Implementierung parametrischer Kurven für CGAL

Diplomarbeit von Ekaterina Langer
Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin

Betreuer: Prof. Dr. Helmut Alt & Dr. Géraldine Morin

August 2002

Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde eine Datenstruktur zur Darstellung von ebenen parametrischen Kurven entworfen und in C++ implementiert.
Die parametrische Kurve ist eine Abbildung $f : [a, b] \to \mathbb{R}^2$, die einem Parameterwert t den entsprechenden Punkt in der 2D Ebene zuordnet. Die Kurve wird durch die Angabe der Funktionen $x(t)$ und $y(t)$ und die Parameterintervallgrenzen $[a, b]$ definiert. Es wurden zwei heuristische Algorithmen zur stückweise linearen Approximation mit adaptivem Schritt entworfen. Anhand von dieser Approximation und einer gegebenen Toleranz ϵ unterstützt die implementierte Datenstruktur effizient die Operationen Zeichnen der Kurve, Bestimmen der Schnittpunkte zweier Kurven, Auswerten der r-ten Kurvenableitung und Bestimmen, ob ein Punkt auf der Kurve liegt.
Bei der so definierten allgemeinen parametrischen Kurve geben die Approximationsmethoden keine Garantie für die maximale Abweichung der Kurve von ihrem stückweise linearen Approximanten. Aus diesem Grund wurden Bézier-Kurven und B-Spline-Kurven als spezielle Arten parametrischer Kurven implementiert. Für diese Kurven sind effiziente Approximationsmethoden bekannt, die auf wiederholten Verfeinerungen ihrer Kontrollpolygone (Subdivision) basieren und sicherstellen, dass die Kurve in der ϵ-Umgebung ihres stückweise linearen Approximanten liegt. Somit ist die Korrektheit der auf Approximationsbasis arbeitenden Algorithmen garantiert.
Bei der Implementierung der Datenstrukturen wurden die Konzepte der generischen Programmierung und die Designziele der C++ Programbibliothek CGAL berücksichtigt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Kapitel 1

Einleitung

1.1 Hintergrund und Motivation

Kurven können sowohl als mathematische Objekte als auch als grundlegende Modellierungselemente in der grafischen Datenverarbeitung betrachtet werden. Parametrische Kurven, definiert als eine stetige Abbildung $\mathbb{R} \to \mathbb{R}^n$, sind eine mögliche Repräsentation von Kurven in der Mathematik. Die Theorie der parametrischen Kurven ist ein gut erforschtes Teilgebiet der Differenzialgeometrie. Untersuchungsgegenstand sind die verschiedenen zulässigen Parameterdarstellungen einer Kurve, die Ermittlung der Krümmung, der Windung und des begleitenden Dreibeins und die Frage, wie diese Parameter die geometrischen Eigenschaften der Kurve widerspiegeln.
Mit der Entwicklung von Systemen zur computergestützten Konstruktion (Computer Aided Design - CAD) entstand die Notwendigkeit einer für den Computer geeigneten Beschreibung von freien Formen. Als Standard hat sich die Parameterdarstellung von Kurven und Flächen durchgesetzt, deren Vorteile sind u.a. die Unabhängigkeit von einem Koordinatensystem, die Fähigkeit beliebige (auch geschlossene und sich selbst schneidende) Formen darzustellen, die leichte Erweiterbarkeit auf höhere Dimmensionen und die Invarianz bezüglich Rotation. Das Entdecken und Erforschen von parametrischen Kurven und Flächen als Repräsentation von freien Formen für das rechnergestützte Konstruieren bilden die Grundlage des rechnergestützten geometrischen Modellierens (Computer-Aided Geometric Design - CAGD). Die wichtigsten Meilensteine der CAGD-Entwicklung sind die Erforschung der Bézier-Kurven von P. Bézier und P. de Casteljau und die Weiterentwicklung der B-Spline-Theorie von C. de Boor und W. Böhm in den siebziger Jahren. Das Ziel der CAGD-Systeme ist ein intuitiver Entwurf und einfache Manipulation von komplexen geometrischen Formen, die auf einer ge-

nauen mathematischen Repräsentation basieren. Um dies zu erreichen, werden geeignete Datenstrukturen zur Darstellung der geometrischen Formen im Rechner, sowie effiziente Algorithmen zur deren Manipulation, benötigt. Eine ähnliche Aufgabe hat diese Diplomarbeit. Es soll eine geeignete Datestruktur für eine durch die Funktionen $x(t)$ und $y(t)$ definierte ebene parametrische Kurve entworfen werden. Diese Struktur soll die folgenden Operationen effizient unterstützen:

- Auswerten der Kurve für einen gegebenen Parameterwert t,
- Bestimmen, ob ein Punkt auf der Kurve liegt,
- Bestimmen der Schnittpunkte mit einer anderen Kurve,
- Auswerten der r-ten Ableitung und
- Zeichnen der Kurve.

Die Implementierung erfolgt in der weit verbreiteten und effizienten Programmiersprache C++, wobei die Konzepte des STL-Standards und die Designziele der Bibliothek CGAL berücksichtigt werden.
CGAL (Computational Geometry Algorithms Library) ist eine C++ Programmbibliothek, die an einigen europäischen und israelischen Universitäten und Instituten, unter anderem an der Freien Universität Berlin, entwickelt wurde. Ausführliche Informationen über dieses Projekt können auf der Internetseite von CGAL `http://www.cgal.org` gefunden werden. Ziel von CGAL ist es, die wichtigsten und effizientesten Lösungen von grundlegenden Problemen der algorithmischen Geometrie (Computational Geometry) in Form einer C++ Bibliothek zur Verfügung zu stellen. Die Elemente der Bibliothek sind in drei Gruppen zusammengefasst. Der Kern (Kernel) besteht aus primitiven Objekten wie z.B. Punkte, Vektoren oder Strecken und aus Prädikatsmethoden, die auf diesen Objekten operieren, z.B. Schnittpunktermittlung. In der Basisbibliothek (Basic Library) sind grundlegende geometrische Datenstrukturen und Algorithmen, wie z.B. konvexen Hülle und Triangulierung, implementiert. Der dritte Teil der Bibliothek ist für die Ein- und Ausgabeoperationen und die Visualisierung zuständig. Die wichtigsten Designziele von CGAL sind Korrektheit, Robustheit, Flexibilität, Effizienz und unkomplizierte Anwendung (easy of use). Unter Korrektheit versteht man, dass die Komponenten der Bibliothek entsprechend ihrer Spezifikation funktionieren. Diese Forderung impliziert nicht Genauigkeit. Zum Beispiel haben viele Probleme der algorithmischen Geometrie keine genaue und zugleich effiziente Lösung. In solchen Fällen werden Approximationsalgorithmen eingesetzt. Die Genauigkeit ihrer Lösungen ist bis zu einer gewissen

Toleranz gewährleistet, die in der Spezifikation des Algorithmus angegeben wird. Auf diese Weise wird eine korrekte Beschreibung des Algorithmus erreicht.Um die Zuverlässigkeit der implementierten Algorithmen zu gewährleisten, deren Entscheidungen oft geometrische Prädikate zu Grunde liegen, wird eine genaue Arithmetik mit reellen verlangt. Leider erfüllt die hardware-unterstützte Arithmetik nicht immer diese Forderung. Deswegen werden bei CGAL Objekte, Algorithmen und Datenstrukturen mit einem Zahlentyp parametrisiert, dessen Arithmetik den Genauigkeitsanforderungen der Anwendung genügt. Um dies zu ermöglichen, werden C++-Templates (parametrisierte Datentypen und Funktionen) eingesetzt. Das führt zur Generalisierung der CGAL-Komponenten und erlaubt ihren Einsatz sowohl in effizienten Anwendungen als auch in Anwendungen, bei denen präzise Arithmetik gefordert wird. Eine weitere Maßnahme zur Erhöhung der Flexibilität der Bibliothek ist ihre modulare Struktur. Sie ermöglicht den Einsatz einzelner Module und begrenzt dadurch den Einarbeitungsaufwand auf das notwendige Modul. Die Möglichkeit der Zusammenarbeit mit anderen C++ Bibliotheken und Programmen ist durch das Implementieren der STL-Konzepte gegeben, die ein Standard für alle C++ Plattformen darstellen. Für den Einsatz von geometrischen Algorithmen ist die Effizienz von großer Bedeutung. Um die Wahl des besten Lösungsansatzes zu ermöglichen, werden für manche Algorithmen unterschiedliche Varianten angeboten, dessen Effizienz höher ist, da sie von bestimmten Eigenschaften der Eingabe ausgehen. Für praktische Zwecke können auch die in der O-Notation versteckten Konstanten für den Laufzeitvergleich von zwei Algorithmen von Bedeutung sein. In solchen Fällen bietet CGAL beide Algorithmen an. Die Forderung, die CGAL-Bibliothek unkompliziert und ohne großen Einarbeitungsaufwand einzusetzen steht leider im Gegensatz zu den übrigen Designzielen, wie etwa Flexibilität und Zuverlässigkeit. Um dem Benutzer den Umgang mit der Bibliothek zu erleichtern, wird eine einheitliche Namenskonvention benutzt. Man ist um vollständige, aber minimale Schnittstellen bemüht. Desweiteren werden verstärkt die immer populärer werdenden STL-Konzepte (Container, Iteratoren und Templates) eingesetzt.
Die wichtigsten Designziele von CGAL wurden beim Entwurf und der Implementierung der in dieser Diplomarbeit entwickelten Algorithmen als Richtlinien benutzt.

1.2 Struktur dieser Arbeit

Im Kapitel 2 werden die mathematischen Darstellungsarten von Kurven (explizite, implizite und parametrische Darstellung) vorgestellt, sowie ihre

Vor- und Nachteile für die verschiedenen Anwendungen diskutiert. Desweiteren wird kurz Subdivision als eine aus dem CAGD-Gebiet stammende Repräsentationstechnik für Kurven eingeführt. Zum Schluss wird die anhand der vorgestellten Darstellungsmöglichkeiten entworfene Klassenstruktur erläutert.

Kapitel 3 befasst sich mit allgemeinen parametrischen Kurven, gegeben durch die Funktionen $x(t)$ und $y(t)$. Zunächst werden die Probleme, die mit einer präzisen Implementierung der oben aufgezählten Operationen verbunden sind, erläutert und zu ihrer Lösung die Anwendung von Approximationsmethoden vorgeschlagen. Dann werden zwei heuristische Ansätze zur Konstruktion eines stückweise linearen Approximanten der Kurve mittels linearer Interpolation vorgestellt und es wird auf das Problem, eine obere Schranke der Abweichung des Approximanten von der Kurve zu finden, eingegangen. Anhand der Approximation wird ein Algorithmus zur diskreten Ermittlung der Ableitung einer parametrischen Kurve entworfen. Desweiteren werden ein Schnittalgorithmus und ein Algorithmus zur Entscheidung, ob ein Punkt auf der Kurve liegt, vorgestellt. Beide Algorithmen arbeiten mit dem stückweise linearen Approximanten, statt mit der Kurve selbst. Zum Schluss werden die Implementierungsaspekte der entworfenen Datenstrukturen für die allgemeine parametrische Kurve und ihren stückweise linearen Approximanten erläutert. Der bei den vorgestellten Approximationsmethoden bestehende Mangel an Garantie bezüglich der Anforderung, dass die Abweichung des Approximanten von der Kurve in vorgegebenen Grenzen bleibt, ist der Grund nach anderen Approximationstechniken zu suchen und weitere Kurvenarten - die Bézier- und B-Spline-Kurven - zu behandeln. Diese polynomielle parametrische Kurven besitzen bestimmte geometrische Eigenschaften, die es ermöglichen, mittels einer Unterteilungstechnik (Subdivision) eine stückweise lineare Approximation zur vorgegebenen Toleranz zu erzeugen. Auf diese Weise wird garantiert, dass die vorgestellten Approximationsalgorithmen mit der gewünschten Präzision arbeiten.

Kapitel 4 behandelt Bézier-Kurven. Zunächst wird die Definition einer Bézier-Kurve in Bernsteinpolynombasis eingeführt und die wichtigsten geometrischen Eigenschaften der Bézier-Kurven erwähnt. Zur Auswertung der Bézier-Kurve am gegebenen Parameterwert t werden die Algorithmen von de Casteljau und Horner vorgeschlagen und deren Laufzeiten verglichen. Desweiteren wird gezeigt, wie die Subdivision-Methode zur Erstellung eines stückweise linearen Approximanten eingesetzt wird, der die Bézier-Kurve innerhalb einer gegebenen Toleranz annähert. Es wird gezeigt, dass die r-te Ableitung einer Bézier-Kurve wiederum eine Bézier-Kurve von Grad $n-r$

ist und zu ihrer Auswertung der de Casteljau-Algorithmus eingesetzt werden kann. Zum Schluss wird die implementierte Datenstruktur zur Darstellung einer Bézier-Kurve erläutert.

Kapitel 5 befasst sich mit B-Spline-Kurven. Als erstes werden die Vorteile der B-Spline-Technik gegenüber der Bézier-Technik erläutert. Die B-Spline-Kurven werden mittels B-Spline-Basisfunktionen definiert und deren geometrischen Eigenschaften diskutiert. Als Auswertungsalgorithmus wird der de Boor-Agorithmus implementiert. Desweiteren wird auf die Approximationsmethoden Einfügen von Knoten und Subdivision eingegangen und die effizientere Subdivision-Technik zur Erstellung eines stückweise linearen Approximanten der B-Spline-Kurve eingesetzt. Zur Auswertung der Ableitung einer B-Spline-Kurve wird der de Boor-Algorithmus verwendet. Das Kapitel endet mit einer Übersicht der entworfenen Datenstruktur zur Darstellung von B-Spline-Kurven.

Kapitel 2

Grundlagen

2.1 Darstellungsarten von Kurven

Eine *ebene Kurve* ist das Bild des reellen Intervalls $[a, b]$ unter einer stetigen, lokal injektiven Abbildung in den Raum $\mathbb{R}^2$

$$f : \mathrm{I} \to \mathbb{R}^2,\ \mathrm{I} \subset \mathbb{R} \tag{2.1}$$

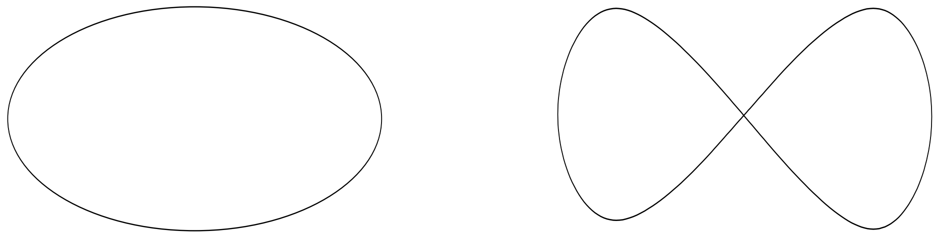
(siehe zum Beispiel [JH89]). Die Abbildung f kann explizit, implizit oder parametrisch angegeben werden. Die *explizite Kurvendarstellung* ist durch die Funktion

$$y = f(x),\ x, y \in \mathbb{R} \tag{2.2}$$

gegeben. Mit dieser Darstellung lassen sich Punkte auf der Kurve sehr einfach bestimmen (Funktion $f(x)$ auswerten). Sie hat aber den gravierenden Nachteil, dass es für einen fixen Wert des Argumentes x genau einen Wert für y gibt. Aus diesem Grund eignet sich die explizite Definition nicht zur Darstellung von geschlossenen und sich selbst schneidenden Kurven wie z.B. Ellipse oder Lemniscate von Bernoulli (siehe Abb. 2.1). Solche Kurven müssen zunächst in Segmente unterteilt werden. Ein anderes Problem ist es, dass explizite Kurven keine vertikalen Tangenten haben können, da die erste Ableitung der Kurve $f'(x)$ parallel zur Ordinatenachse nicht definiert ist. Als weitere Folge der Einwertigkeit ist die explizite Darstellung nicht invariant bezüglich Rotation, einer für die Computer-Graphik unentbehrliche Transformation.

Diese Probleme existieren nicht bei der *impliziten Darstellung*, bei der die Kurve als Lösungsmenge der Gleichung

$$f(x, y) = 0,\ x, y \in \mathbb{R} \tag{2.3}$$



a) Ellipse $\frac{x^2}{a^2} + \frac{y^2}{b^2} - 1 = 0$

b) Lemniscate von Bernouli $(x^2 - y^2)^2 - (x^2 - y^2) = 0$

Abbildung 2.1: Implizite Darstellung einer geschlossenen und einer sich selbstschneidenden Kurve

beschrieben wird. Damit lassen sich komplexe Formen mit relativ niedrigem Polynomgrad darstellen. Implizit definierte Kurven bieten auf natürliche Weise eine Aufteilung der Ebene in eine innere und äußere Region

$$H_i = \{(x, y) \in \mathbb{R}^2 \mid f(x, y) < 0\}$$
$$H_e = \{(x, y) \in \mathbb{R}^2 \mid f(x, y) > 0\}$$

Damit reduziert sich das Problem, ob ein Punkt im Inneren einer von der Kurve begrenzten Region liegt, auf das Bestimmen des Vorzeichens von $f(x, y)$. Ähnlich leicht ist auch die Enscheidung, ob ein Punkt P mit Koordinaten (u, v) auf der Kurve liegt. Dabei werden lediglich die Werte (u, v) in Gleichung 2.3 eingesetzt. Als relativ kompliziert und aufwendig erweist sich dagegen das Berechnen von Punkten auf der Kurve, um zum Beispiel die Kurve zu zeichnen. Für ein gegebenes x kann es mehrere oder gar keine Lösung y geben.
Die parametrische Darstellung vereint die einfache Auswertung eines Punktes auf der Kurve mit der Möglichkeit, beliebige Kurven zu beschreiben. Die *Parameterdarstellung* einer Kurve in der Ebene ist die stetige Abbildung

$$p : [a, b] \to \mathbb{R}^2,\ [a, b] \subset \mathbb{R}. \tag{2.4}$$

Jedem Wert des Parameters $t \in [a, b]$ wird im 2D-Raum ein Punkt

$$p(t) = (x(t), y(t)) \tag{2.5}$$

zugeordnet. Der Parameter t repräsentiert die Position auf der Kurve: für $t = a$ ergibt sich der Angfangspunkt und für $t = b$ der Endpunkt (siehe Abb.2.2). Ein weiterer Vorteil der Parameterdarstellung ist die leichte Erweiterbarkeit auf höheren Dimensionen. Um eine parametrische 2D-Kurve im 3D-Raum zu definieren, wird lediglich eine zusätzliche Funktion $z(t)$

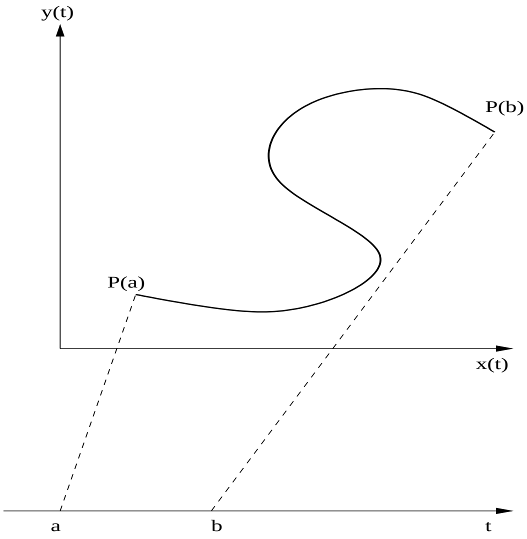


Abbildung 2.2: Parametrische Kurve $p(t),\ t \in [a, b]$

benötigt. Zum Beispiel ist der Einheitskreis in der Ebene parametrisch durch $c(t) = (\cos(t), \sin(t)),\ t \in [0, 2\pi]$ gegeben. In dem $z(t) = t$ gewählt wird, erhält man die parametrische Darstellung der dreidimensionalen Schraubenlinie $h(t) = (\cos(t), \sin(t), t),\ t \in [0, 2\pi]$ (siehe Abb. 2.3).

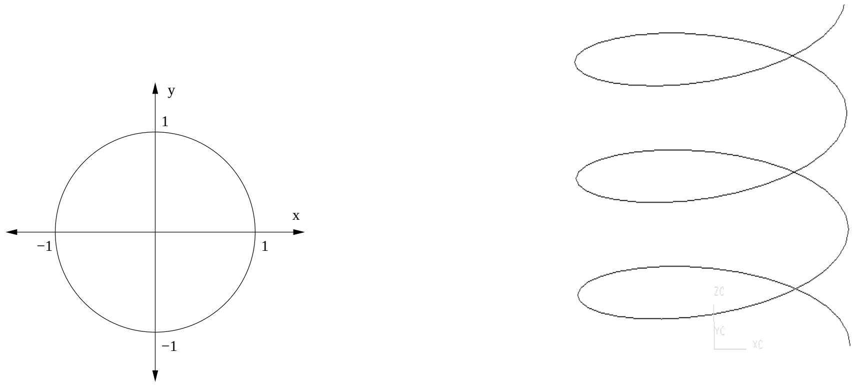
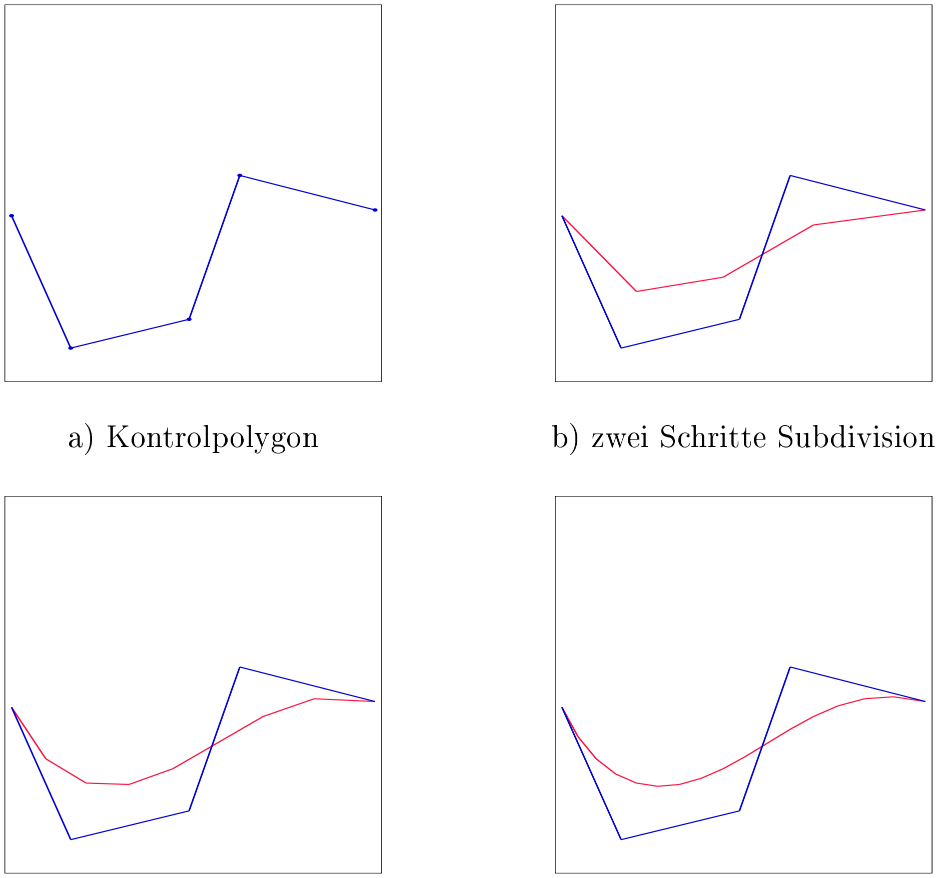


Abbildung 2.3: Die Wahl $z(t) = t$ bildet einen 2D Einheitskreis in eine 3D Schraubenlinie ab

Neben den aufgezählten analytischen Darstellungsarten für Kurven existieren auch prozedurale Methoden. Zum Beispiel werden Fraktale meistens durch rekursive Algorithmen statt mit einer expliziten Formel definiert.
Subdivison ist eine grundlegende CAGD-Technik, bei der glatte Formen durch stückweise lineare Formen dargestellt werden. Die Idee ist die Form der Kurve durch einige *Kontrollpunkte (Stützpunkte)* annähernd anzugeben (wie bei



a) Kontrolpolygon

b) zwei Schritte Subdivision

c) drei Schritte Subdivision

d) vier Schritte Subdivision

Abbildung 2.4: Schrittweise Verfeinerung der Approximation einer Bézier-Kurve vom Grad 4 durch Subdivision

Bézier-Kurven oder B-Splines). Die durch Liniensegmente verbundenen Kontrollpunkte bilden das *Kontrollpolygon*, das eine grobe Approximation der Kurve darstellt und als Ausganspunkt für eine Folge widerholter Unterteilungen dient. Der Grenzwert dieser Folge ist die gewünschte glatte Kurve. Bei jedem Schritt wird das Kontrolpolygon nach einem Subdivisionschema verfeinert und nährt sich der Kurve an (siehe Abb. 2.4). Nach genügend vielen Iterationen konvergiert diese stückweise lineare Approximation zur Kurve. Die Abweichung des Kontrollpolygons von der Kurve schrumpft quadratisch in der Länge h des zu unterteilenden Parameterintervalls $(O(h^2))$. D. h. die Subdivision-Methode hat die beste Konvergenzrate, die von einer Methode zur stückweise linearen Approximation erreicht werden kann. Mit der Subdivisiontechnik lassen sich beliebige Topologien modellieren. Sie bietet eine

einfache Implementierung und effiziente Berechnung.
Da jede der vorgestellten Kurvendarstellungen für bestimmte Anwendungszwecke die am besten geeigenete ist, würde eine Konvertierung zwischen den verschiedenen Darstellungsformen vom großen Vorteil sein. Leider tritt dieser Idealfall relativ selten vor. Es ist nicht immer möglich parametrische oder implizit definierte Kurven in explizite Darstellung umzuwandeln. Dagegen ist die *Implizition*, die Umwandlung einer Parameterdarstellung in implizite Form, immer durchführbar (siehe z.B. [JH89]). Der Konvertierungsprozess ist aber in den meisten Fällen mit großem Rechenaufwand verbunden. Auch die Konvertierung zwischen Subdivision-Kurven und paramettrischen Kurven ist nicht immer möglich (z.B. bei Fraktalen). Für Bézier-Kurven und B-Splines existiert sowohl eine parametrische als auch eine Subdivision-Repräsentation. Das macht diese Kurven besonders wertvoll und ist u.a. einer der Gründe für ihre Implementierung im Rahmen dieser Arbeit.

2.2 Klassenstruktur

Anhand der im Abschnitt 2.1 genannten Darstellungsmöglichkeiten für Kurven habe ich mich für die folgende Klassenstruktur entschieden (siehe Abb. 2.5).
Die abstrakte Klasse `Curve_2` stellt eine allgemeine Kurve in der 2D-Ebene dar, bei der die Repräsentation nocht nicht festgelegt ist. Sie verfügt über eine Zeichenmethode und ein Kriterium zum Bestimmen, ob ein Punkt auf der Kurve liegt. Da in der Klasse `Curve_2` die Darstellungsart der Kurve nicht bekannt ist, sind ihre Methoden abstrakt und werden von den Unterklassen implementiert.
Die Klasse `Implicit_curve_2` stellt eine implizit definierte Kurve dar. Diese Template-Klasse wird mit einem Funktionsobjekt F parametrisiert, das die mit Gleichung (2.3) angegebene implizite Definition implementiert.
`Subdivision_curve_2` ist die Oberklasse für alle Kurven, bei denen die Subdivision-Technik eingesetzt werden kann. Sie speichert die Kontrollpunkte, die die Kurve definieren und verfügt über eine subdivide-Methode, die eine Verfeinerung des Kontrolpolygons vornimmt. Auf diese Klasse wird näher im Abschnitt 4.4 eingegangen.
Die Klasse `Parametric_curve_2` repräsentiert eine ebene parametrische Kurve. Die Funktionen $x(t)$ und $y(t)$, die die Kurve parametrisch nach (2.5) definieren, sowie die Intervallgrenzen a, b für den Parameter t, sind als Datenfelder der Klasse gespeichert. Um die Implementierung allgemein zu halten und dem Benutzer die Möglichkeit zu geben, eigene Typen von Kurven zu definieren, ist die Klasse als Template organisiert. Die Template-Parameter `X` und `Y` geben den Typ der als Funktionsobjekte implementierten Funktionen

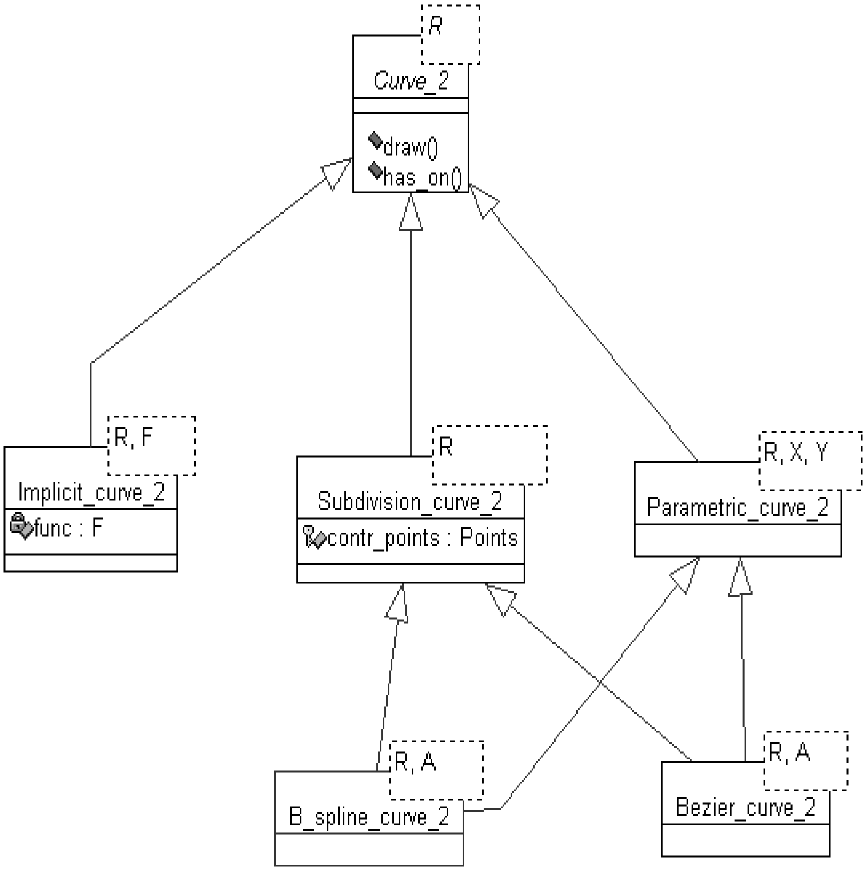


Abbildung 2.5: Klassendiagramm

$x(t)$ bzw. $y(t)$ an. Die Klasse Parametric_curve_2 wird gründlich im Kapitel 3 behandelt.
Die Klassen Bézier_curve_2 und B_spline_curve_2 implementieren die gleichnamigen ebenen Kurven und werden ausführlich in Kapiteln 4 bzw. 5 vorgestellt. B-Spline- und Bézier-Kurven haben eine Parameterdarstellung, lassen sich aber auch mittels Subdivision darstellen. Das führt zu der Vererbungshierarchie in Abb. 2.5, in der Bézier_curve_2 und B_spline_curve_2 jeweils von Parametric_curve_2 und Subdivision_curve_2 erben.
Mit Bezugnahme auf die CGAL-Namenskonventionen für geometrische Objekte enthalten die Klassennamen die Dimension als Suffix (_2 für Kurven in der Ebene). Desweiteren sind alle Klassen mit einem Zahlentyp (auf Abb. 2.5 mit R bezeichnet) parametrisiert. Er gibt den Datentyp an, in dem die Koordinaten gespeichert sind und ist entscheidend für die Genauigkeit und Effiizienz der den Methoden und Prädikaten zugrunde liegende Arithmetik.

Diese Parametrisierung erhöht die Flexibilität der Implementierung und erlaubt dem Benutzer selbst zu entscheiden, welcher Zahlentyp für seine Zwecke am besten geeignet ist. Zum Beispiel gewährt der Datentyp `leda_rational` eine hohe Präzision, wirkt sich aber nachteilig auf die Geschwindigkeit der Operationen aus. Für Anwendungen, bei denen eher die Effizienz entscheidend ist (z.B. Zeichnen von Kurven), eignet sich der C++ Datentyp `double` mit seiner weniger genauen dafür aber hardwareunterstützten und deswegen schnellen Arithmetik besser. Die so entworfene Klassenstruktur lässt sich beliebig um weitere Kurvendarstellungen bzw. Kombinationen der schon existierenden erweitern. Das Abstrahieren der Klasse `Curve_2` als Oberklasse aller Kurvenrepräsentationen erlaubt bei Kurven, für die die Konvertierung zwischen verschiedenen Repräsentationen möglich ist, die jeweils effizienteste Methoden auszuwählen.

Kapitel 3

Parametrische Kurven

Das Ziel dieser Arbeit ist eine geeignete Datenstruktur für parametrische Kurven zu entwerfen, mit der die Operationen:

- Finden eines Punktes auf der Kurve für einen gegebenen Parameterwert t,
- Zeichnen der Kurve,
- Bestimmen, ob ein Punkt auf der Kurve liegt,
- Bestimmen der Schnittpunkte mit einer anderen Kurve und
- Auswerten der r-ten Ableitung

effizient implementiert werden. Gegeben sind das Parameterinterval $[a, b]$ und die Funktionen $x(t)$ und $y(t)$, die nach 2.5 die Kurve definieren. Somit reduziert sich das Problem, einen Punkt auf der Kurve für ein gegebenes t zu finden, auf das Auswerten der Funktionen $x(t), y(t)$. Das Implementieren der restlichen Operationen ist im Allgemeinen schwierig. Man kann aber eine beliebig große Menge von Punkten auf der Kurve berechnen und auf deren Basis einen geeigneten Approximant, der die Kurve in diesen Punkten interpoliert, konstruieren. Die Idee ist es, anstatt mit der Kurve mit ihrem Approximanten und einer gegebenen Toleranz zu arbeiten. Nach [JH89] ist die Aufgabe der klassischen Interpolation eine komplizierte Funktion durch eine einfachere Funktion anzunähern, wobei die beiden Funktionen an vorgegebenen Stellen übereinstimmen. Zu den bekannten Interpolationstechniken zählen lineare Interpolation, Lagrange-Interpolation, Newton-Interpolation, Hermite-Interpolation, Spline-Interpolation u.a. Für die Zwecke dieser Arbeit wurde lineare Interpolation wegen ihrer effizienten Implementierung und Invarianz bezüglich affiner Transformationen gewählt. Ein weiterer Grund für

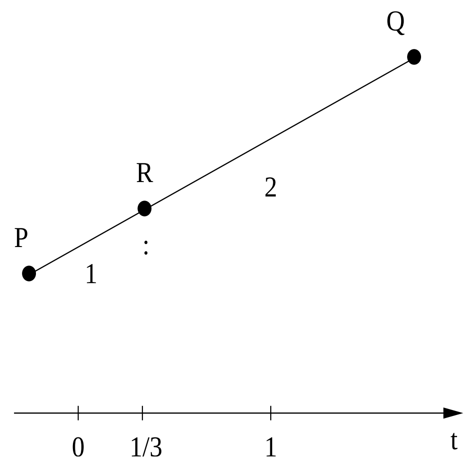


Abbildung 3.1: Lineare Interpolation der Punkte P und Q. Der Punkt $R = \frac{2}{3}P + \frac{1}{3}Q$ teilt die Strecke PQ im Verhältnis 1 : 2

diese Entscheidung ist, dass die Bibliothek CGAL über Datenstrukturen und effiziente Operationen auf lineare Objekte, wie z.B. Linie und Liniensegment, verfügt, die bei der Implementierung eingesetzt werden können (z.B. das Prädikat `has_on(Point pt)` der Klasse `Segment_2`).

3.1 Stückweise lineare Approximation

Das Liniensegment, das die zwei Punkte P und Q interpoliert, ist durch die Gleichung

$$l(t) = (1-t)P + tQ, \ t \in [0,1] \tag{3.1}$$

gegeben (siehe z.B. [Far88]). Für $t = 0$ verläuft $l(t)$ durch den Punkt P und für $t = 1$ durch Q (siehe Abb. 3.1). Da die Summe der Koeffizienten in (3.1) 1 ist, ist lineare Interpolation eine affine Abbildung des reellen Intervalls $[0,1]$ in den affinen Raum. Für drei kollinerare Punkte P, Q und R gilt

$$R = \alpha P + \beta Q, \ \alpha + \beta = 1. \tag{3.2}$$

α und β sind die *barizentrischen Koordinaten* von R bezüglich P und Q. Bei linearer Interpolation sind $\alpha = 1 - t$ und $\beta = t$. Desweiteren teilt der Punkt R die Strecke PQ im gleichen Verhältnis, wie der Parameter t das Intervall $[0,1]$, d. h. $\frac{PR}{RQ} = \frac{t}{1-t}$. Diese Eigenschaft wird in [Far88] benutzt, um lineare Interpolation über einem beliebigen Intervall $[a,b]$ zudefinieren:

$$l(t) = \frac{b-t}{b-a}P + \frac{t-a}{b-a}Q, \ t \in [a,b] \tag{3.3}$$

Seien $P_0, \ldots, P_n$ eine Menge von Punkten, dann wird der *stückweise lineare Interpolant* L dieser Punktemenge als eine Folge gerader Segmente l_i definiert (siehe z.B. [Far88]), die die Punkte P_i, P_{i+1} paarweise verbindet. Sind die Punkte P_i *Stützpunkte* auf der mit Gleichung 2.5 definierten Kurve $p(t)$, d. h. es gilt $P_i = p(t_i)$, dann ist L ein *stückweise linearer Approximant* bezüglich der Kurve $p(t)$. Unter Berücksichtigung von Gleichung 3.3 ist jedes Liniensegment von L durch

$$l_i(t) = \frac{t_{i+1} - t}{t_{i+1} - t_i} P_i + \frac{t - t_i}{t_{i+1} - t_i} P_{i+1},\ t \in [t_i, t_{i+1}] \tag{3.4}$$

gegeben, wobei $P_i = p(t_i)$, $t_i < t_{i+1}$, $t_0 = a$ und $t_n = b$ gelten. Der so konstruierte Approximant stimmt zwar mit der Kurve an den Stützstellen überein, im allgemeinen jedoch nicht überall. Eine Schranke für die Abweichung des Liniensegments l_i von dem interpolierten Kurvensegment zwischen den Stützpunkten P_i, P_{i+1} gibt [Gol]. Danach gilt

$$\max_{i=0\ldots n-1} |p_i(t) - l_i(t)| \leq \frac{(t_{i+1} - t_i)^2}{8} \max |p_i''(t)|,\ t \in [t_i, t_{i+1}] \tag{3.5}$$

vorausgesetzt $p(t)$ ist zwei mal differenzierbar im Intervall $[a, b]$. Zum Zwecke einer besseren Annäherung soll diese Abweichung so klein wie möglich gemacht werden. Dabei ist die Wahl der Parameterwerte t_i an den Stützstellen P_i, d. h. die Parametrisierung entscheidend. In dieser Arbeit werden die folgenden Ansätze implementiert:

Äquidistante (gleichmäßige) Parametrisierung

Bei dieser Methode wird das Parameterintervall $[a, b]$ in $n - 1$ gleich lange Intervalle $[t_i, t_{i+1}]$ unterteilt. Es gilt

$$t_i = a + i\frac{b - a}{n - 1},\ i = 0, 1, \ldots, n - 1 \tag{3.6}$$

In Anlehnung auf die in [GM01] benutzte Notation für gleichmäßige Parametrisierung ist der stückweise lineare Approximant der Kurve mit

$$L_{up} = UP(P, [a, b]),\ P = \{P_0, \ldots, P_n\} \tag{3.7}$$

gegeben. Der Nachteil dieser Vorgehensweise ist, dass sie keine Rücksicht auf die Form der Kurve nimmt. Zum Beispiel werden flache Teile der Kurve mit Stützpunkten versehen, die keine neue Information für die Approximation bringen, dagegen werden starke Schwankungen der Kurve nicht ausreichend angenähert und können sogar gar nicht registriert werden (siehe Abb. 3.2).

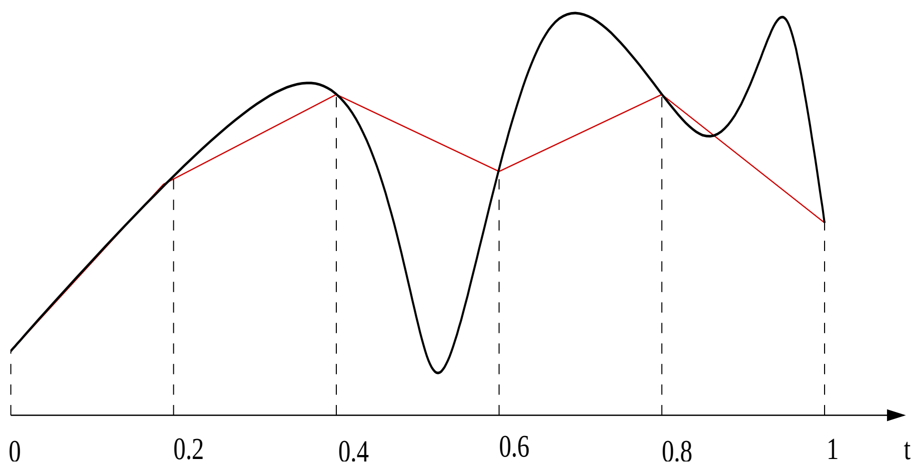


Abbildung 3.2: Gleichmäßige Parametrisierung mit Schritt 0.2

Wegen ihrer einfachen und effizienten Berechnung ist die gleichmäßige Parametrisierung gut geeignet als Initialisierungsparametrisierung, die als Basis für eine im Folgenden erläuterte adaptive Unterteilung des Parameterintervalls dient.

Adaptive Parametrisierung

Bei diesem Ansatz wird das Intervall $[a, b]$ mit einem sich an die Kurvenform anpassenden Schritt s_i unterteilt, d. h.

$$t_0 = a,\ t_n = b,\ t_i = t_{i-1} + s_i,\ i = 1, \ldots, n-1 \tag{3.8}$$

Der dadurch entstehende Approximant stellt eine bessere Annäherung an die Kurve dar (siehe Abb. 3.3). Zur Wahl der Parameterwerte t_i werden hier zwei heuristische Algorithmen vorgestellt. Beide gehen von einer Initialisierungsparametrisierung aus, die schrittweise verfeinert wird, bis eine zufriedenstellende Annäherung erreicht ist. Für eine gegebene Toleranz ϵ wird eine Approximation gesucht, so dass die Kurve $p(t)$ in der *ϵ-Umgebung* ihres stückweise linearen Approximaten liegt, d. h.

$$|p_i(t) - l_i(t)| \leq \epsilon,\ i = 0, 1, \ldots, n-1,\ t \in [t_i, t_{i+1}] \tag{3.9}$$

Der erste Algorithmus geht nach dem Prinzip der Tiefensuche vor. Das Intervall $[t_i, t_{i+1}]$ wird so lange in der Mitte geteilt, bis die Bedingung 3.9 für $t = \frac{t_1 + t_{i+1}}{2}$ erfüllt ist (siehe Alg. 3.1.1) Der Nachteil dieses Algorithmus ist, dass die dadurch erzeugte Parametrisierung das Erfülltsein von Bedingung 3.9 nur für die Mitte des Intervalls $[t_i, t_{i+1}]$ sicherstellt, d. h. es gibt keine Garantie, dass ϵ eine obere Schranke für die Abweichung des linearen Approximanten von der Kurve für alle $t \in [t_i, t_{i+1}]$ ist. Wegen der unkompizierten

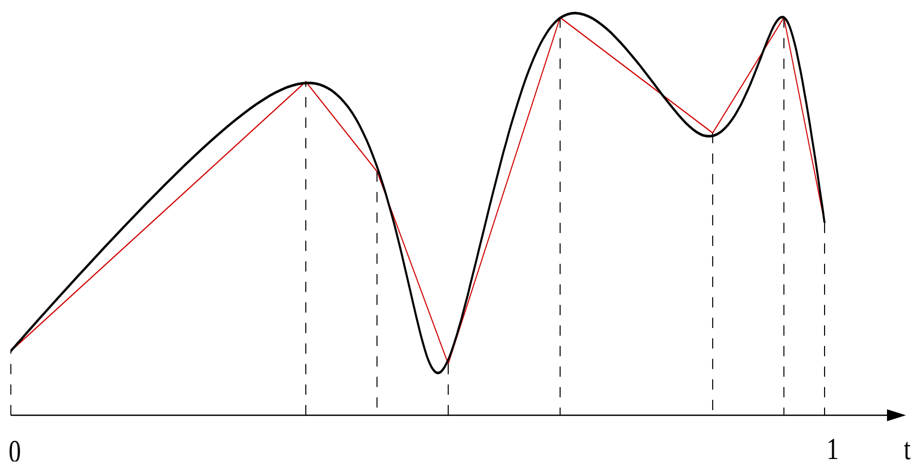


Abbildung 3.3: Adaptive Parametrisierung

test_middle_param(I, ϵ)
Require: I: list of intervals $[t_i, t_{i+1}]$, $\epsilon > 0$
Ensure: $\forall i : |p_i(\frac{t_i+t_{i+1}}{2}) - l_i(\frac{t_i+t_{i+1}}{2})| < \epsilon$
  while I is not empty **do**
    pop the fist interval $[t_i, t_{i+1}]$
    m:=$\frac{t_i+t_{i+1}}{2}$
    if $|p_i(m) - l_i(m)| > \epsilon$ **then**
      subdivide $[t_i, t_{i+1}]$ in $[t_i, m]$ and $[m, t_{i+1}]$
      insert them at the beginning of I
    end if
  end while

Algorithmus 3.1.1: Adaptive Parametrisierung durch Abschätzung der Abweichung in der Intervallmitte

Arithmetik und guter Laufzeit wurde dieser heuristische Ansatz implementiert und zeigt in den Testprogrammen zufriedenstellende Ergebnisse.
Ein anderer Ansatz basiert auf die Abweichungsabschätzung durch die zweite Ableitung der Kurve, wie in Ungleichung 3.5 angegeben. Die Intervallliste wird so oft durchlaufen, bis für alle Intervalle die Bedingung

$$\max(|p''(t_i)|, |p''(t_{i+1})|)\frac{(t_{i+1} - t_i)^2}{8} < \epsilon$$

erfüllt ist. Bei jedem Durchlauf wird ein Intervall $[t_i, t_{i+1}]$, das der Bedingung nicht genügt, geteilt und durch die Intervalle $[t_i, \frac{t_i+t_{i+1}}{2}]$ und $[\frac{t_i+t_{i+1}}{2}, t_{i+1}]$ ersetzt (siehe Alg. 3.1.2). Leider gibt dieser Algorithmus auch keine Garantie für den Wert der maximalen Abweichung des Approximanten von der Kurve.

second_deriv_param(I, ϵ)
Require: I: list of intervals $[t_i, t_{i+1}]$, $\epsilon > 0$, ok=false
Ensure: $\forall i : \max(|p''(t_i)|, |p''(t_{i+1})|)\frac{(t_{i+1}-t_i)^2}{8} < \epsilon$

```
while ! ok do
  ok:=true;
  for all [t_i, t_{i+1}] in I do
    if max(|p''(t_i)|, |p''(t_{i+1})|) (t_{i+1}-t_i)^2/8 > ε then
      m:=(t_i+t_{i+1})/2
      subdivide [t_i, t_{i+1}] in [t_i, m] and [m, t_{i+1}]
      replace [t_i, t_{i+1}] with [t_i, m] and [m, t_{i+1}] in I
      ok:=false
    end if
  end for
  if ! ok then
    calculate the discrete differences p''(t_i)
  end if
end while
```

Algorithmus 3.1.2: Adaptive Parametrisierung mit Abschätzung der 2. Ableitung

Ein Grund dafür ist, dass die Funktion $p''(t)$ nicht bekannt ist und statt dessen mit der diskreten Ableitung der Kurve gearbeitet wird. So wird im Algorithmus $\max|p_i''(t)|$ durch den größeren der beiden zum Intervall gehörenden diskreten Werte $|p''(t_i)|$ und $|p''(t_{i+1}|)$ ersetzt, was aber nicht unbedingt das Maximum der 2. Ableitung in $[t_i, t_{i+1}]$ ist. Im Vergleich zum ersten Ansatz, bei dem für jedes Intervall nur die Mitte m und die Werte $p(m)$ und $l(m)$ berechnet werden, wird hier der Wert der diskreten Ableitung benötigt. Dieser ändert sich aber bei Änderung der Parametrisierung. Um zu vermeiden, dass diese aufwendige Berechnung bei jedem Durchlauf der **for**-Schleife ausgeführt wird, verzichtet man hier auf Tiefensuche. Das hat den Nachteil, dass Intervalle, die die Toleranzbedingung erfüllen immer wieder getestet werden. Die Breitensuche erlaubt aber einmalige Berechnung der diskreten Ableitung pro **while**-Schleifendurchlauf. Auf die Ermittlung der diskreten Ableitung wird in Abschnitt 3.2 näher eingegangen.

3.2 Diskrete Ableitung

Für die mit Gleichung 2.5 definierte parametrische Kurve soll die r-te Ableitung $p^{(r)}$ ermittelt werden. Die Funktion $p^{(r)}(t)$ ist nicht bekannt. Gege-

ben ist der stückweise lineare Approximant L der Kurve $p(t)$ (siehe 3.4). Die Idee ist es, die diskrete Ableitung des stückweise linearen Approximanten zur annähernden Bestimmung von $p^{(r)}(t)$ einzusetzen. Im Folgenden wird die von Morin und Goldman in [GM01] vorgeschlagene Notation der diskreten Ableitung einer gleichmäßig parametrisierten Stützpunktenfolge übernommen und auf adaptive Parametrisierung angepaßt.
Die *diskrete Ableitung* $D^{(1)}$ einer stückweise linearen Funktion mit gleichmäßiger Parametrisierung im Intervall $[a, b]$ (siehe 3.7) ist durch

$$D^{(1)}\mathrm{UP}(P, [a, b]) = \mathrm{UP}\left(\frac{1}{b-a}\Delta_n P, [a, b]\right),\ P = \{P_0, \ldots, P_n\} \qquad (3.10)$$

definiert. Dabei ist Δ_n die normalisierte Differenzmatrix der Größe $n \times (n+1)$, gegeben durch

$$(\Delta_n)_{i,j} = \begin{cases} -n & j = i \\ n & j = i+1 \\ 0 & \text{otherwise} \end{cases}$$

Also ist die diskrete Ableitung des stückweise linearen Approximanten L wiederum eine stückweise lineare Funktion mit Stützpunkten $P_i^{(1)} = \frac{n}{b-a}(P_{i+1} - P_i),\ i = 0, \ldots, n-1$, die gleichmäßig über dem Intervall $[a, b]$ verteilt sind. Da für eine äquidistante Parametrisierung $\frac{b-a}{n} = t_{i+1} - t_i$ gilt, kann man 3.10 so umschreiben:

$$D^{(1)}L = \mathrm{UP}\left(\frac{P_{i+1} - P_i}{t_{i+1} - t_i}, [a, b]\right),\ i = 0, \ldots, n-1 \qquad (3.11)$$

Das Verhältnis $\frac{P_{i+1} - P_i}{t_{i+1} - t_i}$ wird *dividierte Differenz* genannt (siehe zum Beispiel [JH89]). Ableitungen höherer Ordnung können nach [GM01] rekursiv berechnet werden:

$$D^{(r)}L = D^{(1)}D^{(r-1)}\mathrm{UP}(P, [a, b]),\ r = 1, \ldots, n-1 \qquad (3.12)$$

Im Abschnitt 3.1 wurde gezeigt, dass die adaptive Parametrisierung eine bessere Wahl für die Konstruktion eines stückweise linearen Approximanten der Kurve darstellt. Aus diesem Grund wird im Folgenden Gleichung 3.11 für adaptive Parametrisierung angepaßt. Die dividierten Differenzen, die die Stützpunkte $P_i^{(1)}$ von $D^{(1)}L$ darstellen, hängen direkt von den Intervalllängen also nicht von der Art der Parametrisierung ab. Das Problem ist, dass nach jeder Differenzierung ein Stützpunkt weniger zur Verfügung steht. In 3.11 ist das durch die gleichmäßige Verteilung der diskreten Ableitungswerte über das Intervall $[a, b]$ behoben. Bei adaptiver Parametrisierung muss die Wahl der Parameterwerte $t_i^{(1)}$ mit der Länge der Intervalle $[t_i, t_{i+1}]$ abgestimmt werden. Eine einfache Möglichkeit ist die Mitte der Intervalle $[t_i, t_{i+1}]$, d.h. $t_i^{(1)} = \frac{t_{i+1}-t_i}{2}$, zu wählen. Dann fehlen die Ableitungswerte für die Intervallränder $t_0 = a$ und $t_n = b$ ($P_0^{(1)}$ entspricht dem Parameterwert $\frac{t_1-a}{2}$ und $P_{n-1}^{(1)}$ entspricht $\frac{b-t_{n-1}}{2}$) und das Intervall "schrumpft". Um das zu vermeiden, könnte man auch an den Intervallgrenzen die Werte $P_0^{(1)}$ bzw. $P_{n-1}^{(1)}$ annehmen. Das führt aber zu einer unerwünschten Verflachung der Ableitung an den Intervallrändern.

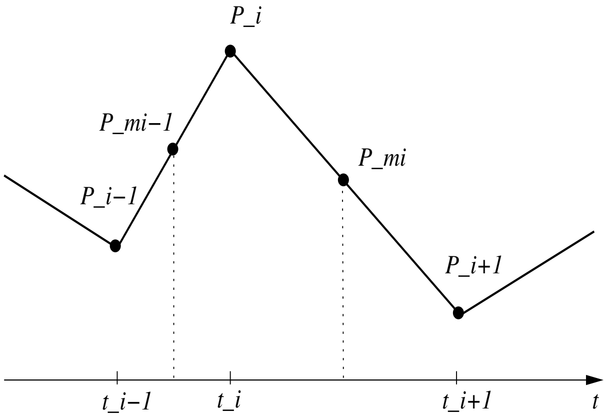


Abbildung 3.4: Ermittlung des diskreten Ableitungswertes $P_i^{(1)}$ als Mittelwert der linken $\frac{2(P_i - P_{m_{i-1}})}{t_i - t_{i-1}}$ und rechten $\frac{2(P_{m_i} - P_i)}{t_{i+1} - t_i}$ dividierten Differenz

In dieser Arbeit wurde eine andere Methode vorgezogen. Für die diskrete Ableitung wird die gleiche Parametrisierung wie bei dem stückweise linearen Approximanten L angenommen. Die diskreten Werte der Ableitung an den Stellen t_i werden als Mittelwert einer linken $D_{l_i} = \frac{2(P_i - P_{m_{i-1}})}{t_i - t_{i-1}}$ und einer rechten $D_{r_i} = \frac{2(P_{m_i} - P_i)}{t_{i+1} - t_i}$ dividierten Differenz ermittelt, wobei $P_{m_i} = \frac{P_i + P_{i+1}}{2}$ ist (siehe Abb. 3.4). An den Intervallenden t_0 und t_n wird die fehlende linke bzw. rechte Differenz als 0 angenommen. Zusammengefasst gilt:

$$P_i^{(1)} = D^{(1)} L(t_i) = \begin{cases} \frac{2(P_{m_0} - P_0)}{t_1 - t_0} & i = 0 \\ \frac{2(P_n - P_{m_{n-1}})}{t_n - t_{n-1}} & i = n \\ \frac{D_{l_i} + D_{r_i}}{2} & 0 < i < n \end{cases} \tag{3.13}$$

Somit wird eine Verteilung der diskreten Ableitungswerte über das ganze Definitionsintervall $[a, b]$ erreicht, die von der Ausgangsparametrisierung t_i

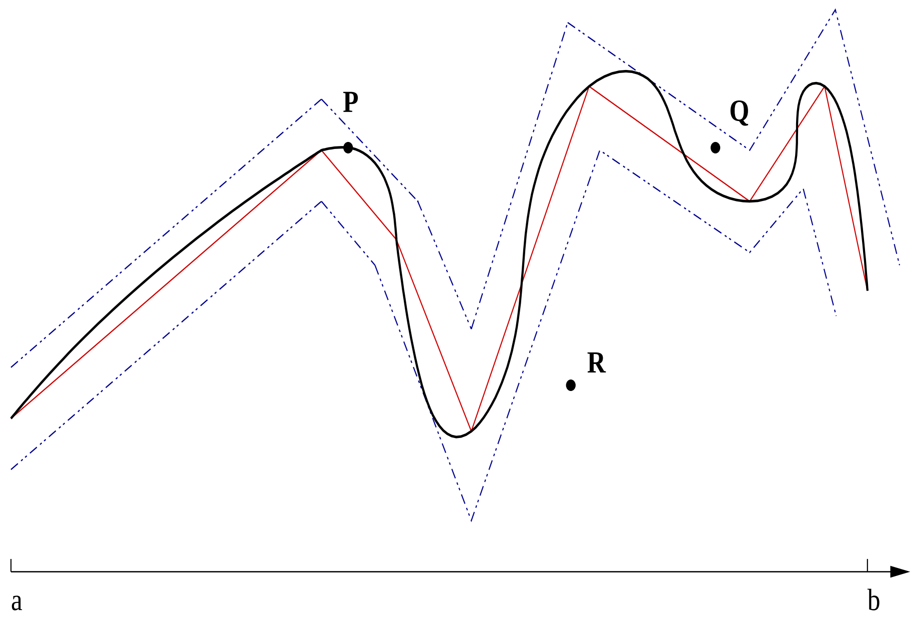


Abbildung 3.5: Parametrische Kurve $p(t)$, die in der ϵ-Umgebung ihres stückweise linearen Approximanten L liegt

abhängt und an den Intervallrändern keine konstante Bereiche erzeugt.

3.3 Algorithmus zur Entscheidung, ob ein Punkt auf der Kurve liegt

Für die mit Gleichung 2.5 definierte parametrische Kurve und einen Punkt Q in der Ebene ist zu entscheiden, ob der Punkt auf der Kurve liegt. Also wird ein $t \in [a, b] : p(t) = Q$ gesucht. Leider ist eine genaue algebraische Lösung in diesem Fall nicht möglich, da die Funktion $p(t)$ nur als eine Black-Box gegeben ist, die zu einem gegebenen Parameterwert einen Punkt auf der Kurve berechnet. Man muss mit einer Näherungsmethode und einen gewissen Fehler arbeiten. In Abschnitt 3.1 wurden zwei Algorithmen vorgestellt, die zu einer gegebenen Toleranz ϵ einen stückweise linearen Approximant der Kurve konstruieren. Im Idealfall sollte die Kurve in der ϵ-Umgebung des Approximanten liegen (siehe Abb. 3.5).

Das Problem, ob der Punkt Q auf der Kurve liegt, wird dann auf die Entscheidung, liegt Q in der ϵ-Umgebung des stückweise lineraren Approximanten der Kurve, zurückgeführt. Für Punkte, die höchstens ϵ vom Approximanten entfernt sind, d.h. $\mathrm{dist}(Q, l_i) <= \epsilon,\ l_i \in L$, wird angenommen,

```
has_on(L, Q, ε)
Require: Q: a point, ε > 0, L: list of line segments l_i : ∀t ∈
  [t_i, t_{i+1}] |l_i(t) − p(t)| <= ε, i = 0 . . . n − 1
Ensure: if true returned ∃i : distance(l_i, Q) <= ε
  num := |L|
  if num=0 or !do_overlap(bbox(Q), bbox(L)) then
    return false
  else if |L| = 1 then
    if squared_distance(l, Q) <= ε^2 then
      return true
    else
      return false
    end if
  else
    m := ⌊num/2⌋
    divide L in L_l := {l_0, . . . , l_m} and L_r := {l_{m+1}, . . . l_{num−1}}
    if has_on(L_l, Q, ε) then
      return true
    else
      return has_on(L_r, Q, ε)
    end if
  end if
```

Algorithmus 3.3.1: Bestimmen, ob ein Punkt auf der Kurve liegt

dass sie auf der Kurve liegen (z.B. die Punkte P und Q auf Abb. 3.5). Bei Punkten ausserhalb der ϵ-Umgebung des Approximanten, d. h. deren Abstand zur Kurve größer 2ϵ ist, fällt die Enscheidung negativ aus (siehe z.B. den Punkt R). Der hier implementierte Algorithmus basiert auf einen *schnellen Ablehnungstest (quick rejection)* durch die Bounding-Boxen des Approximanten L und des Punktes Q. Unter *Bounding-Box* des stückweise linearen Approximanten wird hier das um ϵ erweiterte Min-Max-Rechteck mit linkem unteren Punkt $R_l(min_x - \texttt{epsilon}, min_y - \texttt{epsilon})$ und rechtem oberen Punkt $R_r(max_x + \texttt{epsilon}, max_y + \texttt{epsilon})$. Überlappen die Bounding-Boxen des Approximanten L und des Punktes Q nicht, so befindet sich der Punkt Q außerhalb der ϵ-Umgebung von L und der Algorithmus terminiert. Im Falle einer Überlappung der Bounding-Boxen wird der Approximant in einen linken L_l und einen rechten L_r Approximant mit möglichst gleich vielen Liniensegmenten unterteilt, auf denen wieder der schnelle Ablehnungstest angewendet wird. Besteht der Approximant L aus einem einzigen Liniensegment l_i, wird der Abstand $\text{dist}(Q, l_i) <= \epsilon$ abgeschätzt. Ist das Ergebnis

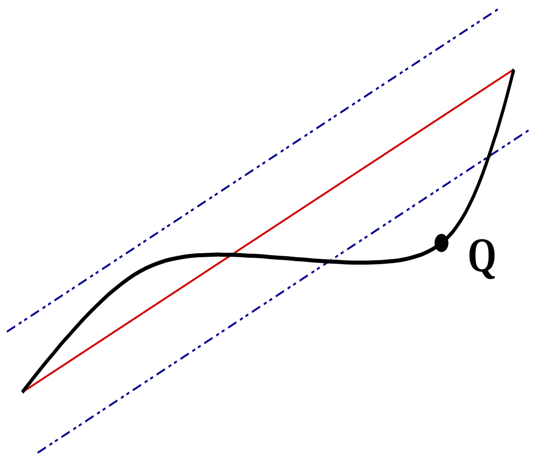


Abbildung 3.6: Beispiel eines Kurvensegmentes, der nicht in der ϵ-Umgebung des approximierenden Liniensegments ist.

dieses Testes positiv, dann ist der Punkt Q höchstens 2ϵ entfernt von der Kurve. Der Pseudo-Code ist in Alg. 3.3.1 angegeben. Durch den Parameter ϵ kann ein beliebiger Genauigkeitsgrad erreicht werden. Allerdings ist zu beachten, dass die im Abschnitt 3.1 vorgestellten heuristischen Approximationsmethoden keine Garantie für die Erfüllung der vom Algorithmus verlangten notwendigen Approximationsbedingung (siehe 3.9) bei allgemeinen parametrischen Kurven geben. D. h. es kann zu der auf Abb. 3.6 dargestellten Situation kommen, bei der nicht der ganze Kurvensegment in der ϵ-Umgebung des approximierenden Liniensegments bleibt. Folglich trifft der Algorithmus die falsche Entscheidung, dass der Punkt O nicht auf der Kurve ist. Anders bei Bézier-Kurven und B-Splines garantieren die implementierten Approximationstechniken die geforderte Bedingung 3.9, wie in Kapitel 4 bzw. 5 gezeigt, und der hier vorgestellte Algorithmus funktioniert korrekt nach seiner Spezifikation.

3.4 Ermittlung der Schnittpunkte zweier Kurven

Bei vielen Anwendungen der grafischen Datenverarbeitung sind die Schnittpunkte von ebenen Kurven zu bestimmen. Am einfachsten ist die Lösung des Schnittproblems für Kurven, die mit ihrer expliziten Darstellung $y_1 = f_1(x)$, $y_2 = f_2(x)$ gegeben sind. In diesem Fall müssen die Nullstellen der Gleichung $f_1(x) - f_2(x) = 0$ ermittelt werden. Diese Arbeit befasst sich aber nur mit Kurven in Parameterdarstellung. In [JH89] werden einige Schnittalgorithmen für parametrische Kurven diskutiert. Die auf der Kenntnis der Funktionen $x(t)$ und $y(t)$ basierenden numerischen und algebraischen Methoden sind hier nicht einsetzbar, da diese Funktionen den Charakter einer Black-Box haben. Die Idee der vorgestellten Unterteilungsmethode ist es, schrittweise die Bereiche der möglichen Schnittpunkten abzuschätzen und zu verkleinern, bis die Kurven in diesen Bereichen mit Liniensegmenten approximiert werden können. Dann sind die Schnittpunkte der Liniensegmente eine Approximation der Kurvenschnittpunkte. Nach diesem Prinzip funktioniert auch der hier implementierte Schnittalgorithmus (siehe Alg. 3.4.1). Zu einer gegebenen Toleranz ϵ werden, wie im Abschnitt 3.1 gezeigt, die

intersections(L_1, L_2, ϵ, P)

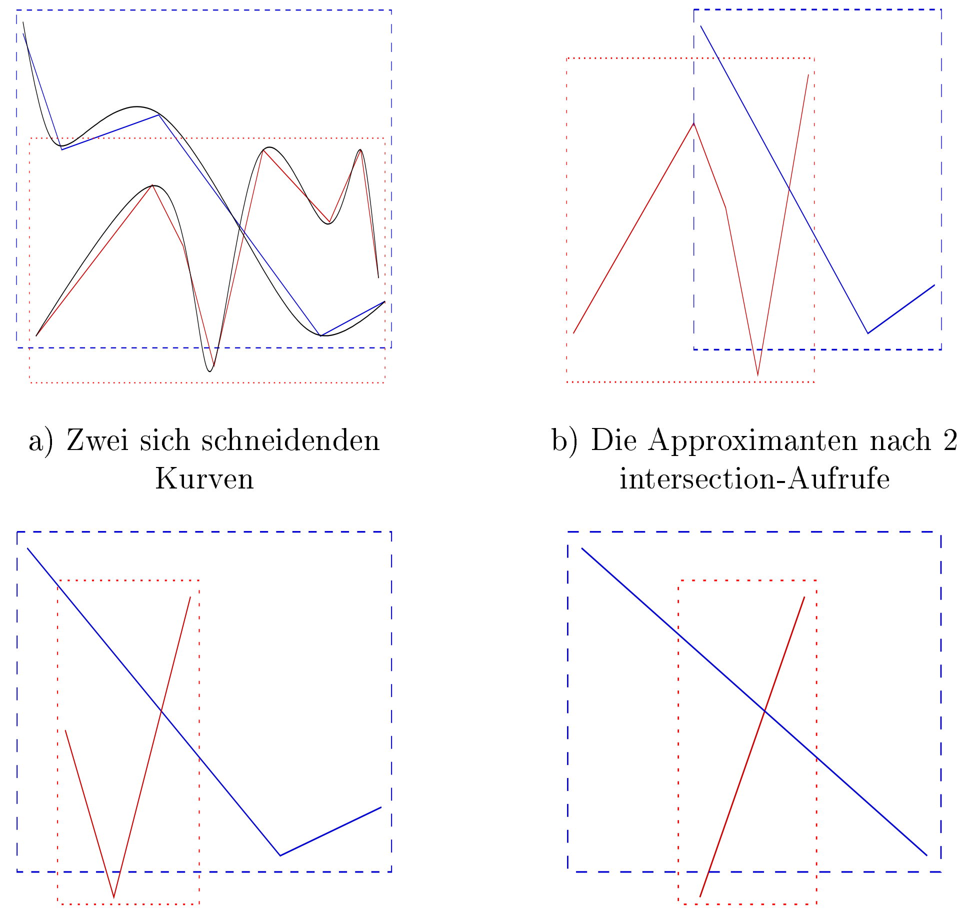
Require: L_1, L_2: list of line segments approximating the two parametric curves within tolerance ϵ, P: list of the intersection points of L_1 and L_2, initially $P = \emptyset$

Ensure: if there is no intersection $P = \emptyset$
  else $P = \{P_i : P_i = l_k \cap l_m, l_k \in L_1, l_m \in L_2\}$
  if $|L_1| < |L_2|$ **then**
    $L_s := L_1, L_l := L_2$
  else
    $L_s := L_2, L_l := L_1$
  end if
  $num := |L_l|$
  if $|L_s| = 0$ **or** !do_overlap(bbox(L_s), bbox(L_l)) **then**
    return
  else if $|L_s| = 1$ and $|L_l| = 1$ **then**
    insert $p := l_s \cap l_l$ in P
    return
  else
    $m := \left\lfloor \frac{num}{2} \right\rfloor$
    divide L_l in $L_{ll} := \{l_0, \ldots, l_{m-1}\}$ and $L_{rl} := \{l_m, \ldots l_{num-1}\}$
    intersections$(L_s, L_{ll}, \epsilon, P)$
    intersections$(L_s, L_{lr}, \epsilon, P)$
    return
  end if

Algorithmus 3.4.1: Ermittlung der Schnittpunkte zweier parametrischen Kurven

stückweise linearen Approximante L_1, L_2 der Kurven erstellt, die idealerweise die Bedingung 3.9 erfüllen. Ein schneller Ablehnungstest der Bounding-

Boxen von L_1, L_2 entscheidet, ob der Algorithmus terminiert oder ob potenzielle Schnittstellen existieren. Im letzten Falle erfolgt die Unterteilung des längeren (derjenige, der mehr Liniensegmente hat) Approximanten in zwei, möglichst gleich lange Approximanten L_l und L_r, deren Bounding-Boxen weiter auf Überschneidungen mit dem kürzeren Approximant getestet werden. Bestehen beide Approximanten aus je einem Liniensegment, dann wird ihr Schnittpunkt zu der Schnittpunktliste hinzugefügt. Dieser Prozess wird in Abb. 3.7 veranschaulicht. Der Parameter ϵ dient der Genauigkeitssteuerung.



a) Zwei sich schneidenden Kurven

b) Die Approximanten nach 2 intersection-Aufrufe

c) Die Approximanten nach 3 intersection-Aufrufe

d) Die Approximanten nach 5 intersection-Aufrufe

Abbildung 3.7: Ermittlung der Schnittpunkte zweier Kurven durch scrittweise Verkleinerung der potenziellen Schnittbereichen

Wie schon im Abschnitt 3.3 erwähnt, besteht das Problem bei allgemeinen parametrischen Kurven in der Erzeugung von stückweise linearen Approximanten, die der Bedingung 3.9 genügen.

3.5 Implementierung der Datenstruktur für parametrische Kurven

Die Klasse `Parametric_curve_2` implementiert eine ebene parametrische Kurve, gegeben durch die Funktionen $x(t)$ und $y(t)$ und die Intervalgrenzen $[a, b]$ (siehe 2.5). Diese die Kurve definierenden Merkmale werden als Datenfelder der Klasse gespeichert. Wie schon am Anfang des Kapitels erläutert, wird für die meisten Operationen auf der Kurve eine stückweise lineare Annäherung zu einer gegebenen Toleranz benutzt. Die Klasse `Piecewise_lin_approximant_2` implementiert einen stückweise linearen Approximant und wird im Abschnitt 3.6 näher behandelt. Ein Objekt dieser Klasse und die dazugehörige Toleranz `eps`, für die idealerweise Bedingung 3.9 erfüllt ist, werden als Datenfelder in der Klasse `Parametric_curve_2` gespeichert. Beim Erzeugen eines Kurvenobjekts wird mittels äquidistanter Parametrisierung (siehe 3.7) ein stückweise linearer Approximant erstellt und `eps` ein Initialisierungswert zugewiesen. Bei allen Methoden, die mit der Approximation der Kurve arbeiten, kann durch das Parameter `epsilon` die gewünschte Toleranz eingestellt werden. Ist beim Aufruf einer solchen Methode `epsilon<eps`, wird eine adaptive Verfeinerung der Approximation (z.B. nach dem Algorithmus 3.1.2) vorgenommen und der Wert von `eps` aktualisiert. Das Speichern des linearen Approximanten führt zwar zu höherem Speicherplatzbedarf, reduziert aber die Aufrufe der teuren Approximationsprozedur z.B. bei Methoden mit geringeren Genauigkeitsanforderungen `epsilon`$\geq$`eps`. Dadurch wird die Effizienz der Implementierung wesentlich erhöht.
Mit Bezug auf die Konzepte der generischen Programmierung ist die Klasse `Parametric_curve_2` ein Template, dessen Parameter, die Typen `X` und `Y`, Pointer zu Funktionen oder Funktionsobjekten sind, die $x(t)$ und $y(t)$ implementieren. Funktionsobjekte sind Klassen, die wie eine Funktion agieren, d. h. den Operator () definieren. Auf diese Weise ist dem Benutzer die Möglichkeit gegeben, mit seinen eigenen Funktionstypen `X` und `Y` den Template `Parametric_curve_2` zu instantiieren und dadurch eigene Typen von parametrischen Kurven zu erzeugen. Zum Beispiel wird der Typ `Epizykloide` wie folgt definiert:

```
typedef Epizykloide_x<NumType> X_e;
typedef Epizykloide_y<NumType> Y_e;
typedef Parametric_curve_2<NumType, X_e *, Y_e *> Epizykloide;
```

wobei die Funktionsobjekte `Epizykloide_x` und `Epizykloide_y` die Funktionen $x(t) = (R+r)\cos(t) - a\cos\left(\frac{R+r}{r}t\right)$ und $y(t) = (R+r)\sin(t) - a\sin\left(\frac{R+r}{r}t\right)$

implementieren.
Dem Konstruktor von `Parametric_curve_2` werden als Parameter zwei Pointer zu den Funktionsobjekten, die $x(t)$, $y(t)$ implementieren, übergeben, so wie die Parameterintervalgrenzen `a` und `b`. Falls keine Werte für `a` und `b` angegeben wurden, wird `a=0` bzw. `b=1` angenommen. Ausserdem sind ein Copy-Konstruktor und ein protected parameterloser Konstruktor vorhanden. Die Klasse verfügt über folgende Methoden:

- `Point calculate_point_at(NumType t)` - berechnet einen Punkt auf der Kurve zu einem gegeben Parameterwert `t`.

- `NumType x(NumType t)` und `NumType y(NumType t)` - werten die Funktionen $x(t)$ bzw. $y(t)$ für ein gegebenes t aus.

- `void initial_approximation(unsigned num=NUMBER)` - erzeugt einen linearen Approximant der Kurve, in dem äquidistante Parametrisierung eingesetzt wird (siehe Abschnitt 3.1). `num` ist die Anzahl der über dem Definitionsintervall $[a, b]$ gleichmäßig verteilten Stützpunkte. Desweiteren wird ein Initialisierungswert für die Toleranz `eps` ermittelt.

- `void refine_approximation(NumType epsilon=EPSILON, bool use_deriv=true)` - verfeinert den linearen Approximant dieser Kurve mit dem Ziel, $|P(t) - L(t)| \leq$ `epsilon` zu erreichen. Der boolische Parameter `use_deriv` ermöglicht die Wahl zwischen den beiden Algorithmen für adaptive Parametrisierung, die in Abschnitt 3.1 dargestellt wurden.

- `void insert_pt_in_middle_approximation(NumType epsilon=EPSILON)` - verfeinert den stückweise linearen Approximant dieser Kurve, wobei adaptive Parametrisierung durch Abschätzung der Abweichung in der Mitte des Intervalls (siehe Alg. 3.1.1) benutzt wird.

- `void second_deriv_approximation(NumType epsilon=EPSILON)`- verfeinert den stückweise linearen Approximant dieser Kurve, wobei adaptive Parametrisierung mittels Abschätzung der 2. Ableitung (siehe Alg. 3.1.2) angewendet wird.

- `void draw()` - zeichnet den stückweise linearen Approximant der Kurve in einem Fenster.

- `Bbox bbox()` - gibt die Bounding-Box des stückweise linearen Approximanten dieser Kurve zurück.

- `bool has_on(const Point& pt, NumType epsilon=EPSILON)` - liefert `true`, wenn der Punkt `pt` in der `epsilon`-Umgebung des stückweise linearen Approximanten der Kurve liegt, `false` sonst.

- `template<class Xfunc, class Yfunc> void intersections(Points & intersec_pts, const Parametric_curve_2<NumType, Xfunc, Yfunc> & curve, NumType epsilon=EPSILON)` - bestimmt die Schnittpunkte der stückweise linearen Approximanten dieser Kurven und gibt sie im Vektor `intersec_pts` zurück.

- `bool evaluate_derivative_at(NumType t, Vector & deriv, unsigned r=DEGREE)` - ermittelt diskret die r-te Ableitung des stückweise linearen Approximanten dieser Kurve und wertet sie für das gegebene `t` aus. Gibt `true` zurück, falls die Ableitung existiert, `false` sonst.

3.6 Implementierung des stückweise linearen Approximanten

Für den stückweise linearen Approximant einer parametrische Kurve wurde folgende Datenstruktur in der Klasse `Piecewise_lin_approximant_2` implementiert. Sie enthält einen Vektor mit Stützpunkten P_i auf der Kurve und einen Vektor mit den dazugehörigen Parameterwerten t_i, d. h $P_i = p(t_i)$. Zusätzlich wird die Anzahl der Elemente der Vektoren verwaltet, da auf sie relativ oft zugegriffen wird.
Unter Berücksichtigung der Konzepte der generischen Programmierung ist die Klasse `Piecewise_lin_approximant_2` als ein Template organisiert, dessen Parameter der Zahlentyp der Koordinaten, ein Pointiterator- und ein Parameteriteratortyp sind. Diese Parametrisierung erhöht die Flexibilität der Implementierung, in dem sie erlaubt, die Stützpunkte bzw. die Parameterwerte aus beliebigen Containern auszulesen. Dem Benutzer stehen folgende Konstruktoren zur Verfügung:

- `Piecewise_lin_approximant_2(const PointIt first, const PointIt last, NumType a=LOW_BOUND, NumType b=UP_BOUND)` - erzeugt einen Approximant mit Stützpunkten die Punkte aus dem Bereich `[first, last)`, die gleichmäßig über dem Intervall `[a, b]` verteilt sind (siehe Abschnitt 3.1).

- `Piecewise_lin_approximant_2(const PointIt first, const PointIt last, const ParamIt first_t)` - konstruiert einen Approximant, dessen Stützpunkten die Punkte aus dem Bereich [first,last) sind und bei Parameterwerten aus dem Bereich [first_t, first_t+(last - first)) auf der Kurve abgelesen wurden. Dieser Konstruktor ermöglicht das Erzeugen von ungleichmäßig parametrisierte Approximanten (siehe Abschnitt 3.1).

Ausserdem sind ein Copy-Kontruktor und ein parameterloser Konstruktor vorhanden. Die Klasse verfügt über folgende Methoden:

- `Point evaluate_at(NumType t)` - ermittelt zum gegebenen Parameter `t` den Punkt auf dem stückweise linearen Approximat . Dabei wird zunächst nach einem Parameterwert $t_i > t$ gesucht, um das Intervall $t \in [t_{i-1}, t_i)$ festzustellen. Dann wird 3.4 eingesetzt um den Punkt $l(t)$ auszuwerten.

- `void draw()` - zeichnet den stückweise linearen Approximant, in dem die Stützpunkte $\{P_i\}$ mit Liniensegmenten verbunden werden.

- `Bbox bbox(NumType epsilon=0)` - erzeugt die um `epsilon` vergrößerte Bounding-Box des stückweise linearen Approximanten, d. h. das Min-Max-Rechteck mit linkem unteren Punkt $R_l(min_x - \texttt{epsilon}, min_y - \texttt{epsilon})$ und rechtem oberen Punkt $R_r(max_x + \texttt{epsilon}, max_y + \texttt{epsilon})$, wobei $\forall P_i \in L : P_i.x \geq min_x \wedge P_i.x \leq max_x \wedge P_i.y \geq min_y \wedge P_i.y \leq max_y$ gilt.

- `void divide(Piecewise_lin_approximant_2 & first, Piecewise_lin_approximant_2 & second)` - unterteilt den stückweise linearen Approximant in zwei Approximante mit möglichst gleich vielen Stützpunkten, wobei `first` $:= \{(P_i, t_i) : i = 0, \ldots, \left\lfloor \frac{num}{2} \right\rfloor\}$ und `second` $:= \{(P_i, t_i) : i = num - \left\lfloor \frac{num}{2} \right\rfloor, \ldots, num\}$

- `bool has_on(const Point& pt, NumType epsilon=EPSILON)` - implementiert Algorithmus 3.3.1 und gibt `true` zurück, falls der Punkt `pt` annähernd auf dem stückweise linearen Approximant liegt, d. h. $\exists i : distance(\overline{P_{i-1}P_i}, Q) <=$ `epsilon`, `false` sonst.

- `void intersections(Points & intersec_pts, constPiecewise_lin_approximant_2 & approx, NumType epsilon=EPSILON)` - implementiert Algorithmus 3.4.1 zum Bestimmen der Schnittpunkte zweier stückweise linearen Approximanten und gibt diese im Vektor `intersec_pts` zurück.

- `bool discrete_derivative(Piecewise_lin_approximant_2 & deriv_approx, unsigned r)` - bestimmt die r-te diskrete Ableitung des stückweise linearen Approximanten wie in Abschnitt 3.2 gezeigt und gibt `true` zurück, wenn sie existiert, `false` sonst.

Kapitel 4

Bézier-Kurven

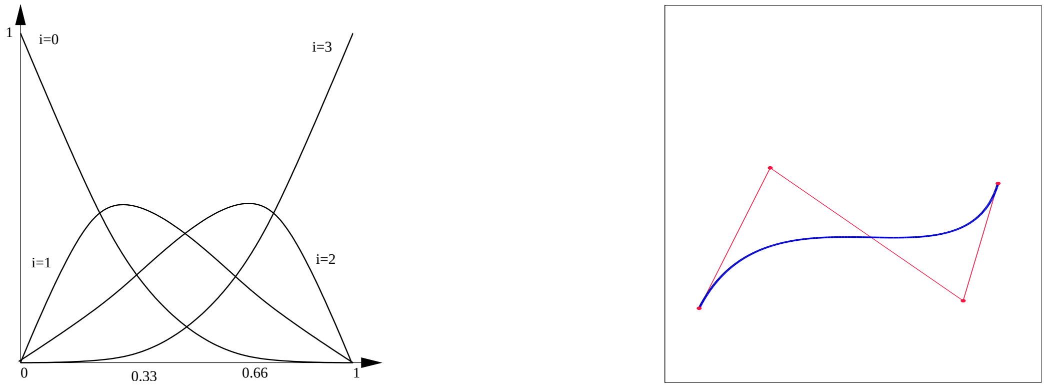
Im Abschnitt 3.1 des vorigen Kapitels wurden zwei Algorithmen für stückweise lineare Approximation von parametrischen Kurven vorgestellt. Der Nachteil dieser Methoden war, dass sie keine Garantie für den maximalen Wert der Abweichung des Approximanten von der Kurve geben. Dem zu Folge gibt es auch keine Garantie für die Korrektheit der implementierten Methoden der Klasse `Parametric_curve_2`, die auf der Basis eines stückweise linearen Approximanten und einer bekannten Toleranz arbeiten. In diesem Kapitel wird eine weitere Technik zur Kurvenapproximation behandelt, die in den 60-ger Jahren unabhängig von Pierre Bézier und Paul de Casteljau für die CAD-Systeme zweier Automobilhersteller entwickelt wurde. Dadurch können anhand von Stützpunkten glatte parametrische Kurven konstruiert werden, die eine durch die Stützpunkte vorgegebene Form approximieren, und bei denen eine obere Schranke der Approximationsabweichung ermittelt werden kann. Diese Kurven werden nach einem ihrer Entdecker *Bézier-Kurven* genannt. Eine explizite Formel für die Berechnung eines Punktes auf einer *Bézier-Kurve vom Grad n* ist durch die Gleichung

$$b(t) = \sum_{i=0}^{n} B_i^n(t) P_i,\ t \in [a, b] \tag{4.1}$$

gegeben (siehe z.B. [Fel92], [JH89], [Gol]). Die Punkte P_i werden *Stützpunkte* oder *Kontrollpunkte* der Bézier-Kurve und der Streckenzug durch diese *Kontrollpolygon* genannt. Die Anzahl der Kontrollpunkte bestimmt den Grad der Kurve ($n = \#Kontrollpunkte - 1$). Die Funktionen $B_i^n(t)$ sind die *Bernsteinpolynome vom Grad n* der Form

$$B_i^n(t) = \frac{n!}{i!(n-i)!} \frac{(t-a)^i (b-t)^{n-i}}{(b-a)^n},\ t \in [a, b] \tag{4.2}$$

Sie bestimmen den Einfluss der Stützpunkte P_i auf den Kurvenverlauf, wie in Abbildung 4.1 verdeutlicht wird. Zwischen den geometrischen Eigenschaften



a) Bernsteinpolynome vom Grad 3 über $[0, 1]$

b) Bézier-Kurve mit vier Kontrollpunkten

Abbildung 4.1: Gewichtung der Kontrollpunkte P_i durch die Bernsteinpolynome B_i^3 bei einer Bézier-Kurve vom Grad 3

der Bézier-Kurven und den algebraischen Eigenschaften der Bernsteinpolynome existiert eine enge Beziehung, die in [Far88] und [Gol] gründlich behandelt wird. Im Folgenden werden die für die Implementierung der Bèzier-Kurven wichtigen Eigenschaften erwähnt:

- *Affine Invarianz*
  Mit Hilfe des Binomial-Theorems

  $$(x+y)^n = \sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) x^i y^{n-i} \tag{4.3}$$

  und durch Einsetzen von $x = \frac{t-a}{b-a}$ und $y = \frac{b-t}{b-a}$ kann gezeigt werden, dass die Koeffizienten in (4.1) zu 1 summieren, d.h.

  $$\forall t : \sum_{i=0}^{n} B_i^n(t) \equiv 1. \tag{4.4}$$

  Daher ist jeder Punkt $b(t)$ eine affine Kombination und die Bézier-Kurve ist invariant bezüglich affinen Transformationen. Diese Eigenschaft ist von großer Bedeutung, da sie die Unabhängigkeit der Kurve von der Wahl des Koordinatensystems garantiert.

- *Die Bézier-Kurve interpoliert ihre Endpukte*
  Der Beweis basiert auf

  $$B_0^n(a) = 1$$
  $$B_n^n(b) = 1$$

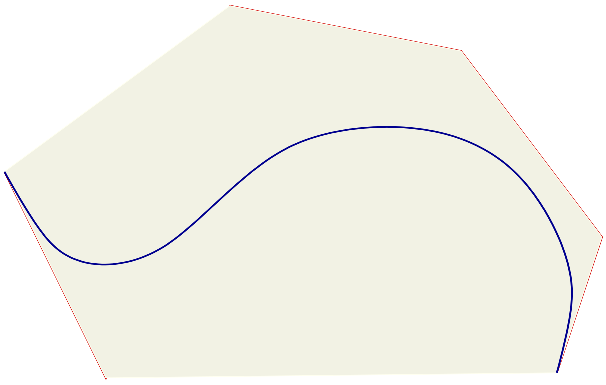


Abbildung 4.2: Bézier-Kurve vom Grad 5 und die konvexe Hülle ihrer Kontrollpunkte

und Gleichung (4.4). Daraus folgt, dass die restlichen Koeffizienten in (4.1) bei $t = a$ bzw. $t = b$ Null sind und

$$b(a) = P_0 \ \wedge \ b(b) = P_n \tag{4.5}$$

gilt. Die Interpolationseigenschaft garantiert, dass durch die Zusammensetzung der Kontrollpolygone von zwei Bézier-Kurven ($P_n = Q_0$), eine stetige Kurve entsteht.

- *Die Bézier-Kurve liegt in der konvexen Hülle ihrer Kontrollpunkte*
  Diese Eigenschaft folgt wegen Gleichung 4.4 und aus der Tatsache, dass für $t \in [a, b]$ alle Bernsteinpolynome positiv sind ($B_i^n(t) \geq 0$). Anhand dieser Eigenschaft kann, wie in Abschnitt 4.2 näher behandelt, eine obere Schranke für die Abweichung der Kurve von ihrem stückweise linearen Approximant abgeleitet werden.

- *Globaler Einfluß der Kontrollpunkte auf die Kurve*
  Diese Eigenschaft folgt aus der Tatsache, dass im Intervall (a, b) $B_i^n \neq 0$ sind und somit beeinflußt jeder Punkt den Kurvenverlauf im gesamten Intervall. Da die Bernsteinpolynome in $[a, b]$ genau ein lokales Maximum bezüglich t bei $t = \frac{i}{n}$ erreichen (siehe Abb. 4.1a), ist an dieser Stelle der Einfluß des Punktes P_i auf die Kurve am stärksten.

- *Beschränkte Schwankung (variation diminishing property)*
  Diese Eigenschaft besagt, dass eine beliebige Gerade die Bézier-Kurve höchstens so oft schneidet, wie sie ihr Kontrollpolygon schneidet. Also folgt die Bézier-Kurve ihr Kontrollpolygon und neigt nicht zu unerwünschten Schwankungen.

4.1 Auswertungsalgorithmen

Zur Berechnung eines Punktes auf einer Bézier-Kurven wurden in dieser Arbeit zwei Ansätze implementiert, die im Folgenden vorgestellt sind. Der rekursive Algorithmus von de Casteljau hat eine quadratische Laufzeit. Der zweite Ansatz basiert auf die Anwendung des Horner-Schemas und läuft in linearer Zeit. Werden beide Algorithmen anhand der Anzahl ausgeführter Realmultiplikationen verglichen ($\frac{1}{2}n(n+1)$ bei de Casteljau und $3n+1$ bei Horner), so ist der Algorithmus von Horner dem de Casteljau-Algorithmus erst bei Bézier-Kurven des Grades $n \geq 5$ vorzuziehen (siehe [Fel92]). Aus diesem Grund wurden in dieser Arbeit beide Algorithmen implementiert. Sie sind als Funktionsobjekte organisiert, deren Konstruktor ein Pointer zu dem Vektor mit den Kontrollpunkten der Bézier-Kurve und die Intervallgrenzen `[a,b]` als Parameter bekommt.

4.1.1 De Casteljau-Algorithmus

Gleichzeitig und unabhängig von P. Bézier hat auch P. de Casteljau auf dem Gebiet der Kurvenapproximationen für CAD-Anwendungen geforscht. Der nach ihm benannte Auswertungsallgorithmus für Bézier-Kurven kann von der mathematischen Definition 4.1 abgeleitet werden (siehe [Fel92]). Folgende Rekursionsgleichung liegt dem Algorithmus zugrunde (siehe z.B. [Far88]):

$$b_i^r(t) = \frac{b-t}{b-a} b_i^{r-1}(t) + \frac{t-a}{b-a} b_{i+1}^{r-1}(t),\ t \in [a,b],\ b_i^0(t) = P_i \tag{4.6}$$

Der Punkt $b_0^n(t)$ ist der gesuchte Punkt auf der Bézier-Kurve zum Parameterwert t. Der Pseudo-Code des Algorithmus ist in 4.1.1 gegeben. Der de

de_Casteljau($\{P_i\}, t$)
Require: $P_0, \ldots, P_n$: the control points of this Bezier curve, t
Ensure: $b_0^n(t)$ is the point on this Bezier curve at t
  for $i := 0$ to n **do**
    $b_i := P_i$
  end for
  for $r := n-1$ down to 0 **do**
    for $i := 0$ to r **do**
      $b_i := \frac{b-t}{b-a} b_i + \frac{t-a}{b-a} b_{i+1}$
    end for
  end for

Algorithmus 4.1.1: De Casteljau-Algoritmus

Casteljau-Algorithmus ist im Funktionsobjekt `DeCasteljau_FO` implementiert.

4.1.2 Horner Algorithmus

Um das Horner-Schema zur Auswertung von Bézier-Kurven anzuwenden, muss Gleichung 4.1 von der Bernsteinbasis in Monombasis überführt werden. Wie in [Gol] gezeigt, erfolgt die Basistransformation durch

$$b(t) = \begin{cases} \left(\frac{b-t}{b-a}\right)^n \sum_{i=0}^{n} \begin{pmatrix} n \\ i \end{pmatrix} P_i u^i, & u = \frac{t-a}{b-t}, \quad t \in [a, \frac{b+a}{2}] \\ \\ \left(\frac{t-a}{b-a}\right)^n \sum_{i=0}^{n} \begin{pmatrix} n \\ i \end{pmatrix} P_i u^{n-i}, & u = \frac{b-t}{t-a}, \quad t \in (\frac{b+a}{2}, b] \end{cases} \tag{4.7}$$

Das Ziel der getrennten Behandlung der Fälle $0 \leq t \leq \frac{b+a}{2}$ und $\frac{b+a}{2} < t \leq b$ ist es, den Parameterwert $u \leq 1$ zu halten und dadurch den Rundungsfehler zu reduzieren. Der Pseudo-Code des Algorithmus ist in 4.1.2. Der Horner-

horner$(\{P_i\}, t)$
Require: $P_0, \ldots, P_n$: the control points of this Bezier curve, t
Ensure: b is the point on this Bezier curve at t
  c:=**binomilas**(n)
  if $t \leq \frac{b+a}{2}$ **then**
    $b := P_n$
    for $i := n$ down to 1 **do**
      $b := b\frac{t-a}{b-t} + P_{i-1}c_{i-1}$
    end for
    $b := b\left(\frac{b-t}{b-a}\right)^n$
  else
    $b := P_0$
    for $i := 1$ to n **do**
      $b := b\frac{b-t}{t-a} + P_i c_i$
    end for
    $b := b\left(\frac{t-a}{b-a}\right)^n$
  end if

Algorithmus 4.1.2: Horner-Algoritmus

Algorithmus ist im Funktionsobjekt `Horner_FO` implementiert.

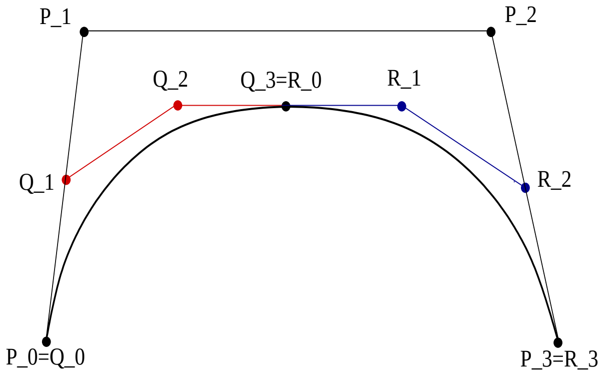


Abbildung 4.3: Unterteilung einer Bézier-Kurve vom Grad 3 in zwei Bézier-Kurven des gleichen Grades

4.2 Stückweise lineare Approximation durch Subdivision

Ähnlich wie bei der allgemeinen parametrischen Kurve wird zum Zwecke der meisten Operationen auf Bézier-Kurven (z.B. Zeichnen, Schnittpunkte bestimmen) ein stückweise linearer Approximant benötigt. Mit den de Casteljau- und Horner-Algorithmen verfügen wir über Methoden zum Auswerten von $b(t)$. Dann kann ein Approximant der Kurve nach den in Kapitel 3.1 vorgestellten Techniken konstruiert werden. Für Bézier-Kurven existieren aber effizientere Approximationsansätze, wie z.B. Graderhöhung (degree elevation) und Subdivision (siehe [GM01]). Die Graderhöhungsmethode nutzt die Tatsache, dass jeder Polynom vom Grad n auch ein Polynom vom Grad $v > n$ ist, um eine Bézier-Kurve vom Grad n in der Bernsteinbasis vom Grad $n+1$ darzustellen. Der Kontrollpunktensatz $\{Q_0, \ldots\ Q_{n+1}\}$, der die Bézier-Kurve in der neuen Bernsteinbasis definiert, hat einen Punkt mehr und erlaubt dadurch zusätzliche Kontrolle auf der Kurve. In [JH89] wird gezeigt, dass das durch wiederholte Graderhöhung entstandene Kontrollpolygon gegen die Bézier-Kurve konvergiert. Folglich kann Graderhöhung zur stückweise linearen Approximation der Kurve eingesetzt werden. Die Konvergenzrate dieser Methode ist $O\left(\frac{1}{v}\right)$, wobei v der erhöhte Grad ist.
Subdivision basiert auf der Spaltung der Bézier-Kurve bei $t = m \in (a, b)$ in zwei neue Bézier-Kurven gleichen Grades, die jeweils im Intervall $[a, m]$ bzw. $[m, b]$ definiert sind und bei deren Zusammensetzung genau die Originalkurve über $[a, b]$ entsteht. In [Gol] wird gezeigt, dass beim rekursiven Fortsetzen dieser Unterteilungen die Kontrollpolygone der Teilkurven gegen die Ausgangskurve konvergieren. Wird die Unterteilung immer in der Mitte

vorgenommen (d. h. $m = \frac{a_i+b_i}{2}$), dann beträgt die Konvergenzrate $O\left(\frac{1}{2^i}\right)$, wobei i die Anzahl rekursiver Unterteilungsaufrufe ist. Also hat rekursive Subdivision eine bessere Konvergenzrate als Graderhöhung und wurde aus diesem Grund als Approximationsmethode in dieser Arbeit vorgezogen. In [Far88] wird gezeigt, wie der de Casteljau-Algorithmus zur Bestimmung der Kontrollpunkte der linken $l(t)$ und rechten $r(t)$ Bézier-Teilkurven eingesetzt wird. Bezeichnet man die Kontrollpunkte der linken Kurve mit Q_j und die der rechten mit R_j, dann gilt folgende *Subdivisionformel* für Bézier-Kurven:

$$\begin{aligned} &Q_j = b_0^j(m),\ m \in (a,b),\ j = 0 \ldots n \\ &R_j = b_{n-j}^j(m), \end{aligned} \tag{4.8}$$

wobei b_i^r mit der de Casteljau-Rekursionsgleichung (siehe 4.6) definiert ist. Der stückweise linearer Approximant der Bézier-Kurve wird kontruiert, in dem die Endkontrollpunkte der Teilkurven mit Liniensegmenten verbunden werden. [Gol] schlägt dafür zwei im Folgenden erläuterten Ansätze vor, die sich in der Bestimmung der notwendigen Anzahl Unterteilungen unterscheiden. Da eine Bézier-Kurve in der konvexen Hülle ihrer Kontrollpunkte liegt (siehe Eigenschaften in Kapitel 4), gilt

$$dist(b(t), \overline{P_0P_n}) < \max_{i=1\ldots n-1}(dist(P_i, \overline{P_0P_n})) \tag{4.9}$$

Auf dieser Abschätzung der maximalen Abweichung der Kurve von dem Liniensegment, das ihre Endpunkte verbindet, basiert der folgende rekursive Approximationsalgorithmus für Bézier-Kurven (siehe Alg. 4.2.1). Der Algorithmus nimmt eine Unterteilung nur dort vor, wo das Liniensegment zwischen den Endkontrollpunkten das Kurvensegment nicht zur gewünschten Toleranz annähert, d. h. $\exists P_i:\ dist(P_i, \overline{P_0P_n}) > \epsilon,\ i = 1\ldots n-1$. Um dies festzustellen, wird bei jedem Durchlauf $n-1$ Mal der Euklidische Abstand eines Punktes zu einem Liniensegment ermittelt. Diese Operation ist aufwendig und ihre wiederholte Ausführung wirkt sich nachteilig auf die Effizienz des Algorithmus aus. Der zweite in [Gol] vorgeschlagene Ansatz, basiert auf der Vorausberechnung der Anzahl notwendiger Unterteilungen. In Gleichung 3.5 wurde die maximale Abweichung eines Kurvensegments von dem Liniensegment, der seine Endpunkte interpoliert, durch den maximalen Wert der zweiten Ableitung abgeschätzt. Da bei Bézier-Kurven

$$\begin{aligned} &max|b''(t)| \le n(n-1)M \\ &M = \max_{k=0\ldots n-2}|P_{k+2} - 2P_{k+1} + P_k| \end{aligned} \tag{4.10}$$

gilt, ist die Abweichung einer Teil-Bezier-Kurve $b_r(t)$ von dem Liniensegment $l_r(t)$, das ihre Endkontrollpunkte verbindet, nach r rekursiven Unterteilungen

rendering($\{P_i\}, L, \epsilon$)

Require: $P_0, \ldots, P_n$: the control points of this Bezier curve, L: empty list of segments, ϵ the approximation tolerance

Ensure: L approximates the Bézier curve to within tolerance ϵ

```
whitin_tolerance := true
for i := 1 to n − 1 do
  if squared_distance(P_i, P_0P_n) > ε then
    whitin_tolerance := false
  end if
end for
if whitin_tolerance then
  insert P_0P_n in L
else
  subdivide this Bézier curve calling deCasteljau({P_i}, (a+b)/2) in
  one left and one right Bézier curves of degree n with control
  points Q_i = b_0^i((a+b)/2) resp. R_i = b_{n−i}^i((a+b)/2)
  rendering({Q_i}, L, ε)
  rendering({R_i}, L, ε)
end if
```

Algorithmus 4.2.1: Approximation durch Subdivision und Ermitteln der maximalen Distanz von einem inneren Kontrollpunkt zum Liniensegment, das die Endkontrollpunkte verbindet.

in der Mitte wie folgt gegeben:

$$dist(b_r(t), l_r(t)) \leq n(n-1)M\frac{(0.5)^{2r}}{8}. \tag{4.11}$$

Sei ϵ die maximal zulässige Abweichung der Bézier-Kurve von ihrem stückweise linearen Approximat, dann ist die Anzahl r der benötigten rekursiven Subdivision-Aufrufe durch

$$r = \left\lceil \log_4\left(\frac{n(n-1)M}{8\epsilon}\right)\right\rceil \tag{4.12}$$

gegeben. Nach r rekursiven Unterteilungen kann jedes Kurvensegment durch das Liniensegment, das seine Endpunkte verbindet, zu der geforderten Toleranz approximiert werden. Der in Alg. 4.2.2 vorgestellte Algorithmus berechnet zunächst die notwendige Anzahl rekursiver Unterteilungsschritte, führt diese aus und verbindet zum Schluss die Unterteilungspunkte mit Liniensegmenten. Der Nachteil dieser Vorgehensweise ist, dass Unterteilungen auch von solchen Kurvensegmenten vorgenommen werden, deren Annäherung durch das Liniensegment schon in den geforderten Grenzen lag. Dafür wird aber

subdivide_n_times($\{P_i\}, L, r$)

Require: $P_0, \ldots, P_n$: the control points of this Bezier curve, L: empty list of segments, $r \geq 0$ integer

Ensure: L contains the linked to segments subdivision points after r rounds subdivision

if $r = 0$ **then**

insert $\overline{P_0 P_n}$ in L

else

subdivide this Bézier curve calling **deCasteljau**($\{P_i\}, \frac{a+b}{2}$) in one left and one right Bézier curves of degree n with control points $Q_i = b_0^i(\frac{a+b}{2})$ resp. $R_i = b_{n-i}^i(\frac{a+b}{2})$

subdivide_n_times($\{Q_i\}, L, r-1$)

subdivide_n_times($\{R_i\}, L, r-1$)

end if

wang_rendering($\{P_i\}, L, \epsilon$)

Require: $P_0, \ldots, P_n$: the control points of this Bezier curve, L: empty list of segments, ϵ the approximation tolerance

Ensure: L approximates the Bézier curve to within tolerance ϵ

$r := \left\lceil \log_4 \left(\frac{n(n-1)M}{8\epsilon} \right) \right\rceil$

subdivide_n_times($\{P_i\}, L, r$)

Algorithmus 4.2.2: Approximation durch Subdivision und Vorausberechnung der notwendigen Subdivision-Aufrufe

der teure Distanztest nicht durchgeführt.
Die beiden hier vorgestellten Methoden konstruieren einen stückweise linearen Approximant der Kurve, der die Bedingung 3.9 erfüllt. Somit ist die Korrektheit der auf der Basis der Approximation arbeitenden Methoden gewährleistet.

4.3 Ableitung

Die erste Ableitung einer Bézier-Kurve ist durch folgende Formel gegeben (siehe z.B. [Far88]):

$$b'(t) = n \sum_{i=0}^{n-1} B_i^{n-1}(t)(P_{i+1} - P_i), \ t \in [a, b] \tag{4.13}$$

Beim Vergleich mit der Definitionsgleichung für Bézier-Kurven (siehe 4.1) fällt auf, dass die Ableitung wiederum eine Bézier-Kurve ist, deren Koeffizienten die Differenzen der Kontrollpunkte $(P_{i+1} - P_i)$ sind. Der Grad der neuen Bézier-Kurve ist um eins niedriger. Für die Auswertung der Ableitung kann der de Casteljau-Algorithmus eingesetzt werden, wobei das Endergebnis mit n multipliziert wird (wegen des Koeffizientes auf der rechten Seite von Gleichung 4.13). Für die Definition von Ableitungen höherer Ordnung wird der *Differenzoperator* $\triangle^r$ eingeführt:

$$\triangle^r P_i = \begin{cases} P_i & r = 0 \\ \triangle^{r-1} P_{i+1} - \triangle^{r-1} P_i & r > 0 \end{cases} \tag{4.14}$$

Damit ist die r-te Ableitung einer Bézier-Kurve wie folgt definiert:

$$b^{(r)}(t) = \frac{n!}{(n-r)!} \sum_{i=0}^{n-r} B_i^{n-r}(t) \triangle^r P_i \tag{4.15}$$

und stellt eine Bézier-Kurve vom Grad $(n-r)$ mit Koeffizienten $\triangle^r P_i$ dar. Die Auswertung der r-ten Ableitung erfolgt, wie in [Far88] gezeigt, durch den de Casteljau-Algorithmus und anschließende Multiplikation mit $\frac{n!}{(n-r)!}$.

4.4 Implementierung

Die Klasse `Bezier_curve_2` implementiert eine ebene Bézier-Kurve, die durch ihre Kontrollpunkte und Gleichung 4.1 definiert ist. Diese Klasse ist als ein Template organisiert, dessen Parameter der Zahlentyp der Koordinaten `NumType` und der Typ eines Auswertungsalgorithmus `A` (siehe Abschnitt 4.1) sind. Durch diese Parametrisierung wird die Flexibilität der Implementierung erhöht. Sie gibt dem Benutzer die Möglichkeit, den für die Zwecke seiner Anwendung effizientesten Auswertungsalgorithmus einzusetzen. Zur Wahl stehen die als Funktionsobjekte implementierten de Casteljau-Algorithmus (`DeCasteljau_FO`), Horner-Algorithmus (`Horner_FO`) und Bernsteinpolynome (`Bernstein_basis_FO`). Um die Benutzung der Klasse einfach zu halten, wird im Falle, dass keine Angabe gemacht wurde, der de Casteljau-Algorithmus als Standardauswertungsalgorithmus angewendet.

Die parametrische Natur der Bézier-Kurven findet Ausdruck in der Vererbung der Klasse `Bezier_curve_2` von der Klasse `Parametric_curve_2` (siehe das Klassendiagramm in Abb. 2.5). Dabei wird die Klasse `Parametric_curve_2` mit den Funktionsobjekttypen `X_FO<NumType, A*>*` und `Y_FO<NumType, A*>*` instanziiert. Die Aufgabe dieser Funktionsobjekte besteht darin, die x- und y-Koordinaten der anhand des Algorithmus `A` ausgewerteten Punkte zurückzugeben. Auf diese Weise ist es möglich, die Funktionen `x(NumType t)`, `y(NumType t)` und `calculate_point_at(NumType t)` der Oberklasse `Parametric_curve_2` direkt in der Unterklasse anzuwenden. Die Definition der Klasse `Bezier_curve_2` ist wie folgt:

```
template <class NumType, class A = DeCasteljau_FO<NumType> >
class Bezier_curve_2 : public Parametric_curve_2<NumType,
                                       X_FO<NumType, A*>*,
                                       Y_FO<NumType, A*>* >,
                       public Subdivision_curve_2<NumType> {
  ...
}
```

Die zweite Vererbungbeziehung zu der Klasse `Subdivision_curve_2` ist dadurch begründet, dass eine Bézier-Kurve auch anhand ihrer Kontrollpunkte und einer Subdivision-Methode dargestellt werden kann (siehe Abschnitt 4.2). Die Klasse `Subdivision_curve_2` ist für die Speicherung der Kontrollpunkte und die Implementierung der Methoden, die auf ihrer Basis arbeiten, zuständig. Die Klasse verfügt über einen Konstruktor, dem die Stützpunkte der Kurve als Parameter übergeben werden. Folgende Methoden sind in der Klasse `Subdivision_curve_2` implementiert:

- `void set_control_points(const PointList& new_contr_pts)` - setzt `new_contr_pts` als neue Kontrollpunkte der Sudivision-Kurve.
- `virtual unsigned degree()` - gibt den Grad der Kurve zurück.
- `bool discrete_differences(Points& diff, unsigned r=1)` - imlementiert die Anwendung des Differenzoperators $\triangle^r$ auf die Kontrollpunkte dieser Subdivision-Kurve (siehe Gleichung 4.14) und gibt die r. diskreten Differenzen im Vektor `diff` zurück. Die Methode gibt `false` zurück, falls die r. diskreten Differenzen nicht existieren (der Fall `r>degree()`).
- `double max_second_diff_value()` - gibt den größten Wert der diskreten Differenzen zweiter Ordnung zurück, d. h.
  $\max_{i=0...n-2} \triangle^2 P_i = \max_{i=0...n-2} |P_{i+2} - 2P_{i+1} + P_i|$.

Ein Objekt der Klasse `Bezier_curve_2` wird durch Angabe der Kontrollpunkte und des Parameterintervalls mit dem Kontruktor `Bezier_curve_2(const Points & contr_pts, NumType a=LOW_BOUND, NumType b=UP_BOUND)` erzeugt. Wenn das Intervall $[a, b]$ nicht explizit angegeben ist, werden $a = 0$ und $b = 1$ angenommen. Desweiteren existieren ein Copy- und ein parameterloser Konstruktor. Neben den von den Oberklassen `Parametric_curve_2` und `Subdivision_curve_2` geerbten Methoden verfügt die Klasse über folgende Methoden:

- `Point deCasteljau(NumType t)` - setzt den de Casteljau-Algorithmus zur Berechnung eines Punktes auf der Bézier-Kurve zum gegebenen Parameterwert `t` ein.

- `Point horner(NumType t)` - setzt den Horner-Algorithmus zur Berechnung eines Punktes auf der Bézier-Kurve zum gegebenen Parameterwert `t` ein.

- `void subdivide(Points& l, Points& r, NumType m=0.5)` - unterteilt diese Bézier-Kurve in eine linke und eine rechte Bézier-Kurve, die den gleichen Grad wie die Originalkurve haben. Für die Unterteilung wird der de Casteljau-Algorithmus für `t=m` ausgeführt (siehe die Subdivisionformel 4.8). Die dabei abgelesenen Kontrollpunkte der Teilkurven werden in den Vektoren `l` bzw. `r` zurückgegeben.

- `void initial_approximation(unsigned num=NUMBER)` - erzeugt einen groben linearen Approximant dieser Bézier-Kurve, der aus den mit Liniensegmenten verbundenen Kontrollpunkten besteht. Außerdem wird die Toleranz `eps` mit $\max_{i=1\ldots n-1}(dist(P_i, \overline{P_0 P_n}))$ initializiert. Da eine Bézier-Kurve in der konvexen Hülle ihrer Kontrollpunkte liegt (siehe Eigenschaften), ist die Approximationsbedingung 3.9 für den so ermittelten Toleranzwert `eps` erfüllt. Diese Methode überschreibt die gleichnamige Methode in der Klasse `Parametric_curve_2`

- `bool within_tolerance(NumType epsilon=EPSILON)` - gibt `true` zurück, falls diese Bézier-Kurve durch den Liniensegmentsegment, der ihre Endkontrollpunkte verbindet, zur geforderten Toleranz `epsilon` approximiert werden kann, d. h. es gilt
$\forall i = 1 \ldots n-1 :\ dist(P_i, \overline{P_0 P_n}) <$ `epsilon` (siehe Abschnitt 4.2).

- `void rendering(Points& points, NumType epsilon=EPSILON)` - implementiert den Approximationalgorithmus 4.2.1. Die Stützpunkte des durch rekursive Subdivision erzeugten stückweise linearen Approximanten dieser Bézier-Kurve werden im Punktenvektor `points` zurückgegeben.

- `void rendering_n_times(Points& points, unsigned n)` - führt `n` rekursive Unterteilungen der Bézier-Kurve durch und gibt die Unterteilungspunkte im Vektor `points` zurück.

- `void refine_approximation(NumType epsilon=EPSILON, bool use_deriv=true)` - verfeinert den stückweise linearen Approximant dieser Bézier-Kurve mittels Unterteilungen bis die Bedingung 3.9 für den geforderten Toleranzwert `epsilon` erfüllt ist. Der Parameter `use_deriv` erlaubt zwischen den beiden im Abschnitt 4.2 vorgestellten Unterteilungsmethoden zu wählen. Bei `use_deriv=true` wird anhand der Formel 4.12 die Anzahl der notwendingen rekursiven Unterteilungen berechnet und damit die Methode `rendering_n_times(Points& points, unsigned n)` aufgerufen. Bei `false` wird `rendering(Points& points, NumType epsilon=EPSILON)` eingesetzt.

- `unsigned sufficient_subdiv_level(NumType epsilon=EPSILON)` - berechnet nach der Wang-Formel (siehe 4.12) die minimale Anzahl rekursive Unterteilungen der Bézier-Kurve, die notwendig sind, um diese Kurve zur geforderten Toleranz `epsilon` zu approximieren.

- `bool derivative(Bezier_curve_2& deriv, unsigned r=1)` - ermittelt die r-te Ableitung dieser Bézier-Kurve, die wiederum eine Bézier-Kurve vom Grad $(n - r)$ ist (siehe Abschnitt 4.3) und gibt diese in `deriv` zurück. Die Methode gibt `true` zurück, falls die r-te Ableitung existiert, `false` sonst.

- `bool evaluate_derivative_at(NumType t, Vector& deriv, unsigned r=DEGREE)` - wertet die r-te Ableitung dieser Bézier-Kurve zu gegebenem Parameter `t` aus und gibt den Wert im Vektor `deriv` zuück. Die Methode gibt `true` zurück, falls die r-te Ableitung existiert, `false` sonst.

- `void draw()` - zeichnet den stückweise linearen Approximant und das Kontrollpolygon dieser Bézier-Kurve.

- `Bbox bbox()` - gibt die Bounding-Box des Kontrollpolygons dieser Bézier-Kurve zurück.

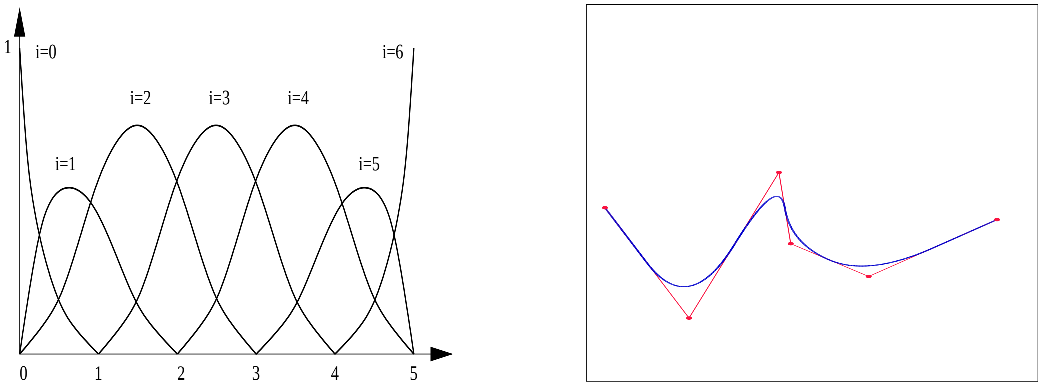
Kapitel 5

B-Splines

Bei den im Kapitel 4 vorgestellten Bézier-Kurven werden anhand von Stützpunkten $P_0, \ldots, P_n$ glatte Kurven konstruiert, die ihr Kontrollpolygon durch einen Polynom des Grades n approximieren. Je komplizierter die anzunähernde Form ist, desto mehr Kontrollpunkte werden benötigt und der Grad der Kurve wächst. Da die Auswertung von Polynomen hohen Grades den Rechenaufwand erhöht und nummerisch unstabil ist, ist die feste Koppelung des Polynomgrades an die Anzahl Kontrollpunkte vom großen Nachteil. Ein weiteres Problem bei Bézier-Kurven ist der globale Einfluß iherer Kontrollpunkte auf den Kurvenverlauf (siehe Eigenschaften im Kapitel 4). *Splines* sind stückweise polynomiale Kurven, deren Teilkurven glatt in einander übergehen. Der Grad der Polynome ist nicht mehr von der Anzahl Stützpunkte abhängig und kann niedrig gehalten werden. Das erlaubt komplizierte Formen, die durch eine große Menge Kontrollpunkte gegeben sind, effizient zu approximieren. Die *Bézier-Spline-Kurven* stellen eine Möglichkeit dar, Splines zu konstruieren. Dabei werden Bézier-Kurven niedrigem Grade zu einer glatten Kurve verbunden. Der Nachteil dieser Methode ist, dass die Forderung, die Kurvenübergänge glatt zu halten, zu Restriktionen bezüglich der Position der Kontrollpukte führt (siehe z.B. [Gol]). In diesem Kapitel werden die *Basis-Spline-Kurven* kurz *B-Splines* behandelt. Bei ihnen wird eine beliebige Menge von Punkten durch Teilkurven niedrigem Grade, die glatt in einander übergehen, approximiert. Ein weiterer Vorteil der B-Spline-Kurven ist, dass Änderungen von Kontrollpunkten den Kurvenverlauf nur lokal beeinflüßen. Die explizite Definition einer *B-Spline-Kurve vom Grad d* ist durch die Gleichung

$$s(t) = \sum_{i=0}^{n} N_i^d(t) P_i, \ n \geq d, \ t \in [t_d, t_{n+1}] \subset \mathbb{R} \tag{5.1}$$

gegeben (siehe z.B. [JH89]). Dabei sind $P_0, \ldots, P_n$ die Kontrollpunkte der Kurve (auch *de Boor-Punkte* genannt). Die Parameterwerte $t_0 \leq t_1 \leq \ldots \leq$



a) Die B-Spline-Funktionen N_i^2 b) B-Spline-Kurve d=2, n=5

Abbildung 5.1: B-Spline-Funktionen und B-Spline-Kurve vom Grad 2 mit sechs Kontrollpunkten und den Knotenvektor $\{0, 0, 0, 1, 2, 3, 4, 5, 5, 5\}$

t_{n+d+1} werden als *Knoten* oder *Trägerwerte* bezeichnet. Sind die Knoten gleichmäßig über dem Trägerintervall verteilt, so spricht man von *uniformen B-Splines*. Ein Beispiel für uniforme B-Spline-Kurven ist in Abb. 5.1b gegeben. $N_i^d(t)$ sind die *normalisierten B-Spline-Funktionen vom Grad d*, die durch folgende Rekursionsgleichung definiert sind:

$$
\begin{aligned}
N_i^0(t) &= \begin{cases} 1 & t \in [t_i, t_{i+1}) \\ 0 & otherwise \end{cases} \\
N_i^d(t) &= \tfrac{t - t_i}{t_{i+d} - t_i} N_i^{d-1}(t) + \tfrac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} N_{i+1}^{d-1}(t), \\
1 &\leq d \leq n,\ 0 \leq i < n - d
\end{aligned}
\tag{5.2}
$$

Ählich wie bei den Bézier-Kurven bestimmen die algebraischen Eigenschaften der B-Spline-Funktionen das geometrische Verhalten der B-Spline-Kurven. Diese Zusammenhänge werden in [JH89] ausführlich behandelt. Hier werden die für die Zwecke dieser Arbeit wichtigen Eigenschaften der B-Splines aufgezählt:

- *Affine Invarianz*
  Für die B-Spline-Funktionen im Intervall $[t_d, t_{n+1}]$ gilt

  $$\forall t \in [t_d, t_{n+1}] : \sum_{i=0}^{n} N_i^d(t) \equiv 1. \tag{5.3}$$

  Dann sind alle Punkte $s(t)$ auf der Kurve affine Kombinationen und die B-Spline-Kurve ist invariant bezüglich affiner Transformationen,

d. h. sie kann unabhängig von einem Koordinatensystem betrachtet werden.

- *lokaler Einfluß der Kontrollpunkte auf dem Kurvenverlauf*
Im Gegensatz zu den Bernsteinpolynomen, die im gesammten Definitionsintervall größer Null sind, kann für die B-Spline-Funktionen

$$\begin{array}{ll} N_i^d(t) > 0 & t \in (t_i, t_{i+d+1}) \\ N_i^d(t) = 0 & otherwise \end{array} \tag{5.4}$$

gezeigt werden. Mit Bezug auf der Definitionsgleichung 5.1 folgt, dass der Kontrollpunkt P_i nur den Verlauf der nächsten $d+1$ Kurvensegmente beeinflußt. Darin besteht ein weiterer Unterschied zwischen den Bézier-Kurven und den B-Splines (siehe Abb. 5.2).

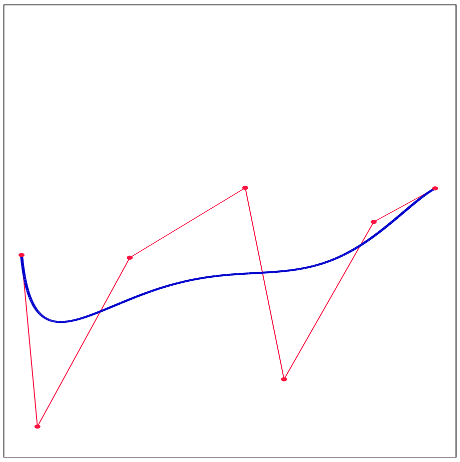
- *Jedes Kurvensegment einer B-Spline-Kurve liegt in der konvexen Hülle seiner $d+1$ Kontrollpunkte*
Diese strenge Konvexhülleneigenschaft folgt aus Gleichungen 5.3 und 5.4. Da die konvexe Hülle der B-Spline-Kurve die Vereinigung der konvexen Hüllen ihrer Teilsegmente darstellt, gilt auch die globale Form dieser Eigenschaft.

- *Verhalten in den Endpunkten*
Im Unterschied zu den Bézier-Kurven interpolieren die B-Spline-Kurven im allegemeinen Fall nicht ihre Endkontrollpunkte. Dies ist, wie in [SA91] gezeigt, durch folgende Wahl der Knoten zu erreichen:

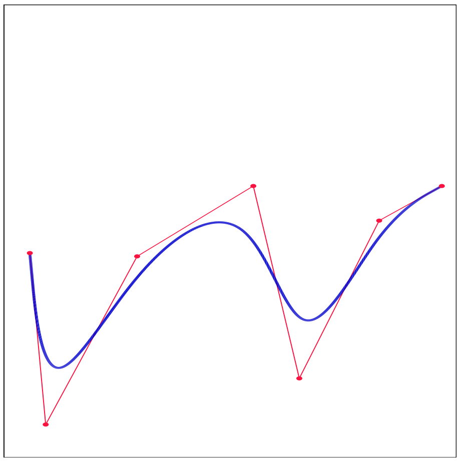
$$\{\underbrace{t_0, \ldots, t_0}_{d+1}, t_1, \ldots, t_{n-1}, \underbrace{t_n, \ldots, t_n}_{d+1}\} \tag{5.5}$$

- *$C^{d-\mu}$-Stetigkeit an μ-fachen Knoten*
Wie in [SA91] gezeigt, sind die Funktionen N_i^d bei paarweise unterschiedlichen Knoten t_i $(d-1)$-mal stetig differenzierbar. Das führt zu glatten Übergängen der B-Spline-Segmente in einander. Das μ-fache Auftreten eines Knotens t_i, verringert die Differenzierbarkeit der B-Spline-Funktion N_i^d an dieser Stelle um das Vielfache μ. Ist $\mu = d$, so interpoliert die B-Spline-Kurve den Kontrollpunkt P_i (siehe [Gol]).

- *Beschränkte Schwankung (variation diminishing property)*
Änlich wie bei Bézier-Kurven kann eine Gerade eine B-Spline-Kurve nicht öfter als ihr Kontrollpolygon schneiden.



a) Bézier-Kurve vom Grad 6



b) B-Spline-Kurve vom Grad 3

Abbildung 5.2: Eine Bézier-Kurve und eine B-Spline mit gleichem Kontrollpunktsatz

Neben der durch die Definitionsgleichung 5.1 gegebenen Auswertungsformel, können Punkte auf B-Spline-Kurven auch nach dem *de Boor*-Algorithmus berechnet werden (siehe z.B. [Gol]). Dieser im Jahre 1972 von C. de Boor veröffentlichte Algorithmus stellt eine Verallgemeinerung des de Casteljau-Algorithmus dar. Folglich sind die Bézier-Kurven ein Sonderfall von B-Spline-Kurven. Dies kann auch durch folgende Wahl des Knotenvektors

$$\{\underbrace{a, \ldots, a}_{d+1}, \underbrace{b, \ldots, b}_{d+1}\}$$

gezeigt werden (siehe [JH89]). Dadurch wird die Rekursion 5.2 für die B-Spline-Funktionen in Gleichung 4.2 umgewandelt, mit der die Bernsteinpolynome definiert sind, d. h. es gilt $N_i^d = B_i^d$. Abbildung 5.2 zeigt den unterschiedlichen Kurvenverlauf bei einer Bézier-Kurve und einer B-Spline-Kurve, die über die gleichen Kontrollpunkte definiert sind.

5.1 Auswertungsalgorithmus

Die Auswertung der B-Spline-Kurve $s(t)$ durch die mit Gleichung 5.1 gegebenen Definitionsformel ist nicht effizient und für praktische Zwecke ungeeignet. Eine effizientere Auswertungsmöglichkeit bietet der im Folgenden vorgestellte Algorithmus von de Boor. Der Zusammenhang zwischen diesem

Algorithmus und der Definition der B-Spline-Kurven mittels den B-Spline-Funktionen wird in [JH89] ausführlich erläutert. Sei $t \in [t_k, t_{k+1}] \subset [t_0, t_n]$ der Parameterwert, an dem die B-Spline-Kurve vom Grad d ausgewertet werden soll, dann gilt

$$s(t) = s_k^d(t), \tag{5.6}$$

wobei $s_k^d(t)$ rekursiv durch

$$s_i^r(t) = \begin{cases} P_i & r = 0 \\ \frac{t_{i+d-r+1}-t}{t_{i+d-r+1}-t_i} s_i^{r-1}(t) + \frac{t-t_i}{t_{i+d-r+1}-t_i} s_{i+1}^{r-1}(t) & 1 \leq r \leq d \end{cases} \tag{5.7}$$

definiert ist (siehe z.B. [Far88]). Diese Rekursion liegt dem de Boor-Algorithmus zugrunde, dessen Pseudo-Code in Alg. 5.1.1 angegegeben ist. Die Laufzeit des

de_Boor$(\{P_i\}, \{t_j\}, d, t)$
Require: $P_0, \ldots, P_n$: the control points of this b-spline curve,
  $t_0, \ldots, t_n$: the knot sequence of this b-spline curve,
  d - the degree of this b-spline, $t \in [t_0, t_n]$
Ensure: s_0 is the point on this b-spline curve at t
  find t_k such that $t \in [t_k, t_{k+1})$
  $j := k - \lfloor \frac{d}{2} \rfloor$ - the index of the first needed control point
  for $i := 0$ to d **do**
  $$s_i := \begin{cases} P_0 & j+i < 0 \\ P_n & j+i > n \\ P_{j+i} & otherwise \end{cases}$$
  end for
  $j := k - d$ - the index of the first needed knot
  for $i := 0$ to $2d$ **do**
  $$u_i := \begin{cases} t_0 & j+i < 0 \\ t_n & j+i > n \\ t_{j+i} & otherwise \end{cases}$$
  end for
  for $r := d - 1$ down to 0 **do**
    for $i := 0$ to r **do**
      if $u_{i+d} - u_{i+d-j} \neq 0$ **then**
        $s_i := \frac{(u_{i+d}-t)s_i+(t-u_{i+d-j})s_{i+1}}{u_{i+d}-u_{i+d-j}}$
      end if
    end for
  end for

Algorithmus 5.1.1: De Boor-Algoritmus

Algorithmus ist $O(d^2 + n)$, d. h. quadratisch in dem üblicherweise kleinen

Grad der B-Spline-Kurve und linear in der Anzahl Kontrollpunkte. Im Vergleich zu dem de Casteljau-Algorithmus, dessen Laufzeit $O(n^2)$ ist, wird der Vorteil der B-Splines gegenüber den Bézier-Kurven deutlich. Die Entkoppelung des Kurvengrades von der Anzahl Kontrollpunkte erlaubt es, auch bei erhöhter Anzahl Kontrollpunkten, den Auswertungsaufwand gering zu halten.
Der de Boor-Algorithmus ist im Funktionsobjekt `DeBoor_FO` implementiert. Der Kontruktor dieser Klasse bekommt als Parameter die Pointer zum Kontrollpunktenvektor und zum Knotenvektor, sowie den Grad der B-Spline-Kurve.

5.2 Approximation durch Subdivision

Ähnlich wie bei Bézier-Kurven und bei allgemeinen parametrischen Kurven ist die Konstruktion eines stückweise linearen Approximanten, der die B-Spline-Kurve zu einer gegebenen Toleranz ϵ annähert, von besonderer Bedeutung. Methoden, wie Zeichnen der Kurve und Bestimmen der Schnittpunkte mit anderen Kurven, arbeiten dann mit dem Approximanten anstatt mit der Kurve. Eine grobe Approximation der B-Spline-Kurve stellt ihr Kontrollpolygon dar. Anhand der Konvexhülleneigenschaft kann ähnlich wie bei Bézier-Kurven eine obere Schranke für die zugehörige Toleranz $\epsilon < \max_{i=1 \ldots n-1}(dist(P_i, \overline{P_0 P_n}))$ ermittelt werden. Für die meisten Anwendungen ist diese Approximation zu grob. Eine Möglichkeit, das Kontrollpolygon zu verfeinern und dadurch eine engere Approximation der Kurve zu erreichen, stellt das *Einfügen von Knoten (knot insertion)* dar. Diese Methode wird ausführlich in [Gol] behandelt und wird hier nur kurz vorgestellt. Sei $s(t)$ eine B-Spline-Kurve vom Grad d, die über den Kontrollpunkten $\{P_i\}$ und der Knotensequenz $\{t_j\}$ definiert ist (siehe 5.1). Durch Einfügen von neuen Werten in den Knotenvektor $\{t_j\}$ entsteht eine verfeinerte Knotensequenz $\{u_k\} : \{t_j\} \subset \{u_k\}$. Gesucht wird ein neuer Kontrollpunktensatz $\{Q_l\}$, der die B-Spline-Kurve $s(t)$ über die neue Knotensequenz $\{u_k\}$ definiert und dessen Kontrollpolygon eine bessere Annäherung an die B-Spline-Kurve darstellt. In [Gol] wird gezeigt, dass für dieses Problem immer eine Lösung existiert. Zur Ermittlung der neuen Kontrollpunkte kann der *Algorithmus von Böhm* eingesetzt werden. Bei diesem Algorithmus kann pro Durchlauf ein Knoten τ μ-fach ($\mu \leq d$) in das Intervall $[t_k, t_{k+1}] : t_k < \tau \leq t_{k+1}$ eingefügt werden. Bei dieser Einfügeoperation werden nur die $d+1$ Kontrollpunkte des k-ten Kurven-Segments, d. h. $P_{k-\lfloor \frac{d}{2} \rfloor} \ldots P_{k-\lceil \frac{d}{2} \rceil}$ betroffen. Sie werden durch

$d+1+\mu$ neue Kontrollpunkte Q_j ersetzt, die wie folgt definiert sind:

$$Q_j = \begin{cases} s_0^j & 0 \leq j \leq \mu \\ s_{j-\mu}^{\mu} & \mu < j < d \\ s_{j-\mu}^{\mu-(j-d)} & d \leq j \leq d+\mu, \end{cases} \tag{5.8}$$

wobei die Terme $s_j^r(t)$ mit der de Boor-Rekursionsgeichung 5.7 gegeben sind. Also nach μ Niveaus des de Boor-Algorithmus kann der neue Kontrollpunktensatz $\{Q_i\}$ bestimmt werden. Führt man diese Einfügeoperationen wiederholt aus, konvergiert die Folge der erzeugten Kontrollpolygone zu der B-Spline-Kurve $s(t)$ (siehe [Far88]). Der Nachteil des Algorithmus von Böhm ist, dass pro Durchlauf nur ein Knoten eingefügt werden kann und folglich nur das Kontrollpolygon zwischen den Punkten $P_{k-\lfloor \frac{d}{2} \rfloor}$ und $P_{k+\lceil \frac{d}{2} \rceil}$ verfeinert wird.
Ein anderer Ansatz, der bei jedem Schritt das ganze Kontrollpolygon modifiziert ist *Subdivision*. Diese Methode wird gründlich in [JW02] behandelt. Sei p^0 das Kontrollpolygon einer uniformen B-Spline-Kurve vom Grad d definiert mit den Kontrollpunkten $\{P_0, \ldots, P_n\}$ über der Knotensequenz $\{t_0, \ldots, t_n\}$. Bei jedem Subdivision-Schritt werden gleichzeitig n Knoten $\tau = \frac{t_i + t_{i+1}}{2}$ in die Mitte jedes Intervalls $[t_i, t_{i+1}]$ eingefügt. Das erzeugt ein verfeinertes Kontrollpolygon p^1 über die Knotensequenz $\{t_0, \frac{t_0+t_1}{2}, t_1 \ldots, \frac{t_{n-1}+t_n}{2}, t_n\}$, das eine bessere Approximation der B-Spline-Kurve darstellt. Die Unterteilung für uniforme B-Splines vom Grad d erfolgt nach dem *Subdivision-Algorithmus von Lane und Riesenfeld*. Danach besteht ein Subdivision-Schritt aus:

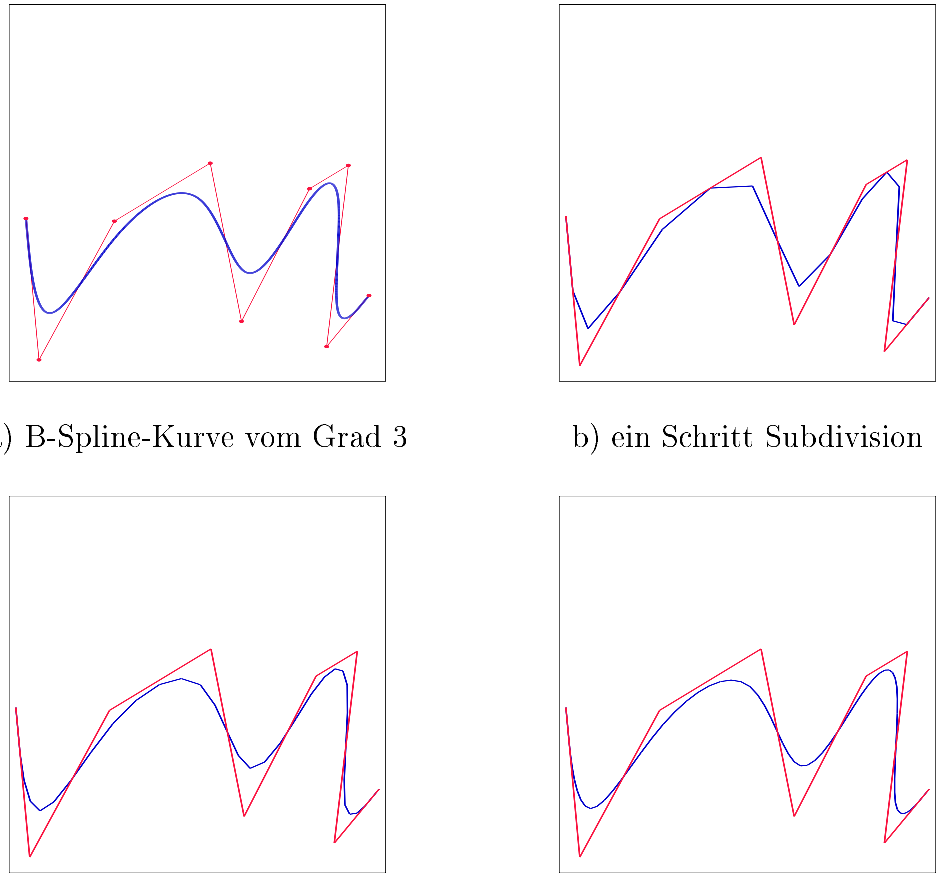
- Duplizieren von jedem Kontrollpunkt, so dass die Kontrollsequenz über den feineren Raster $\{t_0, \frac{t_0+t_1}{2}, t_1 \ldots \frac{t_{n-1}+t_n}{2}, t_n\}$ die Form $\{P_0, P_0, P_1, P_1 \ldots, P_n, P_n\}$ hat.

- d Mal die neue Kontrollsequenz durchlaufen und dabei jeden Punkt P_i durch den Mittelpunkt $\frac{P_i + P_{i+1}}{2}$ ersetzen.

In [JW02] wird gezeigt, dass die durch wiederholte Subdivision erzeugte Folge von Kontrollpolygonen $p^1, \ldots, p^k$ quadratisch zu der B-Spline-Kurve $s(t)$ konvergiert. In [DL] wird eine obere Schranke für die Abweichung einer uniformen B-Spline-Kurve von ihrem Kontrollpolygon ermittelt, die nur vom Grad des B-Splines und den zweiten Differenzen der Kontrollpunkte abhängig ist. Mit Bezug darauf ist die maximale Abweichung wie folgt beschränkt:

$$\max |s(t) - p^0| \leq \frac{(d+1)}{24} \max_i |\triangle^2 P_i|. \tag{5.9}$$

Seien $\triangle^2 P_i^k$ die zweiten Diffrenzen des durch k Subdivision-Schritte verfeinerten Kontrollpolygons p^k, dann wird folgender Zusammenhang zu den zweiten



a) B-Spline-Kurve vom Grad 3

b) ein Schritt Subdivision

c) zwei Schritte Subdivision

d) drei Schritte Subdivision

Abbildung 5.3: Verfeinerung des Kontrollpolygons einer uniformen B-Spline-Kurve vom Grad 3 durch wiederholte Anwendung von Subdivision

Differenzen des Origalkontrollpolygons p^0 gezeigt:

$$\max_i | \triangle^2 P_i^k | = \frac{1}{4^k} \max_i | \triangle^2 P_i | \tag{5.10}$$

Anhand von diesem Ergebnis und der mit Gleichung 5.9 angegebenen oberen Schranke der Abweichung wird eine Abschätzung der Anzahl Subdivision-Aufrufe abgeleitet, die für die Approximation einer uniformen B-Spline-Kurve durch das verfeinerte Kontrollpolygon p^k innerhalb einer gegebenen Toleranz ϵ benötigt werden. Danach gilt:

$$k = \left\lceil \log_4 \tfrac{(d+1) \max_i |\triangle^2 P_i|}{24\epsilon} \right\rceil \tag{5.11}$$

Der für diese Arbeit implementierte Approximationsansatz berechnet nach Gleichung 5.11 die benötigte Anzahl Subdivision-Schritte und führt dem entsprechend oft den Algorithmus von Lane und Riesenfeld aus. Dadurch wird ein stückweise linearer Approximant der B-Spline-Kurve erzeugt, der die Bedingung 3.9 erfüllt, und somit die Korrektheit der auf Apprximationsbasis arbeitenden Methoden garantiert.

5.3 Ableitung

Die r-te Ableitung einer B-Spline-Kurve ist durch folgende Formel gegeben (siehe z.B. [JH89]):

$$
\begin{aligned}
s^{(r)}(t) &= d(d-1)\dots(d-r+1)\textstyle\sum_{i=0}^{n-r} P_i^{(r)} N_i^{d-r}(t), \\
P_i^{(r)} &= \begin{cases} P_i & r = 0 \\ \frac{P_{i+1}^{(r-1)} - P_i^{(r-1)}}{t_{i+d-r} - t_i} & 0 < r < d \end{cases}
\end{aligned}
\tag{5.12}
$$

Beim Vergleich mit der Definitionsgleichung 5.1 wird deutlich, dass die r-te Ableitung wiederum eine B-Spline-Kurve vom Grad $d-r$ und Koefizienten die dividierten Differenzen $P_i^{(r)}$ ist. Diese Tatsache ermöglicht den Einsatz vom de Boor-Algorithmus zum Auswerten der Ableitung an einem gegebenen Parameterwert t. Das Ergebnis wird anschließend mit $d(d-1)\dots(d-r+1)$ multipliziert.

5.4 Implementierung

Die Templateklasse `B_spline_curve_2` implementiert eine B-Spline-Kurve in der Ebene. Die Klasse wird mit dem Zahlentyp der Koordinaten `NumType` und dem Auswertungsalgorithmustyp `A` parametrisiert. Als Standardalgorithmus wird der de Boor-Algorithmus eingesetzt, der im Funktionsobjekt `DeBoor_FO` implementiert ist. Ähnlich wie die Klasse `Bezier_curve_2` hat `B_spline_curve_2` zwei Oberklassen - `Parametric_curve_2` und `Subdivision_curve_2` (siehe Klassediagramm auf Abb. 2.5). Diese Vererbung wird dadurch begründet, dass die mit Gleichung 5.1 definierten B-Spline-Kurven parametrische Kurven sind, die aber auch als Grenzwert einer Folge von Unterteilungen ihres Kontrollpolygons dargestellt werden können. Die Definition der Klasse `B_spline_curve_2` ist wie folgt:

```
template <class NumType, class A = DeBoor_FO<NumType> >
class B_spline_curve_2 : public Parametric_curve_2<NumType,
                                             X_FO<NumType, A*>*,
                                             Y_FO<NumType, A*>* >,
                         public Subdivision_curve_2<NumType> {...}
```

Für die Zwecke der Vererbung wird die Templateklasse `Parametric_curve_2` (siehe Kapitel 3.5) mit den Typen `X_FO<NumType, A*>*` und `Y_FO<NumType, A*>*` instanziiert. Die Funktionsobjekte `X_FO` und `Y_FO` geben die x- bzw. y-Koordinate des durch den Algorithmus `A` berechneten Punktes zurück. Dadurch können die in der Oberklasse `Parametric_curve_2` implementierten Methoden `NumType x(NumType t)`, `NumType y(NumType t)` und `Point calculate_point_at(NumType t)` direkt in der Unterklasse `B_spline_curve_2` eingesetzt werden. Eine B-Spline-Kurve ist durch ihre Kontrollpunkte, den Knotenvektor und den Grad gegeben (siehe Definition 5.1). Die Speicherung der Kontrollpunkte und die Implementierung der auf ihrer Basis arbeitenden Methoden übernimmt die Klasse `Subdivision_curve_2`, die ausführlich im Abschnitt 4.4 behandelt wurde. Der Grad der Kurve und der Knotenvektor werden als Datenfelder der Klasse `B_spline_curve_2` gespeichert. Ein Objekt der Klasse kann mit folgenden Konstruktoren erzeugt werden:

- `B_spline_curve_2(const Points& contr_pts, const ParamVector& knots, unsigned degree=3)` - erzeugt eine B-Spline-Kurve des Grades `degree`, mit Kontrollpunkten `contr_pts` und Knotenvektor `knots`.

- `B_spline_curve_2(const Points& contr_pts, unsigned degree=3, NumType a_=LOW_BOUND, NumType b_=UP_BOUND)` - konstruiert eine uniform B-Spline-Kurve vom Grad `degree` und Kontrollpunkte `contr_pts`, deren Knoten gleichmäßig über dem Intervall `[a,b]` verteilt sind.

Desweiteren sind ein parameterloser Kontruktor und ein Copy-Konstruktor implementiert. Neben den von den Oberklassen geerbten Methoden, verfügt die Klasse `B_spline_curve_2` über folgende Methoden:

- `void set_degree(unsigned d)` - setzt den Grad dieser B-Spline auf `d`. Die Änderung des Grades bei B-Spline-Kurven ist möglich, da der Grad unabhängig von der Anzahl Kontrollpunkte ist. Alerdings gilt Vorbedingung `d >` Anzahl Kontrollpunkte.

- `void set_knots(const ParamVector& knots)` - setzt den Knotenvektor dieser B-Spline-Kurve auf `knots`.

- `void initial_approximation(unsigned num=NUMBER)` - erzeugt eine grobe stückweise lineare Approximation dieser B-Spline-Kurve, die aus dem Kontrollpolygon der Kurve besteht. Desweiteren wird die Toleranz `eps` mit dem Wert $\max_{i=1\ldots n-1}(dist(P_i, \overline{P_0P_n}))$ initializiert. Für den so ermittelten Toleranzwert ist die Approximationsbedingung 3.9 erfüllt, da eine B-Spline-Kurve in der konvexen Hülle ihrer Kontrollpunkte liegt (siehe Eigenschaften). Diese Methode überschreibt die gleichnamige Methode der Klasse `Parametric_curve_2`.

- `unsigned sufficient_subdiv_level(NumType epsilon=EPSILON)` - ermittelt nach Gleichung 5.11 die Anzahl notwendiger Unterteilungen des Kontrollpolygons dieser B-Spline-Kurve, so dass das verfeinerte Kontrollpolygon die Kurve zur geforderten Toleranz `epsilon` approximiert.

- `void rendering(Points& points, NumType epsilon=EPSILON)` - implementiert den im Abschnitt 5.2 vorgestellte Approximationsansatz. Zunächst wird die Anzahl notwendiger Subdivision-Aufrufe bestimmt und dann entsprechend oft die `subdivide()`-Methode ausgeführt. Die Stützpunkte des auf diese Art erzeugten stückweise linearen Approximanten dieser B-Spline-Kurve werden im Punktenvektor `points` zurückgegeben.

- `void refine_approximation(NumType epsilon=EPSILON, bool use_deriv=true)` - verfeinert durch wiederholten Unterteilungen den stückweise linearen Approximant dieser B-Spline-Kurve bis die Bedingung 3.9 für den geforderten Toleranzwert `epsilon` erfüllt ist.

- `void subdivide(Points& pts)` - implementiert den Subdivision-Algorithmus von Lane und Riesenfeld für uniforme B-Splines (siehe Abschnitt 5.2).

- `bool derivative(B_spline_curve_2& deriv, unsigned r=1)` - ermittelt die r-te Ableitung dieser B-Spline-Kurve, die wiederum eine B-Spline-Kurve vom Grad $(d-r)$ ist (siehe Abschnitt 5.3) und gibt diese in `deriv` zurück. Die Methode gibt `true` zurück, falls die r-te Ableitung existiert, `false` sonst.

- `bool evaluate_derivative_at(NumType t, Vector& deriv, unsigned r=DEGREE)` - wertet die r-te Ableitung dieser B-Spline-Kurve zu gegebenem Parameter `t` aus und gibt den Wert im Vektor `deriv` zuück. Die Methode gibt `true` zurück, falls die r-te Ableitung existiert, `false` sonst.

- `void draw()` - zeichnet den stückweise linearen Approximant und das Kontrollpolygon dieser B-Spline-Kurve.

Kapitel 6

Ausblick

Im Rahmen dieser Arbeit wurde eine Datenstruktur zur Darstellung von parametrischen Kurven entworfen und in C++ implementiert, die die Operationen Auswerten der Kurvefunktion für einen gegebenen Parameterwert t, Zeichnen der Kurve, Bestimmen der Schnittpunkte mit einer anderen Kurve, Ermitteln und Auswerten der r-ten Ableitung und das Prädikat, liegt ein Punkt Q auf der Kurve effizient unterstützt.
Eine Kurve wurde als ein abstraktes Objekt mit unterschiedlichen Repräsentationen (implizite, parametrische oder prozedurale) betrachtet. Sind für eine abstrakte Kurve mehrere Darstellungen bekannt oder die Konvertierung zwischen den verschiedenen Repräsentationen möglich, so kann man die jenige Repräsentation auswählen, die die effizienteste Implementierung der auszuführenden Operationen bietet. Darauf basierend wurde eine flexible Klassenstruktur entworfen, die beliebig um weitere Kurvendarstellungen ergänzt werden kann.
Diese Diplomarbeit befasst sich mit allgemeinen parametrischen Kurven, Bézier-Kurven und B-Splines in der Ebene. Möglichkeiten zur Erweiterung bestehen in der Implementierung weiterer Kurvendarstellungen (z.B. von implizit definierten Kurven) oder der Anpassung an höheren Dimensionen (z.B. im 3D-Raum).
Ausgangspunkt für diese Arbeit ist eine parametrische Kurve, definiert durch die Funktionen $x(t)$ und $y(t)$ und die Parameterintervallgrenzen $[a, b]$. Zunächst wurden zwei Algorithmen zur stückweise linearer Approximation mit adaptivem Parametrisierungsschritt entworfen. Der dadurch erzeugte stückweise lineare Approximant wird zum Zeichnen der Kurve und zur Ermittlung ihrer diskreten Ableitung benutzt. Desweiteren wurden auf Basis dieser Annäherung ein Schnittpunktalgorithmus und ein Algorithmus zur Entscheidung, ob ein gegebener Punkt auf der Kurve liegt, entwickelt. Die Genauigkeit dieser Approximationsmethoden wird durch die Angabe einer Toleranz ϵ gesteuert. Das

Problem bei allgemeinen parametrischen Kurven ist die fehlende Garantie, dass die Kurve in der ϵ-Umgebung ihres stückweise linearen Approximanten liegt. Aus diesem Grund wurden in diese Arbeit zwei Spezialfälle von parametrischen Kurven - uniforme B-Splines und Bézier-Kurven einbezogen. Für diese Kurven existieren effiziente Subdivision-Methoden zur stückweise linearen Approximation, die sicherstellen, dass die maximale Abweichung der Kurve vom Approximanten in ϵ-Grenzen bleibt. Dadurch ist die Korrektheit der auf dieser Basis arbeitenden Algorithmen garantiert.Eine mögliche Fortsetzung dieser Arbeit wäre die Implementierung weiterer Kurven, die durch Angabe ihrer Kontrollpunkte und einer Subdivision-Methode dargestellt werden können. Von Interesse ist auch die Ausweitung der vorgestellten Approximationstechniken auf parametrische Flächen.
Ein wichtiger Teil der Arbeit bestand in der Implementierung der vorgestellten Datenstrukturen und Algorithmen. Dabei wurden die Konzepte der generischen Programmierung eingesetzt und die Designrichtlinien der Programmbiblithek CGAl berücksichtigt.

Danksagung

Ich möchte mich bei allen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben. An erster Stelle bei Prof. Dr. Helmut Alt für seine geduldige Betreuung, die interessante Aufgabenstellung und die idealen Arbeitsbedingungen.
Ein riesen merci beaucoup gilt Dr. Géraldin Morin für ihr persönliches Engagement und aktives Mitgestalten dieser Arbeit, für die zahlreichen Literaturhinweisen und fachlichen Ratschläge, die diese Arbeit entscheidend beeinflusst haben und nicht zuletzt für ihre freundliche Fürsorge.
Vikas Kapoor danke ich vielmals für die sofortige und unersetzliche Hilfe bei der Lösung aller möglichen Problemen, die beim Entwerfen, Programmieren und Strukturieren der Implementierung auftraten.
Ich möchte mich auch ganz herzlich bei Winfried Neun bedanken, weil er sich viele Seiten schlechtes Deutsch angetan und korrigiert hat, und mich immer wieder ermuntert hat.
Desweiteren danke ich Dr. Ralph Schultz für seine Verbesserungsvorschläge, sein entgegengebrachtes Veständnis und die flexibelsten Arbeitszeiten der Welt.
Schließlich bedanke ich mich bei meiner Familie und meinen Freunden, die mich in jeder Hinsicht unterstützt haben und eine unermüdliche Aufmunterungsquelle waren.

Literaturverzeichnis

[DL] Jörg Peters D. Lutterkort. Linear envelopes for uniform b-spline curves.

[Far88] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, Inc, 1988.

[Fel92] W. D. Fellner. *Computergrafik.* B.I. Wissenschaftsverlag, 1992.

[GM01] R. Goldman G. Morin. On the smooth convergence of subdivision and degree elevation for bézier curves. *Computer Aided Geometric Design*, 2001.

[Gol] R. Goldman. Class notes.

[JH89] D. Lasser J. Hoschek. *Grundlagen der geometrischen Datenverarbeitung.* B. G. Teubner Stuttgart, 1989.

[JW02] H. Weimer J. Warren. *Subdivision Methods for Geometric Design.* Morgan Kaufmann Publishers, 2002.

[SA91] H. Müller S. Abramowski. *Geometrisches Modellieren.* BI-Wissenschaftsverlag, 1991.