\right) - \underbrace{p\,\frac{N_{n+1,p-1}(t)}{t_{n+p+1}-t_{n+1}}\,\mathbf{P}_n}_{=0 \text{ on clamped knots}} \\
&= \sum_{i=0}^{n-1} p\,\frac{N_{i+1,p-1}(t)}{t_{i+p+1}-t_{i+1}}\left(\mathbf{P}_{i+1}-\mathbf{P}_i\right) = \sum_{i=0}^{n-1} N_{i+1,p-1}(t)\,\mathbf{Q}_i
\end{aligned}
$$

Removing t_0 and t_m from the knot vector yields a B-spline curve of degree $p-1$ with n control points $\mathbf{Q}_0, \ldots, \mathbf{Q}_{n-1}$ and $m-1$ clamped knot values $\{t_1, \ldots, t_{m-1}\}$

$$\boldsymbol{C}'(t) = \sum_{i=0}^{n-1} N_{i,p-1}(t)\,\mathbf{Q}_i$$

Appendix B

More results

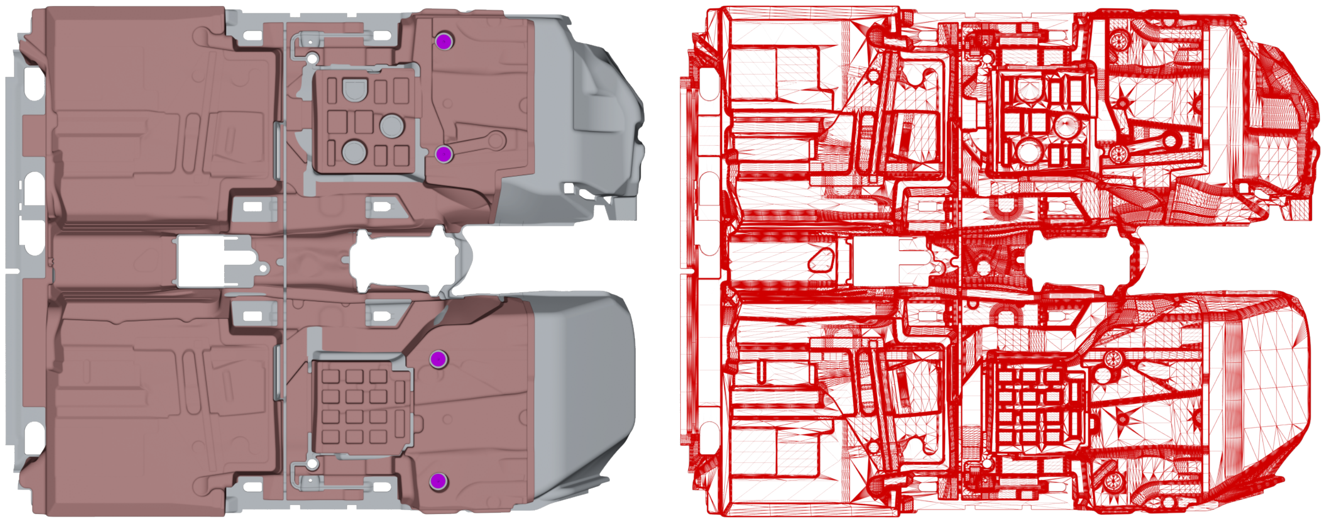


Figure B.1: Floor segment of a car; left: diffuse color, right: tessellated wireframe.

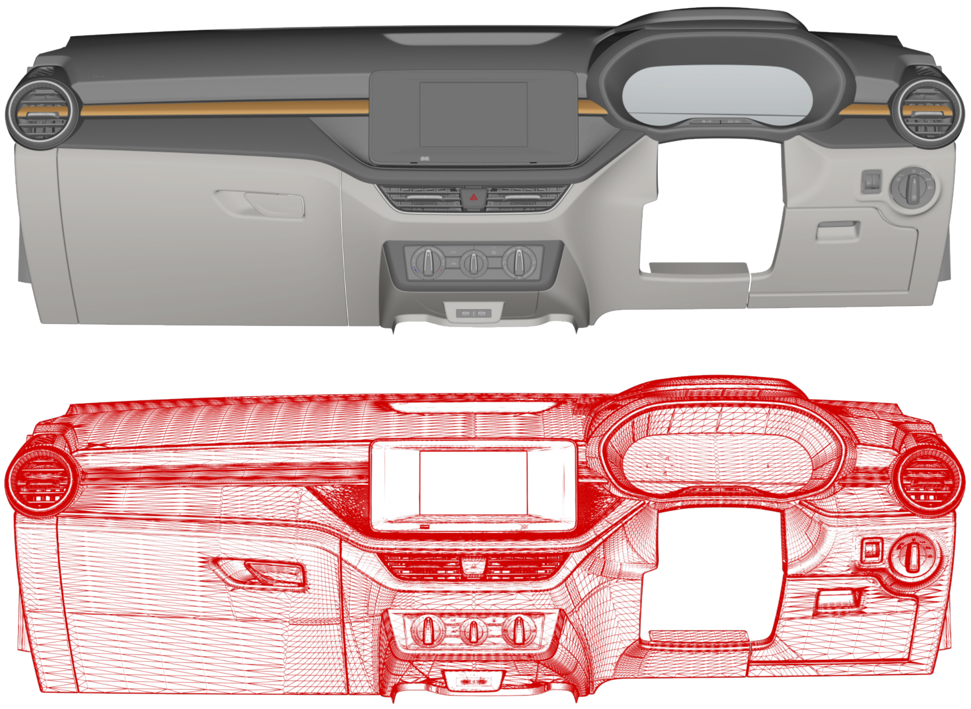


Figure B.2: Front panel design; top: diffuse color, bottom: tessellated wireframe.