



HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Entwicklung von „SubVis“ zur Visualisierung von Unterteilungsalgorithmen

Tobias Keh, Simon Kessler, Felix Born

Konstanz, 29.02.2016

PROJEKTARBEIT

Inhaltsverzeichnis

1 Einführung	1
2 Grundlagen	3
2.1 Unterteilungsalgorithmen	3
2.1.1 Auswahl von Unterteilungsalgorithmen	4
2.1.2 Catmull-Clark	6
2.1.3 Doo-Sabin	9
2.1.4 Loop	12
2.1.5 Butterfly	14
2.1.6 Modified Butterfly	16
2.2 Rendering	18
2.2.1 Interpolating Renderer	18
2.2.2 B-Spline Renderer	19
2.2.3 Box-Spline Renderer	19
3 Tools und Bibliotheken	20
3.1 Unterteilungsalgorithmen und Datenstrukturen	20
3.1.1 OpenMesh	20
3.1.2 Surface_mesh	21
3.1.3 OpenSubdiv	23
3.1.4 CGoGN	25
3.1.5 Computational Geometry Algorithms Library (CGAL) . .	25
3.1.6 Vergleich	26
3.2 Rendering	27
3.2.1 OpenGL	27
3.3 Grafische Oberfläche	28
3.3.1 Qt	28
3.3.2 libQGLViewer	29
3.4 IDE	30
3.5 Quellcodeformatierung	30
3.6 Dokumentation	31

4 SubVis	32
4.1 Anforderungen	32
4.2 Tools und Bibliotheken	33
4.3 Entwicklungsprozess	33
4.4 Grafische Oberfläche	34
4.5 Implementierung	37
4.5.1 Gesamtarchitektur	37
4.5.2 Speicherverwaltung	38
4.5.3 Model	40
4.5.4 View	41
4.5.5 Plugin	45
4.6 Erstellung eines Plugins	45
4.7 Dokumentation	45
4.8 Buildprozess	46
4.9 Installation	47
4.10 Benutzerhandbuch	47
4.10.1 Generelle Oberfläche	47
4.10.2 Rückgängig/Wiederherstellen	47
4.10.3 Laden/Speichern	48
4.10.4 Screenshots	48
4.10.5 Triangulieren	48
4.10.6 Ansichten synchronisieren	48
4.10.7 Splitscreen umschalten	48
4.10.8 Editieren	48
5 Subdivision Plugin	50
5.1 GUI	50
5.2 Rendering	50
5.2.1 Interpolating Renderer	50
5.2.2 B-Spline Renderer	52
5.2.3 Box-Spline Renderer	52
5.2.4 None Renderer	52
5.3 Algorithmen	52
5.3.1 Vererbungshierarchie	53
5.3.2 Catmull-Clark	53
5.3.3 Doo-Sabin	54
5.3.4 Loop	54
5.3.5 Butterfly	55
5.3.6 Modified Butterfly	55

6 Projektverlauf	57
6.1 Aufgabenverteilung	57
6.2 Verlauf der beiden Semester	57

1 Einführung

Mittels eines Polygonnetzes lassen sich Flächen verschiedener Formen beschreiben. Meist wird ein sogenanntes Kontrollnetz verwendet, welches von einem Modellierer erstellt wird. Dieses wird dann durch computergestützte Verarbeitung verfeinert, um eine glatte Fläche zu erzeugen. Hierbei stellt sich die Herausforderung, mit möglichst wenigen Polygonen im Kontrollnetz eine glatte Oberfläche erzeugen zu können. Ein kleines Kontrollnetz spart Speicherplatz und senkt die Komplexität bei Änderungen im Netz. Ein in der Computergrafik häufig verwendeter Ansatz zur Berechnung von glatten Oberflächen aus einem Kontrollnetz, ist die Anwendung von Unterteilungsalgorithmen. Hier soll insbesondere der Algorithmus von Catmull-Clark erwähnt werden. Pixar verwendet diesen für die Entwicklung von animierten Filmen (siehe Abbildung 1.1).

Das Projekt *SubVis* setzt sich zum Ziel ein Programm zu entwickeln, das einige Unterteilungsalgorithmen implementiert und visualisiert. Des Weiteren sollen auf Kontrollnetze affine Abbildungen ermöglicht werden. Dabei soll einerseits das Kontrollnetz (Polygonnetz) als auch die durch Anwendung von Unterteilungsalgorithmen entstehende Limesfläche dargestellt werden. Dies dient dazu, die Vielzahl an Algorithmen an einem Ort zu bündeln und über eine schlanke, übersichtliche Anwendung zu Verfügung zu stellen. Hiervon sollen insbesondere Studenten und andere interessierte Personen profitieren, die sich mit der Thematik auseinandersetzen möchten. Der modulare Aufbau und eine gute Dokumentation sollen nachfolgenden Projekten (Abschlussarbeiten, Teamprojekte, etc.) die Möglichkeit geben, die Anwendung weiter zu entwickeln bzw. zu erweitern.

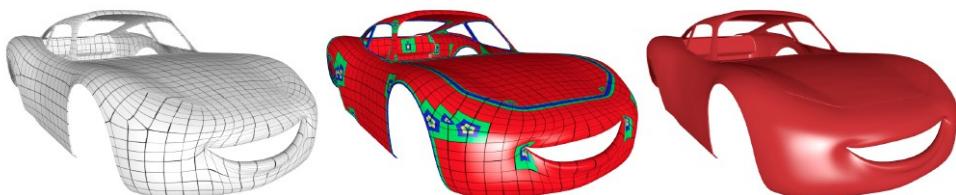


Abbildung 1.1: Anwendung von Catmull-Clark [**niessner2012feature**]

Es wird nachfolgend eine kurze Einführung in die Thematik gegeben, gefolgt von der Evaluierung verschiedener Tools und Bibliotheken. Danach folgt eine Beschreibung des Programms *SubVis*. Abschließend wird der Projektverlauf dargelegt.

2 Grundlagen

In diesem Kapitel wird eine kurze Einführung in die Grundlagen von Unterteilungsalgorithmen gegeben. Anschließend wird beschrieben, wie die erzeugten Netze gerendert werden.

2.1 Unterteilungsalgorithmen

Unterteilungsalgorithmen erzeugen aus einem Ausgangspolygonnetz eine glatte Fläche. Die glatte Zielfläche ist dabei der Grenzwert eines unendlichen, rekursiven Verfeinerungsschemas. Abbildung 2.1 visualisiert die Anwendung eines Unterteilungsalgorithmus auf eine Kurve und auf eine Fläche. Nach mehrfacher Anwendung der Unterteilung konvergiert die Kurve oder Fläche gegen die glatte Zielkurve bzw. Zielfläche.

Unterteilungsalgorithmen kann man anhand ihrer Eigenschaften kategorisieren. Ein Unterscheidungskriterium betrifft die Art und Weise, wie unterteilt wird. Man unterscheidet dabei zwischen *primal* und *dual*.

Primal Bei dieser Strategie wird die Oberfläche unterteilt („face split“). Abbildung 2.2 stellt diese Methode für ein Dreiecksnetz und ein Vierecksnetz dar.

Dual Auf der anderen Seite ist es möglich Eckpunkte in mehrere Eckpunkte

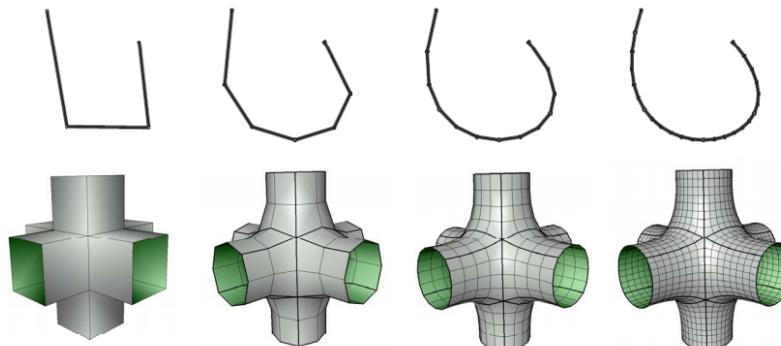


Abbildung 2.1: Unterteilungsalgorithmus - Kurve und Fläche
[Standford.24.07.2015]

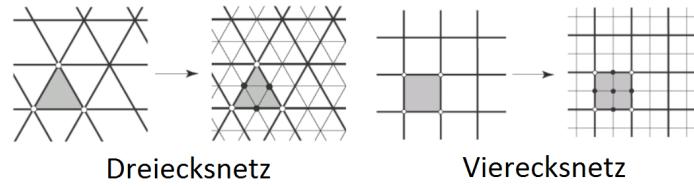


Abbildung 2.2: Primal (face split) [Standford.24.07.2015]

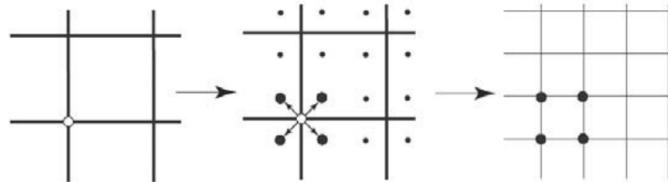


Abbildung 2.3: Dual (vertex split) [Standford.24.07.2015]

aufzusplitten („vertex split“). Diese Methode ist in Abbildung 2.3 abgebildet.

Ein weiteres wesentliches Merkmal ist, ob Kontrollpunkte interpoliert werden oder nicht.

Approximation Kontrollpunkte werden nicht interpoliert.

Interpolation Kontrollpunkte werden interpoliert.

Tabelle 2.1 und Tabelle 2.2 listet die bekanntesten Unterteilungsalgorithmen auf und ordnet diese den Kategorien zu. Zu jedem Algorithmus ist die „Glättigkeit“ der Oberfläche angegeben (C-Stetigkeit). Die Unterteilungsalgorithmen erzeugen auf beliebig angeordnete Netze C^k stetige Flächen. In den Tabelle 2.1 und Tabelle 2.2 wird die Stetigkeit für den *irregulären* und den *regulären* Fall angegeben. Die C-Stetigkeit kann auch als Maß über die Qualität des Unterteilungsalgorithmus fungieren. [Zorin.subdivcourse]

2.1.1 Auswahl von Unterteilungsalgorithmen

Für das Projekt sollen folgende Algorithmen implementiert werden:

- Catmull-Clark
- Loop
- Doo-Sabin
- Butterfly

Tabelle 2.1: Unterteilungsalgorithmen primal [Zorin.subdivcourse] [Standford.24.07.2015] [Kobbelt3] [Velho:2001:SUB:2246196.2246206] [Peters:1997:SSS:263834.263851] [Inter3Subdiv] [Sqrt5]

	Primal		
	Dreiecksnetz	Vierecksnetz	Polygonnetz
Approximation	Loop $[C^1/C^2]$ $\sqrt{3}$ -Subdivision $[C^1/C^2]$	Catmull-Clark $[C^1/C^2]$ $\sqrt{5}$ -Subdivision	Simplest $[C^1/C^1]$ 4-8 Subdivision $[C^1/C^4]$
Interpolation	Butterfly $[C^0/C^1]$ Mod. Butterfly $[C^1/C^1]$ Interp. $\sqrt{3}$ -Subdivision $[C^1/C^1]$	Kobbelt $[C^1/C^1]$	

Tabelle 2.2: Unterteilungsalgorithmen dual [Zorin.subdivcourse]

Dual
Doo-Sabin $[C^1]$
Bi-Quartic $[C^1/C^3]$

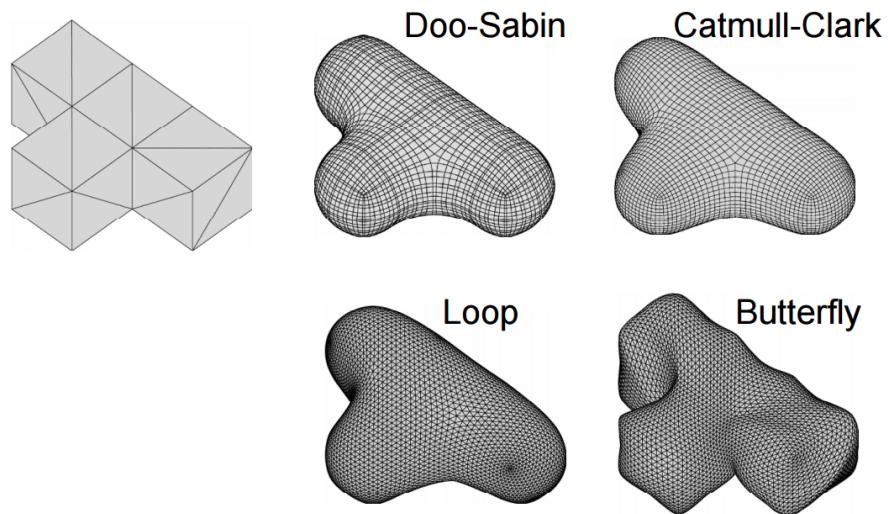


Abbildung 2.4: Vergleich der Unterteilungsalgorithmen [Standford.24.07.2015]

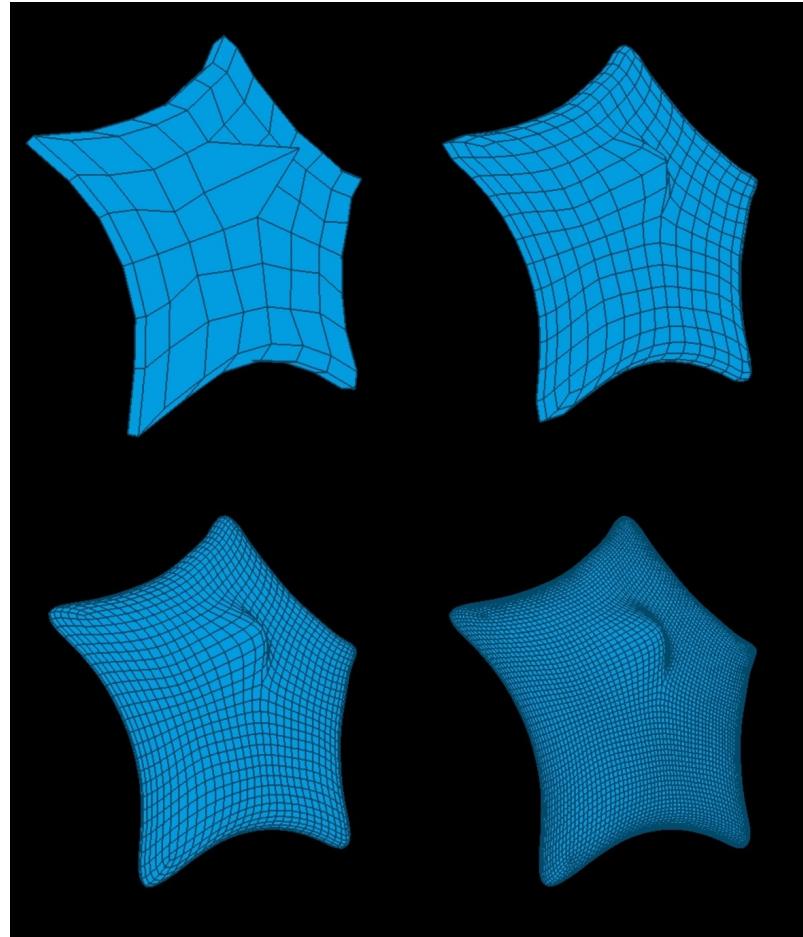


Abbildung 2.5: Drei Unterteilungsschritte mit dem Catmul-Clark Algorithmus

- Modified Butterfly

Dies sind die wichtigsten Vertreter für Dreiecks- und Vierecksnetze (siehe [[Zorin.subdivcourse](#)] [[Zorin01aunified](#)]). Prinzipiell sind jedoch noch weitere Algorithmen denkbar.

Abbildung 2.4 vergleicht die vier ausgewählten Unterteilungsalgorithmen Catmull-Clark, Loop, Doo-Sabin und Butterfly. Man erkennt deutlich den interpolierenden Unterteilungsalgorithmus (Butterfly), da dieser durch die harten Interpolationsbedingungen im Vergleich zu den approximierenden Algorithmen „welliger“ ist. [[Zorin.subdivcourse](#)]

2.1.2 Catmull-Clark

Allgemein

Der Catmull-Clark Algorithmus wurde 1978 von Edwin Catmull und James Clark entwickelt. Er basiert auf bi-kubischen uniformen B-Spline Flächen. Der

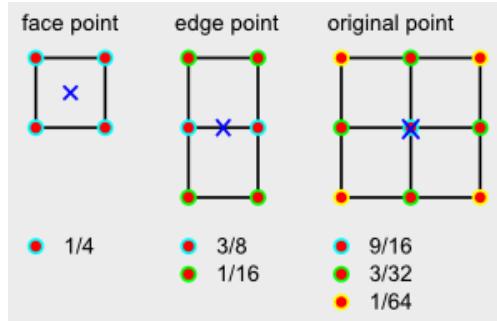


Abbildung 2.6: Catmull-Clark Masken für ein Viereck
[yoshihitoyagi.23.12.2015]

ursprüngliche Algorithmus war nur für Vierecksnetze definiert. Hier wird eine erweiterte Form verwendet, die mit beliebiger Topologie umgehen kann. Das neu erzeugte Netz ist immer ein Vierecksnetz. Jedes n-Gon im Eingabenetz wird in n Quads im Ausgabenetz umgewandelt. Die Kontrollpunkte des Netzes werden durch Unterteilung approximiert. Catmull-Clark erzeugt im Normalfall C^2 Flächen, an extraordinären Stellen mit Valenz ungleich vier jedoch nur C^1 . In Abbildung 2.5 sind 3 Unterteilungsschritte dargestellt. [Zorin.subdivcourse] [Standford.24.07.2015]

Unterteilungs- und Randregeln

Um ein Kontrollnetz nach Catmull-Clark zu unterteilen sind vier Schritte notwendig.

1. Füge für jedes Polygon einen neuen Punkt hinzu (Face Point).
2. Füge für jede Kante einen neuen Punkt hinzu (Edge Point).
3. Berechne für jeden alten Kontrollpunkt die neue Position (Original Point).
4. Verbinde die Punkte (Face Point, Edge Point, Original Point), sodass diese ein neues verfeinerten Netz erzeugen (Face Split).

Abbildung 2.6 zeigt die Masken, mit denen die jeweiligen Punkte für ein Viereck berechnet werden können. Im folgenden wird der allgemeinere Fall für beliebige Polygone beschrieben.

Face Point Der Face Point berechnet sich als Mittelwert der Kontrollpunkte des Polygons. Bei einem Viereck wie in Abbildung 2.6 hat somit jeder Kontrollpunkt ein Gewicht von 1/4.

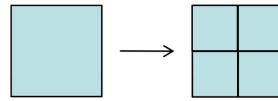


Abbildung 2.7: Catmull-Clark Face Split [[Standford.24.07.2015](#)]

Edge Point Der Edge Point wird von den anliegenden Polygonen beeinflusst. Dieser errechnet sich aus dem Durchschnitt der zwei benachbarten Face Points und den beiden Kontrollpunkten der Kante.

Original Point Jeder Original Point wird aktualisiert. Die Berechnung wird folgendermaßen durchgeführt:

$$\text{original_point} = Q/n + 2R/n + S(n - 3)/n$$

n : Valenz

Q : Durchschnitt der umliegenden Face Points

R : Durchschnitt aller umliegenden Mittelpunkte der Ecken (\neq Edge Point)

S : Original Point

Face Split Nun sind alle notwendigen Punkte berechnet. Die Punkte müssen durch Kanten zu gültigen Quads zusammengesetzt werden. Ein Quad besteht aus den vier Ecken:

- Original Point
- Edge Point 1
- Face Point
- Edge Point 2

Für jedes Polygon mit n Ecken entstehen n Quads. Abbildung 2.7 zeigt das Split-Schema für ein Viereck. [[rosettacode.23.12.2015](#)] [[rorydriscoll.23.12.2015](#)] [[yoshihitoyagi.23.12.2015](#)]

Randregeln Für Ränder (boundary cases) können die Koeffizienten für kubische Splines verwendet werden. Diese führen zu akzeptablen Ergebnissen, garantieren formal jedoch keine C^1 Stetigkeit mehr. [[Zorin.subdivcourse](#)] Abbildung 2.8 zeigt die Gewichtung für Ränder. Der Edge Point wird durch den Mittelwert der Ecken berechnet. Um den neuen Original Point zu berechnen, wird die rechte Maske aus Abbildung 2.8 verwendet.

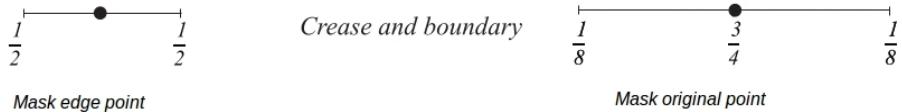


Abbildung 2.8: Catmull-Clark Randregel [Zorin.subdivcourse]

2.1.3 Doo-Sabin

Allgemein

Der Algorithmus Doo-Sabin wurde 1978 von Daniel Doo und Malcolm Sabin entwickelt. Doo-Sabin ist eine Verallgemeinerung von bi-quadratischen uniformen B-Spline Flächen und kann auf Netzen mit beliebigen Polygonen arbeiten. Das verfeinerte Ergebnis nach einem Unterteilungsschritt besteht hauptsächlich aus Vierecken, an extraordinären Stellen jedoch auch aus beliebigen Polygonen. Die Kontrollpunkte werden approximiert. Doo-Sabin erzeugt eine C^1 stetige Fläche. Drei Unterteilungsschritte sind in Abbildung 2.9 abgebildet. [Zorin.subdivcourse]

Unterteilungs- und Randregeln

Ein Unterteilungsschritt lässt sich bei Doo-Sabin in vier Punkte gliedern:

1. Berechnen der Face Points für jedes Polygon.
2. Berechnen der Edge Points für jede Kante.
3. Berechnen der neuen Vertices (New Points), die das verfeinerte Netz bilden.
4. Erzeugen der neuen Seiten (Vertex Split).

Abbildung 2.10 veranschaulicht den Zusammenhang.

Face Point Der Face Point wird analog zu Catmull-Clark als Mittelwert der Kontrollpunkte des Polygons berechnet.

Edge Point Der Edge Point errechnet sich aus dem Mittelwert der beiden Ecken einer Kante.

New Point Der New Point ist der Mittelwert aus Face Point, den beiden Edge Points und dem alten Vertex (siehe Abbildung 2.10). Das unterteilte Netz besteht nur noch aus diesen New Points.

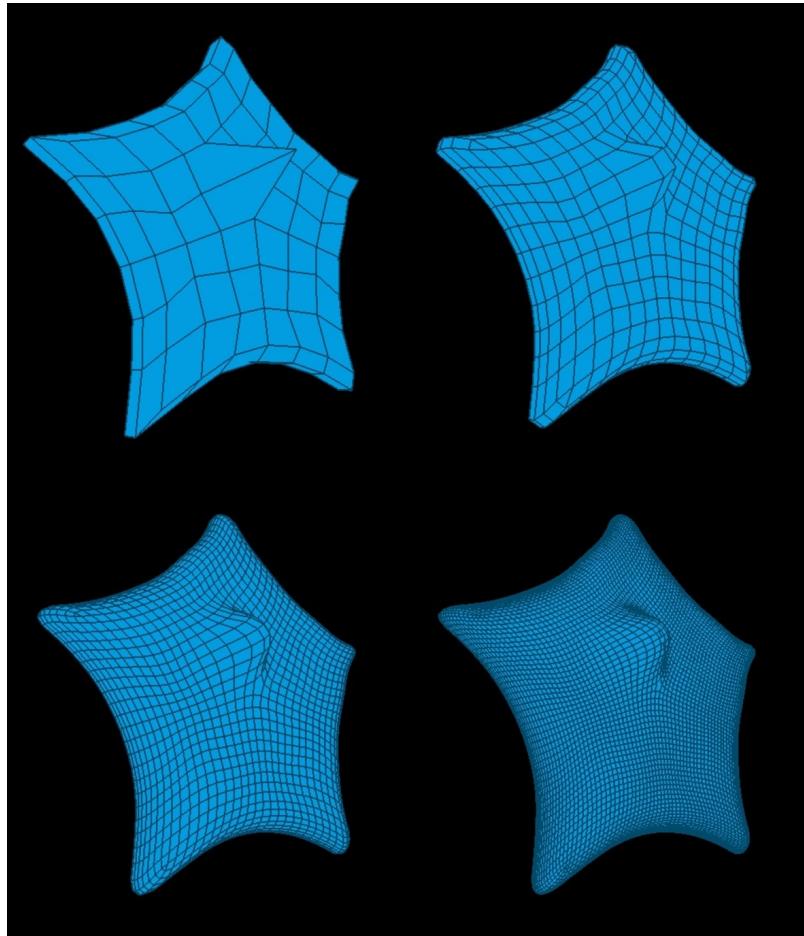


Abbildung 2.9: Drei Unterteilungsschritte mit dem Doo-Sabin Algorithmus

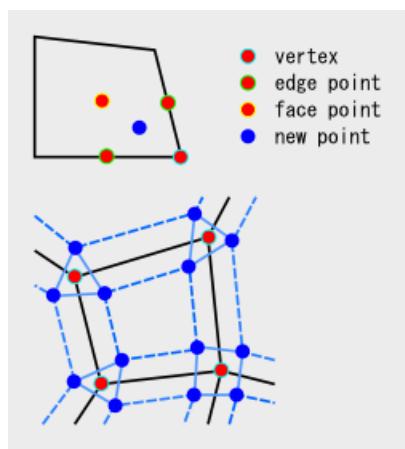


Abbildung 2.10: Doo-Sabin Unterteilungsregel [[Yoshihitoyagi.doosabin](#)]

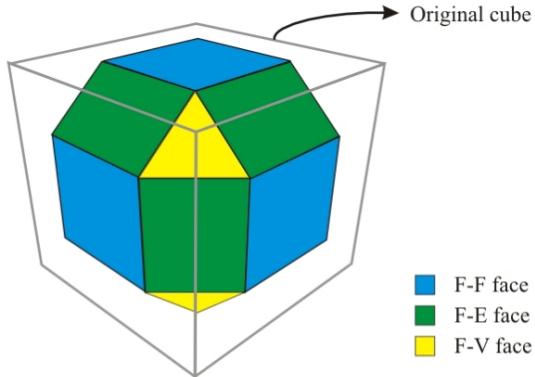
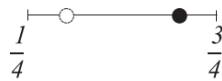


Abbildung 2.11: Die drei unterschiedlichen Seitentypen, die durch Doo-Sabin erzeugt werden [Olsen].



Mask for boundary vertices

Abbildung 2.12: Doo-Sabin Randregel [Zorin.subdivcourse]

Vertex Split Im Gegensatz zu den bisher vorgestellten Algorithmen handelt es sich bei Doo-Sabin nicht um einen Unterteilungsalgorithmus mit Face Split, sondern mit Vertex Split. Das bedeutet, dass nicht eine Seite in mehrere Seiten aufgeteilt wird, sondern jeder Vertex in mehrere Vertices aufgespalten wird. Dies ist in Abbildung 2.10 gut erkennbar. Somit entstehen bei einer Unterteilung mit Doo-Sabin drei unterschiedliche Seitentypen, die in Abbildung 2.11 markiert sind.

Face-Face Für jede Seite entsteht eine neue Seite. Diese erzeugte Seite kann ein beliebiges Vieleck sein.

Vertex-Face Für jeden Vertex wird eine neue Seite erstellt. Auch diese kann ein beliebiges Vieleck sein.

Edge-Face Für jede Kante entsteht immer ein Viereck.

[Yoshihitoyagi.doosabin] [UniCalifornia] [Olsen]

Randregeln Die Randregel ist in Abbildung 2.12 abgebildet. Es wird jede Randkante durch zwei Vertices ersetzt. Diese Maske erzeugt am Rand einen quadratischen Spline. [Zorin.subdivcourse]

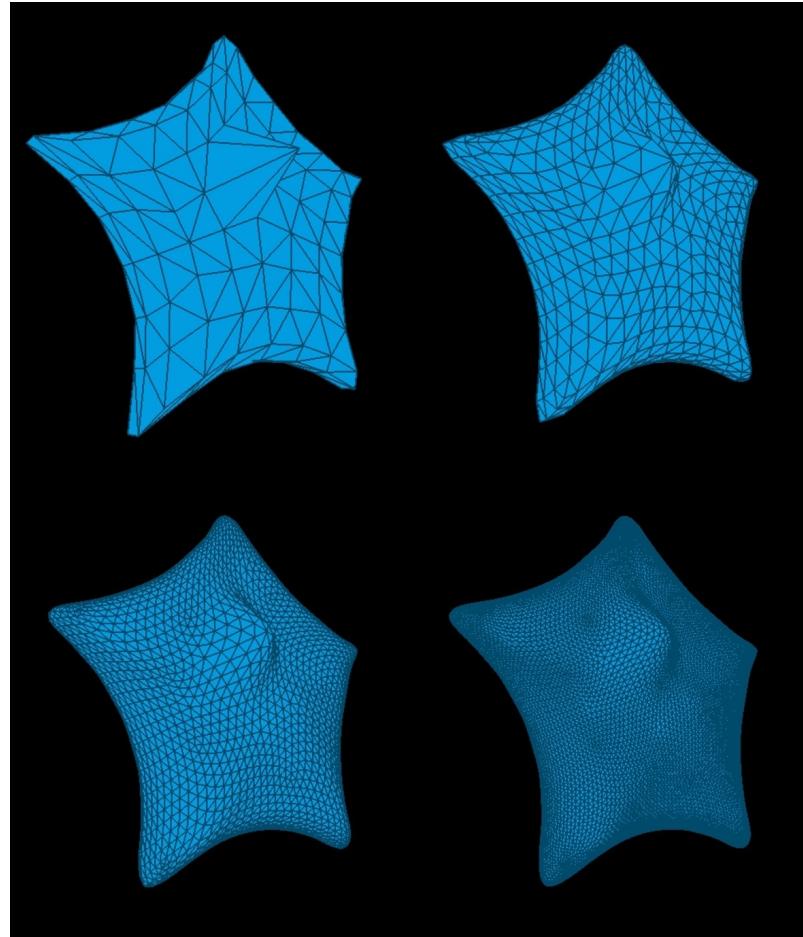


Abbildung 2.13: Drei Unterteilungsschritte mit dem Loop Algorithmus

2.1.4 Loop

Allgemein

Charles Loop hat 1987 einen Unterteilungsalgorithmus für Dreiecksnetze entwickelt. Der Loop Algorithmus basiert auf quartischen Box Splines und approximiert die Kontrollpunkte. An extraordinären Stellen mit Valenz ungleich sechs erzeugt Loop C^1 stetige Flächen, im regulären Fall C^2 . Abbildung 2.13 zeigt drei Unterteilungsschritte. [Zorin.subdivcourse] [Standford.24.07.2015]

Unterteilungs- und Randregeln

Die Unterteilung erfolgt in drei Schritten.

1. Berechne für jede Kante einen Edge Point. Dieser wird auch als Odd Vertex bezeichnet.

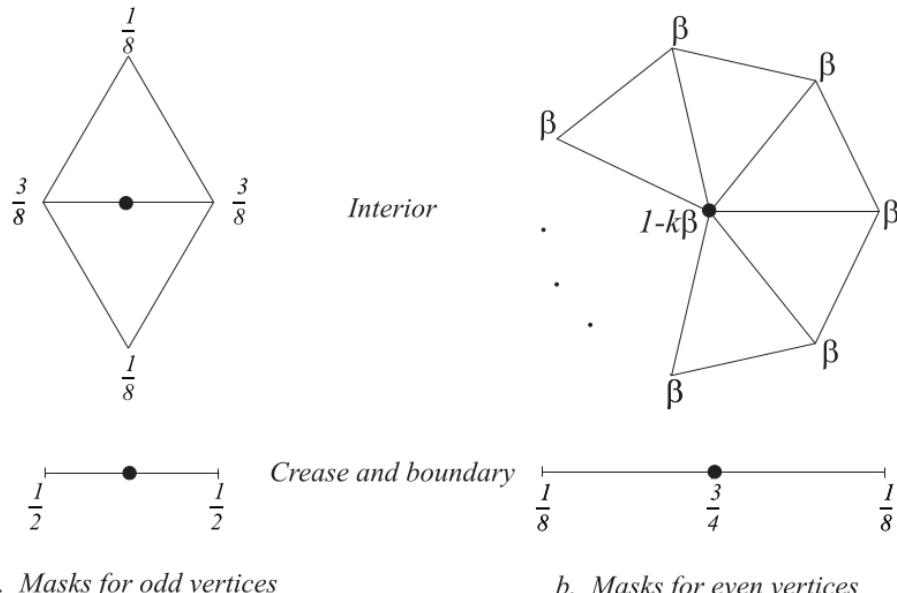


Abbildung 2.14: Loop Maske [Zorin.subdivcourse]

2. Berechne für jeden Vertex eine neue Position. Dieser wird auch als Even Vertex bezeichnet.
3. Ersetze jedes Dreieck durch vier neue Dreiecke.

Abbildung 2.14 zeigt die Masken für die Unterteilung und für Randfälle. Entlang des Randes (boundary/crease) wird eine kubische Spline Kurve erzeugt. [Zorin.subdivcourse]

Odd Vertex Die Odd Vertices werden nach der linken Maske aus Abbildung 2.14 berechnet. Für Ränder wird der Mittelwert errechnet.

Even Vertex Bei Even Vertices muss für die Berechnung der Faktor β bestimmt werden. Abbildung 2.14 veranschaulicht auf der rechten Seite die Berechnung. Der Even Vertex wird nach der Bestimmung von β wie folgt berechnet werden:

$$\text{even_vertex} = \text{old_vertex} \cdot (1 - k \cdot \beta) + \text{sum_of_surrounding_vertices} \cdot \beta$$

k : Valenz

Für die Bestimmung von β gibt es mehrere Möglichkeiten:

Original nach Loop $\beta = \frac{1}{k} \left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4} \cdot \cos\left(\frac{2\pi}{k}\right) \right)^2 \right)$

Variation nach Warren

- für $k > 3$, $\beta = \frac{3}{8k}$

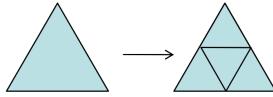


Abbildung 2.15: Loop Face Split [Standford.24.07.2015]

- für $k = 3$, $\beta = \frac{3}{16}$

Die Variation nach Warran hat den Vorteil, dass diese einfacher berechnet werden kann und somit eine performantere Implementierung möglich ist. Die Variante nach Warren ist auch in SubVis implementiert.

Für Ränder muss β nicht bestimmt werden. Hier kann das einfache Schema aus Abbildung 2.14 angewendet werden. [Carnegie] [Standford.Loop] [Zorin.subdivcourse]

Face Split Jedes Dreieck wird schließlich, wie in Abbildung 2.15 gezeigt, in vier neue Dreiecke aufgeteilt.

2.1.5 Butterfly

Allgemein

Der Butterfly Algorithmus ist ein interpolierender Unterteilungsalgorithmus, der von Nira Dyn, David Levine und John A. Gregory entwickelt wurde. Butterfly arbeitet auf Dreiecksnetzen und erzeugt an regulären Stellen C^1 stetige Flächen, an extraordinären Stellen (Valenz gleich drei oder größer sieben) jedoch lediglich C^0 . Abbildung 2.16 zeigt drei Unterteilungsschritte. [Standford.24.07.2015] [Zorin.subdivcourse] [Seeger01asub-atomic] [Gamasutra] [Sharp] [Zorin:1996:ISM:237170.23

Unterteilungs- und Randregeln

Der Algorithmus besteht aus zwei Schritten:

1. Berechne für jede Kante einen Edge Point.
2. Ersetze jedes Dreieck durch vier neue Dreiecke.

Edge Point Die Berechnung des Edge Points wird mit dem sogenannten Eight-Point Stencil aus Abbildung 2.17 durchgeführt. Die Regel für den Randfall ist daneben abgebildet.

Da die Punkte beim Butterfly interpoliert werden, müssen die alten Vertices (wie bisher bei approximierenden Algorithmen) nicht neu berechnet werden.

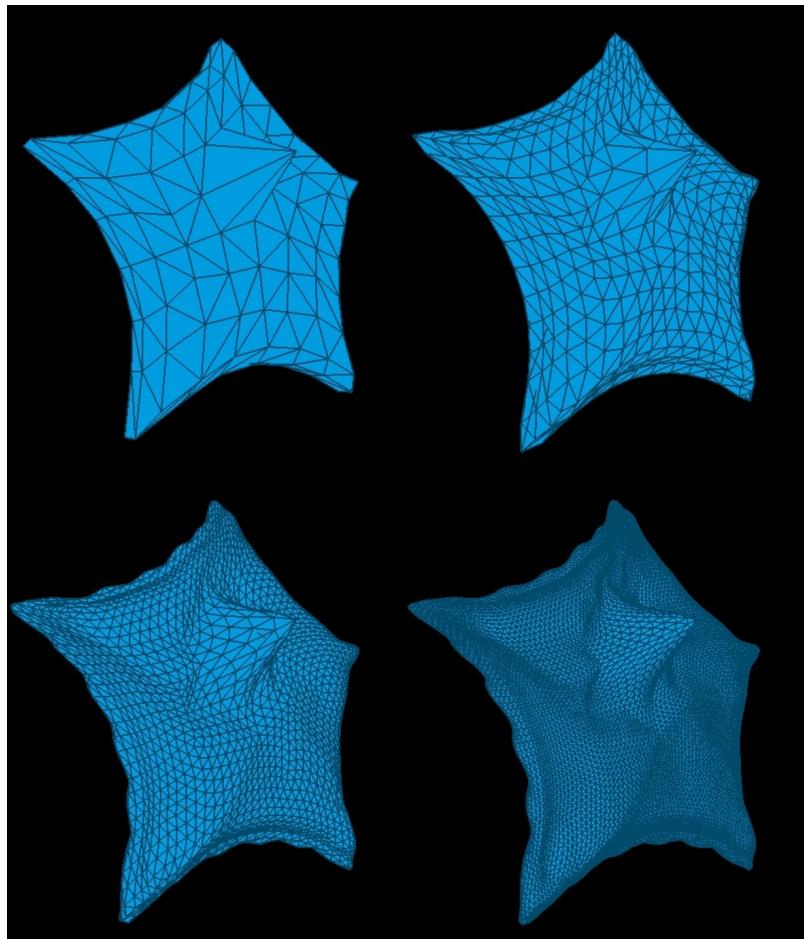


Abbildung 2.16: Drei Unterteilungsschritte mit dem Butterfly Algorithmus

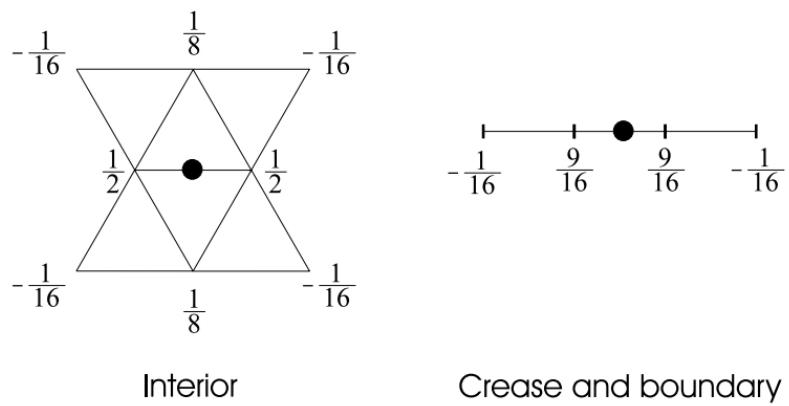


Abbildung 2.17: Butterfly Eight-Point Stencil und Randregel
[Seeger01asub-atomic]

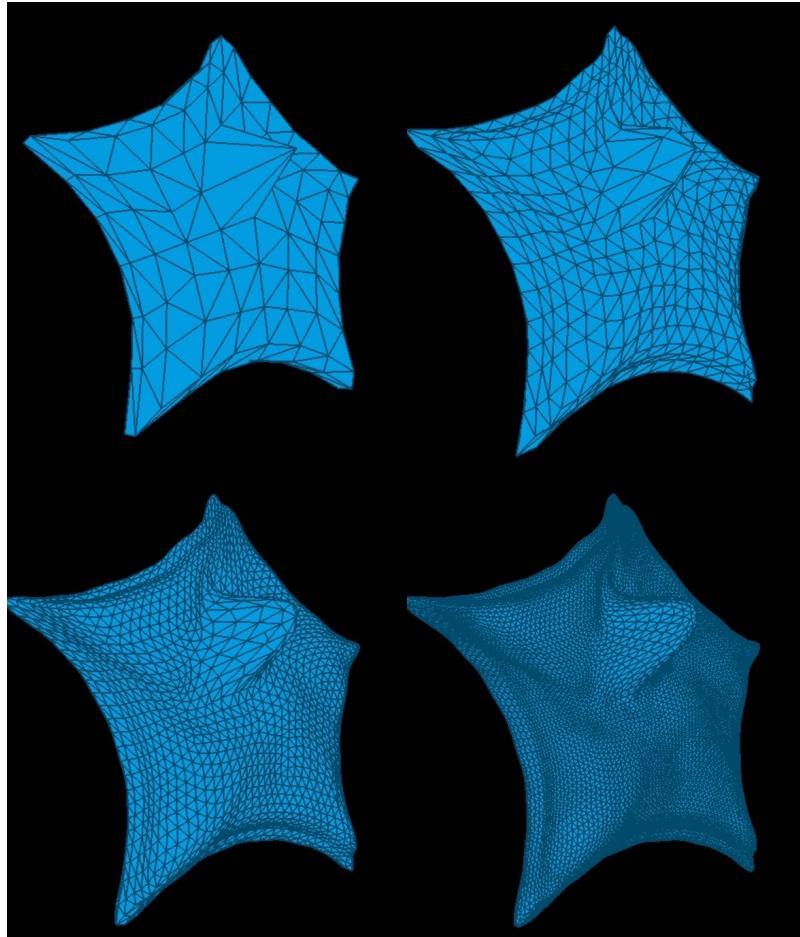


Abbildung 2.18: Drei Unterteilungsschritte mit dem Modified Butterfly Algorithmus

Face Split Der Face Split ist identisch zum Loop Face Split. Jedes Dreieck wird in vier neue Dreiecke aufgeteilt (Abbildung 2.15). [Standford.24.07.2015] [Zorin.subdivcourse] [Seeger01asub-atomic] [Gamasutra] [Sharp] [Zorin:1996:ISM:237170.237171]

2.1.6 Modified Butterfly

Allgemein

Der Modified Butterfly ist eine Erweiterung des klassischen Butterfly Algorithmus und wurde von Denis Zorin, Peter Schröder und Wim Sweldens entwickelt. Der Unterteilungsalgorithmus garantiert mit der Modifikation C^1 stetige Flächen für beliebige Dreiecksnetze. In Abbildung 2.18 sind drei Unterteilungsschritte dargestellt. [Zorin.subdivcourse] [Gamasutra] [Sharp]

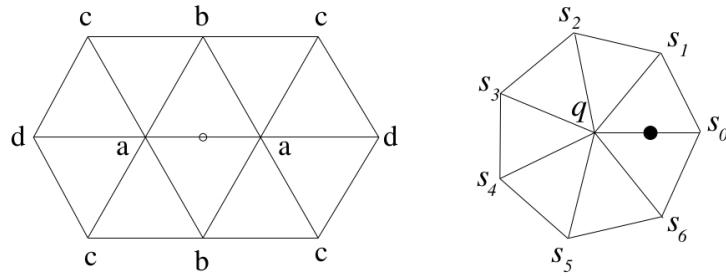


Abbildung 2.19: Modified Butterfly Ten-Point Stencil und Maske für einen extraordinären Vertex [Zorin:1996:ISM:237170.237254]

Unterteilungs- und Randregeln

Der Modified Butterfly kann wie auch schon der Butterfly in zwei Schritten zusammengefasst werden:

1. Berechne für jede Kante einen Edge Point.
2. Ersetze jedes Dreieck durch vier neue Dreiecke.

Der Unterschied liegt darin, dass die Edge Points beim Modified Butterfly unterschiedlich berechnet wird. Die Berechnung hängt davon ab, ob die andliegenden Vertices der Kante regulär oder extraordinär sind.

Edge Point Für die Berechnung des Edge Points werden vier Fälle unterschieden:

Kante verbindet zwei reguläre Vertices ($Valenz = 6$): In diesem Fall wird eine Erweiterung des Butterfly Schemas verwendet. Das sogenannte Ten-Point Stencil ist in Abbildung 2.19 links abgebildet.

Die Gewichte sind: $a = 1/2 - w$, $b = 1/8 + 2w$, $c = -1/16 - w$, $d = w$. w kann dabei geeignet klein gewählt werden. Zorin et al. verwenden in [Zorin:1996:ISM:237170.237254] $w = 0$. In diesem Fall ist das Ten-Point Stencil identisch mit dem original Butterfly Eight-Point Stencil.

Kante verbindet K-Vertex ($Valenz \neq 6$) und Sechs-Vertex ($Valenz = 6$): In diesem Fall wird das rechte Schema aus Abbildung 2.19 verwendet. Die Gewichte werden abhängig von der Valenz K verschieden gewählt.

$$j = 0, \dots, K-1 \text{ und } q = 3/4$$

- $K = 3$: $s_0 = 5/12$, $s_{1,2} = -1/12$
- $K = 4$: $s_0 = 3/8$, $s_2 = -1/8$, $s_{1,3} = 0$
- $K \geq 5$: $s_j = (\frac{1}{4} + \cos(\frac{2\pi j}{K}) + \frac{1}{2} \cdot \cos(\frac{4\pi j}{K})) / K$

Kante verbindet zwei extraordinäre Vertices: Es wird nach dem obigen Schema jeweils für beide Vertices ein Edge Point bestimmt und davon der Durchschnitt errechnet.

Randkante: Die Randregel ist mit den Gewichten $s_{-1} = -1/16$, $s_0 = 9/16$, $s_1 = 9/16$, $s_2 = -1/16$ identisch zur Butterfly Randregel (Abbildung 2.17).

[[Zorin:1996:ISM:237170.237254](#)] [[Zorin.subdivcourse](#)] [[Gamasutra](#)] [[Sharp](#)]

Face Split Der Face Split ist identisch mit dem Butterfly Face Split. [[Zorin:1996:ISM:237170.237254](#)] [[Zorin.subdivcourse](#)]

Erweiterte Randregeln Für den Modified Butterfly Algorithmus gibt es noch eine ganze Reihe von Randregeln. Diese treten dann ein, wenn die Maske für die Unterteilung zu groß ist (ein einzelnes Randdreieck ...). Diese Regeln und Sonderfälle werden jedoch ziemlich komplex und aufwändig in der Implementierung und sind nicht in SubVis implementiert. Daher werden diese hier nicht weiter diskutiert. Bei weiterem Interesse ist das Dokument [[Zorin.subdivcourse](#)] empfehlenswert.

2.2 Rendering

Das Rendering eines Netzes erfolgt auf zwei unterschiedliche Arten. So können entweder nur die tatsächlich vorhandene Daten (Punkte, Kanten, Flächen), oder aber durch Interpolation angereicherte Daten als Limesfläche gerendert werden. Das Rendering tatsächlich vorhandener Daten erfolgt unabhängig vom angewendeten Subdivision Algorithmus. Das Rendering der Limesfläche mittels interpolierter Daten ist hingegen abhängig vom gewählten Algorithmus. Die Limesfläche kann auf drei unterschiedliche Arten gerendert werden. Mittels Interpolation, interpretiert als B-Spline, oder interpretiert als Box-Spline.

2.2.1 Interpolating Renderer

Beim Rendering mittels Interpolation sind die Punkte des vorhandenen Netzes Teil der Limesfläche. Die Bereiche, die zwischen den vorhandenen Punkten des Netzes liegen, müssen per Interpolation bestimmt werden. Hierbei kommen Algorithmen wie die bereits beschriebenen Subdivision Algorithmen zum Einsatz, mit deren Hilfe zusätzliche Punkte erzeugt und gerendert werden können.

2.2.2 B-Spline Renderer

OpenGL Evaluators - ; Punkte als Kontrollpunkte Bezier Fläche - wie ist Bezier Fläche definiert? - Beschreibung mit Bild wie Bezier Fläche funktioniert

2.2.3 Box-Spline Renderer

Basiswechsel - von wo nach wo muss Basis gewechselt werden? - Behandlung wie Bezier - Verwendung des B-Spline Renderers

3 Tools und Bibliotheken

In diesem Kapitel werden Bibliotheken und Programme untersucht, die für das Projekt verwendet werden können.

3.1 Unterteilungsalgorithmen und Datenstrukturen

Im Bereich Unterteilungsalgorithmen gibt es viele bereits implementierte Datenstrukturen und Algorithmen. Gesucht wird eine einfache Datenstruktur, um Polygonnetze verarbeiten zu können. Diese sollte so wenig Overhead wie möglich mitbringen. Im Allgemeinen besteht solch eine Datenstruktur aus Ecken (Vertices), Kanten (Edges) und Flächen (Faces). Zusätzlich muss noch die Beziehung zwischen den Objekten abgespeichert werden.

3.1.1 OpenMesh

OpenMesh wird von der RWTH Aachen entwickelt und stellt eine mächtige Datenstruktur für Polygonnetze bereit. Es steht unter der LGPL v3 Lizenz („with exception“) und kann somit problemlos verwendet werden.

OpenMesh implementiert eine Datenstruktur für Polygonnetze. Darauf hinaus sind bereits einige Unterteilungsalgorithmen implementiert, die auf der OpenMesh Datenstruktur arbeiten können. Zum Funktionsumfang gehören folgende Algorithmen:

1. Uniform subdivision
 - Loop
 - Sqrt3
 - Modified Butterfly
 - Interpolating Sqrt3
 - Composite
 - Catmull Clark
2. Adaptive subdivision

- Adaptive Composite
3. Simple subdivision
 - Longest Edge

OpenMesh implementiert eine *Halfedge* Datenstruktur. Diese *kantenbasierende* Datenstrukturen speichern die Information über die Verbindungen zwischen Eckpunkten in den Kanten, während *flächenbasierte* Datenstrukturen die Verbindungsinformation zwischen den Eckpunkten und Nachbarn in den Flächen speichern.

Jede Kante referenziert also folgende Objekte:

- zwei Eckpunkte
- eine Fläche
- die nächsten zwei Kanten der Fläche

Halfedge bedeutet nun, dass eine Kante in zwei Halbkanten (Halfedge) aufgeteilt wird. Jede Halbkante hat nur eine Richtung. Zwei Ecken A und B sind also über zwei Halbkanten (erste Halkante von A nach B und zweite Halbkante von B nach A) miteinander verbunden. Dies bringt den Vorteil, dass man über die Kanten einer Fläche sehr einfach iterieren kann. Man muss dazu lediglich den Halbkanten folgen.

OpenFlipper

Aufbauend auf OpenMesh wurde von der RWTH Aachen zusätzlich das flexible Plugin-basierte Framework OpenFlipper entwickelt. Damit können geometrische Objekte modelliert und verarbeitet werden. Intern wird auf die Datenstruktur OpenMesh zurückgegriffen. Für die grafische Oberfläche wird QT verwendet. Mit OpenFlipper kann man über die Oberfläche Netze erstellen und die in OpenMesh implementierten Subdivision Algorithmen anwenden. Abbildung 3.1 zeigt die Benutzeroberfläche von OpenFlipper.

3.1.2 Surface_mesh

Surface_mesh [Sieger.] ist eine einfache und effiziente Datenstruktur um Polygongeometrie beschreiben zu können. Die Datenstruktur wurde als einfachere Alternative zu OpenMesh von der Bielefeld Graphics & Geometry Group entwickelt. Die Datenstruktur soll einfach zu benutzen sein und eine bessere Performance

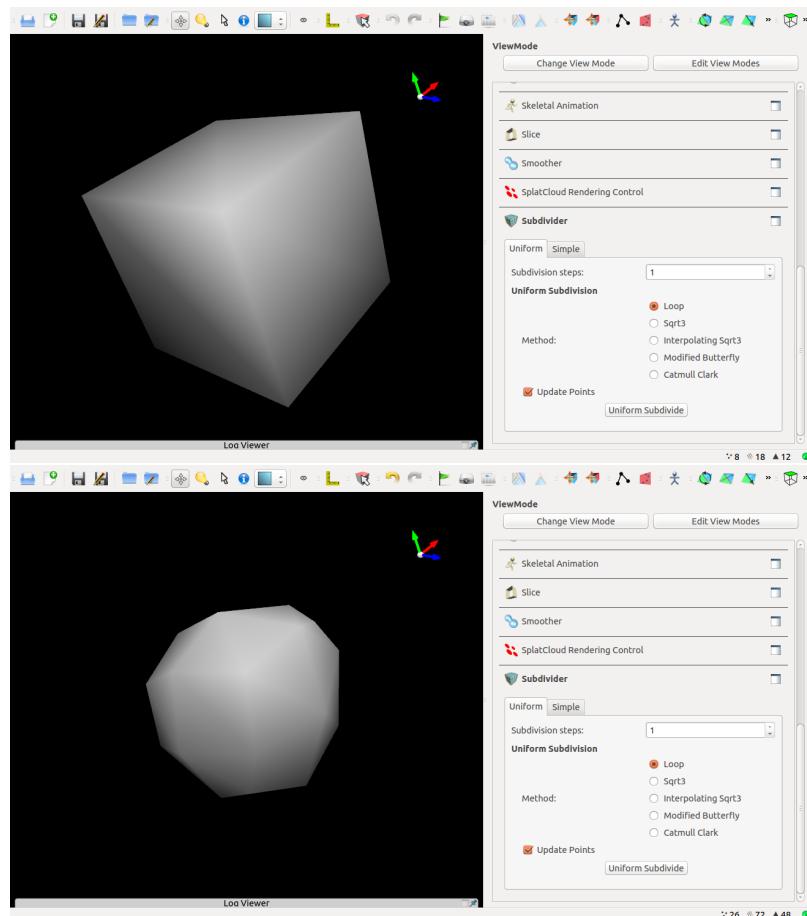


Abbildung 3.1: OpenFlipper

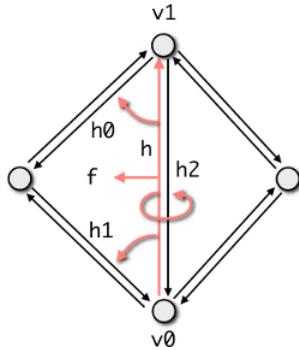


Abbildung 3.2: Surface_mesh - Halfedge Verbindungen [OpenGP.24.07.2015]

und geringeren Speicherverbrauch mitbringen. Analog zu OpenMesh implementiert Surface_mesh eine Halfedge Datenstruktur. Die Verbindungsinformation der Kanten werden also in einem Paar aus zwei gerichteten Halbkanten gespeichert. Abbildung 3.2 visualisiert den Zusammenhang der Halbkanten und Ecken.

Da Surface_mesh auch als Halfedge Datenstruktur implementiert ist, kann ähnlich effizient zu OpenMesh über die Kanten iteriert werden. Listing 3.1 zeigt einige Basisoperationen die möglich sind. Die Operationen aus Listing 3.1 sind zum besseren Verständnis in Abbildung 3.2 dargestellt.

Listing 3.1: Surface_mesh - Basisoperationen

```

1 Surface_mesh::Halfedge h;
2 Surface_mesh::Halfedge h0 = mesh.next_halfedge_handle(h);
3 Surface_mesh::Halfedge h1 = mesh.prev_halfedge_handle(h);
4 Surface_mesh::Halfedge h2 = mesh.opposite_halfedge_handle<-
    (h);
5 Surface_mesh::Face      f   = mesh.face_handle(h);
6 Surface_mesh::Vertex    v0 = mesh.from_vertex_handle(h);
7 Surface_mesh::Vertex    v1 = mesh.to_vertex_handle(h);

```

Um das Netz zu verändern oder zu editieren unterstützt die Datenstruktur high-level Operationen zum Verändern der Topologie. Mit *Edge Collapse*, *Edge Split* und *Edge Flip* kann das Netz geändert werden. Die Operationen sind in Abbildung 3.3 dargestellt.

Die Bibliothek steht unter der *GNU Library General Public License*, was eine problemlose Verwendung in diesem Projekt ermöglicht.

3.1.3 OpenSubdiv

OpenSubdiv wird von Pixar entwickelt und ist eine mächtige Bibliothek, die Unterteilungsalgorithmen und Datenstrukturen implementiert. Die Bibliothek

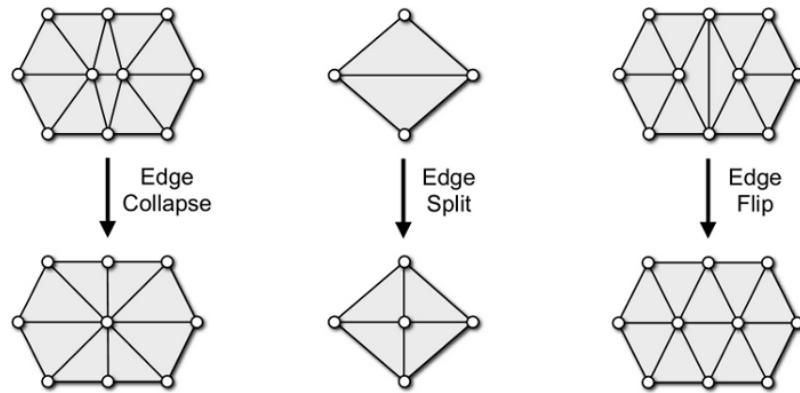


Abbildung 3.3: Surface_mesh - high-level Operationen zum Ändern der Topologie [OpenGP.24.07.2015]

ist optimiert auf Performance und unterstützt paralleles Rechnen auf CPU und GPU. Primär wird die Bibliothek von Pixar zum erstellen von animierten Filmen verwendet. OpenSubdiv ist lizenziert unter der Apache License und darf somit frei für kommerzielle und nicht kommerzielle Projekt genutzt werden.

Abbildung 3.4 zeigt den Aufbau der OpenSubdiv Bibliothek. Sie besteht insgesamt aus den vier Schichten Sdc (Subdivision Core), Vtr (Vectorized Topological Representation), Far (Feature Adaptive Representation) und Osd (Open-Subdiv) [Pixar.27.07.2015].

Sdc ist die unterste Schicht in der Architektur und implementiert die Unterteilungsdetails. Dazu gehören Typen, Optionen und Eigenschaften für die konkreten Unterteilungsalgorithmen.

Vtr beinhaltet Klassen, die das Netz für effiziente Verfeinerung in einer Zwischenrepräsentation darstellen. Diese Schicht ist nur für den internen Gebrauch gedacht.

Far ist die zentrale Schnittstelle, um Polygonnetze mit Unterteilungsalgorithmen zu verarbeiten.

Osd beinhaltet geräteabhängigen Code, um Objekte aus der Schicht Far auch in unterschiedlichen Backends wie CUDA oder OpenCL ausführbar zu machen.

Von OpenSubdiv werden die Unterteilungsalgorithmen *Catmull-Clark*, *Loop* und *Bilinear* unterstützt [Pixar.27.07.2015].

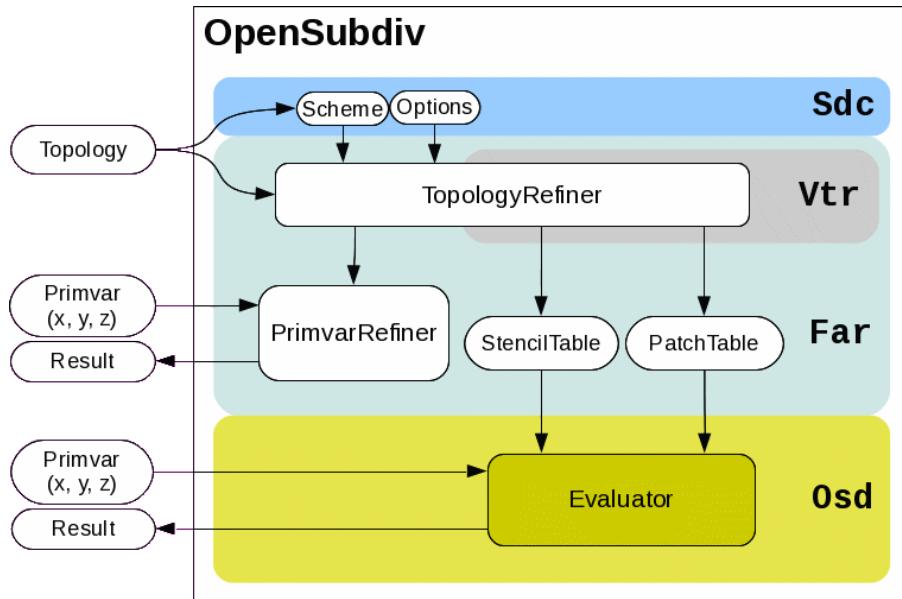


Abbildung 3.4: Pixar OpenSubdiv Architektur [Pixar.27.07.2015]

3.1.4 CGoGN

CGoGN ist eine Geometric Modeling C++ Bibliothek und implementiert eine Datenstruktur für n-dimensionale Netze als Combinatorial Maps. Diese Implementierung unterscheidet sich zu den Halfedge Datenstrukturen von OpenMesh und Surface_mesh deutlich. Diese sind zwar alle effizient, haben jedoch Probleme beim Umgang mit Objekten von unterschiedlichen Dimensionen. Für jeden Problemfall muss die spezielle Datenstruktur verwendet werden. All diese Strukturen lassen sich jedoch auf Combinatorial Maps zurückführen. Diesen allgemeineren Ansatz geht CGoGN. CGoGN implementiert bereits den Unterteilungsalgorithmus Catmull-Clark [CGoGN.27.07.2015].

3.1.5 Computational Geometry Algorithms Library (CGAL)

CGAL ist ein mächtiges Softwareprojekt mit einer Vielzahl an Datenstrukturen und Algorithmen. Neben Unterteilungsalgorithmen werden auch eine Reihe anderer Themengebiete abgedeckt (Voronoi Diagramme, Konvexe Hülle Algorithmen, Räumliche Suche, etc.). Für die Repräsentation von Netzen gibt es bei CGAL mehrere Möglichkeiten.

Surface_mesh Zum einen implementiert CGAL die bereits vorgestellte Datenstruktur Surface_mesh.

3D Polyhedral Surface Neben Surface_mesh kann auch die von CGAL entwickelte Halfedge Datenstruktur Polyhedral verwendet werden.

CGAL ist sehr mächtig und komplex. Die Bibliothek ist sogar in den meisten Paketquellen der Linux Distributionen enthalten (z. B. Ubuntu) und kann darüber sehr leicht installiert werden. Es sind auch bereits die Unterteilungsalgorithmen *Catmull-Clark*, *Doo-Sabin*, *Loop* und *Sqrt3* implementiert [CGAL.27.07.2015].

3.1.6 Vergleich

Tabelle 3.1 gibt einen Überblick über die vorgestellten Bibliotheken und listet auf, welche der ausgewählten Unterteilungsalgorithmen (*Catmull-Clark*, *Loop*, *Butterfly* und *Doo-Sabin*) bereits implementiert sind.

Tabelle 3.1: Vergleich der Unterteilungsalgorithmus Bibliotheken

Bibliothek	Datenstruktur	Unterteilungsalgorithmen
OpenMesh	Halfedge	Catmull-Clark, Loop, Butterfly
Surface_mesh	Halfedge	keine
OpenSubdiv	Halfedge	Catmull-Clark, Loop
CGoGN	combinatorial maps	Catmull-Clark
CGAL	Halfedge	Catmull-Clark, Loop, Doo-Sabin

Mesquite	1.57
CGAL_list	2.83
CGAL_vector	2.75
OpenMesh	3.37
Surface_mesh	1.13

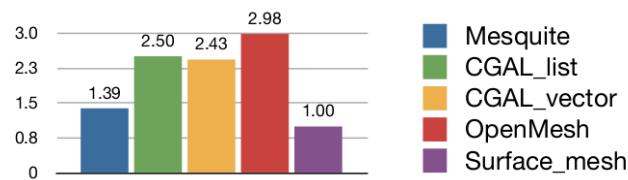


Abbildung 3.5: Benchmarks mit Surface_mesh [Sieger.]

Mit Ausnahme von Surface_mesh, das wirklich nur die Polygonnetz-Datenstruktur mit elementaren Algorithmen implementiert, sind bei den anderen Bibliotheken bereits einige Unterteilungsalgorithmen implementiert. Bei Verwendung von OpenMesh oder CGAL könnte man sich viel Arbeit sparen, da dort schon fast alle gewünschten Algorithmen implementiert sind. Ziel dieses Projektes ist jedoch eine einfache und schnelle Implementierung der Algorithmen. Surface_mesh selbst besteht nur aus wenigen Dateien und ist die schlankeste Bibliothek von allen vorgestellten. Abbildung 3.5 zeigt die Ergebnisse eines Benchmark-Vergleichs für die unterschiedlichen Datenstrukturen. In dem Diagramm werden die Zeiten relativ zu Surface_mesh angegeben.

Man erkennt deutlich, dass die Datenstrukturen von CGAL und OpenMesh vergleichsweise langsam sind. Da Surface_mesh sehr kompakt ist (es besteht nur aus sehr wenigen C++ und H-Dateien) und in dem Benchmark auch sehr gute Ergebnisse liefert, fällt die Wahl auf Surface_mesh.

3.2 Rendering

Im folgenden Unterkapitel werden Bibliotheken untersucht, mittels derer das Rendering von Polygonnetzen und Limesflächen realisiert werden kann.

Im Gegensatz zur großen Auswahl an unterschiedlichen Möglichkeiten wie beispielsweise bei den Datenstrukturen, beschränkt sich diese hier auf nur wenige Möglichkeiten. Die Wahl ist auf OpenGL wegen seinem großen Funktionsumfang und seiner weiten Verbreitung gefallen.

3.2.1 OpenGL

Bei OpenGL handelt es sich um eine sehr weit verbreitete cross-platform Bibliothek, die für 2-D und 3-D Rendering geeignet ist. Die API-Spezifikation von OpenGL beinhaltet eine Vielzahl von Befehlen, welche sowohl softwarebasiertes, als auch Hardware beschleunigtes Rendering auf der Grafikkarte ermöglichen. Das ermöglicht eine effiziente Umsetzung des Renderings von Polygonnetzen und Flächen. Das Rendering von Polygonnetzen erfolgt über eine Pipeline aus Vertex und Fragment Shadern. Diese Pipeline rendert die Ecken (Vertices), Kanten (Edges), oder Flächen (Faces), färbt und setzt die Helligkeit den Einstellungen entsprechend. Das Rendern von Flächen ist mit OpenGL mittels Evaluators und dem zugehörigen GLU NURBS Interface möglich. Evaluators erzeugen Kurven und Flächen mit einer Bezier (oder Bernstein) Basis. Zum Rendering von Kurven und Flächen anderer Basen muss die Basis entsprechend transformiert werden. Bei der Benutzung von Evaluators ist zudem zu beachten, dass die Granularität der Auswertung dem Einsatzzweck entsprechend gewählt wird. Hierbei muss ein Kompromiss zwischen einer hohen Qualität und einer kurzen Dauer der Auswertung gefunden werden. Das GLU NURBS Interface ist eine High-Level Bibliothek, die auf Evaluators aufbaut und deren Benutzung vereinfacht, indem beispielsweise häufig auftretende Kombinationen von Aufrufen zusammengefasst werden.

OpenGL bietet außerdem Optionen, um das Rendering zu optimieren. Beim Rendern des Netzes werden die Koordinaten der Ecken (Vertices) aller Flächen (Faces) an die Grafikkarte übertragen. Da die meisten Ecken Teil mehrerer Flächen sind, kommt es zu mehrfacher Übertragung von Koordinaten. Um die Menge an übertragenen Daten zu minimieren, besteht die Möglichkeit Ecken (Vertices) getrennt von den zugehörigen Koordinaten zu übertragen. Hierbei werden die Koordinaten in einer mathematischen Menge gespeichert und den Ecken (Vertices) werden jeweils Indizes zu den zugehörigen Koordinaten in der Menge zugeordnet. So kann sichergestellt werden, dass keine Duplikate von Ko-

ordinaten übertragen werden. Diese Optimierung wird als „Indizierung“ bezeichnet.

Eine weitere Option zur Optimierung ist das „Depth Testing“. Es besteht darin, das Rendering von Ecken (Vertices), Kanten (Edges) und Flächen (Faces) nur auf die für die Kamera sichtbaren Bereiche zu beschränken. Hierzu werden die Entferungen von Elementen zur Kamera vor dem Rendering verglichen. Nur die Elemente, deren Entfernung zur Kamera am geringsten sind, werden tatsächlich gerendert.

3.3 Grafische Oberfläche

In diesem Abschnitt werden die Bibliotheken, die für die Darstellung nötig sind, vorgestellt.

3.3.1 Qt

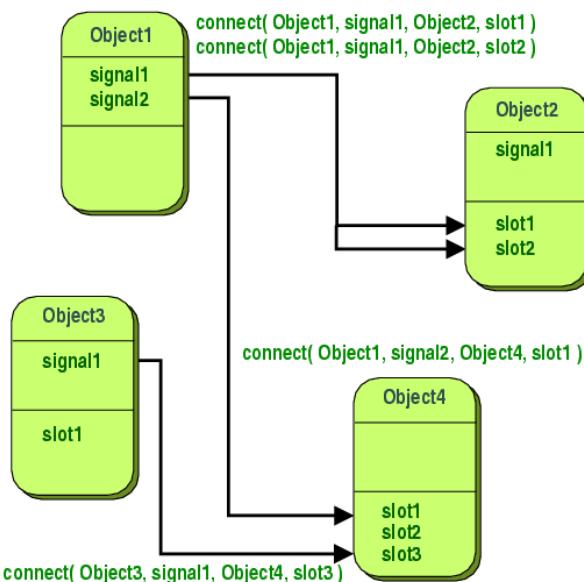


Abbildung 3.6: Signal-Slots Konzept von Qt [Qt]

Qt ist eine C++ basierte Bibliothek zur Entwicklung von grafischen Anwendungen [Qt]. Der Vorteil von Qt besteht darin, dass die Anwendungen auf den verschiedenen Betriebssystemen annähernd nativ aussehen, in dem wann immer möglich, die nativen Widgets verwendet werden. Eine Besonderheit von Qt ist das Signals-Slots Konzept (vgl. Abbildung 3.6). Dieses kann als eine Art Beobachter-Entwurfsmuster betrachtet werden. Dabei entspricht ein Signal einem *notify* und ein Slot einem *Beobachter*. Mittels einer *connect* Funktion wird die Verbindung zwischen den Komponenten (n:m) hergestellt. Zur Defini-

tion von Signalen und Slots werden Makros verwendet, die vom Meta-Object-Compiler *moc* in standardkonformen C++ Code umgewandelt werden.

Es gibt die Möglichkeit das alte String-basierte Konzept oder das neue Zeiger-basierte Konzept (ab Qt 5) zu nutzen (siehe Listing 3.2 und Listing 3.3).

Listing 3.2: Altes Signal-Slots Konzept [[QtWiki](#)]

```
1 connect(sender, SIGNAL (valueChanged(QString,QString)),
2     receiver, SLOT (updateValue(QString)));
```

Listing 3.3: Neues Signal-Slots Konzept [[QtWiki](#)]

```
1 connect(sender, &Sender::valueChanged,
2     receiver, &Receiver::updateValue);
```

Die neue Syntax ist eindeutiger, besitzt eine bessere Typprüfung zur Übersetzungszeit und erlaubt das Verbinden von Funktionen unabhängig davon, ob diese als Slot definiert wurden.

Qt wird unter der LGPL verteilt.

3.3.2 libQGLViewer

Die Bibliothek bietet einige grundlegende Funktionen zur Erstellung von 3D OpenGL Betrachtern mit C++ [[libQGLViewer](#)]. Sie bietet unter anderem folgende Funktionen bzw. erleichtert deren Erstellung:

- Kamera ist im Raum frei positionierbar
- Weltkoordinatensystem
- Verschiebung von Koordinatensystem und Objekten
- Objektselektion mit der Maus
- Screenshots
- Tastaturkürzel und Maus-Bindings

Insbesondere das Konzept des *Manipulated Frames* ist von besonderem Interesse für den Editiermodus. Es erlaubt, einen Frame dem Viewer zuzuordnen und auf diesem mit der Maus affine Abbildungen auszuführen. Über die Methoden *getPosition()* und *setPosition(..)* kann die aktuelle Position im Raum abgefragt bzw. gesetzt werden. Zu beachten ist allerdings, dass zu einem Zeitpunkt nur jeweils ein Frame pro Viewer gesetzt werden kann. Somit muss bei mehreren

beweglichen Objekten und damit Frames in der Anwendung manuell zwischen den Frames umgeschaltet werden.

Maus- und Tastaturtasten können beliebig neu belegt werden und somit vorhandene Funktionen eingeschränkt oder auf andere Tasten gelegt werden. Dies ermöglicht eine optimale Konfigurierbarkeit auf den jeweiligen Verwendungszweck.

Um einen eigenen Viewer basierend auf der Bibliothek zu realisieren, muss lediglich die Klasse *QGLViewer* erweitert werden. Durch überschreiben der Methoden *init()* und *draw()* werden die anwendungsspezifischen OpenGL-Draw-Calls umgesetzt.

Gegenüber dem OpenGL Konzept einer festen Kamera [**OpenGLCamera**] besitzt ein QGLViewer eine frei positionierbare Kamera, was intuitiver ist. Das Kamera-Objekt kann ausgelesen und modifiziert werden, um die Kamera im Raum zu verschieben.

Die Software ist unter der GPL frei verfügbar.

3.4 IDE

In Bezug auf Qt und C++ bietet sich die IDE *Qt Creator* [**QtCreator**] an. Diese bietet alle gewohnten Funktionen einer IDE wie Syntaxhervorhebung, Autovervollständigung, GUI-Designer, Debugger und Git-Integration. Insbesondere der GUI-Designer nimmt viel Arbeit ab, da so einiges an Boilerplate-Code automatisch erzeugt werden kann.

Projekte werden in *.pro* Dateien konfiguriert. Diese Datei ist plattformunabhängig und relativ schlank gehalten. Erst bei Ausführung durch qmake wird ein plattformspezifisches Makefile generiert, welches dann mittels Make ausgeführt werden kann.

In der IDE können auch verschiedene sogenannte *Beautifier* verwendet werden, welche den Quellcode formatieren.

3.5 Quellcodeformatierung

Ein weit verbreitetes Tool, um Quellcode zu formatieren ist *Artistic Style (astyle)* [**astyle**]. Insbesondere in einem Projekt mit mehreren Entwicklern ist es sinnvoll, ein solches Tool zu benutzen, um einen einheitlichen Stil bezüglich der Formatierung zu erhalten. Astyle bietet diverser vorgefertigte Stile an, erlaubt aber auch die feingranulare Definition von eigenen Stilen. Die Konfiguration erfolgt entweder über Kommandozeilenparameter oder eine Options-Datei, welche

eine bessere Übersicht bietet. Astyle lässt sich im QtCreator über ein Plugin nutzen und einem Tastaturkürzel zuweisen.

3.6 Dokumentation

Im Bereich C++ ist Doxygen [[Doxygen](#)] eine weit verbreitete Quellcodedokumentationslösung. Prinzipiell ähnelt es JavaDoc, in dem Quellcode direkt mittels spezielle annotierten Kommentaren versehen wird. Diese Annotationen werden dann mittels Transformation in ein Ausgabeformat (z. B. HTML oder PDF) überführt. Der Vorteil liegt darin, dass so die Dokumentation sehr nahe und eng mit dem Quellcode verbunden ist, was die Aktualität der Dokumentation begünstigt. Qt Creator bietet Syntaxhervorhebung für die speziellen Kommentare und erleichtert so die Verwendung.

Doxygen steht unter der GPL.

4 SubVis

Nachfolgend wird das Programm *SubVis* spezifiziert und dessen Architektur vorgestellt. Außerdem wird auf die konkrete Implementierung, Tools, Bibliotheken und Dokumentation eingegangen.

4.1 Anforderungen

- Architektur
 - Erweiterbarkeit durch andere Algorithmen und Visualisierungen mittels Plugins
 - Plugins können eigene GUI-Elemente in dafür vorgesehenen Bereichen zeichnen
 - Anwendung von verschiedenen Algorithmen auf den jeweiligen Polygonnetzen
 - Rückgängig/Wiederherstellen von Operationen
- GUI
 - Darstellung des Kontrollnetzes
 - Darstellung der Limesfläche
 - Rotation des Objektes
 - Translation des Objektes
 - Skalierung des Objektes
 - Editiermodus: Verschieben eines Vertex anhand seiner Normalen
 - Abbrechen von langlaufenden Unterteilungsschritten
 - Statistik des Polygonnetzes anzeigen
 - Splitscreen zur Darstellung von zwei Polygonnetzen
- Dateiformate / IO
 - OFF-Format und NOFF (mit Farben/Normalen)
 - Laden und Speichern von Polygonnetzen

- Unterteilungsalgorithmen
 - Catmull-Clark
 - Loop
 - Doo-Sabin
 - Butterfly
 - Modified Butterfly
- Funktionen
 - Variable Anzahl von Unterteilungsschritten
 - Beleuchtungsmodus wählbar

4.2 Tools und Bibliotheken

SubVis greift auf die Bibliotheken Qt, libQGLViewer, OpenGL Mathematics, astyle, Surface_mesh, OpenGL und Doxygen zurück. Die verwendeten Versionen sind in Tabelle 4.1 ersichtlich.

Tabelle 4.1: Versionen der Bibliotheken

Bibliothek	Version
Qt	5.4.1
libQGLViewer	2.6.1
Surface_mesh	1.0
OpenGL	10.1.3-0ubuntu0.4 ¹
Doxygen	1.8.6
GLM	libglm-dev 0.9.5.1-1
Astyle	2.03

4.3 Entwicklungsprozess

Um eine gemeinsame Entwicklungsumgebung zu schaffen, wurden gewisse Bibliotheken, Tools und Plattformen spezifiziert. Dies betrifft einerseits die Bibliotheken und Tools aus Tabelle 4.1. Andererseits auch das Betriebssystem, Versionsverwaltung, IDE und Programmiersprachenstandard.

Als Betriebssystem wird Ubuntu 14.04 LTS verwendet. Die Sprachfeatures von C++ sollen maximal dem C++11 Standard entsprechen. Als Versionsverwaltung wird Git in Verbindung mit dem Git-Server des IOS an der HTWG eingesetzt.

Die Entwicklung findet auf dem *develop*-Branch und eventuellen Feature-Branches statt. Dabei wird ein Branch pro Feature erstellt werden und erst dann in den *develop*-Branch gemerkt werden, wenn das Feature funktioniert. Als IDE wird der vorgestellte Qt Creator verwendet.

Zur Formatierung des Quellcodes wird `astyle` verwendet. Die Options-Datei `style.astylerc` definiert für das Projekt die Formatierungsregeln.

Als Style Guide für den Programmierstil wird der Google C++ Style Guide verwendet [**GsgC++**]. Abweichungen sind jedoch in begründeten Fällen erlaubt und werden dokumentiert.

Das Projekt teilt sich in zwei Verzeichnisse auf: *SubVis* und *dev-doc*. *SubVis* enthält die gleichlautende Anwendung als Qt-Projekt und unterteilt sich noch in die Ordner *app*, *lib* und *objs*. *app* enthält die Anwendungsteile die selbst entwickelt werden, *lib* die Drittherstellerbibliotheken und *objs* 3D-Modelle zum Testen. *dev-doc* dient der weiterführenden Dokumentation. Das Verzeichnis enthält z. B. Diagramme, diesen Bericht und andere hilfreiche Dokumente zur Entwicklung.

4.4 Grafische Oberfläche

Die grafische Oberfläche bietet eine aufgeräumte Ansicht der wichtigsten Elemente (siehe Abbildung 4.1). Im oberen Bereich findet sich die Menüleiste mit Dateioperatoren, Editierwerkzeugen, Optionen zur Steuerung der Ansicht und Optionen zur Steuerung des Renderings. Darunter befindet sich eine Toolbar, welche häufig verwendete Aktionen enthält. Am unteren Ende des Fensters ist eine Statusleiste integriert, welche Informationen über das Polygonnetz anzeigt (Anzahl an Knoten etc.). Der Hauptbereich der Anwendung ist die Darstellung von zwei Ansichten, welche separat bedient werden können und verschiedene Zustände der Polygonnetze darstellen können. Diese unterstützen Rotation, Zoom und Translation. Über die Tabs darüber kann zwischen der generischen Polygonnetzansicht und Plugin-spezifischen Ansichten umgeschaltet werden. Geladene Plugins werden auf der rechten Seite in Form von Tabs angezeigt. Diese besitzen eine jeweils individuelle GUI.

Die Anwendung bietet eine Rückgängig/Wiederherstellen Funktion in Form von zwei Pfeilen. Durch diese können Modifikationen an den Polygonnetzen rückgängig gemacht werden. Manchmal ist es von Vorteil, wenn die beiden getrennten Ansichten sich synchronisieren. Dies kann über die Funktion „Synchronize Views“ erreicht werden. Nach der Aktivierung bewegen sich die Kameras der beiden Ansichten absolut synchron. Eine Art „ablenkungsfreier Mo-

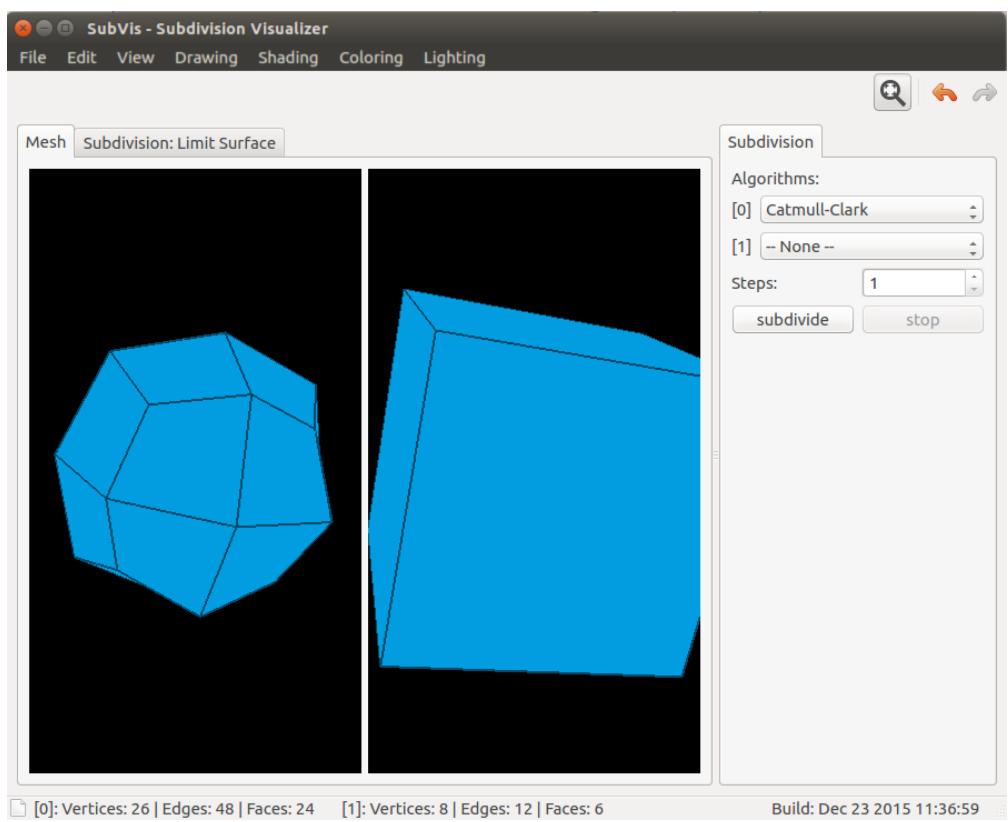


Abbildung 4.1: Kontrollnetz in SubVis

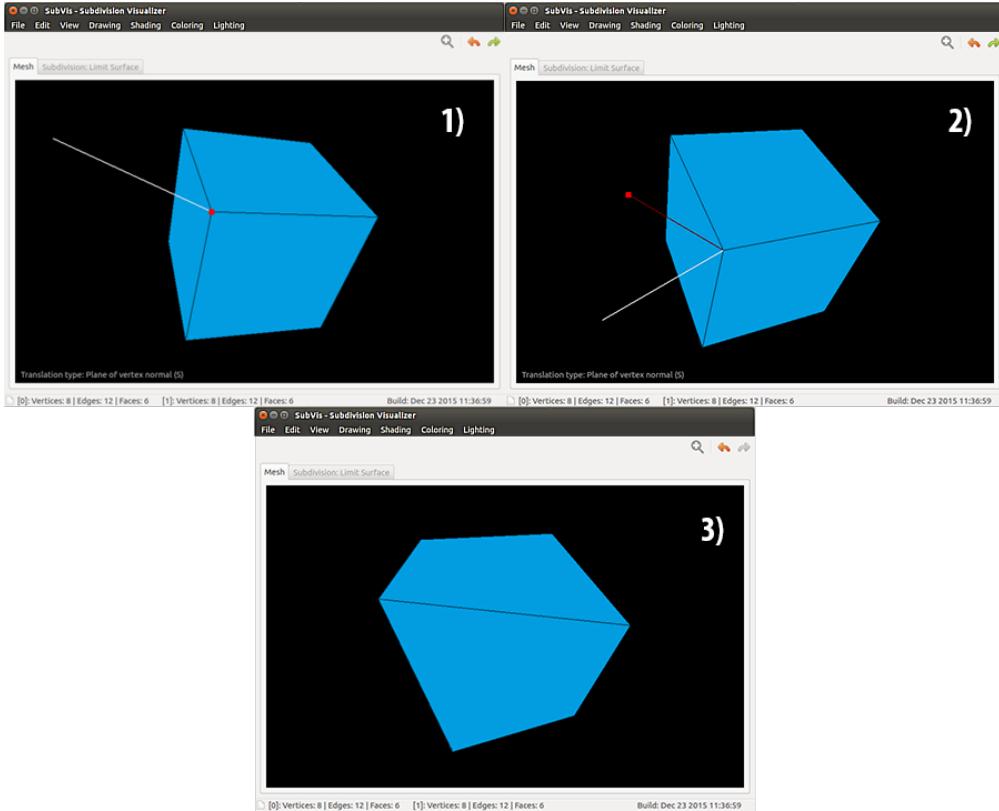


Abbildung 4.2: 1) Selektion eines Vertex 2) Translation 3) Resultierendes Polygonnetz

dus“ lässt sich erreichen, in dem die zweite, rechte Ansicht ausgeblendet wird. Zusätzlich kann die rechte Tableiste der Plugins ausgeblendet oder verkleinert werden.

Alle Menü- und Toolbar-Elemente lassen sich auch über Tastenkürzel erreichen.

Durch Aktivieren des Editiermodus (siehe Abbildung 4.2) wird in den ablenkungsfreien Modus geschaltet. Nach Doppelklick auf einen Vertex oder dessen unmittelbare Umgebung wird dieser selektiert, vergrößert in rot dargestellt und dessen Normale in weiß eingezeichnet (siehe Schritt 1 von Abbildung 4.2). Während der Vertex mittels STRG+linke Maustaste im Raum bewegt wird, wird zusätzlich eine rote Linie zwischen der Ursprungsposition und der neuen Position angezeigt (siehe Schritt 2 von Abbildung 4.2). Dabei bewegt sich der Vertex lediglich auf der Ebene, die durch die Normale des Vertex definiert wird. Damit eine beliebige Positionierung im Raum möglich ist, kann zwischen der Ebene der Normale und einer Ebene, die orthogonal zur dieser steht mit der Taste S umgeschaltet werden. Durch erneutes Doppelklicken (an einer beliebigen Stelle) wird der Editiervorgang beendet und das veränderte Polygonnetz

dargestellt (siehe Schritt 3 von Abbildung 4.2).

Die Menüpunkte „Drawing“, „Shading“, „Coloring“ und „Lighting“ bieten die Möglichkeit, das Rendering des Objekts anzupassen. Im „Drawing“ Menü kann zwischen dem Zeichnen von Ecken (Vertices), Kanten (Edges) und Flächen (Faces) gewählt werden. Der Menüpunkt „Shading“ ermöglicht es den Shading Modus zu wählen. Hierbei kann zwischen „Flat“ und „Smooth“ gewählt werden. Bei der Auswahl von „Flat“ werden alle Elemente mit den ihnen zugeordneten Farbwerten gezeichnet. Bei der Auswahl von „Smooth“ werden sowohl die den Elementen zugeordneten Farbwerte als auch die der benachbarten Elemente berücksichtigt. Dadurch ergibt sich eine Interpolation der Farbwerte in den Zwischenräumen, ein Gradient mit den jeweiligen Farbwerten als Start- und Endwert. Färbung und Beleuchtung können über „Coloring“ beziehungsweise „Lighting“ aktiviert oder deaktiviert werden. Die Färbung wird im Quellcode berechnet und kann dort entsprechend angepasst werden.

4.5 Implementierung

In diesem Kapitel wird auf die konkrete Architektur, Implementierung und detaillierte Entwicklungsentscheidungen eingegangen.

4.5.1 Gesamtarchitektur

Die grundlegende Architektur ist in Abbildung 4.3 als UML-Komponentendiagramm dargestellt. Grundsätzlich wird auf eine Model-View Architektur gesetzt.

Die Model-Komponente kapselt eine Datenstruktur, die das Polygonnetz enthält und bietet grundlegende Polygonnetzoperationen und Import- bzw. Persistenzoperationen. Sie sendet außerdem bei Modifikation eines Polygonnetzes ein Signal an alle Listener aus. Der Zugriff auf das Polygonnetz ist nur lesend möglich, jeder schreibende Zugriff muss zuerst eine Kopie erstellen und diese nach der Modifikation wieder in die Model-Komponente laden. Dadurch wird ein gewisser Grad an Immutabilität erreicht, welcher die Robustheit der Anwendung erhöht.

Die View-Komponente ist für die Darstellung/Editieroperationen verantwortlich. Diese verbindet auch die Signale der anderen Komponenten mit den Slots der Plugins bzw. Model-Komponente. Zusammengesetzt aus den Komponenten *tabs_plugins* (GUI-Elemente der einzelnen Plugins), *toolbar* (Werkzeugleiste), *tabs_viewer* (generische Polygonnetz-Ansicht und Plugin-spezifische Ansicht) und *menu_bar* (Menüleiste) bildet die View-Komponente alle grundlegenden Darstellungsoperationen ab. Die Komponente *tabs_viewer* enthält einerseits ei-

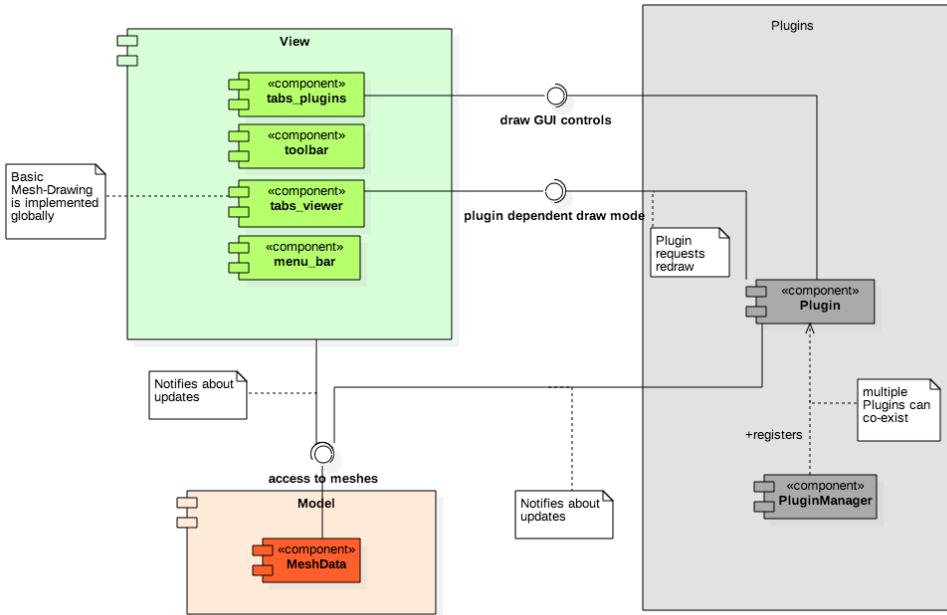


Abbildung 4.3: Gesamtarchitektur von SubVis

eine generische Ansicht, die das Polygonnetz rendert und einen Editiermodus bereitstellt. Andererseits eine Ansicht, dessen Darstellung durch das aktive Plugin gesteuert wird. Dies erlaubt den Plugins maximale Freiheit bezüglich der Darstellung, da so das Polygonnetz gerendert werden kann aber auch eine völlig unabhängige Darstellung erfolgen kann. Durch Verbindung mit den Signalen der Model-Komponente, wird auf Veränderungen im Polygonnetz reagiert.

Plugins werden von einem *PluginManager* verwaltet, bei welchem die Plugins zuvor registriert werden müssen. Dieser erlaubt die Verwendung von mehreren Plugins, wovon jedoch nur eines zu einem Zeitpunkt aktiv sein kann (durch Auswahl des jeweiligen Tabs in der GUI). Plugins werden auch über Veränderungen in der Model-Komponente benachrichtigt. Sie können auch ein Signal aussenden, um anzugeben, dass sie ein erneutes Rendering durchführen möchten. Dieses Signal wird von der View-Komponente erkannt und in einen Aufruf der draw-Methode des Plugins umgesetzt. Die Plugins erhalten die Möglichkeit eine eigene GUI zu erstellen und ein spezifisches Rendering durchzuführen.

4.5.2 Speicherverwaltung

In C++ Programmen muss eine Auseinandersetzung mit den Themen Speicherverwaltung und Ownership erfolgen. Der C++11 Standard bietet die Möglichkeit mit den Typen *unique_ptr* und *shared_ptr* die Speicherverwaltung automatisch

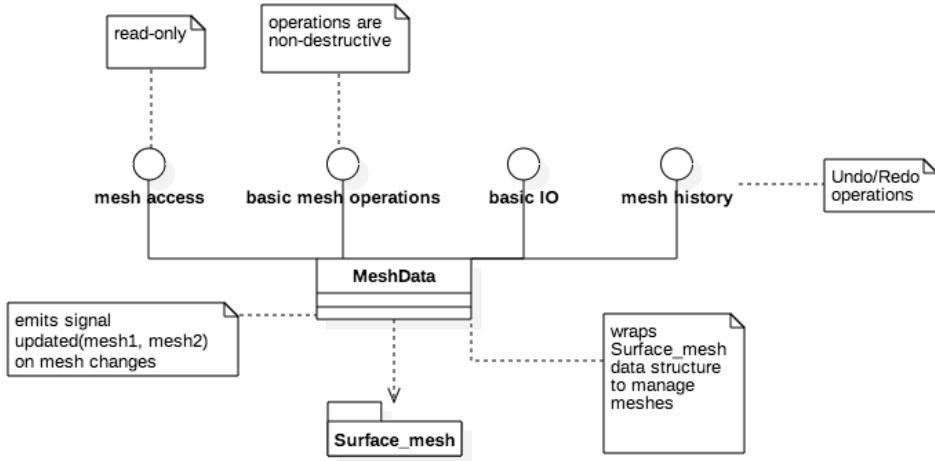


Abbildung 4.4: Architektur der Model-Komponente von SubVis

und explizit zu machen. Im Gegensatz zu einer manuellen Speicherverwaltung werden somit mehrfache oder fehlende *free* Aufrufe verhindert, was die Stabilität des Systems erhöht. Der Typ `unique_ptr` kapselt einen Zeiger und gibt dessen Speicher frei, sobald die Variable ihren Gültigkeitsbereich verlässt oder neu zugewiesen wird [C++Ref]. Somit wird das Konzept des eindeutigen Ownerships einer Ressource umgesetzt. Dahingegen erlaubt der Typ `shared_ptr` mehrere Besitzer und gibt den Speicher der Ressource erst frei, sobald die Variable durch keinen Besitzer mehr erreichbar ist. Dadurch kann der Ownership auf mehrere Instanzen verteilt werden, die alle gleichberechtigte Besitzer sind.

SubVis benutzt ausschließlich diese beiden Typen zur Speicherverwaltung. Ausnahmen sind lediglich Container-Typen, die eine ähnliche Semantik haben und Ressourcen die von Qt verwaltet werden. Die von Qt verwalteten Ressourcen werden automatisch bei Beendigung des Programms freigegeben, da diese eine Parent-Child-Beziehung besitzen und mit Zerstörung des Hauptfensters von Qt entsprechend behandelt werden. Die durchgehende Verwendung von `unique_ptr` und der Move-Semantik in C++11 macht jederzeit deutlich, welches Objekt Besitzer einer Ressource ist. Zusätzlich wird wo immer möglich auf Referenzen zurückgegriffen. Diese zeigen an, dass kein Ownership-Wechsel stattfindet und lediglich ein vorübergehender Zugriff auf die Variable gewährt wird. Referenzen haben gegenüber Zeigern den Vorteil, dass sie stets gültig sind und nie `NULL` sein können.

4.5.3 Model

Abbildung 4.4 zeigt die Klassen der Model-Komponente. Die Schnittstelle ist in der Abbildung in mehrere Teilschnittstellen aufgegliedert, um die Komponenten verständlicher darzustellen. Als Datenstruktur zur Verwaltung der Polygonnetze wird Surface_mesh verwendet. Diese wird auch als Ein- und Ausgabetyp der Model-Komponente benutzt.

Grundsätzlich verwaltet die Komponente stets zwei Instanzen der Datenstruktur. Diese werden über die Nummerierung 0 und 1 (als *Index* bezeichnet) identifiziert. Jede Anfrage nach der Datenstruktur wird mit einem Paar (`std::pair`) dieser beiden Instanzen beantwortet. Dabei handelt es sich um einen lediglich lesenden Zugriff (`const`). Diese Entscheidung wurde zugunsten eines besser nachvollziehbaren Programmflusses getroffen, da Immutabilität Software einfacher und robuster macht. Wenn ein Polygonnetz bzw. eine Kopie davon verändert wurde, dann kann dieses in die Model-Komponente mit der IO-Schnittstelle geladen werden. Dies muss immer als Paar erfolgen. Jedoch wird aus Komfort eine Methode angeboten, eine einzelne Instanz zu laden, wobei diese einfach dupliziert wird. Des Weiteren können Dateien im Format obj, off und stl geladen bzw. im Format off gespeichert werden.

Über Veränderungen der Datenstruktur werden andere Komponente informiert, in dem das Signal *updated* ausgesendet wird. Diese enthält als Parameter beide aktuellen Datenstrukturinstanzen.

Die Komponente realisiert darüber hinaus eine Historie der Instanzen. Es wird eine festgelegte Anzahl an Paaren von Instanzen im Hauptspeicher gehalten. Mit den entsprechenden Methoden kann in dieser Historie beliebig vor und zurück gesprungen werden. Dies ermöglicht die einfache Realisierung einer Rückgängig/Wiederherstellen Funktion. Die Implementierung nutzt einen `std::vector` und einen Index, um in diesem Array vor und zurück zu springen. Bei jedem neuen Historieeintrag wird der Speicher der Datenstrukturen optimiert durch Aufruf einer Garbage-Collection Funktion. Beim Hinzufügen von neuen Paaren in die Historie können sich drei Fälle ergeben:

- a) Der Index zeigt auf ein Element vor der neuesten Historie (zuvor wurde zurück gesprungen).
- b) Die Historie hat die maximale Kapazität erreicht.
- c) Der Index zeigt auf das aktuellste Element und die Historie hat noch Kapazität.

Im Fall c) ist nichts zu machen, außer das Paar hinzuzufügen und den Index

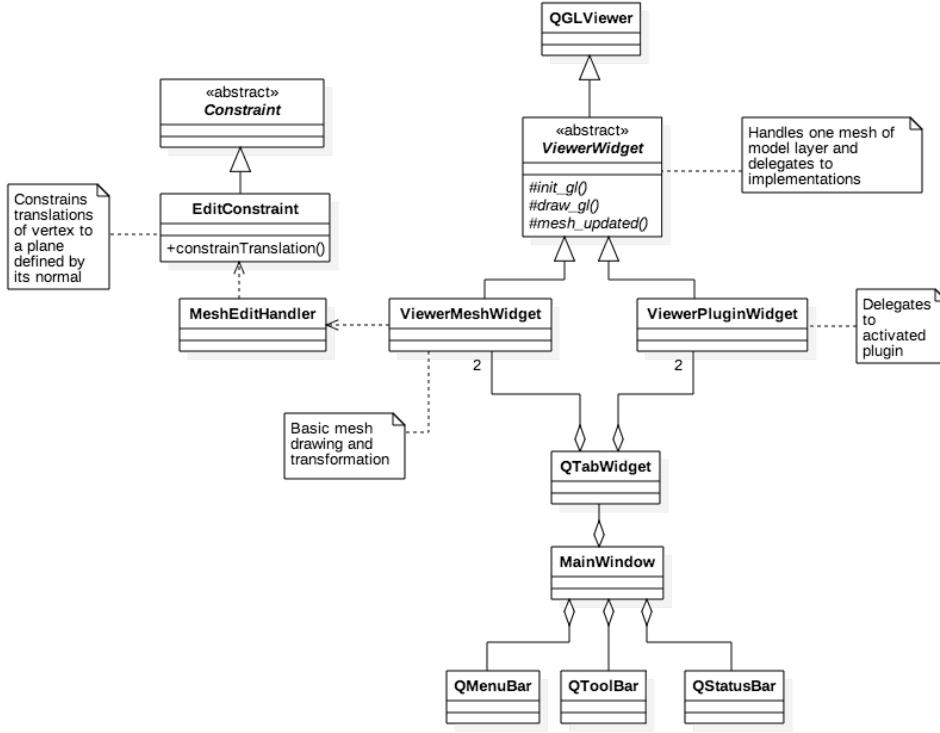


Abbildung 4.5: Architektur der View-Komponente von SubVis

zu erhöhen. Im Fall a) müssen alle Einträge ab dem Index bis zum aktuellsten Element entfernt werden. Dadurch ist auch garantiert, dass auf jeden Fall ein Element Platz hat. Bei Fall b) müssen Elemente entfernt werden. Dabei werden alle Elemente gelöscht, bis auf das erste Element (leere Polygonnetze) und das zweite Element (zuletzt geladene, unveränderte Polygonnetze).

4.5.4 View

Die View-Komponente setzt wo möglich auf vorhandene Qt-Klassen oder erweitert diese (siehe Abbildung 4.5). Die zentrale Klasse ist *MainWindow*. Sie wird zum Teil von Qt aus einer Formulardatei generiert (in einem ui_mainwindow.h Header) und zum Teil manuell implementiert. *MainWindow* besteht aus den UI-Klassen für Menü-, Werkzeug- und Statusleiste. Des Weiteren aus zwei QT-SubWidgets. Eines davon wird als Container für die Plugin-GUIs benutzt (pro Plugin ein Tab). Das andere ist der Container für die beiden Viewer-Tabs *Kontrollnetz-Rendering* und *Plugin-spezifisches Rendering*. Der aktuell aktive Plugin-Tab wird automatisch dem Plugin-spezifischen Ansicht-Tab zugeordnet und kann eine eigene Beschriftung setzen. Für die Realisierung von zwei simultan aktiven Ansichten pro Tab wird ein *QSplitter* eingesetzt, welcher die

verfügbare Fläche in zwei gleiche Teile unterteilt.

ViewerWidget

Die Oberklasse ViewerWidget stellt einen generischen Viewer für SubVis dar. Sie erweitert QGLViewer und verbindet sich mit dem *updated* Signal der Model-Komponente. Mittels des Template-Patterns können die Unterklassen von ViewerWidget ihre OpenGL Draw-Calls ausführen. ViewerWidget initialisiert zuvor einige Maus-Bindings, kümmert sich um das Layout und reicht das konkrete Polygonnetz an die Unterklasse weiter. Ein Viewer wird stets mit einer ID instantiiert, die den Index des Polygonnetzes (vgl. Model-Komponente) spezifiziert. Somit müssen die Unterklassen lediglich ein einzelnes Polygonnetz verwalten und nicht zwischen beiden unterscheiden.

ViewerPluginWidget

Diese Klasse ist im Prinzip lediglich ein dünner Wrapper um ein Plugin und delegiert alle nötigen Aufrufe an das Plugin weiter.

ViewerMeshWidget

Die Klasse handhabt das Rendering des Kontrollnetzes und ermöglicht die Editieroperationen. Alle Editieroperationen und damit zusammenhängende Events werden an die Klasse *MeshEditHandler* delegiert.

Als Grundlage für das Rendering des Kontrollnetzes dienen die Flächen (Faces). Aus diesen können Daten sowohl für das Rendering der Ecken (Vertices), Kanten (Edges) und Flächen (Faces) gewonnen werden. Dazu wird über alle Flächen des Kontrollnetzes iteriert und die Indizes jeweiligen Eckpunkte in einer temporären Datenstruktur gespeichert. Dabei handelt es sich um einen Vektor mit einer sehr geringen Zugriffszeit. Dieses Iterieren erfolgt jeweils nach dem Wechsel des Rendering Modus, oder einem Update des Kontrollnetzes. Dies tritt auf bei der Anwendung eines Algorithmus auf das Kontrollnetz, oder beim Laden eines neuen Kontrollnetzes. Beim Rendering der Kanten ist noch ein weiterer Schritt notwendig. Da alle Kanten einzeln gerendert werden, müssen die Eckpunkte der Flächen jeweils doppelt berücksichtigt werden. Diese liegen bei beispielsweise vier Eckpunkten nach dem Iterieren in folgender Form vor: 1, 1, 2, 2, 3, 3, 4, 4. Im nächsten Schritt wird der erste Eintrag des Vektors an dessen Ende verschoben: 1, 2, 2, 3, 3, 4, 4, 1. Nun können die Ecken jeweils aufeinanderfolgend verbunden werden und es ergeben sich die Kanten der Fläche.

Beim Rendering des Kontrollnetzes wird in einem ersten Schritt die Optimierung des „Depth Testing“ aktiviert. Dies sorgt dafür, dass nur die für die Kamera sichtbaren Elemente gerendert werden. Abhängig von den vom Benutzer gewählten Einstellungen werden im nächsten Schritt Beleuchtung und Färbung aktiviert beziehungsweise deaktiviert. Die Färbung wird im Quellcode erstellt und kann dort entsprechend in der Funktion „create_color_values()“ definiert werden. Die Farbwerte müssen als Vektor von RGB Werten vorliegen. Zu Demonstrationszwecken werden die Absolutwerte der X-, Y- und Z-Koordinaten der Ecken als RGB Farbwerte interpretiert. Daraufhin wird der vom Benutzer gewählte Shading Modus aktiviert. Im Anschluss daran erfolgt das tatsächliche Rendering im entsprechenden Modus (Ecken, Kanten, Flächen). Abschließend wird die OpenGL wieder in den ursprünglichen Zustand zurückversetzt.

Das Rendering der Limesfläche erfolgt mittels dreier unterschiedlicher Renderer. Zur Verfügung stehen ein interpolierender Renderer, der die Fläche basierend auf einer Subdivision Berechnung der Stufe vier rendert. Zusätzlich wird ein interpolierendes Shading angewendet, wodurch die Fläche keine erkennbaren Flächen aufweist, sondern für den Benutzer wie die tatsächliche Limesfläche erscheint.

Der B-Spline Renderer rendert die Fläche mit Hilfe von OpenGL Evaluators. Hierbei werden die Punkte des Kontrollnetzes als Kontrollpunkte interpretiert und eine Bezier Fläche gerendert. Die Berechnung und das anschließende Rendering der Bezier Fläche wird über das OpenGL NURB Interface durchgeführt.

Der Box-Spline Renderer wandelt zuerst die Basis des Kontrollnetzes in eine Bezier Basis. Daraufhin wird das Rendering des B-Spline Renderers verwendet.

MeshEditHandler

Der Handler reagiert auf die entsprechenden Maus und Tastenevents. So löst ein Doppelklick entweder den Selektions- oder den Speichermechanismus aus. Die Taste S steuert die Art des Constraints der Translation des bewegten Vertex.

Mittels *Color Picking* wird versucht den Vertex unter dem Mauszeiger zu bestimmen. Dazu wird zuerst ein Mapping zwischen eindeutiger ID und dem Vertex selber hergestellt. Zuerst wird die Fläche gelöscht und mit weißer Farbe überzeichnet. Weiß wurde ausgewählt, da aufgrund der Zuordnung ein schwarzer Hintergrund die ID 0 bekommen würde, was eine gültige Vertex ID ist. Zur Erhöhung der Performance wird mittels *glScissor* der Zeichenbereich auf ein kleines 10x10 Pixel Feld um den Mausklick herum eingegrenzt. Danach wird jeder Vertex in diesem Feld durch einen Punkt mit der Größe 10 Pixel gerendert. Entscheidend ist hierbei die Farbe, diese muss für jeden Vertex eindeutig sein.

Zur Berechnung der Farbe wird die Vertex ID byteweise auf ein RGB-Array ge mappt. Das höchstwertigste Byte wird nicht genutzt, da das Color-Picking mit Alpha-Kanal nicht funktionierte. Die Funktion ist in Listing 4.1 exemplarisch dargestellt. Dadurch resultiert die Höchstanzahl an Vertices für eine zuverlässige Selektion von $2^{24} = 16777216$ Vertices.

Listing 4.1: Umrechnung Index nach Farbwert

```

1 const unsigned char* index_ptr = &index;
2 for (int i = 0; i < 3; i++)
3     rgb[i] = index_ptr[i];

```

Nach dem Rendern der Vertices wird mittels *glReadPixels* die Farbe unter dem Mauszeiger ausgelesen. Mittels der inversen Funktion in Listing 4.2 wird der Farbwert wieder in einen RGB-Wert umgerechnet.

Listing 4.2: Inverse Funktion: Farbwert nach Index

```

1 int index = 0;
2 unsigned char* index_ptr = (unsigned char*) &index;
3 for (int i = 0; i < 3; i++)
4     index_ptr[i] = rgb[i];

```

Wenn ein Vertex gefunden wurde, wird dieser in einem *ManipulatedFrame* an der Position (0, 0, 0) mit einem roten Punkt gezeichnet. Dieser Frame kann nun mit der Maus bewegt werden und die aktuelle Position mittels *getPosition* wieder ausgelesen werden. Der Frame ist in der Bewegung jedoch eingeschränkt durch die Klasse *EditConstraint*, welche die Translationen auf die Ebene der Vertex-Normalen oder eine dazu orthogonale Ebene einschränkt. Die Berechnung der Normale ist in *Surface_mesh* schon implementiert und folgt dabei dem gängigen Schema: Zuerst werden die Normalen der umgebenen Flächen berechnet und aufaddiert, dann wird der entstandene Vektor normalisiert. Dieser ergibt nun die Vertex-Normale.

EditConstraint

Diese Klasse beinhaltet ein *LocalConstraint* Objekt, welches ermöglicht Translationen und Rotationen eines Frame einzuschränken. Sie erlaubt es, einen Vektor anzugeben der als Normale einer Ebene interpretiert wird und die Translationen auf diese Ebene einschränkt. Zusätzlich kann durch ein Flag eine orthogonale Ebene als Einschränkung definiert werden. Diese wird dadurch charakterisiert, dass ein orthogonaler Vektor berechnet wird der als Ebenen-Normale interpretiert wird.

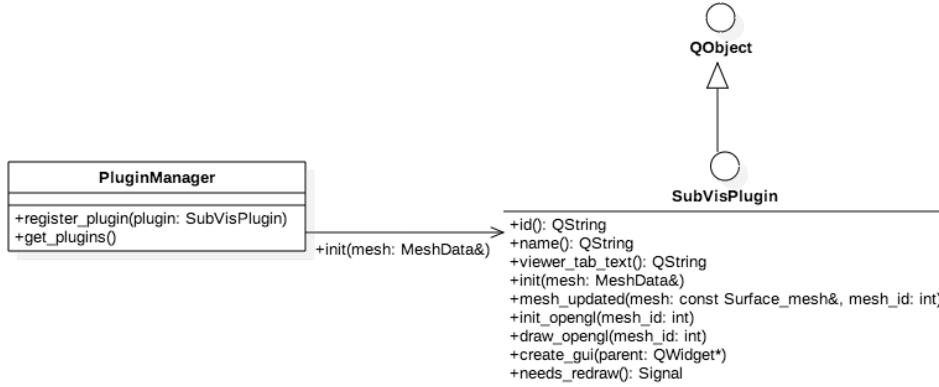


Abbildung 4.6: Architektur der Plugin-Komponente von SubVis

4.5.5 Plugin

Die Plugin-Komponente besteht aus einem *PluginManager* (siehe Abbildung 4.6) und der Plugin-Schnittstelle. Alle Plugins werden vor Programmstart am *PluginManager* registriert. Dieser ruft dann die *init* Methode der Plugins auf und übergibt eine Referenz auf die Model-Komponente.

4.6 Erstellung eines Plugins

Zur Erstellung eines Plugins ist es lediglich notwendig von der Klasse *SubVisPlugin* abzuleiten und alle Methoden zu implementieren (pure virtual). Das Plugin und alle Dateien sollten in ein Unterverzeichnis von *app/plugins/* gelegt werden und in der *app.pro* Datei notiert werden. Danach muss das Plugin registriert werden, was in der Datei *main.cpp* geschieht (siehe Listing 4.3).

Listing 4.3: Registrierung eines Plugins

```

1 // Register your plugins here:
2 subvis_app.register_plugin(std::move(PluginPtr {new ←
3     SubdivPlugin}));
```

Ein erneutes Übersetzen des Programms beendet den Vorgang und das Plugin ist lauffähig und auswählbar.

4.7 Dokumentation

Die Dokumentation besteht aus diesem Bericht sowie den Quellcodekomentaren und einer README.md Datei. Durch Quellcodedokumentierung sollen

Entwickler befähigt werden schnell in das Projekt einsteigen zu können und das Programm weiter zu entwickeln bzw. durch Plugins zu erweitern.

Das Programm ist durchgängig mit Doxygen-Kommentaren versehen, welche mittels eines einfachen *make doc* in HTML überführt werden. Zusätzlich ist eine README.md Datei vorhanden, welche einen kurzen Überblick über das Projekt gibt und eine Art „getting started“ darstellt.

Zusätzlich wurde das Programm mit vielen Debug-Nachrichten versehen, um eine schnelle Fehlersuche bzw. ein einfacheres Verständnis der Abläufe zu ermöglichen.

4.8 Buildprozess

Der Build wird mittels *qmake* durchgeführt. Dazu wird, wie in Qt-Projekten üblich eine *.pro* Datei definiert, die alle Quellen enthält. Diese wird ausgelesen und ein übliches Makefile erstellt, welches ausgeführt werden kann. Einige Bibliotheken werden dynamisch, andere statisch gelinkt. Die statischen Bibliotheken sind in den Quellen des Projekts enthalten (Verzeichnis lib). Die genaue Auflistung ist in Tabelle 4.2 ersichtlich.

Tabelle 4.2: Linking der Bibliotheken

Bibliothek	Linking
Surface_mesh	statisch
libQGLViewer	statisch
Qt	dynamisch
OpenGL	dynamisch
GLM	dynamisch

Um das Projekt bauen zu können, sind folgende Schritte notwendig:

1. Sicherstellen, dass alle dynamischen, benötigten Bibliotheken installiert sind (siehe Tabelle 4.1 und Tabelle 4.2)
2. Im Hauptverzeichnis SubVis *qmake* ausführen
3. *make*
4. *make doc* für die Dokumentation
5. *make clean* für die Bereinigung der app-Builds
6. *make distclean* um die app-Builds und libs-Builds zu bereinigen

Sollen die Debug-Nachrichten ausgeschaltet werden, muss in der Datei *app.pro* der Wert *debug* von der *CONFIG* Variable entfernt werden.

4.9 Installation

Es sind lediglich die Laufzeitabhängigkeiten (siehe Tabelle 4.2) zu installieren. Alle statischen Abhängigkeiten sind im Programm enthalten. Zur Ausführung muss lediglich die Datei *SubVis* gestartet werden.

4.10 Benutzerhandbuch

Dies soll eine kurze Übersicht über die Funktionen und deren Verwendung sein.

4.10.1 Generelle Oberfläche

Abschnitt 4.4 beschreibt die Oberfläche. Das Fenster kann beliebig vergrößert werden und bietet die Möglichkeit die rechte Seitenleiste in der Größe zu verändern. Dazu muss lediglich der Rand zwischen Seitenleiste und Viewer mit der Maus verschoben werden. Wird der Rand fast ganz nach rechts gezogen, verschwindet die Seitenleiste. Durch ein erneutes nach links ziehen des rechten Randes erscheint sie wieder. Alle Aktionen der Werkzeugleiste können auch über die Menüleiste benutzt werden. Wenn eine Aktion aktiv ist, wird diese optisch eingedrückt dargestellt. Jede Aktion hat ein zugewiesenes Tastaturkürzel, welches neben dem Menüeintrag dargestellt wird. Da sich viele Aktionen auf ein spezifisches Polygonnetz beziehen (linke Seite bzw. rechte Seite), bieten die Tastaturkürzel meist die Möglichkeit mittels nachgelagertem Drücken der Taste 1 oder 2 die linke bzw. rechte Seite auszuwählen.

4.10.2 Rückgängig/Wiederherstellen

Jede Veränderung am Polygonnetz (z. B. Anwendung eines Algorithmus, Verschieben eines Vertex) führt zu einem neuen Historieneintrag. Um eine Aktion rückgängig zu machen werden die Pfeile oben rechts benutzt. Wenn kein weiterer Schritt zurück oder nach vorne möglich ist, wird der jeweilige Pfeil ausgegraut. Es wird nur eine bestimmte Anzahl von Schritten abgespeichert. Wird diese Höchstgrenze überschritten, werden die ältesten Modifikationen aus der Historie entfernt.

4.10.3 Laden/Speichern

Über das Menü *File* können Dateien geladen und gespeichert werden. Beim Laden werden beide Polygonnetze ersetzt. Über die Historie ist stets das unveränderte, zuletzt geladene Polygonnetz erreichbar. Beim Speichern ist darauf zu achten, die korrekte Dateiendung anzugeben, andernfalls wird eine Fehlermeldung angezeigt.

4.10.4 Screenshots

Ein Screenshot kann im Menü *File* angefertigt werden. Es lässt sich das Dateiformat und der Speicherort festlegen.

4.10.5 Triangulieren

Für manche Algorithmen kann es notwendig sein, dass das Polygonnetz zuvor trianguliert wurde. Dies lässt sich leicht über das Menü *Edit* erledigen. Dabei kann ausgewählt werden, welches Polygonnetz bearbeitet werden soll.

4.10.6 Ansichten synchronisieren

Standardmäßig sind beide Ansichten der Polygonnetze synchronisiert. Das heißt, die Bewegung der Kamera in einer Ansicht bewegt die andere in gleichem Maße. Dies kann im Menü *View* umgeschaltet werden.

4.10.7 Splitscreen umschalten

Sollte nur ein einzelnes Polygonnetz von Interesse sein, so lässt sich die rechte Ansicht über das Menü *View* ausblenden. Die Daten bleiben erhalten, lediglich die Ansicht wird nicht dargestellt bzw. gerendert.

4.10.8 Editieren

Um einen Vertex zu verschieben, muss der Editiermodus aktiviert werden. Dies erfolgt im Menü *Edit*. Gleichzeitig wird die rechte Ansicht und die Seitenleiste ausgeblendet. Ein Doppelklick auf einen Vertex färbt diesen rot und stellt ihn als größeres Viereck dar (siehe Abbildung 4.2). Nun kann der Vertex auf der Ebene bewegt werden. Dazu muss die Tastenkombination STRG+linke Maustaste benutzt werden. Hilfreich ist es dabei, sich vorzustellen, dass der Vertex an einem Faden (rote Linie) hängt und dabei nur um die weiße Linie schwingt. Die Richtung der Linie kann mittels der Taste *S* orthogonal umgeschaltet werden. Somit sind alle Punkte im Raum erreichbar. Soll der Vertex an seiner neuen

Position bleiben, muss lediglich ein erneuter Doppelklick ausgeführt werden (an beliebiger Stelle). Der Vertex wurde nun erfolgreich verschoben.

5 Subdivision Plugin

Die Hauptfunktionalität des Programms SubVis wird ebenfalls in einem Plugin bereitgestellt. Dieses erlaubt die Anwendung verschiedener Unterteilungsalgorithmen und das Rendering deren Limes-Flächen.

Abbildung 5.1 gibt einen Überblick über die Architektur. Die Hauptklasse ist *SubdivisionAlgorithmsPlugin* welche auch die SubVisPlugin Schnittstelle implementiert. Die GUI, das Rendering und die Algorithmen wurden in jeweils eigene Klassen ausgelagert.

5.1 GUI

Die GUI besteht aus zwei Auswahllisten für den Algorithmus pro Polygonnetz, einer Einstellung für die Anzahl an Unterteilungsschritten und einem Start bzw. Stop Button (siehe Abbildung 5.2).

5.2 Rendering

Extras - Render Modi (Vertices, Edges, Faces) - Licht (an/aus) - Shading (flat/smooth) - Färbung (an/aus; über Code konfigurierbar; Vector, der entsprechend indiziert zugegriffen wird)

Jeder Renderer erweitert die abstrakte Klasse *GLRenderer*. Diese ruft mittels Template-Pattern eine Methode *render* ihrer Unterklasse auf, um den Rendervorgang zu starten. Bei Veränderungen der Model-Komponente wird das Polygonnetz kopiert, damit Unterklassen das Polygonnetz verändern können. Ziel ist es die Limesfläche des Kontrollnetzes zu rendern. Abhängig vom angewendeten Unterteilungsalgorithmus wird die Limesfläche unterschiedlich berechnet.

5.2.1 Interpolating Renderer

Bei dem Modified Butterfly und dem Butterfly Algorithmus handelt es sich um interpolierende Unterteilungsalgorithmen. Hier geht die Limesfläche durch das Kontrollnetz und es wird ein interpolierender Renderer benötigt. Die Kontrollpunkte werden interpoliert und die Oberflächenbeläuchtung anhand der Vertex Normalen berechnet, sodass die Oberfläche glatt erscheint.

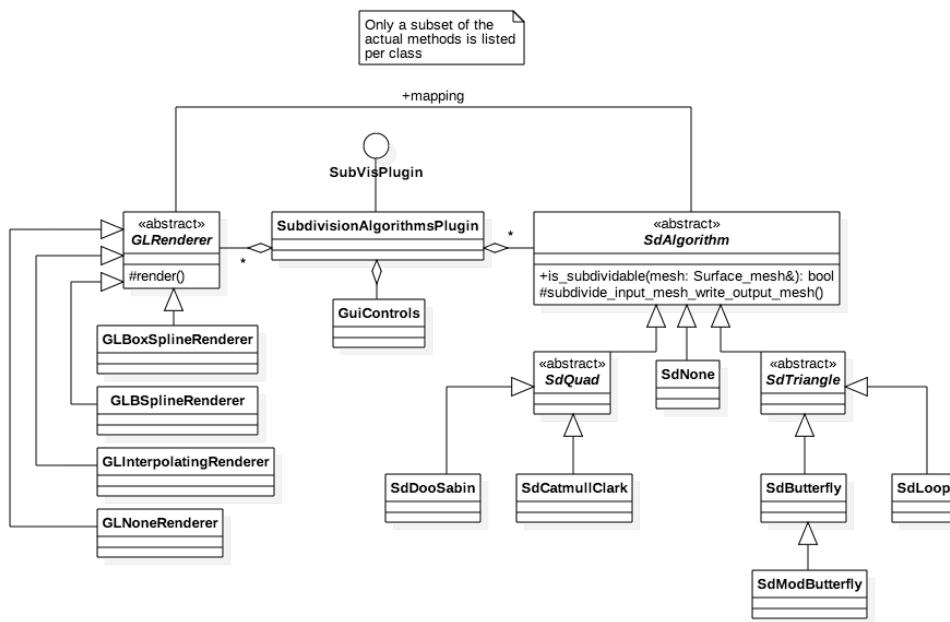


Abbildung 5.1: Architektur des Subdivision-Plugins

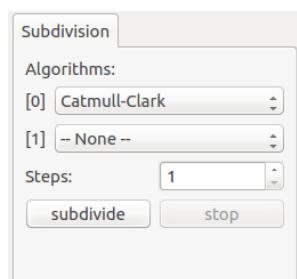


Abbildung 5.2: GUI-Elemente des Plugins

5.2.2 B-Spline Renderer

Catmull-Clark und Doo-Sabin sind Unterteilungsalgorithmen auf Basis von kubischen und quadratischen B-Splines. Für ein Rendering kann die B-Spline Tensorproduktfläche nach Bezier gewandelt werden. Die Kontrollpunkte können dann als Bezier Tensorproduktfläche interpretiert werden. Aktuell ist an dieser Stelle noch der *Interpolating Renderer* implementiert.

Rendering mit Evaluators

5.2.3 Box-Spline Renderer

Der Unterteilungsalgorithmus Loop basiert auf Box-Splines. Die Fläche kann zum Rendern in ein Bezier-Dreiecksnetz gewandelt werden. Aktuell ist an dieser Stelle noch der *Interpolating Renderer* implementiert.

//TODO Tobi

5.2.4 None Renderer

Der None Renderer ist eine Dummy Implementierung für den Fall, dass kein Unterteilungsalgorithmus ausgewählt ist. Dann wird nichts dargestellt.

5.3 Algorithmen

Die Algorithmen werden über die abstrakte Klasse *SdAlgorithm* zusammengefasst. Diese ermöglicht es der GUI, abzufragen, ob ein bestimmtes Polygonnetz von diesem Algorithmus unterteilt werden kann oder nicht, was sich in der Auswahlmöglichkeit der GUI niederschlägt. Außerdem implementiert die Klasse die Verwaltung der Polygonnetze und ein Thread-basiertes unterteilen. Somit müssen die konkreten Klassen der Algorithmen lediglich eine Methode überschreiben und sich nur auf die korrekte Implementierung fokussieren. Es werden vor jedem Aufruf der Unterklasse die Variablen *input_mesh* und *output_mesh* entsprechend initialisiert. Die Unterklassen müssen dann pro Aufruf von *subdivide_input_mesh_write_output_mesh* das Eingangsnetz lesen und das Ergebnis in der Variable *output_mesh* speichern.

Listing 5.1: Signatur der Unterteilungsfunktion

```
1 void subdivide_threaded(const Surface_mesh& mesh,
2 std::function<void(std::unique_ptr<Surface_mesh>)> ←
3     callback,
3 const int steps = 1);
```

Das Thread-basierte Rendering ist über einen Callback-Mechanismus implementiert (vgl. Listing 5.1), welcher garantiert, dass die Callback-Funktion auf dem UI-Thread ausgeführt wird. Die Funktion `subdivide_threaded` ruft eine Worker-Funktion mittels `QtConcurrent::run` auf, welche in einer Schleife die konkrete Implementierung der Unterteilung ausführt. Die Worker-Funktion sendet am Ende ihrer Ausführung ein Signal `finished` aus. Dieses wiederum wird von der Klasse empfangen und in einen Aufruf der Callback-Funktion umgesetzt. Durch Verwendung des Signal-Slot Konzepts ist garantiert, dass der Slot und somit der Callback auf dem UI-Thread ausgeführt wird. Der implementierte Callback-Mechanismus macht somit das Threading wesentlich einfacher und benötigt keine Synchronisierungsmechanismen.

Als konkreter Callback wird eine Funktion verwendet, die bei Beendigung das Ergebnis in die Model-Komponente lädt (was ein erneutes Zeichnen des Polygonnetzes zur Folge hat).

5.3.1 Vererbungshierarchie

SdAlgorithm ist die Klasse von der alle Unterteilungsalgorithmen abgeleitet werden. Die Klasse implementiert wie bereits beschrieben alle Basisfunktionen. Die Algorithmen Catmull-Clark, Doo-Sabin, Loop, Butterfly und Modified Butterfly lassen sich aber noch weiter klassifizieren und zusammenfassen. Daher werden die beiden Unterklassen *SdQuad* und *SdTriangle* eingeführt, die von *SdAlgorithm* erben.

SdQuad vereinigt gemeinsame Funktionen von Catmull-Clark und Doo-Sabin. Dazu gehört das Verwalten von Properties (Edge Point, Face Point ...) und die Berechnung des Face Points.

SdTriangle fasst die Unterteilungsalgorithmen für Dreiecksnetze zusammen. Loop, Butterfly und Modified Butterfly verwenden die gleiche Face Split Funktion und benötigen ähnlichen Properties.

Die konkreten Implementierungen der Unterteilungsalgorithmen erben schließlich von *SdQuad* oder *SdTriangle*. Durch diese Architektur wird redundanter Code vermieden.

5.3.2 Catmull-Clark

Die Implementierung von Catmull-Clark ist wie in Unterabschnitt 2.1.2 beschrieben durchgeführt. *SdCatmull* erbt von *SdQuad*. Der Algorithmus kann mit einem beliebigen Polygonnetz umgehen. Die Randregeln sind vollständig implementiert. Abbildung 5.3 zeigt einen Würfel und das Netz nach einer Unterteilung.

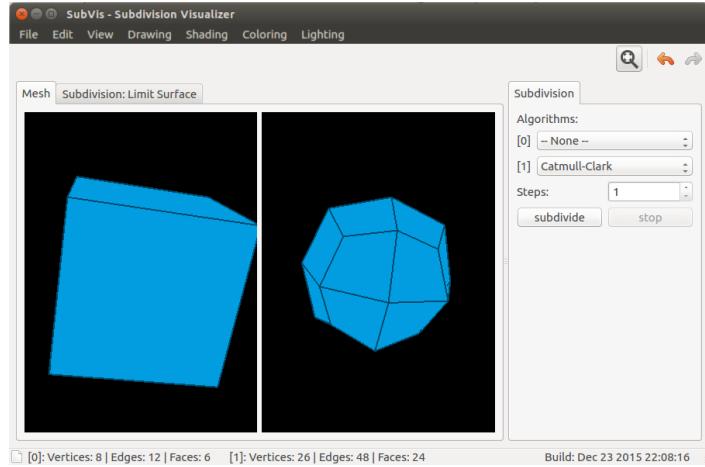


Abbildung 5.3: SubVis - Catmull-Clark

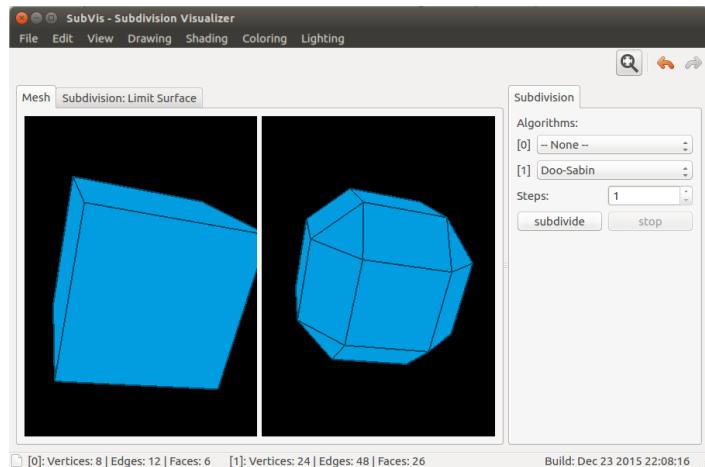


Abbildung 5.4: SubVis - Doo-Sabin

5.3.3 Doo-Sabin

Der Doo-Sabin Algorithmus ist inklusive Randregel wie in Unterabschnitt 2.1.3 beschrieben umgesetzt. Die Klasse *SdDooSabin* erbt von *SdQuad*. Abbildung 5.3 zeigt einen Unterteilungsschritt eines Würfels.

5.3.4 Loop

Loop ist in der Klasse *SdLoop* implementiert und erbt von *SdTrianlge*. Die Implementierung erfolgt wie in Unterabschnitt 2.1.4 beschrieben und unterstützt die beschriebenen Randregeln. Die Berechnung der Even Vertices erfolgt dabei nach der Variation nach Warren. Abbildung 5.5 zeigt die Unterteilung eines Icosahedrons.

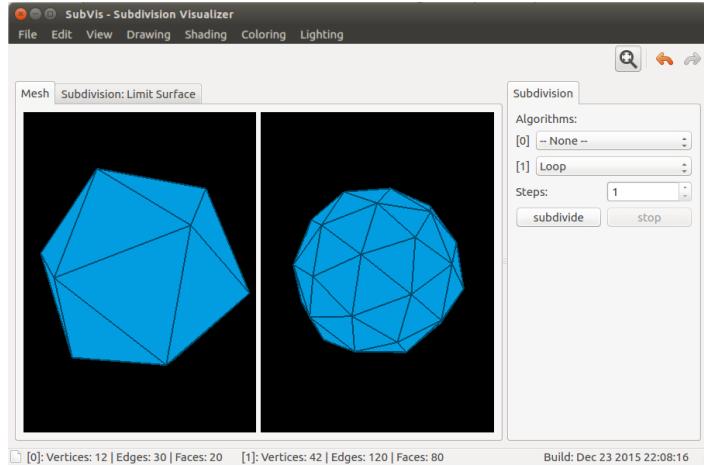


Abbildung 5.5: SubVis - Loop

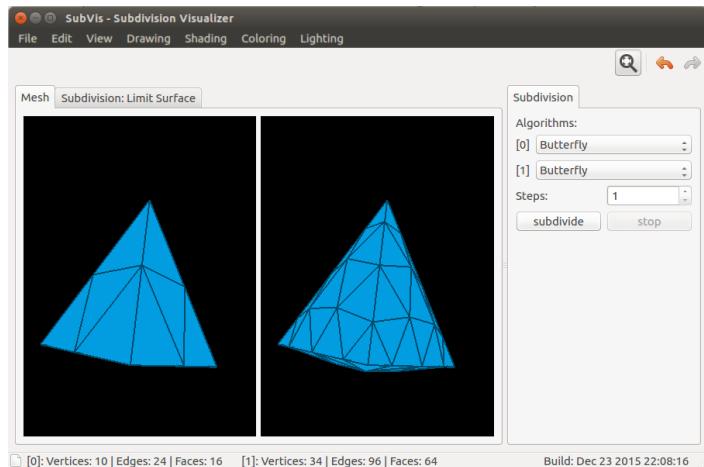


Abbildung 5.6: SubVis - Butterfly

5.3.5 Butterfly

Der Butterfly Algorithmus ist nach Unterabschnitt 2.1.5 inklusive Randregel in der Klasse *SdButterfly* implementiert. Für die Unterteilung wird das klassische Eight-Point Stencil verwendet. *SdButterfly* erbt von *SdTrianlge*. Abbildung 5.6 zeigt einen Unterteilungsschritt.

5.3.6 Modified Butterfly

Der Modified Butterfly Algorithmus ist nach Unterabschnitt 2.1.6 implementiert. Die Klasse *SdButterfly* erbt jedoch nicht von *SdTriaangle*, sondern von *SdButterfly*, da die Unterteilung beim Modified Butterfly im regulären Fall bis auf Ausnahmeregeln identisch mit dem Butterfly Algorithmus ist. In *SdButterfly* muss für eine korrekte Implementierung somit nur noch die Methode

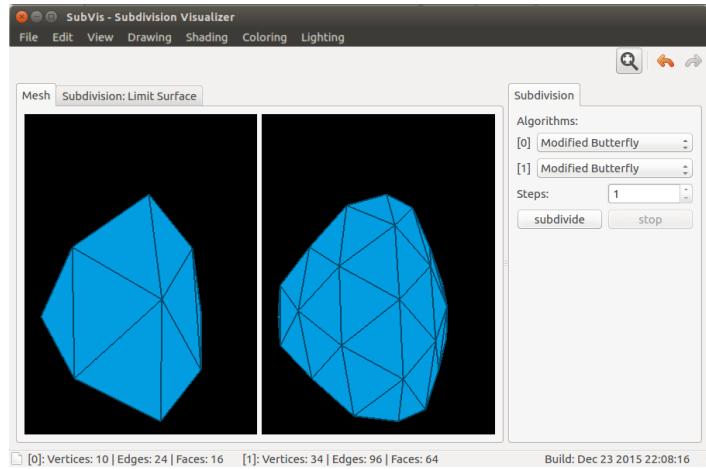


Abbildung 5.7: SubVis - Modified Butterfly

compute_edge_point überschrieben werden, die die Sonderfälle behandelt. Die Implementierung verwendet das klassische Eight-Point Stencil (bzw. Ten-Point Stencil mit $w = 0$).

Der Modified Butterfly Algorithmus ist mit der Standard Randregel implementiert. Die erweiterten Randregeln, die am Ende von Unterabschnitt 2.1.6 erwähnt werden, sind nicht umgesetzt. Abbildung 5.7 zeigt einen Unterteilungsschritt.

6 Projektverlauf

In den folgenden Abschnitten wird der Projektverlauf beschrieben und dabei auf die erreichten und geplanten Ergebnisse eingegangen, sowie der bisherige Ablauf des Projekts bewertet. Außerdem wird auf die Arbeitsteilung eingegangen.

6.1 Aufgabenverteilung

Für die Entwicklung werden die Themen und Aufgaben unter den Teammitgliedern in drei Arbeitspakete aufgeteilt.

Architektur, Oberfläche, IDE, Editiermodus Hierzu gehört die Oberfläche, Bedienung, Architektur inkl. Plugin-System und die Spezifikation einer Entwicklungsumgebung. Des Weiteren die Implementierung des Editiermodus.

Rendering und Darstellung Berechnung und Darstellung der Kontrollnetze und deren Limesfläche mit OpenGL.

Unterteilungsalgorithmen Umfasst die Implementierung der ausgewählten Unterteilungsalgorithmen.

Tabelle 6.1: Aufgabenverteilung unter den Teammitgliedern

Arbeitspaket	Teammitglied
Architektur, Oberfläche, IDE, Editiermodus	Simon Kessler
Rendering und Darstellung	Tobias Keh
Unterteilungsalgorithmen	Felix Born

Tabelle 6.1 zeigt die vorgenommene Aufgabenverteilung des letzten Semesters.

6.2 Verlauf der beiden Semester

Im ersten Semester lag der Schwerpunkt darin, einen Überblick über die theoretischen Grundlagen zu erhalten. Hierzu gehören die Unterteilungsalgorithmen

und die Darstellung der Kontrollnetze. Ein weiter wichtiger Punkt war die Evaluierung von Bibliotheken, die bereits Algorithmen und vor allem Datenstrukturen implementieren. Als Ergebnis wurde die Datenstruktur Surface_mesh als am geeignetsten bewertet.

Des Weiteren wurde die Architektur entworfen und in einem UML-Diagramm festgehalten. Außerdem entstand eine einheitliche Entwicklungsumgebung basierend auf Qt und dem Qt Creator unter Ubuntu. Diese enthält die nötigen Build-Dateien, Verzeichnisse, Bibliotheken und Git-Konfigurationen. Jedes Teammitglied besitzt somit eine einheitliche, funktionierende Entwicklungsumgebung. Zusätzlich wurde eine Dokumentationslösung auf Quelltextebene mittels Doxygen implementiert.

Da für alle Beteiligten der Umgang mit C++ und insbesondere Computergrafik ein neues Themengebiet darstellt, war ein ausführlicher theoretischer Einstieg in das Thema notwendig. Hierbei hat sich die Unterstützung durch Herrn Prof. Dr. Georg Umlauf und Pascal Laube als sehr hilfreich erweisen. In den Projektbesprechungen wurden technische Details geklärt und der theoretische Hintergrund erläutert. Erst durch Verstehen der Details bezüglich der Unterteilungsalgorithmen, des Renderings und der Anwendbarkeit der Unterteilungsalgorithmen auf verschiedene Kontrollnetze konnten die Tools und Bibliotheken entsprechend evaluiert werden.

Während des Semesters wurden lediglich für sehr kurze Zeiträume und eng eingegrenzte Themengebiete die Aufgaben zwischen den Teammitgliedern verteilt. Dadurch wurde sichergestellt, dass eine gemeinsame Wissensbasis entsteht, um späteren Missverständnissen vorzubeugen. Dies hat sich als hilfreich erwiesen und ermöglicht es nun an verschiedenen Teilmodulen der Anwendung parallel zu arbeiten.

Im zweiten Semester wurde das Programm SubVis implementiert. Dabei wurden kleinere Änderungen an der Architektur gegenüber der Spezifizierung im ersten Semester vorgenommen. Zuerst wurde eine GUI und Model-Schicht entwickelt. Diese dienten als Plattform, um die weiteren Komponenten zu implementieren. Danach wurden die einzelnen Algorithmen, Renderingverfahren und der Editiermodus ergänzt. In Anlehnung an agile Entwicklungsmodelle wurden zweiwöchige Meetings mit den Betreuern Prof. Dr. Georg Umlauf und Pascal Laube durchgeführt. Dort wurde auch der aktuelle Stand der Software präsentiert oder Fragen geklärt. So konnte frühzeitig auf Änderungs- und Funktionswünsche eingegangen werden.

Schlussendlich kann das Projekt als erfolgreich bezeichnet werden, da alle vorgesehenen Funktionen und darüber hinaus noch weitere implementiert wur-

den. Für alle beteiligten Studenten war dies ein überaus lehrreiches Projekt mit einigen neuen Programmiersprachen und Anwendungsdomänen.