UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Rendering von Freiformflächen

Bachelorarbeit

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

vorgelegt von

Mike Schank

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

Zweitgutachter: M. Sc. Bastian Krayer
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im November 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)

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 Freiformflächen 2
2.2 Bézierkurven 3
2.3 Bernsteinpolynome 5
2.4 Bézierflächen 6
2.5 Der de-Casteljau Algorithmus 8

3 Trimmung **9**
3.1 Vereinfachtes Trimmen 9
3.1.1 Rechteck und Quadrat 9
3.1.2 Kreis 10
3.1.3 Ellipse 11
3.2 Das Newton-Verfahren 12
3.2.1 Trimming mit Hilfe des Newton-Verfahrens 14

4 Implementierung **17**
4.1 Tessellierung 17
4.1.1 Tessellation-Control-Shader 18
4.1.2 Tessellation-Evaluation-Shader 20
4.2 Trimmung 23
4.2.1 Vereinfachtes Trimmen 23
4.2.2 Trimmen mit Hilfe des Newton-Verfahrens 24
4.2.3 Fragment-Shader 29
4.3 Anlegen einer neuen Freiformfläche 31

5 Ergebnisse und Grenzen **33**
5.1 Ergebnisse 33
5.2 Grenzen 40

6 Fazit **42**

Abbildungsverzeichnis

1 Bézierfläche mit blauem Kontrollpunktnetz [10] 2
2 Quadratische Bézierkurve 3
3 Kubische Bézierkurve 3
4 Gewichtsfunktionen für die kubische Bézierkurve [15] 5
5 Bézierfläche mit Kontrollnetz [14] 7
6 De-Casteljau Algorithmus mit 4 Kontrollpunkten und $t = 0.5$ 8
7 Beispiel für eine Trimmung mit Rechteck 9
8 Beispiel für eine Trimmung mit Kreis 10
9 Beispiel für eine Trimmung mit Ellipse 11
10 Newton-Verfahren Illustration (Abbildung nach [3]) 12
11 Trimmung mit Bézierkurven 14
12 Trimmung mit verschachtelten Bézierkurven 17
13 Tessellation-Werte für *Quad-Tessellation* (Abbildung nach [4]) 20
14 Bézierfläche mit 16 Kontrollpunkten 33
15 Bézierfläche mit Trimmung eines Kreises und Textur 34
16 Bézierfläche mit Trimmung eines Rechteckes (Intervall) und Textur 34
17 Bézierfläche mit Trimmung eines Quadrates (Intervall) und Textur 35
18 Bézierfläche mit Trimmung einer Ellipse und Textur 35
19 Bézierfläche mit Trimmung von zwei Ellipsen und Textur . . 36
20 Bézierfläche mit mehreren Trimmflächen (Quadrat, Rechteck, Kreis, Ellipse) 36
21 Bézierfläche im Wireframe-Modus (*Tessellation-Level* = 32) . . 37
22 Bézierfläche mit Trimmung einer quadratischen Bézierfläche 37
23 Bézierfläche mit Trimmung einer quadratischen Bézierfläche in einer anderen quadratischen Bézierfläche 38
24 Bézierfläche mit Trimmung einer kubischen Bézierfläche . . 38
25 Bézierfläche mit Trimmung einer kubischen Bézierfläche in einer anderen kubischen Bézierfläche 39
26 Bézierfläche mit Trimmung einer kubischen Bézierfläche in einer anderen kubischen Bézierfläche im Wireframe-Modus 39
27 Rendern von mehreren getrimmten Bézierflächen 40
28 Fehler bei der Berechnung der Trimmfläche 40
29 7 Wiederholungen 41
30 6 Wiederholungen 41
31 $\epsilon = 0.1$ 41
32 $\epsilon = 0.01$ 41
33 $\epsilon = 0.001$ 41
34 $\epsilon = 0.0001$ 41

Listings

1 Tessellation-Control-Shader 18
2 Render-Methode 19
3 Tessellation-Evaluation-Shader (Teil 1) 20
4 Tessellation-Evaluation-Shader (Teil 2) 21
5 Tessellation-Evaluation-Shader (Teil 3) 21
6 Tessellation-Evaluation-Shader (Teil 4) 22
7 Trimmungverfahren für einfache Trimmformen (Fragment-Shader) 23
8 Funktionen für das Newton-Verfahren (quadratisch) (Fragment-Shader) Code nach [1] 24
9 Funktionen für das Newton-Verfahren (kubisch) (Fragment-Shader). Code nach [1] 25
10 Newton-Verfahren. Code nach [1] 26
11 Trimmung mit dem Newton-Verfahren (Teil 1) 27
12 Trimmung mit dem Newton-Verfahren (Teil 2) 28
13 Trimmung mit dem Newton-Verfahren (Teil 3) 29
14 Anlegen einer Freiformfläche (Teil 1) 31
15 Anlegen einer Freiformfläche (Teil 2) 32
16 Rendern der Freiformfläche 32
17 Bézierkurventrimmung (einzel, quadratisch) 33

Abstract

In recent years, a lot of changes happened in the field of computer science, especially in computer graphics. Besides the conventional representation of objects by rendering of triangles, there is another way, since the introduction of the tessellation shader, which allows a representation of freeform surfaces. The implementation of the tessellation shader enables fast computationial methods in the grapchics processing unit (GPU). This paper deals with the implementation of these freeform surfaces. The focus is on the bézier surfaces, which are a kind of freeform surfaces. The computation does not happen on the central processing unit (CPU) but on the GPU, precisely in the tessellation shader, to ensure it is fast and efficient.

The trimming of trivial objects like circles, rectangles or ellipses and the trimming of quadratic an cubic bézier curves is also described. This last method of trimming is expanded so that it is possible to render a bézier surface within trimmed bézier curves.

The implementation is an extension of the Framework from the university of Koblenz-Landau (CVK).

Zusammenfassung

In den letzten Jahren, hat sich im Bereich der Informatik, genauer der Computergrafik, viel verändert. Neben der herkömmlichen Darstellung von Objekten mittels Rendern von Dreiecken, gibt es seit der Einführung des Tessellation-Shaders ein weiteres Verfahren, welches es ermöglicht Freiformflächen darzustellen. Die Implementierung des Tessellation-Shaders erlaubt es, auf eine schnelle Berechnung auf dem Grafikprozessor (GPU, engl. graphics processing unit) zurückzugreifen. In dieser Arbeit wird auf die Implementierung dieser Freiformflächen eingegangen. Dabei stehen die Bézierflächen, welche eine Art Freiformflächen sind, im Fokus. Um eine effiziente Berechnung zu gewährleisten, findet diese nicht auf dem Hauptprozessor (CPU, engl. central processing unit), sondern auf der GPU, im Tessellation-Shader, statt.

Neben der Tessellierung der Bézierflächen, wird auf das Trimmen von trivialen Objekten wie Kreis, Rechteck oder Ellipse, sowie auf das Trimmen von quadratischen und kubischen Bézierkurven eingegangen. Letzteres wird noch erweitert, sodass es möglich ist eine Bézierfläche innerhalb getrimmten Bézierkurven zu rendern.

Die Implementierung ist eine Ergänzung zum Framework der Universität Koblenz-Landau (CVK).

1 Einleitung

Die Darstellung von Freiformflächen wird in vielen Bereichen des modernen computergestützten Design genutzt. Vor allem in CAD-Programmen (CAD, engl. computer aided design) findet es häufig Anwendung, z.B. beim Design von Konsumgütern oder im Karosseriebau von Autoherstellern [17]. Die Berechnung erfolgt dabei auf der GPU, weil diese, im Gegensatz zu der CPU, in der Lage ist Berechnungen parallel zu lösen. Dies ermöglicht eine effizientere Berechnung der einzelnen Positionen der Vertices der Freiformfläche. Es gibt mehrere Darstellungsmöglichkeiten von Freiformflächen. Dabei hat jede ihre Vor- und Nachteile. Es gibt z.B. die Methode der NURBS (engl. non-uniform rational b-spline) [7] oder die Methode der Bézierflächen. Letztere Methode ist die, auf die hier eingegangen wird. Das Ganze erfolgt mit Hilfe des Tessellation-Shaders. Ohne diesen, könnten die Flächen auch mit der normalen Darstellung von Dreiecken oder durch Raytracing berechnet werden. Um jedoch ein schnelles Rendern auf der GPU zu ermöglichen, wird der Tessellation-Shader benutzt.

Der Zweck dieser Arbeit ist es, ausgehend vom Framework der Universität Koblenz-Landau (CVK), eine Klasse zu implementieren, die es ermöglicht Freiformflächen zu rendern. Bei diesen Freiformflächen handelt es sich um Bézierflächen. Neben der Darstellung dieser Flächen, wird auf das Trimmen der Flächen eingegangen, insbesondere das Trimmen mit Hilfe des Newton-Verfahrens.

Um die Vorgehensweise der Implementierung zu beschreiben, wird zunächst in Kapitel 2 auf die mathematischen und theoretischen Grundlagen eingegangen. Dieses Kapitel übermittelt die nötigen Grundkenntnisse über die Bézierkurven, die Bézierflächen, die Bernsteinpolynome sowie den de-Casteljau Algorithmus um die Implementierung nachvollziehen zu können. In Kaptiel 3 wird der theoretische Hintergrund beim Trimmen erklärt. Zunächst das vereinfachte Trimmen und danach das Trimmen mit dem Newton-Verfahren, welche das Trimmen einer Bézierkurve ermöglicht. In Kapitel 4, welches das wichtigste Kapitel ist, geht es um die Implementierung, also den Programmcode der Arbeit (sämtlicher Code ist in der Programmiersprache C++ geschrieben). Zuerst wird der Codeabschnitt im Tessellation-Control-Shader (TCS) und im Tessellation-Evaluation-Shader (TES) Schritt für Schritt erklärt. Beide sind für das Rendern von Freiformflächen von großer Bedeutung. Danach wird der Code für das Trimmen erklärt, insbesondere das Newton-Verfahren. In Kapitel 5 sind die Ergebnisse und Grenzen dieser Implementierungsart zu sehen. Abgerundet wird das Ganze mit einem Fazit.

2 Grundlagen

In diesem Kapitel wird auf die grundlegenden Verfahren, die für das Rendern und Trimmen von Freiformflächen benötigt werden, eingegangen. Im Vordergrund stehen dabei die theoretischen Hintergründe aus mathematischer Sicht. Insbesondere wird das Trimmen mit Hilfe des Newton-Verfahrens, welches das Herausschneiden von Freiformkurven ermöglicht, genauer erklärt.

2.1 Freiformflächen

Freiformflächen werden in der Computergrafik und in CAD-Programmen benutzt um die Oberfläche von dreidimensionalen geometrischen Elementen zu beschreiben, die keine gewöhnliche Geometrieform haben, wie z.B. Kreise, Würfel, Zylinder oder Tori. Die Art, Geometrie auf diese Art darzustellen, hat ihren Ursprung in der Automobil- und Luftfahrtindustrie. Heute ist sie jedoch in allen computergestützten Designdisziplinen sehr verbreitet [17]. Die Flächen werden mit Hilfe von sogenannten Kontrollpunkten in einem dreidimensionalem Raum dargestellt. Egal wie man die Kontrollpunkte verschiebt, die Fläche passt sich immer dem Kontrollpunktnetz an.

Die meisten modernen Freiformflächen basieren auf dem Prinzip der sogenannten NURBS. Es gibt auch andere Methoden wie z.B die Coonsfläche [16], die Gordonfläche oder die Bézierfläche [17]. Diese Arbeit beschäftigt sich mit dem Rendering von Letzterem.

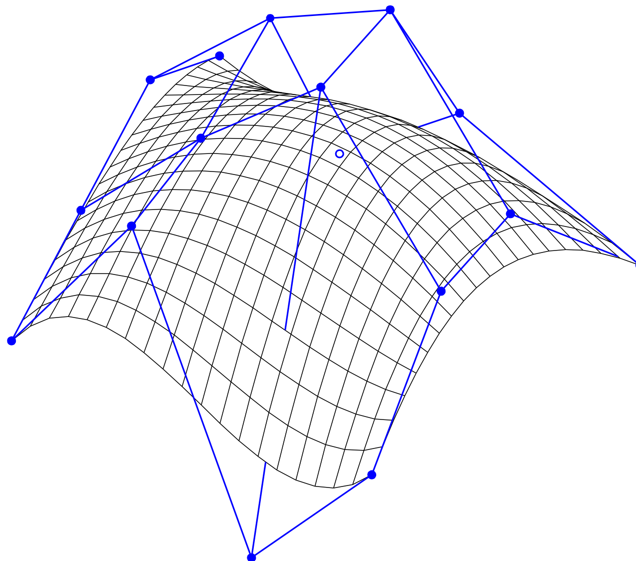

Abbildung 1: Bézierfläche mit blauem Kontrollpunktnetz [10]

2.2 Bézierkurven

Als Bézierkurve wird eine parametrisierte Kurve bezeichnet, die aus einer bestimmten Anzahl von Kontrollpunkten approximiert wird. Sie sind ein wichtiges Hilfsmittel wenn es um die Darstellung von Freiformflächen geht. Wegen ihrer leichten mathematischen Handhabung, finden sie oft Verwendung in Vektorgrafikprogrammen sowie in CAD-Programmen.

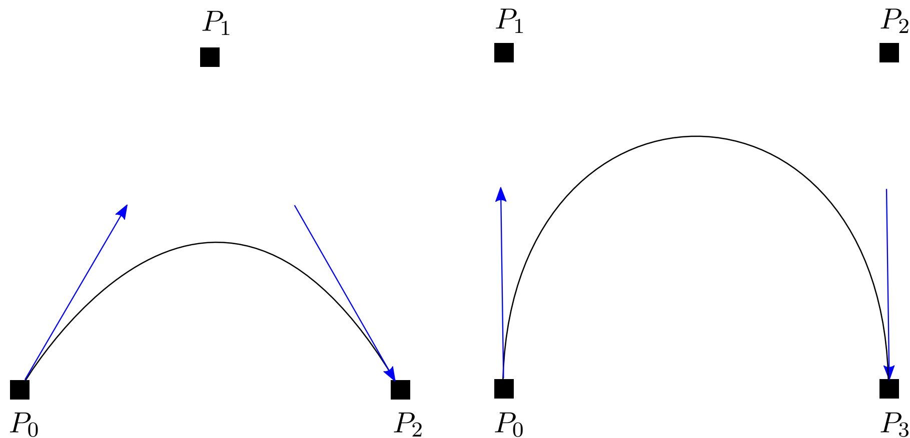

Abbildung 2: Quadratische Bézierkurve **Abbildung 3:** Kubische Bézierkurve

Die Anzahl der Kontrollpunkte bestimmt den Grad der Bézierkurve. Dieser Grad n wird durch die Formel: *Anzahl der Kontrollpunkte - 1* berechnet. Die zwei am häufigsten vorkommenden Arten sind die quadratischen (Grad 2) und die kubischen (Grad 3) Bézierkurven. Das Verfahren kann auf beliebig viele Kontrollpunkte erweitert werden. Auf der Abbildung 2 ist eine quadratische Bézierkurve zu sehen, mit den Kontrollpunkten P_0, P_1 und P_2. Eine kubische Bézierkurve ist, mit den Kontrollpunkten P_0, P_1, P_2 und P_3, auf Abbildung 3 zu sehen.

Zu beachten ist, dass die Kurve immer durch den Anfangs- bzw. Endkontrollpunkt verläuft. Zu den weiteren Eigenschaften der Bézierkurven gehört, dass sich die approximierte Kurve immer in der konvexen Hülle der jeweiligen Kontrollpunkte befindet. Außerdem sind die Tangenten des ersten und letzten Kontrollpunktes (in den Abbildungen 2 und 3 dargestellt durch blaue Pfeile) immer festgelegt. Es ist zu erkennen, dass die Tangente des ersten Kontrollpunktes mit dem zweiten Kontrollpunkt verbunden ist und die Tangente des letzten Kontrollpunktes mit dem vorletzten Kontrollpunkt.

Die mathematische Formel zur Berechnung eines beliebigen Punktes auf einer quadratischen Bézierkurve lautet wie folgt:

$$P(t) = \mathbf{(1-t)^2} \cdot P_0 + \mathbf{2 \cdot t \cdot (1-t)} \cdot P_1 + \mathbf{t^2} \cdot P_2 \tag{1}$$

mit:

- P_0: erster Kontrollpunkt
- P_1: zweiter Kontrollpunkt
- P_2: dritter Kontrollpunkt
- $t \in [0, 1]$

Um eine Bézierkurve zu erhalten, muss jeder Punkt mit Hilfe der Formel (1) definiert werden. Dabei wird eine beliebige Anzahl an Werten zwischen 0 und 1 für t eingesetzt. Je mehr Werte eingesetzt werden, desto höher ist die Genauigkeit der daraus resultierenden Bézierkurve.

Die Formel zur Berechnung eines beliebigen Punktes auf einer kubischen Bézierkurve lautet wie folgt:

$$P(t) = \mathbf{(1-t)^3} \cdot P_0 + \mathbf{3 \cdot t \cdot (1-t)^2} \cdot P_1 + \mathbf{3 \cdot t^2 \cdot (1-t)} \cdot P_2 + \mathbf{t^3} \cdot P_3 \tag{2}$$

mit:

- P_0: erster Kontrollpunkt
- P_1: zweiter Kontrollpunkt
- P_2: dritter Kontrollpunkt
- P_3: vierter Kontrollpunkt
- $t \in [0, 1]$

Die Funktionsweise der Formel (2) ist mit der vorherigen identisch. Der einzige Unterschied ist die Anwesenheit eines weiteren Kontrollpunktes und die Änderung der Gewichtsfunktionen. Die in den Formeln fett dargestellten Koeffizienten sind als Gewichtsfunktionen definiert.

In Abbildung 4 sind die Gewichtsfunktionen für die kubische Bézierkurve zu sehen. Die Gewichtsfunktion $(1-t)^3$ ist durch die dunkelblaue Linie dargestellt und beschreibt das Gewicht vom ersten Kontrollpunkt (Wenn $t = 0$ (x-Achse), dann ist $Gewicht = 1$ (y-Achse)). Die Gewichtsfunktion $3 \cdot t \cdot (1-t)^2$ ist durch die grüne Linie dargestellt und beschreibt das Gewicht vom zweiten Kontrollpunkt. Die Gewichtsfunktion $3 \cdot t^2 \cdot (1-t)$ ist durch die rote Linie dargestellt und beschreibt das Gewicht vom dritten Kontrollpunkt und die Gewichtsfunktion t^3 ist durch die hellblaue Linie dargestellt und beschreibt das Gewicht vom vierten Kontrollpunkt. Sie sind immer positiv und die Summer der Gewichtsfunktionen für jeden Wert t ist immer genau 1. Sie werden auch Bernsteinpolynome genannt.

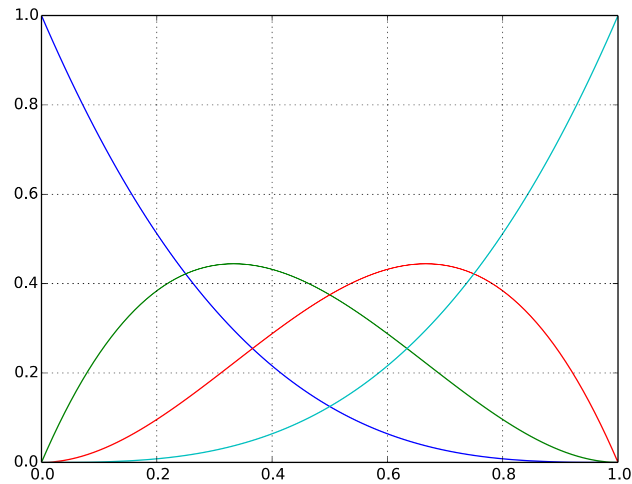

Abbildung 4: Gewichtsfunktionen für die kubische Bézierkurve [15]

2.3 Bernsteinpolynome

Obwohl die Bernsteinpolynome, die als Basis für die Bézierkurven dienen, schon seit 1912 bekannt sind, wurden sie erst etwa 50 Jahre danach bekannter als sie von dem französischen Ingenieur Pierre Bézier benutzt wurden um Karosserien von Autos der Firma Renault zu designen [15].

Sie sind wie folgt definiert:

$$B_{i,n}(t) = \binom{n}{i} \cdot t^i \cdot (1-i)^{n-i} \tag{3}$$

mit $\binom{n}{i}$ als Binomialkoeffizient:

$$\binom{n}{i} = \frac{n!}{i! \cdot (n-i)!} \tag{4}$$

Durch Zusammenfügen der Formeln (3) und (4) entsteht:

$$B_{i,n}(t) = \frac{n!}{i! \cdot (n-i)!} \cdot t^i \cdot (1-i)^{n-i} \tag{5}$$

Die Formel (5) kann benutzt werden um mit Hilfe der Kontrollpunkte eine Bezierkurve zu berechnen:

$$P(t) = \sum_{i=0}^{n} \cdot B_{i,n}(t) \cdot P_i \tag{6}$$

Nach Einsetzen der Formel (5) in (6) ergibt sich:

$$P(t) = \sum_{i=0}^{n} \cdot \frac{n!}{i! \cdot (n-i)!} \cdot t^i \cdot (1-i)^{n-i} \cdot P_i \qquad (7)$$

mit:

- $t \in [0,1]$
- $n \in \mathbb{N}$: Grad des Polynoms
- P_i: Kontrollpunkt mit dem Index i.

Somit kann mit Hilfe der Bernsteinpolynome eine Bézierkurve dargestellt werden. Bei einem Vergleich zwischen der Formel (2) aus Kapitel 2.2 und der Formel (6) wird auch klar, dass diese, nach Einfüllen von konkreten Werten der vier Kontrollpunkte, identisch sind. Das Verständnis der Bernsteinpolynome ist wichtig, um die Berechnung von Bezierflächen zu verstehen, da diese auf jenen aufbauen.

2.4 Bézierflächen

Ausgehend von der Formel (6), die eine einfache Bézierkurve im zweidimensionalem Raum beschreibt, kann das gleiche Verfahren angewandt werden um eine Bézierfläche im dreidimensionalem Raum darzustellen.

Ausgangspunkt ist dabei eine normale Bézierkurve:

$$P(u) = \sum_{i=0}^{n} \cdot B_{i,n}(u) \cdot P_i \qquad (8)$$

Nun soll jeder vorkommende Kontrollpunkt P_i wieder eine Bézierkurve bestimmen:

$$P_i = P_i(v) = \sum_{j=0}^{m} \cdot B_{j,m}(v) \cdot P_{i,j} \qquad (9)$$

Nach Einsetzen der Formel (9) in (8) entsteht die Formel für die Berechnung einer Bézierfläche:

$$P(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \cdot B_{i,n}(u) \cdot B_{j,m}(v) \cdot P_{i,j} \qquad (10)$$

Die Koeffizienten $P_{i,j} \in \mathbb{R}^3$ sind die jeweiligen Kontrollpunkte und die daraus resultierende Fläche heißt Bézierfläche.

Die Flächenpunkte $P(u,v)$ werden mit Hilfe der Parameter (u,v) aus dem Tessellation-Shader berechnet. Dazu wird der eindimensionale de-Casteljau Algorithmus iterativ angewendet. Einmal in *u-Richtung* und einmal in *v-Richtung*. Die Reihenfolge spielt dabei keine Rolle.

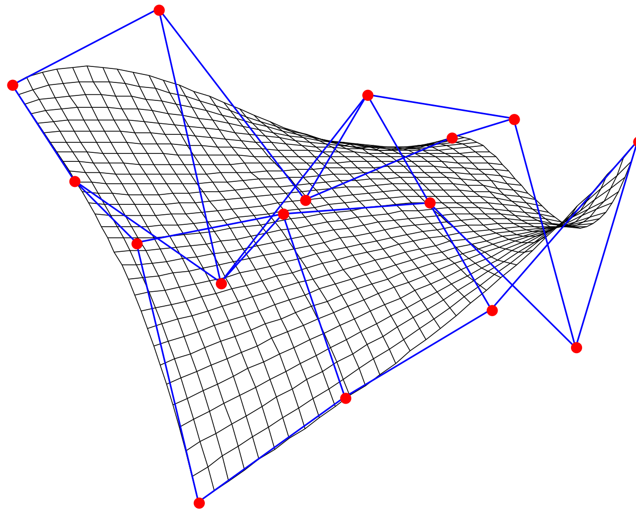

Abbildung 5: Bézierfläche mit Kontrollnetz [14]

Eigenschaften der Bezierflächen [8]:

- Die Bézierfläche liegt, genau wie die Bézierkurve, in einer konvexen Hülle die durch ein Kontrollpunktnetz aus $n \times m$ Punkten bestimmt ist. Es gilt:

$$B_{i,n}(u) \cdot B_{j,m}(v) >= 0 \tag{11}$$

$$\sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,n}(u) \cdot B_{j,m}(v) = 1 \tag{12}$$

  $P(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \cdot B_{i,n}(u) \cdot B_{j,m}(v) \cdot P_{i,j}$ wird auch als Konvexkombination der Kontrollpunkte $P_{i,j}$ bezeichnet.

- Bei linearen Transformationen und Translationen verhält sich die Bézierkurve gleich zu ihren Kontrollpunkten.

- Alle im *(u,v)-Raum* liegenden Linien sind Bézierflächen (ähnlich zu den Bézierkurven, die ebenfalls durch Anfangs- und Endkontrollpunkt verlaufen). Das gilt auch für die vier Eckkanten.

- Die vier Eckpunkte der Bézierfläche entsprechen den vier Eckpunkten der Kontrollpunkte.

- Letztere Eigenschaft gilt üblicherweise nicht für die anderen Punkte auf der Bézierfläche, außer es handelt sich um eine flache Ebene (Kontrollpunkte liegen alle auf einer Ebene).

2.5 Der de-Casteljau Algorithmus

Der de-Casteljau Algorithmus ist nach seinem Erfinder Paul de Casteljau benannt [11]. Aufgrund der Rekursivität der Bernsteinpolynome lässt sich für jeden Parameter t und die gegebenen Kontrollpunkte $P_0, ..., P_n$ einen genauen Punkt $P(t)$ auf der Bézierkurve bestimmen.

Abbildung 6: De-Casteljau Algorithmus mit 4 Kontrollpunkten und $t = 0.5$

Der Algorithmus führt eine iterierte lineare Interpolation von jeweils zwei benachbarten Punkten aus, die im gleichen Verhältnis t unterteilt werden. Auf der Abbildung 6 ist zu erkennen, dass am Anfang drei mal zwischen den jeweiligen initialen Eckpunkten P_0, P_1, P_2 und P_3 interpoliert wird. Diese Methode wird rekursiv fortgesetzt bis am Ende nur noch ein Punkt $P(t)$ übrig bleibt. Dieser Punkt ist der für den Parameter t entsprechende Wert auf der Bézierkurve. Je mehr Werte $0 <= t <= 1$ dem Algorithmus übergeben werden, desto detailreicher wird die Kurve am Ende dargestellt.

Der vorletzte Rekursionsschritt berechnet automatisch die Tangente für den Punkt $P(t)$ (dargestellt durch den günen Pfeil). Das ist später, wenn es um die Implementierung geht ,noch wichtig, um die Normalen der Bézierfläche zu berechnen.

3 Trimmung

Das Trimmen von Oberflächen und Freiformflächen wird in der Computergrafik häufig beim Oberflächen-Modelling, Textur-Mapping und medizinischen Simulationen benutzt [5]. Es bietet die Möglichkeit gewünschte Formen aus einer Freiformfläche herauszuschneiden. In dieser Arbeit wird ein vereinfachtes Verfahren der Trimmung vorgestellt, bei dem es darum geht, triviale Formen wie Ellipsen, Kreise, Rechtecke und Quadrate auszuschneiden.

Danach wird eine komplexere Methode, die sich mit dem Newton-Verfahren befasst, vorgestellt. Letzteres macht es möglich, Bézierkurven aus der Freiformfläche herauszuschneiden.

3.1 Vereinfachtes Trimmen

Beim vereinfachten Trimmen gibt es drei verschiedene Verfahren. Eines für das Trimmen von Rechtecken, welches ebenfalls für Quadrate funktioniert, eines für das Trimmen von Kreisen und eines für Ellipsen.

3.1.1 Rechteck und Quadrat

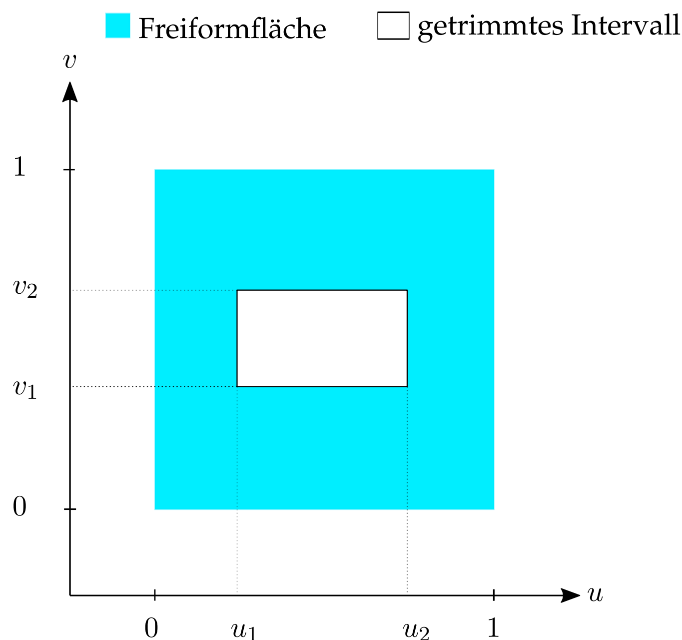

Abbildung 7: Beispiel für eine Trimmung mit Rechteck

Um bei einer Freiformfläche ein Rechteck oder gegebenermaßen ein Quadrat herauszuschneiden, werden dem Fragment-Shader ein *u-Wert* und

ein *v-Wert* aus dem TES übergeben. Diese beiden Werte haben einen *float-Wert* zwischen 0 und 1. Ebenfalls übergeben werden noch vom Nutzer frei wählbare *float-Werte* (in der Abbildung 7 dargestellt durch die Werte u_1, u_2 und v_1, v_2). Diese werden mit den *uv-Koordinaten* abgeglichen und je nachdem ob sie größer oder kleiner sind als die *uv-Koordinaten* wird gerendert. Je nach Anpassung des Intervalls, kann man ein beliebig großes Rechteck herausschneiden.

3.1.2 Kreis

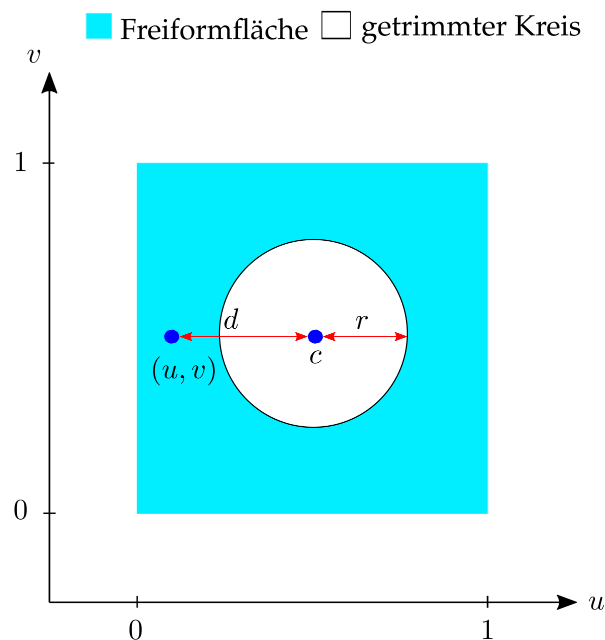

Abbildung 8: Beispiel für eine Trimmung mit Kreis

Um bei einer Freiformfläche einen Kreis herauszuschneiden kann der Nutzer im Hauptprogramm einen Wert für das Kreiszentrum auf der Oberfläche und einen Wert für den Radius des Kreises an den Fragment-Shader übergeben (c für das Zentrum und r für den Radius in Abbildung 8). Hier wird mit Hilfe des *glsl-Befehls distance()* (glsl engl. für OpenGL Shading Language) überprüft, ob die Distanz (d in Abbildung 8) zwischen den momentan zu rendernden *uv-Koordinaten* ((u, v) in Abbildung 8) und dem Kreiszentrum kleiner ist als der Kreisradius ($d < r$). Ist dies der Fall, so wird getrimmt. Andernfalls wird normal gerendert.

3.1.3 Ellipse

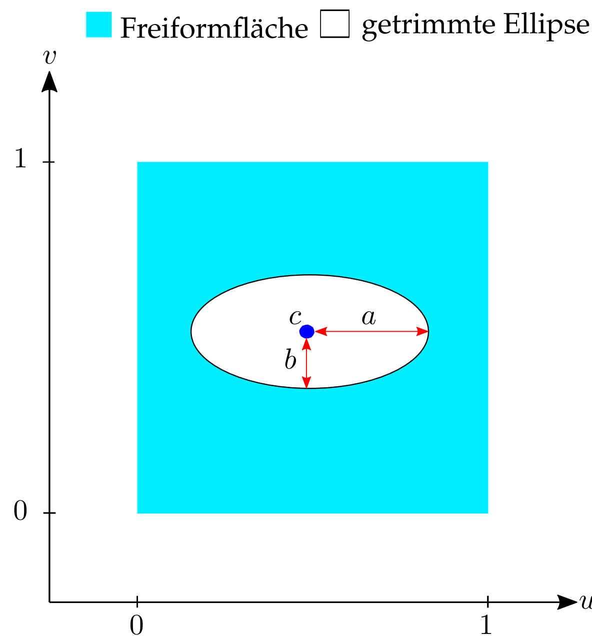

Abbildung 9: Beispiel für eine Trimmung mit Ellipse

Um bei einer Freiformfläche eine Ellipse herauszuschneiden kann der Nutzer im Hauptprogramm zwei Radien angeben (a und b in Abbildung 9). Einen horizontalen Radius (a) und einen vertikalen Radius(b). Dazu kommt noch der Mittelpunkt der Ellipse (c in Abbildung 9) und die *uv-Koordinaten* aus dem TES.

Daraufhin kann mit Hilfe folgender Gleichung getrimmt werden [12]:

$$\frac{(x - m_1)^2}{a^2} + \frac{(y - m_2)^2}{b^2} = 1 \tag{13}$$

mit:

- m_1: *x-Koordinate* des Mittelpunkts
- m_2: *y-Koordinate* des Mittelpunkts

Diese Gleichung kann folgend angepasst werden um im Fragment-Shader einen Schnittpunkttest durchzuführen:

$$\frac{(u - c_1)^2}{a^2} + \frac{(v - c_2)^2}{b^2} <= 1 \tag{14}$$

mit:

- c_1: *u-Koordinate* des Mittelpunkts

- c_2: *v-Koordinate* des Mittelpunkts
- u: *u-Koordinate* aus dem TES
- v: *v-Koordinate* aus dem TES

Diese Gleichung wird für jeden Punkt auf der Freiformfläche ausgeführt und es wird überprüft ob die zu rendernde *uv-Koordinate* die Gleichung erfüllt oder nicht. Dementsprechend wird nur gerendert, wenn die Gleichung unerfüllt bleibt.

3.2 Das Newton-Verfahren

Das *Newton-Verfahren* ist eine Technik um die Nullstellen nichtlinearer Gleichungen zu finden. Es ist leicht zu implementieren und konvergiert schnell, was es sehr effizient macht. Wegen dieser Eigenschaften wird es oft benutzt um verschiedene reale Probleme zu lösen [6].

Es wird auch noch das *Newton-Raphson-Verfahren* genannt, weil es nicht nur von Isaac Newton (1669) sondern auch von Joseph Raphson (1690) unabhängig voneinander entwickelt wurde. Beide entwickelten leicht abweichende Verfahren. Raphsons Verfahren ist das, was heute unter dem Namen *Newton-Verfahren* bekannt ist [6].

Das Newton-Verfahren funktioniert wie folgt:

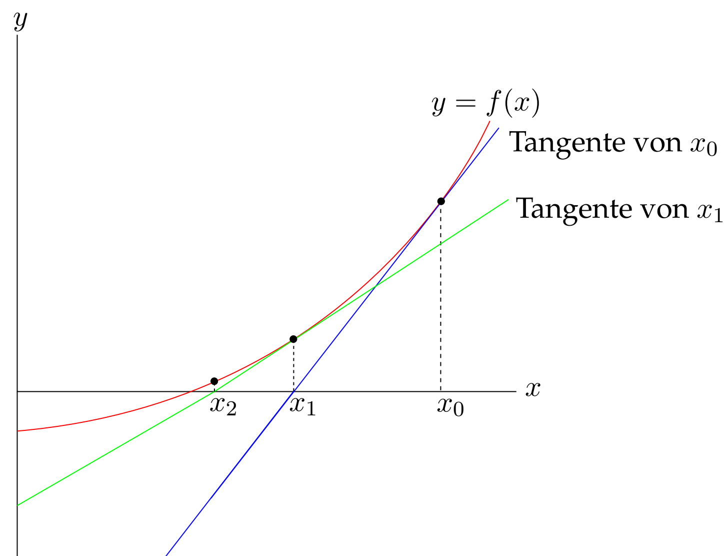

Abbildung 10: Newton-Verfahren Illustration (Abbildung nach [3])

Ausgangspunkt sind eine nicht lineare Funktion $f(x)$, von der die Nullstellen zu finden sind. Außerdem wird eine initiale Approximation x_0 (s. Abbildung 10) der Lösung benötigt. Da diese Approximation, die auch einem Zufallswert oder einer Schätzung entsprechen kann, womöglich nicht

nahe an der finalen Lösung liegt, bietet das *Newton-Verfahren* einen Weg, mit Hilfe dieser initialen Schätzung, die Nullstelle zu finden, die am nächsten an ihr liegt [3]. Dies passiert folgendermaßen:

Es wird die Tangente vom initialen Schätzwert x_0 berechnet. Diese Tangente schneidet die *x-Achse* an einer neuen Stelle, die näher an der Nullstelle liegt als x_0. Diese wird als x_1 definiert (s. Abbildung 10), x_1 ist die neue Approximation. Danach wird die Tangente von x_1 berechnet, welche sich wieder an einer neuen Stelle mit der *x-Achse* schneidet, die wiederum näher liegt als x_1. Diese wird als x_2 definiert (s. Abbildung 10). Dieses Verfahren wird rekursiv fortgesetzt, bis der Absolutwert der Differenz von zwei aufeinander folgenden Approximationen kleiner ist als ein vorher festgelegtes Epsilon ϵ.

Der, pro Rekurionsschritt, neue Approximationswert x_{n+1} wird wie folgt berechnet [13]:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{15}$$

mit:

- x_{n+1}: vorherige Approximation
- $f'(x_n)$: erste Ableitung von $f(x_n)$

Diese Rekursionsvorschrift (15) wird auch *Newton-Iteration* genannt [13].

Obwohl dieser Algorithmus schnell terminiert, hat er verschiedene Nachteile [2]:

- Wenn $f(x)$ zu komplex ist, kann es zu Schwierigkeiten bei der Berechnung von $f'(x)$ kommen.
- Wenn $f'(x) = 0$ (s. Gleichung 15), läßt sich das *Newton-Verfahren* nicht anwenden.
- Bei einer unendlich, stark oszillierenden Gleichung konvergiert der Algorithmus sehr langsam nahe einem lokalen Maximum oder lokalem Minimum.
- Die Methode kann nicht angemessen verwendet werden, wenn $f(x)$, nahe des Schnittpunktes mit der *x-Achse*, beinahe horizontal liegt.

3.2.1 Trimming mit Hilfe des Newton-Verfahrens

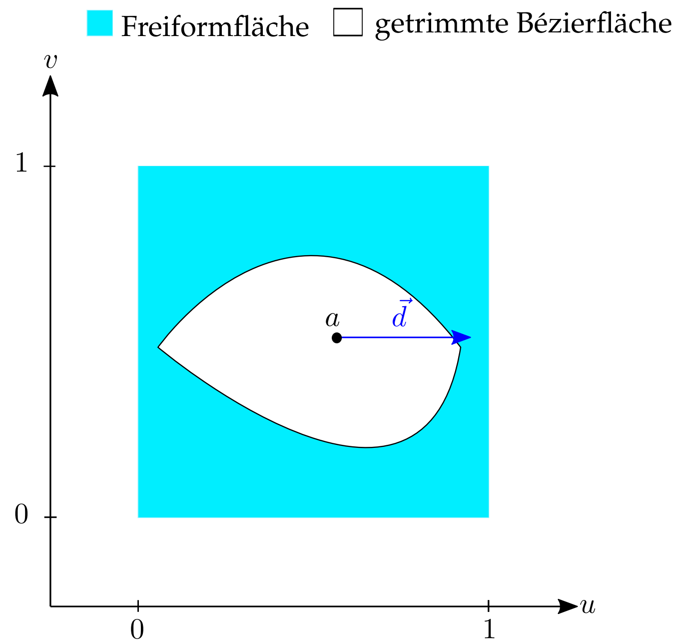

Abbildung 11: Trimmung mit Bézierkurven

Um eine Bézierfläche aus einer Freiformfläche herauszuschneiden, muss sie in zwei quadratische Bézierkurven unterteilt werden. Dabei ist zu beachten, das jeweils der erste und der letzte Kontrollpunkt von beiden Bézierkurven die gleichen sind. Somit ergeben sich im ganzen vier Kontrollpunkte, die die Bézierfläche bestimmen. Nun müssen, für die obere und die untere Hälfte der Freiformfläche, jeweils die zwei Schnittpunkte mit der oberen bzw. unteren Bézierkurve ermittelt werden. Die obere Hälfte liegt unter den zwei gemeinsamen Kontrollpunkten und die untere Hälfte ist die, die unter selbigen liegt.

Dazu muss die folgende Gleichung gelöst werden:

$$(1-t)^2 \cdot P_0 + 2 \cdot t \cdot (1-t) \cdot P_1 + t^2 \cdot P_2 = a + x \cdot \vec{d} \tag{16}$$

mit:

- $P_0, ..., P_n$: Kontrollpunkte der Bézierkurve.
- $(1-t)^2, 2 \cdot t \cdot (1-t), t^2$: Bernsteinpolynome.
- 0 <= t <= 1
- x: variabler Parameter.

- $\vec{d}$: Richtungsvektor $\binom{1}{0}$ (s. Abbildung 11, aus anschaulichen Gründen kleiner dargestellt)

Nun wird mit den folgenden drei Schritten kontrolliert, ob sich die zu rendernden *uv-Koordinaten* (s. a in Abbildung 11) in oder außerhalb der Bézierkurve befinden oder nicht:

1. In der folgenden Gleichung werden die zwei *t-Werte* (t_1 und t_2) mit Hilfe des *Newton-Verfahrens* berechnet. Als initiale Approximation wird für t_1, 1 benutzt und für t_2, 0 (darf vertauscht werden). Außerdem werden nur die *v-Koordinaten* bei dieser ersten Gleichung berücksichtigt. Die Variable x verschwindet ebenfalls, weil die *v-Koordinate* des Vektors $\vec{d} = 0$.

$$(1-t)^2 \cdot P_0^v + 2 \cdot t \cdot (1-t) \cdot P_1^v + t^2 \cdot P_2^v = a \quad (17)$$

   Da es sich bei a um eine bekannte Variable handelt (*v-Koordinate*), ist die Gleichung mit dem *Newton-Verfahren* lösbar.

2. Nun werden mit den beiden aus der Gleichung 17 berechneten Werten t_1 und t_2 jeweils ein Wert x_1 und x_2 berechnet. Dazu wird noch einmal die Gleichung 16 benutzt. Jetzt werden nur die *u-Koordinaten* berücksichtigt. Da diese bei dem Vektor $\vec{d}$ eins ist, bleibt die Variable x alleine übrig:

$$(1-t)^2 \cdot P_0^u + 2 \cdot t \cdot (1-t) \cdot P_1^u + t^2 \cdot P_2^u = a + x \quad (18)$$

   Da es sich auch hier bei a um eine bekannte Variable handelt (*u-Koordinate*) und in dem Polynom $(1-t)^2 \cdot P_0^u + 2 \cdot t \cdot (1-t) \cdot P_1^u + t^2 \cdot P_2^u$ keine Unbekannte mehr vorkommt, kann die Gleichung umgeschrieben werden.

   Setze $y = (1-t)^2 \cdot P_0^u + 2 \cdot t \cdot (1-t) \cdot P_1^u + t^2 \cdot P_2^u$, so ergibt sich:

$$x = y - a \quad (19)$$

   mit $x = x_1$ bzw $x = x_2$ (abhängig vom *t-Wert*).

   Der Wert x beschreibt eine Teillänge des Vektors $\vec{d}$. Die Distanz zwischen dem Punkt a und dem errechneten Schnittpunkt auf der Bézierkurve beträgt genau diesen Wert x.

3. Als letzter Schritt muss anhand der Werte von x_1 und x_2 ermittelt werden, ob sich der Vektor $\vec{d}$, ausgehend von Punkt a mit der Bézierkurve schneidet oder nicht. Hier wird zwischen vier Fällen unterschieden:

Erster Fall:

x_1 und x_2 sind beide positiv. Dies bedeutet, dass der Vektor $\vec{d}$ sich zweimal mit der Bézierkurve schneidet. Also wird hier gerendert, da der Punkt a außerhalb der Bézierkurve liegen muss.

Zweiter Fall:

Selbiges Szenario ist der Fall, wenn x_1 und x_2 negativ sind.

Dritter Fall:

x_1 ist positiv und x_2 ist negativ. Also wird nicht gerendert, da der Punkt innerhalb der Bézierkurve liegen muss.

Vierter Fall:

Selbiges gilt für den Fall, dass x_1 negativ ist und x_2 positiv.

Für die zweite Bézierkurve müssen die Schritte 1-3 wiederholt werden. So lassen sich beide Bézierkurven getrennt voneinander trimmen. Das gleiche Verfahren funktioniert mit kubischen Bézierkurven analog. Zu beachten ist hier nur, dass sich die Bernsteinpolynome verändern. Aus der Gleichung (16) wird somit folgende Gleichung:

$$\boldsymbol{(1-t)^3} \cdot P_0 + \boldsymbol{3 \cdot t \cdot (1-t)^2} \cdot P_1 + \boldsymbol{3 \cdot t^2 \cdot (1-t)} \cdot P_2 + \boldsymbol{t^3} \cdot P_3 = a + x \cdot \vec{d} \quad (20)$$

Für das Rendern von einer Bézierfläche innerhalb einer anderen Bézierfläche (s. Abbildung 12), kann die gleiche Methode verwendet werden. Es ist zu beachten, dass es vier verschiedene Bézierkurven gibt. Somit braucht es nicht nur zwei sondern vier *t-Werte* und vier *x-Werte*. Das Verfahren muss also doppelt angewandt werden. Beim Vergleich zwischen den *x-Werten* wird ebenfalls nur gerendert, wenn einer der zwei folgenden Fällen eintritt:

Erster Fall:

$x_1 > 0 \land x_2 < 0 \land \neg(x_3 > 0 \land x_4 < 0)$

zweiter Fall:

$x_1 < 0 \land x_2 > 0 \land \neg(x_3 < 0 \land x_4 > 0)$

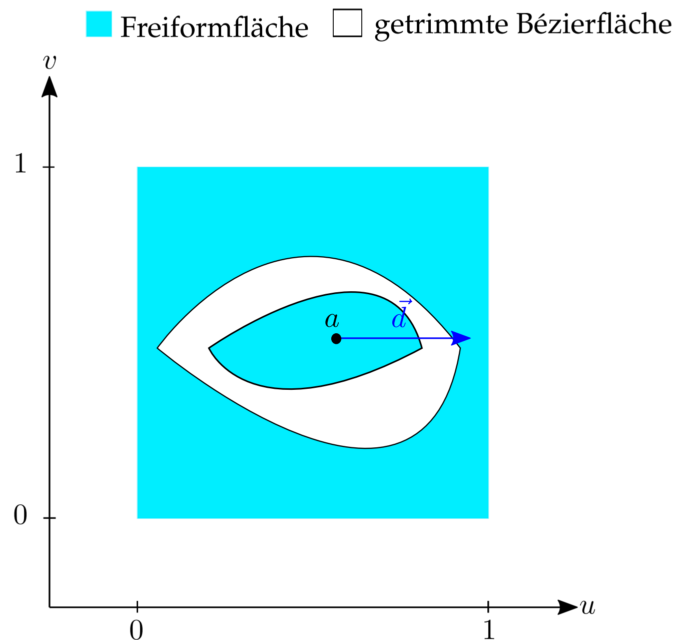

Abbildung 12: Trimmung mit verschachtelten Bézierkurven

Auch diese Methode ist problemlos mit kubischen Bézierkurven durchführbar.

4 Implementierung

Im folgenden Kapitel wird die Implementierung der Bézierflächen, sowie das Trimmen der einfachen Formen und der Bézierflächen, beschrieben. Auf den TCS und den TES wird dabei genauer eingegangen. Das Trimmen findet im Fragment-Shader statt und wird dementsprechend in einem eigenen Unterkapitel beschrieben.

4.1 Tessellierung

Der Tessellation-Shader wurde mit der OpgenGL-Version 4.0 eingeführt. Er steht also nur auf neuerer Hardware zur Verfügung. Er besteht aus drei Shader-Stufen. Davon sind jedoch nur zwei programmierbar.

Die erste Stufe, der TCS, wird sofort nach dem Vertex-Shader aufgerufen. Dieser ist programmierbar. Hier wird bestimmt, welches Tessellierungs-Level jedes Primitiv haben wird. Es können noch andere Transformationen mit den eingelesenen Patches vorgenommen werden. Zum Beispiel kann die Größe der Patches verändert werden, indem Vertices hinzugefügt oder entfernt werden. Es ist jedoch nicht möglich einen Patch wegzulassen oder

mehrere Patches darzustellen. Ein Patch ist ein Primitiv was aus einer vorher festgelegten Anzahl n an Vertices besteht. Somit formen jede n Vertices einen Patch. Der TCS ist optional. Falls kein TCS programmiert ist, wird ein Standard-TCS verwendet [9]. Die genauere Programmierung des TCS, für das Rendering von Freiformflächen, wird in Kapitel 4.1.1 beschrieben.

Im Anschluss an den TCS wird der sogenannte Tesselator aufgerufen. Dieser ist nicht programmierbar. In diesem Schritt werden die neuen Primitive für die jeweiligen Patches generiert. Die Darstellung dieser Primitive ist von folgenden Faktoren abhängig [9]:

- Dem Tessellation-Level, der im TCS festegelegt wurde oder dem Standardwert entspricht.
- Dem Abstand zwischen den tesselierten Vertices.
- Den eingelesenen Primitiv-Typen (*quads* oder *triangles*).
- Der Reihenfolge der Generierung der Primitive (*cw* (engl. clockwise) oder *ccw* (engl. counterclockwise)).

Die drei letzten Faktoren werden im TES festgelegt (s. Kapitel 4.1.2).

Nach dem Tesselator wird der programmierbare TES aufgerufen. Hier werden die oben stehenden Werte festgelegt und der eigentliche Algorithmus programmiert, der die neuen Positionen der erzeugten Vertices berechnet. Dieser ist, im Gegensatz zum TCS, obligatorisch. Falls keiner vorhanden ist findet keine Tessellierung statt [9].

4.1.1 Tessellation-Control-Shader

Der Tessellation-Control-Shader, der für das Rendern von Freiformflächen benötigt wird, unterscheidet sich im Grunde nicht von einem herkömmlichen TCS. Er sieht wie folgt aus:

```
1  #version 450
2
3  layout(vertices = 16) out;
4  uniform float tessLevel;
5
6  void main()
7  {
8      gl_out[gl_InvocationID].gl_Position =
9      gl_in[gl_InvocationID].gl_Position;
10     if (gl_InvocationID == 0) {
11             gl_TessLevelInner[0] = tessLevel;
12             gl_TessLevelInner[1] = tessLevel;
13             gl_TessLevelOuter[0] = tessLevel;
```

```
14            gl_TessLevelOuter[1] = tessLevel;
15            gl_TessLevelOuter[2] = tessLevel;
16            gl_TessLevelOuter[3] = tessLevel;
17     }
18 }
```

Listing 1: Tessellation-Control-Shader

In Zeile 3 wird die Anzahl an Vertices pro Patch angegeben. Diese entspricht hier der Anzahl der Kontrollpunkte (4×4) der Freiformfläche. Der hier angegebene Wert muss gleich dem zweiten Parameter der *glPatchParameter()-Methode* und dem dritten Parameter der *glDrawArrays()-Methode* sein, die in der Klasse *CVK_FreeFormSurface.cpp* zu finden sind:

```
1 //rendering method
2 void CVK::FreeFormSurface::render() {
3   glPatchParameteri(GL_PATCH_VERTICES, 16);
4   glDrawArrays(GL_PATCHES, 0, 16);
5 }
```

Listing 2: Render-Methode

Die Uniform-Variable *tessLevel* wird im Hauptprogramm je nach Belieben festgelegt. Für eine detailreiche Darstellung der Freiformfläche ist ein Wert von mindestens 16 zu empfehlen.

In Zeile 8 und 9 werden die Positionen der eingelesenen Vertices festgelegt, bevor sie im TES verändert werden können. Die im TCS eingebettete Variable *gl_InvocationID* enthält den Index des jeweiligen Vertex im momentanen Patch. Es handelt sich dabei um einen Integer-Wert zwischen 0 und der Anzahl an Vertices pro Patch minus 1 (in diesem Fall 0-15). Falls es sich dabei um den ersten Vertex handelt (*gl_InvocationID == 0*), wird das Tessellation-Level der Freiformfläche angepasst. Dabei wird das zu rendernde Rechteck folgendermaßen unterteilt:

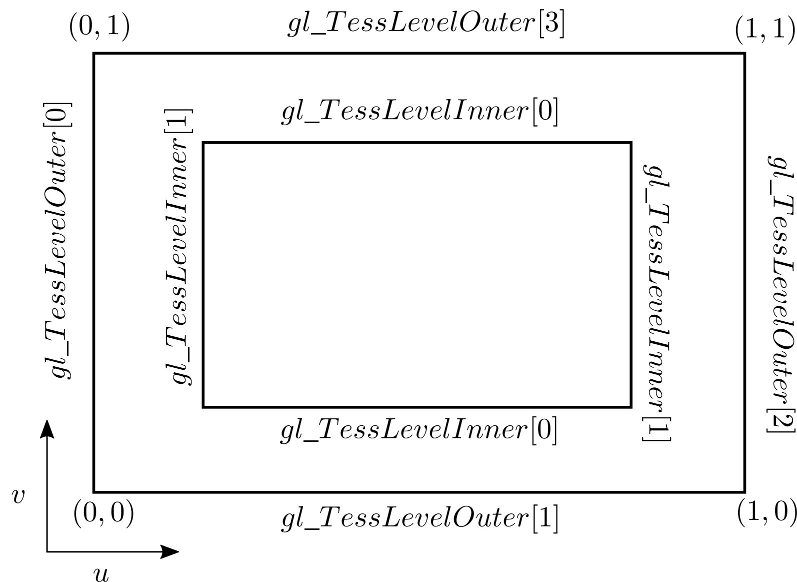

Abbildung 13: Tessellation-Werte für *Quad-Tessellation* (Abbildung nach [4])

Jeder Kante wird dabei ein Wert zugewiesen, der beschreibt, wie genau unterteilt wird. Um eine gleichmäßige Struktur zu erhalten, wird jeweils das gleiche Tessellation-Level pro Kante benutzt.

4.1.2 Tessellation-Evaluation-Shader

Der Tessellation-Evaluation-Shader für das Rendern von Freiformflächen sieht folgerndermaßen aus:

```
1  #version 450
2
3  layout(quads, equal_spacing, ccw) in;
4
5  //uniform variables
6  uniform mat4 viewMatrix;
7  uniform mat4 modelMatrix;
8  uniform mat4 projectionMatrix;
9
10 //out vectors
11 out vec3 passNormal;
12 out vec2 passTessCoord;
13 out vec2 passUVCoord;
14 out vec4 position;
```

Listing 3: Tessellation-Evaluation-Shader (Teil 1)

Die Zeile 3 legt die drei in Kapitel 4.1 beschriebenen Faktoren fest. Der erste Parameter sagt aus, dass es sich bei den Patches um sogenannte *Quads* handelt. Der zweite Parameter definiert den Abstand zwischen den neu generierten Vertices. Da dieser überall gleich sein soll wird der Parameter *equal_spacing* benutzt. Der dritte Parameter gibt die Reihenfolge der Generierung der Primitive an.

Zeile 6 - 8 definiert die übergebenen Uniform-Matrizen die nötig sind und Zeile 11 - 14 die übergebenen Vektoren.

Die nächste Methode ist der de-Casteljau Algorithmus:

```
1  //de-Casteljau algorithm
2  vec4 deCasteljau(in vec4 p0, in vec4 p1, in vec4 p2,
3    in vec4 p3, inout vec3 tangU, inout vec3 tangV,
4    in float t, int index)
5  {
6    vec4 cPoints[4] = vec4[4](p0, p1, p2, p3);
7    for (int i = 2; i >= 0; i--) {
8      for (int j = 0; j <= i; j++) {
9         cPoints[j] = (1-t)*cPoints[j]
10                        +t*cPoints[j+1];
11     }
12     if(index == 1 && i == 1) tangU =
13       normalize((cPoints[1].xyz - cPoints[0].xyz));
14     if(index == 0 && i == 1) tangV =
15       normalize((cPoints[1].xyz - cPoints[0].xyz));
16   }
17   return cPoints[0];
18 }
```

Listing 4: Tessellation-Evaluation-Shader (Teil 2)

Diese Methode gibt für einen gegebenen *t-Wert* und die vier Kontrollpunkte einer Bézierkurve einen Punkt auf dieser zurück. Hier wird der Algorithmus nur eindimensional angewendet. In Zeile 12 - 15 werden, falls es sich um den vorletzten Schritt des Algorithmus handelt (*i == 1*), die Tangenten berechnet. Einmal die Tangente in *u-Richtung* (*tangU*) und einmal die in *v-Richtung* (*tangV*). Um zu wissen, welche Tangente berechnet werden soll, wird ein Hilfsparameter *index* übergeben. Nach Durchlaufen der Methode, sind neben dem Rückgabewert, die Tangenten in Variablen gespeichert.

Um eine Freiformfläche zu rendern, muss dieser Algorithmus doppelt angewandt werden:

```
1
2  //dual de-Casteljau algorithm
3  void doubleDeCasteljau(float u, float v,
4    out vec4 position, out vec3 tangU, out vec3 tangV)
```

```
5  {
6    vec4 uPositions[4];
7    for(int i = 0; i <= 12; i += 4){
8      uPositions[i/4] = deCasteljau(gl_in[i].gl_Position,
9      gl_in[i+1].gl_Position, gl_in[i+2].gl_Position,
10     gl_in[i+3].gl_Position, tangU, tangV, u, 0);
11   }
12   position = deCasteljau(uPositions[0], uPositions[1],
13   uPositions[2], uPositions[3], tangU, tangV, v, 1);
14 }
```

Listing 5: Tessellation-Evaluation-Shader (Teil 3)

Dazu werden zunächst die Punkte auf der Bézierkurve in *u-Richtung* berechnet und in einem Array *uPositions[]* gespeichert. Anschließend wird der de-Casteljau Algorithmus ein zweites Mal in *v-Richtung*, mit den zuvor berechneten Punkten aus *uPositions[]* , ausgeführt. Der finale Punkt wird in der Variablen *position* gespeichert, der später *gl_Position* zugewiesen wird.

```
1
2  void main()
3  {
4    //defining uv variables
5    float u = gl_TessCoord.x;
6    float v = gl_TessCoord.y;
7    vec3 tangU, tangV;
8
9    /*generating interpolated de-Casteljau positions
10   once in the u-direction and once in the v-direction*/
11   doubleDeCasteljau(u, v, position, tangU, tangV);
12
13   //calculating the normal
14   passNormal = normalize(cross(tangU, tangV));
15
16   /*passing tessellation coordinates necessary for
17   the simple interval trimming*/
18   passTessCoord = vec2(u, v);
19   //passing uv-coordinates necessary for the textures
20   passUVCoord = vec2(u,v);
21   gl_Position = position;
22 }
```

Listing 6: Tessellation-Evaluation-Shader (Teil 4)

In der *main-Methode* wird zunächst der *u-Wert* und der *v-Wert* definiert. Dazu werden die in den TES eingebetteten Variablen *gl_TessCoord.x* und

gl_TessCoord.y benutzt. Diese Variablen haben immer einen float-Wert zwischen 0 und 1. Als nächstes wird der *de-Casteljau Algorithmus* für die Interpolation der Punkte angewendet, die Normalen der Fläche werden mit Hilfe des Kreuzproduktes der beiden Tangenten berechnet und die Tessellierungs-Koordinaten u, v werden an den Fragment-Shader weitergegeben. Diese werden ein zweites Mal in einer anderen Variable gespeichert um im späteren Verlauf, im Fragment-Shader, die Übersicht besser zu behalten.

4.2 Trimmung

In diesem Kapitel wird die Implementation des Trimmungverfahrens erläutert. Um Effizienz zu bewahren, findet das Trimmen im Fragment-Shader statt.

4.2.1 Vereinfachtes Trimmen

Im folgenden Code befinden sich die *if-Abfragen* für das Trimmen von Rechtecken, Kreisen und Ellipsen:

```
1   //circle trimming
2   if(circBool == 1 && (distance(passTessCoord,
3     trimCenter) < trimRadius))
4     discard;
5   //interval trimming
6   if(intervalBool == 1 && (u >= trimValuesx.x && u
7     <= trimValuesx.y && v >= trimValuesy.x && v
8     <= trimValuesy.y))
9     discard;
10  //ellipse trimming
11  if(ellipseBool == 1 && pow(u -
12    trimEllipseCenter.x, 2)/pow(trimRadius1, 2)
13    + pow(v -
14    trimEllipseCenter.y, 2)/pow(trimRadius2, 2)
15    <= 1.0)
16    discard;
17  /*color definition with phong shading with two
18  lightsoures*/
19  fragmentColor = vec4(diffuseColor*ambientColor
20    + phi * diffuseColor + psi * specularColor
21    + phi2 * diffuseColor + psi2 * specularColor,
22    1.0f);
```

Listing 7: Trimmungverfahren für einfache Trimmformen (Fragment-Shader)

In der ersten *if-Abfrage* wird mit Hilfe der glsl-Methode *disance()* abgefragt, ob sich die momentan zu rendernden Koordination *passTessCoord* im Kreis befinden oder nicht. Ist dies der Fall, so wird mit dem Befehl *discard*, der Fragment-Shader abgebrochen und es wird nichts gerendert. Falls die Bedingung nicht erfüllt ist, wird normal weitergerendert (s. Zeile 19).

Die Variable *circBool* ist ein *boolean* der mit einer Methode im Hauptprogramm auf *true* gesetzt werden muss, damit das Trimmen möglich ist. Dies dient dem Trimmen von mehreren und verschiedenen Formen in der selben Freiformfläche. Die Variablen *trimCenter* und *trimRadius* werden ebenfalls vom Benutzer im Hauptprogramm mit einer Methode übergeben.

In der zweiten *if-Abfrage* wird das *Intervalltrimming* ausgeführt (Rechtecke und Quadrate). Auch hier muss ein *boolean* auf true gesetzt sein. Es folgt die einfache Überprüfung, mit Hilfe der *u-Koordinate* und *v-Koordinate*, ob diese sich im getrimmten Intervall befinden oder nicht. Der Vektor *trimValuesx* entpsricht dabei dem horizontalen Intervallbereich und *trimValuesy* dem vertikalen.

In der letzten *if-Abfrage*, befindet sich in der Bedingung die mathematische Formel einer Ellipse (s. Kaptiel 3.1.3), die erfüllt werden muss, damit der Rendervorgang abgebrochen wird. Auch hier ist ein *boolean* erforderlich. Um mehrere Formen in einer Freiformfläche zu trimmen, müssen an dieser Stelle mehrere *if-Abfragen* mit entsprechenden Variablen kontrolliert werden.

4.2.2 Trimmen mit Hilfe des Newton-Verfahrens

Für das Trimmen mit Hilfe des Newton-Verfahrens müssen zunächst deren Funktionen definiert werden:

```
1  //define function
2  float f(float t, float v)
3  {
4    return pow((1-t), 2)*p0.y + 2*t*(1-t)*p1.y +
5      pow(t, 2)*p2.y - v;
6  }
7
8  //define derivative function
9  float df(float t)
10 {
11   return /*write derivative function of f1 here
12         for example: return -((8*t - 4)/5);*/;
13 }
14
15 //define function
16 float f2(float t, float v)
```

```
{
  return pow((1-t), 2)*p0.y + 2*t*(1-t)*p3.y +
    pow(t, 2)*p2.y - v;
}

//define derivative function
float df2(float t)
{
  return /*write derivative function of f2 here*/;
}
```

Listing 8: Funktionen für das Newton-Verfahren (quadratisch) (Fragment-Shader) Code nach [1]

Für jede Bézierkurve muss an dieser Stelle eine Funktion definiert werden. Sollen also zwei Bézierkurven getrimmt werden, werden vier Funktionen benötigt. Zum ersten die normale Funktion mit den Bernsteinpolynomen zweiten Grades für quadratische Bézierkurven und zum zweiten dessen Ableitung ersten Grades.

Bei kubischen Bézierkurven passen sich die Bernsteinpolynome an:

```
//define function
float f3(float t, float v)
{
  return pow((1 - t), 3)*p0.y + 3*t*pow((1-t), 2)*
    p1.y+3*pow(t,2)*(1-t)*p2.y+pow(t,3)*p3.y - v;
}
//define derivative function
float df3(float t)
{
  return /*write derivative function of f3 here*/;
}
//define function
float f4(float t, float v)
{
  return pow((1 - t), 3)*p0.y + 3*t*pow((1-t), 2)*
    p4.y+3*pow(t,2)*(1-t)*p5.y+pow(t,3)*p3.y - v;
}
//define derivative function
float df4(float t)
{
  return /*write derivative function of f4 here*/;
}
```

Listing 9: Funktionen für das Newton-Verfahren (kubisch) (Fragment-Shader). Code nach [1]

Das Trimmen von Bézierkurven in anderen Bézierkurven funktioniert in diesem Schritt analog. Es müssen nur jeweils vier Funktionen plus vier Ableitungen definiert werden.

Das Newton-Verfahren sieht wie folgt aus:

```
//newton iteration
float newtonitr(float x0, float v)
{
  float h, x1, epsilon;
  epsilon = 0.001;

  for(int i = 0; i < 10; i++)
  {
    h = f(x0, v)/df(x0);
    x1 = x0 - h;

    if(abs(h) < epsilon)
    {
      return x1;
    }
    x0 = x1;
  }
}
```

Listing 10: Newton-Verfahren. Code nach [1]

Der Parameter v entspricht der *v-Koordinate (passTessCoord.y)*, die in der Methode *newtonIterationTrimming* (s. Listing 11), übergeben wird.

Der Parameter $x0$ entspricht dem initialen Schätzwert der Methode. Bei dem hier benutzten Verfahren ein *float-Wert* von 0.0 oder 1.0 (s. Listing 11).

In Zeile 4 werden drei *float-Werte* deifniert: h, $x1$ und $epsilon$. Letzterem wird ein beliebiger Genauigkeitswert zugewiesen. Je kleiner dieser Wert ist, desto genauer wird das Resultat, die Nullstelle, berechnet. Allerdings steigt damit auch die Berechnungszeit und die Effizienz leidet darunter. Erfahrungsgemäß ist ein Wert von 0.001 ausreichend und führt zu guten Ergebnissen.

Die *for-Schleife* durchläuft maximal zehn Durchgänge. Auch hier kann ein beliebig hoher Wert gewählt werden. In der Vorschleife wird der Algorithmus, der in Kapitel 3.2 beschrieben wird, angewandt. Sobald der Absolutwert des gefundenen Approximationswertes h kleiner ist als der vorher definierte Wert $epsilon$, gibt die Methode diesen Wert als Endresultat zurück.

Diese Methode wird in der folgenden Methode benutzt um das Trimmen auszuführen:

```
//trimming
void newtonIterationTrimming(float u, float v)
{
  //newton iteration trimming for first bezier curve
  float t1 = newtonitr(1.0, v);
  float t2 = newtonitr(0.0, v);

  float x1 = pow((1-t1), 2)*p0.x + 2*t1*(1-t1)*p1.x
    + pow(t1, 2)*p2.x - u;
  float x2 = pow((1-t2), 2)*p0.x + 2*t2*(1-t2)*p1.x
    + pow(t2, 2)*p2.x - u;

  if(x1 > 0.0 && x2 < 0.0 && t1 >= 0.0 && t1 <= 1.0
    && t2 <= 1.0 && t2 >= 0.0) discard;

  //newton iteration trimming for second bezier curve
  t1 = newtonitr2(1.0, v);
  t2 = newtonitr2(0.0, v);

  x1 = pow((1-t1), 2)*p0.x + 2*t1*(1-t1)*p3.x
    + pow(t1, 2)*p2.x - u;
  x2 = pow((1-t2), 2)*p0.x + 2*t2*(1-t2)*p3.x
    + pow(t2, 2)*p2.x - u;

  if(x1 > 0.0 && x2 < 0.0 && t1 >= 0.0 && t1
    <= 1.0 && t2 <= 1.0 && t2 >= 0.0)
  discard;
}
```

Listing 11: Trimmung mit dem Newton-Verfahren (Teil 1)

In Zeile 5 und 6 werden die zwei *t-Werte* berechnet, die für den weiteren Verlauf des Algorithmus aus Kapitel 3.2 benötigt werden. Für diese Berechnung wird das Newton-Verfahren ausgeführt. Einmal mit einem *t-Wert* von 1.0 und einmal mit einem *t-Wert* von 0.0. Mit diesen zwei Werten, können die Werte $x1$ und $x2$, mit Hilfe der Bernsteinpolynome berechnet werden, die für die Bedingung in der *if-Abfrage* benötigt werden.

Dieses Verfahren wird einmal für die erste und einmal für die zweite Bézierkurve ausgeführt. Die Methode *newtonitr2* in Zeile 17 und 18 ist identisch mit der Methode *newtonitr*, nur dass die in der Methode verwendeten Funktion f und df variieren, da es sich um eine andere Bézierkurve handelt. Folglich wird der Rendervorgang abgebrochen, sobald die Bedingungen aus Kapitel 3.2.1 erfüllt werden. Die Bedingungen $t1 >= 0.0 \&\& t1 <= 1.0 \&\& t2 <= 1.0 \&\& t2 >= 0.0$ sind notwendig, damit nicht unterhalb/ober-

halb der beiden Eckkontrollpunkten der oberen/unteren Bézierkurve getrimmt wird.

Das Trimmen von Bézierkurven in anderen Bézierkurven funktioniert analog, nur müssen anstatt von zwei *x-Werten*, vier *x-Werte* und dementsprechend auch vier *t-Werte* berechnet werden:

```
void newtonIterationTrimming2(float u, float v)
{
  //newton iteration trimming for first bezier curve
  float t1 = newtonitr1(1.0, v);
  float t2 = newtonitr1(0.0, v);
  float t3 = newtonitr11(1.0, v);
  float t4 = newtonitr11(0.0, v);
  float x1 = pow((1-t1), 2)*p0.x + 2*t1*(1-t1)*p1.x
    + pow(t1, 2)*p2.x - u;
  float x2 = pow((1-t2), 2)*p0.x + 2*t2*(1-t2)*p1.x
    + pow(t2, 2)*p2.x - u;
  float x3 =  pow((1-t3), 2)*p4.x + 2*t3*(1-t3)*p5.x
    + pow(t3, 2)*p6.x - u;
  float x4 =  pow((1-t4), 2)*p4.x + 2*t4*(1-t4)*p5.x
    + pow(t4, 2)*p6.x - u;

  if((x1 > 0.0 && x2 < 0.0 && t1 >= 0.0 && t1 <= 1.0
    && t2 <= 1.0 && t2 >= 0.0 && !(x3 > 0.0 &&
     x4 < 0.0)))  discard;
  //newton iteration trimming for second bezier curve
  t1 = newtonitr2(1.0, v);
  t2 = newtonitr2(0.0, v);
  t3 = newtonitr12(1.0, v);
  t4 = newtonitr12(0.0, v);

  x1 = pow((1-t1), 2)*p0.x + 2*t1*(1-t1)*p3.x
    + pow(t1, 2)*p2.x - u;
  x2 = pow((1-t2), 2)*p0.x + 2*t2*(1-t2)*p3.x
    + pow(t2, 2)*p2.x - u;
  x3 = pow((1-t3), 2)*p4.x + 2*t3*(1-t3)*p7.x
    + pow(t3, 2)*p6.x - u;
  x4 = pow((1-t4), 2)*p4.x + 2*t4*(1-t4)*p7.x
    + pow(t4, 2)*p6.x - u;
  if((x1 > 0.0 && x2 < 0.0 && t1 >= 0.0 && t1 <= 1.0
    && t2 <= 1.0 && t2 >= 0.0 && !(x3 > 0.0 &&
    x4 < 0.0)))  discard;
}
```

Listing 12: Trimmung mit dem Newton-Verfahren (Teil 2)

Die Code-Abschnitte Listing 11 und Listing 12 sind für das Trimmen von quadratischen Bézierkurven. Mit kubischen Bézierkurven funktionert der Algorithmus auf die gleiche Weise, nur müssen dann die kubischen Bernsteinpolynome verwendet werden.

4.2.3 Fragment-Shader

Der Fragment-Shader setzt sich aus dem Code-Abschnitt Listing 7 und folgendem Code zusammen:

```
1  #version 450
2
3  //ssbo
4  layout(std430, binding = 1) buffer trimPointsSsbo
5  {
6    vec2 trimPoints_data[];
7  };
8
9  in vec3 passNormal;
10 in vec4 position;
11 in vec2 passTessCoord;
12 in vec2 passUVCoord;
13
14 //uniform variables
15 uniform sampler2D colortexture;
16 //interval trimming
17 uniform vec2 trimValuesx;
18 uniform vec2 trimValuesy;
19 //circle trimming
20 uniform float trimRadius;
21 uniform vec2 trimCenter;
22 //ellipse trimming
23 uniform float trimRadius1;
24 uniform float trimRadius2;
25 uniform vec2 trimEllipseCenter;
26 //booleans
27 uniform int circBool;
28 uniform int intervalBool;
29 uniform int ellipseBool;
30 uniform int bezierBool;
31 uniform int bezierBool2;
32 uniform int cubicBezierBool;
33 uniform int cubicBezierBool2;
34
```

```
35 out vec4 fragmentColor;
36
37 //bezier trimming control points
38 vec2 p0 = trimPoints_data[0];
39 vec2 p1 = trimPoints_data[1];
40 vec2 p2 = trimPoints_data[2];
41 vec2 p3 = trimPoints_data[3];
42 vec2 p4 = trimPoints_data[4];
43 vec2 p5 = trimPoints_data[5];
44 vec2 p6 = trimPoints_data[6];
45 vec2 p7 = trimPoints_data[7];
46 vec2 p8 = trimPoints_data[8];
47 vec2 p9 = trimPoints_data[9];
48 vec2 p10 = trimPoints_data[10];
49 vec2 p11 = trimPoints_data[11];
50
51 void main(){
52   //phong shading
53   vec3 lightPosition = vec3(0.0, 10.0, 0.0);
54   vec3 lightPosition2 = vec3(0.0, -10.0, 0.0);
55   vec3 lightVector = normalize(lightPosition
56     - position.xyz);
57   vec3 lightVector2 = normalize(lightPosition2
58     - position.xyz);
59
60   vec3 eye = normalize(-position.xyz);
61   vec3 reflection = normalize(reflect(-lightVector,
62     passNormal));
63   vec3 reflection2 = normalize(reflect(-lightVector2,
64     passNormal));
65
66   float phi = max(dot(passNormal, lightVector), 0);
67   float psi = pow(max(dot(reflection, eye), 0), 150);
68   float phi2 = max(dot(passNormal, lightVector2), 0);
69   float psi2 = pow(max(dot(reflection2, eye), 0), 150);
70
71   //defining colors
72   vec3 ambientColor = vec3(0.1, 0.1, 0.1);
73   vec3 diffuseColor = vec3(texture(colortexture,
74         passUVCoord).rgb);
75   vec3 specularColor = vec3(1.0, 1.0, 1.0);
76
77   //trimming
78   float u = passTessCoord.x;
```

```
79    float v = passTessCoord.y;
80
81    if(bezierBool == 1){
82      newtonIterationTrimming(u, v);
83    }
84    if(bezierBool2 == 1){
85      newtonIterationTrimming2(u, v);
86    }
87    if(cubicBezierBool == 1){
88      newtonIterationTrimmingCubic(u, v);
89    }
90    if(cubicBezierBool2 == 1){
91      newtonIterationTrimmingCubic2(u, v);
92    }
93  }
```

Listing 13: Trimmung mit dem Newton-Verfahren (Teil 3)

Zeile 4 - 7 werden für den Ssbo (Shader Storage Buffer Object) benötigt. Die Kontrollpunkte der zu trimmenden Bézierkurven werden in Zeile 38 - 49 mit einem Ssbo definiert. Es können insgesamt 12 Kontrollpunkte sein, sofern eine kubische Bézierkurve in einer anderen kubischen Bézierkurve gerendert wird.

In Zeile 9 - 12 werden die Vektoren aus dem TES eingelesen und anschließend die uniform-Variablen definiert.

Der erste Teil der main-Methode (Zeile 53 - 75) ist für das Phong-Shading notwendig. Danach werden die *uv-Koordinaten* definiert und zum Schluss erfolgt eine *if-Abfrage*, die den selben Zweck hat wie die aus dem vereinfachten Trimmungsverfahren, nach dessen Erfüllung das Newton-Verfahren ausgeführt wird.

4.3 Anlegen einer neuen Freiformfläche

Mit der Klasse *CVK::FreeFormSurface* wird im Hauptprogramm eine neue Freiformfläche mit entsprechender Textur angelegt. Die Generierung der Textur stammt aus dem CVK:

```
1  CVK::FreeFormSurface *freeform;
2  CVK::Texture *brickTex;
```

Listing 14: Anlegen einer Freiformfläche (Teil 1)

In der *initScene()-Methode* wird die Freiformfläche und die Textur erzeugt. Dabei kann entweder der Standard-Konstruktor benutzt werden oder es können beliebige Kontrollpunkte als Parameter definiert werden:

```
void initScene()
{
  freeform = new CVK::FreeFormSurface();
  /*oder
  freeform = new CVK::FreeFormSurface(glm::vec3 p0,
    glm::vec3 p1, glm::vec3 p2, glm::vec3 p3,
    glm::vec3 p4,  glm::vec3 p5, glm::vec3 p6,
    glm::vec3 p7, glm::vec3 p8, glm::vec3 p9,
    glm::vec3 p10, glm::vec3 p11, glm::vec3 p12,
    glm::vec3 p13, glm::vec3 p14, glm::vec3 p15) */
  brickTex = new CVK::Texture(RESOURCES_PATH
    "/brick.bmp");
}
```

Listing 15: Anlegen einer Freiformfläche (Teil 2)

In der main-Methode wird unter anderem die Model-Matrix festgelegt, Transformationen ausgeführt und die Kontrollpunkte können verschoben werden:

```
    //render freeform
    glm::mat4 modelMatrix = glm::mat4(1.0f);
    modelMatrix = glm::translate(modelMatrix,
      glm::vec3(0.0, 1.0, 0.0));
    freeform->setModelMatrix(shaderHandle,
      modelMatrix);
    freeform->setSpecificControlPoint(glm::vec3(-0.17,
      0.5, 0.17), 5);
    freeform->setSpecificControlPoint(glm::vec3(0.17,
      0.5, 0.17), 6);
    freeform->setSpecificControlPoint(glm::vec3(-0.17,
      0.5, -0.17), 9);
    freeform->setSpecificControlPoint(glm::vec3(0.17,
      0.5, -0.17), 10);
    freeform->enableCircularTrimming(shaderHandle);
    freeform->circularTrimming(0.2f,
      glm::vec2(0.5, 0.5),  shaderHandle);
    cubeTex->bind();
    freeform->render();
    freeform->disableCircularTrimming(shaderHandle);
```

Listing 16: Rendern der Freiformfläche

Außerdem muss das Trimmen aktiviert und deaktiviert werden und die Textur wird gebunden. Schlussendlich kann mit der *render-Methode* gerendert werden.

Um eine Bézierkurve zu trimmen, werden die Kontrollpunkte der Bézierkurven in der main-Methode mit der *setBezierCurveTrimming()-Methode* festgelegt. Dabei ist es wichtig, dass die Kontrollpunkte von links beginnend im Uhrzeigersinn übergeben werden:

```
1 freeform->setBezierCurveTrimming(p0, p1, p2, p3);
```

Listing 17: Bézierkurventrimmung (einzel, quadratisch)

Bei $p0, p1, p2, p3$ handelt es sich um zweidimensionale Vektoren.

Das Gleiche gilt auch für kubische Bézierkurven. Beim Trimmen von einer Bézierkurve in einer anderen, werden die Kontrollpunkte beginnend mit den äußeren Kurven im Uhrzeigersinn definiert. Gefolgt von den inneren.

5 Ergebnisse und Grenzen

5.1 Ergebnisse

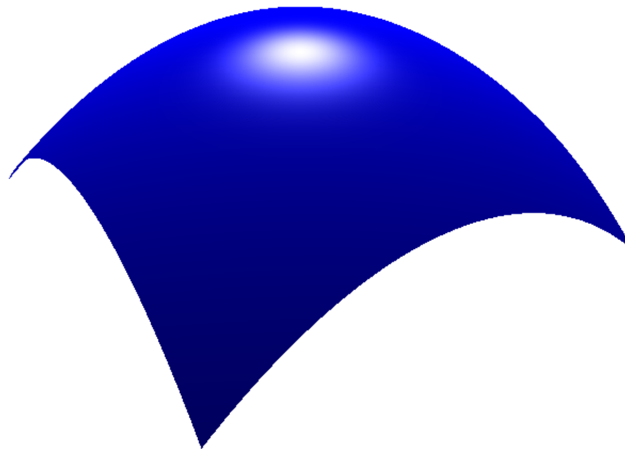

Abbildung 14: Bézierfläche mit 16 Kontrollpunkten

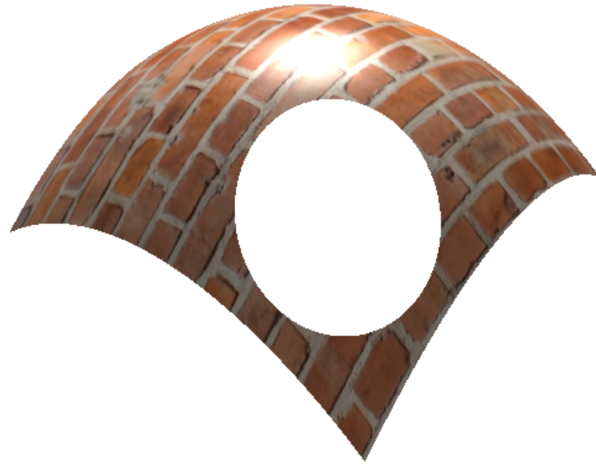

Abbildung 15: Bézierfläche mit Trimmung eines Kreises und Textur

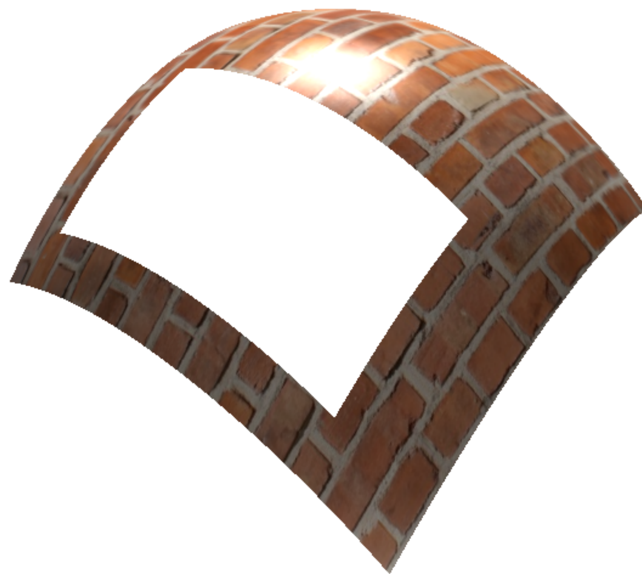

Abbildung 16: Bézierfläche mit Trimmung eines Rechteckes (Intervall) und Textur

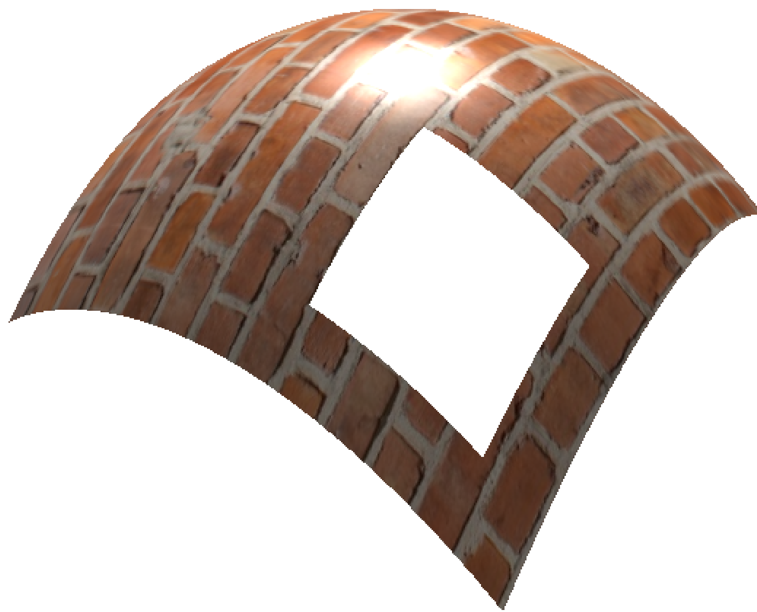

Abbildung 17: Bézierfläche mit Trimmung eines Quadrates (Intervall) und Textur

Abbildung 18: Bézierfläche mit Trimmung einer Ellipse und Textur

Abbildung 19: Bézierfläche mit Trimmung von zwei Ellipsen und Textur

Abbildung 20: Bézierfläche mit mehreren Trimmflächen (Quadrat, Rechteck, Kreis, Ellipse)

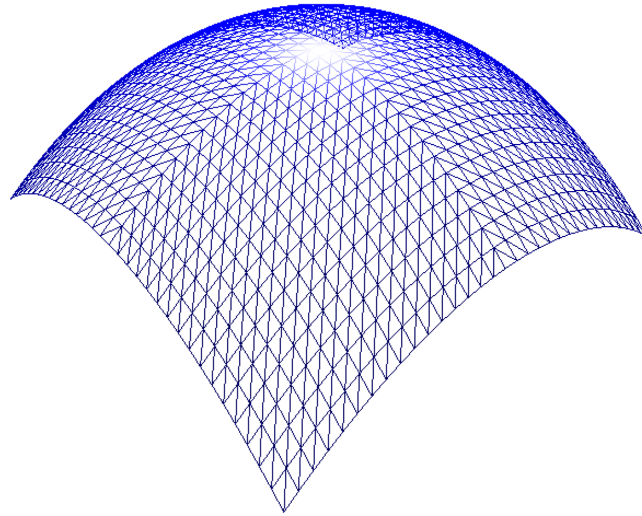

Abbildung 21: Bézierfläche im Wireframe-Modus (*Tessellation-Level* = 32)

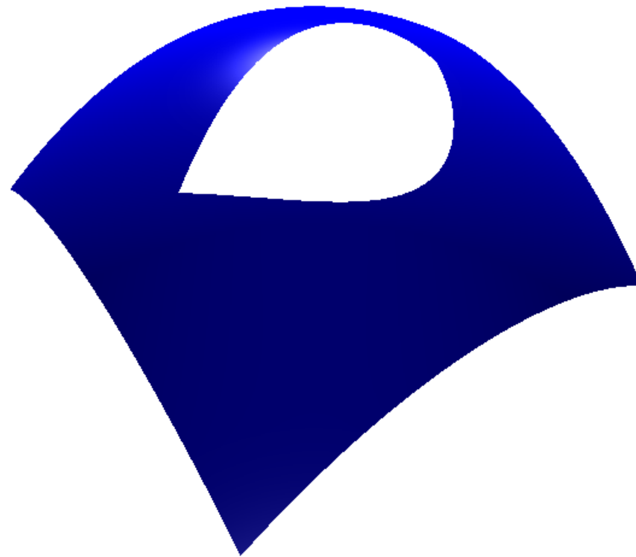

Abbildung 22: Bézierfläche mit Trimmung einer quadratischen Bézierfläche

Abbildung 23: Bézierfläche mit Trimmung einer quadratischen Bézierfläche in einer anderen quadratischen Bézierfläche

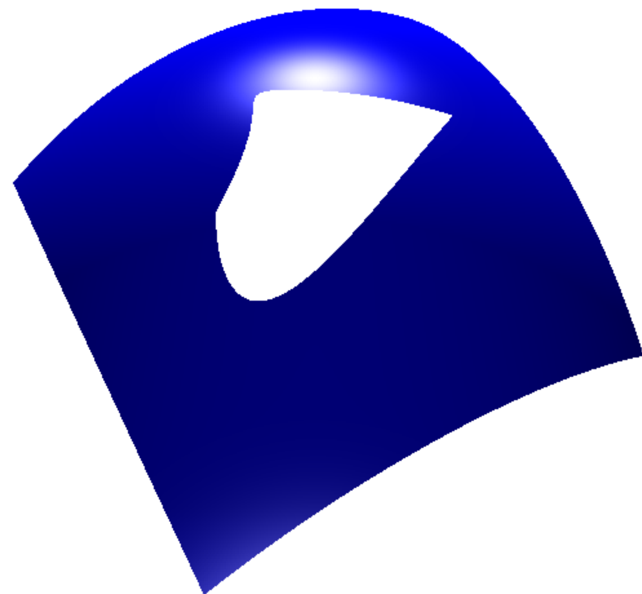

Abbildung 24: Bézierfläche mit Trimmung einer kubischen Bézierfläche

Abbildung 25: Bézierfläche mit Trimmung einer kubischen Bézierfläche in einer anderen kubischen Bézierfläche

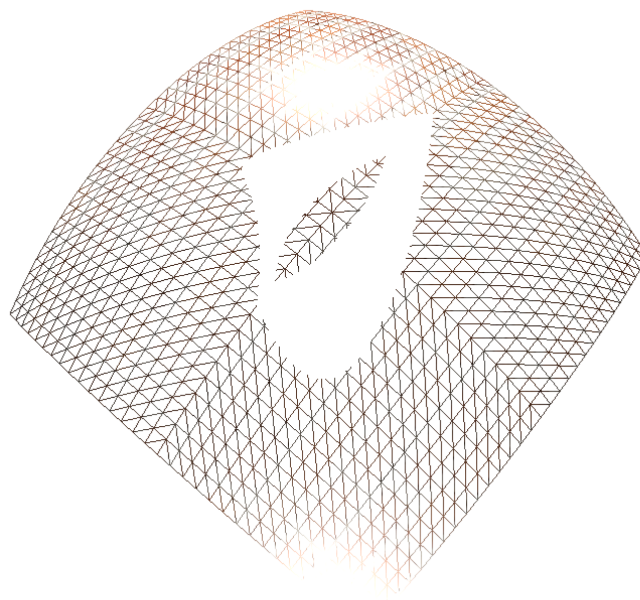

Abbildung 26: Bézierfläche mit Trimmung einer kubischen Bézierfläche in einer anderen kubischen Bézierfläche im Wireframe-Modus

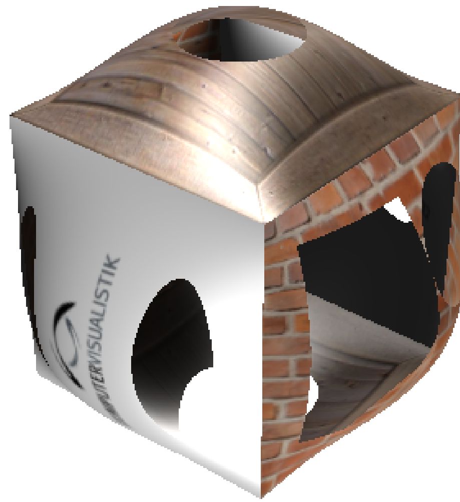

Abbildung 27: Rendern von mehreren getrimmten Bézierflächen

5.2 Grenzen

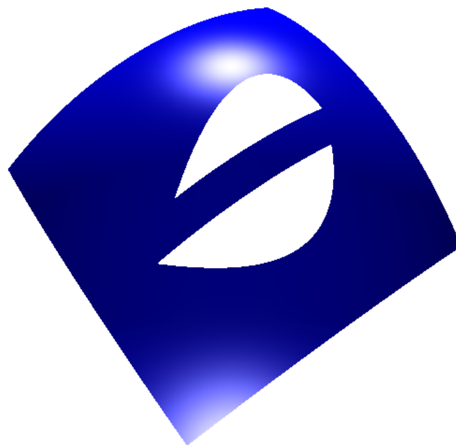

Abbildung 28: Fehler bei der Berechnung der Trimmfläche

Aufgrund der Implementierungsart, kommt es zu Fehlern der Trimmungsberechnungen im Bereich des ersten und letzten Kontrollpunktes. Diese Fehlberechnung erfolgt, weil der erste und letzte Kontrollpunkt nicht auf der selben *y-Achse* liegen. Diese beiden Punkte müssen auf einer horizontalen Ebene liegen damit der Algorithmus funktioniert.

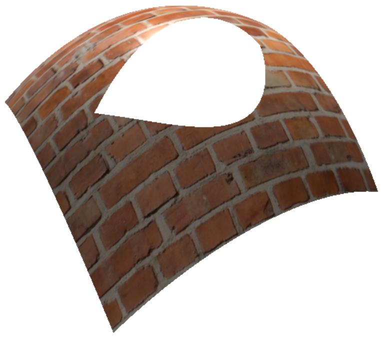

Abbildung 29: 7 Wiederholungen

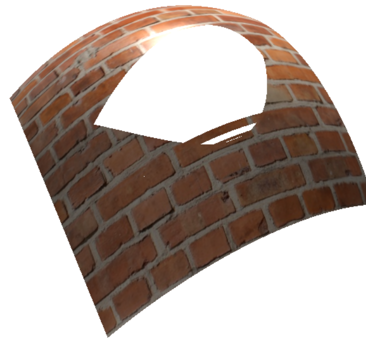

Abbildung 30: 6 Wiederholungen

Links in Abbildung 29 wurde das Newton-Verfahren mit sieben Wiederholungen ausgeführt und rechts in Abbildung 30 mit sechs Wiederholungen. Es ist zu sehen, dass wenn das Newton-Verfahren nicht mit ausreichend Wiederholungen ausgeführt wird, es zu Fehlern bei der Trimmung kommt. Da jede Freiformfläche anders ist, ist es ratsam einen Puffer einzuplanen und mindestens zehn Wiederholungen auszuführen.

Abbildung 31: $\epsilon = 0.1$

Abbildung 32: $\epsilon = 0.01$

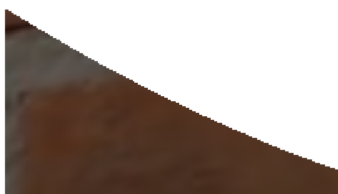

Abbildung 33: $\epsilon = 0.001$

Abbildung 34: $\epsilon = 0.0001$

In Abbildung 31 wurde ein *epsilon* von 0.1 im Newton-Verfahren benutzt. Dadurch kommt es zu starken Artefakten oder zu Fehlern wie in der

Abbildung 31 zu sehen ist. Wird das *epsilon* auf 0.01 erhöht, ist die Kante glatter aber es sind immer noch zu starke Abstufungen zu sehen (s. Abbildung 32). Erst ab einem *epsilon*-Wert von 0.001, wird eine ausreichend geglättete Kante berechnet, wie in Abbildung 33 zu sehen ist. Wird der *epsilon*-Wert auf 0.0001 verkleinert (s. Abbildung 34), wird die Kante zwar noch schärfer gerendert, aber der Mehraufwand im Vergleich zu einem *epsilon*-Wert von 0.001 lohnt sich nicht für eine minimale Verbesserung.

6 Fazit

In dieser Arbeit wurde eine Methode für das Rendern von Freiformflächen vorgestellt. Es handelt sich dabei um die Darstellung von Bézierflächen die sich das Prinzip der Tessellierung zu Nutzen macht, um eine effiziente Berechnung zu gewährleisten. Die hier implementierten Freiformflächen haben jeweils 16 (4×4) Kontrollpunkte. Des Weiteren wurden sie mit Texturen erweitert und mit dem Phong-Shader beleuchtet. Der zweite Teil der Ausarbeitung behandelte das Thema der Trimmung. Das Trimmen macht es möglich mehrere Formen gleichzeitig aus einer Freiformfläche herauszuschneiden. Dabei kann es sich um einfache Formen wie Kreise, Rechtecke, Quadrate oder Ellipsen handeln oder auch um Bézierkurven. Das Trimmen der Bézierkurven wurde nochmal erweitert, um es möglich zu machen, dass eine Bézierfläche innerhalb getrimmten Bézierkurven gerendert werden kann. Aufgrund der Implementierungsart ist dies nur möglich, wenn der jeweils erste und letzte Kontrollpunkt der beiden Bézierkurven auf einer horizontalen Ebene liegen. Diese Einschränkung könnte bei einer Weiterführung der Arbeit vermieden werden indem eine andere Implementierungsart verwendet wird.

Außerdem befasst sich die Arbeit mit einer Form der Freiformflächen, nämlich der Bézierfläche. Um komplexere Freiformflächen modellieren zu können, kann eine Fläche mit mehr Kontrollpunkten berechnet werden (hier ist zu beachten, dass der Maximalwert an Kontrollpunkten an die Variable *GL_MAX_PATCH_VERTICES* gebunden ist, die implementierungsabhängig ist [9]) oder es könnte auf das Prinzip der NURBS zurückgegriffen werden.

Eine weitere Verbesserung der Freiformfläche könnte ein variierendes *Level-of-Detail* ermöglichen. Eine mögliche Berechnung der Krümmung an der Oberfläche könnte genutzt werden um, an den Stellen mit hoher Krümmung, das *Level-of-Detail* zu erhöhen.

Literatur

[1] Codewithc. C program for newton raphson method. https://www.codewithc.com/c-program-for-newton-raphson-method/, 2014. [Online; Stand 19. Oktober 2018].

[2] Codewithc. Newton raphson method algorithm and flowchart. https://www.codewithc.com/newton-raphson-method-algorithm-flowchart/, 2014. [Online; Stand 10. Oktober 2018].

[3] Paul Dawkins. Section 4-13: Newtons method. http://tutorial.math.lamar.edu/Classes/CalcI/NewtonsMethod.aspx, 2003 - 2018. [Online; Stand 10. Oktober 2018].

[4] Richard S Wright Graham Sellers, Nicholas Haemel. Primitive processing in open gl. http://www.informit.com/articles/article.aspx?p=2120983, 2013. [Online; accessed 17-October-2018].

[5] Jituo Li, Guodong Lu, Dongliang Zhang, and Yoshiyuki Sakaguti. Searching a 3d region for surface trimming. *The International Journal of Advanced Manufacturing Technology*, 30(11):1093–1100, Oct 2006.

[6] Juan C. Meza. Newton's method. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(1):75–78, 2011.

[7] Les Piegl and Wayne Tiller. *The NURBS book*. Springer Science & Business Media, 2012.

[8] David F. Rogers. *An introduction to NURBS with historical perspective*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[9] OpenGL Wiki. Tessellation — opengl wiki,. http://www.khronos.org/opengl/wiki_opengl/index.php?title=Tessellation&oldid=14135, 2017. [Online; accessed 17-October-2018].

[10] Wikipedia. Bezierfläche — wikipedia, die freie enzyklopädie. https://de.wikipedia.org/w/index.php?title=Bezierfl%C3%A4che&oldid=181122200, 2018. [Online; Stand 3. Oktober 2018].

[11] Wikipedia. De-casteljau-algorithmus — wikipedia, die freie enzyklopädie. https://de.wikipedia.org/w/index.php?title=De-Casteljau-Algorithmus&oldid=180603288, 2018. [Online; Stand 8. November 2018].

[12] Wikipedia. Ellipse — wikipedia, die freie enzyklopädie. https://de.wikipedia.org/w/index.php?title=Ellipse&oldid=180961958, 2018. [Online; Stand 9. Oktober 2018].

[13] Wikipedia. Newton-verfahren — wikipedia, die freie enzyklopädie. https://de.wikipedia.org/w/index.php?title=Newton-Verfahren&oldid=175045370, note = "[Online; Stand 10. Oktober 2018]", 2018.

[14] Wikipedia contributors. Bézier surface — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=B%C3%A9zier_surface&oldid=763496242, 2017. [Online; accessed 9-October-2018].

[15] Wikipedia contributors. Bézier curve — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=B%C3%A9zier_curve&oldid=865389492, 2018. [Online; accessed 30-October-2018].

[16] Wikipedia contributors. Coons patch — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Coons_patch&oldid=849839282, 2018. [Online; accessed 30-October-2018].

[17] Wikipedia contributors. Freeform surface modelling — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Freeform_surface_modelling&oldid=862115044, 2018. [Online; accessed 30-October-2018].