UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Rendering von Freiformflächen

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Liam Oliver Bast

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: M.Sc. Kevin Keul
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im April 2018

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	□	□

....

(Ort, Datum) (Unterschrift)

Abstract

In no other field of computer science has the hardware been evolved more quickly than in computer graphics. Therefore the GPU offers, aside from the pure rendering of triangles, a bunch of further pipeline steps that allows visualisation of other graphics objects, like freeform surfaces.

This bachelor's thesis is about the rendering of freeform surfaces, in particular bezier surfaces. For that reason an implementation for management and visualisation of bézier surfaces was created for the rendering framework of the university Koblenz (CVK). For this purpose first a triangulation was implemented and finally a tesselation of bezier surfaces with normals and texture coordinates, as well as the handling of trim curves.

Zusammenfassung

In keinem Bereich der Informatik hat sich die Hardware so rasant entwickelt, wie im Bereich der Computergraphik. Dabei bietet die GPU, neben der reinen Darstellung von Dreiecken, inzwischen auch eine Reihe weiterer Pipeline-Schritte, die auch die Darstellung von anderen graphischen Objekten, wie zum Beispiel den Freiformflächen, ermöglicht.

Diese Arbeit beschäftigt sich mit dem Rendering von Freiformflächen, insbesondere dem der Bézierflächen. Dafür wurde für das Rendering Framework der Universität Koblenz (CVK) eine entsprechende Implementierung zur Verwaltung und Darstellung von Bézierflächen erstellt. Dazu wurde zunächst die Triangulation und schließlich die Tessellierung der Bézierflächen mit Normalen und Texturkoordinaten, sowie die Behandlung von Trimmkurven erstellt.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Ziel der Arbeit . . . 1
1.2 Struktur der Arbeit . . . 2

2 Grundlagen **3**
2.1 Freiformkurven . . . 3
2.2 Freiformflächen . . . 3
2.2.1 Getrimmte Freiformflächen . . . 4
2.3 Bézier . . . 5
2.3.1 Bernsteinpolynome . . . 6
2.3.2 Bézierkurven . . . 7
2.3.3 De Casteljau-Algorithmus . . . 7
2.3.4 Eigenschaften . . . 8
2.3.5 Bézierflächen . . . 8
2.3.6 Vor- und Nachteile . . . 9
2.4 B-Splines . . . 9
2.4.1 Knotenvektor . . . 11
2.4.2 B-Spline Flächen . . . 12
2.4.3 Vor- und Nachteile . . . 13
2.5 NURBS . . . 13
2.5.1 NURBS-Flächen . . . 15
2.5.2 Vor- und Nachteile . . . 15

3 Implementierung **17**
3.1 Triangulierung . . . 17
3.2 Tesselierung . . . 21
3.3 Trimmung . . . 24

4 Ergebnisse **27**

5 Fazit **34**
5.1 Ausblick . . . 34

Abbildungsverzeichnis

1 Unterschied einer interpolierten und approximierten Kurve. [1] 3
2 Beispiel einer Freiformfläche mit Kontrollpunkten. [2] 4
3 Trimmkurven und deren Orientierung. 5
4 Bézierkurve mit Kontrollpolygon und konvexer Hülle. . . . 6
5 Graphische Darstellung des De Casteljau-Algorithmus. . . . 7
6 Graphische Darstellung des doppelten De Casteljau-Algorithmus. [3] 9
7 Konvexe Hülle für verschiedene k-Werte. [4] 11
8 Geschlossene periodische B-Spline der Ordnung k=4. (a) Kontrollpolygon: $B_1B_2B_3B_4B_5B_6B_7B_8B_1$; (b) Kontrollpolygon: $B_8B_1B_2B_3B_4B_5B_6B_7B_8B_1B_2$. [4] 12
9 Lokale Veränderung einer B-Spline durch Positionsänderung eines Kontrollpunktes. [4] 13
10 Einfluss der homogenen Koordinate eines Punktes auf den Kurvenverlauf einer NURBS. [4] 15
11 Tesselierungs Level bei quadratischen Patches. [5] 22
12 Triangulierte 3x3 Bézierfläche. 27
13 Triangulierte 3x3 Bézierfläche mit angezeigten Normalen. . . 28
14 Triangulierte 3x3 Bézierfläche mit Textur und angezeigten Normalen. 28
15 Triangulierte 3x3 Bézierfläche mit Textur. 29
16 Tesselierte 4x4 Bézierfläche. 30
17 Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe eines Intervalls. 31
18 Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe einer Distanz zu einem Punkt. 31
19 Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe eines Polygonzuges. 32
20 Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe einer Bézierkurve. 32
21 Tesselierte 4x4 Bézierfläche mit Textur und Trimmung mithilfe einer Bézierkurve. 33

1 Einleitung

Die computergraphische Darstellung von Flächen findet in vielen Gebieten Anwendung. Einige Beispiele dafür wären Werbung und Film, aber auch in der Automobilindustrie beim Karroseriebau oder auch beim Schiffsbau etc. sind solche Darstellungen nicht mehr wegzudenken. Bei der Berechnung solcher Flächen ist es neben der einfachen Triangulierung auf der CPU heute auch möglich, die Berechnung von Objekten komplett auf der GPU durchzuführen. Dies kann zum Beispiel mithilfe eines Tesselation-Shaders durchgeführt werden. Die GPU ist zwar im Allgemeinen nicht so leistungsstark wie die CPU, kann jedoch viele kleinere Operationen schneller ausführen aufgrund der höheren Parallelitätfähigkeit. Wodurch die Berechnung vieler Punkte eines Objekts wesentlich schneller ablaufen kann. Diese Objekte können dann auf dem Bildschirm *gerendert* werden. Es wird also ein zweidimensionales Bild aus einer geometrischen Beschreibung eines dreidimensionalen Modells erzeugt. [6] Es gibt verschiedene Ansätze, dies zu erreichen, so zum Beispiel mithilfe von Triangulierung, also die Zerlegung des Objekts in Dreiecke, die Tesselierung im GPU-Shader, Raytracing und einige mehr.

1.1 Ziel der Arbeit

Das Ziel dieser Bachlorarbeit ist die Implementierung einer Klasse für das Rendering-Framework CVK der Universität Koblenz zur Darstellung und Verwaltung von tesselierten Freiformflächen. Des Weiteren sollte eine Trimmung auf die berechneten Flächen angewandt werden.

Dazu sollten zunächst die Grundlagen und die Mathematik hinter den Freiformflächen erarbeitet werden, um daraufhin eine Implementierung zur Tesselierung von getrimmten Freiformflächen mit Normalen und Texturkoordinaten zu erstellen. Vordergründlich sollte sich hierbei auf die Bézierflächen konzentriert werden.

1.2 Struktur der Arbeit

Die Arbeit gliedert sich in vier weitere Kapitel. Dabei sollen in Kapitel 2 zunächst die Grundlagen erklärt werden. Hier werden für die Arbeit wichtige Begriffe, wie dem der Freiformkurve, Freiformfläche und Trimmung definiert. Danach werden kurz einige Arten der Freiformkurven vorgestellt, wobei auf die Bézierkurven und -flächen ein besonderes Augenmerk gelegt wird. Anschließend wird in Kapitel 3 die Implementierung der Klassen zur Darstellung von Freiformflächen erläutert. Hier wird sowohl eine Triangulierung der Bézierfläche als auch eine Berechnung durch Tesselierung im Shader angeführt. Im nächsten Kapitel werden die Ergebnisse dargestellt und im letzten Kapitel wird dann Fazit gezogen und es wird ein Ausblick gegeben.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen der Freiformkurven und Freiformflächen behandelt. Dabei wird vor allem auf die Bézierkurven und Bézierflächen eingegangen, da diese zur Implementierung genutzt wurden. Des Weiteren werden aber sowohl B-Splines als auch NURBS vorgestellt.

2.1 Freiformkurven

Eine *Freiformkurve* lässt sich analytisch nicht eindeutig beschreiben. Solche Kurven werden anhand von Kontrollpunkten im Raum aufgespannt und können durch verschiedene Algorithmen approximiert oder interpoliert werden. [7] Bei der Interpolation einer Kurve, wird das Polynom gesucht, mit dem niedrigsten Grad, welches durch alle Punkte verläuft. Die resultierende Kurve schneidet also alle Kontrollpunkte. Bei der Approximation hingegen beeinflussen die Kontrollpunkte den Verlauf der Kurve. Sie schneidet die Kontrollpunkte also im Allgemeinen nicht (siehe Abbildung 1). [1]

Bei den in dieser Arbeit vorgestellten Beispielen für Freiformkurven handelt es sich um Approximationen von Freiformkurven.

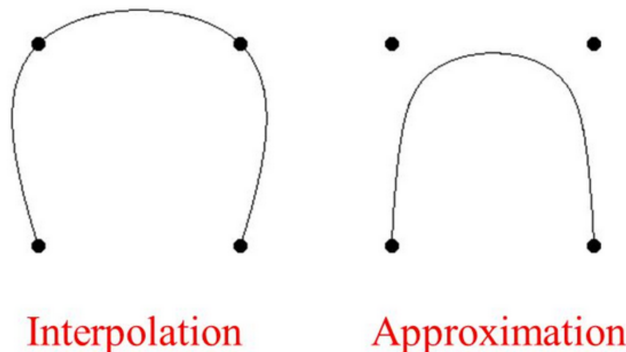

Abbildung 1: Unterschied einer interpolierten und approximierten Kurve. [1]

2.2 Freiformflächen

Die *Freiformfläche* ist eine Fläche, die sich nicht durch einfache geometrische Objekte, wie Kugel, Kegel etc. modellieren und darstellen lässt. [7] Ähnlich wie bei den Freiformkurven wird auch die Freiformfläche mithilfe von Kontrollpunkten im Raum aufgespannt. Dies geschieht aber hier in zwei Dimensionen.

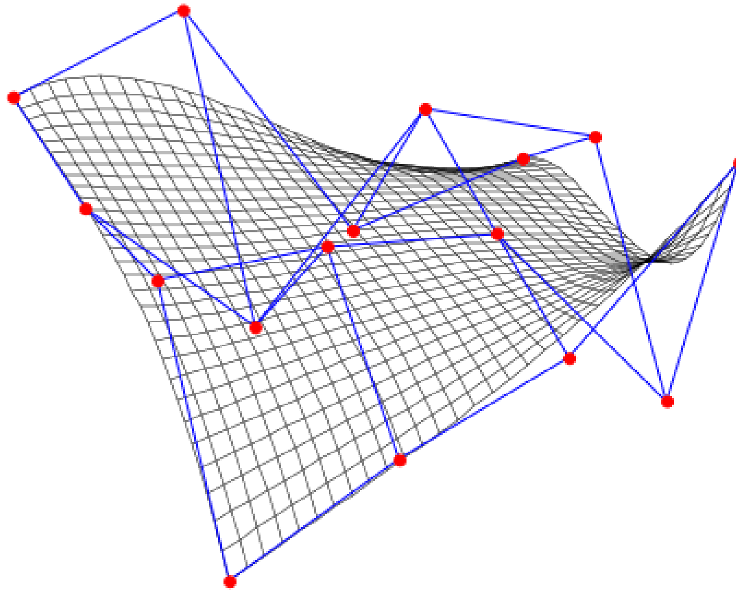

Abbildung 2: Beispiel einer Freiformfläche mit Kontrollpunkten. [2]

2.2.1 Getrimmte Freiformflächen

Die *Trimmung* einer Freiformfläche beschreibt das Herausschneiden eines Teils der Fläche. Die Fläche kann dabei sowohl durch Einschränkung des Parameterbereichs, zum Beispiel durch Angabe eines Intervalls, in dem nicht gerendert werden soll, als auch durch Angabe einer Trimmkurve auf der Fläche getrimmt werden.

Für die Trimmung mithilfe von Trimmkurven muss jedoch auf folgende Punkte geachtet werden:

- Die Trimmkurve muss geschlossen sein.
- Bei mehreren Trimmkurven dürfen sich die einzelnen Kurven nicht überschneiden.
- Die Kurven müssen konsistent orientiert sein (Abbildung 3). Eine Elternkurve sollte einen anderen Umlaufsinn haben als ihr Kind, um Innen und Außen unterscheiden zu können.

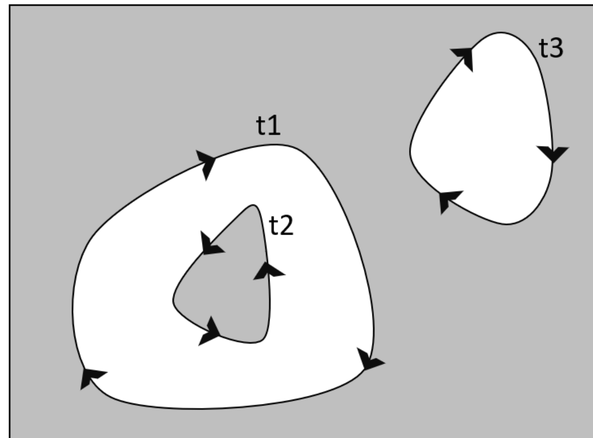

Abbildung 3: Trimmkurven und deren Orientierung.

2.3 Bézier

Die Bézierkurve und die darauf aufbauende Bézierfläche wurden Anfang der 1960er Jahre von Pierre Bézier und Paul de Casteljau entwickelt. Die Bézierkurven werden in der Computergrafik verwendet, da sie mathematisch verhältnismäßig leicht zu handhaben sind. Anwendung finden diese zum Beispiel im Computer Aided Design (CAD), bei der Erstellung von Illustrationen und Schriftarten. [8]

Eine *Bézierkurve* ist eine parametrisch modellierte Kurve, die durch Kontrollpunkte und Basisfunktionen bestimmt wird. Dabei bilden die Kontrollpunkte ein Kontrollpolygon, das die Kurve einrahmt. Als Basisfunktion werden bei der Bézierkurve *Bernsteinpolynome* verwendet. Der Verlauf der Kurve kann durch Verändern der Kontrollpunkte modifiziert werden. Punkte können dabei verschoben, entfernt oder hinzugefügt werden.
Die Anzahl der Kontrollpunkte hat jedoch Auswirkungen auf den Grad des Bernsteinpolynoms. Dieser entspricht der Anzahl der Kontrollpunkte minus eins.

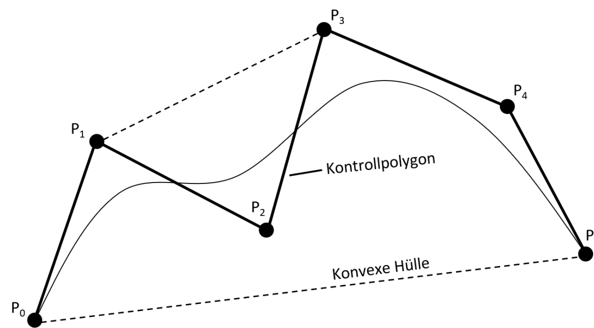

Abbildung 4: Bézierkurve mit Kontrollpolygon und konvexer Hülle.

2.3.1 Bernsteinpolynome

Das Polynom $B_{n,i}$ heißt i-tes Bernsteinpolynom vom Grad n: [9]

$$\begin{aligned} B_{n,i}(t) &= \binom{n}{i} * t^i * (1-t)^{n-i} \\ &= \frac{n!}{i! * (n-i)!} * t^i * (1-t)^{n-i} \quad (0! \equiv 1) \end{aligned}$$

Bei der Berechnung von Bézierkurven spielt die Rekursionseigenschaft der Bernsteinpolynome eine besondere Rolle, da der De Casteljau-Algorithmus darauf aufbaut. Sei $t \in [0,1]$ dann gelten:

$$B_{n,i}(t) = t * B_{n-1,i-1}(t) + (1-t) * B_{n-1,i}(t) \textit{ für } i = 1 \ldots (n-1) \tag{1}$$

$$B_{n,n}(t) = t * B_{n-1,n-1}(t) \tag{2}$$

$$B_{n,0}(t) = (1-t) * B_{n-1,0}(t) \tag{3}$$

Für ein festes $n \in \mathbb{N}$ bilden die Bernsteinpolynome $B_{n,i}$ für $i = 0 \ldots n$ eine Basis im Vektorraum $\mathbb{P}_n$ der reelen Polynome vom Grad n.

Durch diese Eigenschaft ist es möglich, jedes beliebige Polynom eines Grades n als Linearkombination von Bernsteinpolynomen darzustellen. Dadurch können Bézierkurven anhand von vorgegebenen Punkten parametrisiert werden. Weiterhin gilt, dass alle Bernsteinpolynome im Intervall [0,1] positiv sind.

2.3.2 Bézierkurven

Eine Bézierkurve wird durch $n+1$ Kontrollpunkte $P_0 \dots P_n$ spezifiziert, die den Verlauf der Kurve beeinflussen. Jeder Punkt P_i wird dabei von einem Bernsteinpolynom $B_{n,i}$ gewichtet:

$$P(t) = \sum_{i=0}^{n} P_i * B_{n,i}(t) \qquad 0 \leq t \leq 1 \tag{4}$$

Alle Bernsteinpolynome für $0 \leq t \leq 1$ sind ungleich Null, somit beeinflussen alle Kontrollpunkte in diesem Intervall den Kurvenverlauf der Bézierkurve. [10]

2.3.3 De Casteljau-Algorithmus

Der *De Casteljau-Algorithmus* basiert auf der Rekursivität der Bernsteinpolynome. Er geht davon aus, dass anhand einer Menge von Kontrollpunkten $P_0 \dots P_n$, schrittweise ein von t abhängiger Punkt P_0^n auf der Bézierkurve errechnet werden kann. Dabei beeinflussen sich immer zwei aufeinanderfolgende Punkte. Um die Bernsteinpolynome nicht aufwendig auswerten zu müssen, entwarf Paul de Casteljau ein Iterationsverfahren, welches durch das wiederholte Bilden von Teilverhältnissen der Strecken zwischen Punkten den gesuchten Punkt auf der Kurve approximiert. [10] Dies geschieht nach der folgender Formel, wobei $k = 0 \dots n$ die jeweilige Iterationstiefe angibt:

$$P(t)_i^{(k+1)} = (1-t) * P(t)_i^{(k)} + t * P(t)_{i+1}^{(k)} \tag{5}$$

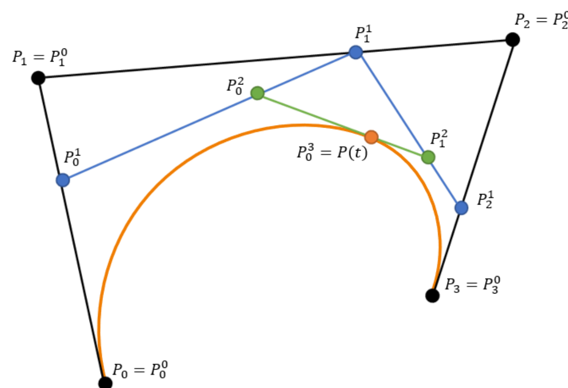

Abbildung 5: Graphische Darstellung des De Casteljau-Algorithmus.

2.3.4 Eigenschaften

- Die Basisfunktionen sind real.
- Die Summe der Basisfunktionen ist für jeden gegebenen Parameter t gleich eins:

$$\sum_{i=0}^{n} B_{n,i}(t) \equiv 1 \tag{6}$$

- Der Grad des Bernsteinpolynoms entspricht der Anzahl der Kontrollpunkte minus eins.
- Die Kurve folgt generell dem Verlauf des Kontrollpolygons.
- Der erste und letzte Kontrollpunkt liegt auf der Bézierkurve.
- Die Tangentenvektoren der Start- und Endpunkte entsprechen der Verbindung zwischen erstem und zweitem bzw. vorletztem und letztem Kontrollpunkt.
- Die Kurve wird begrenzt durch die konvexe Hülle des Kontrollpolygons; diese entspricht dem größten Polygon, welches durch verbinden der Kontrollpunkte entsteht.
- Bézierkurven sind invariant gegen affine Transformationen. Eine affine Transformation (z. B. Skalierung, Rotation, Translation) der Bézierkurve kann somit durch eine Transformation des Kontrollpolygons geschehen.
- Eine Gerade schneidet die Kurve höchstens so oft, wie sie das Kontrollpolygon schneidet.

2.3.5 Bézierflächen

Die *Bézierfläche* wird ähnlich wie die Bézierkurve berechnet, allerdings ist hier der Parameterraum zweidimensional. [7] Eine Bézierfläche des Grades (n, m) ist definiert durch:

$$P(u, v) = \sum_{i=0}^{n} \sum_{i=0}^{m} P_{i,j} * B^{u}_{n,i}(u) * B^{v}_{m,j}(v) \tag{7}$$

Dabei sind $B^{u}_{n,i}(u)$ die Basisfunktionen in u-Richtung und $B^{v}_{m,j}(v)$ die Basisfunktionen in v-Richtung. [4] Die Bézierfläche kann mittels eines doppelten De Casteljau errechnet werden. Hierbei wird dieser zunächst in u-Richtung ausgeführt und danach mithilfe der Punkte aus dem ersten Durchgang ein weiteres mal angewendet. Wie in Abbildung 6 dargestellt, werden zunächst die Bézierkurven mithilfe der Punkte 1 bis 4, 5 bis 8, 9 bis 12 und

13 bis 16 berechnet, so entstehen für jedes u vier neue Punkte. Auf diese Punkte wird dann erneut der De Casteljau angewendet, um einen Punkt auf der Fläche für die Parameter (u, v) zu errechnen.

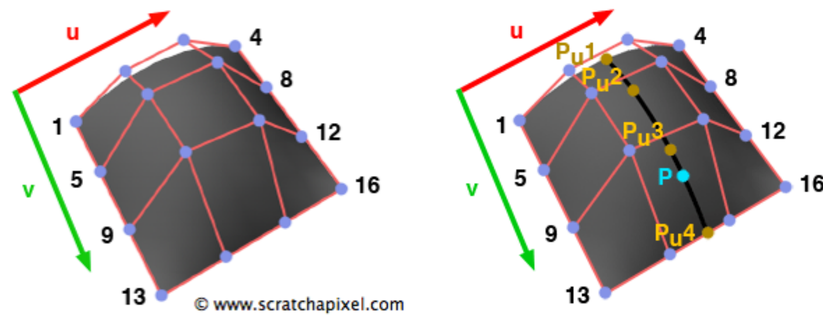

Abbildung 6: Graphische Darstellung des doppelten De Casteljau-Algorithmus. [3]

2.3.6 Vor- und Nachteile

Ein Vorteil der Bézierflächen ist, dass sie sich vergleichsweise einfach berechnen lassen. Die Fläche liegt innerhalb des Kontrollgrids und spiegelt dessen Formeigenschaften wider. Es entstehen stetige und glatte Flächen. Nachteile der Bézierflächen sind zum Einen, dass jeder Kontrollpunkt die Kurve global beeinflusst und somit jede Veränderung eines Kontrollpunktes den gesamten Verlauf der Kurve verändert; dadurch sind lokale Änderungen der Kurve kaum bis gar nicht möglich. Zum anderen gibt es keine Kontrollmöglichkeiten über die Parameterintervalle. So liegen die maximalen Einflüsse der Kontrollpunkte bei einer quadratischen Bézierfläche bei t gleich 0, 1/2 und 1.

2.4 B-Splines

In der Computergrafik werden Splines zur Darstellung von Freiformkurven und -flächen verwendet. Der Begriff *Spline* stammt aus dem Englischen und wird beim Schiffsbau verwendet. Dort bezeichnet er eine lange dünne Latte (Spline), welche an einzelnen Punkten mit sogenannten Molchen (Gewichten) fixiert wird, um die innere Spannung, die durch die Biegung entsteht, zu verringern. [7]

Erstmals wurden Splines in einer Veröffentlichung von Isaac J. Schoenberg aus dem Jahr 1946 genannt. [11]

In diesem Abschnitt werden die *B-Splines* (Basis-Splines) behandelt. Diese werden ähnlich wie die Bézierkurven durch Kontrollpunkte aufgespannt und sind ebenfalls rekursiv definiert. Die B-Spline ist eine Generalisierung der Bézierkurve. So kann mit einer B-Spline auch eine Bézierkurve dargestellt werden. Durch die Angabe eines Knotenvektors und der Ordnung des Polynoms kann der Einfluss der Kontrollpunkte beeinflusst bzw. auf ein Intervall beschränkt werden. Dadurch können durch Veränderung einzelner Kontrollpunkte auch lokale Änderungen erreicht werden. [4]

Eine B-Spline wird durch folgende Formel definiert; dabei sind P_i die Kontrollpunkte mit $i = 1 \ldots n$ und $N_{i,k}$ die normalisierte Basisfunktionen:

$$P(t) = \sum_{i=1}^{n} P_i * N_{i,k}(t) \quad t_{min} \leq t \leq t_{max}, \quad 2 \leq k \leq n \tag{8}$$

Die i-te normalisierte B-Spline Basisfunktion der Ordnung k wird mithilfe der Cox de Boor-Formel rekursiv berechnet. [4] Der daraus resultierende Spline hat den Grad $k - 1$:

$$N_{i,1}(t) = \begin{cases} 1, & \text{falls } x_i \leq t \leq x_{i+1} \\ 0, & \text{sonst} \end{cases}$$

$$N_{i,k}(t) = \frac{(t - x_i)N_{i,k-1}(t)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - t)N_{i+1,k-1}(t)}{x_{i+k} - x_{i+1}} \quad (\frac{0}{0} \equiv 0) \tag{9}$$

Eine B-Spline der Ordnung k hat mindestens k Kontrollpunkte mit $k \leq n$. Die Werte des Knotenvektors werden durch x_i repräsentiert. Für diese gilt, dass sie in aufsteigender Reihenfolge angeordnet sind $x_i \leq x_{i+1}$. Die Anzahl der Werte x_i eines Knotenvektors ist festgelegt als die Summe der Anzahl der Kontrollpunkte und der Ordnung k des Splines, also $n + k$. Die Kurve folgt dem Verlauf des Kontrollpolygons und iegt in dessen konvexer Hülle. Diese wird bei den B-Splines durch k bestimmt. Ein Punkt einer Kurve der Ordnung k liegt in der konvexen Hülle von k Nachbarpunkten (siehe Abbildung 7). [4]

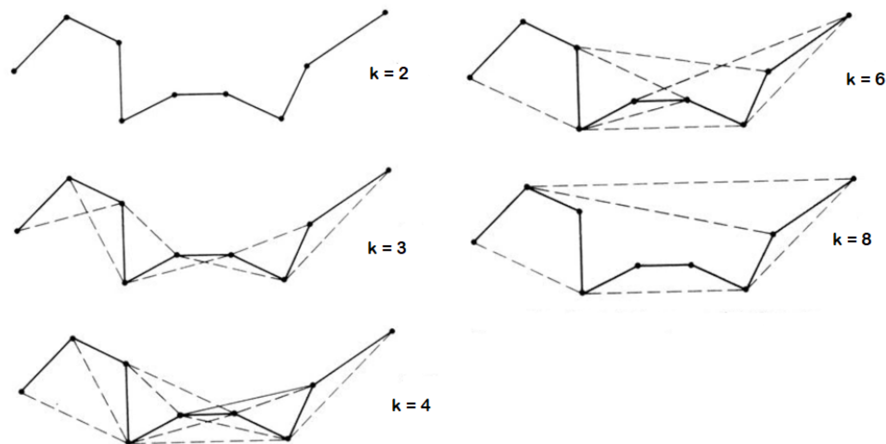

Abbildung 7: Konvexe Hülle für verschiedene k-Werte. [4]

2.4.1 Knotenvektor

Es gibt zwei verschiedene Arten von Knotenvektoren: *uniforme* und *nicht uniforme*. Dabei können diese sowohl *periodisch* als auf *offen* sein.

Ein uniformer Knotenvektor zeichnet sich dadurch aus, dass die Abstände zwischen den Werten äquidistant sind. Dies sorgt für einen gleichmäßigen Verlauf der Kurve.

$$[-0.5 \quad -0.25 \ \ 0 \ \ 0.25 \ \ 0.5]$$

Dabei wird der Knotenvektor in der Praxis meist auf das Interval [0,1] normalisiert oder die Werte werden beginnend bei Null um eins erhöht. Beispiele dafür sind folgende Knotenvektoren:

$$[0 \ \ 0.25 \ \ 0.5 \ \ 0.75 \ \ 1]$$
$$[0 \ \ 1 \ \ 2 \ \ 3 \ \ 4]$$

Ein periodisch uniformer Knotenvektor führt zu einer periodisch uniformen Basisfunktion [4] mit:

$$N_{i,k}(t) = N_{i-1,k}(t-1) = N_{i+1,k}(t+1) \tag{10}$$

Offen uniforme Knotenvektoren haben am Anfang und am Ende k gleiche Werte, dadurch wird erreicht, dass die Kurve durch den Anfangs- und Endpunkt verläuft:

$$[0 \ \ 0 \ \ 1 \ \ 2 \ \ 3 \ \ 3] \text{ für } k = 2$$

Um den Verlauf einer Bézierkurve mittels eines B-Splines zu simulieren, wird ein offen uniformer Knotenvektor verwendet. Dabei muss die Anzahl der Kontrollpunkte mit der Ordnung des B-Splines übereinstimmen, also $k = n$:

$$[0\ 0\ 0\ 1\ 1\ 1]$$

Ein nicht uniformer Knotenvektor zeichnet sich dadurch aus, dass dessen Werte entweder einen ungleichmäßigen Abstand und/oder mehrere gleiche innere Werte besitzt. Auch diese können sowohl periodisch als auch offen sein:

$[0\ 0\ 0\ 1\ 1\ 2\ 2\ 2]$	*offen*
$[0\ 1\ 2\ 2\ 3\ 4]$	*periodisch*
$[0\ 0.28\ 0.5\ 0.72\ 1]$	*periodisch*

Um eine geschlossene Kurve zu erhalten, ist die Nutzung von periodischen Knotenvektoren sinnvoll (siehe Abbildung 8). Hierfür wird eine Wiederholung von insgesamt $k-2$ Kontrollpunkten am Anfang oder Ende des Kontrollpolygons benötigt. [4]

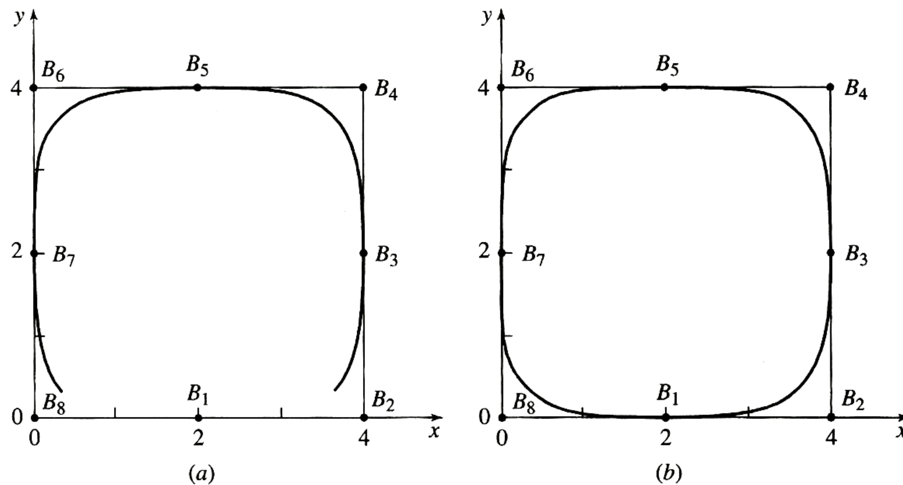

Abbildung 8: Geschlossene periodische B-Spline der Ordnung k=4. (a) Kontrollpolygon: $B_1B_2B_3B_4B_5B_6B_7B_8B_1$; (b) Kontrollpolygon: $B_8B_1B_2B_3B_4B_5B_6B_7B_8B_1B_2$. [4]

2.4.2 B-Spline Flächen

Die B-Spline Fläche wird ähnlich wie die Kurve mit folgender Formel definiert, jedoch kommt hier eine zweite Dimension hinzu. Dabei beschreiben $P_{i,j}$ die Punkte des Kontrollgrids mit $i = 1 \ldots n$ und $j = 0 \ldots m$. $N^u_{i,k}$ und $N^v_{j,l}$ definieren die B-Spline Basisfunktionen (siehe Gleichung 9) in u- bzw. v-Richtung: [4]

$$P(u,v) = \sum_{i=1}^{n} \sum_{j=1}^{m} P_{i,j} * N^u_{i,k}(u) * N^v_{j,l}(v) \tag{11}$$

Die Anzahl der Kontrollpunkte in u-Richtung sind damit $n + 1$ und in v-Richtung $m + 1$. Die Form der Fläche wird hier durch die beiden Knotenvektoren $[X]$ und $[Y]$ bestimmt, die sowohl offen oder periodisch als auch uniform oder nicht uniform sein können. Hierbei ist es nicht von Nöten gleiche Vektoren für beide Richtungen zu verwenden. [7]

2.4.3 Vor- und Nachteile

Der größte Vorteil der B-Splines gegenüber den Bézierkurven ist die Möglichkeit, mithilfe der Knotenvektoren und der Angabe des Grades der Kurve lokale Änderungen an der Kurve zu erreichen. Somit sind B-Splines im Allgemeinen flexibler als Bézierkurven, auch wenn sie dadurch etwas komplexer zu implementieren sind.

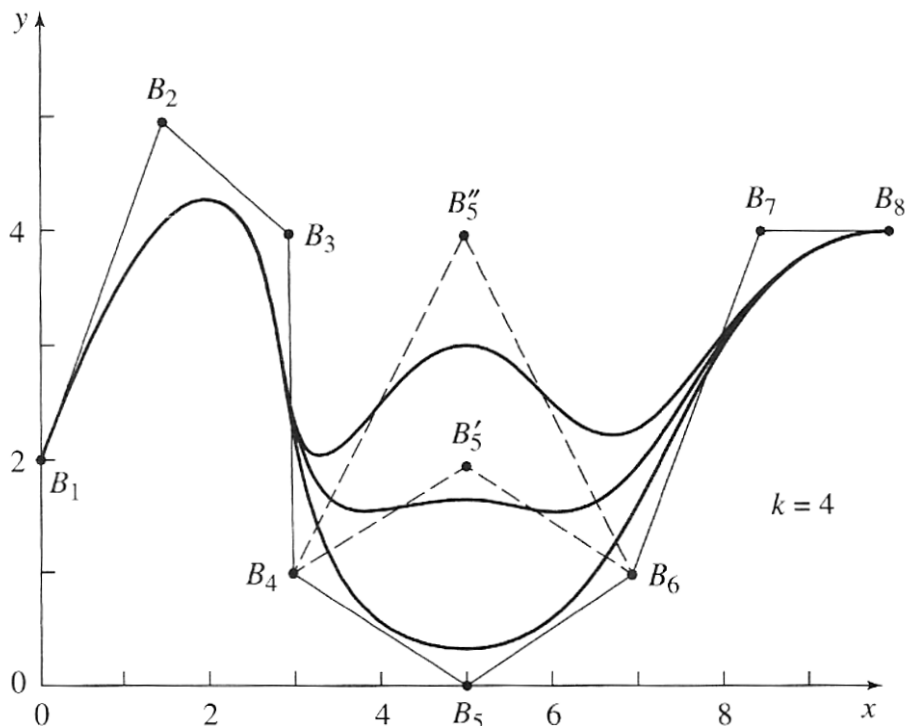

Abbildung 9: Lokale Veränderung einer B-Spline durch Positionsänderung eines Kontrollpunktes. [4]

2.5 NURBS

NURBS (Non-Uniform Rational Basis Spline) sind ein weiteres Beispiel zur Approximation von Freiformkurven und stellen eine Generalisierung von B-Splines und Bézierkurven dar. In den 1960er Jahren wurden die NURBS

von Steve Coons erstmals eingeführt. [12] NURBS sind nicht uniform, dies bedeutet, dass der Einflussbereich eines Kontrollpunktes variieren kann, was vor allem beim Modellieren von unregelmäßigen Flächen von Vorteil ist. Die Rationalität bezieht sich auf die Gleichung, welche die Kurve definiert. Diese ist nicht wie bei B-Splines und Bézierkurven durch ein einziges Summenpolynom gegeben, sondern beschreibt ein Verhältnis von zwei Polynomen. Dadurch ist es möglich, geometrische Objekte wie zum Beispiel Kreise oder Ellipsen darstellen zu können. [4]

Die Idee hinter den NURBS ist es, mithilfe einer homogenen Koordinate h die Gewichtung bzw. den Einfluss der einzelnen Kontrollpunkte zu steuern (siehe Abbildung 10). Diese Koordinate kann dabei sowohl positiv sein, dann zieht der Kontrollpunkt die Kurve an, als auch negativ, dann wird die Kurve von diesem Punkt abgestoßen. Ist $h = 0$, so hat der Punkt keinen Einfluss auf die Kurve. Geht h gegen unendlich, so wird die Kurve sich dem Punkt immer stärker annähern, sodass sie nahezu durch ihn verläuft. Um jedem Punkt eine homogene Koordinate zuzuordnen, werden die Kontrollpunkte der Kurve im vierdimensionalen Raum definiert und danach in den dreidimensionalen Raum projiziert. NURBS sind definiert durch folgende Gleichung; dabei beschreiben P_i^h die Kontrollpunkte mit der homogenen Koordinate h und $i = 1 \dots n$. $N_{i,k}$ repräsentiert die nicht rationale B-Spline Basisfunktion (siehe Gleichung 9):

$$P(t) = \sum_{i=1}^{n} P_i^h * N_{i,k}(t) \tag{12}$$

Die Projektion in den dreidimensionalen Raum geschieht durch die Division durch die homogene Koordinate:

$$P(t) = \frac{\sum_{i=1}^{n} P_i * h_i * N_{i,k}(t)}{\sum_{i=1}^{n} h_i * N_{i,k}(t)} = \sum_{i=1}^{n} P_i * R_{i,k}(t) \tag{13}$$

Dabei sind B_i die dreidimensionalen Kontrollpunkte und $R_{i,k}$ die rationalen B-Spline Funktionen mit:

$$R_{i,k}(t) = \frac{h_i * N_{i,k}(t)}{\sum_{i=1}^{n} h_i * N_{i,k}(t)} \quad h_i \geq 0 \;\; \forall i \tag{14}$$

Wenn für alle Gewichte h_i gilt $h_i = 1$, so entspricht die rationale Basisfunktion der Basisfunktion der B-Splines: $R_{i,k} = N_{i,k}$. Somit ist eine B-Spline Kurve ein Spezialfall der NURBS. Wird zudem noch ein offener Knotenvektor verwendet mit $k = n$, so erhält man eine Bézierkurve. [4]

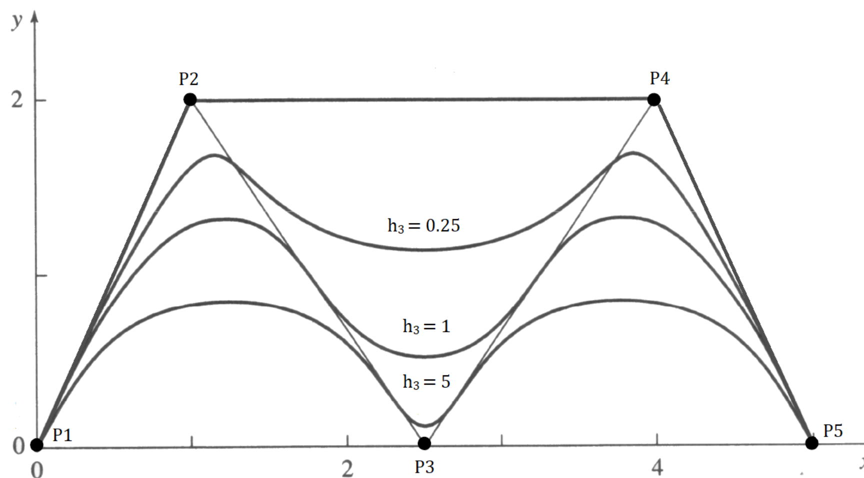

Abbildung 10: Einfluss der homogenen Koordinate eines Punktes auf den Kurvenverlauf einer NURBS. [4]

2.5.1 NURBS-Flächen

Auch die NURBS-Flächen lassen sich wieder äquivalent zu deren Kurvendefinition definieren:

$$P(u,v) = \sum_{i=1}^{n} \sum_{j=1}^{m} P_{i,j}^{h} * N_{i,k}^{u}(u) * N_{j,l}^{v}(v) \tag{15}$$

Dabei sind $P_{i,j}^{h}$ die vierdimensionalen Kontrollpunkte des Kontrollgrids mit der homogenen Koordinate h. $N_{i,k}^{u}$ und $N_{j,l}^{v}$ sind die nicht rationalen Basisfunktionen der B-Splines (siehe Gleichung 9) in u- und v-Richtung. Auch hier werden die Kontrollpunkte durch Division der homogenen Koordinate i in den dreidimensionalen Raum projiziert, wodurch die rationalen Basisfunktionen der NURBS entstehen. Die Knotenvektoren $[X]$ und $[Y]$ werden hier äquivalent zu denen der B-Splines verwendet. [4]

2.5.2 Vor- und Nachteile

Die NURBS sind von allen drei vorgestellten Freiformkurven bzw. Freiformflächen die flexibelsten. Mit ihnen ist es auch möglich, Kreise und Ellipsen darzustellen, wohingegen Bézier und B-Splines diese nur annähern können. Da NURBS eine Generalisierung von B-Splines und Bézierkurven sind, teilen sie auch die meisten Eigenschaften. NURBS sind außerdem auch invariant gegen projektive Transformationen, das bedeutet eine projektive Transformation der Kurve kann durch die projektive Transformation ihrer Kontrollpunkte erreicht werden, was weder bei B-Splines noch Bézierkurven möglich ist. Ein Nachteil der NURBS gegenüber den anderen beiden Verfahren ist es, dass sie im Vergleich schwerer zu handhaben und implementieren sind.

3 Implementierung

Die Implementierung der Bézierflächen für das CVK der Universität Koblenz wurde auf zwei verschiedene Arten angegangen. Zum einen wurde mittels Triangulierung eine Bézierfläche erzeugt, die sich an den bereits vorhandenen Objekten in der CVK Bibliothek orientiert.

Zum anderen wurde eine weitere Klasse erstellt zur Darstellung einer tesselierten Bézierfläche mithilfe eines Tesselation-Shaders. Hier wurde auch eine Trimmung der Fläche durchgeführt.

3.1 Triangulierung

Bei der Triangulierung wird die Fläche zunächst auf Anwendungsseite berechnet. Die daraus entstehenden Punkte werden dem Shader übergeben, der diese zu Dreiecken zusammenfügt und dann rendert.

Nachfolgend sind die Member-Variablen der Klasse BézierSurface aufgelistet. Diese bestehen aus einem Vektor *m_controllPoints*, der alle Punkte des Kontrollpolygons enthält und zwei Integern *m_controllU* und *m_controllV*, die die Dimensionen bzw. die Anzahl der Kontrollpunkte der Fläche in u- und v- Richtung angeben. Des Weiteren besitzt die Klasse einen dreidimensionalen Vektor *m_tangent*, der die Tangente im aktuellen Punkt angibt. Dieser wird später zur Berechnung der Normale benötigt.

```
private:
        std::vector<glm::vec3> m_controllPoints;
        int m_controllU;
        int m_controllV;
        glm::vec3 m_tangent;
```

Die Methode *deCasteljau* implementiert den De Casteljau-Algorithmus für eine Bézierkurve mit beliebig vielen Kontrollpunkten. Dabei bekommt sie einen Vektor *points* mit den Kontrollpunkten der Kurve übergeben. Die Methode berechnet dann für einen Parameter t den Punkt $P(t)$ auf der Kurve. Die bei jedem Zwischenschritt berechneten Punkte *pi* werden dem Hilfsvektor c hinzugefügt, die Anzahl der Punkte nimmt somit in jedem Durchgang um eins ab. Der Algorithmus wird so lange durchgeführt bis nur nur noch ein Punkt übrig ist; dieser wird dann zurückgegeben. Zur Berechnung der Normalen des Punktes wird die Tangente in diesem Punkt benötigt, welche durch die Strecke zwischen den beiden Punkten im vorletzten Schritt gegeben ist(siehe Abbildung 5: Punkte P_0^2 und P_1^2). Gespeichert wird sie dann in die Member-Variable *m_tangent*.

```
glm::vec3 CVK::BezierSurface::deCasteljau(std::vector<glm::vec3> points, float t)
{
        std::vector<glm::vec3> vec = points;
        std::vector<glm::vec3> c;

        while(vec.size()>1)
        {
                //compute tangent for normal
                if(vec.size()==2)
                {
                        m_tangent = vec.at(1)-vec.at(0);
```

```
                }
                c.clear();

                for(int i = 0; i < vec.size()-1; i++)
                {
                        glm::vec3 pi = (1-t) * vec[i] + t * vec[i+1];
                        c.push_back(pi);
                }

                vec.clear();
                vec = c;
        }
        return vec[0];
}
```

In der *create* Methode wird die Bézierfläche berechnet, dessen Übergabeparameter u und v dabei die Anzahl der Unterteilung in beide Richtungen angeben. Hierfür werden $u + 1$ bzw. $v + 1$ Punkte berechnet. Diese berechnen sich, indem zunächst der De Casteljau in u-Richtung durchgeführt wird. Danach wird mit den resultierenden Punkten erneut der De Casteljau durchgeführt, allerdings in v-Richtung. So entsteht für jedes Paar (u, v) ein Punkt auf der Fläche $P(u, v)$. In diesem Schritt wird auch die Tangente des Punktes in *m_tangent* gespeichert. Um die Normale generieren zu können, werden allerdings zwei Vektoren benötigt. Aus diesem Grund werden für jedes (u, v) die Punkte auf der Fläche nochmals berechnet, allerdings dieses mal zunächst in v- und dann in u-Richtung. Dadurch entsteht eine weitere Tangente, welche orthogonal zur Ersten liegt. Mithilfe des Skalarprodukts der beiden Tangenten lässt sich nun die Normale berechnen. Welche dann dann in *m_normals* aus der Klasse *CVK::Geometry* gespeichert wird.

Da das Objekt als indizierte Liste an den Shader übergeben wird, wird eine Indexliste gefüllt, die die Reihenfolge der Vertices bestimmt. Ebenfalls werden hier die UV-Koordinaten übergeben, welche für die Texturierung benötigt werden. Diese entsprechen den beiden t-Werten in u- und v-Richtung.

```
void CVK::BezierSurface::create(int u ,int v)
{

        std::vector<glm::vec3> bPoints;
        std::vector<glm::vec3> biPoints;
        glm::vec3 bTangent;

        std::vector<glm::vec3> cPoints;
        std::vector<glm::vec3> ciPoints;
        glm::vec3 cTangent;

        float tU, tV;


        for(int uTemp = 0; uTemp <= u; uTemp++){
                tU = uTemp / (float) u;

                for(int vTemp = 0; vTemp <= v; vTemp++){
                        tV = vTemp / (float) v;


                        biPoints.clear();
                        bPoints.clear();

                        ciPoints.clear();
                        cPoints.clear();

                        //De Casteljau in u-direction for a given amount (u+1) of tU values
                        for(int i = 0; i < m_controllV ;i++){
                                for(int j = 0; j < m_controllU; j++){
                                        bPoints.push_back(m_controllPoints.at(i * m_controllU + j));
```

```
                        }
                        biPoints.push_back(deCasteljau(bPoints, tU));
                        bPoints.clear();
                }
                //De Casteljau in v-direction for a given amount (v+1) of tV values
                //Important for creating normals
                for(int i = 0; i < m_controllU ;i++){
                        int off = 0;
                        for(int j = 0; j < m_controllV; j++){
                                cPoints.push_back(m_controllPoints.at(i + off));
                                off += m_controllU;
                        }
                        ciPoints.push_back(deCasteljau(cPoints, tV));
                        cPoints.clear();
                }
                //Creating a vertice on the surface for every pair(tU,tV)
                //Also calculates first tangent for normal
                m_vertices.push_back(glm::vec4(deCasteljau(biPoints, tV), 1.0f));
                bTangent = m_tangent;

                //Calculate second tangent for normal
                deCasteljau(ciPoints, tU);
                cTangent = m_tangent;

                //Calculate normal with the two tangents
                m_normals.push_back(glm::vec3(glm::normalize(glm::cross(bTangent,
                                                                        cTangent))));
                //Set UV-coordinates to tU,tV
                m_uvs.push_back(glm::vec2(tU, tV));
        }

    }

    // create index list for vertices
    int offset = 0;
    for (int j = 0; j < u; j++){
        for (int i = 0; i < v; i++){

                //first triangle
                m_index.push_back( offset + i);
                m_index.push_back( offset + i + v+1);
                m_index.push_back( offset + i + 1);

                //second Triangle
                m_index.push_back( offset + i + 1);
                m_index.push_back( offset + i + v+1);
                m_index.push_back( offset + i + v+1 + 1);

        }
        offset += v+1;
    }

    //number of indices
    m_indices = m_index.size();

    //number of points
    m_points = m_vertices.size();

    createBuffers();
}
```

Im Konstruktur wird die *create* Methode aufgerufen.

```
CVK::BezierSurface::BezierSurface(std::vector<glm::vec3> controllPoints,
                                  int controllU, int controllV, int u, int v)
{
    m_controllPoints = controllPoints;
    m_controllU = controllU;
    m_controllV = controllV;
    create(u, v);
}
```

Der leere Konstruktor generiert ein Beispiel einer 3x3 Bézierfläche mit einer Unterteilung von 20x20 (siehe Abbildung 12 bis 15).

```
CVK::BezierSurface::BezierSurface()
{
    glm::vec3 b00(-1.0f, 0.0f,-1.0f);
    glm::vec3 b10( 0.0f, 1.0f,-1.0f);
    glm::vec3 b20( 1.0f, 0.0f,-1.0f);
```

```
        glm::vec3 b01(-1.0f, 1.0f, 0.0f);
        glm::vec3 b11( 0.0f, 2.0f, 0.0f);
        glm::vec3 b21( 1.0f, 1.0f, 0.0f);

        glm::vec3 b02(-1.0f, 0.0f, 1.0f);
        glm::vec3 b12( 0.0f, 1.0f, 1.0f);
        glm::vec3 b22( 1.0f, 0.0f, 1.0f);


        m_controllPoints.push_back(b00);
        m_controllPoints.push_back(b10);
        m_controllPoints.push_back(b20);

        m_controllPoints.push_back(b01);
        m_controllPoints.push_back(b11);
        m_controllPoints.push_back(b21);

        m_controllPoints.push_back(b02);
        m_controllPoints.push_back(b12);
        m_controllPoints.push_back(b22);


        m_controllU = 3;
        m_controllV = 3;

        create(20, 20);
}
```

Aufgrund von Schwierigkeiten bei der Berechnung der Normalen wurde ein Geometry-Shader implementiert. Dieser ermöglicht das visuelle Anzeigen der Normalen (Abbildung 13 und 14).

```
void main(void){

        mat4 MVP = projectionMatrix * viewMatrix * modelMatrix;

        for(int i = 0; i < gl_in.length(); ++i){

                passColor = vec4(0.0f,0.0f,1.0f,1.0f);
                gl_Position = MVP *gl_in[i].gl_Position;
                EmitVertex();

                passColor = vec4(0.0f,0.0f,1.0f,1.0f);
                mat3 NormalMatrix = mat3(transpose(inverse(viewMatrix*modelMatrix)));
                vec4 n = vec4(normalize(NormalMatrix * passNormal[i]), 0.0f);
                gl_Position = projectionMatrix * (viewMatrix * modelMatrix
                                                * gl_in[i].gl_Position + n * scale);
                EmitVertex();
                EndPrimitive();
        }
}
```

Der Aufruf der Bézierfläche in der Applikation geschieht genau wie der der anderen Objekte des CVKs. Da die Klasse *BezierSurface* von der Oberklasse aller Objekte *CVK::Geometry* erbt, kann sie einem *CVK::Node* hinzugefügt und durch diesen auch gerendert werden.

```
        scene_node = new CVK::Node("Scene");

        CVK::BezierSurface *bezier = new CVK::BezierSurface();

        CVK::Node *bezier_node_up = new CVK::Node("bezier_up");
        bezier_node_up->setModelMatrix( glm::translate(glm::mat4( 1.0f), glm::vec3( 0, -1, 0)));
        bezier_node_up->setMaterial(mat_brick);
        bezier_node_up->setGeometry(bezier);
        scene_node->addChild(bezier_node_up);
```

3.2 Tesselierung

Bei der Tesselierung werden im Gegensatz zur Triangulierung nur die Kontrollpunkte an den Shader übergeben. Die Fläche wird dann innerhalb des Shaders berechnet; damit geschieht die Berechnung ausschließlich auf der GPU.

Die Klasse *BezierSurfaceTesselation* benötigt somit nur einen Vektor für die Kontrollpunkte *m_controllPoints* als Member-Variable. Der Vektor *m_trimPoints* und *m_bezier* der Klasse *BezierCurve* werden später bei der Trimmung der Fläche benötigt.

```
        std::vector<glm::vec3> m_controllPoints;
        std::vector<glm::vec2> m_trimPoints;
        BezierCurve *m_bezier;
```

Im leeren Konstruktor wird wieder ein Beispiel einer Bézierfläche generiert. Diesmal wird hier ein 4x4 Kontrollgrid verwendet. Auf die Methode *createTrim*, welche die Trimmkurve der Fläche initialisiert, wird bei der Trimmung weiter eingegangen.

```
CVK::BezierSurfaceTesselation::BezierSurfaceTesselation()
{
        glm::vec3 b00(-1.5f, 0.0f,-1.5f);
        glm::vec3 b10(-0.5f, 1.0f,-1.5f);
        glm::vec3 b20( 0.5f, 1.0f,-1.5f);
        glm::vec3 b30( 1.5f, 0.0f,-1.5f);

        glm::vec3 b01(-1.5f, 1.0f,-0.5f);
        glm::vec3 b11(-0.5f, 2.0f,-0.5f);
        glm::vec3 b21( 0.5f, 2.0f,-0.5f);
        glm::vec3 b31( 1.5f, 1.0f,-0.5f);

        glm::vec3 b02(-1.5f, 1.0f, 0.5f);
        glm::vec3 b12(-0.5f, 2.0f, 0.5f);
        glm::vec3 b22( 0.5f, 2.0f, 0.5f);
        glm::vec3 b32( 1.5f, 1.0f, 0.5f);

        glm::vec3 b03(-1.5f, 0.0f, 1.5f);
        glm::vec3 b13(-0.5f, 1.0f, 1.5f);
        glm::vec3 b23( 0.5f, 1.0f, 1.5f);
        glm::vec3 b33( 1.5f, 0.0f, 1.5f);


        m_controllPoints.push_back(b00);
        m_controllPoints.push_back(b10);
        m_controllPoints.push_back(b20);
        m_controllPoints.push_back(b30);

        m_controllPoints.push_back(b01);
        m_controllPoints.push_back(b11);
        m_controllPoints.push_back(b21);
        m_controllPoints.push_back(b31);

        m_controllPoints.push_back(b02);
        m_controllPoints.push_back(b12);
        m_controllPoints.push_back(b22);
        m_controllPoints.push_back(b32);

        m_controllPoints.push_back(b03);
        m_controllPoints.push_back(b13);
        m_controllPoints.push_back(b23);
        m_controllPoints.push_back(b33);


        createTrim();
        createBuffers();
}
```

Die Methode *render* rendert die Bézierfläche. Da hier zum Rendern ein Tesselation-Shader verwendet wird, wird ein spezieller Aufruf benötigt. Zunächst muss mit *glPatchParameteri* die Anzahl der Vertices angegeben

werden, die einen *Patch* definieren. Dies ist in diesem Beispiel 16, da eine 4x4 Fläche gerendert werden soll.

```
void CVK::BezierSurfaceTesselation::render()
{
	glPatchParameteri( GL_PATCH_VERTICES, 16);
	glDrawArrays(GL_PATCHES, 0, 16);
}
```

Wichtig ist dabei, dass die gewählte Anzahl nicht die maximal mögliche Anzahl übersteigt. Die Höhe von *GL_MAX_ PATCH_VERTICES* hängt vom jeweiligen System ab, in diesem Fall lag sie bei 32. Um die Zahl herauszufinden, kann eine einfache Abfrage verwendet werden.

```
	GLint MaxPatchVertices = 0;
	glGetIntegerv(GL_MAX_PATCH_VERTICES, &MaxPatchVertices);
	printf("Max supported patch vertices: %d\n", MaxPatchVertices);
```

Im ersten Teil des Tesselation-Shaders, dem *Tesselation-Control-Shader* (TCS), wird das Level der Tesselierung festgelegt, also wie stark die Fläche unterteilt wird (siehe Abbildung 11). In diesem Fall wird eine Uniform Variable *level* übergeben, damit der Wert flexibler über die Anwendung geändert werden kann.

```
layout (vertices = 16) out;

uniform float level;

void main( )
{
	if(gl_InvocationID == 0)
	{
		gl_TessLevelInner[0] = level;
		gl_TessLevelInner[1] = level;

		gl_TessLevelOuter[0] = level;
		gl_TessLevelOuter[1] = level;
		gl_TessLevelOuter[2] = level;
		gl_TessLevelOuter[3] = level;
	}
	gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

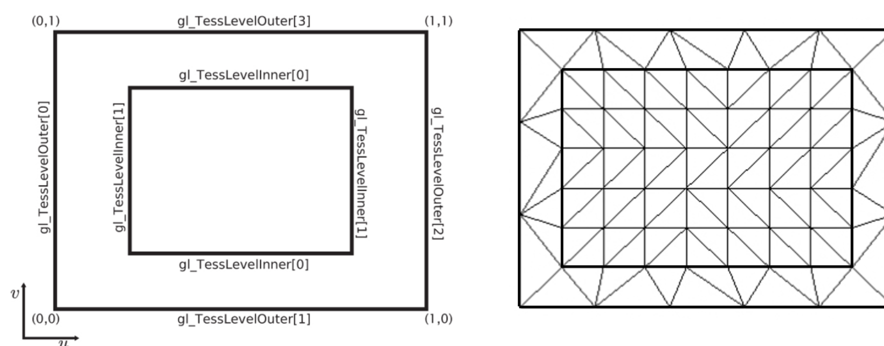

Abbildung 11: Tesselierungs Level bei quadratischen Patches. [5]

Im zweiten Schritt, dem *Tesselation-Evaluation-Shader* (TES) wird die Bézierfläche anhand der Kontrollpunkte berechnet. In der aktuellen Version sind

nur 4x4 Bézierflächen möglich. Die *deCasteljau* Methode im TES arbeitet somit auch nur auf Bézierkurven mit vier Kontrollpunkten.

```
vec4 deCasteljau(float t, vec4 p0, vec4 p1, vec4 p2, vec4 p3)
{
        vec4 p01 = (1-t) * p0 + t * p1;
        vec4 p12 = (1-t) * p1 + t * p2;
        vec4 p23 = (1-t) * p2 + t * p3;

        vec4 p012 = (1-t) * p01 + t * p12;
        vec4 p123 = (1-t) * p12 + t * p23;

        vec4 p0123 = (1-t) * p012 + t * p123;

        return p0123;
}
```

Durch *createSurface* wird dann der Punkt $P_{u,v}$ der Fläche berechnet, dabei wird zunächst der De Casteljau in v-Richtung angewandt und dann mit den resultierenden Punkten in u-Richtung.

```
vec4 createSurface(float u, float v)
{
        vec4 p[4];

        for(int i = 0; i < 4; i++)
        {
                p[i] = deCasteljau(v, gl_in[i+0].gl_Position, gl_in[i+4].gl_Position,
                                gl_in[i+8].gl_Position, gl_in[i+12].gl_Position);

        }
        return deCasteljau(u, p[0], p[1], p[2], p[3]);
}
```

In der *main* Methode des TES wird *createSurface* mit den Tesselierungskoordinaten *gl_TessCoord*, welche im TCS festgelegt wurden, aufgerufen, um den jeweiligen Punkt der Fläche zu berechnen.

Die Tangenten werden ebenfalls mit *createSurface* berechnet. Hier wird jeweils ein Punkt auf der Fläche berechnet, dessen Positionen sich um 0.001 in der x- beziehungsweise y-Koordinate vom eigentlichen Punkt unterscheiden. Darauf werden die Strecken zwischen den neuen Punkten *bpos* und *cpos* und dem eigentlichen Punkt *pos* berechnet. Durch das Skalarprodukt der beiden Tangenten erhält man dann die Normale für den Punkt auf der Fläche. [5]

Auch die Texturkoordinaten werden hier übergeben, diese entsprechen den Tesselierungskoordinaten *gl_TessCoord*.

```
void main()
{
        vec4 pos = createSurface(gl_TessCoord.x, gl_TessCoord.y);
        vec4 P = viewMatrix * modelMatrix * pos;

        vec4 bpos = createSurface(gl_TessCoord.x, gl_TessCoord.y + 0.001);
        vec4 cpos = createSurface(gl_TessCoord.x + 0.001, gl_TessCoord.y);
        bTangent = normalize(bpos.xyz - pos.xyz);
        cTangent = normalize(cpos.xyz - pos.xyz);
        vec3 Normal = normalize(cross(bTangent,cTangent));

        passNormal = mat3(viewMatrix* modelMatrix) * Normal;
        passLight = lightPos - P.xyz;
        passView = -P.xyz;
        passTcoord = vec2(gl_TessCoord.x, gl_TessCoord.y);

        gl_Position = projectionMatrix * P;
}
```

3.3 Trimmung

Die Trimmung der Bézierfläche geschieht im Fragment-Shader. Hier wird zunächst eine einfache Beleuchtung berechnet. Die Trimmung der Fläche wurde auf verschiedene Arten realisiert. Anhand einer If-Abfrage wird entschieden, ob das Fragment gerendert wird oder nicht. Zunächst wurde ein Intervall angegeben, in dem nicht gerendert werden sollte (siehe Abbildung 17).

```
void main()
{
        vec3 diffuseA = texture( colorTexture, passTcoord.xy).rgb;
        vec3 specularA = vec3(0.7);
        float shininess = 100.0;

        vec3 N = normalize(passNormal);
        vec3 L = normalize(passLight);
        vec3 V = normalize(passView);

        vec3 R = reflect(-L, N);

        if(passTcoord.x < 0.25f || passTcoord.x > 0.75f ||
                                        passTcoord.y < 0.25f || passTcoord.y > 0.75f)
        {
                diffuse = texture( colorTexture, passTcoord).rgb;
                vec3 specular = pow(max(dot(R,V),0.0), shininess) * specularA;
                color = vec4(diffuse + specular, 1.0);
                color.a = 1.0;
        }else
                discard;
}
```

In der zweiten Variante wurde alles in einem festgelegten Abstand zu einem festgelegten Punkt nicht gerendert (siehe Abbildung 18).

```
        if(distance(passTcoord, vec2(0.75f,0.75f))> 0.15f)
        {
                diffuse = texture( colorTexture, passTcoord).rgb;
                vec3 specular = pow(max(dot(R,V),0.0), shininess) * specularA;
                color = vec4(diffuse + specular, 1.0);
                color.a = 1.0;
        }else
                discard;
```

Im nächsten Schritt sollte die Trimmung mithilfe eines Polygonzuges erfolgen. Dazu wird in der Methode *createTrim* dieser initialisiert und dann in den Vektor *m_trimPoints* eingefügt.

```
void CVK::BezierSurfaceTesselation::createTrim()
{
        //Polygontrail as trimcurve
        glm::vec2 t0(0.25f, 0.25f);
        glm::vec2 t1(0.15f, 0.5f);
        glm::vec2 t2(0.25f, 0.75f);
        glm::vec2 t3(0.5f, 0.85f);
        glm::vec2 t4(0.75f, 0.75f);
        glm::vec2 t5(0.85f, 0.5f);
        glm::vec2 t6(0.75f, 0.25f);
        glm::vec2 t7(0.5f, 0.15f);
        glm::vec2 t8(0.25f, 0.25f);


        m_trimPoints.push_back(t0);
        m_trimPoints.push_back(t1);
        m_trimPoints.push_back(t2);
        m_trimPoints.push_back(t3);
        m_trimPoints.push_back(t4);
        m_trimPoints.push_back(t5);
        m_trimPoints.push_back(t6);
        m_trimPoints.push_back(t7);
        m_trimPoints.push_back(t8);
}
```

Die Werte für das Trimmobjekt werden dann mithilfe eines SSBOs (*Shader Storage Buffer Object*) an den Shader übergeben.

```
        //SSBO for trimcurve
        GLuint trimObject_ssboID;
        glGenBuffers(1, &trimObject_ssboID);
        glBindBuffer(GL_SHADER_STORAGE_BUFFER, trimObject_ssboID);
        glBufferData(GL_SHADER_STORAGE_BUFFER, m_trimPoints.size() * sizeof(glm::vec2),
                                        m_trimPoints.data(),GL_STATIC_COPY);
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, trimObject_ssboID);
```

```
layout( std430, binding=1) buffer trimObject_ssboID
{
        vec2 trimObject[];
};
```

Die Methode *trimTest* testet, ob ein Pixel sich innerhalb oder außerhalb des Polygonzugs befindet. Dies wird ähnlich einem Clipping-Algorithmus mithilfe der Normale der einzelnen Seiten des Polygonzugs errechnet (siehe Abbildung 19).

```
bool trimTest()
{
        vec2 vec;

        for(int i= 0; i<trim.length()-1; i++)
        {
                vec = trim[i+1]-trim[i];

                vec2 normal = vec2(-vec.y,vec.x);
                normal = normalize(normal);
                float t = dot(normal, passTcoord-trim[i]);

                if(t > 0){
                        return true;
                }
        }
        return false;
}
```

In der Trimmabfrage wird nur gezeichnet, wenn die Methode *trimTest True* zurückgibt. In diesem Fall befindet sich das Fragment außerhalb der Trimmkurve und wird deshalb gezeichnet.

```

        for(int i= 0; i<trimObject.length(); i++)
        {
                trim[i]= trimObject[i];
        }

        if(trimTest()){
                vec3 diffuse = max(dot(N,L),0.0) * diffuseA;
                vec3 specular = pow(max(dot(R,V),0.0), shininess) * specularA;
                color = vec4(diffuse + specular, 1.0);
                color.a = 1.0;
        }else
                discard;


}
```

Um die Fläche auch mithilfe von Bézierkurven trimmen zu können, wurde eine weitere Klasse *BezierCurve* implementiert, welche eine zweidimensionale Bézierkurve beschreibt. In dieser wurde ebenfalls der De Casteljau implementiert, welcher aber in diesem Fall einen zweidimensionalen Vektor übergeben bekommt.

```
glm::vec2 BezierCurve::deCasteljau(std::vector<glm::vec2> points, float t)
{
        std::vector<glm::vec2> vec = points;
```

```
        std::vector<glm::vec2> c;

        while (vec.size()>1)
        {
                c.clear();

                for (int i = 0; i < vec.size() - 1; i++)
                {
                        glm::vec2 pi = (1 - t) * vec[i] + t * vec[i + 1];
                        c.push_back(pi);
                }

                vec.clear();
                vec = c;
        }
        return vec[0];
```

In der Methode *initCurve* wird die Kurve initialisiert, dabei gibt der Parameter t an, aus wie vielen Punkten die resultierende Kurve bestehen soll, nämlich $(t+1)$ Punkte.

```
void BezierCurve::initCurve(float t)
{
        for (float i = 0; i <= t; i++) {
                m_trimPoints.push_back(deCasteljau(m_controllPoints, (i / t)));
        }
}
```

Der leere Konstruktor gibt ein Beispiel einer Bézierkurve. Um die Kurve als Trimmkurve zu verwenden, muss sie geschlossen sein. Aus diesem Grund wurde der Anfangspunkt gleich dem Endpunkt gewählt. Der Paramerter t wurde mit 99 initialisiert, sodass die Kurve aus 100 Punkten besteht (siehe Abbildung 20 und 21).

```
BezierCurve::BezierCurve()
{
        glm::vec2 t0(0.5f, 0.25f);
        glm::vec2 t1(0.2f, 0.5f);
        glm::vec2 t2(0.35f, 0.9f);
        glm::vec2 t3(0.75f, 0.65f);
        glm::vec2 t4(0.5f, 0.25f);

        m_controllPoints.push_back(t0);
        m_controllPoints.push_back(t1);
        m_controllPoints.push_back(t2);
        m_controllPoints.push_back(t3);
        m_controllPoints.push_back(t4);

        initCurve(99);
}

BezierCurve::BezierCurve(std::vector<glm::vec2> controllPoints)
{
        initCurve(99);
}
```

In *createTrimCurve* wird die gewünschte Trimmkurve für die Bézierfläche gesetzt.

```
void CVK::BezierSurfaceTesselation::createTrimCurve(BezierCurve bezier)
{
        m_bezier = bezier;
        m_trimPoints = *m_bezier->getTrimPoints();
}
```

4 Ergebnisse

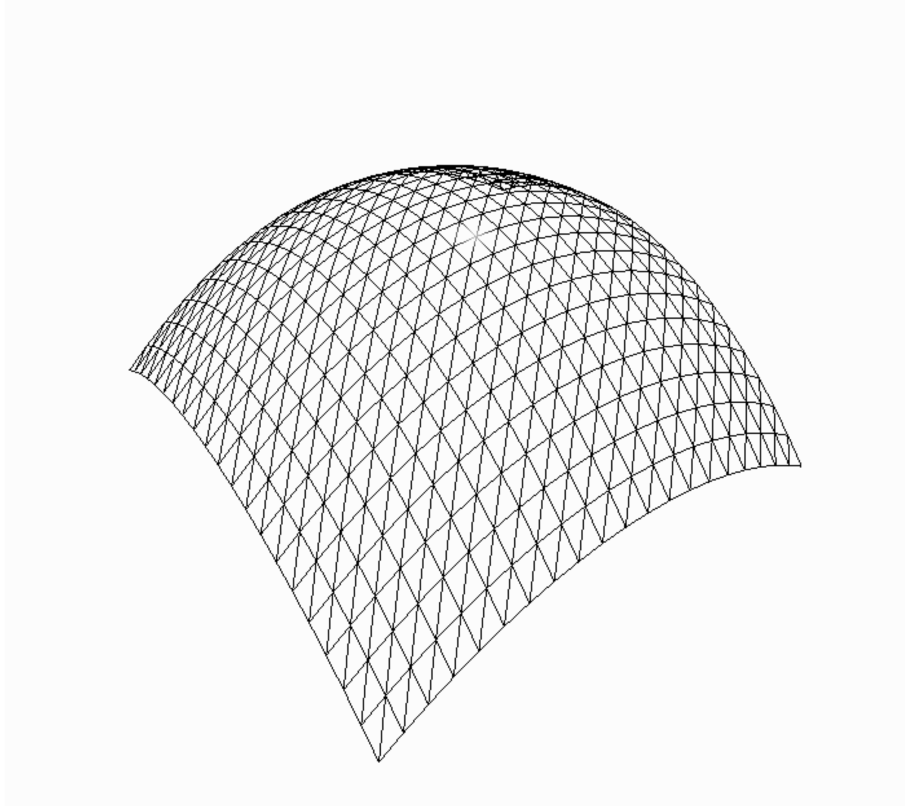

Abbildung 12: Triangulierte 3x3 Bézierfläche.

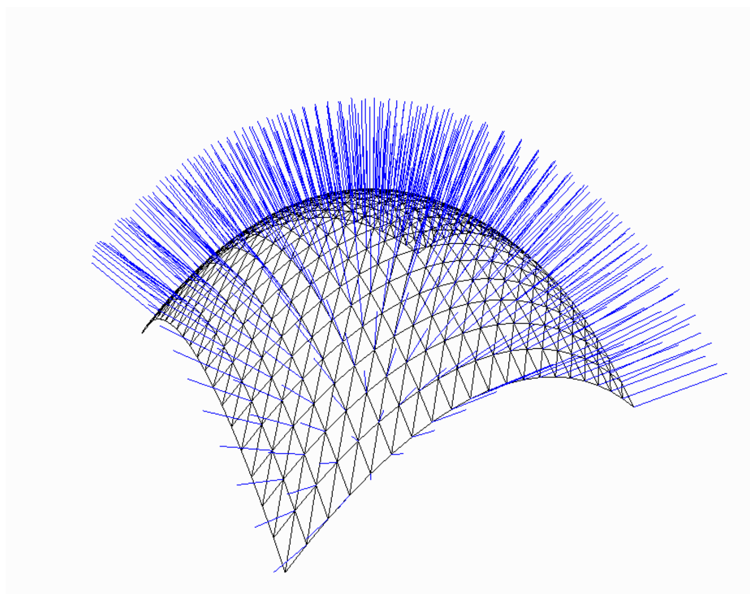

Abbildung 13: Triangulierte 3x3 Bézierfläche mit angezeigten Normalen.

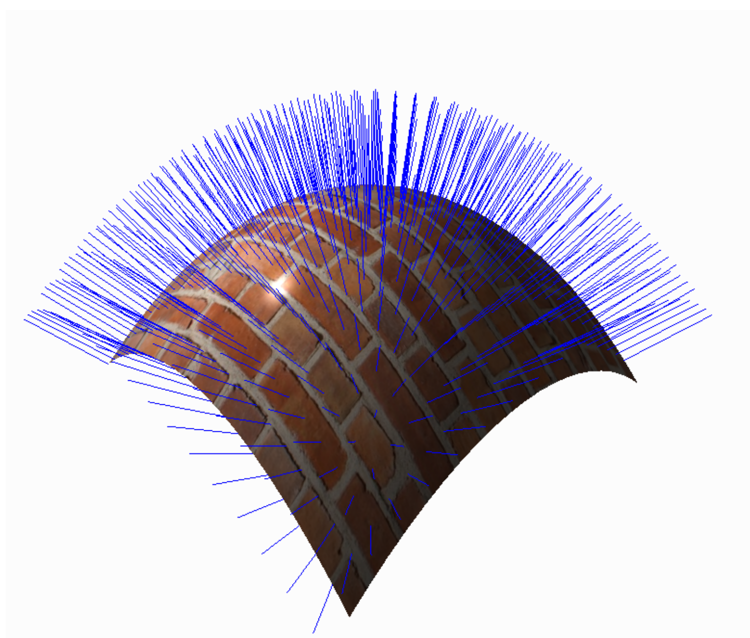

Abbildung 14: Triangulierte 3x3 Bézierfläche mit Textur und angezeigten Normalen.

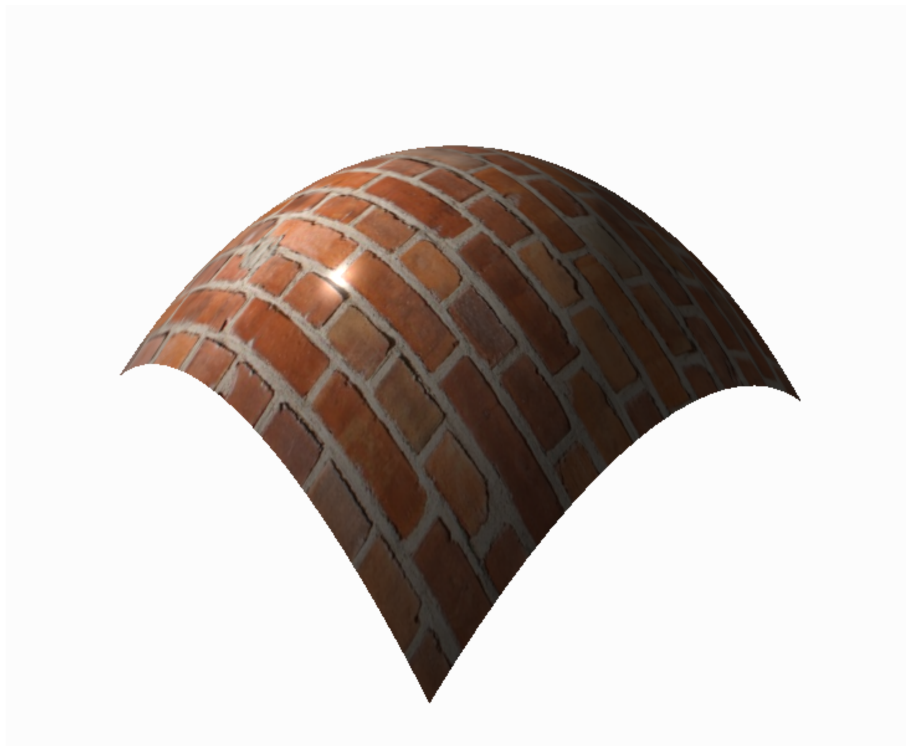

Abbildung 15: Triangulierte 3x3 Bézierfläche mit Textur.

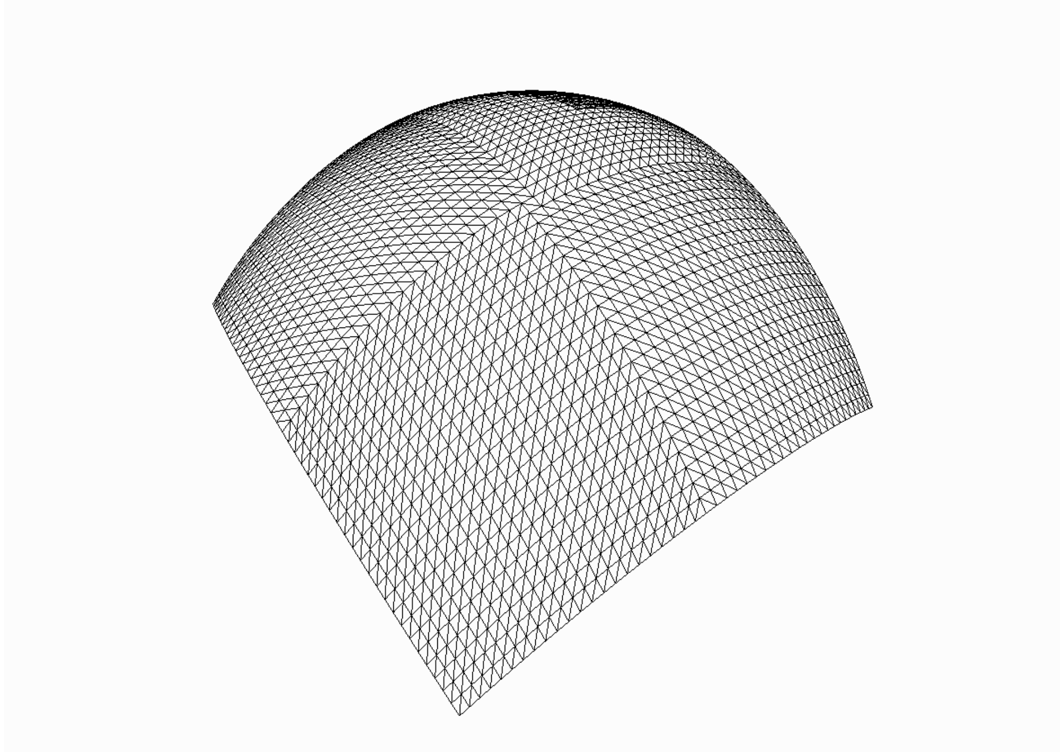

Abbildung 16: Tesselierte 4x4 Bézierfläche.

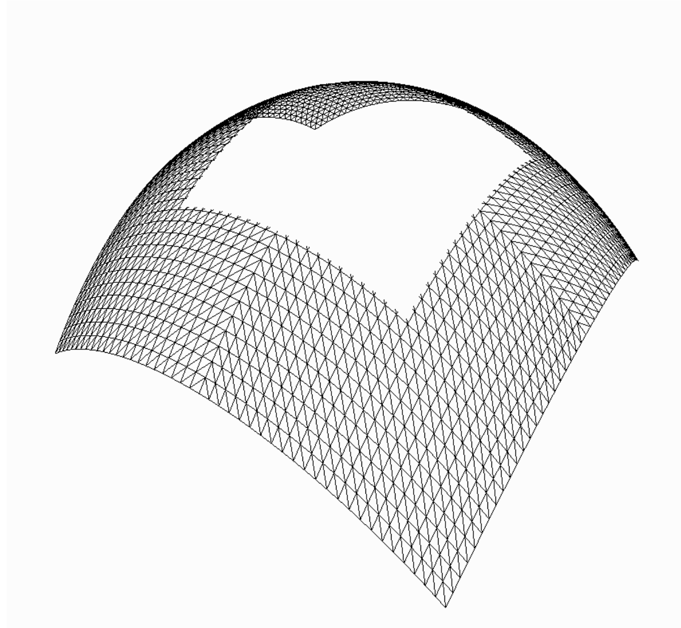

Abbildung 17: Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe eines Intervalls.

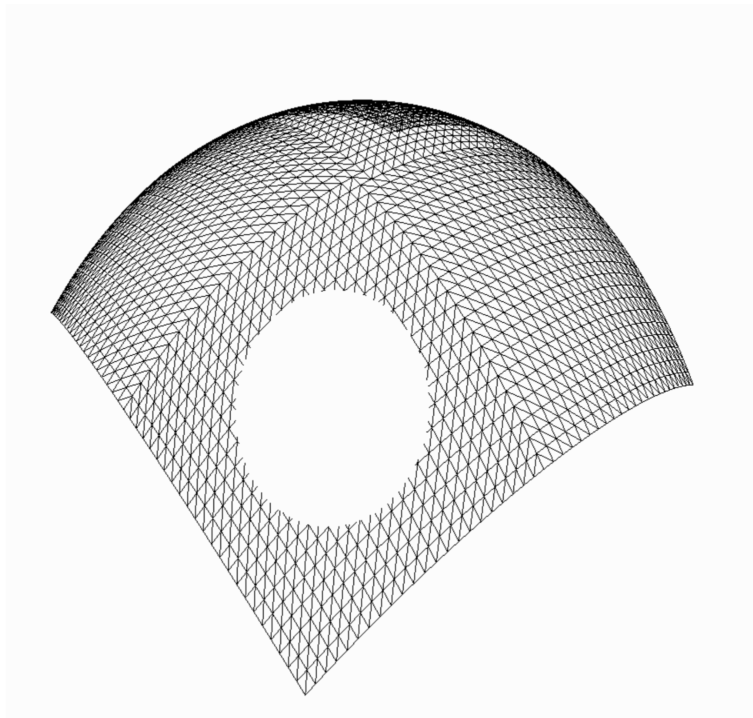

Abbildung 18: Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe einer Distanz zu einem Punkt.

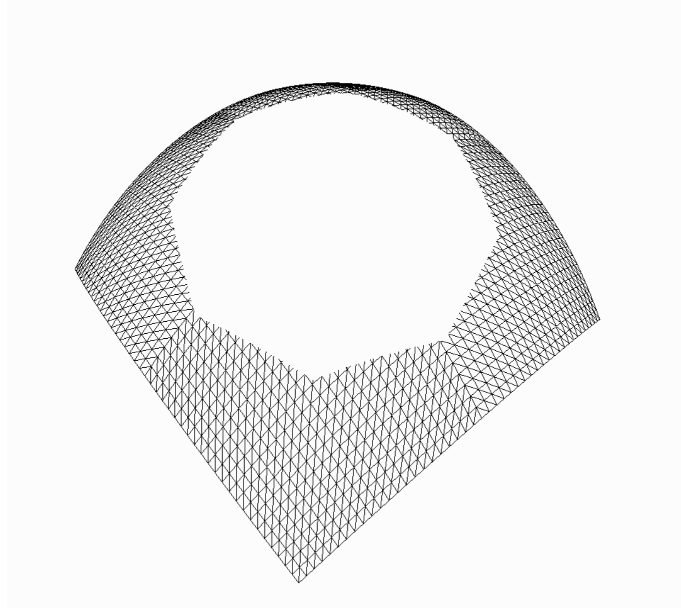

Abbildung 19: Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe eines Polygonzuges.

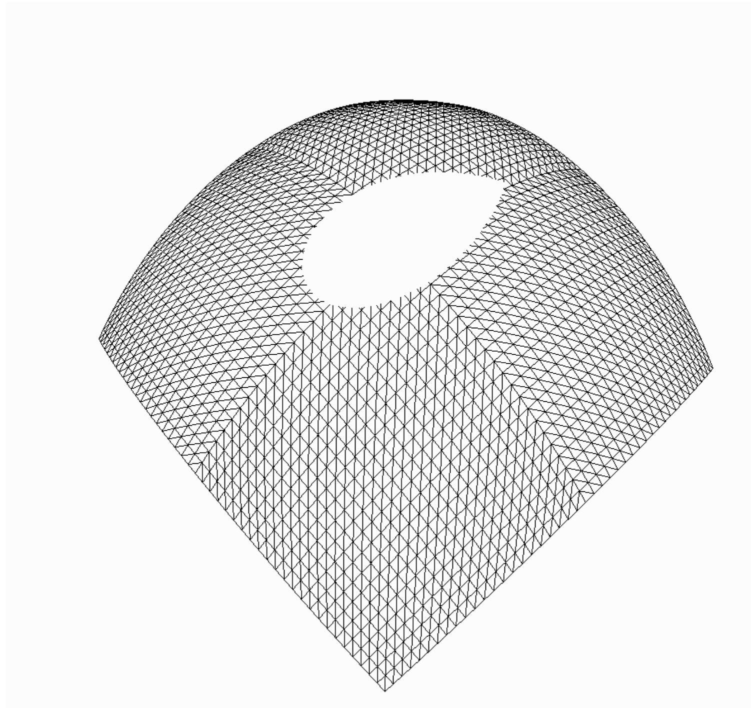

Abbildung 20: Tesselierte 4x4 Bézierfläche mit Trimmung mithilfe einer Bézierkurve.

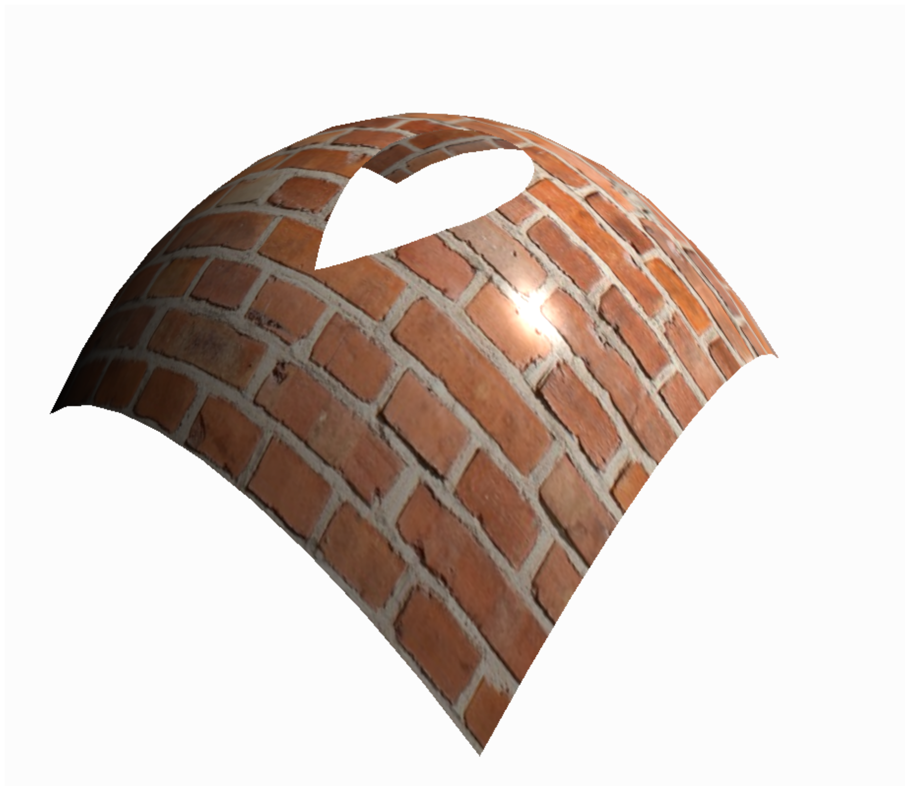

Abbildung 21: Tesselierte 4x4 Bézierfläche mit Textur und Trimmung mithilfe einer Bézierkurve.

5 Fazit

In dieser Bachlorarbeit wurden zwei Verfahren zum Rendern von Freiformflächen implementiert. Zum einen die Triangulierung der Bézierfläche und zum anderen die Tesselierung einer Bézierfläche. Auf Letzteres wurde zudem eine Trimmung mithilfe verschiedener Verfahren angewandt. Des Weiteren wurden für beide Verfahren die Normalen und Texturkoordinaten berechnet, um eine realistische Beleuchtung und Texturierung der Flächen zu ermöglichen.

Vorrausgehend wurden dazu zunächst wichtige Grundlagen und die mathematischen Hintergründe vorgestellt. Auch wurden weiterführende Verfahren zu Darstellung von Freiformflächen aufgezeigt.

5.1 Ausblick

Die implementierten Klassen können in vielerlei Hinsicht noch erweitert werden. So kann eine Trimmung auch mithilfe der Newton-Iteration durchgeführt werden, indem mithilfe der Schnittpunkte einer Geraden mit dem Trimmobjekt getestet werden kann, ob sich ein Punkt innerhalb des Trimmobjekts befindet oder nicht.

Aktuell beschränkt sich die Anzahl der Trimmobjekte einer Fläche auf eins. Durch die Newton-Iteration könnten hier auch mehrere möglich sein.

Außerdem könnte man den *Level of Detail*, also den Grad, wie stark die Fläche in einzelne Dreiecke unterteilt wird, dynamischer wählen. Dieser könnte zum einen von der Position der Kamera abhängig gemacht werden, sodass die Fläche höher aufgelöst wird, wenn die Kamera nah am Objekt dran ist. Auch könnte man den Grad der Unterteilung an der Krümmung der Fläche festmachen, sodass die Auflösung an Stellen mit starker Krümmung höher ist als an leicht gekrümmten Stellen. So bieten zum Beispiel Duchaineau et al. [13] und auch Catmul [14] Algorithmen zu einer solchen Realisierung.

Die Klasse der tesselierten Fläche ist aufgrund des Renderns mit einem Tesselation-Shader schwerer in das CVK einzubinden und erbt aus diesem Grund nicht von der Oberklasse aller Objekte (*CVK::Geometry*). Sie kann deshalb auch nicht als Knoten (*CVK::Node*) gerendert werden.

Aktuell beschränkt sich die Dimension der tesselierten Fläche auf 4x4, in weiteren Schritten könnte man diese variabler gestalten. Jedoch beschränkt sich dies dabei weiterhin auf die maximal zugelassene Zahl an Kontrollpunkten (*GL_MAX_PATCH_VERTICES*).

Im nächsten Schritt könnten dann auch die anderen beiden vorgestellten Ansätze der Freiformflächen implementiert werden, also B-Spline- und NURBS-Flächen.

Literatur

[1] Fredo Durand and Barb Cutler. Curves & surfaces. https://groups.csail.mit.edu/graphics/classes/6.837/F04/lectures/16_curves_surfaces.ppt. MIT EECS 6.837. Zugriff am 10.03.2018.

[2] Bézier surface. https://en.wikipedia.org/wiki/Bezier_surface. Wikipedia. Zugriff am 01.04.2018.

[3] Scratchapixel. Bézier curves and surfaces: the utah teapot. https://www.scratchapixel.com/lessons/advanced-rendering/bezier-curve-rendering-utah-teapot/bezier-surface. Zugriff am 10.03.2018.

[4] David F. Rogers. *An Introduction to NURBS - With Historical Perspective*. Morgan Kaufmann Publishers, 2001.

[5] Nicholas Haemel Graham Sellers, Richard S. Wright. *OpenGL SuperBible Sixth Edition*. Addison-Wesley, 2014.

[6] Dave Shreiner. *OpenGL Programming Guide Seventh Edition*. Addison-Wesley, 2010.

[7] Olga Kasemir. *Rendering von dreidimensionalen, getrimmten Freiformflächen*. Studienarbeit, Universität Koblenz-Landau, 2007.

[8] Bézierkurven. https://de.wikipedia.org/wiki/Bezierkurve. Wikipedia. Zugriff am 07.04.2018.

[9] Mario Holldack. Bézierkurven. http://www.math.uni-frankfurt.de/~numerik/lehre/Vorlesungen/Pros-11/Ausarbeitungen/holl.pdf., 2011. Goethe-Universität Frankfurt am Main. Zugriff am 09.02.2018.

[10] Oliver Vornberger. Computergrafik 2016. http://docplayer.org/36376595-Computergrafik-oliver-vornberger-universitaet-osnabrueck.html. Universität Osnabrück. Zugriff am 07.04.2018.

[11] Isaac J. Schoenberg. *Contributions to the problem of approximation of equidistant data by analytic functions*, pages 45–99, 112–141. Bd. 4. Quart. Appl. Math., 1946.

[12] Steve Coons. *Surfaces for computer-aided design of spaceforms*. MIT Proj. MAC, MAC-TR-41, 06/1967.

[13] D. E. Sigeti M. A. Duchaineau, M. Wolinsky. Roaming terrain: real-time optimally adapting meshes. *IEEE Visualization*, 1997.

[14] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Tech. Rep. UTEC-CSc-74-133, University of Utah, Salt Lake City, Utah.