Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

**Mika Nickel**

## Eine Shape Grammar für organische Formen

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Mika Nickel

# Eine Shape Grammar für organische Formen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Informatik Technischer Systeme
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Peer Stelldinger

Eingereicht am: 13.06.2024

**Mika Nickel**

**Thema der Arbeit**
Eine Shape Grammar für organische Formen

**Stichworte**
prozedurale Content Generierung, formale Grammatiken, Shape-Grammar, organische Formen, Bézierkurven, Bézierflächen

**Kurzzusammenfassung**
In dieser Arbeit wird eine Möglichkeit für die Erweiterung einer Shape Grammar um gekrümmte Formen untersucht. Dafür werden eine Darstellung gekrümmter Formen mithilfe von parametrisierten Kurven und Flächen, sowie die Grundlagen und Funktionen der Shape Grammar vorgestellt. Auf Basis dieser Konzepte wird eine konkrete Implementation einer Shape Grammar vorgestellt, in der die gekrümmten Formen und deren Operationen auf Basis von Bézierkurven und Bézierflächen umgesetzt sind.

**Mika Nickel**

**Title of the paper**
A Shape Grammar for organic Shapes

**Keywords**
procedural content generation, formal grammar, shape-grammar, curved shapes, Bezier curves, Bezier surfaces

**Abstract**
This paper examines a method of shape-grammar that is extended by non-rectilinear shapes. Therefore an internal representation of curved shapes based on parametric curves and parametric surfaces, as well as the general basics and base functions of the shape-grammar is presented. Based on these concepts a concrete shape-grammar implementation that is able to operate on non-rectilinear shapes composed of Bezier curves and Bezier surfaces is presented.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

Die prozedurale Content Generierung wird für die automatische Erstellung diverser Inhalte verwendet. Dabei findet sie gerade im Bereich der Videospiele hohe Anwendbarkeit. Durch den Einsatz von prozeduraler Content Generierung kann eine breite Fläche an Inhalten generiert werden. So können besipielsweise große digitale Welten oder auch Charakter Animationen erstellt werden, ohne das diese manuell von Hand erstellt werden müssen. Dies verkürzt die Entwicklungszeiten und verringert die Kosten, was gerade durch die stetige Erhöhung der Komplexität ein Gewinn ist.

Für die Umsetzung von prozeduraler Content Generierung gibt es dabei verschiedenste Methoden. Einer dieser Methodiken, die auch in dieser Arbeit Anwendung findet, ist die Generierung von Inhalten auf Basis von formalen Grammatiken in Form der sogenannten Shape-Grammar. Die Shape Grammar finden, vor allem mit der Einführung der CGA(computer-generated architecture) Shape durch Wonka et al.[21], sowie der weiteren Spezifikation durch Müller et al.[10], häufig Anwendung bei der Generierung von Architektur bzw. Gebäuden. Die Anwendungsgebiete sind jedoch vielfältig, so hat Orsborn et al.[12] eine Shape-Grammar zur Generierung von Fahrzeugen vorgestellt, die Merkmale von unterschiedlichen Fahrzeugkategorien vereint.

Diese Arbeit beschäftigt sich mit der Frage, ob die Umsetzung einer Shape Grammar auch mit gekrümmten Formen möglich ist. Das Ziel der Arbeit ist es, ein System zu entwickeln, dass neben geradlinigen geometrischen Formen zusätzlich mit gekrümmten Formen arbeiten kann. Dazu soll im Rahmen dieser Arbeit eine bestehende Shape Grammar Implementation erweitert werden, sodass diese neben den bereits implementierten geometrischen Formen auch gekrümmte Formen abdeckt. Dazu müssen neue Arten von Formen und die Logik für die Shape Operationen der neuen Formen implementiert werden. Hierbei soll die Shape-Grammar gekrümmte Formen im 2D-Raum und 3D-Raum unterstützen, die mindestens die Standard Operationen unterstützen.

Die vorliegende Arbeit besteht aus sieben Kapiteln. Im zweiten Kapitel, Grundlagen, werden die formalen und mathematischen Grundlagen erläutert, die im Verlauf der Arbeit Anwendung finden. Das dritte Kapitel, Stand der Technik, stellt weitere Arbeiten vor, die sich mit gekrümmten Formen innerhalb des Shape Grammar Kontextes beschäftigen. Zudem wird auf die Arbeiten von Wonka et al.[21] und Müller et al.[10] eingegangen, welche die Split-Grammar bzw. CGA Shape vorstellen. Im vierten Kapitel, Konzept, erfolgt eine theoretische Einführung eines Ansatzes zur Umsetzung der Problemstellung. Zudem erfolgt eine kurze Vorstellung der bereits umgesetzten Konzepte, sowie eine Abgrenzung dieser zu den neuen Ansätzen. Zuletzt erfolgt eine Zusammenfassung der funktionalen Anforderung an das System. Das fünfte Kapitel stellt eine konkrete Umsetzung der Konzepte aus dem vierten Kapitel vor. Hierfür werden die vorgenommenen Erweiterungen an der bereits implementierten Shape Grammar erläutert. Dazu gehört die konkrete Vorstellung der internen Umsetzung der neuen Shapes, die Beschreibung der wichtigsten Operationen zur Umsetzung der Shape Rules, sowie die Umsetzung der Erstellung der Meshes. Im sechsten Kapitel, Evaluation, erfolgt eine Bewertung des Systems auf Grundlage der, im vierten Kapitel gestellten, Anforderungen. Zudem wird auf Einschränkungen der Implementierung und andere Randfälle eingegangen. Es erfolgt außerdem eine Performance Messung für unterschiedlich präzise Meshes. Im siebten Kapitel, Fazit und Ausblick, wird ein Fazit zu den Ergebnissen der Arbeit gezogen und es wird ein Ausblick über mögliche zukünftige Erweiterungen der Arbeit gegeben.

# 2 Grundlagen

Im folgenden Kapitel sollen die mathematischen und formalen Grundlagen vorgestellt werden, die im Verlauf der Arbeit genutzt werden.

## 2.1 Prozedurale Content Generierung

Die Prozedurale Content Generierung ist die automatische Erstellung von Inhalten durch formale Algorithmen. Die Generierung erfolgt dabei ohne bzw. durch geringe menschliche Interaktion. Zum Beispiel durch das Anpassen der möglichen Ergebnisse anhand von Parametern. Prozedurale Content Generierung ist gerade im Bereich der Videospiele eine häufig angewendete Technik, um Spieleinhalte in Form von Leveln, Maps, Quests, Texturen und Charakteren zu generieren. Die Gründe für den Einsatz von PCG sind dabei vielfältig[20]. In den ersten Spielen die PCG genutzt haben, wie Akalabeth, Elite und Sentinel, ist die PCG hauptsächlich als eine Art der Datenkompression genutzt worden. Elite nutzt PCG beispielsweise, um ein Universum bestehend aus 8 Galaxien mit je 256 Sonnensystemen mit bis zu 12 Planeten pro Sonnensystem zu generieren. Ein weiterer Grund für die Nutzung von PCG ist die Erhöhung des Wiederspielwertes[16]. Beispiele dafür sind die beiden Spiele Rogue und Diablo, die PCG für die Generierung von Dungeonlayouts verwenden. Dabei ist anzumerken, dass die generierten Inhalte nicht zwingend zufällig sind und auch deterministisch sein können. Im Fall von Elite wurde dafür zum Beispiel ein fester Seed(4096) für die Generierung genutzt. Im Fall von Diablo und Rogue muss die Generierung bestimmten Regeln folgen, um gültige Ergebnisse zu erzielen. So könnte zum Beispiel ein Dungeon der keinen Ausgang besitzt nicht beendet werden und sollte somit nicht generiert werden. Heutzutage wird die PCG bei der Entwicklung neuer Spiele interessant, da durch den Anstieg der Komplexität auch der Entwicklungsufwand größer wird. Was zum Wachstum von Entwicklerteams führt und damit auch mit größeren Kosten verbunden ist. PCG könnte dabei eine Möglichkeit sein, dem Wachstum entgegenzuwirken und die Kosten für alle Beteiligten zu senken[20, 16, 4].

## 2.2 L-Systeme

L-Systeme ist die Kurzform für Lindenmayer Systeme. Die L-Systeme wurden ursprünglich vom Biologen Aristid Lindenmayer als formales Modell für die Darstellung von Pflanzen bzw. der Entwicklung von Pflanzen konzipiert[7]. Die L-Systeme finden jedoch auch vermehrt Anwendung im Bereich der Computergrafik. Dabei werden sie vorallem bei der Erstellung von Fraktalen und der Modellierung von Pflanzen eingesetzt. Die Funktionsweise von L-Systemen beruht auf einem Ersetzungssystem. Dies bedeutet, dass komplexe Objekte erstellt werden, indem die Teile eines einfachen initialen Objektes, genannt Axiom, sukzessiv durch komplexere Teile ersetzt werden, bis ein gewünschter Detailgrad erreicht ist. Wie und welche Teile des Objektes ersetzt werden sollen, wird dabei durch Ersetzungsregeln definiert. Ein typisches Beispiel für ein Ersetzungssystem ist die *Snowflake Curve*, zu sehen in Abbildung 2.1. Dabei ist

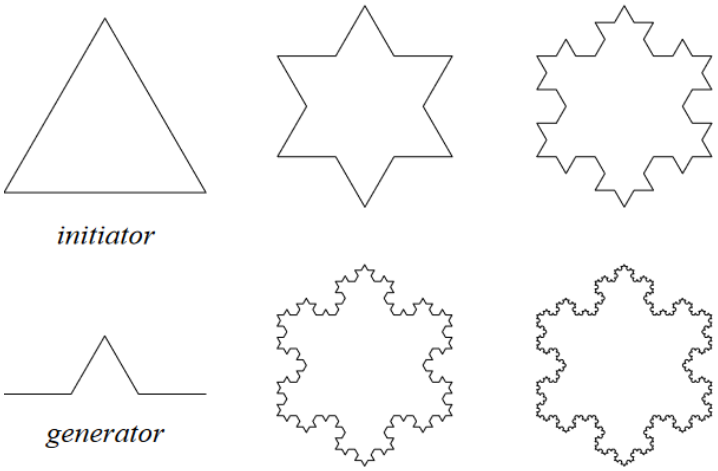



Abbildung 2.1: Konstruktion der Snowflake Curve(übernommen aus [13])

der initiator das initiale Objekt und der generator definiert wie die einzelnen geraden Linien einer Form ersetzt werden sollen[13]. L-Systeme weisen in ihrer Funktionsweise Ähnlichkeiten zu den Chomsky Grammatiken auf, dabei handelt es sich ebenfalls um Ersetzungssysteme. Der Unterschied zwischen den beiden Ersetzungssystemen liegt in der Anwendung der Ersetzungsregeln. Die Chomsky Grammatiken führen die Ersetzung sequenziell durch, während die L-Systeme alle Symbole eines Wortes gleichzeitig ableiten. Im folgenden soll zunächst ein Beispiel für ein L-System gegeben werden, bevor das System formal definiert wird. Gegeben sei ein Alphabet bestehend aus den Symbolen a und b. Für jeden Buchstaben wird eine Ableitungsregel definiert.$a \rightarrow ab$ und $b \rightarrow a$. Die Produktion startet beim Axiom $b$. Folgend sind vier Ableitungsschritte dargestellt.

```
b   -> a
a   -> ab
ab  -> aba
aba -> abaab
...
```

Die Ableitung kann fortgeführt werden, bis der gewünschte Grad erreicht ist. Formal kann ein L-System als ein geordnetes Tripel $G = (V, \omega, P)$ definiert werden. Wobei $V$ die Menge aller Symbole ist. $V*$ ist die Menge aller Wörter und $V+$ ist die Menge aller nicht leeren Wörter. $\omega$ ist das Axiom und $P \subset V \times V*$ ist eine endliche Menge an Ersetzungsregeln. Die Ersetzungsregeln sind ein geordnetes Paar $(a, \chi)$ und werden geschrieben: $a \to \chi$. Das Symbol $a$ ist dabei der predecessor und das Wort $\chi$ ist der successor[7, 13]. Ein L-System ist deterministisch, solange pro Buchstabe nur eine Ableitungsregel definiert ist.

## 2.3 Shape Grammar

Die Shape Grammar zuerst erwähnt von George Stiny und James Gips in [19], ist eine Grammatik, die zur Generierung von Formen genutzt werden kann. Sie weist dabei Ähnlichkeiten zu den Chomsky Grammatiken auf. Mit dem Unterschied, dass die Chomsky Grammatiken mit Symbolen arbeiten, während die Shape Grammar auf Shapes arbeiten. Die Sprache die eine Shape Grammar definiert, ist ebenfalls eine Menge an Shapes. Um diese Shapes zu erzeugen, werden Regeln definiert. Diese Regeln bestimmen wie eine Shape in eine neue Shape überführt werden soll und bestehen je aus einer linken Seite und einer rechten Seite. Die linke Seite gibt dabei die Shape an, auf die die Regel angewendet werden kann. Die rechte Seite gibt das Ergebnis der Regel an. Wenn eine Regel auf eine Shape angewendet wird, so wird die ursprüngliche Shape durch das Ergebnis der Regel ersetzt. Die Shapes werden erzeugt, indem beginnend bei einer initalen Shape, die Ableitungsregeln angewendet werden. Der Ableitungsprozess wird fortgeführt bis keine Regel angewendet werden kann. Formal kann die Shape Grammar als 4er-Tupel definiert werden: $SG = V_t, V_m, R, I$. $Vt$ ist dabei eine Menge an Terminalsymbolen. Dies sind Shapes die sich nicht weiter ableiten lassen. Vm ist eine Menge an nicht-Terminalsymbolen. Dies sind Shapes die weiter abgeleitet werden können. R ist eine Menge an Ableitungsregeln. Die Ableitungsregeln sind ein geordnetes Paar $(u, v)$, häufig auch als $u \to v$ geschrieben. Dabei ist $u$ ein Element aus $Vm$ und stellt die Shape dar, auf die die Regel angewendet werden kann und $v$ ist ein Element aus Vm oder Vt und stellt das Ergebnis der Regel dar.$I$ ist die initiale Shape und besteht aus einem Element von Vm. Die intiale Shape I ist im Normalfall so gewählt, dass mindestens eine Ableitungsregel anwendbar ist.[19, 18, 3]

## 2.4 Bézierkurven

Die Bézierkurve ist eine Art der Freiformkurven. Sie wird über eine Anzahl von Kontrollpunkten definiert und über den Parameter $t$ parametrisiert, sodass $P(t)$ einen Punkt auf der Kurve darstellt. Der Grad $n$ der Kurve ist abhängig von der Anzahl der Kontrollpunkte. Sei $N$ die Anzahl an Kontrollpunkten dann ist der Grad $n = N - 1$. Die Kontrollpunkte bestimmen den Verlauf der Kurve. Für $t \in [0, 1]$ startet die Kurve beim ersten Kontrollpunkt und endet im letzten Kontrollpunkt. Die inneren Kontrollpunkte liegen nicht auf der Kurve, ziehen die Kurve aber in deren Richtung. Je näher die Kurve einem Kontrollpunkt ist, desto stärker wirkt dieser auf den Verlauf der Kurve ein. Für $t \in [0, 1]$ liegt die Kurve somit innnerhalb der konvexen Hülle der Kontrollpunke[2, 1, 14].

Im folgenden wird auf zwei Arten der Konstruktion eingegangen. Eine rekursive Konstruktion mithilfe des De-Casteljau Algorithmus sowie eine explizite Konstuktruktion unter Verwendung der Bernstein Polynome.

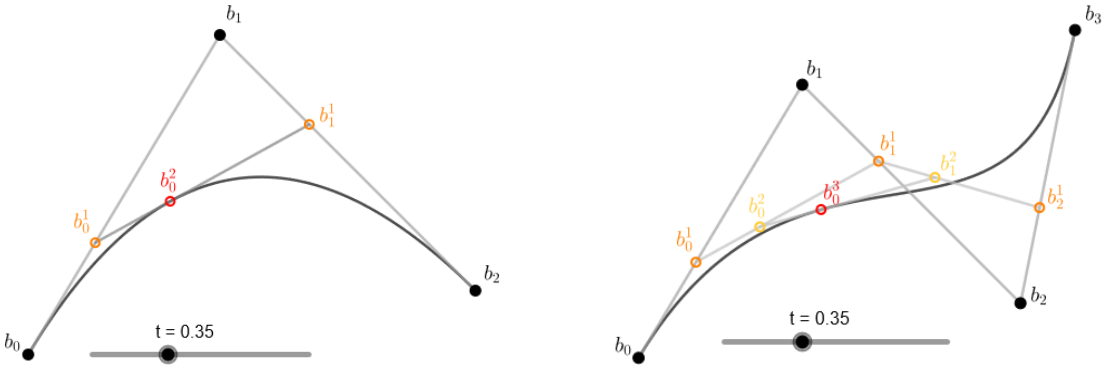
### 2.4.1 De-Casteljau Algorithmus

Der De-Casteljau Algorithmus ist ein wichtiger Algorithmus im Bereich der Konstruktion von Kurven und Oberflächen. Er basiert auf der wiederholten Anwendung von linearer Interpolation. Angewendet auf eine Kurve vom Grad $n = 2$ soll folgendes gelten. Seien $b_0, b_1, b_2 \in R3$ beliebige Kontrollpunkte und sei $t \in \mathbb{R}$. So ergeben sich durch lineare Interpolation der Punkte $b_0$ und $b_1$ sowie der Punkte $b_1$ und $b_2$ die neuen Punkte $b_0^1$ und $b_1^1$.
$b_0^1(t) = (1 - t)b_0 + tb_1$
$b_1^1(t) = (1 - t)b_1 + tb_2$
Wiederholte Anwendung der linearen Interpolation mit den Punkten $b_0^1$ und $b_1^1$ führt zum Punkt $b_0^2$.
$b_0^2(t) = (1 - t)b_0^1(t) + tb_1^1(t)$
Der Punkt $b_0^2$ bildet für jedes beliebige $t \in [0, 1]$ einen Punkt auf der Kurve innerhalb des Kontrollpolygons[2, 1, 15], zu sehen in Abbildung 2.2a. Das Verfahren kann für Kurven beliebigen Grades n erweitert werden, dabei gilt folgendes. Gegeben sei $b_0, b_1, b_2, ..., b_i \in R3$ und $t \in R$. Dann gilt:

$$b_e^r(t) = (1 - t)b_e^{r-1}(t) + tb_{i+1}^{r-1}(t) \begin{cases} r = 1, ..., i \\ e = 0, ..., i - r \end{cases}$$

Wobei dann die Punkte $b_i^0 = b_i$ die Kontrollpunkte darstellen und $b_0^i$ den Punkt auf der Kurve

für den Parameter $t$ darstellt. Ein Beispiel einer kubischen De-Casteljau Konstruktion ist in Abbildung 2.2b zu sehen.




(a) Quadratischer De-Casteljau Algorithmus(in Anlehnung an [2])

(b) Kubischer De-Casteljau Algorithmus(in Anlehnung an [2])

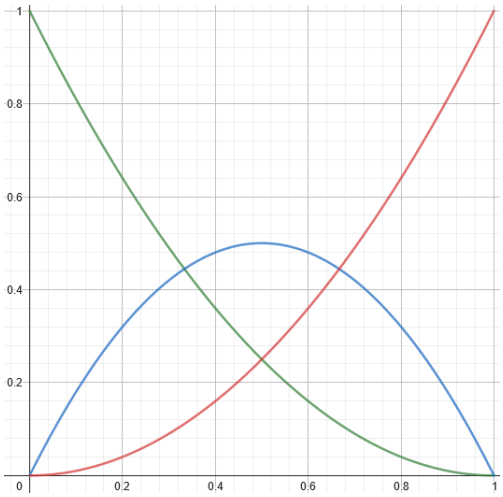
Abbildung 2.2: Beispiele De-Casteljau Algorithmus

#### Aufteilung einer Kurve

Neben der Konstruktion von Kurven findet der Algorithmus auch Anwendung bei der Aufteilung von Kurven. Dabei wird eine Kurve in zwei oder mehr seperate Kurven aufgeteilt, deren Verlauf zusammen die Ursprungskurve ergibt. Dabei dienen die interpolierten Punkte als Kontrollpunkt der neuen Kurven. Folgend die Konstruktion für eine Kurve vom Grad $n = 3$. Seien $c_0, c_1, c_2, c_3 \in R3$ beliebige Kontrollpunkte und sei $b_0^i$ ein Punkt auf der Kurve. Sei zudem die Interpolation wie in 2.4.1 definiert, dann definieren die Punkte $b_0, b_0^1, b_0^2, b_0^3$ neue Kontrollpunkte für eine Kurve vom Punkt $b_0$ bis zum Punkt $b_0^2$. Der Schnittpunkt wird dabei über $t$ parametrisiert[15].

### 2.4.2 Bernstein Polynome

Die Bernstein Polynome bilden bei der Konstruktion von Bézierkurven eine parametrisierte Basisfunktion, welche über den Parameter $t \in \mathbb{R}$ Gewichte für die einzelnen Kontrollpunkte bestimmt. Seien also $b_0, b_1, b_2, ..., b_i \in R3$ beliebige Kontrollpunkte und $n \in N$ der Grad der Kurve, dann kann über
$p(t) = \sum_{i=0}^{n} c_i \cdot B_i^n(t)$
ein Punkt $p(t)$ auf der Kurve bestimmt werden. Dabei gilt $B_i^n(t)$ ist die Basisfunktion(die Bernsteinpolynome) in der Form $B_i^n(t) = (\binom{n}{i})t^i(1-t)^n - i, t \in [0,1]$[2, 14, 15]. Eine wichtige Eigenschaft ist dabei die Interpolation der Endpunkte, da für $t = 0$ gilt $B_0^n(0) = 1, B_1^n(0) =$




(a) Quadratische Bernstein Polynome


(b) Kubische Bernstein Polynome

Abbildung 2.3: Visualisierung Bernstein Polynome

$0, B_2^n(0) = 0, ..., B_n^n(0) = 0$, äquivalent gilt für $t = 1$ $B_n^n(1) = 1, B_{n-1}^n(1) = 0, B_{n-2}^n(1) = 0, ..., B_0^n(1) = 0$. Dadurch wird sichergestellt das bei $t = 0$ und $t = 1$ jeweils nur der Start bzw. Endpunkt berücksichtigt wird, sodass die beiden Punkte $b_0$ und $b_i$exakt auf der Kurve liegen. Eine weitere Eigenschaft der Bernstein Polynome ist, dass die Summe der einzelnen Funktionen für ein gegebenes t immer $\sum_{i=0}^{n} B_i^n = 1$ ergibt, zu sehen in 2.3.

### 2.4.3 Äquivalenz De-Casteljau und Bernsteinpolynome

G. Farin hat in [2] gezeigt das gilt: $B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t)$ Dies zeigt das die Konstruktion von Bézierkurven mittels Bernsteinpolynomen und De-Casteljau Algorithmus vom Ergebnis äquivalent sind.

## 2.5 Splines

Die Idee hinter Splines besteht darin, mehrere Kurven von niedrigem Grad zu einer komplexeren Kurve zusammenzufügen. Die Übergänge zwischen Kurvensegmenten können in die 3 Stetigkeiten $G_0 Stetigkeit, G_1 Stetigkeit, G_2 Stetigkeit$ eingeordnet werden. $G_0$ ist erreicht, wenn sich die beiden Kurvensegmente berühren. $G_1$ Stetigkeit setzt $G_0$ Stetigkeit vorraus. Zudem müssen die Tangenten in die gleiche Richtung weisen. Bei Bézierkurven muss dafür das letzte Segment des Kontrollpolygons n in die gleiche Richtung zeigen, wie das erste Segment des Kontrollpolygons n+1. $G_2$ Stetigkeit setzt $G_1$ Stetigkeit vorraus. Bei Bézierkurven muss

dafür das letzte Segment des Kontrollpolygons n die gleiche Richtung und Länge wie das erste Segment des Kontrollpolygons n+1 haben.

## 2.6 Bézierflächen

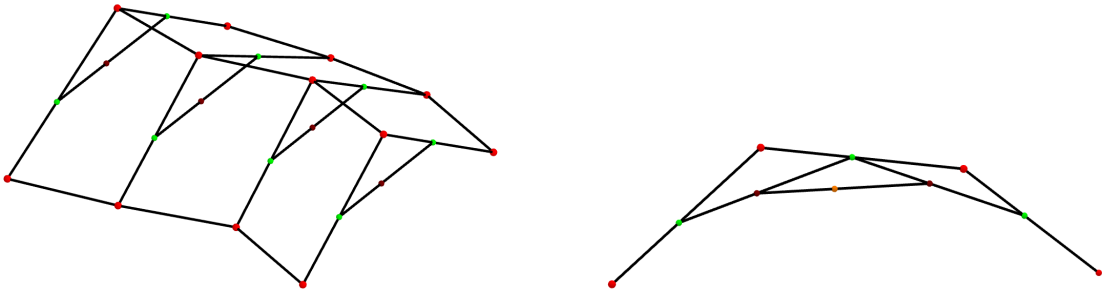
Die Bézierflächen sind eine Erweiterung der Bézierkurven in den 3D-Raum. Sie werden über ein Gitter von Kontrollpunkten $(m+1)x(n+1)$ definiert und über die zwei Parameter $u$ und $w$ parametrisiert. Ähnlich zu den Kurven beinflussen auch die Kontrollpunkte der Bézierfläche den Verlauf der Fläche. Dabei liegen die vier Eck-Kontrollpunkte genau auf der Fläche und die inneren Kontrollpunkte dienen als eine Art Ankerpunkt, zu denen die Fläche gezogen wird. Die beiden Konstruktionsarten der Bézierkurve, über die Bernstein Polynome und den De-Casteljau Algorithmus, können für die Konstruktion von Bezier Flächen erweitert werden und sollen im folgenden vorgestellt werden[14, 15].

### 2.6.1 Bernstein Polynome

Die Konstruktion von Bézierflächen mittels Bernstein Polynomen ist ähnlich zur Konstruktion von Kurven in 2.4.2. Die Bernsteinpolynome bilden weiterhin Gewichtsfunktionen für die Kontrollpunkte, mit dem Unterschied, dass bei der Flächenkonstruktion für die Parameter $u$ und $w$ je eine eigene Basisfunktion genutzt wird. Für ein Kontrollpunkt Gitter in der Form $(m+1)x(n+1)$ hat die Basisfunktion für den Parameter $u$ die Form $B_i^m(u) = \binom{m}{i}u^i(1-u)^m - i, u \in [0,1]$ und die Basisfunktion für den Parameter $w$ die Form $B_j^n(w) = \binom{n}{j}w^j(1-w)^n - j, w \in [0,1]$. Die vollständige Flächenberechnung hat damit die Form $p(u,w) = \sum_{i=0}^{m}\sum_{j=0}^{n} c_{i,j} \cdot B_i^m(u) \cdot B_j^n(w)$. Durch ersetzen des Parameters u oder w durch eine Konstante können Bézierkurven berechnet werden, die auf der definierten Fläche liegen[14, 15].

### 2.6.2 Scaffolding Construction

Die Scaffolding Construction ist eine Erweiterung des De-Casteljau Algorithmus für Bézierflächen. Diese Art der Konstruktion basiert auf der wiederholten Anwendung des De-Casteljau Algorithmus für Kurven 2.4.1. Wie in 2.4 zu sehen wird der De-Casteljau Algorithmus zunächst auf Kurven bestehend aus den Kontrollpunkten $(P_{0,0}, P_{0,1}, P_{0,2}, ..., P_{0,n})$, $(P_{1,0}, P_{1,1}, P_{1,2}, ..., P_{1,n})$,..., $(P_{m,0}, P_{m,1}, P_{m,2}, ..., P_{m,n})$. Die Ergebnisse der einzelnen Kurven, zu sehen in Abbildung 2.4a als dunkelrote Punkte, werden danach als neue Kurve interpretiert, auf die der Algorithmus ein weiteres Mal angewendet wird, zu sehen in Abbildung 2.4b. Das daraus resultierende Ergebnis ist der konstruierte Punkt auf der Fläche[15]. Die



(a) Anwendung des De-Casteljau Algorithmus auf die Reihen

(b) Anwendung des De-Casteljau Algorithmus auf die Ergebnisse aus a)

Abbildung 2.4: Visualisierung Scaffolding Construction(in Anlehnung an [15])

Scaffolding Construction kann wie der De-Casteljau Algorithmus auch dazu genutzt werden, eine gegebene Fläche in zwei seperate Flächen zu zerteilen. Dafür wird wie im ersten Schritt, entsprechend der Schnittrichtung, der De-Casteljau Algorithmus auf die Spalten bzw. die Reihen der Kontrollpunkt Matrix angewendet. Die linke Seite des De-Casteljau Dreiecks stellt dann die Kontrollpunkte für die linke Teilfläche dar, die rechte Seite die Kontrollpunkte der rechten Teilfläche.

## 2.7 Triangulation

### 2.7.1 Ear-Clipping Algorithmus

Der Ear-Clipping Algorithmus wird zur Triangulation von Polygonen genutzt. Er basiert auf der Annahme, dass jedes einfache Polygon, welches mindestens 4 Vertices umfasst, mindestens zwei sogenannte 'ears' besitzt. Ein 'ear' bezeichnet ein Dreieck, welches sich aus drei aufeinanderfolgenden Vertices des Polygons zusammensetzt. Dabei muss genau eine Seite des Dreiecks innerhalb des Polygons liegen, während die anderen beiden Seiten gleichzeitig auch Seiten des Polygons sind.[9] Sei ein Polygon definiert durch die Vertices $P = V_1, V_2, V_3, ..., V_n (n \leqq 4)$. Die Vertices $V_i, V_{i+1}, V_{i+2}$ formen ein ear wenn die Kante von $V_i$ und $V_{i+2}$ innerhalb des Polygons $P$ liegt. Ein 'ear' kann entfernt werden, indem der Vertex $V_{i+1}$ aus dem Polygon entfernt wird.[9, 8] Der Basis Algorithmus sieht vor, je ein geeignetes 'ear' für das Polygon zu finden und zu entfernen. Dieser Vorgang wird so lange fortgesetzt, bis von dem Ursprungspolygon noch ein Dreieck übrig bleibt. Dieses Dreieck und alle entfernten Dreiecke bilden zusammen eine Triangulation für das Ursprungspolygon. Der Basis Algorithmus hat eine Komplexität von $O(n^3)$.[8] Im folgenden soll der Ablauf des Algorithmus detaillierter beschrieben werden. Dafür wird der Algorithmus von J. Rourke herangezogen, dies ist eine modifizierte Version

des Grundalgorithmus die die Komplexität auf $O(n^2)$ reduziert. Input ist ein Polygon der Form$P = V_1, V_2, V_3, ..., V_n$ der Output ist eine Triangulation bestehend aus $n - 2$ Dreiecken. Der Ablauf lässt sich grob in drei Schritte aufteilen: In Schritt 1 werden die Innenwinkel aller Vertices berechnet, bei einem Winkel $< 180$ wird der Vertex als Convex und bei einem Winkel $> 180$ als Reflex markiert.In Schritt 2 werden alle Vertices ermittelt die ein 'ear-tip' sind, dies ist der Fall wenn der Vertex als Convex markiert ist und die Closure $C(V_{i-1}, v_i, v_{i+1})$ keinen Vertex beinhaltet, der Reflex ist. In Schritt 3 wird der Vertex $V_i$der den kleinsten Innenwinkel aufweist und ein 'ear-tip' ist gewählt und entfernt, dabei wird das Dreieck bestehend aus$V_{i-1}, V_i, V_{i+1}$ dem Output $T$ hinzugefügt und der Innenwinkel sowie der 'ear-tip' Status der Vertices $V_{i-1}$ und $V_{i+1}$ geupdated. Der dritte- Schritt wird wiederholt bis $n - 2$ Dreiecke dem Output $T$ hinzugefügt wurden.[11]

# 3 Stand der Technik

## 3.1 Split Grammar

Die Shape Grammar die von G. Stiny und J. Gips 1971[19] vorgestellt wurde, bietet formal ein mächtiges Werkzeug zur Darstellung vieler verschiedener Designs. Das Problem ist die konkrete Implementierung des Ansatzes. Da die Ableitungregeln auf Sub-Shapes definiert sind, müsste eine konkrete Implementierung ein aufwendiges Matching Verfahren beinhalten, welches in der Lage ist, den Marker bei unterschiedlicher Skalierung und Rotation zu bestimmen. Im Jahr 1982 hat G. Stiny die Set Grammar vorgestellt [17]. Diese definieren die Shapes als gelabelte Objekte, was eine Implementation vereinfacht.

Dieser Ansatz wird auch in der Split Grammar genutzt, die 2003 von Wonka et al.[21] vorgestellt wurde. Bei der Split Grammar handelt es sich um eine Grammatik, die für die Erstellung von Gebäuden entwickelt wurde. Der Hauptaspekt der Split Grammar ist die Darstellung der Shapes auf Basis von umschliessenden Bounding-Shapes. Diese Bounding-Shapes sind über einen Punkt $P$, drei ortogonale Vektoren $X, Y, Z$ und eine Größe $S$ definiert. Die Abbildung 3.1 zeigt ein Beispiel einer solchen Bounding-Box, auch *Scope* gennant[10]. Die Ableitungsregeln werden zudem, anders als bei bisherigen Shape Grammars, in zwei Typen aufgeteilt. Die beiden möglichen Typen sind dabei die *conversion rule* und die *split rule*. Eine *conversion Rule* überführt eine Shape in eine neue Shape. Eine *split rule* ermöglicht das Unterteilen einer Shape in mehrere neue Shapes[21, 10]. Der Aufbau der Regeln sieht wie folgt aus:

*id: predecessor : cond* $\rightarrow$ *successor : prob*

- id: Ist die ID der Regel
- predecessor: Ist das Symbol der Shape die ersetzt werden soll
- cond: Ist eine Bedingung die wahr sein muss, damit die Regel angewendet werden kann
- succesor: Ist das Symbol der abgeleiteten Shape

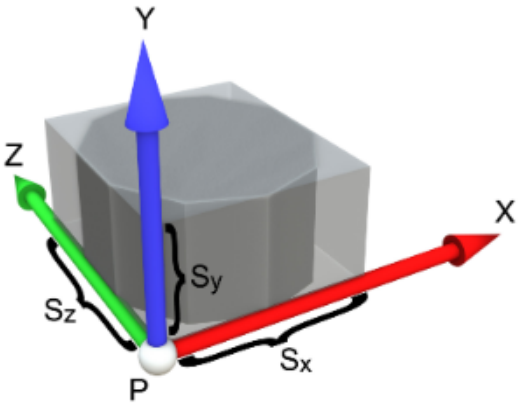



Abbildung 3.1: Scope einer Shape(übernommen aus [10])

- prob: Ist die Wahrscheinlichkeit mit der die Regel gewählt wird

Die bekanntesten split rules sind: *basic split*(oder auch *subdivide*) und *repeat*. Die *basic split* Regel unterteilt eine Shape entlang einer Achse in $n$ Teilshapes, die Größe der einzelnen Teilshapes und die Achse des Splits muss dabei angegeben werden[10]. Ein Beispiel einer solche Regel(übernommen aus [10]) könnte wie folgt aussehen:

*1: fac → Subdiv("Y",3.5,0.3,3,3,3){ floor | ledge | floor | floor | floor }*

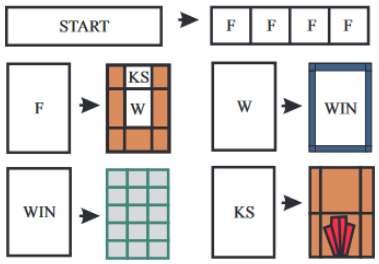
Die *repeat* Regel unterteilt eine Shape entlang einer Achse in soviele gleichgroße Teilshapes wie möglich, die Größe der Teilshapes und die Achse muss dabei angegeben werden. Ein Beispiel einer solchen Regel(übernommen aus [10]) könnte wie folgt aussehen[10]:

*1: floor → Repeat("X",2){ B }*

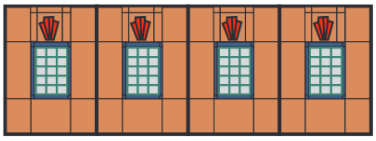
Abbildung 3.2a zeigt beispielhaft die Ableitungsregeln einer einfachen Split Grammar. Die weißen Shapes sind dabei non-terminal Shapes und die farbigen Shapes sind terminal Shapes. Das Ergebnis der Ableitung der Regeln aus 3.2a ist in Abbildung 3.2b dargestellt.

## 3.2 Ansätze zur Generierung gekrümmter Formen

In dieser Section sollen die Grundlegenden Ansätze weiterer Arbeiten aufgezeigt werden, wie gekrümmte Formen mittels Shape Grammars umgesetzt werden können.




(a) Beispiel für Regelen einer Split-Grammar (übernommen aus [21])



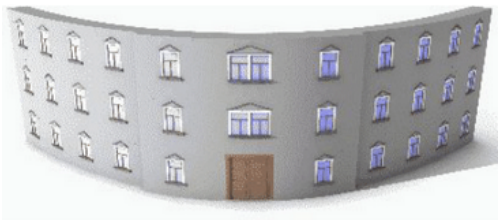
(b) Ergebniss der Ableitung des Beispiels (übernommen aus [21])

Abbildung 3.2: Beispiel Split Grammar

### 3.2.1 Erstellung von gekrümmten Designs durch Deformation

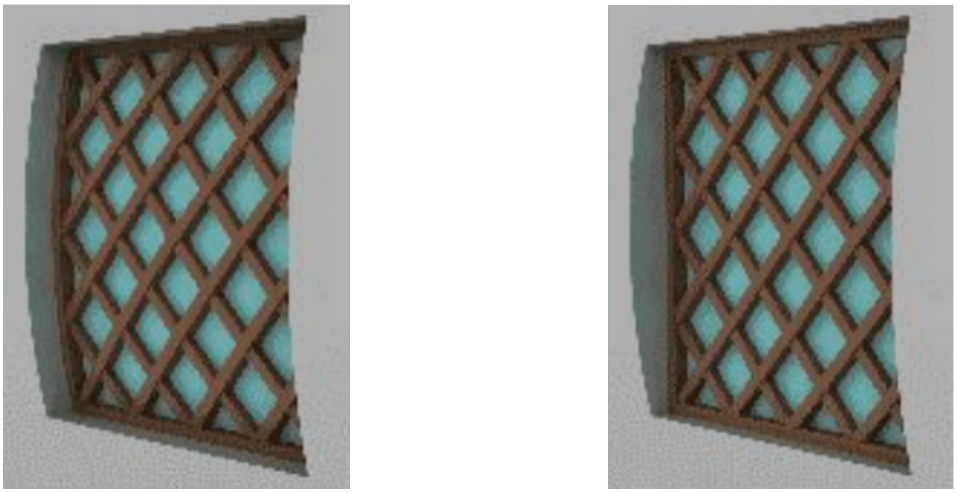
Zur Erstellung von gekrümmten Designs mittels Shape Grammar stellt die Arbeit von René Zmugg et al. eine erweiterte Form der split Grammar vor, die mittels Deformation der Shapes die Erzeugung von gekrümmten Modellen ermöglicht. Die Deformation soll dabei nicht als allgemeiner Post-Processing Schritt angewendet werden, sondern soll in den Ableitungsprozess der Grammatik integriert werden. Dadurch sollen mögliche auftretende Probleme behoben werden. Ansonsten könnte es bei längenabhängigen Operation wie z.B. dem repeated split zu unvorhergesehenen Ergebnissen führen, da die gekrümmten Flächen möglicherweise eine höhere Länge aufweisen. Als Beispiel, könnte eine gekrümmte Fassade möglicherweise mehr Fenster aufnehmen, als das nicht deformierte Gegenstück. Ein Beispiel ist in Abbildung 3.3 zu sehen. Zudem würde die Deformation als Post-Processing Step in der Hierarchie auf alle Formen angewendet werden[22]. Ein Beispiel dafür ist in Abbildung 3.4 dargestellt, in der die Fenster mitsamt der Fassade deformiert werden.

(a) Nicht-deformierte Fassade (übernommen aus [22])



(b) Deformierte Fassade unter Anwendung von deformation-aware operations (übernommen aus [22])

Abbildung 3.3: Beispiel deformation-awareness



(a) Deformiertes Fenster (übernommen aus [22])

(b) Korrektur der Deformation des Fensters (übernommen aus [22])

Abbildung 3.4: Auswirkung der Deformation als Post-Processing Schritt

Für die Umsetzung werden dabei je zwei verschiedene Arten des *deformed* splits definiert. Eine Standardmäßige Operation die den split auf die nicht-deformierte Shape anwendet und danach das Ergebnis deformiert(dies ist gleichzusetzen mit der Deformation als Post-Processing Step). Und eine spezielle *deformation aware* Operation.[22]

### 3.2.2 Shape Grammar definiert über Kurven

Die Arbeit von Jowers und Earl[5] beschäftigt sich damit, die Detektion der Subshapes zur Anwendung der Shape Rules auf nicht geradlinige Formen zu erweitern. Dazu haben Jowers und Earl einen *intrinsic matching* Algorithmus entwickelt, der mit parametrisierten Kurven arbeitet. Mithilfe dieses Algorithmus kann bestimmt werden, ob zwei Kurven über beliebige euklidische Transformationen aufeinander abgebildet werden können[5]. Dies ermöglicht den Vergleich von gekrümmten Formen zur Bestimmung der Ableitungsregeln innerhalb, eines Shape Grammar Interpreters. Weiterführennd haben Jowers und Earl eine Implementation einer solchen Shape Grammar auf Basis von Quadratischen Bézierkurven vorgestellt[6].

# 4 Konzept

Das folgende Kapitel soll einen Ansatz zur Umsetzung einer Shape Grammar für gekrümmte bzw. organische Formen vorstellen. Dabei wird zunächst auf die allgemeinen Konzepte der Shape Grammar eingegangen. Danach werden Ansätze vorgestellt, die zeigen sollen, wie Shape Operationen auf den organischen Formen umgesetzt werden können.

## 4.1 Allgemeine Konzepte einer Grammatik

In dieser Section soll auf den allgemeinen Aufbau, sowie auf die Bestandteile einer Shape-Grammar eingegangen werden. Zudem sollen mögliche Sonderfälle bzw. besondere Eigenschaften erläutert werden. Die Konzepte, auf die in dieser Section eingegangen wird, waren bereits umgesetzt und sollen hier hauptsächlich der Vollständigkeit und dem besseren Verständnis dienen.

### 4.1.1 Aufbau einer Grammatik

Eine konkrete Grammatik wird innerhalb einer .grammar Datei definiert und besteht aus zwei Teilen. Den Variablen und den Shape Rules.

Der Variablen-Abschnitt wird durch den String "*Variables:*" eingeleitet, danach kann eine beliebige Anzahl an Variablen definiert werden. Diese bestehen aus einem Bezeichner und einem Wert in der Form:

**<variableIdentifier> = <value>**

Die Variablen können dann im Shape Rule Abschnitt genutzt werden z.B. als Parameter einer Shape Operation oder für Bedingungen(siehe 4.1.3). Durch die Nutzung von Variablen wird es möglich, schnelle Anpassungen an den möglichen Ergebnissen der Grammatik vorzunehmen, ohne das Vorkenntnisse über die konkrete Grammatik benötigt werden. So kann beispielsweise eine Grammatik, die Häuser generiert, mit einer Variable *maxNumberOfFloors* erstellt werden,

die im Shape Rules Teil genutzt wird, um die maximale Anzahl der Stockwerke zu begrenzen.

Der zweite Abschnitt einer Grammatik beinhaltet die Shape Rules. Das sind Regeln die bestimmen, welche Shapes in neue Shapes überführt werden können und wie diese Shapes überführt werden. Eine Shape Rule besteht dabei mindestens aus einer *predecessorID*, also der ID der Shape, auf die die Regel angewendet werden soll. Einer oder mehreren *Shape-Operationen* zur Bestimmung, wie die Shape überführt werden soll. Und einer oder mehreren *successorIDs*, dies sind die IDs die den neu erstellten Shapes zugeordnet werden. Die Syntax einer Shape Rule ist dabei wie folgt:

**<predecessorID> –> <shapeOperation>(<parameter>) {successorID1, ..., successorIDn}**

Im folgenden ist ein Beispiel einer einfachen Grammatik zu sehen, welche aus einem Polygon in Form eines Quadrates einen Würfel erstellt. Als Voraussetzung muss ein Axiom mit der ID *Square* erstellt worden sein . Dieses Polygon hat in diesem Beispiel eine Kantenlänge von zwei. Über die Variable *height* wird die Höhe der Ergebnis Shape bestimmt, in diesem Fall ebenfalls zwei. Über die Shape Rule wird die Shape mit der ID *Square* bis zu der Höhe *height* extrudiert. Das Ergebnis der Shape Rule ist eine neue Shape in Form eines Würfels mit der ID *Cube*.

```
1 Variables:
2 height = 2.0;
3
4 Rules:
5 Square --> extrude(height) {Cube}
6 #Cube --> ...
```

### 4.1.2 Wahrscheinlichkeiten

Um Varianz bei den Ergebnissen einer Grammatik zu erzeugen, können in den Shape Rules Wahrscheinlichkeiten angegeben werden. Diese ermöglichen den Bestand mehrerer Shape Rules mit der gleichen predecessorID. Bei Angabe von Wahrscheinlichkeiten ist zu beachten, dass die Summe der Wahrscheinlichkeiten für die Shape Rules einer predecessorID genau 1 betragen muss. Im Ableitungsprozess wird dann unter Berücksichtigung der spezifizierten Wahrscheinlichkeit eine der passenden Regeln ausgewählt. Die Syntax ist wie folgt:

**predecessorID –> <shapeOperation>(<parameter>) {successorID} : <possibility>**

Als Beispiel:

**Square –> extrude(2) Cube : 0.6**

**Square –> extrude(6) Prism : 0.4**

### 4.1.3 Bedingungen und dynamische Veränderung von Variablen

Die Shape Rules können auch um Bedingungen erweitert werden, dabei handelt es sich um boolesche Ausdrücke die zu *true* ausgewertet werden müssen, damit die Shape Rule angewendet werden darf. Die Syntax ist wie folgt:

**<predecessorID> : <boolean expression> –> <shapeOperation>(<parameter>) {<successorID>}**

Zusätzlich können die Variablen auch in den Shape Rules verändert werden. Im folgenden ist ein Beispiel zu sehen, dass eine Grammatik zeigt, die mit der Variablen *segments2Add* die Anzahl vorgibt, wie häufig ein neues Segment in Y-Richtung ergänzt werden soll. Die Variable wird dazu bei jeder Ausführung der Shape-Rule, die eine extrude-Operation beinhaltet, dekrementiert.

```
1 Variables:
2 segmentHeight = 0.50;
3 segments2Add = 2;
4
5 Rules:
6 Square --> extrude(segmentHeight) Prism segments2Add--
7 Prism --> component_split("side_faces"){S} component_split("top"){T}
8
9 T : segments2Add > 0 --> extrude(segmentHeight) Prism segments2Add--
```

### 4.1.4 Anwendung mehrerer Shape Operationen

Die Shape Operationen und die Shape Rules stehen nicht in einer 1:1 Beziehung zueinander. Das heißt, es können auch mehrere Shape Operationen in einer Shape Rule verwendet werden. Die

Shape Operationen werden in diesem Fall alle auf die Shape mit der angegebenen predecessorID angewendet und nicht auf das Ergebnis der vorherigen Operation der Shape Rule.

## 4.2 Erzeugung der organischen Formen

Die organischen Shapes sollen nicht als Axiom fungieren, sondern sollen in den normalen Ableitungsprozess integriert werden. Im 2D sollen die organischen Shapes dabei aus den Polygon Shapes erzeugt werden. Dafür muss Funktionalität für eine Shape Operation ergänzt werden, die eine Ableitung von geometrischen Shapes in organische Shapes ermöglicht. Die organischen Shapes im 3D-Raum sollen aus den organischen Formen im 2D-Raum erzeugt werden können. Dazu kann die Shape Operation extrude genutzt werden. Um die Variabilität der Shapes zu steigern, sollen die zweidimensionalen organischen Formen auch eine erweiterte Extrude-Operation unterstützen, die ein schräges extrude ermöglicht.

## 4.3 Darstellung der organischen Formen

Die organischen Shapes können nur schwer direkt über Vertices definiert werden, da im allgemeinen eine große Anzahl benötigt wird, um Krümmungen zu repräsentieren. Stattdessen soll eine mathematische Darstellung angewendet werden, über die ein möglichst großes Spektrum an Shapes definiert werden kann und die es ermöglicht, Shape Operationen zu definieren.

Im 2D-Raum sollen die organischen Formen durch Bézierkurven dargestellt werden. Die Kurven bilden dabei die Segmente der Shape und können äquivalent zu den Strecken einer Polygon Shape betrachtet werden. Eine konkrete zweidimensionale organische Form besteht somit aus mehreren Kurven die zusammen einen geschlossenen Spline bilden. Ein Beispiel ist in Abbildung 4.1 zu sehen. Diese Abbildung zeigt einen geschlossenen Spline als mögliche Repräsentation einer organischen Shape. Die einzelnen Kurvensegmente des Splines sind zur besseren Übersicht abwechselnd in unterschiedlichen Farben dargestellt.

Die dreidimensionalen organischen Formen werden mittels einer extrude Operation aus den zweidimensionalen Formen erzeugt. Dadurch entsteht basierend auf der Fläche der 2D-Shape eine Zylinderähnliche Shape wie in Abbildung 4.2 zu sehen.

Diese Shapes sollen über Bézierflächen dargestellt werden. Die einzelnen Bézierflächen definieren dabei Segmente der Mantelfläche. Die Grund- und Deckfläche der Shape wird über

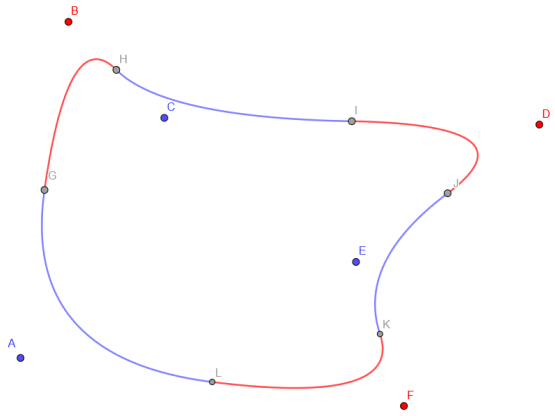



Abbildung 4.1: Beispiel eines Splines einer zweidimensionalen organischen Shape

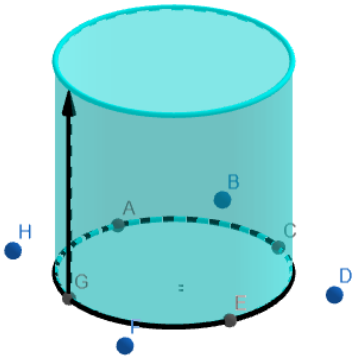



Abbildung 4.2: Darstellung einer einfachen dreidimensionalen Shape

die Bézierflächen intern mitdefiniert, da die Kontrollpunkte der Bézierflächen, für ein festes $n$ oder $m$ auch Kurven darstellen. Die Grundfläche kann somit über einen Spline definiert werden, die Kurven des Spline ergeben sich aus den Kontrollpunkten $(0, 0), (0, 1), ..., (0, m)$ der Bézierflächen. Diese Kurven sollen zur Vereinfachung im folgenden als Basiskurven einer Fläche bezeichnet werden. Die Deckfläche kann analog dazu über die Kontrollpunkte $(n, 0), (n, 1), ..., (n, m)$ der Bézierflächen bestimmt werden.

Im folgenden sollen zur besseren Lesbarkeit die zweidimensionalen organischen Shapes *ClosedSplineShapes* und die dreidimensionalen organischen Shapes *3DSplines* genannt werden.

## 4.4 Konzepte der Shape Operationen

Im folgenden soll auf die einzelnen Shape Operationen eingegangen werden. Dabei soll zunächst auf die Funktion und die Syntax eingegangen werden. Und danach eine mögliche Umsetzung für die organischen Shapes gegeben werden.

### 4.4.1 Extrude

Die Extrude-Operation kann auf Polygon-, ClosedSpline- und Segment3DShapes angewendet werden. Sie überführt die Shape in eine neue Shape der nächsthöheren Dimension, indem die Flächen „herausgezogen“ werden. So ensteht beispielsweise aus einer *PolygonShape* eine *PrismShape* oder aus einer *ClosedSplineShape* eine *Spline3DShape*. Die Operation hat dabei die folgende Syntax:

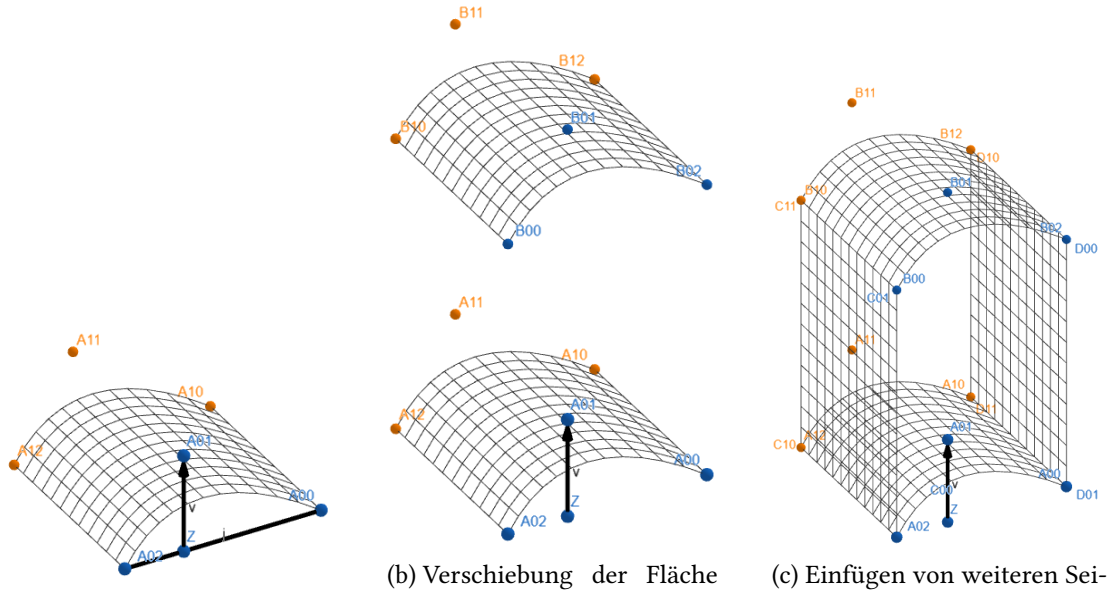
**<predecessorID> –> extrude(<height>) {<successorID>}**

Der Parameter *height* ist eine Gleitkommazahl und bestimmt die Höhe der Ergebnis Shape.

Angewendet auf ClosedSpline Shapes wird diese Operation, indem aus den Kurven der ClosedSplineShape Bézierflächen erzeugt werden. Dafür werden, für jede Kurve des Splines, die Kontrollpunkte der Kurve kopiert und als Kontrollpunkte der Basiskurve der Bézierfläche gesetzt. Sie definieren somit die Punkte $(0,0), (0,1), ..., (0,m)$ der Fläche. Die gleichen Kontrollpunkte werden dann um *height* in Y-Richtung verschoben und bilden die Punkte $(1,0), (1,1), ..., (1,m)$ der Fläche. Die so definierten Flächen bilden zusammen die Mantelfläche der neuen Shape.

Bei der Umsetzung für Segment3D Shapes muss im ersten Schritt die Richtung des extrudes bestimmt werden., da durch die gekrümmte Form der Shape keine Achse eindeutig gewählt werden kann. Stattdessen wird eine Strecke zwischen den äußeren Kontrollpunkten der Basiskurve der Bézierfläche erzeugt, zu der ein ortogonal verlaufender Einheitsvektor bestimmt wird, der in Richtung des inneren Kontrollpunktes zeigt. Entlang dieses Vektors werden, entsprechend der Belegung des Parameters *height*, die Kontrollpunkte der Segment3D Shape verschoben. Die beiden Flächen werden dann, an den äußeren Kontrollpunkten, mit Flächen vom Grad $n = 1$ und $m = 1$ verbunden, sodass die Mantelfläche geschlossen wird. Diese Mantelfläche erfüllt die Definition einer gedrehten Spline3D Shape und kann somit als Ergebnis der Operation genutzt

werden. In Abbildung 4.3 sind die Schritte der Extrude-Operation anhand einer einfachen




(a) Bestimmmung des Verschiebungsvektors

(b) Verschiebung der Fläche entlang des Verschiebungsvektors

(c) Einfügen von weiteren Seitenflächen zum schließen der Shape

Abbildung 4.3: Darstellung der Extrude Operation auf einer Segment3D Shape

Segment3D Shape dargestellt. Abbildung 4.3a zeigt den Verschiebungsvektor $\vec{v}$ der ortogonal zur Strecke j verläuft. In diesem Fall liegt der Vektor genau auf dem Punkt $A_{01}$. Dies ist zufällig bedingt, da der Abstand zwischen der Strecke j und dem Punkt $A_{01}$ genau eins beträgt. Ist dies nicht der Fall muss der Vektor normalisiert werden. Abbildung 4.3b zeigt die neue Oberseite der Shape, die durch Verschiebung entlang des Vektors $\vec{v}$ entsteht. Abbildung 4.3c zeigt die beiden Flächen, die die äußeren Kontrollpunkte der Ober- und Unterseite verbinden.

### 4.4.2 Split

Die Split-Operation unterteilt eine Shape in mindestens zwei Subshapes, wobei alle Subshapes zusammen wieder die ursprüngliche Shape bilden. Die Operation kann auf Shapes vom Typ Polygon, Prism, ClosedSpline, Spline3D und Segment3D angwendet werden und hat die folgende Syntax:

**<predecessorID> –> split(<axis>, <splitSize1>, <splitSize2>, ..., <splitSizeN>) {<successorID1>, <successorID1>, ..., <successorIDN>}**

Der Parameter *axis* ist ein String und bestimmt die Achse des Splits. Die Parameter *splitSize* sind Gleitkommazahlen und bestimmen die Größe der Subshapes entlang der Achse des Splits. Die Split Sizes können absolut oder relativ zur Größe der Shape angegeben werden. Für absolute spltSizes wird eine normale Gleitkommazahl angegeben. Für relative splitSizes wird an das Argument ein *r* angehängt. Die Summe der splitSizes muss in diesem Fall bei 1 liegen. Ein Beispiel könnte sein:
*Lot –> split("X", 0.4r, 0.6r) {LeftSide, RightSide}.*

Für die Umsetzung soll die Unterteilung der Shapes entsprechend der splitSizes über Ebenen erfolgen. Für ClosedSplineShapes soll ein Divide and Conquer Ansatz genutzt werden, sodass die Berechnungen lokal auf den einzelnen Kurven durchgeführt werden. Dafür werden für jede Kurve die Schnittpunkte der Schnittebene mit der Kurve berechnet. Weist dabei eine Kurve keinen Schnittpunkt auf, ist sie nicht betroffen und kann unverändert übernommen werden. Weist eine Kurve einen Schnittpunkt auf, wird sie mittels des De-Casteljau Algorithmus in zwei Teilkurven aufgeteilt. Nachdem die Schnittpunkte auf allen Kurven berechnet worden sind, müssen die Kurven, entsprechend der Schnittebenen, zu neuen Splines zusammengefügt werden. Dabei ist es an den Schnittstellen nötig, entlang der Schnittebene Kurven vom Grad $n = 1$ einzufügen um die Splines zu schließen.
In Abbildung 4.4 ist eine eine Split-Operation, anhand einer einfachen ClosedSpline Shape,

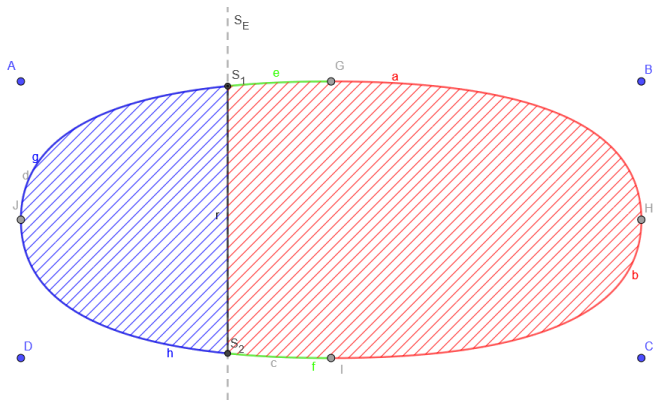



Abbildung 4.4: Split-Operation im 2D

dargestellt. Die in rot dargestellten Kurven zeigen die nicht vom Split betroffenen Kurven. Die grünen und blauen Kurven sind Teilkurven der Kurven $c$ und $d$, die am Schnittpunkt mit der Geraden aufgeteilt wurden. Die Strecke $r$ symbolisiert die Kurve vom Grad $n = 1$ die verwendet wird um die Splines zu schließen. Die Kurven werden auf zwei Splines aufgeteilt, welche die beiden Subshapes definieren. Die linke Subshape beinhaltet dabei die Kurven $g$ und $h$

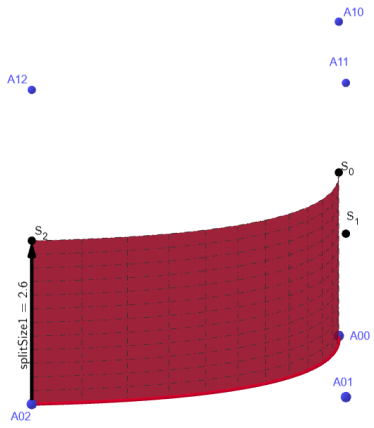
sowie die Strecke $r$. Die rechte Subshape beinhaltet die Kurven $a, b, e$ und $f$ sowie die Strecke $r$.

Die Umsetzung für Spline3D Shapes muss zusätzlich den Split in Y-Richtung abdecken. Die funktionsweise eines splits in Y-Richtung unterscheidet sich dabei von einem Split in X- oder Z-Richtung, weshalb die Umsetzung in zwei seperate Teile aufgeteilt wird.

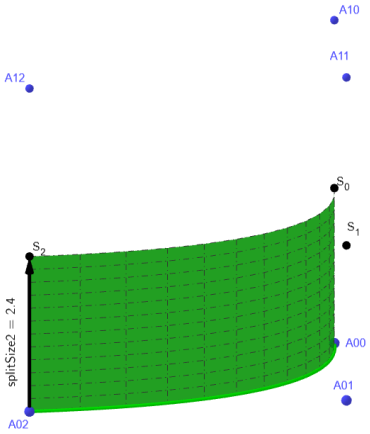
Die Umsetzung der Operation, in X- und Z-Richtung, kann wie bei ClosedSpline Shapes erfolgen, indem auf dem Basis-Spline der Spline3D Shape gearbeitet wird. Zusätzlich müssen die neu generierten Subshapes in Spline3D Shapes überführt werden, indem aus den Kurvensegmenten Flächen mit der Höhe der ursprünglichen Spline3D Shape erzeugt werden.

Der Split in Y-Richtung kann auch wieder auf 2 Fälle aufgeteilt werden. In den Split einer nicht geshifteten Shape und den Split einer geshifteten Shape. Ist die Shape nicht geshifted können die Subshapes aus dem Basis-Spline der Ursprungsshape und einer Höhe erzeugt werden. Die Höhe der Subshapes ergibt sich direkt über die Parameter, wenn sie absolut sind oder kann mittels linearer Interpolation der Höhe der Ursprungs-Shape berechnet werden. Zur Erzeugung der Subshapes werden die Kontrollpunkte aller Kurven des Basis-Splines jeweils um den Wert des entsprechenden *splitSize* Paramters in Y-Richtung verschoben. Die Kurven des Basis-Splines der Subshape müssen ab der zweiten Subshape um die Summe der vorherigen Höhen in Y-Richtung verschoben werden, um Überschneidungen zu vermeiden. Die Abbildung 4.5 zeigt die Anwendung der Split-Operation anhand einer einzelnen Bézierfläche, um die Darstellung zu vereinfachen. Bei Anwendung auf einer konkreten Shape würde das Vorgehen lediglich für jede Fläche wiederholt werden. Die Abbildung 4.5a zeigt die Erzeugung der ersten Subshape, bei der die Kontrollpunkte der Basiskurven um 2.6 in Y-Richtung verschoben werden. Die beiden Abbildungen 4.5b und 4.5c zeigen die Erstellung der zweiten Subshape. In 4.5b wird zunächst die Fläche mit der korrekten Höhe erstellt. In 4.5c werden die Kontrollpunkte der Fläche um die Summe der vorherigen splitSizes verschoben, sodass sie sich an der richtigen Position befindet.

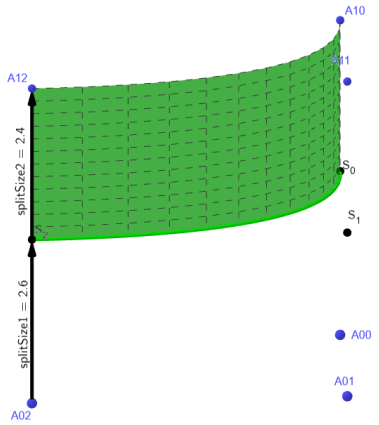
Ist die Shape geshifted muss neben der Höhe der Subshapes auch die Verschiebung in X- und Z-Richtung bestimmt werden. Dafür wird für die Kontrollpunkte aller Kurven des Basis-Splines eine Strecke zwischen dem aktuell betrachtetem Kontrollpunkt und seinem oberen Gegenstück gebildet. Für die Kontrollpunkte $(0,0), (0,1), (0,2)$ würden somit die Strecken $((0,0) -> (1,0)), ((0,1) -> (1,1)), ((0,2) -> (1,2))$ gebildet werden. Die Schnittpunkte der Schnittebene mit den erzeugten Strecken, bestimmen die Kontrollpunkte



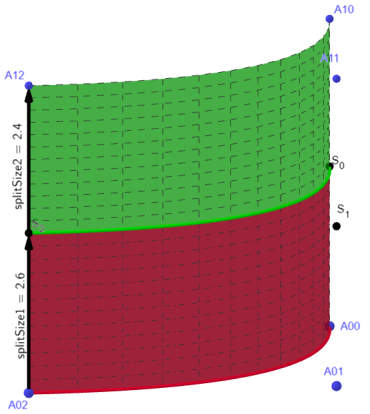

(a) Erzeugung von Subshape 1 durch Verschiebung der Basiskurve in Y-Richtung




(b) Erzeugung von Subshape 2 durch Verschiebung der Basiskurve in Y-Richtung




(c) Verschiebung der Kontrollpunkte um die Summe der vorherigen Verschiebungen




(d) Darstellung beider Subshapes

Abbildung 4.5: Darstellung der Split Operation einer nicht geshifteten Spline3D Shape

$(1,0),(1,1),...,(1,n)$ der neuen Flächen. Gleichzeitig bestimmen sie die Kontrollpunkte der Kurven des Basis-Splines der folgenden Subshape. Die Kontrollpunkte $(1,0),(1,1),...,(1,n)$ der Flächen der letzten Subshape sind die Kontrollpunkte $(1,0),(1,1),...,(1,n)$ der Flächen der Ursprungs-Shape.

Die Abbildung 4.6 zeigt die Split Operation einer geshifteten Spline3D Shape anhand einer einzelnen Bézierfläche. Die Schnittpunkte $S_0, S_1$ und $S_2$ dienen als Kontrollpunkte $(1,0),(1,1),(1,2)$ der ersten Subshape(rot), sowie als Kontrollpunkte der BasisKurve der

zweiten Subshape(grün). Die Kontrollpunkte $(1,0),(1,1),(1,2)$ der Subshape der zweiten Subshape(grün) sind die Kontrollpunkte $(1,0),(1,1),(1,2)$ der Ursprungs-Shape.

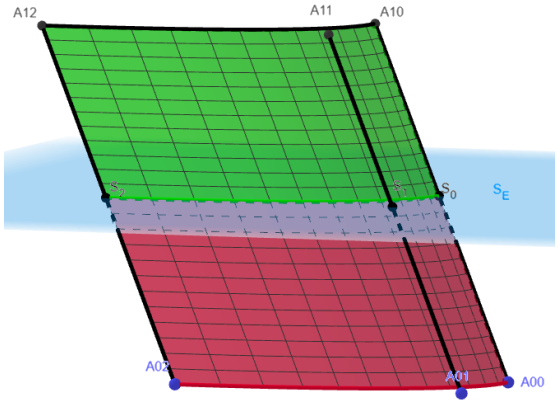



Abbildung 4.6: Darstellung der Split Operation über geshifteten Shape anhand einer Bezier-Fläche

### 4.4.3 ComponentSplit

Die Component-Split-Operation unterteilt eine Shape in eine neue Shape pro Rand-Element der ursprungs-Shape. Eine Shape vom Typ Prism wird somit in Polygon Shapes unterteilt, die die Randflächen repräsentieren. Die Operation kann auf Shapes vom Typ Prism und Spline3D angewendet werden und hat die folgende Sytax:

**<predecessorID> –> component_split(<componentsplitType>){<successorID1>, <successorID2>, ..., <successorIDn>}**

Der Parameter *componentsplitType* ist ein String und bestimmmt welche Randelemente ins Ergebnis aufgenommen werden. Die ünterstützten Argumente sind *„side_faces“*, wodurch die Seitenflächen der Shape als je eine eigene Shape zurückgegeben werden und das Argument *„top“* über das die Oberseite zurückgegeben wird.

Die Randelemente der Spline3D Shapes werden über die Flächen definiert. Die Randelemente für *„side_faces“* stellen dabei die einzelnen Bézierflächen dar, welche die Mantelfläche der Shape bilden. Das Randelement für *„top“* wird über den Spline der Deckfläche der Shape definiert.

### 4.4.4 makeSpline

Die makeSpline-Operation arbeitet auf PolygonShapes und überführt diese in ClosedSpline Shapes, indem die Ecken des Polygons abgerundet werden. Die Syntax sieht folgendermaßen aus:

**<predecessorID> –> make_spline() {<successorID>}**

Um die Polygone in Splines zu überführen müssen die Seiten des Polygons in Kurven überführt werden. Dafür werden in der Mitte der Seiten neue Punkte eingefügt. Diese Punkte bilden die Start- und Endpunkte der Kurven, während die Eckpunkte des Polygons die inneren Kontrollpunkte der Kurven bilden. In Abbildung 4.7 ist das Konzept der makeSpline

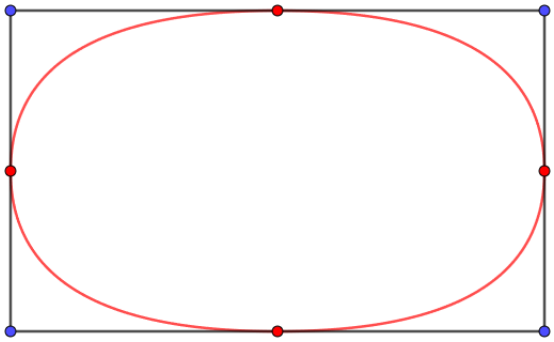


Abbildung 4.7: Visualisierung der Kontrollpunkte bei der Operation *makeSpline*

Operation visuell dargestellt. Die ursprüngliche Shape ist ein Polygon, dass über die blauen Punkte dargestellt ist. Die roten Punkte sind die Mittelpunkte der Strecken zwischen je zwei Kontrollpunkten des Polygons.

### 4.4.5 advancedExtrude

Die advancedExtrude-Operation ist eine erweiterte Extrude-Operation, die auf ClosedSpline Shapes angewendet werden kann. Die Operation ermöglicht es, dass die Shape während des „Herausziehens“ zusätzlich in X- und Z-Richtung verschoben wird und/oder die Shape während des Extrude Vorgangs rotiert wird. Die Rotation erfolgt dabei um die Y-Achse, die Rotation kann also abhängig von der Positionierung der Shape im globalen Koordinatensystem beeinflusst werden. Die Syntax der Operation sieht wie folgt aus:

**<predecessorID> –> adv_extrude(<height>, <shiftX>, <shiftZ>, <rotation>) {<successorID>}**

Der *height* Parameter ist äquivalent zum gleichnamigen Parameter der Extrude-Operation. Die Parameter *shiftX* und *shiftZ* geben die Verschiebung während des Extrude-Vorgangs an. Der Parameter *rotation* gibt den Rotationswinkel in Grad an, um den die Shape während des extrudes rotiert werden soll.

Die Umsetzung des Shifts kann in den Extrude-Vorgang der Standard Extrude-Operation integriert werden. Wenn die Kontrollpunkte der Basiskurven der Bézierflächen um den *height* Wert in Y-Richtung verschoben werden, müssen die Kontrollpunkte zusätzlich um den Wert *shiftX* in X-Richtung und um *shiftZ* in Z-Richtung verschoben werden.

Für die Rotation wird der maximale Rotationswinkel in $n$ gleichgroße Schritte aufgeteilt. Dabei ist die Maximalgröße eines solchen Rotationsschritts $45°$. Die Kontrollpunkte der Kurvensegmente der SplineShape werden dann pro Rotationsschritt zunächst, wie beim Standard Extrude in Y-Richtung verschoben. Die Höhe der Verschiebung hängt in diesem Fall aber von der Anzahl der Rotationsschritte ab und berechnet sich wie folgt. Sei $t$ der aktuell betrachtete Rotationsschritt, $n$ die Anzahl aller Schritte und $h$ das Argument *height* der Operation, dann gilt für die Verschiebung des aktuellen Rotationsschritts $h_{\text{step}} = h \cdot \frac{t}{n}$. Auf die gleiche Weise werden die Werte für die Verschiebung in X- und Z-Richtung berechnet. Nachdem die Kontrollpunkte verschoben worden sind, werden sie, um den aktuellen Rotationswinkel, um die Y-Achse rotiert. Der Winkel ergibt sich aus der Summe des aktullen und allen vorherigen Rotationsschritten.

## 4.5 Konzepte Mesh Generierung

Die verwendeten Meshes sollen wie bei den bereits bestehenden Shapes Triangle-Meshes sein. Die Vertices für die Mesh Generierung können dafür dynamisch über die Kurvensegmente berechnet werden. Es ist zu beachten, dass Krümmungen über die Meshes nicht exakt dargestellt werden können. Sie können aber approximiert werden. Dafür müssen die Vertices am Rand einer Shape möglichst dicht beieinander liegen. Für die Berechnung der Vertices muss daher ein kleines $t$ gewählt werden. Die eigentliche Triangulation erfolgt dann über den Ear-Clipping

Algorithmus.

Für organische 3D Shapes können Vertices auf der Fläche mittels eines Parameters u und v berechnet werden, die ein Punktgitter bilden. Aus diesem Gitter können dann Triangles gebildet werden. Der Parameter $u$ sollte hier möglichst klein gewählt werden, um die Krümmung in X-Richtung möglichst gut approximieren zu können.

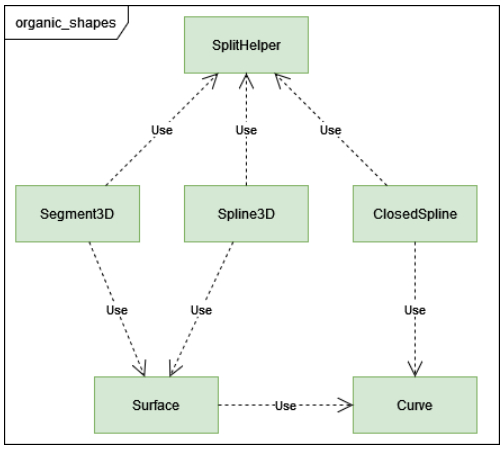
## 4.6 Abgrenzung bestehender Bestandteile zu neuen Bestandteilen

Die Logik der Shape Grammar, wie das Erstellen einer Grammatik aus einer .grammer Datei, sowie die Erstellung des Shape Trees basierend auf der erzeugten Grammatik war bereits vollständig implementiert. Die Erweiterungen von Klassen bzw. das Einfügen neuer Klassen wurde daher auf der Ebene der Shape Klassen und der Klassen der Shape-Operationen durchgeführt.

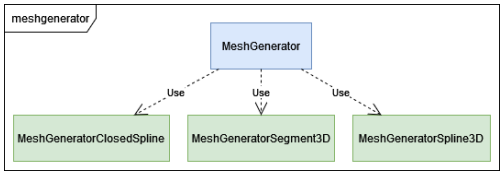
Für die Nutzung einer konkreten Shape werden jeweils eine Shape-Klasse und ein Mesh Generator für die entsprechende Shape-Klasse benötigt. Für die drei neuen Shapes *ClosedSpline*, *Spline3D* und *Segment3D* müssen daher die ShapeKlassen *ClosedSplineShape*, *Splime3DShape* und *Segment3DShape* sowie die Mesh Generatoren *MeshGeneratorClosedSpline*, *MeshGeneratorSpline3D* und *MeshGeneratorSegment3D* implementiert werden. Zusätzlich sollen für die Shape-Klassen die Hilfsklassen *SplitHelper*, *Curve* und *Surface* implementiert werden.

Auf der Ebene der Shape-Operationen müssen die beiden neuen Klassen *ShapeOperationMakeSpline2D* und *ShapeOperationAdvancedExtrude* für die Shape-Operationen makeSpline und advancedExtrude implementiert werden. Die Klassen der Shape-Operationen Extrude, Split und ComponentSplit müssen erweitert werden, sodass die Operationen mit den neuen Shapes kompatibel sind. Bei der Component-Split-Operation ist es daher nötig eine Component-Split-Strategy in Form der Klasse *ComponentSplitSpline3D* zu implementieren.

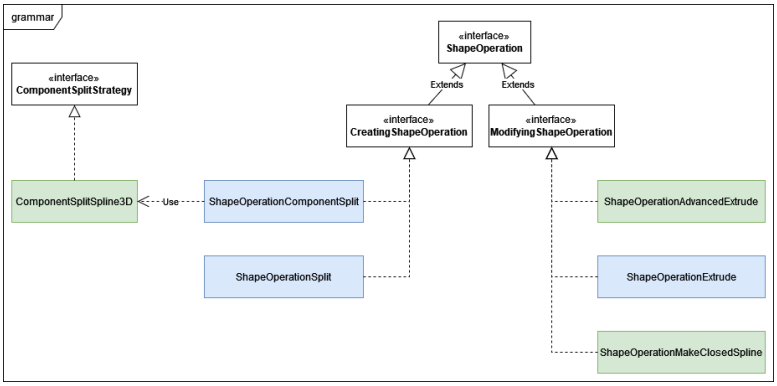
Die Abbildung 4.8 zeigt Diagramme der Klassen die erweitert werden sollen bzw. die neu hinzugefügt werden sollen. Der Status der Klassen ist dafür farblich markiert. Klassen die neu hinzugefügt wurden, sind grün hinterlegt, Klassen die erweitert wurden, sind blau hinterlegt und Klassen die ohne Änderungen übernommen wurden, sind weiß hinterlegt.




(a) Klassen des Package organic_shapes




(b) Klassen des Package meshgenerator




(c) Klassen des Package grammar

Abbildung 4.8: Auflistung der neuen(grün) bzw. erweiterten(blau) Klassen

## 4.7 Anforderungen

- Erzeugung einer zweidimensionalen gekrümmten Shape aus einem Polygon
- Unterteilung der zweidimensionalen gekrümmten Shapes in zwei oder mehrere Subshapes
- Überführung der zweidimensionalen gekrümmten Shapes in dreidimensionale gekrümmten Shapes
- Überführung der zweidimensionalen gekrümmten Shapes in verzerrte dreidimensionale gekrümmten Shapes

- Überführung der zweidimensionalen gekrümmten Shapes in rotierte dreidimensionale gekrümmten Shapes
- Unterteilung der dreidimensionalen gekrümmten Shapes in zwei oder mehrere Subshapes
- Unterteilung der dreidimensionalen gekrümmten Shapes in ihre Randelemente
- unabhängige Darstellung der Randelemente
- Unterteilung der Randelemente in zwei oder mehrere Subshapes
- Überführung einzelner Randelemente in dreidimensionalen gekrümmten Shapes
- Darstellung der zweidimensionalen gekrümmten Shapes mittels eines Triangle-Meshes
- Darstellung der dreidimenionalen gekrümmten Shapes mittels eines Triangle-Meshes
- Darstellung einzelner Randelemente mittels eines Triangle-Meshes

# 5 Realisierung

## 5.1 Allgemeiner Aufbau der Shapes

Jede Shape, mit Ausnahme des Axioms, wird mithilfe einer Parent Shape und einer Shape Operation erstellt. Um die Shape eindeutig identifizieren zu können, wird dabei jeder Shape eine ID zugewiesen. Jede Shape muss also in der Lage sein, Informationen über ihren Zustand zu speichern. In Abbildung 5.1 ist ein Klassendiagramm dargestellt, das den Aufbau der Informationsverwaltung zeigt. Die Klassen PolygonShape und PrismShape dienen dabei als Beispiele für konkrete Klassen.

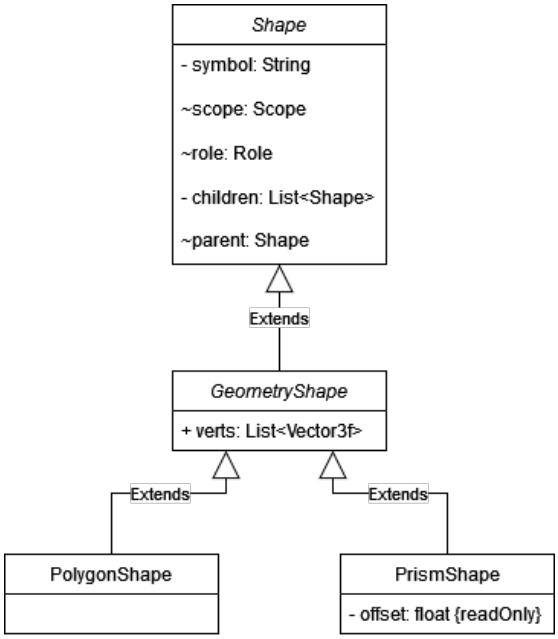



Abbildung 5.1: Klassendiagramm allgemeiner Aufbau der Shapes

Im folgenden sollen die Felder der abstrakten Klasse Shape anhand eines Beispiels erläutert werden. Sei das Axiom mit der ID *Start* eine PolygonShape.

Die Regel erzeugt aus dem Polygon Axiom eine Prism Shape:

*Start –> Extrude(2) Prism*

Die abstrakte Klasse Shape beinhaltet die allgemeinen Informationen der Shape. Die Variable *symbol* dient als ID für die Shape und wird im Ableitungsprozess für die Auswahl einer Regel benötigt. Im Beispiel hat das Axiom das symbol *Start* und kann somit abgeleitet werden. Das Symbol der abgeleiteten PrismShape ist *Prism*. Die Shapes haben zudem eine Referenz auf die Shape, aus der sie abgeleitet worden sind. Dies wird im Feld *parent* gespeichert. Am Beispiel gesehen, ist der *parent* der Shape *Prism* die Shape *Start*. Zusätzlich zum Feld *parent* ist auch die andere Richtung der Ableitung, in Form des Feldes *children* gegeben. Eine Shape kann dabei keine oder auch mehrere Kinder haben. Im Beispiel hat die Shape *Start* die Shape *Prism* als Kind. Das Feld *scope* definiert einen Scope, ein lokales Koordinatensystem, für eine Shape. Der Scope einer Shape stellt dabei auch Informationen über die Außmaße der Shape zur Verfügung. Die Variable *role* bestimmt, ob es sich um eine Dach Shape handelt oder nicht. Handelt es sich um eine Dach Shape ist der Wert ROOF, ansonsten ist der Wert DEFAULT. Da die organischen Formen nicht für den Einsatz als Dach implementiert worden sind, wird diese Variable im Verlauf der Arbeit immer als DEFAULT angenommen.

Die abstrakte Klasse *GeometryShape* ist die Basis für alle Shapes die mithilfe von Vertices dargestellt werden können. Dazu gehören wie in Abbildung 5.1 zu sehen beispielsweise die Polygon und die Prism Shapes. Die Klasse speichert die Vertice Information in einer Liste im Feld *verts*.

## 5.2 Erweiterung der Shape Klassen: ClosedSpline

Die bestehenden Shape Klassen werden um die Klasse *ClosedSpline* erweitert. Diese Klasse repräsentiert die organische Formen im 2D-Raum. Als interne Repräsentation der Shape nutzt die Klasse einen Spline in Form des Feldes *segments*, einer Liste von Curve Objekten 5.3. Die anderen allgemeinen Information zur Shape wie das Symbol, die parent Shape und die children Shapes wird von der abstrakten Klasse Shape vererbt. In der Abbildung ist der Aufbau der *ClosedSplineShape* Klasse in einem Klassendiagramm dargestellt5.2.

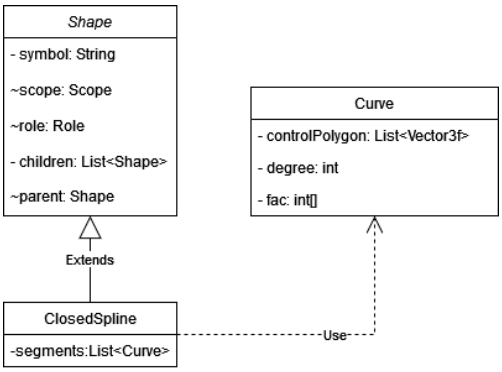



Abbildung 5.2: Klassendiagramm Aufbau ClosedSplineShape

## 5.3 Curve

Die Klasse *Curve* ist eine Hilfsklasse, die eine Implementation für Bézierkurven bereitstellt. Die Bézierkurve wird intern über eine Liste von Kontrollpunkten und einen Grad definiert. Der Verlauf der Kurve ist dadurch mathematisch über die Formel ... definiert. Die Kontrollpunkte werden über Objekte des Typs *Vector3f*, also dreidimensionale Vektoren definiert. Die Reihenfolge der Kontrollpunkte innerhalb der Liste ist dabei entscheidend für den Verlauf der Kurve. Das erste Element der Liste ist stets der Anfangspunkt der Kurve, die folgenden Elemente beziehen sich auf die inneren Kontrollpunkte der Kurve und das letzte Element ist der Endpunkt der Kurve. Der Grad wird als Integer über das Feld *degree* definiert und wird vorallem zur Berechnung der Basisfunktion verwendet.

Neben der Definition des Verlaufs der Kurve beinhaltet die Klasse *Curve* Operationen die auf einzelne Kurven angewendet werden können. Die Methode *calculateVertices()* berechnet mithilfe der Bernsteinpolynome als Basisfunktion Vertices die auf der Kurve liegen. Bei der Berechnung wird dafür der Wert von t, beginnend bei 0, schrittweise inkrementiert, bis der Wert 1 erreicht wird. Die Schrittweite des Inkrements wird über das Feld *stepSizeT* bestimmt.

## 5.4 Erweiterung der Shape Klassen: Spline3D

Die Klasse *Spline3DShape* ist eine Implementation für organische Formen im 3D-Raum. Wie im Konzept 4.3 bereits erwähnt, werden für die Darstellung Bézierflächen genutzt, welche die Mantelfläche der Shape repräsentieren. Die einzelnen Bézierflächen werden über das Feld *segments* verwaltet. Dabei handelt es sich um eine Liste von *Surface* Objekten. Die Klasse

*Spline3DShape* erbt wie die *ClosedSplineShape* Klasse auch die allgemeinen Variablen über die Klasse *Shape*.

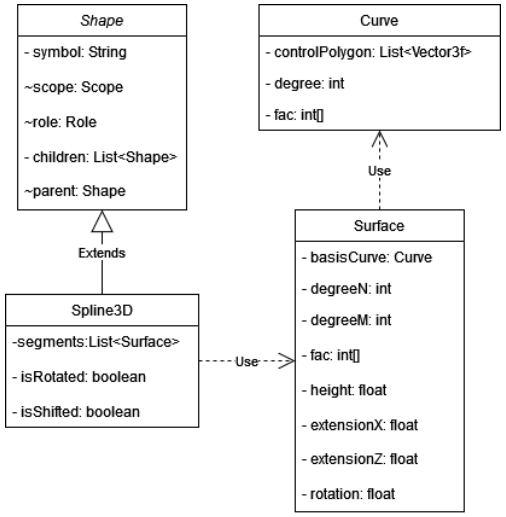



Abbildung 5.3: Klassendiagramm Aufbau der Spline3DShape

## 5.5 Erweiterung der Shape Klassen: Segment3D

Die Klasse *Segment3DShape* ist eine weitere Klasse für dreidimensionale organische Formen. Diese Klasse ist ähnlich zur *Spline3DShape* Klasse aufgebaut. Die *Segment3DShape* stellt jedoch nur eine einzelne Bézierfläche dar. Dafür stellt sie anstatt einer Liste von Surface Objekten ein einzelnes *Surface* Objekt bereit. Diese Klasse wird hauptsächlich zur Realisierung der Component-Split-Operation auf *Spline3DShape* Objekten benötigt, da ein Component-Split auf einer *Spline3DShape* einzelne Bézierfläche als Ergebnis zurückgibt. Die Klasse *Spline3DShape* kann dafür nicht verwendet werden, da Operationen wie Extrude nicht für *Spline3DShapes* definiert sind.

## 5.6 Surface

Die Klasse *Surface* ist eine Hilfsklasse, die eine Implementation für Bézierflächen bereitstellt. Die so erzeugten Bézierflächen sind auf die Anwendung innerhalb der beiden Klassen *Spline3DShape* und *Segment3DShape* abgestimmt. Die Bézierfläche wird intern über die Felder *basisCurve*, *extensionX*, *height*, *extensionZ* und *rotation* definiert. Das Feld *basisCurve* ist eine Kurve, welche die unteren Kontrollpunkte der Fläche definiert. Die Felder *rotation*, *height*

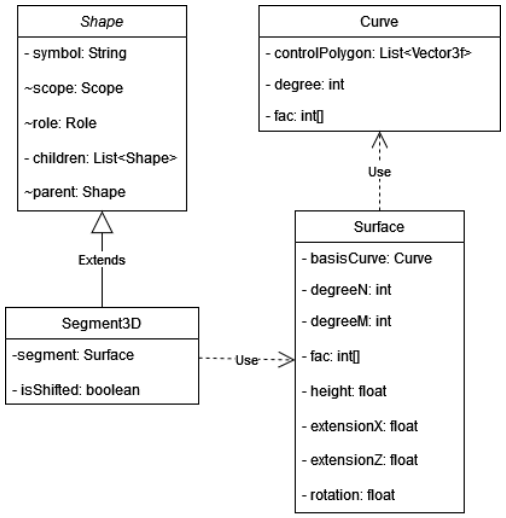


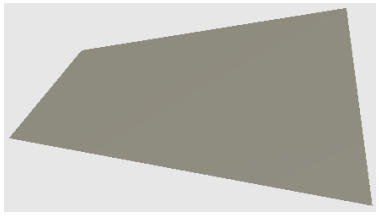

Abbildung 5.4: Klassendiagramm Aufbau der Segment3DShape

und *extensionZ* bilden zusammen einen Verschiebungsvektor, wobei *extensionX* den X-Anteil, *height* den Y-Anteil und *extensionZ* den Z-Anteil darstellt. Das Kontrollpunkte Netz der Fläche kann über diese Variablen mithilfe der Methode *getControlNet()* berechnet werden. Das Kontrollpunkte Netz wird in Form eines *Vector3f* Arrays berechnet. Die Art der Berechnung der Kontrollpunkte hängt davon ab, ob die Shape rotiert ist. Ist die Shape nicht rotiert, ist der Grad der Fläche $n = 1, m =$ Grad der Basiskurve. Die Kontrollpunkte [0][0], [0][1], ..., [0][m] ergeben sich aus der Belegung der Kontrollpunkte der Basiskurve. Die Kontrollpunkte [1][0], [1][1], ..., [1][m] werden berechnet indem die Kontrollpunkte der Basiskurve um den Verschiebungsvektor verschoben werden. Ist die Fläche rotiert, werden die Kontrollpunkte der Basiskurve schrittweise rotiert und verschoben. Dieser Vorgang ist für hohe Rotationswinkel nötig, um eine gleichmäßige Rotation der Shape zu realsieren. Würden die Kontrollpunkte der Basiskurve in einem Schritt rotiert werden, würde beispielsweise bei einer Rotation von 360° keine Rotation sichtbar sein. Die einzelnen Schritte werden so gewählt, dass die Erhöhung des Rotationswinkels pro Schritt gleichbleibend ist und 45° nicht überschreitet. Dafür wird die Anzahl der Schritte berechnet über: $n =$ rotation$/45$. Die Schrittweite pro Schritt kann mittels: rotationPerStep $=$ rotation $* 1/n$ berechnet werden. Der Verschiebungsvektor muss ebenfalls angepasst werden: veschiebungSchritt $=$ verschiebungsvektor $* s/n$, wobei $s$ der aktuelle Schritt beginnend bei $s = 0$ ist. Danach werden die Kontrollpunkte iterativ berechnet. Dafür werden die Kontrollpunkte der Basiskurve zunächst mittels des inversen Transfomationsvektors des Scopes in das globale Koordinatensystem überführt. Dort werden sie um den entsprechenden Verschiebungsvektor des aktuellen Schrittes verschoben und um den entspre-

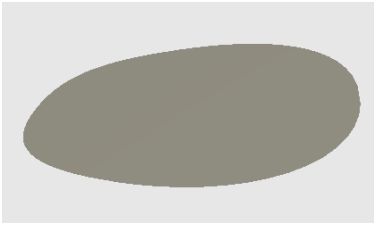
chenden Winkel um die Z-Achse rotiert. Die Ergebnisse werden zurück in Scopekoordinaten übersetzt und dem Array an der Stelle [s][0], [s][1],..., [s][m] hinzugefügt.

## 5.7 Operation: make_spline

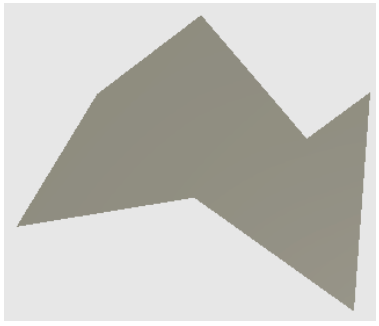
Die Umsetzung basiert, wie im Konzept erwähnt, auf dem Einfügen neuer Vertices auf den Strecken zwischen benachbarten Vertices des Polygons. In der konkreten Umsetzung wird über die Vertice Liste der Polygon Shape iteriert, in jedem Durchlauf $i$ wird dabei der Mittelpunkt der Strecke von Vertex $i$ zu Vertex $(i+1)\%$numberOfVertices, also der Mittelpunkt zweier benachbarter Vertices des Polygons berechnet. Die berechneten Mittelpunkte werden in einer Liste *newStartingPoints* zusammengefasst. Über diese Liste wird nun iteriert und kann somit in Verbindung mit den ursprünglichen Vertices des Polygons dazu genutzt werden, Kurvensegmente zu erstellen. Sei dabei $n$ der aktuelle Iterationsschritt, dann enthält das Kurvensegment die folgenden Kontrollpunkte, *newStartingPoints.get(n), polygonShape.get(n+1%numberOfVertices)* und *newStartingPoints.get(n+1%numberOfVertices)*. In Abbildung 5.5 sind zwei verschiedene Polygone und die entsprechenden Ergebnisse der *make_spline* Operation dargestellt.



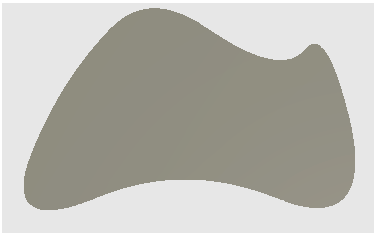
(a) Einfaches Polygonales Axiom



(b) Überführung des Axioms in eine Spline2D Shape



(c) Komplexeres Polygonales Axiom



(d) Überführung des Axioms in eine Spline2D Shape

Abbildung 5.5: Darstellung der Shape Operation make_spline

## 5.8 Operation: Extrude und advancedExtrude

Diese Section beschäftigt sich mit der Anwendung der *extrude* Operation auf *ClosedSpline* und *Segment3D* Shapes, sowie mit der Anwendung der *advancedExtrude* Operation auf die Shape *ClosedSpline*.

### 5.8.1 Umsetzung auf ClosedSpline Shapes

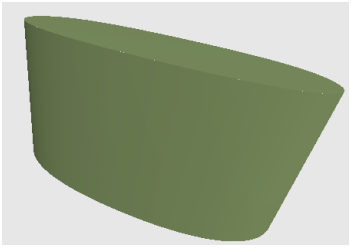
Die Umsetzung der Extrude-Operation auf *ClosedSpline* Shapes erfolgt durch das Überführen der einzelnen Kurvensegmente des Splines in Bézierflächen. Dafür kann aus der aktuellen Kurve und einer Höhe, die über den *height* Parameter der Operation definiert ist, ein Bézierflächen-Objekt erstellt werden. Die genauen Kontrollpunkte der Fläche werden dann intern über die Methode *getControlNet()* berechnet. Der Ablauf der *advancedExtrude* Methode ist ähnlich. Auch hier wird für aus jeder Kurve eine Fläche erzeugt. In diesem Fall müssen beim Erstellen der Klasse die zusätzlichen Parameter *extensionX* und *extensionZ* gesetzt werden. In Abbildung 5.6 sind die Ergebnisse verschiedener Extrude-Operationen visualisert. Die Abbildung 5.6a zeigt ein Standard extrude. In Abbildung 5.6b wird ein advanced_extrude mit Verschiebung in X- und Z-Richtung dargestellt. Und in Abbildung 5.6c und 5.6d werden advanced_extrude-Operationen mit Rotation dargestellt.

### 5.8.2 Umsetzung auf Segment3D Shapes

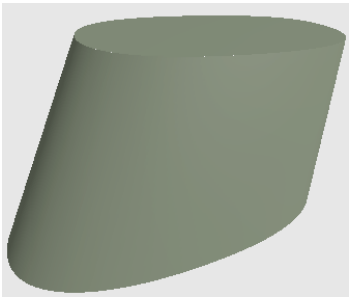
Die Umsetzung der Extrude-Operation auf *Segment3DShapes* erfolgt über die Methode *extrude(float)* der *Segment3DShape* Klasse. Die Methode berechnet einen Verschiebungsvektor über den eine verschobene Kopie der Basiskurve der Bézierfläche der *Segment3DShape* erstellt wird. Die so erzeugte Kurve bildet ein Segment eines Splines und wird um die Basiskurve der *Segment3DShape* Fläche, sowie zwei Verbindungskurven erweitert, die den Spline schließen. Die beiden Verbindungskurven besitzen zwei Kontrollpunkte, die jeweils den Startpunkt der Basiskurve und den Startpunkt der verschobenen Basiskurve bzw. den Endpunkt der Basiskurve und den Endpunkt der verschobenen Basiskurve wiedergeben. Aus dem so erzeugten Spline und den Feldern *extensionX*, *height* und *extensionZ* der Fläche der *Segment3DShape* wird ein *Spline3DShape* Objekt erstellt.

#### Berechnung des Verschiebungsvektors

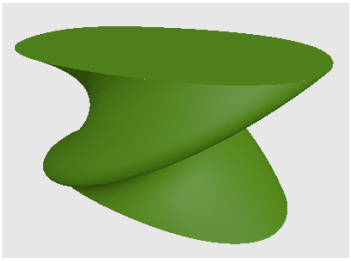
Für die Berechnung des Verschiebungsvektors wird eine Ebene erstellt, die ortogonal zur Strecke zwischen den beiden äußeren Kontrollpunkten der Basiskurve verläuft. Als Punkt $p$ der Ebene



(a) Einfaches Extrude eine Spline2D Shape



(b) Advanced Extrude mit Verschiebung in X- und Z-Richtung



(c) Advanced Extrude mit Rotation um 180°(Y-Achse = Mittelpunkt)

(d) Advanced Extrude mit Rotation um 360°(Y-Achse $\neq$ Mittelpunkt)

Abbildung 5.6: Darstellung der Ergebnisse der extrude Operationen

wird der innere Kontrollpunkt der Basiskurve genutzt. Die Normale $n$ ist ein Einheitsvektor der vom Startpunkt der Kurve in Richtung Endpunkt der Kurve zeigt. Berechnet wird die Normale über die Normierung des Vektors $\vec{se} = \vec{e} - \vec{s}$. Der Vektor $\vec{e}$ stellt dabei den Endpunkt der Kurve dar und der Vektor $\vec{s}$ entsprechend den Startpunkt. Die so definierte Ebene verläuft ortogonal zum Vektor $\vec{se}$. Die Richtung des Verschiebungsvektors ergibt sich dann durch Normierung des Vektors $\vec{ip} = \vec{p} - \vec{i}$, wobei $\vec{i}$ der Schnittpunkt der Strecke zwischen den äußeren Kontrollpunkten und Ebene ist und der $\vec{p}$ der Punkt $p$ der Ebene ist. Der Verschiebungsvektor wird dann durch Skalierung des Vektors $\vec{ip_0}$ mit dem *height* Parameter der Methode erstellt.

## 5.9 Operation: Split

### 5.9.1 Vorverarbeitung Split

Im folgenden soll zunächst auf die Vorverarbeitung der Split Values eingegangen werden. Wie im Konzept erwähnt, erfolgt die Berechnung der SubShapes über Ebenen. Eine Subshape

ist dabei ein Teil der Shape, die im positiven Raum von je zwei Ebenen den sogenannten *splittingPlanes* liegt, zu sehen in Abbildung 5.7. Diese zeigt welcher Teil einer Shape für die zwei Ebenen, definiert über $p_1 = (0,0), n_1 = (1,0)$ und $p_2 = (6,0), n_2 = (-1,0)$, als *subShapes* bestimmt wird. Die *subShape* ist durch Schraffur hervorgehoben.

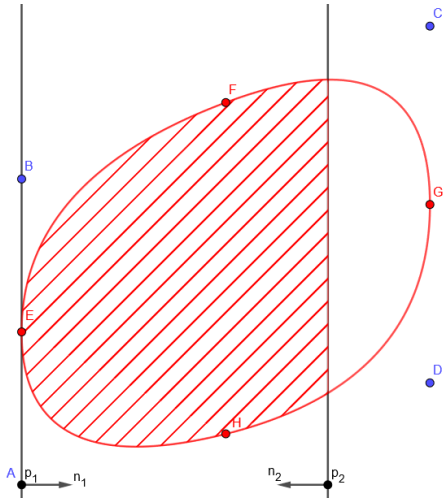



Abbildung 5.7: Definition einer *subShape* über Ebenen

Im ersten Schritt werden die Values der Split-Operation in absolute Größen umgerechnet, die die Ausdehnung der splits auf der entsprechenden Achse bestimmen. Über diese Werte können dann *splittingPlanes* erzeugt werden, die über einen Punkt auf der Ebene $p$ und eine Normale $n$ definiert werden. Die Normale einer *splittingPlane* ist dabei stets ein Einheitsvektor der auf der splitting Achse verläuft.

Die erste erzeugte *splittingPlane* liegt stets im Koordinatenursprung, hat also immer den Wert $p = (0,0,0)$. Der Wert $p$ der folgenden Planes wird berechnet aus einem *offset* und einer *splitSize*(Element aus *absoluteSplitSizes*). Der Wert des *offset* startet bei $0$ und wird nach der Berechnung einer Plane um die verwendete *splitSize* erhöht. Die Berechnung des Punktes $p$ erfolgt dann über:

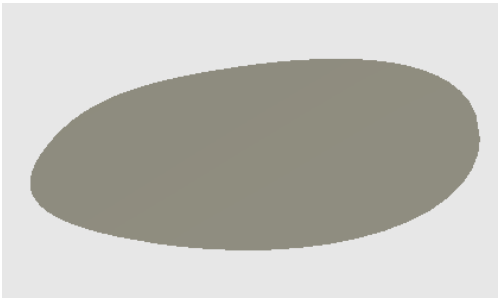
$$p = (\text{offset} + \text{splitSize}) * n$$

Die so berechneten *splittingPlanes* haben ihren positiven Raum entlang der entsprechen-

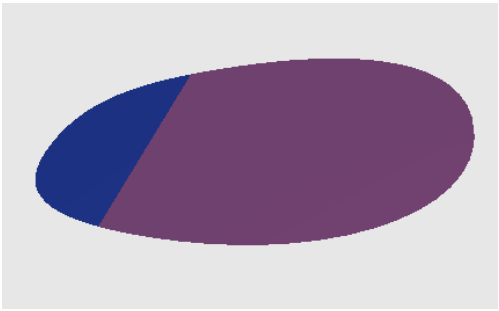
den Achse. Bei Anwendung der Split Operation muss daher die hintere *splittingPlane* invertiert werden.

### 5.9.2 Umsetzung der Split Operation auf ClosedSplineShapes

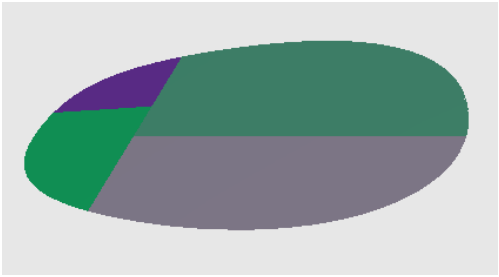
Die Split-Operation wird mittels der Methode *splitBetween(List<Plane> splittingPlanes, Axis axis, String id)* umgesetzt. Die Methode berechnet die Subshape die durch die beiden *Plane* Elmente der *splittingPlanes* Liste definiert sind. Die Methode berechnet für die einzelnen Kurven des Splines die entsprechenden Teilkurven und definiert über diese dann den Spline der Subshape. Dafür werden die folgenden Schritte iterativ auf den einzelnen Kurven durchgeführt. Abbildung 5.8 zeigt die Ergebnisse einer Extrude Operation angewendet auf eine *ClosedSplineShape*. Die Abbildung 5.8b zeigt dabei die Unterteilung in X-Richtung mit den Split-Sizes $0.4r$ und $0.6r$. In Abbildung 5.8b werden die beiden Subshapes erneut unterteilt, diesmal in Z-Richtung. Die Split-Sizes der linken Shape sind dabei $0.4r$ und $0.6r$ und die der rechten Shape $0.6r$ und $0.4r$.



(a) Ausgangsshape für die split Operation



(b) Unterteilung mittels split("X", 0.4r, 0.6r)



(c) Erneute Unterteilung der Shapes in Z-Richtung)

Abbildung 5.8: Darstellung der Ergebnisse der Shape Operation *split* für 2D-Shapes

### 5.9.3 Verarbeitung pro Kurve

Zunächst werden die Schnittpunkte der *splittingPlanes* mit der Kurve berechnet.Basierend auf den Ergebnissen der Schnittpunktberechnung kann für eine Kurve einer von vier Fällen eintreten, die im folgenden erläutert werden.

Beide t-Werte der Schnittpunktberechnung liegen bei $t < 0$ oder $t > 1$. In diesem Fall wird überprüft, ob der Start- und Endpunkt der Kurve zwischen beiden *splittingPlanes* liegt. Liegt die Kurve innerhalb der beiden *splittingPlanes* wird sie unverändert übernommen. Liegt sie außerhalb, ist die Kurve nicht betroffen und wird nicht übernommen. In diesem Fall sind keine weiteren Verarbeitungen nötig und die nächste Teilkurve kann berechnet werden.

Der Schnittpunkt der ersten Plane liegt bei $t_{P1} < 0$ oder $t_{P1} > 1$ und der Schnittpunkt der zweiten Plane liegt bei $0 <= t_{P2} <= 1$. In diesem Fall befindet sich die Kurve vom Startpunkt $t_1 = 0$ bis zum Punkt $t_2 = t_{P2}$ in der Subshape. Das Ergebniss wird somit auf $t_1 = 0, t_2 = t_{P2}$ gesetzt.

Der Schnittpunkt der ersten Plane liegt bei $0 <= t_{P1} <= 1$ und der Schnittpunkt der zweiten Plane liegt bei $t_{P2} < 0$ oder $t_{P2} > 1$. In diesem Fall befindet sich die Kurve vom Punkt für $t_1 = t_{t_{P}1}$ bis zum Punkt für $t_2 = 1$ in der Subshape. Das Ergebniss wird somit auf $t_1 = t_{P1}, t_2 = 0$ gesetzt.

Liegen beide Schnittpunkte innerhalb vom Intervall, $[0, 1]$ dann wird eine Teilkurve beginnend bei $t_{P1}$ bis zu $t_{P2}$ berechnet.

### 5.9.4 Berechnung der Teilkurven

Die Methode *splitCurveAt(float t1, float t2)* berechnet eine Teilkurve die beim Punkt für *t1* startet und beim Punkt für *t2* endet. Die inneren Kontrollpunkte werden mithilfe des De-Casteljau Algorithmus 2.4.1 berechnet, sodass der ursprüngliche Kurvenverlauf beibehalten wird. Ein Beispiel für eine Kurve vom Grad $n = 2$ mit den Kontrollpunkten $P_0 = (1, 1), P_1 = (4, 6), P_2 = (10, 2)$ ist in Abbildung 5.9 zu sehen. Ist der Startpunkt bei $t = 0$ bzw. der Endpunkt bei $t = 1$ kann der einfache De-Casteljau Algorithmus angewendet werden. Hierfür wird das De-Casteljau Dreieck durch lineare Interpolation zwei benachbarter Punkte, mit dem entsprechenden Wert für $t$, gebildet. Im Falle des Starts bei $t_1 = 0$ bildet die linke Diagonale

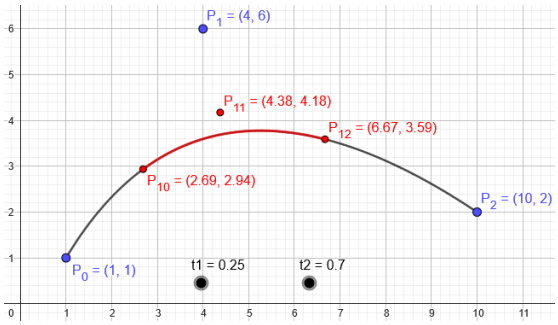
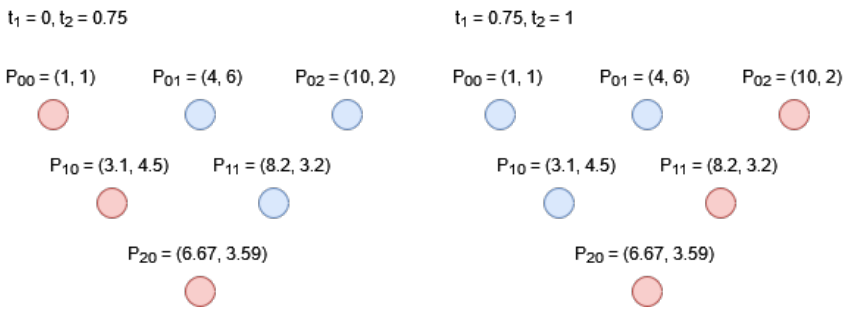
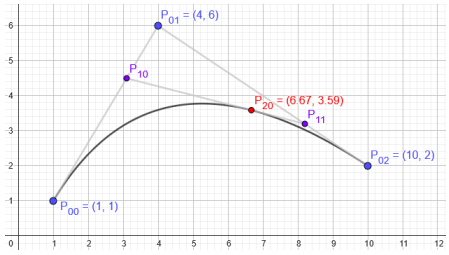



Abbildung 5.9: Beispielkurve für Methode splitCurveAt

die Kontrollpunkte der Teilkurve, ist der Endpunkt $t_2 = 1$ bildet die rechte diagonale die Kontrollpunkte der Teilkurve, zu sehen in Abbildung 5.10a und 5.10b.
Ist der Startpunkt $t_1 \neq 0$ und der Endpunkt $t_2 \neq 1$ muss der De-Casteljau Algorithmus zweimal




(a) Kontrollpunkte(rot) für $t_1 = 0$, $t_2 = 0.75$ (b) Kontrollpunkte(rot) für $t_1 = 0.75$, $t_2 = 1$

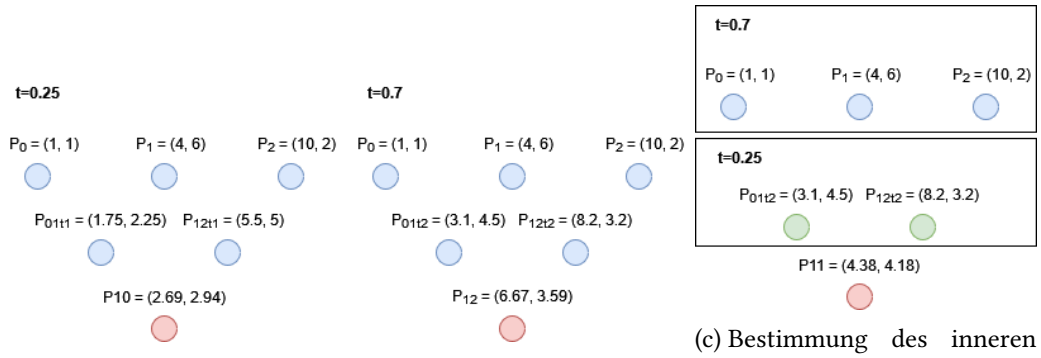



(c) Visualisierung lineare Interpolation

Abbildung 5.10: De-Casteljau Dreiecke für Beispiel

durchlaufen werden. In diesem Fall ist der Startpunkt definiert über die Spitze des De-Castelaju Dreiecks für $t = t_1$ und der Endpunkt ist definiert über die Spitze des De-Casteljau Dreiecks für $t = t_2$. Der innere Kontrollpunkt wird dann berechnet, indem die lineare Interpolation zur Berechnung der Spitze des De-Casteljau Dreiecks mit dem $t$ Wert des jeweils anderen

Durchlaufs durchgeführt wird. Dieser Vorgang ist in 5.11 visualisiert. Die Abbildungen 5.11a und 5.11b stellen dabei die De-Casteljau Dreiecke für die Werte $t_1$ und $t_2$ dar. Der sich daraus ergebende Start- oder Endpunkt ist jeweils hervorgehoben. Die Abbildung 5.11c zeigt die Bestimmung des inneren Kontrollpunktes. Die in grün dargestellten Punkte werden dabei mit dem anderen $t$ interpoliert.




(a) De-Casteljau Dreieck für $t_1$ (b) De-Casteljau Dreieck für $t_2$ (c) Bestimmung des inneren Kontrollpunktes

Abbildung 5.11: De-Casteljau Dreiecke für Beispiel

### 5.9.5 Erzeugung der Subshape

Nachdem alle Teilkurven berechnet wurden, kann die Subshape aus den Teilkurven erstellt werden. Dazu müssen die äußeren Kontrollpunkte der Teilkurven die keinen Verbindungspunkt aufweisen, mit dem entsprechenden Gegenstück durch eine Kurve vom Grad $n = 1$ verbunden werden. Abbildung 5.12 zeigt ein Beispiel anhand einer einfachen orgnaischen Shape. Die in grün dargestellten Strecken bilden die eingefügten Kurven vom Grad $n = 1$.

### 5.9.6 Umsetzung der Split Operation auf Spline3D Shapes

Die Umsetzung des Splits auf *Spline3DShapes* erfolgt mitttels der Methode *splitBetween(List<Plane> splittingPlanes, Axis axis, String id)* der *Spline3DShape* Klasse. Wie bereits im Konzept erwähnt muss diese Shape auch den Split in Y-Richtung unterstützen. Der Split in X- und Z-Richtung ist für die Spline3D Shapes nur anwendbar, wenn die Shape nicht geshiftet ist. Basierend darauf ist die Methode *splitBetween()* der Spline3D Shape in drei Hilfsmethoden aufgeteilt, die jeweils eine Art split abdecken. Die Methode *planeSplitXZ(List<Plane> splittingPlanes, Axis axis, String id)* ist für den Split in X- und Z-Richtung verantwortlich. Die Methoden *planeSplitY(List<Plane> splittingPlanes, String id)* und *tiltedSplit(List<Plane> splittingPlanes, String id)* sind jeweils für

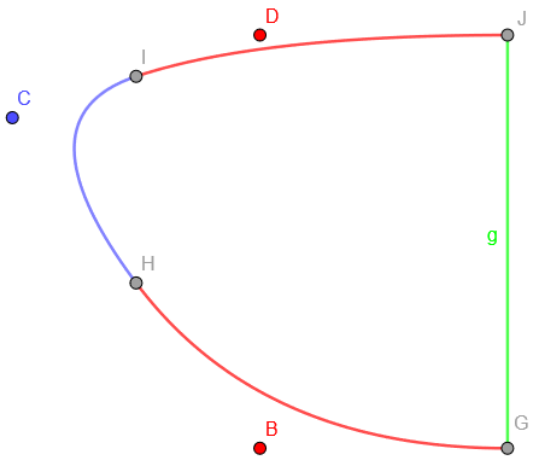



Abbildung 5.12: Darstellung der Verbinsungsstrecke

den Split einer nicht geshifteten Shape und den Split einer geshifteten Shape in Y-Richtung zuständig.

### 5.9.7 planeSplitXZ

Basierend auf der Einschränkung, dass eine Spline3D Shape nur in X- und Z-Richtung gesplitted werden kann, wenn die Shape nicht geshiftet ist, kann der Split wie auf ClosedSpline Shapes angewendet werden. Die dafür verwendeten Kurven sind die Basis-Kurven der einzelnen Flächen die im Feld *basisCurve* abgelegt sind. Sind die Teilkurven berechnet und der neue Spline gebildet, müssen die einzelnen Kurven des Spline wieder in Flächen überführt werden. Dazu können aus den einzelnen Kurven in Zusammenhang mit der Höhe der Ursprungs Shape *Surface* Objekte erstellt werden.

### 5.9.8 planeSplitY

Beim Split einer nicht geshifteten Shape in Y-Richtung bleiben die Basis-Kurven der einzelnen Flächen gleich. Nur die Höhen der Flächen müssen entsprechend der *splitSize* angepasst werden. Die *splitSize* kann über die beiden *splittingPlanes* berechnet werden indem ein Vektor vom Punkt $(0, Y_1, 0)$ bis zum Punkt $(0, Y_2, 0)$ aufgestellt wird, wobei $Y_1$ gleich dem Y Wert der ersten und $Y_2$ gleich dem Y-Wert der zweiten Ebene entspricht. Der Vektor kann somit definiert werden über

$$\vec{v} = \begin{pmatrix} 0 \\ Y_2 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ Y_1 \\ 0 \end{pmatrix}.$$

Die Subshape wird dann mittels der Basis-Kurven der Flächen der parent Shape und der entsprechenden Höhe, welche durch die Länge des Vektors $v$ bestimmt wird, erstellt. Zusätzlich wird der Scope der Subshape um den Vektor $\vec{e}$ verschoben.

$$\vec{e} = \begin{pmatrix} 0 \\ Y_1 \\ 0 \end{pmatrix}$$

Dadurch wird dafür gesorgt, dass die Subshape im globalen Koordinatensystem in der korrekten Höhe ist.

### 5.9.9 tiltedSplitY

Beim Split einer geshifteten Shape in Y-Richtung kann der Verlauf der Basis-Kurven der einzelnen Flächen beibehalten werden. Es ist jedoch gegebenenfalls nötig die komplette Kurve in X- und Z-Richtung zu verschieben. Für das Split müssen zudem neue Verschiebungsvektoren bestehend aus *extensionX*, *height* und *extensionZ* für die Flächen berechnet werden. Die Split-Operation basiert somit auf der Berechnung zweier Verschiebungsvektoren. Dem Vektor *shiftingVectorBasis*, der die Verschiebung der Basis-Kurven angibt, Sowie dem Vektor *extensionVector*, der die Belegung der Felder *extensionX*, *height* und *extensionZ* beinhaltet. Die Berechnungen basieren darauf, dass die Verschiebung für die Kontrollpunkte der Basis-Kurven gleich sind. Die Berechnungen können somit auf einem beliebigen Kontrollpunkt einer beliebigen Basis-Kurve ausgeführt werden.

Die Berechnung des Vektors *shiftingVectorBasis* erfolgt über ein Skalar, der die Split-Size ins Verhältnis zur Größe der Ursprungs-Shape setzt. Die Berechnung des Skalars ergibt sich aus:

$$\lambda = (\vec{n_1} * \vec{p_1} - \vec{n_1} - \vec{a})/\vec{n_1} * (\vec{b} - \vec{a})$$

wobei $\vec{a}$ gleich ein beliebiger Kontrollpunkt einer Basis-Kurve ist und $\vec{b}$ der Vektor $\vec{a}$ verschoben um die Werte von *extensionX*, *height* und *extensionZ* ist. Der Vektor $\vec{n_1}$ ist die Normale der ersten *splittingPlane* und $\vec{p_1}$ ist der Punkt auf der Ebene der ersten *splittingPlane*. Über diesen Skalar kann mittels linearer Interpolation der Schnittpunkt zwischen der ersten *splittingPlane* und der Strecke, die von $\vec{a}$ bis $\vec{b}$ verläuft, berechnet werden. $\vec{s} = (\vec{b} - \vec{a}) * \lambda$. Der Vektor vom Ausgangspunkt $\vec{a}$ bis zum Schnittpunkt $\vec{s}$ bildet den Vektor *shiftingVectorBasis*.

Die Berechnung des Vektors *extensionVector* läuft ähnlich ab. Auch hier wird zunächst ein Skalar berechnet, um im Anschluss den Schnittpunkt mit der Ebene zu bestimmen. In diesem

Fall wird zur Berechnung die zweite *splittingPlane* genutzt, wodurch die Berechnung wie folgt durchgeführt wird:

$$\lambda = (\vec{n_2} * \vec{p_2} - \vec{n_2} - \vec{a})/\vec{n_2} * (\vec{b} - \vec{a})$$

Wobei $n_2$ und $p_2$ die Normale und der Punkt der zweiten splittingPlane sind. Die Berechnung des Schnittpunktes erfolgt über $\vec{s_2} = (\vec{b} - \vec{a}) * \lambda$. Der Vektor *extensionVector* wird dann über *shiftingVectorBasis* und den Schnittpunkt $\vec{s_2}$ erstellt. Dabei ist *shiftingVectorBasis* der Startpunkt und der Schnittpunkt $\vec{s_2}$ bestimmt den Endpunkt.

Nach der Berechnung der Vektoren wird für jede Kurve ein neues Curve Objekt erstellt, bei dem die X- und Z-Werte um den X- und Z-Anteil des *shiftingVectorBasis* Vektors verschoben sind. Aus diesen Kurven werden Spline3DShape Objekte erstellt bei denen die Felder *extensionX*, *height* und *extensionZ* entsprechend den Werten des Vektors *shiftingVectorSplit*belegt sind. Der Scope wird danach um den Y-Anteil des Vektors *shiftingVectorBasis* transformiert, um die Verschiebung der Basis-Flächen abzuschließen.

### 5.9.10 Umsetzung des Splits auf Segment3D Shapes

Im Fall der *Segment3DShapes* kann die Split-Operation der *Spline3DShape* übernommen werden. Der Unterschied besteht darin, dass in diesem Fall nur auf einer einzigen Fläche gearbeitet werden muss, anstatt über alle Flächen zu iterieren.

## 5.10 Operation: Component-Split

### 5.10.1 Umsetzung auf Spline3DShapes

Anders als beim Component-Split einer Prism Shape, können bei der Spline3D Shape nicht alle Rand-Elemente als eine Shape von niedrigerem Grad(ClosedSpline Shape) dargestellt werden. Für die Umsetzung der Component-Split Operation auf Spline3D Shapes wurde daher die Shape *Segment3D* eingeführt, die eine Repräsentation einer einzelnen Bézierfläche darstellt 5.5. Dadurch können die Bézierflächen, die die Segmente der Mantelfläche der Spline3D Shape definieren, als Shape für die Seitenflächen genutzt werden.

Für die Component-Split-Operation, werden für alle konkreten Shapes *ComponentSplit-Strategys* erstellt, welche die Methode *generateShapesFrom(Shape shape, List<String> ids, Type type)* zur Erzeugung der Randelemente bereitstellt. Der Parameter *type* gibt an, welche Ran-

delemente zurückgegeben werden sollen. Das Enum *Type* war dafür bereits gegeben und bestand aus den Konstanten *SIDE_FACES* und *TOP*. Für die Anwendung auf *Spline3DShapes* wurde das Enum um die Konstante *BOTTOM* erweitert. Die Component-Split-Operation kann dadurch, unabhängig von der lokalen Ausrichtung der Spline3DShape, stets gleich ausgeführt werden. Unter Verwendung der Konstanten *BOTTOM* wird somit stets die Grundfläche der Spline3DShape zurückgegeben und bei Verwendung von *TOP* wird stets die Deckfläche zurückgegeben. Äquivalent dazu gibt die Operation bei Verwendung der Konstanten *SIDE_FACES* stets die einzelnen Bézierflächen der *Segment3DShape* zurück. Dabei ist zu beachten, dass die Randelemente unterschiedlichen Klassen angehören. Die Randelemente die über die Konstanten *TOP* und *BOTTOM* zurückgegeben werden, sind vom Typ *ClosedSplineShape*. Während die Randelemente der Konstanten *SIDE_FACES* vom Typ *Segment3DShape* sind. Dies ist der Hauptgrund, weshalb die Operation unabhängig von der lokalen Ausrichtung sein soll. Bei Anwendung der Operation ist somit klar definiert, welche Art der Shape zurückgegeben wird. In Abbildung 5.13 ist das Ergebnis einer Component-Split-Operation dargestellt, welche die Oberseite und die Seitenflächen beinhaltet.

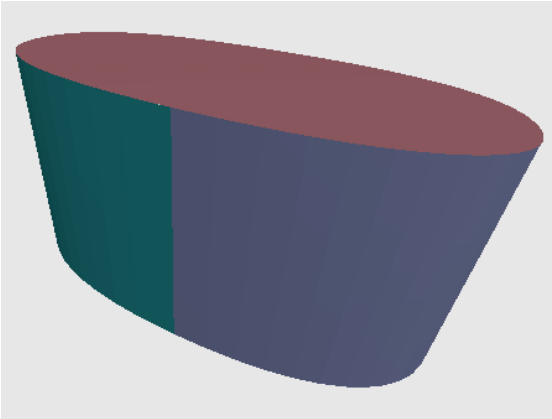


Abbildung 5.13: Component-Split Types = top, side_faces

### 5.10.2 Erzeugung der Randelemente

Die Erzeugung der Randelemente kann in drei Fälle aufgeteilt werden. In die Erzeugung der *side_faces*, die *top* und die *bottom*. Die *side_faces* werden erzeugt, indem iterativ die einzelnen Flächen der *Spline3DShape* in *Segment3DShapes* überführt werden. Dazu steht ein Konstruktor der Klasse *Segment3DShape* bereit, der über eine parent-Shape(*parent*), eine ID(*symbol*) und eine Surface Objekte erstellen kann. Die Unterseite, für den Fall *bottom*, wird erstellt, indem die Basiskurven der einzelnen Flächen iterativ einer Liste hinzugefügt werden und über diese

Liste ein *ClosedSplineShape* Objekt erstellt wird. Für die Erzeugung der Oberseite muss der Scope des erstellten *ClosedSplineShape* Objekts zusätzlich um extensionX in X-Richtung, height in Y-Richtung und extensionZ in Z-Richtung verschoben werden. Durchgeführt wird dies über die *translate(Vector3f)* Methode der *Shape* Klasse.

## 5.11 Mesh Generation

Für die Mesh Generation benötigen die Mesh Generatoren eine konkrete Repräsentation der Shapes in Form von Vertices. Im Falle der gekrümmten Formen beinhalten die Klassen nur die Felder, um die Kurven bzw. Flächen mathematisch zu definieren. Zudem können die Shapes auf Grundlage der Vertices nicht exakt dargestellt werden, sondern müssen approximiert werden. Dafür stellen die beiden Hilfsklassen *Curve* und *Surface* jeweils die Methode *calculateVertices()* bereit, die den Verlauf der Kurve bzw. der Fläche mittels Vertices approximiert und im Falle der Kurve als Liste und im Falle der Fläche als zweidimensionales Array zurückgibt. Aus diesen Vertices können die Mesh Generatoren für *ClosedSplineShapes*, *Spline3DShapes* und *Segment3DShapes* im Anschluss Meshes generieren. In den folgenden zwei Sections soll zunächst auf die Berechnung der Vertice Approximation der Kurven und Flächen eingegangen werden. Danach wird auf die Mesh Generatoren der einzelnen Shape Klassen für gekrümmte Shapes eingegangen.

### 5.11.1 Berechnung der Approximation einer Bézierkurve

Die Berechnung der Approximation für Bézierkurven erfolgt direkt über die Formel:

$$p(t) = \sum_{i=0}^{n} c_i * B_i^n(t)$$

über die für ein beliebiges $t = [0, 1]$ ein einzelner Punkt auf der Kurve berechnet wird. Die Basisfunktion $B_i^n(t)$ bilden die Bernsteinpolynome, berechnet über:

$$B_i^n(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}, t \in [0, 1].$$

Der konkrete Ablauf der Methode besteht darin, mit einer vordefinierten Schrittweite(aktuell 0.075), über eine Variable $t$ beginnend bei $t = stepSize_t$ zu iterieren, bis $t >= 1$ ist. Dabei wird in jedem Iterationsschritt die Formel zur Berechnung eines Punktes auf der Kurve ausgewertet und das Ergebnis einer Liste hinzugefügt. Die Liste wird nach Abschluss der Iteration zurückgegeben. Zur Beschleunigung der Berechnung der Bernstein Polynome werden die

Fakultäten bis zum benötigten Grad bei der Erstellung des *Curve* Objektes berechnet und über das Feld *fac[]* abgespeichert. Ein möglicherweise auftretender Sonderfall ist die Berechnung der Vertices für eine Kurve vom Grad $n = 1$. In diesem Fall werden die beiden Kontrollpunkte der Kurve zurückgegeben.

### 5.11.2 Berechnung der Approximation einer Bézierfläche

Die Berechnung der Approximation für Bézierflächen erfolgt nach dem gleichen Prinzip. Die Formel zur Berechnung eines Punktes muss in diesem Fall für Flächen erweitert werden auf $p(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} c_{ij} B_i^m(u) B_j^n(v)$. Die Basisfunktion $B_i^n(t)$ bilden weiterhin die Bernsteinpolynome. Da für die Berechnung eines konkreten Punktes auf der Fläche nun die beiden Variablen $u$ und $v$ benötigt werden, muss auch die Iteration angepasst werden. Aus der einfachen wird somit eine verschachtelte Iteration. In der inneren Iteration wird die Variable $v$ jeweils um die Schrittweite *stepSize* erhöht. In der äußeren Iteration wird die Variable $u$ jeweils um *stepSize* erhöht. Innerhalb der innerern Schleife wird dabei jeweils die Formel zur Berechnung eines Punktes auf der Fläche, unter Berücksichtugung der Belegung von $u$ und $v$, ausgewertet. Die Ergebnisse der Berechnung werden in einem Array abgespeichert, sodass die innere Schleife den jeweilige Spalten-Index erhöht und die äußere Schleife den Zeilen-Index bestimmt. Die benötigten Kontrollpunkte werden dynamisch zu Beginn der Berechnung über die Methode *getControlNet()* berechnet. Für den Fall das die Fläche einen Grad von $n = 1, m = 1$ hat, werden die vier berechneten Kontrollpunkte zurückgegeben.

### 5.11.3 Mesh Generator ClosedSpline

Für die Mesh Generierung der ClosedSpline Shape werden die Approximationen der Bézierkurven in ein *ClosedPolygon* überführt. Diese Klasse war bereits implementiert und stellt ein geschlossenen Polygon anhand einer Liste von Vertices dar. Die *ClosedPolygon* Klasse stellt unter anderem auch eine *triangulate()* Methode bereit, welche das Polygon mittels des Ear-Cutting Algorithmus trianguliert. Diese Triangulierung, in Form einer Index Liste, wird im Mesh Generator verwendet, um die Dreiecke dem Mesh hinzuzufügen.

### 5.11.4 Mesh Generator Segment3D

Für die Mesh Generierung der Segment Shapes wird ausgenutzt, dass die Approximation einer Bézierfläche, die über eine feste Schrittweite berechnet wird, eine regelmäßige Punktwolke bildet. Aus dieser Punktwolke kann die Triangulation bestimmt werden, indem aus dem Rechteck, welches jeweils vier benachbarte Punkte aufstellen, je zwei Dreiecke erzeugt werden.

Dafür wird über das Approximations Array iteriert, für jeden Punkt definiert über die Indizes (n, m) werden zwei Dreicke gebildet bestehend, aus [(n, m), (n+1, m+1), (n+1, m)] und [(n, m), (n, m+1), (n+1, m+1)]. Die letzte Spalte bzw. Reihe des Arrays wird dabei übersprungen. Die Array Zugriffe werden danach in einen Index für die Vertice Liste des Mesh Generators übersetzt, damit der Generator die Dreiecke aus der Vertice Liste bilden kann. In Abbildung 5.14 soll das Vorgehen anhand einer einfachen Fläche mit mehrerern innerern Kontrollpunkten veranschaulicht werden.

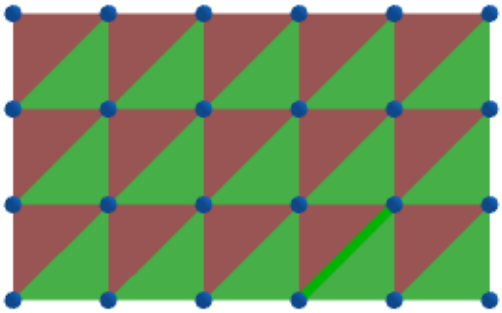


Abbildung 5.14: Vorgehen Triangulierung von Flächen

### 5.11.5 Mesh Generator Spline3D

Der Mesh Generator für *Spline3DShapes* ist eine erweiterte Form des Mesh Generators für *Segment3DShapes*. Die Grundfunktionalität des Mesh Generators für *Segment3DShapes* wird übernommen und auf die einzelnen Bézierflächen Approximationen angewendet. Die so erzeugte Triangulation stellt die Mantelfläche der Shape dar. Die so generierte Triangulation beinhaltet nur die Mantelfläche der Shape. Zur Triangulation der Grund- und Deckfläche der Shape wird aus dem Approximations Arrays jeweils die erste und die letzte Zeile extrahiert. Diese bilden die Approximation der Basiskurve bzw. der oberen Kurve und werden genutzt um über den *MeshGeneratorClosedSpline* Meshes für die Grund- und Deckfläche zu erstellen. Die Meshes der Mantelfläche, Grundfläche und Deckfläche werden danach vereinigt und bilden so das Mesh der Spline3D Shape.

# 6 Evaluation

## 6.1 Evaluation der Generierung von 2D-Shapes

Im 2D-Raum konnte die Generierung von gekrümmten organischen Shapes vollständig umgesetzt werden. Die *make_spline* Operation ist größtenteils mit den Polygon Shapes kompatibel. Eine kleine Einschränkung bilden degenerierte Polygone, die Shapes mit Schleifen erzeugen würden. Solche Shapes können vom Mesh Generator und den anderen Operationen nicht verarbeitet werden. Da es sich in diesen Fällen bei den Ausgangspolygonen schon um nicht korrekt definierte Formen handelt, ist dies nicht wirklich als Einschränkung einzuschätzen. Da die von der *make_spline* Operation erzeugten Kurven immer den Grad $n = 2$ haben, sind die Kurven stets parabelförmig. Dies schränkt die Erzeugung der möglichen Shapes geringfügig ein. Um dem entgegenzuwirken, könnten die zugrundeliegenden Polygon Shapes proaktiv detaillierter erstellt werden. Neben der Erstellung der Shape konnten auch die Basis Operationen *split* und *extrude* sowie die erweiterte Operation *advanced_extrude* vollständig umgesetzt werden. Bei der *split* Operation ist eine kleinere Einschränkung aufgefallen. Bei konkav geformten Kurvensegmenten kann es passieren, dass der Split die Enden der konkav geformten Krümmung schneidet, dadurch entsteht eine Subshape aus zwei seperaten Einzelteilen, zu sehen in Abbildung 6.1. Dies führt bei der Ableitung und vorallem bei der Mesh Generierung zu Fehlerfällen. Eine mögliche Lösung für dieses Problem, wäre die erneute Unterteilung der Shape entlang der jeweils anderen Achse, sodass beide Teile unabhängige Shapes bilden, die

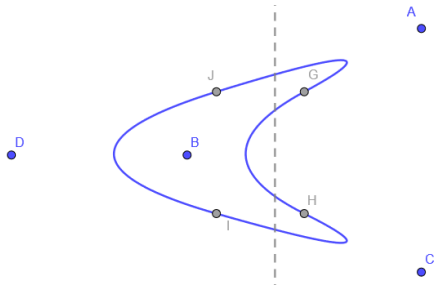



Abbildung 6.1: Split einer konkaven Shape an den Enden

sich eine successorID teilen. Für dieses Vorgehen müsste aber zunächst eine Erkennung für den Fall implementiert werden.

## 6.2 Evaluation der Generierung von 3D Shapes

Im 3D-Raum war die Generierung und Verarbeitung der Shapses mit einigen Einschränkungen verbunden, konnte jedoch trotzdem größtenteils umgesetzt werden. Eine dieser Einschränkungen betrifft die Darstellung der Shapes. Diese können in der aktuellen Implementierung zwar gekrümmte Seitenflächen sowie organische Grund- und Deckflächen darstellen, die Grund- und Deckflächen sind jedoch immer flach und können nicht gekrümmt werden. Zudem konnten die Standard Operationen nicht alle Arten der Shapes abdecken. So ist zum Beispiel eine *split* Operation nicht auf eine rotierte Shape anwendbar, was in diesem Fall hauptsächlich auf die Komplexität zurückzuführen ist. Eine solche Operation müsste folgende Schritte für die einzelnen Flächen durchlaufen. Zunächst müssten die Schnittpunkte berechnet werden, was aufgrund des möglicherweise hohen Grads der Fläche nicht trivial ist, zudem können aufgrund der Rotation mehrere Schnittpunkte vorhanden sein. Nach Berechnung der Schnittpunkte müssten die Flächen unterteilt werden. Aufgrund der Rotation würden hier möglichweise, ähnlich wie bei der konkaven Kurve, mehrere unabhängige Segmente erzeugt werden, die alle erkannt und mit den anderen Segmenten der anderen Flächen verbunden werden müssten, um die eigentliche Shape zu erstellen. Dies wäre augrund der vielen Kontrollpunkte ein großer Rechenaufwand. Die Ableitung von rotierten Formen ist daher allgemein nicht möglich. Dies fällt jedoch nicht wirklich negativ auf, da ohne weitere spezielle Operationen eine solche Verarbeitung nur schwer sinnvoll in eine Shape-Grammar integriert werden kann. Die *split* Operation ist auf auf geshifteten Spline3D Shapes nur in Y-Richtung anwendbar. Der split in X- und Z-Richtung ist nicht mit der internen Darstellung der Spline3D Shape kompatibel, mehr dazu in 6.4.

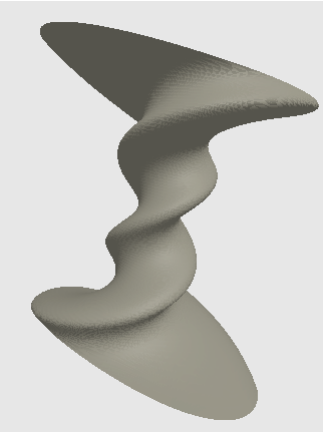
Die Component-Split Operation konnte auch mit einer leichten Abwandlung umgesetzt werden. Die Component-Split Types beziehen sich im Falle der Spline3D Shape unabhängig von der lokalen Ausrichtung immer auf die gleichen Seiten. Der Type *side_faces* bezieht sich immer auf die einzelnen Flächen der Mantelfläche, *bottom* bezieht sich immer auf den Spline, bestehend aus den Basiskurven und *top* bezieht sich immer auf die verschobenen Basiskurven. Dieses Vorgehen hat den Vorteil, dass die Component-Split Types klar definiert sind. Der Nachteil ist, dass durch dieses Vorgehen das Konzept der automatischen Ausrichtung aus [21] gebrochen wird. Dadurch wird die Erstellung der Grammatiken komplizierter, da während der

Erstellung nun Annahmen über die Ausrichtung der Shapes getroffen werden müssen. Zudem muss, bei der Erstellung der Grammatik, die Generierung von Randformen verhindert werden, die bei einem Extrude in andere Shapes hineinragen.

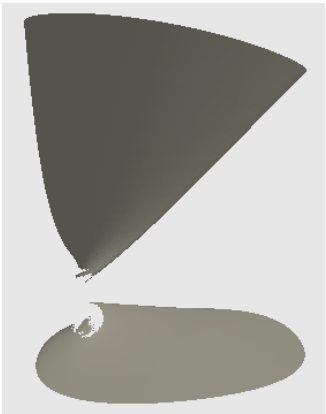
Die Extrude Operation ist auf Segment3D Shapes fehlerhaft definiert. Die dynamisch berechnete Richtung des extrudes zeigt immer in Richtung der Krümmung. Im Falle einer konkaven Fläche, wird die Shape somit in die entgegengesetzte Richtung extrudiert. Dies ist problematisch, da somit alle konkaven Flächensegmente nach einer *component_split* Operation praktisch nicht mehr weiterverarbeitet werden können und somit die allgemeinen Möglichkeiten der Shape Generierung stark eingeschränkt werden.

## 6.3 Besondere Randfälle

In dieser Section soll auf besondere Randfälle eingegangen werden, die bisher noch nicht erwähnt wurden. Einer dieser Fälle betrifft die *advanced_extrude* Operation, die bei der Angabe eines Rotationswinkels ab $585°$ degenerierte Meshes erstellt. Ein Beispiel ist in Abbildung 6.2 zu sehen. Die Abbildung 6.2a zeigt die Shape um $584°$ rotiert und die Abbildung 6.2b zeigt die degenerierte Form bei einem Winkel von $585°$. Dieses Verhalten ist für das Aufzeigen der Funktionalität nicht wirklich problematisch, da hierfür auch die niedriegeren Winkel genügen. Problematisch wird es, wenn die Operation für eine konkrete Anwendung genutzt wird, die diese hohen Winkel benötigt. Da dieser Fehler innerhalb eines $1°$ Schrittes auftritt ist zudem davon auszugehen, das es sich um einen Fehler in der Implementierung handelt.



(a) Rotation um $584°$



(b) Rotation um $585°$

Abbildung 6.2: Probleme bei der Rotation von Shapes

## 6.4 Problem der gewählten internen Darstellung der 3D-Shapes

Die gewählte interne Darstellung der dreidimensionalen organischen Shapes ist sehr auf die Standardoperationen zugeschnitten, die in Bezug auf organische Formen eher eingeschränkt sind. Dadurch, dass im Laufe des Projekt viele Annahmen bei der Implementierung getroffen wurden, ist die Erweiterung um neue Shape Operationen schwierig durchzuführen. Dies ist schon bei simplen Shape Operationen wie dem *advanced_extrude* zu beobachten. Die dafür nötige Erweiterung in Form der zusätzlichen Felder *extensionX* und *extensionZ* ist gut umsetzbar. Als Folge der neuen Shapes, definieren auch die bestehenden Operationen Formal neue Shapes. Diese Shapes sind jedoch nicht immer einfach sofort umsetzbar, wie das Beispiel der *split* Operation zeigt. Beim split einer geshifteten Shape kann der Split so gesetzt werden, dass nur ein Teil der oberen Basiskurven abgeschnitten wird. Die daraus resultierenden Flächen wären zum Teil flach und können somit nicht mit der aktuellen internen Repräsentation dargestellt werden. Ein Beispiel ist in Abbildung 6.3 zu sehen. Aber auch auf der Fläche kann der Split gesetzt werde, sodass die obere Fläche nicht aus der unteren hergeleitet werden kann. Für diese beiden Fälle müsste die interne Darstellung der Bézierflächen oder der Spline3D Shape angepasst werden.

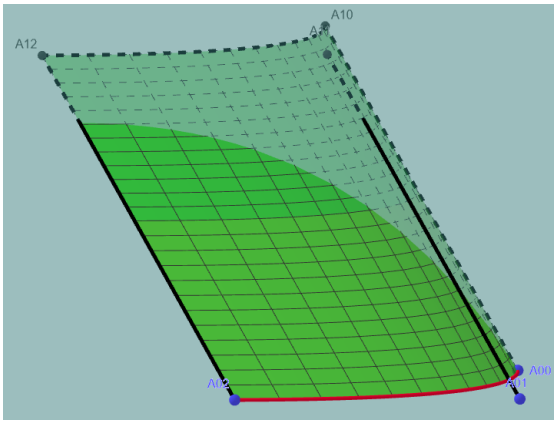



Abbildung 6.3: Visualisierung einer teils geschnittenen Fläche

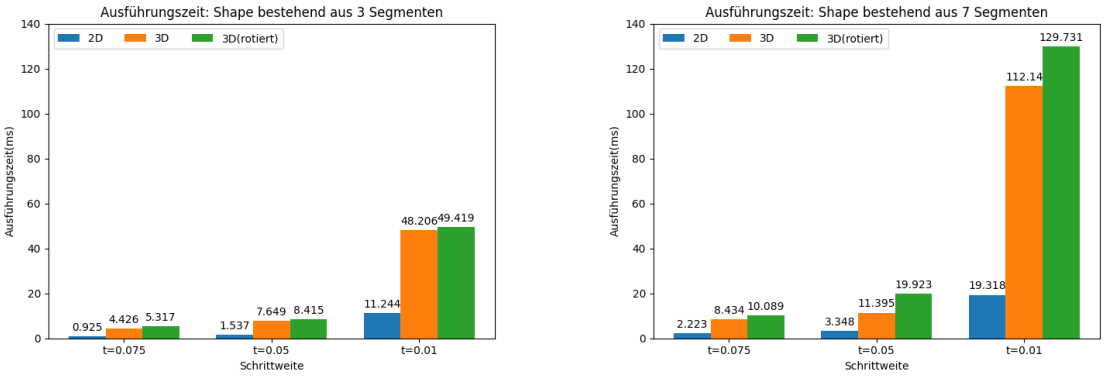
Es muss also eine Abwägung zwischen der Vielfalt der Shapes und der Umsetzbarkeit getroffen werden. Die Standard Operationen bieten nicht viel Varianz in den organischen Shapes bieten aber Einschränkungen, die die Umsetzung vereinfachen. Durch die Ergänzung von neuen Shape Operation wird der Raum an möglichen unterschiedlichen Shapes stetig

erhöht. Gleichzeitig steigt jedoch auch die Komplexität an, da die neuen Shapes, in einem gewissen Rahmen, kompatibel sein sollten. In dieser Arbeit ist die Darstellung der Shapes stark vereinfacht, es kann aber trotzdem gezeigt werden, dass eine Umsetzung einer Shape Grammar mit gekrümmten Formen, bis zu einem gewissen Grad, möglich ist.

## 6.5 Performance der Mesh Generierung

Da die Mesh Generierung aufgrund der Approximation der Kurven gegebenfalls mit vielen Vertices arbeiten muss, soll die Performance der Mesh Generierung verschiedener Shapes bestimmt werden. Dazu wird die Ausführungszeit die zur Generierung der Meshes einzelner Shapes benötigt wird, für verschiedene Schrittweiten bei der Berechnung der Approximationen, gemessen. Die Messungen wurden dabei auf einer AMD Ryzen 5 3600X CPU und einer maximalen Heap size von 2000MB vorgenommen. Die verschiedenen Schrittweiten sind hierbei $0.075, 0.05$ und $0.01$. Die Messungen teilen sich dabei in zweidimensionale Shapes und dreidimensionale Shapes auf und sollen zudem auf einer einfachen Shape und einer komplexeren Shape durchgeführt werden. Im 2D-Raum wird die Messung daher auf einer Shape bestehend aus drei Segmenten und einer Shape bestehend aus 7 Segmenten durchgeführt. Im 3D-Raum werden die genutzten Shapes aus den verwendeten 2D-Shapes erstellt. Dafür wird einmal eine Standard extrude Operation genutzt und einmal eine advanced_extrude Operation mit einer Rotation um $180°$. Die Rotation sorgt dafür, dass die einzelnen Flächen einen höheren Grad aufweisen, was die Komplexität und damit die Ausführungszeit erhöhen könnte.

Die Ergebnisse der Zeitmessung, dargestellt in 6.4, zeigen, dass die Verringerung der Schrittweite zusammen mit der Komplexität der Shape bzw. der Anzahl der Flächensegmente die Ausführungszeit am stärksten beeinflusst. Die Verringerung der Schrittweite sorgt dabei bei dreidimensionalen Shapes ungefähr für einen quadratischen Anstieg der Ausführungszeit. Bei zweidimensionalen Shapes sorgt die Verringerung der Schrittweite ungefähr für einen linearen Anstieg. Die Erhöhung des Grads der Flächen mittels Rotation führt bei einer kleinen Segment Anzahl nur zu einer geringen Erhöhung der Ausführungszeit. Bei höherer Segment Anzahl steigt die Auswirkung der Rotation, hat aber weiterhin einen eher geringen Einfluss. Im allgemeinen ist die Ausführungszeit auch bei einer Schrittweite von $t = 0.01$ in einem akzeptablen Bereich, bei deutlich komplexeren Shapes oder der Erzeugung mehrerer Shapes könnte es sinnvoll sein die Schrittweite, auf Kosten der Mesh Qualität, zu erhöhen. Eine weitere Verringerung der Schrittweite, ausgehend von $t = 0.01$, ist gerade bei eher komplexen Shapes

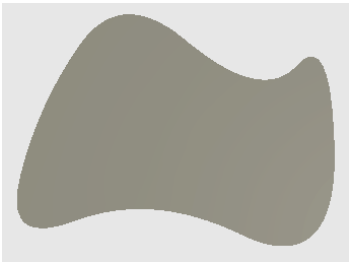



(a) (b)

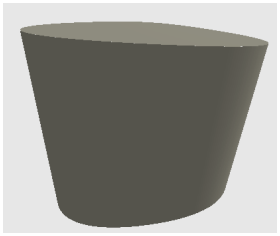
Abbildung 6.4: Ausführungszeiten der Mesh Generierung

aufgrund der quadratischen Komplexität nur in kleinen Schritten empfehlenswert. Die für die Performance Messung genutzen Shapes sind in Abbildung 6.5 aufgelistet.

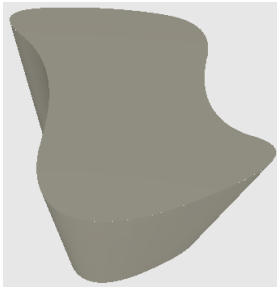
(a) ClosedSplineShape aus 3 Segmenten



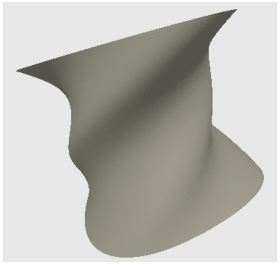
(b) ClosedSplineShape aus 7 Segmenten



(c) Spline3DShape aus 3 Segmenten

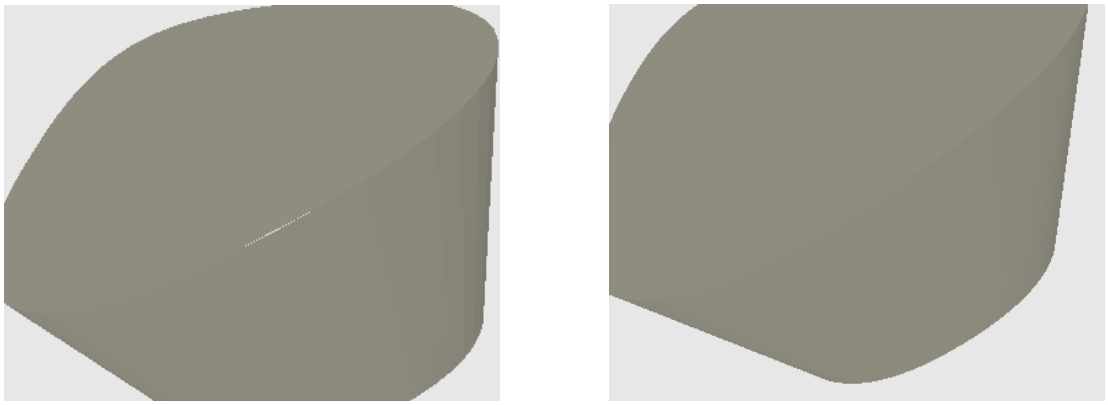


(d) Spline3DShape aus 7 Segmenten



(e) Spline3DShape aus 3 Segmenten(rotiert)

Abbildung 6.5: Für die Zeitmessung genutzte Shapes



(a) Abweichung bei $t = 0.05$ (b) Abweichung bei $t = 0.01$

Abbildung 6.6: Darstellung der Abweichung für unterschiedliche Schrittweiten

## 6.6 Evaluation der Meshes

Die erstellten Meshes weisen teilweise an den Übergängen der Seitenflächen zu der Deck- bzw. Grundfläche minimale Lücken auf. Dies wird bei Erhöhung der Schrittweite zunehmend deutlicher. Diese Lücken treten dabei vorallem an den Übergängen zwischen zwei Segmenten der Seitenfläche auf, wie in Abbildung 6.6a zu sehen. Eine mögliche Erklärung dafür ist die Berechnung der Approximation der Flächen und Kurvensegmente. Die Berechnung beginnt bei den Flächen bei $t = 0$, während sie bei den Kurven bei $t =$ stepSizeT beginnt. Die Flächen haben somit auf der XZ-Ebene einen weiteren Vertex und approximieren den Verlauf genauer. Diese Abweichung wird dabei mit der Erhöhung der Schrittweite zunehmend größer, was die erhöhte Sichtbarkeit bei größeren Schrittweiten erklären würde. Im Umkehrschluss kann die Schrittweite aber auch verringert werden, um die Abweichung entsprechend zu verringern. Bei einer Schrittweite von $t = 0.01$ ist die Abweichung so gering, dass sie kaum bzw. nicht zu erkennen ist. Ein Beispiel dafür ist in Abbildung 6.6b zu sehen.

# 7 Fazit und Ausblick

Diese Arbeit beschäftigt sich mit der Erweiterung einer CGA-Shape Grammar Implementation, um gekrümmte Formen im 2D- und 3D-Raum. Ziel der Arbeit ist es zu zeigen, dass eine Shape-Grammar auch mit nicht geradlinigen Formen arbeiten kann.

Zur Umsetzung dieses Zieles wurde die bestehende Implementation um neue Shapes und neue Shape-Operationen erweitert. Dafür wurden gekrümmte zweidimensionale Shapes auf Basis von Bézierkurven und gekrümmte dreidimensionale Shapes auf Basis von Bézierflächen implementiert. Zudem wurden die Shape Operationen *make_spline* und *adv_extrude* eingeführt.

Basierend darauf können mögliche Erweiterungen in unterschiedlichen Bereichen vorgenommen werden.

Es könnte somit, basierend auf der eigentlichen Anwendung der CGA Shape, die Implementation für die Erstellung von Architektur mit gekrümmter Form erweitert werden. Dazu könnten neue Operationen zum Einfügen von Türen und Fenstern erstelllt werden. Dabei besteht die Herausforderung der Nutzung vordefinierter Modelle, für Türen und Fenster, da aufgrund der unterschiedlichen Krümmungen kein allgemeingültiges Modell gefunden werden kann. Weiterführend können zudem neue Dach Shapes und Shape Operationen für die Verarbeitung dieser implementiert werden.

Eine weitere mögliche Weiterführung wäre die Erweiterung der möglichen darstellbaren organischen Shapes durch neue eigene Shape Operationen. Hierbei könnte der Fokus auf der Erstellung von Shapes mit gekrümmter Ober- und Unterseite liegen. Dazu könnte eine Shape Operation *make_sphere* implementiert werden, die eine sphärenartige Shape erzeugen kann, indem die einzelnen Kurven einer ClosedSplineShape in Dreicksflächen überführt werden deren Spitze der Mittelpunkt des Splines ist. Zudem könnten weitere Shape Operation zur Deformation einzelner Flächen erzeugt werden. Die Herausforderung liegt hierbei

im allgemein in der Erhaltung der Gültigkeit der Operationen auf so vielen Shapes wie möglich.

Abschließend kann festgehalten werden, dass das Ziel, zu zeigen, dass eine Shape Grammar auch mit gekrümmten Formen arbeiten kann, erfüllt wurde. Die Shape Grammar kann auf Basis der Standard-Operationen *extrude*, *split* und *component_split* vollständig mit gekrümmten Formen arbeiten. Mit der Einschränkung, dass durch diese Operationen nur 3D-Shapes mit einer flachen Ober- und Unterseite erstellt werden können. Mit der Erweiterung der Shape Operation um *adv_extrude* ist die Umsetzung mit einigen zusätzlichen Einschränkung verbunden, wie der nicht möglichen Anwendung der Split-Operation in X- oder Z-Richtung bei geshifteten Shapes, sowie der nicht möglichen Anwendung der Split-Operation auf rotierten Shapes. Abseits der genannten Einschränkungen konnten jedoch aus einer initialen Polygon Shape, als Axiom, gekrümmte Formen erstellt werden und diese entsprechend der Shape Operationen weiter abgeleitet werden. Dies zeigt, dass die Shape Grammar zumindest mit einfachen gekrümmten Formen arbeiten kann.

# Literaturverzeichnis


[1] Samuel R. Buss. *3-D computer graphics: A mathematical introduction with OpenGL*. Cambridge University Press, New York, 2003.

[2] Gerald E. Farin. *Curves and surfaces for computer aided design: A practical guide*. Computer science and scientific computing. Acad. Press, Boston, Mass. u.a., 3. ed. edition, 1993.

[3] James Gips. *Shape grammars and their uses: artificial perception, shape generation and computer aesthetics*. Springer, 1975.

[4] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden, and Alexandru Iosup. Procedural content generation for games. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 9(1):1–22, 2013.

[5] Iestyn Jowers and Christopher Earl. The construction of curved shapes. *Environment and Planning B: Planning and Design*, 37(1):42–58, 2010.

[6] Iestyn Jowers and Christopher Earl. Implementation of curved shape grammars. *Environment and Planning B: Planning and Design*, 38(4):616–635, 2011.

[7] A. Lindenmayer. Mathematical models for cellular interactions in development. i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[8] Gang Mei, John C Tipper, and Nengxiong Xu. Ear-clipping based algorithms of generating high-quality polygon triangulation. In *Proceedings of the 2012 International Conference on Information Technology and Software Engineering: Software Engineering & Digital Media Technology*, pages 979–988. Springer, 2013.

[9] Gary H Meisters. Polygons have ears. *The American Mathematical Monthly*, 82(6):648–651, 1975.

[10] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06*, New York, New York, USA, 2006. ACM Press.

[11] Joseph o'Rourke. *Computational geometry in C*. Cambridge university press, 1998.

[12] SETH ORSBORN, JONATHAN CAGAN, RICHARD PAWLICKI, and RANDALL C. SMITH. Creating cross-over vehicles: Defining and combining vehicle classes using shape grammars. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 20(3):217–246, 2006.

[13] Przemysław Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. The virtual laboratory. Springer, New York and Berlin and Heidelberg and Barcelona and Budapest and Hong Kong and London and Milan and Paris and Santa Clara and Singapore and Tokyo, first soft cover printing edition, 1996.

[14] David Salomon. *Curves and surfaces for computer graphics*. SpringerLink Bücher. Springer, New York, 2007.

[15] David Salomon. *The computer graphics manual*. Texts in computer science. Springer, London, 2011.

[16] Gillian Smith. *An Analog History of Procedural Content Generation*. 2015.

[17] G. Stiny. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[18] George Stiny. Introduction to shape and shape grammars. *Environment and planning B: planning and design*, 7(3):343–351, 1980.

[19] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. In *IFIP congress (2)*, volume 2, pages 125–135. Citeseer, 1971.

[20] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. *Procedural Content Generation: Goals, Challenges and Actionable Steps*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.

[21] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677, 2003.

[22] René Zmugg, Wolfgang Thaller, Ulrich Krispel, Johannes Edelsbrunner, Sven Havemann, and Dieter W. Fellner. Procedural architecture using deformation-aware split grammars. *The Visual Computer*, 30(9):1009–1019, 2014.

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

Hamburg, 13.06.2024 Mika Nickel