Generierung von Textur-Atlanten für aus Kameraansichten rekonstruierte Objekte

Studienarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: Mirko Lauff
in: Computergraphik

Gutachter: Prof. Dr.-Ing. Peter Eisert

eingereicht am: 20. August 2015

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Berlin, den 20. August 2015 Mirko Lauff

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Texture Mapping ermöglicht es auf einfache Weise, komplexe Geometrien und Farbverläufe auf die Oberfläche von 3D-Objekten zu projizieren und so einen visuellen Eindruck zu erzeugen, der einerseits reichhaltiger und realistischer und andererseits günstiger in Anbetracht von Performance und Speicherkosten ist als die bloße Färbung einzelner Flächen eines Objekts. Die Zunahme der Rechenleistung, verbesserte Hardware und programmierbare GPUs ermöglichen eine immer größere Anzahl an darstellbaren Objekten innerhalb einer Szene mit der Folge, dass das Laden einer Textur pro Objekt einen zeitlichen Engpass darstellt, der eine flüssige Darstellung gerade von bewegten Szenen verhindert. Das Laden der Textur von der Festplatte in den Speicher der Grafikkarte, um Veränderungen an der Textur vornehmen zu können, stellt die zeitintensivste Operation bei der Darstellung künstlich erzeugten Szenen dar, das sich unter dem naiven Ansatz, eine Textur pro Objekt zu laden, potenziert. Die Lösung dieses Problems ist vom Gedankenansatz jedoch relativ einfach: Anstelle jeweils eine Textur pro Objekt in den Speicher der Grafikkarte zu laden, werden die Texturen mehrerer Objekte zusammengelegt und als großer Verbund, dem Texturatlas, einmalig zur Grafikkarte hochgeladen, wo sie dort für weitere Operationen zur Verfügung stehen.
Textur-Atlanten stellen große Texturkarten dar, Konglomerate, die aus vielen einzelnen Texturen oder Einzelteilen einer großen Textur zusammengesetzt sind. Jede Sub-Textur nimmt dabei nur einen Teil des Atlas ein. Durch Modifizierung der uv-Koordinaten der UV-Map eines Objektes können den einzelnen Objekten einer Szene bestimmte Texturen innerhalb des Atlas zugewiesen werden. Für wenige Objekte oder Körper mit geringer Komplexität lässt sich im optimalen Fall der Atlas sogar per Hand passend zur Szene kreieren; mit steigender Anzahl der einzelnen Texturen ist dieser Ansatz aus offensichtlichen Gründen jedoch weder effizient noch flexibel. Gerade beim Entfernen oder Hinzufügen weiterer Texturen, zwei Operationen, die stets mit einer Neuordnung der Elemente des Atlas verbunden sind, lohnt es sich deshalb, den Aufbau zu automatisieren.
Texture Maps liegen meist in quadratischer Form vor, wobei die Seitenlängen als Zweierpotenzen angelegt sind, da dies aus Geschwindigkeitsgründen bei zum Bei-

spiel der Indizierung der einzelnen Daten besonders günstig ist. Gerade aber bei komplexeren Objekten kann es jedoch vorkommen, dass die abzubildenden Flächenstücke sich nur unter starker Verzerrung oder unter mangelhafter Ausnutzung der bereitgestellten Fläche auf der Textur abbilden lassen: Die Folge des letzten Punktes ist ungenutzter und somit verschwendeter Speicherplatz innerhalb der Texturkarte, ein Stück Fläche ohne Daten, der trotzdem in den Speicher geladen werden muss, um die Textur benutzen zu können. Texturatlanten kompensieren dieses Problem, indem die einzelnen Texturen kompakt innerhalb des Atlas angeordnet werden. Freistehende Flächen innerhalb einer Sub-Textur können so effektiv zur Platzierung anderer Sub-Texturen genutzt werden unter der Voraussetzung, dass die Offsetwerte der Texturen wie Länge, Höhe und Startpunkt innerhalb des Atlas bekannt sind, um sie später wieder referenzieren zu können.
Mit dem Einzug der 3D-Scanner und der digitalen Bildverarbeitung hat sich das Problem der Texturierung merklich verschoben: Ein dreidimensionales Objekt muss nun nicht unbedingt von einem Künstler erschaffen und zur Steigerung der Optik in möglichst realistische Texturen *verpackt* werden, sondern kann aus den durch den Scanner ermittelten Punktwolken rekonstruiert werden. Struktur und Farbe müssen nicht künstlich oder von Hand erzeugt werden, sondern sind Teil des aufgenommen Objektes, der Prozess der Texturierung ist bei der Erstellung des Objekts durch dessen Aufnahme schon vollendet. Die jetzige Aufgabe lautet daher, das aufgenommene Objekt so zu verarbeiten, dass seine komplexe Oberfläche als 2D-Karte vorliegt und als Textur für dieses Objekt oder andere verwendet werden kann. Und welche Schritte sind notwendig, wenn die Oberfläche eines Objekts sich nicht nur als ein großes Stück in eine Ebene einbetten lässt, sondern weiterhin in kleinere Teilstücke zerlegt werden muss? Wie werden diese Teilstücke auf der Ebene ausgerichtet und referenziert? Gibt es Randbedingungen wie Größe, Anzahl oder Ort des Schnittes, wie ein Objekte in Teilstücke zerlegt werden sollte und welche resultieren in einem besonders visuell ansprechenden Ergebnis?

1.1 Aufbau der Arbeit

Die nachfolgende Arbeit soll folgende Aspekte untersuchen: Ausgehend von triangulierten Polygonnetzen eingescannter Objekte sollen die Schritte der Segmentierung, Parametrisierung und des kompaktem Packings als drei Kernbegriffe der Atlasgenerierung erläutert werden. Im Ersten Abschnitt wird zwischen Single-Chart- und Multi-Chart-Ansätzen unterschieden, wobei der Fokus im Zuge der Erstellung des Texturatlas auf den letzteren liegen soll. Anhand des D-Chart Algorithmus von [Jul05] sollen die wichtigsten Vorgänge bei der Segmentierung eines Modells erklärt und veranschaulicht werden.
Der zweite Teil befasst sich mit der Einbettung der gewonnen Teilnetze in 2D-Ebenen und gibt Auskunft über die verschiedenen Verfahren der Parametrisierung. Der dritte Abschnitt erläutert darauf aufbauend das effiziente Einordnen der ge-

wonnenen Charts in eine Texturkarte und somit die Vollendung des Textur-Atlas. Im Mittelpunkt stehen hier die historische Entwicklung der speziellen Such- und Sortieralgorithmen sowie neuere Verfahren, wie die von [Nö11].
Der letzte Teil dieser Arbeit widmet sich der Auswertung der vorgestellten Verfahren und schafft Ausblick auf mögliche Verbesserung und interessante Untersuchungsgebiete.

1.2 Schwerpunkt

Die Komplexität der der Textur-Atlas Generierung unterbindet eine allzu detaillierte Analyse der einzelnen Teilgebiete, ohne den Rahmen dieser Arbeit zu sprengen. Aus diesem Grund wurde der Schwerpunkt dieser Ausarbeitung auf die Segmentierung eines beliebigen triangulierten Netzes gelegt. Dazu wurde im Rahmen dieser Studienarbeit ein Programm entworfen, das die Segmentierung und Planarisierung von Eingabenetzen wie im ersten Teil der Arbeit nachbearbeitet und graphisch darstellt. Die erarbeitete Software verwendet JavaScript, lässt sich im Browser öffnen und ausführen und nutzt Teile der neuen HTML API, um die Inhalte dieser Arbeit darzustellen. Die gewonnenen Ergebnisse werden abschließend in Kapitel 5 vorgestellt.

KAPITEL 2

Segmentierung

Ziel der Mesh-Segmentierung ist das Ausfindigmachen und Einteilen der Dreiecke einer triangulierten 3D-Oberfläche mit gleichen geometrischen Eigenschaften und räumlicher Nähe zueinander in größere Verbünde, den sogenannten Charts. Die Segmentierung findet stets in Bezug auf die im nächsten Kapitel beschriebene Parametrisierung statt und soll das Mesh so günstig vorbereiten, dass die Teilstücke des Eingabenetzes möglichst entwickelbar sind, mit dem Ziel, Fehler, die im Verlauf der Atlasgenerierung über die Parameterisierung entstehen, schon von vorne herein zu minimieren.
Die Auswahl der Dreiecke, die zu einem Chart hinzugefügt werden, sowie die Anzahl der Charts sind maßgebend für die spätere Qualität der Texturen und letztendlich für den Atlas, sodass schon von vornherein verschiedene Bedingungen berücksichtigt werden müssen: Die Charts sollten so gewählt werden, dass bei ihrer Planarisierung der Stauch- und Dehnungsfaktor möglichst gering ausfällt und die Anzahl der Texturartefakte, also der sichtbaren Bruchstellen der Textur, minimiert werden. Beide Bedingungen verhalten sich reziprok zueinander. Es gilt daher stets zu überprüfen, ob es sinnvoll ist, viele kleine, gut parametrisierbare Charts zu erzeugen, die zu einer erhöhten Sichtbarkeit an Bruchkanten (den Kanten, an denen zwei unterschiedliche Charts zusammenkommen) führen, oder die Anzahl dieser Diskontinuitäten, auf Kosten einer möglichst gelungenen Parametrisierung zu verringern. Dieser Trade-off wird darüber hinaus durch den Umstand erschwert, dass die topologischen Eigenschaften des Meshes ebenfalls Auswirkungen auf die Sichtbarkeit der erzeugen Fehler haben: So sind Bruchlinien innerhalb Gebiete mit hoher Krümmung durchaus wünschenswert, da visuelle Fehler und Artefakte in solchen Regionen vom menschlichen Auge schwerer zu erfassen sind, wie weiter unten gezeigt wird.

2.1 Single-Chart und Multi-Chart Ansätze

Für jeden Chart, in den das Eingabenetz aufgeteilt wird, muss ein Schnitt durch das Mesh erfolgen, wobei, wie oben erwähnt, jeder dieser Schnitte jedoch eine Bruchkante

erzeugt, die bei dem späteren Zusammensetzen der einzelnen Charts als Texturartefakt sichtbar wird. Es erscheint also auf den ersten Blick lohnenswert, die Anzahl der Schnitte innerhalb des Netzes so gering wie möglich zu halten, sodass sich die Frage stellt, ob sich ein beliebiges Eingabenetz mit nur einem Schnitt so zerlegen lässt, dass eine entwickelbare Fläche entsteht, die für spätere Parameterisierung geeignet ist.
Gu et al. beschreiben in [Gu02] einen solchen Single-Chart Ansatz, in dem mithilfe des *region growing* Verfahrens ein Trennlinie gefunden werden kann, die es erlaubt, das Modell mittels eines einzigen Schnittes so aufzuteilen, dass das Modell genau eine Randkurve besitzt und homöomorph zu einer Scheibe ist. Durch weitere optionale Einschnitte, ausgehend vom Rand, kann das Resultat für die Parametrisierung in Hinblick auf Verzerrung und Dehnung bei der Planarisierung verbessert werden, wie später gezeigt wird.
Die Idee hinter dem *region growing* Algorithmus ist folgende: Ausgehend von einem Startdreieck auf dem Eingabenetz, dem *seed*, werden die benachbarten Dreiecke nacheinander untersucht und abgearbeitet. Dabei bilden die verarbeiteten Dreiecke eine wachsende, scheibenartige Region, die mit fortlaufender Iteration das gesamte Eingabenetz überdeckt. Die Kanten, an denen die wachsende Region auf sich selbst trifft, bilden folglich einen Schnitt, durch den das Netz zu einer Scheibe aufgetrennt werden kann.
In seiner einfachsten Variante ist der so erzeugte Schnitt von eher mangelhafter Natur, weil er einerseits übermäßig lang ist, da an der Kantenschleife meist noch unnötige Kantenbäume hängen, und andererseits die Randkurve des Schnitts sehr gezackt und unregelmäßig ausfällt, sodass eine Nachbearbeitung für einen glatten, kurzen Schnitt meist unabdingbar ist.
Um dies besser zu verstehen, soll die Arbeitsweise des Algorithmus hier genauer erläutert werden. Am Anfang des *region growing* Algorithmus werden das Startdreieck und alle Knoten und Kanten des Eingabenetzes markiert, da sie alle potentiell auf der zu suchenden Schnittkante liegen können. Daraufhin wird, ausgehend vom Startdreieck, genau dasjenige Dreieck als nächstes ausgewählt, welches noch nicht markiert ist, genau zu einem markierten Dreieck benachbart ist und eine minimale geodätische Distanz zum Startdreieck besitzt. Dabei versteht man unter der geodätischen Distanz die Metrik, die der gemittelten Distanz der einzelnen Knoten des Dreiecks über den Kanten des Netzes zum Startdreieck entspricht.
Das so ausgewählte Dreieck wird ebenfalls markiert, während die Markierung der Kante, die zwischen den beiden Dreiecken liegt, gelöscht wird. Der Grund dafür ist, dass diese Kante nun innerhalb der wachsenden Region liegt und somit keine Berührungskante der Region mit sich selbst mehr werden kann. Der Algorithmus wird gleichbleibend fortgesetzt, bis keine unmarkierten Dreiecke mehr vorhanden sind, die genau zu einem markierten Dreieck benachbart sind.
Der nächste Schritt besteht darin, alle unmarkierten Dreiecke zu finden, die genau zwei markierten Dreiecken als Nachbarn besitzen und minimale geodätische Distanz aufweisen. Im passenden Fall werden nun die Kantenmarkierungen der Kanten

aufgehoben, die zu zwei markierten Dreiecken benachbart sind. Dies wird solange fortgesetzt, bis nur noch unmarkierte Dreiecke vorhanden sind, die in Nachbarschaft zu drei markierten Dreiecken liegen. Diese letzten Dreiecke können nun ebenfalls markiert werden, und von den drei markierten Kanten kann anschließend eine beliebige Kantenmarkierung entfernt werden.

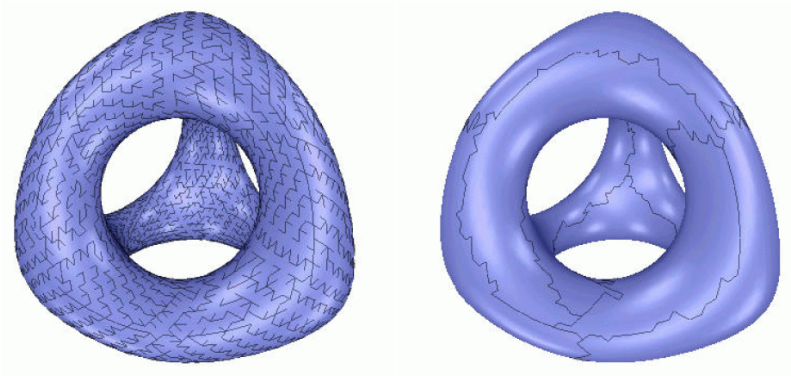

Abbildung 2.1: Links: Das Resultat des *region growing* Algorithmus. Rechts: Überarbeitetes Resultat, bei dem überschüssige Kantenbäume entfernt wurden.

Das Resultat des *region growing* Algorithmus ist nun eine Menge von (markierten) Kanten, die den topologischen Schnitt bilden und eventuell wie oben beschrieben einer Nachbearbeitung bedürfen, da sie aus Kantenschleifen und unnötigen Kantenbäumen bestehen. Um letztere zu entfernen, wird nun mit den markierten Knoten gearbeitet.
Hierbei werden zuerst alle Markierungen von Knoten aufgehoben, die nicht zu einer markierten Kante benachbart sind. Anschließend werden die Markierungen von Knoten entfernt, die inzident zu nur einer markierten Kante sind, sowie die Markierung dieser Kante. Dieser Schritt entfernt iterativ die unnötigen Kantenbäume von den Ästen ausgehend, bis nur noch verbundene Schleifen übrigbleiben. Der so erhaltene topologische Schnitt ist in den meisten Fällen jedoch noch sehr irregulär, was sich bei der Texturierung durch deutliche Diskontinuitäten bemerkbar macht.
Um die unnötige Länge zu kürzen und fehlende Glätte einzuführen, wird der topologische Schnitt in einzelne Schnittpfade zerlegt, wobei Start- und Endknoten dieser Pfade stets Schnittknoten sind, das heißt, die Knoten, die zu mehr als zwei Kanten des topologischen Schnitts gehören. Um die einzelnen Schnittpfade zu glätten, wird mithilfe eines leicht modifizierten Dijkstra-Algorithmus der kürzeste Weg zwischen den zwei Schnittknoten mit der Einschränkung gesucht, dass der neue Pfad keinen anderen Pfad schneidet. Somit ist das Suchfeld des neuen Pfades durch die umliegenden Schnittpfade eingegrenzt und effizient lösbar. Dabei muss berücksichtigt werden, dass die Optimierung eines Schnittpfades den Suchraum der benachbarten Pfade modifiziert, sodass diese wiederum auf eine mögliche Verbesserung untersucht werden müssen, was zu einer iterativen Optimierung der Pfade führt, bis ein stabiles Resultat erzielt wird.

Gerade bei Modellen mit mehreren Komponenten erweist sich dieser Single-Chart Ansatz jedoch als ungeeignet, da der topologische Schnitt sehr komplex ausfallen und zu übermäßige Streckungen und Verzerrungen führen kann, die im Postprocessing durch zusätzliche Schnitte gelöst werden müssen. Es liegt daher nahe, das Modell nicht in einem einzigen Chart, sondern in mehrere zu unterteilen und diese einzeln zu behandeln. Im Folgenden sollen zwei Segmentierungsalgorithmen veranschaulicht vorgestellt werden, die sich dieser Methodik bedienen.

2.2 Least Square Conformal Maps Segmentierungs-Algorithmus

Dieses Segmentierungsverfahren stellt einen Teil eines größeren Parametrisierungs-Algorithmus dar, wie er in [Lé02] vorgestellt wird. Ausgehend von den gefundenen Features auf der 3D-Oberfläche, zerlegt der Segmentierungs-Algorithmus jenes Modell in ein Set von Charts, sodass zwei maßgebende Kriterien erfüllt werden:

1. Die Grenzen der Charts müssen so positioniert sein, dass die meisten Diskontinuitäten zwischen den einzelnen Charts in Bereichen liegen, in denen sie keine Texturartefakte erzeugen oder diese kaum wahrnehmbar sind.

2. Die einzelnen Charts müssen homöomorph zu einer Scheibe sein und es muss danach möglich sein, sie ohne zu großer Deformation parametrisieren zu können.

Zum ersten Punkt fanden [Lé02] wie auch [She02] heraus, dass es zur Minimierung der Artefakte vorteilhaft ist, Chartgrenzen in Bereichen mit hoher Gaußscher Krümmung anzulegen. Grund dafür ist, dass Licht- und Schattenberechnungen unter anderem auf den Normalen des Modells beruhen und ausgeführt werden, was zur Erzeugung besonders scharfer Veränderungen des Lichts in Bereichen hoher Krümmung führt. Gerade aber in diesen Bereichen werden Texturartefakte durch den Mensch deutlich weniger stark wahrgenommen, da diese im Vergleich zu korrekten Schatteninformationen und den starken Lichtveränderungen weniger auffallen.
Dem zweiten Punkt widmeten sich Lévy et al., indem sie ein automatisches Verfahren konstruierten, dass das manuelle Segmentieren eine Benutzers nachahmt. Der virtuelle Benutzer versucht dabei, das Eingabenetz möglichst in Körper aufzuteilen, die einem Zylinder ähneln, da die so gewählte Fläche stets entwickelbar ist, das heißt, dass sie sich ohne innere Formverzerrung in die euklidische Ebene transformieren lässt. Der Segmentierungsalgorithmus teilt sich dabei in drei Phasen auf:
In einem ersten Schritt werden die Features des Eingabenetzes ermittelt. Ein *mesh feature* ist dabei eine stückweise lineare Kurve, die die wichtigen Charakteristiken des Eingabemesh beschreibt. In diesem Fall wird ein Mesh-Feature auf die Kanten, durch die es repräsentiert wird, beschränkt und bei der Berechnung der Features wird das Schärfekriterium nur an den Kanten angewendet. Wie in [Hü01] wird hier das Krümmungsverhalten, genauer die Winkel zwischen den Normalen der an der Kante

anliegenden Flächen, berechnet und ausgewertet. Diese *second order differences* beschreiben [Hü01] durch ein Gewicht $w(e)$ für eine Kante e wie folgt:

$$w(e) = cos\left(\frac{n_i}{\|n_i\|} \cdot \frac{n_j}{\|n_j\|}\right)^{-1}$$

Von den so gefundenen Kanten werden nur ein gewisser Teil beibehalten. Die Anzahl ist abhängig von dem vorher festgelegten Grenzwert τ. Für jede übriggebliebene Kante wird nun die Featurekurve ermittelt. Dazu wird eine Tiefensuche von den Kanten aus gestartet, die in jedem neuen Tiefenabschnitt weitere Kanten zur der Menge der potentiellen Featuremenge hinzufügt, solange die Summe der Schärfe der einzelnen Kanten, also der Winkel zwischen den Normalen der Flächen zu beiden Seiten einer Kante, über einen festgelegten Schwellwert liegt. Ist die so ermittelte Featurekurve länger als die frei gewählte Mindestlänge, werden alle so herausgefilterten Kanten als Feature markiert. Zusätzlich werden die Nachbarkanten als *feature neighbours* markiert, sodass diese bei weiteren Suchen nicht ebenfalls als Feature erkannt werden.

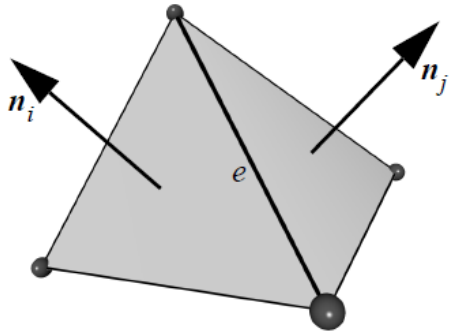

Abbildung 2.2: Berechnung einer Feature-Kante mittels *second order differences*: Der Winkel zwischen den Normalen n_i und n_j beschreibt die Krümmung der an der Kante anliegende Dreiecksflächen.

Nach der Featuredetektierung können die Charts in einem zweiten Schritt erstellt werden. Ausgehend von einer Menge an *Seeds*, den Startpolygonen, werden die einzelnen Charts simultan entwickelt, bis sich alle Dreiecksoberflächen des Meshes genau in einem Chart befinden. Damit die einzelnen Chartgrenzen auch wirklich wie oben angesprochen auf den vorher herausgefilterten Features liegen, benutzen [Lé02] einen modifizierten Dijkstra-Algorithmus. Hierzu werden von den Rändern und den detektierten Features ausgehend für jedes Dreieck die Entfernung zu jedem Feature berechnet. Die Seeds für die zu entwickelnden Charts befinden sich dann in lokalen Maxima dieser Distanzfunktion, also möglichst zentral innerhalb der umliegenden Features. Sollte es dadurch vorkommen, dass zwei Seeds in unmittelbarer Nähe liegen, sodass deren Grenzen sich innerhalb einer festgelegten ε-Umgebung um den Seed herum treffen, werden beide Charts zu einem einzigen neuen Chart verschmolzen

und die neue Seedposition aus den ursprünglichen Positionen gemittelt.
In einem letzten Schritt werden die so erzeugten Charts auf ihre Gültigkeit überprüft. [Lé02] stellten fest, dass eine Überlappung durch eine Selbstüberschneidung an den Grenzen der Charts durchaus möglich ist, wie es oben im Single-Chart Ansatz erklärt wurde. Durch einen Stencil-Test zum Beispiel lassen sich jedoch solche Überschneidungen erkennen und anschließend beseitigen: Stencil-Pixel, die doppelt gezeichnet werden, deuten dabei auf mögliche Überlappungen hin und führen dazu, dass in einem solchen Fall der betroffene Chart entlang der Kanten der betroffenen Region zerschnitten und als neuer Chart unterteilt und gehandhabt wird.

2.3 D-Chart Algorithmus

Anders als der oben beschrieben Algorithmus versucht der D-Chart Algorithmus nicht, möglichst entwickelbare Charts über die vorher gefundenen Features zu bestimmen, sondern baut die einzelnen Regionen während der Wachstumsphase eigenständig auf und verschiebt oder erzeugt gegebenenfalls komplette Chartzentren. Um ein optimales Ergebnis zu erzielen, besitzt er sogar eine Nachbearbeitungsphase, die eventuell zu komplexe Ränder vereinfacht oder entstandene Überlappungen aufhebt.

D-Charts Übersicht

Der D-Charts Algorithmus von [Jul05] benutzt einen Regionenwachstumsansatz mit einem iterativen Lloyd-Schema. Der Lloyd-Algorithmus beginnt mit der Verteilung der Set-Schwerpunkte der vorher festgelegten k-verschieden Anfangssets, wobei die Positionierung zufällig oder über vorgegebene heuristisches Verfahren erfolgen kann. Darauf hin werden die Vertizes dem Set zugeordnet, zu dessen Schwerpunkt sie sich am nächsten befinden. Nach dieser ersten Zuordnung der Punkte in die Sets wird für jedes Set mittels einer Metrik (zum Beispiel die Distanz aller im Set befindlichen Vertizes zueinander) der eigene Set-Schwerpunkt, das sogenannte Baryzentrum, erneut ermittelt, was zu Verschiebung der Barizentren führt. Anschließend erfolgt eine neue Neuverteilung der Vertizes in die einzelnen Sets, wobei Vertizes jetzt wiederum in das Set des Baryzentrum fallen, zu dem sie am nächsten gelegen sind, was dazu führen kann, dass Vertizes ihr vorher bestimmtes Set wechseln. Mit jeder Iteration verteilen sich die Zentren gleichmäßiger über die Oberfläche, sodass sich die einzelnen Sets in Form und Größe annähern. Die Iteration selbst wird solange fortgesetzt, bis eine Konvergenz erreicht wird, das heißt, die Vertizes nicht mehr ihre zugewiesene Gruppe wechseln, die Schwerpunkte der Sets sich nicht mehr verändern oder eine vorher festgelegt Iterationstiefe erreicht wurde. Als Folge dieses Algorithmus erhält man ein Voronoi-Diagramm der Vertizes, die nun in Verbünde eingeteilt sind, die eine minimal geodätische Distanz zu ihren Set-Zentren besitzen und so eine günstige Ausgangslage bei der Erzeugung der Charts bilden.

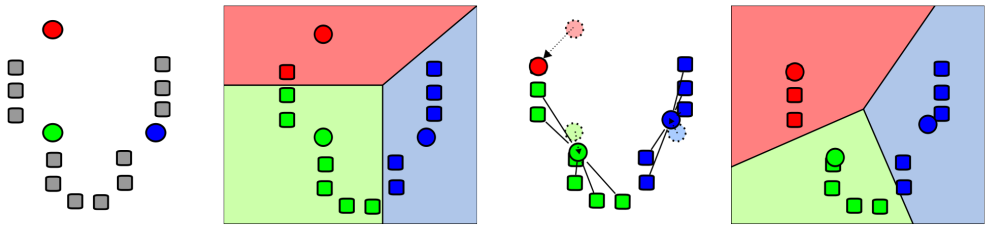

Abbildung 2.3: beispielhafter Durchlauf des Lloyd-Algorithmus mit $k = 3$ Sets: Die Set-Zentren (Kreise) werden zufällig auf der Oberfläche verteilt. Anschließend werden die Vertizes (Rechtecke) dem Set mit dem ihnen am nächsten liegenden Zentrum zugeordnet. Das neue Set-Zentrum wird bestimmt und führt zur Verschiebung der alten Zentren. Schließlich werden die Vertizes erneut in die drei Sets eingeteilt.

In dem D-Chart Algorithmus wird der Lloyd-Algorithmus so verändert, dass die Anzahl der Sets nur anfänglich fest vorgegeben wird. Darüber hinaus wird ein Fehlergrenzwert F_{max} eingeführt. Dieser Fehlergrenzwert legt fest, ob ein Dreieck zu einem bestehenden Chart hinzugefügt werden kann oder ob ein neues Set, das heißt ein neuer Chart, erstellt werden muss. Auch beim späteren Zusammenlegen zweier Charts gibt F_{max} an, ob dies möglich ist oder nicht. Somit bestimmt der Fehlergrenzwert direkt die Anzahl der Charts, die je nach Bedarf und Vereinbarkeit steigen oder fallen können.
Unter Berücksichtigung dieser Konzepte soll im folgenden der Aufbau des Algorithmus genauer betrachtet werden. Insgesamt ist dieser in drei Schritte aufgebaut, die wie folgt definiert sind:

- **Modifizierte Lloyd Iterationen:** Die durch die Segmentierung zu erstellenden Charts wachsen aus sogenannten Seeds während der Lloyd-Iteration heran, ohne dabei den vorher festgelegt Fehlergrenzwert überschreiten zu dürfen. Dies soll garantieren, dass die einzelnen Charts entwickelbar bleiben. Dreiecke, die den Fehlergrenzwert überschreiten, werden nicht zu den wachsenden Charts hinzugefügt.

- **Schließen der Löcher:** Fehlende Dreiecke, die in der ersten Phase keinem Chart zugeordnet wurden, werden nun nochmal evaluiert. Mit Rücksicht auf den erzeugenden Fehlerwert, den ein Einfügen der Dreiecke in einen Chart nach sich trägt, können sie bereits bestehenden Charts zugeordnet werden oder bilden einen gänzlich neuen, eigenen Chart.

- **Nachbearbeitung:** Im letzten Schritt werden die so entstandenen Charts weiter verbessert. Komplexe Ränder der Charts werden begradigt, ohne den Fehlerwert zu erhöhen, und benachbarte Charts werden zusammengefügt, falls das Ergebnis entwickelbar bleibt. Außerdem werden in Bereichen mit hohem Fehlerwert Einschnitte vorgenommen.

Bevor es nun jedoch zum eigentlichen Algorithmus kommt, lohnt es sich, die Begriffe des Proxy, der nun häufig genannten Entwickelbarkeit und der verwendeten Kostenfunktionen zu klären.
Das anvisierte Ziel der Segmentierung stellt die Einteilung des Meshes in (fast) entwickelbare Charts dar. Diese Bedingung alleine jedoch reicht für eine sinnvolle Segmentierung nicht aus: Ein Dreiecksstreifen ist per Definition entwickelbar. Die Einteilung eines beliebigen Meshes in wenige, dafür sehr lange Dreiecksstreifen würde demnach die Voraussetzung für eine entwickelbare Segmentierung vollends erfüllen, jedoch wäre das Resultat offensichtlich mit seinen sehr langen Kanten eher unhandlich für weitere Verarbeitungen. Die Herausforderung besteht also darin, diese einzelnen Charts möglichst kompakt zu kreieren, ohne dass diese dadurch ihre Eigenschaften bezüglich ihrer Entwickelbarkeit verlieren. Doch wann genau ist ein dreidimensionaler Körper überhaupt entwickelbar?
Eine Standarddefinition einer entwickelbaren Oberfläche ist, dass diese an allen Punkten eine Gaußsche Krümmung von Null besitzt. [Jul05] merken an, dass diese Definition für eine nützliche Segmentierung eher ungeeignet ist und schränken in Folge dessen den Erkennungsmechanismus auf eine Untermenge der entwickelbaren Oberflächen ein, und zwar auf die der Vereinigung von einachsigen Kegel. Die Idee dahinter ist, dass sich eine Oberfläche als eine Vereinigung von Kegeln mit einer ausgerichteten Achse und einem Öffnungswinkel darstellen lässt. Ein Chart repräsentiert die Grundfläche eines solchen Kegels, wenn der Winkel zwischen der Normalen der Oberflächen und einer gemeinsamen Achse, die der Achse des Kegels entspricht, in jedem Punkt konstant ist. Diese Winkelbedingung bildet die Basis des D-Chart Segmentierungsprozesses. [Jul05] definieren ein Proxy für jeden Chart C als das Paar $< N_C, \Theta_C >$, wobei N_C der Einheitsvektor ist, der die Achse des Kegel repräsentiert, und Θ_C der konstante Öffnungswinkel dieses Kegels ist. Um zu erkennen, wie gut ein gegebenen Dreieck t mit der Normale n_t in einen gegebenen Chart C passt, definieren [Jul05] den *fitting error* als quadratische Abweichung der Dreiecksnormalen zur Kegelaxe wie folgt:

$$F(\mathcal{C},t) = (N_C \cdot n_t - cos\Theta_C)^2$$

Des Weiteren werden zwei weitere Metriken eingeführt, die daraufhin zielen, einerseits sehr runde und kompakte Charts und andererseits sehr gerade und einfache Randkurven zu erzeugen. Beide dienen dazu, möglichst nur geeignete Dreiecke zum Chart hinzuzufügen, ohne dass die Gesamteigenschaften des Charts darunter leiden. Die Kompaktheit wird dabei mittels

$$C(C,t) = \pi \frac{D(S_C,t)^2}{A_C}$$

erfasst, wobei S_C das Startdreieck (Seed) des Charts ist, $D(S_C,t)$ die Länge des kürzesten Weges innerhalb des Charts vom Seed zum aktuellen Dreieck und A_C

die Fläche des Charts C. Für Dreiecke, die so auf dem Rand eines Kreises fallen würden, der ideal kompakt ist, ergibt diese Metrik 1 und wird kleiner, je weiter sich das aktuelle Dreieck von dieser Idealposition entfernt. Um möglichst gerade Ränder bei den Charts zu erzeugen, wird darüber hinaus das Verhältnis zwischen der Länge aller Kanten des aktuellen Dreiecks, die sich innerhalb des Kreises befinden, den der Chart aufspannt, zu der Länge der Kanten, die sich außerhalb dieses Kreises befinden, gemessen und ausgewertet:

$$P(C,t) = \frac{l_{outer}(C,t)}{l_{inner}(C,t)}$$

Letztendlich bilden diese drei Metriken eine Kostenfunktion, die angeben, wie 'teuer' das Hinzufügen des aktuellen Dreiecks zu einem Chart ist:

$$Cost(C,t) = F(C,t)^{\alpha} C(C,t)^{\beta} P(C,t)^{\gamma}$$

Da die Begriffe des Proxy, der Entwickelbarkeit und der Kostenfunktion nun hinreichend geklärt sind, soll sich nun dem eigentlich D-Chart Algorithmus gewidmet werden.

2.3.1 Modifizierte Lloyd Iterationen

Ausgehend von dem oben genannten drei Schritten werden dem Algorithmus in der Initialisierungsphase die Anzahl der erwünschten Charts sowie deren Seeds als Eingabeparameter übergeben. In der Theorie ist es möglich, jedes beliebige Dreieck als Seed auszuwählen, doch empirisch hat es sich gezeigt, dass es im Sinne besserer Resultate und einer schnelleren Konvergenz ist, wenn die Startdreiecke so gewählt werden, dass sie maximal weit voneinander entfernt liegen. Für jeden Seed wird dann der oben beschriebene optimale Proxy errechnet und angepasst. Dieser besteht aus einem normalisierten Achsenvektor und den dazugehörigen Vektor, der die gewichteten Fitting-Error zwischen dem imaginären Kegel, der durch den Chart dargestellt werden soll, und den tatsächlichen Dreiecken des Charts minimiert. Dazu muss mithilfe des Newton-Verfahrens das Constraints-Optimierungsproblem

$$\min_{N_c,\Theta_C} \frac{1}{A_C} \sum A_t F(C,t)$$

gelöst werden, wobei hier A_t die Fläche des Dreiecks t ist und $F(C,t)$ der Fitting-Error wie oben beschrieben. Nachdem der Proxy für einen Chart errechnet wurde, wird gegebenfalls ein neuer Seed für den entsprechenden Chart benötigt, da die Achse des neu errechneten Proxies eventuell nicht mehr mit der Dreiecksnormale übereinstimmt. Das neue Seed-Dreieck sollte immer einen möglichst geringen Fitting-Error bezüglich des Proxies besitzen und so zentral wie möglich im zu bildenden Chart liegen. Hierfür werden die ersten k Dreiecke des Charts mit minimalen Fitting-Error

untersucht und das dem Zentrum am nächsten liegende als neuer Seed ausgewählt. In der darauf folgenden Wachstumsphase werden den Charts neue Dreiecke des Meshes hinzugefügt. Beginnend mit nur einem Dreieck, dem Seed, werden die drei benachbarten Dreiecke in einer *priority queue* eingefügt. Für diese Dreiecke wird bestimmt, zu welchen Charts sie benachbart sind und wie hoch die Kostenfunktion wäre, sie in den entsprechenden Chart einzufügen. Während die Prioritätswarteschlange nicht leer ist, wird anschließend das Dreieck mit den niedrigsten Kosten aus dieser entfernt und zum entsprechenden Chart hinzugefügt, solange

1. das Dreieck noch keinem Chart zugewiesen wurde, was passieren kann, wenn es mehrere Einträge in der Liste der benachbarten Charts besitzt, und

2. der *fitting error* des Dreiecks mit dem Chart unter einem festgelegten Grenzwert $F_{\max}$ liegt, damit der Chart entwickelbar bleibt.

Erfüllt das Dreieck nicht diese zwei Bedingungen wird es übersprungen. Andernfalls werden die drei benachbarten Dreiecke des nun eingefügten Dreieckes ermittelt und ebenfalls in die Warteschlange gestellt. Der Prozess gilt als beendet, wenn die Warteschlange abgearbeitet wurde. Anschließend werden die Proxies neu berechnet und die Wachstumsphase erneut gestartet. Dies wiederholt sich, bis nur noch ein kleiner Prozentsatz (unter 5%) an Dreiecken in einer Wachstumsphase von einem Chart in einen anderen überführt wurde. Da die Beendigung der Wachstumsphase auf diese Weise nicht garantiert ist, führt man darüber hinaus noch eine Beschränkung der Anzahl an Iterationen ein, nach denen man einfach Konvergenz annimmt.

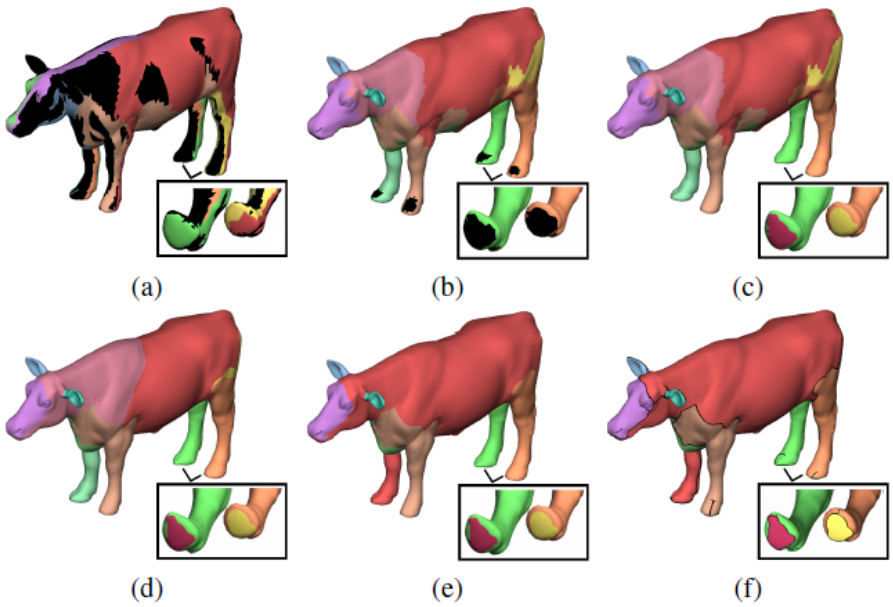

Abbildung 2.4: verschiedene Stadien des D-Chart Algorithmus: (a)Wachstum der Charts nach einer Iteration. (b) Charts nach letzter Wachstumsphase. (c)Schließen der übriggebliebenen Löcher. (d) Begradigung der Randkurven der einzelnen Charts. (e) Vereinigung von benachbarten Charts. (f)Zusätzliches Einfügen von Einschnitten.

2.3.2 Schließen der Löcher

Nachdem die Lloyd Iterationen annähernd Konvergenz erlangt haben, ist ein Großteil der Dreiecke genau einem Chart zugeordnet. Durch den fehlergebundenen Wert $F_{\max}$ kann es jedoch vorkommen, dass es immer noch Dreiecke gibt, die keinem Chart zugehörig sind. Diese Dreiecke werden nun gesammelt und kategorisiert, je nachdem ob sie ein größeres oder ein eher kleines Loch innerhalb der Segmentierung erzeugen. Kleine Löcher werden gefüllt, indem die umliegenden Charts in die Löcher expandieren. Der so eingeführte Fehler wird später in der Nachbearbeitung durch zusätzliche Schnitte wieder ausgeglichen.
Bei großen Löchern dagegen wird ein neuer Chart erzeugt. Alle Dreiecke innerhalb des Loches werden diesem neuen Chart zugewiesen. Anschließend wird wie oben beschrieben der optimale Proxy und der Seed des Charts errechnet. Von dem so errechneten Startdreieck aus wird der Chart noch einmal „richtig“ erzeugt, um das Loch auszufüllen. Auf diese Art kann es vorkommen, dass neue Löcher innerhalb des so entstehenden Charts vorkommen, die rekursiv wie gerade beschrieben, behandelt werden. Diese auf den ersten Blick eher komplizierte Variante zur Erzeugung neuer Charts birgt den Vorteil, dass auf diese Weise bessere Resultate bei der Erzeugung der Proxies in der Iterationsphase zustande kommen. Das Gesamtergebnis wird bei der Erzeugung erheblich optimiert und der Wachstumsprozess für die Löcher

füllenden Charts beschleunigt, was sich gerade in Bereichen mit hoher Gaußscher Krümmung für das Wachstum von Charts positiv auswirkt: Die Krümmung sorgt für hohe Fehlerwerte der zu bearbeitenden Dreiecke, was folglich zum Stoppen des Wachstums des gerade aktuellen Charts nach wenigen Iterationen und zur Generierung eines neuen Charts führt. Die Folge sind ausschweifend viele und besonders kleine Charts. Durch die oben beschriebene Zuweisung aller Dreiecke innerhalb eines Loches zu einem einzigen Chart wird dies unterbunden.

2.3.3 Nachbearbeitung

Im letzten Schritt des D-Chart Algorithmus werden drei große Operationen durchgeführt: Es werden die Ränder der Charts begradigt, benachbarte Charts zu einem gemeinsamen Chart zusammengefügt, sofern die Bedingungen zur Entwickelbarkeit erhalten bleiben, und zum Schluss partielle Schnitte in das Mesh eingefügt.

Ränder begradigen: Nach erreichter Konvergenz in der Wachstumsphase sind die Ränder der Charts oft äußerst komplex und gezackt, besonders in Bereichen, in denen der Fitting-Error der Dreiecke und benachbarten Charts sehr ähnlich sind. Grund dafür ist, dass die Dreiecke in diesen Regionen trotz gleicher Fehlerwerte zu den Seeds der benachbarten Charts, demjenigen Chart zugeordnet werden, zu dem sie sich am nächsten befinden, wobei die Zuteilung in Bereichen gleicher Distanz sogar zufällig erfolgen kann. Um die Segmentierung diesbezüglich auszubessern, werden genau diese *Fuzzy-Regionen* gesucht: Dazu werden beide Charts virtuell weiter ineinander expandiert. Es existiert nun ein Bereich zwischen zwei Charts, in dem die Dreiecke virtuell zu beiden Charts gehören. Anschließend werden zwei Endpunkte, das heißt zwei Vertizes, die auf dem Rand dieser Fuzzy-Region maximal weit auseinander liegen, gesucht und mittels des kürzesten Weges innerhalb dieser Region verbunden. Dieses neue Segment stellt nun die neue Grenze zwischen den Charts dar und sollte erheblich glatter sein.

Charts zusammenfügen: Die Anzahl der Charts kann weiter reduziert werden, indem benachbarte Charts miteinander verbunden werden. Dazu muss der daraus resultierende Chart natürlich weiterhin entwickelbar bleiben. Dies ist der Fall, wenn die Gaußsche Krümmung am gemeinsamen Rand der beiden Charts Null ergibt. Ein einfacher Test ist es, die gemeinsame Randregion B ebenfalls als Chart zu behandeln, einen Proxy der Form $< N_B, \Theta_B = \pi/2 >$ zu erzeugen und zu errechnen, ob die gemittelte Summe der gewichteten Fitting-Errors der einzelnen Dreiecke

$$\frac{1}{A_b}\sum_{t\in B} A_t F(B,t) < \nu$$

unter dem vorher festgelegten Grenzwert ν liegt. Im positiven Fall können die beiden Charts so vereint werden.

Schnitte einfügen: Wie oben beschrieben, können beim Schließen der Löcher Dreiecke zu Charts hinzugefügt werden, die Regionen mit hohem Fitting-Error

erzeugen. Diese Regionen wiederum stellen ein Problem bei der späteren Parameterisierung dar, da sie eine hohe Verzerrung oder Belastung in den umliegenden Regionen verursachen. Um die Belastung zu minimieren, werden partielle Schnitte ausgehend von den Rändern hin zur Region mit hohem Fitting-Error eingefügt. Jede solche Naht reduziert die Belastung in einem kreisförmigen Gebiet in diesem Bereich mit der Spitze des Schnittes als Mittelpunkt und einem Radius, der der Länge des Einschnittes entspricht. Dies wiederum bedeutet aber, dass gerade großflächige kritische Regionen ebenfalls sehr lange Einschnitte benötigen, um die Spannungen zu entfernen, und dass, wie oben veranschaulicht, die dadurch resultierende Textur-Bruchkante deutlich sichtbarer wird.

2.4 Schwerpunktausarbeitung

Als ergänzende Ausarbeitung liegt dieser Arbeit ein Programm mit eigenem Segmentierungsalgorithmus bei, das über ein graphisches Interface in Form einer HTML-Seite angesteuert werden kann und hier kurz angesprochen werden soll. Eine genaue Erklärung sowie grundlegende Abläufe werden in Kapitel 5 behandelt.
In seiner ursprünglichen Ausführung beinhaltete das Programm zwei Multichart-Ansätze, die im Programm unter **Simple Chart Creation** und **Simple Chart Creation**$^+$ vorgefunden werden konnten. Die Idee des SCC-Algorithmus ist dabei folgende: Um ein gegebenes Mesh wird ein Hüllkörper gelegt, dessen Oberflächen eine Menge von Normalen darstellt. Um das Eingabenetz nun in verschiedene Charts aufzuteilen, wird für jedes Dreieck auf dem Netz die Normale dieses Dreiecks berechnet. Das Programm nutzt dazu das normierte Kreuzprodukt zweier beliebiger Vektoren, die sich aus den drei Vertizes des Dreiecks bilden lassen. Anschließend wird die gewonnene Normale mit allen Normalen des Hüllkörpers verglichen. Im Programm wird der Hüllkörper als ein vierzehnseitiges Objekt angenommen, dessen Flächen durch seine vierzehn Normalen, die jeweils eine eigene Färbung besitzen, repräsentiert werden. Beim Vergleich der Dreiecksnormale mit den Hüllkörper-Normalen wird diejenige Hüllkörper-Normale ausgewählt, bei der die Winkeldifferenz zwischen beiden Normalen am geringsten ist. Die so ausgewählte Normale des Hüllkörpers bestimmt anschließend die Färbung und damit den Chart, zu dem das Dreieck hinzugefügt wird. Unter der Voraussetzung, dass Dreiecksnormale und Hüllkörpernormale beide normalisiert sind, berechnet der SCC-Algorithmus den Innenwinkel wie folgt:

$$
\begin{aligned}
|\vec{n}_1| \cdot |\vec{n}_2| \cdot \cos(\gamma) &= \vec{n}_1 \cdot \vec{n}_2 \\
\gamma &= \arccos\left(\frac{\vec{n}_1 \cdot \vec{n}_2}{|\vec{n}_1| \cdot |\vec{n}_2|}\right) \\
\gamma &= \arccos(\vec{n}_1 \cdot \vec{n}_2)
\end{aligned}
$$

Dies entspricht auch der Kantengewichtung zur Featuredetektion, mithilfe der *second order differences* beschrieben wurde. Nach der Iteration über alle Oberflächendreie-

cke des Eingabenetzes ist dieses anschließend in mehrere Charts unterschiedlicher Färbung unterteilt. Folgendes Bild zeigt den SCC-Algorithmus bei der Ausführung auf dem Mesh *horse.off*:

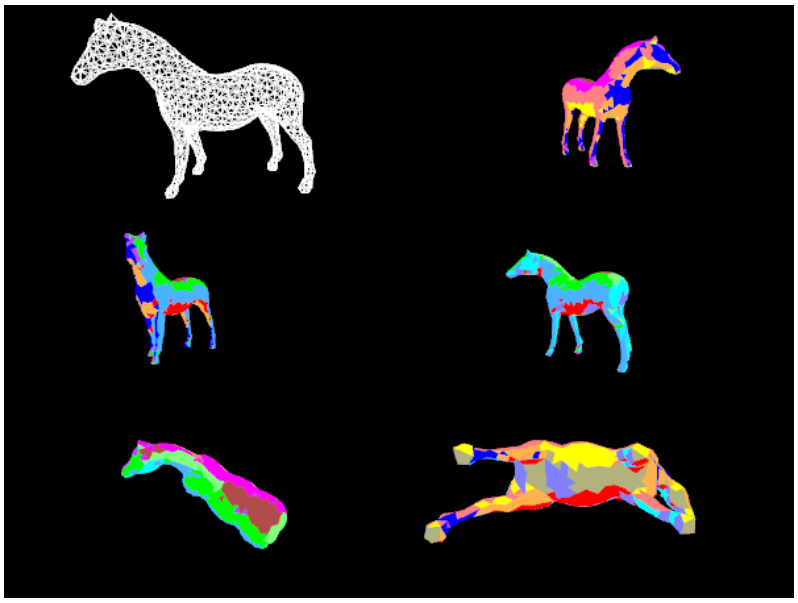

Abbildung 2.5: Horse-Mesh vor (weiß) und nach der Anwendung des SCC-Algorithmus. Deutlich zu sehen: Kantige Chartgrenzen und starke Schwankungen in Größe der einzelnen Charts.

Deutlich zu sehen sind dabei die Schwachpunkte des Algorithmus: Durch eine fehlende Größenkontrolle ist es praktisch möglich, Charts zu erzeugen, die nur aus sehr wenigen Dreiecken bis hin zu einem einzigen Dreieck bestehen. Dies tritt vor allem in Bereichen mit Krümmungsschwankungen auf; ein typisches Problem des *region growin*-Algorithmus. Ein weiterer Negativpunkt, der mit dem ersten einhergeht, ist die Anzahl der erzeugten Charts. Da die erlaubte Größe eines Charts keine Rolle spielt, werden zuweilen unnötig viele Charts erzeugt. Des Weiteren findet keine Nachbearbeitung der Charts statt, sodass deren Ränder starke Verzackungen aufweisen können und teilweise tief in andere Charts *hineinschneiden*.

Mesh	Charts	min. Chart Size	max. Chart Size	average Chart Size
horse.off	203	1	162	7
torus.off	30	1	438	80
teapot.off	142	1	120	10

Tabelle 2.1: Statistiken des SCC-Algorithmus angewandt auf verschiedene Eingabenetze

Dieser naive Ansatz zur Generierung einer erfolgreichen Segmentierung zeigt bereits erste Problemstellungen, die es im Verlauf dieser Studienarbeit zu lösen galt. Die

fehlende Festlegung der Anzahl der zu erzeugenden Charts sowie deren Mindest- und Maximalgröße sind entscheidenden Informationen, die es zu berücksichtigen gilt. Ein ebenso großes Manko stellt die konstante Anzahl an Hüllkörper-Normalen da, die zwar einerseits für reproduzierbare Ergebnisse sorgt, jedoch die Topologie des Eingabenetzes kaum bis gar nicht berücksichtigt und so nur mangelhafte Ergebnisse erzielt.
Das fertige Programm, das dieses Probleme berücksichtigt wird in Kapitel 5 genauer erläutert.

KAPITEL 3

Parametrisierung

Nach erfolgreicher Segmentierung des Eingabenetzes in einzelne Charts stellt sich nun die Frage, wie die einzelnen 3D-Oberflächenstücke, dass heißt die Positionen der Vertizes, in eine Ebene eingebettet werden können. Die einfachste Antwort zur Ermittlung der passenden 2D-Koordinaten für jeden Vertex des Netzes erhält man durch die lotrechte Projektion der 3D-Oberflächestücke auf eine Ebene. Der große Nachteil bei einem solchen Verfahren ist, dass die topologische Eigenschaften der 3D-Oberfläche bei dieser Art der Projektion komplett verloren gehen und darüber hinaus Fehler in Form von Streckungen und Verzerrungen eingeführt werden, die bei der Rücktransformationen, also dem Texture Mapping, deutlich werden.
Um eine Lösung auf diese Fragestellung zu finden, soll im Folgenden erst einmal eine günstige Definition gegeben werden, stets unter der Perspektive, die oben genannte Probleme zu überwinden oder, sofern dies nicht möglich, zu minimieren. Es werden verschiedene lokale Verfahren für Objekte mit Scheiben-Topologie und globale Methoden vorgestellt, die die Überführung einer beliebigen Fläche in den Parameterraum ermöglichen.
Viele dieser Methoden kommen im Zuge des Texture Mappings zum Tragen, beschreiben jedoch in der gängigen Literatur häufig eine Abbildung vom 2D-Raum in den 3D-Raum. Da durch die Kameraaufnahmen bereits ein 3D-Körper samt Textur vorliegt und die Textur ab diesem Zeitpunkt extrahiert werden muss, ist diese Funktionsrichtung eher unerwünscht. Stattdessen wird genau nach der inversen Funktion gefragt, also der Abbildung des dreidimensionalen Raumes auf eine Ebene. Bei dieser inversen Funktion handelt es sich jedoch auch um nichts Weiteres als eine Parametrisierung, sodass diese als ebensolche benannt und gehandhabt wird. Der Übersichtlichkeit halber wird der Leser aber explizit darauf hingewiesen, in welche Richtung die entsprechende Funktion operiert, um Verwirrung zu vermeiden.

3.1 Definition und Aufbau

Eine kurze und geläufige Definition einer Parametrisierung findet sich in [Flo01] und lautet wie folgt:

> Eine parametrische Oberfläche ist durch ein eins-zu-eins Mapping $\phi : \Omega \to \mathbb{R}^3$, mit der Parameterdomäne $\Omega \in \mathbb{R}^2$, definiert und wir nennen ϕ eine *Parametrisierung* dieser Oberfläche.

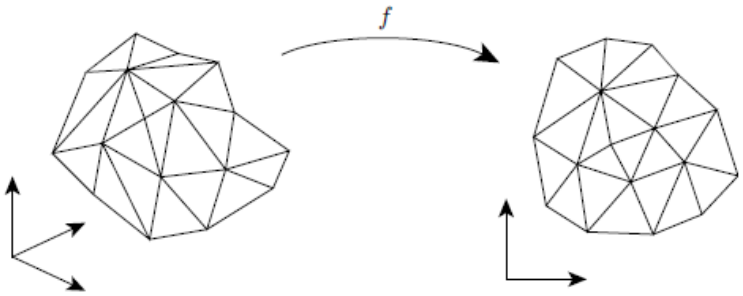

Abbildung 3.1: stückweises lineares Mapping eines triangulierten Meshes.

In der Computergraphik generell und in dieser Arbeit speziell geht man genau von der inversen Funktion aus. Man ist in Besitz einer Menge an Punkten $\boldsymbol{x}_i = (x_i, y_i, z_i)$, die aus der Abtastung einer Oberfläche gewonnen wurden, wobei das Resultat der Abtastung so komplex ist, dass es sich nicht einfach als Graph einer bivariaten Funktion der Form $f(x,y)$ darstellen lässt. Wählt man eine geeignete Domäne Ω und geeignete Parameterpunkte $\boldsymbol{u}_i = (u_i, v_i)$ in Ω, dann ist das unabhängige Konstruieren drei einzelner Funktionen $s_1, s_2, s_3 : \Omega \to \mathbb{R}$ ein valide, wenn auch einfache Lösung, wenn gilt:

$$s_1(\boldsymbol{u}_i) \approx x_i,$$
$$s_2(\boldsymbol{u}_i) \approx y_i,$$
$$s_3(\boldsymbol{u}_i) \approx z_i$$

Setzt man dann

$$\boldsymbol{s}(u,v) = (s_1(u,v), s_2(u,v), s_3(u,v)),$$

so erhält man

$$\boldsymbol{s}(u_i, v_i) \approx (x_i, y_i, z_i).$$

Als Folge kann das Problem der Parametrisierung auf das Finden passender Parameterpunkte $\boldsymbol{u}_i$, von den Datenpunkten $\boldsymbol{x}_i$ ausgehend, reduziert werden. Ein solche Repräsentation wird regulär genannt, wenn die Funktionen s_1, s_2, s_3 glatt sind, also so oft wie nötig differenzierbar sind und die Vektoren

$$\boldsymbol{s}_1 = \frac{\partial \boldsymbol{s}}{\partial u},$$

$$\boldsymbol{s}_2 = \frac{\partial \boldsymbol{s}}{\partial v}$$

an jedem Punkt linear unabhängig sind, das heißt, ihr Kreuzprodukt nicht Null ist. Eine wichtige Tatsache aus der Differentialgeomtrie ist, dass es keine distanzerhaltenende (isometrische) Parametrisierungen für allgemeine Oberflächen gibt. Isometrische Parametrisierungen auf Ebenen existieren ausschließlich für entwickelbare Oberflächen, das heißt Oberflächen mit einer Gaußschen Krümmung von Null. Da diese Voraussetzung in den meisten Fällen nicht gegeben ist, lässt sich eine isometrische Parametrisierung meist nur fehlerbehaftet annähern, wobei die oben beschriebenen Chart-Algorithmen helfen können, die eingeführte Verzerrungen teilweise stark zu reduzieren.
Es ist daher auch nicht verwunderlich, dass sich verschiedene Verfahren herausgebildet haben, deren Ziel die Erhaltung jeweils spezieller Netzeigenschaften ist. So unterscheiden [She06] die verschiedenen planaren Parametrisierungen in vier Kategorien: solche, die die eingeführte Verzerrung gänzlich ignorieren, konformale und authalen Methoden sowie Parametrisierungen, die den *stretch*, also die Zerrung der Textur, minimieren.
Ziel der konformalen Parametrisierung ist es, die Winkelveränderungen der einzelnen Dreiecke beim Übergang vom dreidimensionalen in den zweidimensionalen Raum möglichst gering zu halten, also möglichst winkelerhaltend zu transformieren. Dabei bleiben die lokalen Charakteristika des Netzes erhalten, während sich die Strecken zwischen einzelnen Knoten um einen Skalierungsfaktor verändern können.
Die authalische Parametrisierung dagegen versucht, die Flächen der einzelnen Dreiecke zu erhalten.
Über den Grad der Verzerrung hinaus können Parametrisierungen anhand weiterer Faktoren untersucht und ausgewählt werden. [She06] fasst diese zusammen unter:

Art der Randkurve: Viele Methoden gehen davon aus, dass die planare Ebene, auf die die 3D-Oberfläche gemappt wird, fest vordefiniert und konvex ist. Solche Parametrisierungen lassen sich typischerweise mit recht einfachen Formeln beschreiben und sind in der Regel auch sehr schnell. Parametrisierungen mit freier Randkurve arbeiten typischerweise langsamer, da die Randkurve Teil der Lösung der Parametrisierung ist und ebenfalls errechnet werden muss. Dafür weisen die so ausgeführten Parametrisierungen deutlich weniger Verzerrung auf.

Robustheit: Bei vielen Anwendungen wird vorausgesetzt, dass das Mapping der Parametrisierung bijektiv ist. Manchmal reicht dafür eine lokale Bijektivität, sodass sich die Orientierung der einzelnen Netz-Dreiecke nicht ändert. Manchmal wird eine globale Bijektivität benötigt, damit sich die Randkurve nicht überschneidet. Nur eine Untermenge von Parametrisierungsmethoden kann lokale oder globale Bijektivität garantieren, während andere Methoden dies nur bewerkstelligen können, wenn das Eingabenetz gewisse Bedingungen erfüllt.

Numerische Komplexität: Die existierenden Methoden lassen sich grob in lineare und nonlineare Methoden gruppieren, je nach verwendeten Optimierungsmethoden. Auch hier gilt der Trade-off: Lineare Optimierungen sind einfacher zu implementieren und besitzen eine schnellere Laufzeit auf Kosten höhere Verzerrungen.

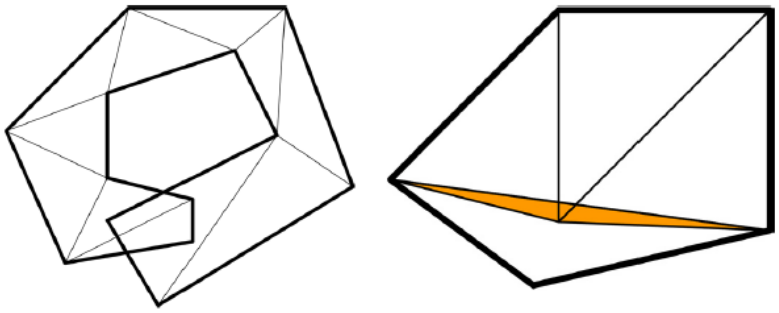

Abbildung 3.2: Nicht-bijektive Parametriseriungen: (a) planare Einbettung mit globalen Überlappungen; (b) planare Einbettung mit lokalen Überlappungen

3.2 Planare Einbettung

Eine der ältesten Methoden, die das *graph embedding*, also das Einbetten eines Graphen in eine Ebene, beschreibt, findet man in [Tut63]. Dabei lässt sich Tuttes Formulierung direkt auf triangulierte Netze übertragen und liefert ein generelles Rahmenwerk für viele später folgenden Methoden. Die Graphen-Einbettung besteht aus zwei Schritten: Als erstes werden die Randvertizes des Netzes auf den Rand einer konvexen 2D-Region gemappt. Anschließend werden die Positionen aller inneren Vertizes bestimmt, indem man das Baryzentrum der Nachbarn jedes Vertex ermittelt und ihn dort positioniert. Dies resultiert in einem linearen Gleichungssystem $\boldsymbol{L}$ für u- und v-Koordinaten, das so zu lösen ist, dass gilt:

$$\boldsymbol{L}u = 0,$$

$$\boldsymbol{L}v = 0,$$

$$\boldsymbol{L}_{i,j} = \begin{cases} -\sum_{k \neq i} \boldsymbol{L}_{i,k} & i = j \\ \omega_{ij} & (i,j) \in E \\ 0 & \text{sonst,} \end{cases},$$

wobei ω_{ij} Gewichte sind, die für jede Kante im Mesh definiert werden. Tutte wählt in seiner Arbeit für die Gewichte stets $\omega_{ij} = 1$ für jede Kante (i,j) im Mesh. Wichtig anzumerken ist, dass diese Art der Parametrisierung die Netzgeometrie nicht beachtet

und so Verzerrungen einführt, aber nicht minimiert.

3.3 Konformale Parametrisierung

Viele konformale Parametrisierungen basieren auf dieser Grundidee von Tutte und variieren nur in der Gewichtung der Kanten ω_{ij}, wobei die Wahl der Gewichte im Wesentlichen den Grad der Verzerrung und die Bijektivität der Parametrisierung beeinflusst. Einer der bekanntesten Gewichtungen in der Computergraphik stellen die *cotangent weights* dar:

$$w_{ij} = \frac{1}{2}\left(\cot \alpha_{ij} + \cot \beta_{ij}\right),$$

wobei α_{ij} und β_{ij} die gegenüberliegenden Winkel innerhalb der von zwei Dreiecken mit gemeinsamer Kante (i,j) sind. Die Formel stellt eine Minimierung der Dirichlet Energie dar. Dazu muss natürlich geklärt werden, wie diese überhaupt definiert ist und was sie repräsentiert.

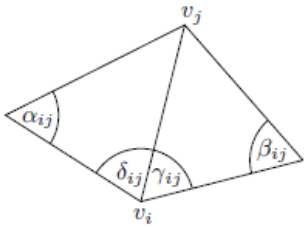

Abbildung 3.3: Die verwendeten Winkel bei der Erzeugung der Cotangens-Gewichte w_{ij} (Harmonische Gewichte) und der *mean value* Gewichte.

In [Sch03] gut beschrieben findet sich folgende Formulierung:

> Es sei eine Parametrisierung $f : \Omega \to M$ der Oberfläche M über der planaren Domäne Ω gegeben. Dann ist die Dirichlet Energie $E_D(f)$ der Parametrisierung f definiert als
>
> $$E_D(f) = \frac{1}{2}\int_{\Omega} |\nabla f|_g^2,$$
>
> wobei g die Metrik verdeutlicht, mit der der Betrag ausgewertet werden muss.

Aus Vereinfachungsgründen wird g oft als euklidische Metrik angenommen, wodurch

sich die Dirichlet Energie vereinfachen lässt zu:

$$\begin{aligned} E_D(f) &= \frac{1}{2}\int_{\Omega} |\nabla f|^2 \\ &= \frac{1}{2}\int_x \int_y |\nabla f|^2 \, dx \; dy, \end{aligned}$$

wobei $(x,y) \in \Omega \subset \mathbb{R}^2$ gilt. Doch wie steht jetzt die Dirichlet Energie in Verbindung zur Oberfläche M? Mittels einfacher Herleitung lässt sich zeigen, dass es sich dabei zunächst um eine konservative Abschätzung des Flächeninhaltes der Oberfläche M handelt. Dabei ist $f_x = f_{(x,y)} = \frac{\partial}{\partial x} f_{(x,y)}$ der Tangentialvektor an der Stelle $f(x,y)$ auf der Oberfläche M (analog für f_y). Der Betrag des Vektorprodukts $|f_x \times f_y|$ berechnet demzufolge den Flächeninhalt des infinitesimal kleinen Parallelogramms, das durch die beiden Tangentialvektoren aufgespannt wird. Integriert man diesen nun über die gesamte Domäne, ergibt sich der Flächeninhalt von M als:

$$\begin{aligned} & \int_x \int_y |f_x \times f_y| \, dx \; dy && \text{(Flächeninhalt von M)} \\ \leq & \int_x \int_y |f_x| \cdot |f_y| \, dx \; dy \\ \leq & \frac{1}{2}\int_x \int_y \left(f_x^2 + f_y^2\right) dx \; dy \\ = & \frac{1}{2}\int_x \int_y |\nabla f|^2 \, dx \; dy && \text{(Dirichlet Energie von f).} \end{aligned}$$

In einem letzten Schritt können Pinkall und Polthier in [Pin93] zeigen, indem sie die Dirichlet Energie diskretisieren und über die alle Dreiecke einer triangulierten Oberfläche errechnen, dass tatsächlich gilt:

$$\begin{aligned} E_D(f) &= \frac{1}{2}\int_{\Omega} |\nabla f|^2 \, dx \; dy \\ &= \frac{1}{4} \sum_{j \in \mathcal{N}(i)} \omega_{ij} \left(\boldsymbol{u}_i - \boldsymbol{u}_j\right)^2 \\ &= \frac{1}{4} \sum_{j \in \mathcal{N}(i)} \left(\cot \alpha_{ij} + \cot \beta_{ij}\right) \left(\boldsymbol{u}_i - \boldsymbol{u}_j\right)^2, \end{aligned}$$

wobei $\boldsymbol{u_i} \in \mathbb{R}^2$ die Abbildung des Vertex $\boldsymbol{x}_i \in \mathbb{R}^3$ auf der Domäne Ω und $\mathcal{N}(i)$ die 1-Ring Nachbarschaft des Knoten $\boldsymbol{u_i}$ darstellt.

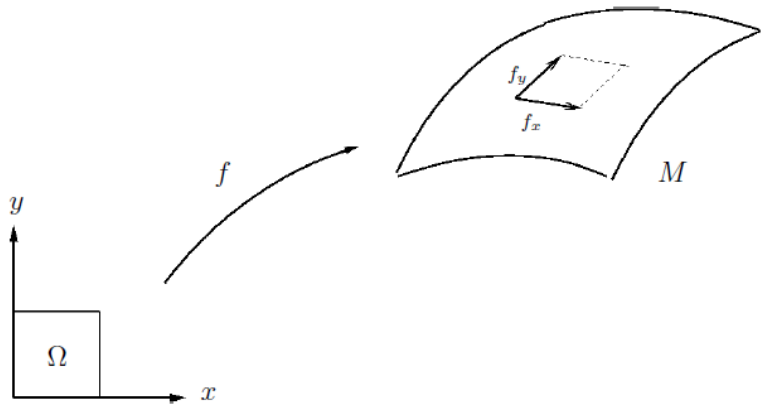

Abbildung 3.4: Abbildung der Domäne Ω auf die Oberfläche M durch die Parametrisierung f. f_x und f_y stellen die Tangentialvektoren von M dar. Zu berücksichtigen ist hier, dass es sich hier um Parametrisierung von dem zweidimensionalen in den dreidimensionalen Raum handelt.

Der größte Nachteil dieser Parametrisierung, trotz der bewussten Reduzierung der Winkelverzerrung, ist die Tatsache, dass die Gewichtungen ω_{ij} negativ werden können, sobald das Mesh stumpfe Winkel enthält. Als unmittelbare Folge davon kann es vorkommen, dass die Parametrisierung nicht mehr bijektiv ist, sodass Anwendungen, die dies voraussetzen, auf eine andere Parametrisierung zurückgreifen müssen.
Floater konnte jedoch in [Flo03] zeigen, dass sich die Bestimmung der Gewichtungen für die Parametrisierung vereinfachen lässt:

$$\omega_{ij} = \left(\tan\left(\frac{\gamma_{ij}}{2}\right) + \tan\left(\frac{\delta_{ij}}{2}\right)\right) / \left|\left|c(i) - c(j)\right|\right|,$$

wobei $c(i)$ und $c(j)$ die 3D Positionen der Vertizes i und j sind und γ_{ij} und δ_{ij} die Winkel in den zwei Dreiecken, die durch die Kante (i,j) verbunden sind. Diese *mean value* Gewichtungen ω_{ij} sind so stets positive und die Parametrisierung ist beweisbar immer bijektiv, mit dem einzigen Nachteil, dass die Winkelverzerrungen nun größer ausfallen können, als bei oben genannten harmonischen Gewichtungen.

3.4 Distanz- und authalische Parametrisierung

Im Gegensatz zu der konformalen Parametrisierung, die für alle glatten Oberflächen existiert, können distanzerhaltende Parametrisierungen nur auf entwickelbaren Oberflächen ausgeführt werden. Deswegen zielen solche Verfahren auch eher darauf ab, die lineare Verzerrung zu minimieren, anstatt sie komplett zu beseitigen. Um den Grad der Zerrung zu messen, führten [San01] zwei Metriken L^2 und L^∞ ein, in

denen $L^2(T)$ und $L^\infty(T)$ über ein Dreieck T wie folgt definiert sind:

$$L^2(T) = \sqrt{\frac{\Gamma^2 + \gamma^2}{2}},$$

$$L^\infty(T) = \Gamma,$$

wobei Γ und γ den größten beziehungsweise kleinsten Singulärwert der Ersten Fundamentalform, also der konstanten partiellen Ableitungen der Funktion, die die 2D-Oberfläche aufspannt, darstellt. Diese Art der Bewertung der Güte bezüglich der Dehnung der Oberfläche durch die Parametrisierung dient als generelles Messwerkzeug und hat deshalb Einzug in viele neuere Verfahren gefunden.
Authalische Parametrisierungen verhalten sich dagegen wesentlich anders: Während das Riemann-Mapping-Theorem zeigt, dass konformale Parametrisierungen von einer scheibenhaften Oberfläche auf eine planare Ebene mit vorher fixierter Randkurve stets existieren und diese auch eindeutig sind, kann dies von authalischen Parametrisierungen nicht behauptet werden. Dies zeigen Floater und Hormann in [Flo05], indem sich Kreise, egal um welchen Winkel man sie vom Zentrum aus rotiert, stets auf sich selbst beziehungsweise andere Kreise mappen lassen, ohne dass sich die Fläche der Kreise dadurch verändert:

Sei $f : S \to S$ ein Mapping von der Einheitsscheibe S auf sich selbst. Benutzt man die Polarkoordinaten $x = r\cos\theta$, $y = r\sin\theta$, findet man leicht heraus, dass die Determinante der Jakobimatrix jedes Mappings $f(x,y) = (u(x,y), v(x,y))$ wie folgt ausgedrückt werden kann:

$$\begin{aligned} detJ(f) &= u_x v_y - u_y v_x \\ &= \frac{1}{r}\left(u_r v_\theta - u_\theta v_r\right). \end{aligned}$$

Bedenkt man, dass das Mapping $f : S \to S$ definiert ist durch

$$r(\cos\theta, \sin\theta) \mapsto r\left(\cos\left(\theta + \phi(r)\right), \sin\left(\theta + \phi(r)\right)\right), \tag{3.1}$$

für $0 \leq r \leq 1$ und $-\pi < \theta \leq \pi$, wobei $\phi : [0,1] \to \mathbb{R}$ eine beliebige Funktion ist. Dieses Mapping projiziert jeden Kreis mit Radius r im Ursprung zentriert auf sich selbst, wobei zusätzlich eine Rotation um den Winkel $\phi(r)$ erfolgt. Wenn ϕ differenzierbar ist, so gilt dies auch für f und die Ableitung zeigt, dass

$$u_r v_\theta - u_\theta v_r = r, \tag{3.2}$$

völlig unabhängig von der gewählten Funktion ϕ ist.

Floater und Hormann schließen daraus, dass die Determinanten der Jakobimatrix demzufolge Eins sein muss ($|J(f)| = 1$) und dass f demzufolge ein authalisches Mapping darstellt, ungeachtet der gewählten univariaten Funktion ϕ. Aus diesem Grund verwenden neuere Methoden, wie zum Beispiel [Des02], lineare Trade-offs zwischen Winkel- und Flächenverzerrung beziehungsweise unterstützen ein Zweistufenverfahren, bei dem erst ein konformales Mapping ausgeführt wird, das daraufhin in einem Post-Processing Schritt nochmals ausgebessert wird.

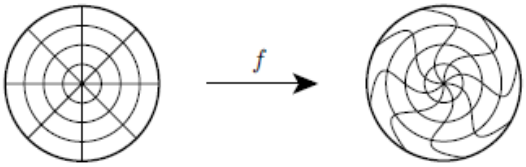

Abbildung 3.5: Abbildung der Domäne Ω auf die Oberfläche M durch die Parametrisierung f. f_x und f_y stellen die Tangentialvektoren von M dar. Zu berücksichtigen ist, dass es sich hier um Parametrisierung von dem zweidimensionalen in den dreidimensionalen Raum handelt.

KAPITEL 4

Packing

Nach erfolgreicher Durchführung der Parametrisierung und Segmentierung ist man nun im Besitz von planaren Flächenstücken, die in einer eins-zu-eins Beziehung mit der Oberfläche des 3D-Körpers stehen. In einem letzten Schritt gilt es nun, die oben gewonnenen Charts möglichst effizient in den Atlas, der Texturkarte, einzubetten, das heißt, sie so anzuordnen, dass möglichst wenig ungenutzter Raum übrig bleibt. Dieses Problem wird als *chart packing* Problem zusammengefasst. Dabei stellt das *chart packing* Problem nur eine alternative Form des 2D-Rucksackproblems (*bin packing problem*) dar und gehört somit zu den NP-harten Optimierungsproblemen, für die sich nur Heuristiken finden lassen, die fast optimale oder lokal optimale Lösungen erzeugen.
Der nachfolgende Teil soll bisherige Arbeiten bezüglich solcher Heuristiken erläutern, angefangen bei recht simplen Approximation, einfache rechteckige Charts zu bearbeiten, indem die einzelnen Charts ihrer Größe nach sortiert und eingefügt werden, über die Generalisierung dieses Problems, die auch Rotationen der einzelnen Charts und immer komplexer werdende Randformen erlaubt, bis hin zur freien Platzierung einzelner Charts innerhalb der Domäne. Abschluss und Hauptaugenmerk liegt dabei auf den letzten Algorithmus, dem *Modulo Valid Packing* Algorithmus, der den bis dahin stets konstanten Suchraum hingehend so erweitert, dass es den Charts erlaubt ist, sich um den Atlas herum zu entwickeln.

4.1 Charts mit rechteckiger Randkurve

Einer der ersten Arbeiten zur Modellierung des zweidimensionalen Packing-Problems wurde von Gilmore und Gomory in [Gil65] vorgestellt, die als direkte Erweiterung ihres eindimensionalen *bin packing* Problems entstand ([Gil61],[Gil63]). In dem darin beschriebenen Ansatz werden nacheinander Spalten mit fester Höhe und Weite erzeugt und mit Elementen gefüllt. Eine gefüllte Spalte beinhaltet eine Untermenge der zu lagernden Objekte und wird *pattern* genannt. In ihrem Modell besteht ein Pattern A_j aus einem Spaltenvektor mit n Elementen a_{ij} $(1 = 1, \ldots, n)$, die die

Werte Eins oder Null annehmen, je nachdem, ob das entsprechende Objekte im j-ten Pattern gelagert ist oder nicht. Das Set aller erlaubten Lösungen, bei denen jedes Objekt nur in einem Pattern lagert, wird durch die Binär-Matrix A dargestellt, die aus allen möglichen A_j Spalten ($j = 1, \ldots, M$) besteht und das korrespondierende mathematische Modell lautet:

$$\min \sum_{j=1}^{M} x_j, \tag{4.1}$$

wobei x_j den Wert Eins annimmt, wenn das Pattern zur Lösung gehört, und Null sonst. Das Ziel dieses Algorithmus ist, die kleinste Anzahl von Pattern zu finden, die es erlaubt, alle Objekte zu speichern.

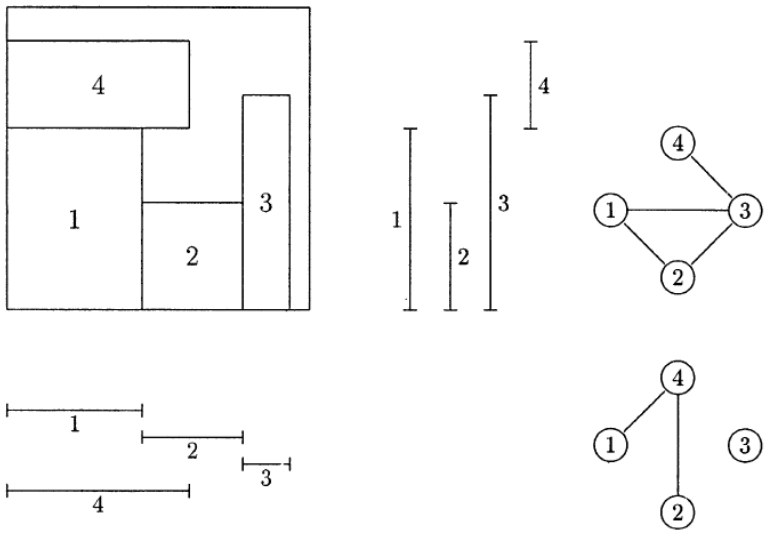

Abbildung 4.1: Modellansatz von Fekete und Schepers

Einen komplett anderen Ansatz verfolgen Fekete und Schepers in [Fek00]: Mittels Graphentheorie charakterisieren sie die erlaubten Packings der Objekte innerhalb eines Containers mit fester Weite W und Höhe H. Dazu werden zwei Graphen $G_w = (V, E_w)$ und $G_h = (V, E_h)$ bestimmt, und jedes Objekt, das in dem Container landet durch einen Knoten v_i repräsentiert. Es existiert eine Kante zwischen zwei Knoten (v_i, v_j) genau dann, wenn sich die Projektion der Objekte i und j entlang der horizontalen beziehungsweise vertikalen Achse des Containers überlappen. Fekete und Schepers können so zeigen, dass für jedes erlaubte Packing dann folgende

Aussagen gelten:

$$\sum_{v_i \in S} w_i \leq W$$
$$\sum_{v_i \in S} h_i \leq H;$$
$$E_w \cup E_h = \emptyset.$$

Zwei weitere wesentliche Einlagerungsstrategien werden in [Cof80] vorgestellt, den sogenannten *Next-Fit Decreasing Height* (NFDH) und *First-Fit Decreasing Height* (FFDH) Algorithmus. Unter der Annahme, dass die einzulagernden Objekte bekannt und nach ihrer Größe absteigend geordnet sind, werden sie mithilfe der beiden Heuristiken in den Container gelegt, der wiederum in verschiedene Level unterteilt ist. Dabei wird beim NFDH-Algorithmus das nächste zu behandelnde Objekt linksbündig in das aktuelle Level gelegt. Sollte das Objekt nicht mehr in das aktuelle Level passen, wird das Level *geschlossen* und ein neues Level darüber erzeugt. Als horizontale Hilfslinie dient dabei die Höhe des größten Objekts im gerade geschlossenen Level, welches sich stets ganz links im aktuellen Level befindet. Anschließend wird in dem neu erzeugten Level das gerade zu behandelnde Objekt linksbündig an erster Stelle platziert, und es beginnt die Bearbeitung des nächsten Objekts.
Beim FFDH-Algorithmus wird dagegen das Level nicht geschlossen, sobald das aktuelle Objekt dort nicht mehr hinein passt. Zwar wird ebenso ein neues Level erzeugt, falls das Objekt in kein Level passt, beim nächsten Objekt werden trotzdem auch alle vorherigen Level durchsucht und das Objekt an der ersten Position eingefügt, an der es passt.
Einen dritten Ansatz stellt der *Best-Fit Decreasing Height*-Algorithmus (BFDH) dar, bei dem die Objekte ebenfalls linksbündig in einem Level angeordnet und neue Level erzeugt werden, wenn das Objekt in keinen vorherigen Abschnitt mehr passt. Wesentlicher Unterschied zu FFDH ist, dass ein Objekt, das in mehrere Level passen würde, in das Level gelegt wird, in dem es den geringsten Abstand zum darüber liegenden Level hat. So wird effektiv der ungenutzte horizontale Raum minimiert.

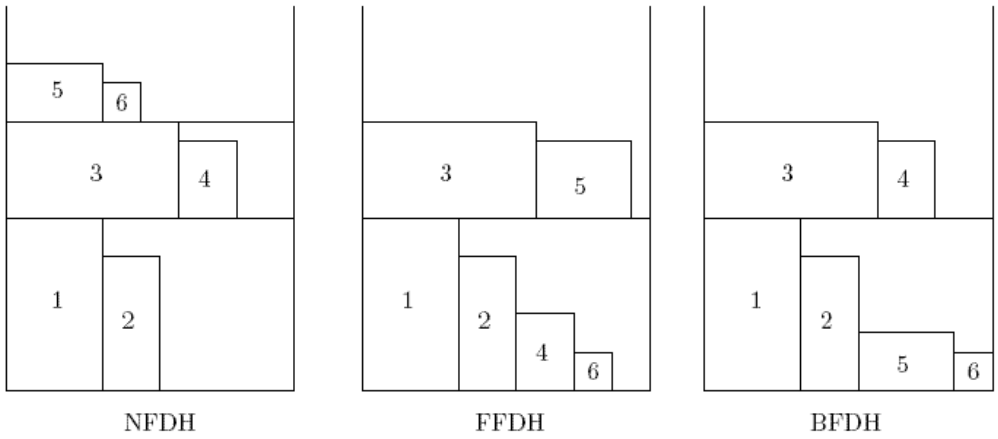

Abbildung 4.2: Anordnung der Objekte in einem Container mittels NFDH, FFDH und BFDH Heuristik.

Einen weiteren klassischen Ansatz findet man in [Bak80]: Der *Bottom-Left* (BL) Algorithmus teilt den zur Verfügung stehenden Container nicht in einzelne Level ein, sondern sortiert die einzelnen Objekte der Weite nach absteigend und positioniert sie in die unterste linke freie Position.
All diese relativ einfachen, klassischen Ansätze arbeiten sehr zufriedenstellend nahe des Kompaktheitsoptimums (siehe dazu: [Lod02]). Mit zunehmender Komplexität der 3D-Modelle und den oben vorgestellten Verfahren zur Erstellung von möglichst entwickelbaren Charts ist eine Behandlung der in die Texture Map einzubettenden Charts als bloße Rechtecke oder Quadrate eher unzureichend. Komplexe Ränder und teilweise innere Löcher erzwingen einen neuen Ansatz, um möglichst viele Charts platzsparend einzubetten. Darum sollen im Folgenden nun mehrere Verfahren mit freier Randkurve vorgestellt werden.

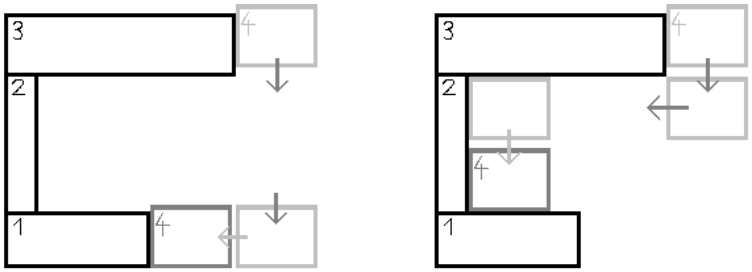

Abbildung 4.3: Bottom-Left Heuristik und eine leicht verbesserte BL-Alternative.

4.2 Charts mit freier Randkurve

Einen großer Unterschied zwischen rechteckigen Objekten und Objekten mit freier Randkurve stellt der Schnitttest zwischen den Objekten dar, der im irregulären Fall

wesentlich komplexer ausfällt. Um dennoch die gewünschten Resultate zu erzielen, ist es unerlässlich, dass verschiedene Approximationstechniken und geometrische Algorithmen in den verwendeten Packing-Algorithmus integriert werden.
Eine häufig verwendete Idee, um den Schnitttest zu beschleunigen, ist es, die die komplexen Konturen der Objekte nur anzunähern. Oliveira und Ferreira zeigen dies in [Oli93], indem sie die verschiedenen zu verarbeitenden irregulären Formen rasterisieren. Die Formen werden als Menge von winzigen Quadraten dargestellt, was wiederum ein schnelles Überprüfen auf Überlappung zwischen den einzelnen Objekten möglich macht, was jedoch stark auf Kosten der Genauigkeit geht, da besonders bei Rundungen Alias-Effekte auftreten. Eine Evaluierungsfunktion bestraft dabei mögliche Überlappungen der rasterisierten Objekte und zielt darauf ab, diese gänzlich zu vermeiden.
Einen anderen Weg geht Hiroyuki Okano in [Oka02]: Für einzulagernde Objekte wird erst die konvexe Hülle des Objekts bestimmt und die längste Kante dient als *baseline*, um die Ausrichtung des Objekts und seine Position innerhalb eines Containers zu bestimmen. Anschließend wird das Objekt in Linien parallel zur Baseline abgetastet und das Ergebnis in Form von mehreren binären Scanlinien dargestellt, das heißt, jedes Objekt besteht aus einer gewissen Anzahl von parallelen Liniensegmenten, teils unterschiedlicher Länge. Diese Liniensegmente werden möglichst eng verpackt und gelten fortan als ein neues Objekt, das in den Container gelagert werden kann.

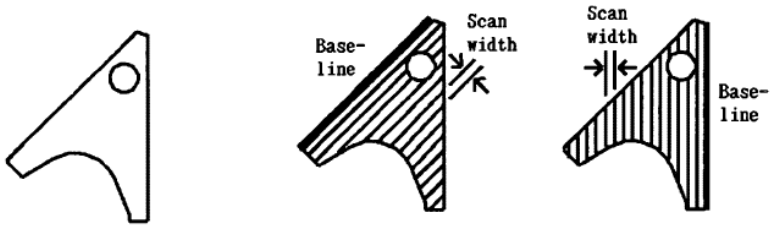

Abbildung 4.4: Objekte als Zusammenfassung von Linien-Segmenten, die anschließend in Containern verpackt werden.

4.3 Modulo Valid Packing Algorithmus

Eine recht neue Methode zum Einlagern der Charts innerhalb des Texturatlas stellt der *Modulo Valid Packing* Algorithmus dar. Die Idee hinter dem Algorithmus kann aus historischer Sicht sehr einfach als natürlicher Schritt in der Entwicklung der Packing Algorithmen angesehen werden.
Angefangen mit den oben beschriebenen, recht simplen Algorithmen, die einzelne Charts der Größe nach sortieren und nacheinander an Positionen innerhalb der Domäne positionieren, sodass eine minimale horizontale Linie entsteht, über die Generalisierung dieses Algorithmus, bei der die einzelnen Charts darüber hinaus auch rotiert werden dürfen, um einen Platzminimierung zu erhalten, bis hin zur

freien Platzierung einzelner Charts innerhalb der Domäne, ist es nur logisch, den bisher stets konsistenten Such- und Platzierungsraum ebenfalls einer genaueren Untersuchung zu unterziehen, sobald ein Chart platziert werden soll. Kurz gesagt, muss der jetzige Suchraum so erweitert werden, dass auch Charts erlaubt sind, die sich um den Atlas *herumwickeln*. Dies soll im Folgenden genauer erläutert werden.

4.3.1 Definition des Suchraums

Eingabe für den *Modulo Valid Packing* Algorithmus ist ein Mesh, bei dem jeder Vertex mit der Nummer des Charts versehen wurde, dem er während der Segmentierung zugewiesen wurde, sowie ein 2D Vektor $x = (u, v)^T$, der den Wert erhält, den der Vertex durch die Parameterisierung innerhalb dieses Charts zugewiesen bekommt. Vertizes an den Rändern zweier Charts können als Folge durchaus mehrmals markiert worden sein, da sie in mehreren Charts auftreten können. Wichtig ist festzuhalten, dass eine Parameterisierung hier, anders als im Parameterisierungskapitel vorgestellt, eine Funktion ist, die für die Vertizes eines jeden Charts i die entsprechende 2D Parameterdomäne D_i auf einen Teil der 3D Meshoberfläche S_i mappt.

$$P_i : D_i \subset \mathbb{R}^2 \mapsto S_i \subset \mathbb{R}^3$$

Ein valides Packing für eine gegeben Texturauflösung mit der Weite w und der Höhe h lässt sich als das Mapping

$$T_i : x \in \mathbb{R}^2 \mapsto s(R_i x + t_i) \in \mathbb{R}^2$$

formulieren, das es zu finden gilt, wobei R_i eine 2×2 Rotationsmatrix, t_i einen Translationsvektor und $s \in \mathbb{R}$ den globane Skalierungsfaktor darstellt. Darüber hinaus muss das Mapping die Eigenschaft haben, dass sich die einzelnen Parameterdomänen D_i nicht überlappen und kompakt verpackt sind, das heißt, dass sie innerhalb der Domäne, die durch w und h definiert ist, passen.
Für ein *Modulo Valid Packing* ist dies jedoch nicht ausreichend, weshalb der Suchraum verändert werden muss. Mit der Modulofunktion

$$M : (u, v)^T \mapsto (u \bmod w, v \bmod h)^T$$

lässt sich ein *Modulo Valid Packing* auf das Finden eines Mappings T_i reduzieren, das die folgenden drei Eigenschaften aufweist, wobei die letzte Eigenschaft optional ist und nur zur Optimierung dient:

1. **Keine Selbstüberlappung:** $\underset{i}{\forall} (M(D_i)$ injektiv$)$
2. **Keine Überlappung:** $\underset{i \neq j}{\forall} (M(D_i) \cap M(D_j) = \emptyset)$
3. **Maximale Packeffizienz:** $\frac{\sum_i area(D_i)}{w \cdot h}$ ist maximal

Unter diesen Bedingungen durchläuft der Algorithmus folgende Schritte, um eine gute Annäherung für ein *Modulo Valid Packing* zu finden:

1. Es wird ein globaler Skalierungsfaktor s ausgewählt.
2. Die Domänen D_i werden diskretisiert und nach ihrer Größe absteigend sortiert.
3. Die diskretisierten Domänen werden iterativ positioniert, solange dadurch ein valides Modulopacking erzeugt wird. Durch diesen Schritt werden die Werte der Rotationsmatrix R_i und des Translationsvektors t_i festgelegt.
4. Je nach Ausgang des Schrittes 3 wird der Skalierungsfaktor s erneuert.
5. Der Algorithmus wird solange fortgesetzt, bis eine gewisse Packeffizienz τ erreicht wird.

4.3.2 Skalierung

Der globale Skalierungsfaktor s selbst wird dabei wie folgt berechnet: Ausgehend von einer optimalen Packeffizienz von 100% gilt

$$s_{opt} = \sqrt{\frac{w \cdot h}{\sum_i area(D_i)}}.$$

Offensichtlich gilt jetzt $s = f \cdot s_{opt}$ mit $f \in (0, 1]$.
Um jetzt den fehlenden Faktor f zu bestimmen, führen [Nö11] eine Binärsuche auf dem Intervall $(0, 1]$ aus, wobei sie die untere Grenze $l = 0.1$, die obere Grenze $u = 0.9$ und den jetzigen Wert von f auf $f = \frac{l+u}{2}$ setzen. Anschließend wird versucht, ein valides Modulo Packing zu erzeugen. Sollte dies gelingen, wird die untere Grenze $l = f$ gesetzt, ansonsten wird u auf $u = f + c$ erhöht. Die Autoren beobachten, dass der Skalierungsfaktor auch darüber entscheidet, ob ein Modulo Packing zustande kommen kann oder nicht. Um den Algorithmus mehr Robustheit zu verleihen, setzen sie den letzten fehlenden Parameter c auf $c = \frac{u-f}{\sigma=3}$ und iterieren über ihren Algorithmus solange, bis ein gültiges Modulo Packing den Schwellenwert $u - l < \tau = 0.01$ erreicht.

4.3.3 Diskretisierung

[Nö11] benutzen aus Geschwindigkeitsgründen eine diskretisierte Domäne, da das Resultat (die Texture Map) letztendlich ebenfalls in Pixel diskretisiert wird. Darüber hinaus erlaubt dies eine weit aus effizientere Behandlung der einzelnen Charts als zum Beispiel stückweise lineare Funktionen. [Nö11] finden ebenfalls heraus, dass eine direkte Abhängigkeit zwischen Texturauflösung und Laufzeit ihres Algorithmus vorliegt, während die Auflösung an sich eher einen geringen Einfluss auf die Packeffizienz ausübt. Als Folge dessen schlagen Nöll und Stricker eine Domänengröße, die $\frac{1}{16}$ der

Zielauflösung beträgt, für ein empirisch guten Trade-off zwischen Geschwindigkeit und Effizienz vor.
Nachdem die Parameterdomäne diskretisiert wurde, muss nun für jeden Chart eine Annäherung seiner Form gefunden werden, um ein effizientes Bearbeiten zu ermöglichen. Dies geschieht auf gleicher Weise mittels Diskretisierung und in Abhängigkeit des Skalierungsfaktors s. Mit den so gewonnen rasterisierten Charts lassen sich für diese einfach die charakteristischen Linien, also Ober- und Unterkanten, sowie die linke und rechte Kante, berechnen, als auch deren Anteil an der Horizontlinie. Da die einzelnen Charts nacheinander in die Domäne eingefügt werden, wird nebenher auch die aktuelle Packdichte mittels der vier diskreten Horizontlinien berechnet.

4.3.4 Platzierung der Charts

Da das Packproblem *NP-hart* ist und man annehmen kann, dass $NP \neq P$ ist, gibt es nur eine Möglichkeit, ein optimales *Packing* zu erzeugen, indem man alle Kombinationen von R_i, t_i und s ausprobiert, die ein valides Packing erzeugen, und dasjenige mit der höchsten Dichte auswählt. Da dies klarerweise kein brauchbarer Ansatz ist, greift man auf Heuristiken zurück, die nah-optimale Lösungen in annehmbarer Zeit liefern. [Nö11] greifen hierbei auf die *Best-Fit Decreasing Height* Strategie zurück, die oben behandelt wurde.

4.3.5 Platzierungsstrategien

Als erstes werden die einzelnen Charts bezüglich ihrer Fläche absteigend sortiert und anschließend an der Position innerhalb der Domäne eingefügt, bei der sie eine lokale objektive Funktion minimieren. [Nö11] berechnen für jeden Chart 16 verschiedene Rotationsinstanzen und lassen jede horizontal und vertikal über die Domäne wandern, indem sie jeweils einen Wert des Translationsvektors t_i fixieren und den anderen so wählen, dass dieser funktional minimal ist, das heißt, eine möglichst geringe ungenutzte Fläche entsteht. Der Nachteil dieses Verfahrens ist, das jetzt zwei lokale Optimierungsstrategien agieren, nämlich eine, die ein Minimum in horizontaler Sicht, und eine, die in vertikaler Sicht ein lokales Optimum sucht, um ein globales Optimum zu erreichen. Dies kann genau zu einem reziproken Verhalten des erwünschten Effektes führen und große Löcher innerhalb der Domäne erzeugen, die zu einer Verringerung der Packeffizienz führen. Um dieses unerwünschte Verhalten zu reduzieren, führen [Nö11] zwei weitere Strategien ein.
Zusätzlich zu den vier äußeren Horizontlinien des gerade zu verpackenden Charts werden vier weitere innere Horizontlinien berechnet, die die innere vertikale und horizontale Struktur des Charts beziehungsweise die Lücken innerhalb des Charts beschreiben. Dies erlaubt das effiziente Finden von Lücken innerhalb der Domäne, die wiederum, sofern groß genug, als Platzierungsort für entsprechend kleine Charts verwendet werden können. Darüber hinaus kann so eine genauere Berechnung des ungenutzten Platzes stattfinden. Die Auswahl des Platzierungsortes des Charts

findet also nun anhand der vier Strategien der äußeren horizontalen und vertikalen Linien und der inneren horizontalen und vertikalen Horizontlinien statt, stets mit dem Bemühen, den ungenutzten Platz auf der Domäne zu minimieren.
Für alle vier Platzierungsstrategien wird zusätzlich eine fünfte Strategie hinzugefügt, die *mirrored* Strategie. Anstatt also nur den verschwendeten Platz zwischen dem Boden des Packings und der oberen Horizontlinie des aktuellen Charts zu minimieren, wird eine Minimierung der oberen Horizontlinie des Packings und der unteren Horizontlinie des einzufügenden Charts angestrebt. Die gleiche Strategie tritt beim horizontalen Fall ein, beim dem versucht wird, den Platz zwischen linker Horizontlinie des Charts und rechter Horizontlinie des Packings zu minimieren. Aus diesen 8 Strategien wird schließlich die valide Platzierung gewählt, die am wenigsten freien Platz übrig lässt. Nach einem erfolgreichen Platzieren wird die Rasterisierung des momentanen Packings erneuert. Bei einer Platzierung eines Charts innerhalb des bereits bestehenden Packings kann davon ausgegangen werden, dass ein kompakteres Packing erzeugt wird. Als Folge dessen bestrafen [Nö11] alle Platzierungsstrategien, die den aktuellen Chart über das aktuelle Packing anlegen würden mit einem Faktor von $\lambda = 1.1$ in ihren Experimenten. Für jeden gewählten Skalierungsfaktor s wird so in dem Platzierungsschritt eine einzigartige Sequenz der Positionierungsparameter R_i und t_i für jeden Chart i errechnet, bis alle Charts erfolgreich in der Domäne eingebettet sind.

KAPITEL 5

Visualisierung

Nachdem die theoretischen Grundlagen der Generierung von Textur-Atlanten in den vorherigen Kapitel gelegt worden sind, sollen diese nun mittels einer technischen Demonstration veranschaulicht werden. Die auf den Schwerpunkt Mesh-Segmentierung ausgelegte Arbeit soll hier durch einfache Beispiele ergänzt werden, mit dem Ziel, ausgewählte Ansätze der Segmentierung, wie sie oben in der Arbeit vorgestellt wurden, zu realisieren und zu veranschaulichen.

5.1 Programmüberblick

Um möglichst schnell und plattform-unabhängig an Ergebnisse zu gelangen, wurde das komplette Programm in JavaScript geschrieben. Dabei werden ebenfalls viele neuen Eigenschaften von HTML5[1] und CSS3[2] genutzt, insbesondere das Canvas-Element [3] mit seinem 2D- und 3D-Kontext, besser bekannt unter WebGL[4], sodass ein entsprechender Browser zum Ausführen des Codes von Nöten ist, wobei betont werden muss, dass diese Arbeit auf Cross-Browser-Kompatibilität verzichtet und klar den Schwerpunkt auf Google Chrome als ausführenden Browser legt.
Vieles, aber nicht alles, wurde per Hand geschrieben. Gerade jedoch bei der Manipulation der DOM-Elemente sowie bei vielen mathematischen Operationen wurde auf bereits existierende Bibliotheken zurückgegriffen. Zudem wurde die Hilfsbibliothek Underscore.js verwendet [5], die das Handhaben von Objekten und Arrays innerhalb von Javascript deutlich erleichtert. Die graphische Darstellung in 3D erfolgt über die JavaScript Bibliothek Three.js[6]. Da diese in der Version R61 noch keine Matrix-Matrix-Multiplikation und Addition beherrscht, wurden diese manuell über

1 `http://dev.w3.org/html5/markup`
2 `http://www.w3.org/Style/CSS/`
3 http://www.w3.org/TR/html5/embedded-content-0.html#the-canvas-element
4 `http://www.khronos.org/webgl`
5 `http://underscorejs.org/`
6 urlhttp://threejs.org/

Prototyping hinzugefügt.
Benutzer-Interaktivität der Eingabemaske und Veränderungen der Webseite werden durch *jQuery* der Version 1.8.2 erreicht[1] sowie dem CSS-Framework Bootstrap in der Version 3.2.0 [2].
Für eine flüssigere 3D-Darstellung wurde darüber hinaus die Utility-Bibliothek *WebGL-Utils*[3] hinzugefügt, deren hauptsächlicher Bestandteil in der Funktion *requestAnimFrame* zu finden ist, die den Speicherbedarf von Browser-Tabs, die sich im Hintergrund befinden, deutlich reduziert.
Abgerundet wird das Programm durch RequireJS [4], einem File- und Modullader von James Burke.
Das Programm wurde erfolgreich auf Chrome 5+ getestet und liegt dieser Arbeit bei.

5.2 Benutzeroberfläche

Beim Aufrufen des Programms befindet sich der Nutzer auf der Settings-Seite. Unter den einzelnen Menüpunkten befinden sich verschiedene Parameter, die die Atlantengenerierung festlegen, die im Folgenden beschrieben werden soll.

1. **Choose a mesh**: Der Benutzer hat die Wahl mittels der neuen HTML5 File-API auf eine lokale Mesh-Datei zuzugreifen oder durch die untere Drop-Down Box eine der mitgelieferten Files auszuwählen.

2. **Segmentation Parameters**: Diese Optionen beschreiben das Verhalten des Segmentierungsalgorithmus. Die *maximale Normalendifferenz* (Max. Normal Difference) gibt den maximalen Winkelunterschied der Normalen zweier benachbarter Dreiecksflächen an, bei der die eine Fläche zu einem bestehenden Chart hinzugefügt wird.
   Die *minimale Chartgröße* (Min. Chart Size) ist ein Faktor, der besagt, dass Charts solange zusammengefügt werden, bis sie mindestens ein gewisses Größenverhältnis zum größten Chart haben. In der Standardeinstellung gilt daher, dass alle Charts mindestens neunzehntel der Fläche des größten Charts besitzen.
   Die *maximale Chartanzahl* (Max. Chart Number) gibt die maximale Anzahl an Teilstücken ein, in die das Netz am Ende der Segmentierung eingeteilt werden soll. Der Algorithmus wird, sofern möglich, versuchen diesen Wert einzuhalten, kann aber bei Bedarf den Wert erhöhen, falls die Topologie des Netzes dies

1 `http://jquery.com/`
2 urlhttp://getbootstrap.com
3 `http://code.google.com/p/webglsamples/source/browse/book/extension/webgl-utils.js`
4 urlhttp://requirejs.org

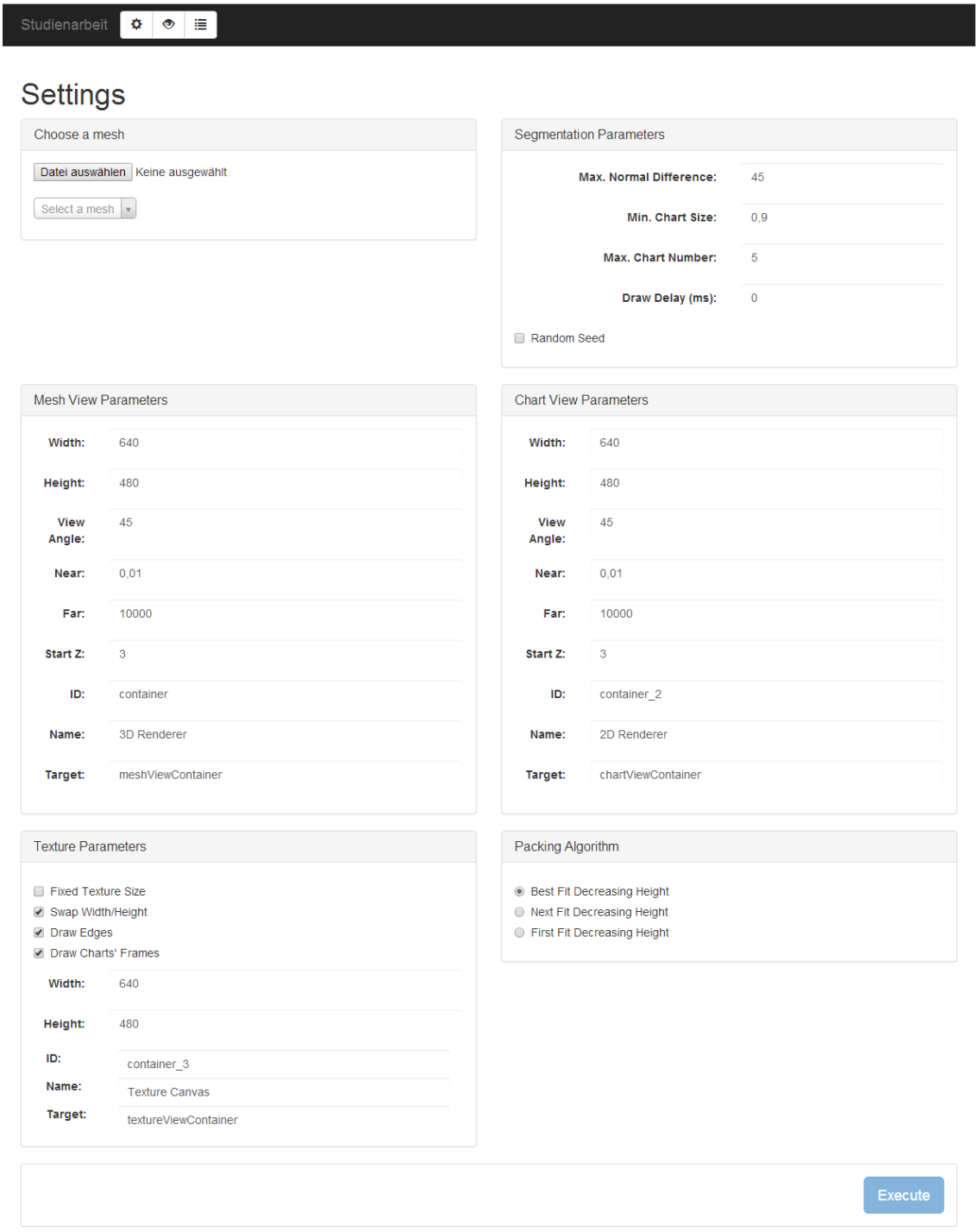

Abbildung 5.1: Einstellungsoberfläche des Programms. Alle Eingaben besitzen Standardwerte, sodass nur die Auswahl des zu bearbeitenden Meshes obligatorisch ist.

verlangt.
Die *Zeichenverzögerung* (Draw Delay (ms)) gibt die Zeit in Millisekunden an, die das Programm wartet, bevor es die einzelnen Flächen des Netzes und deren Zugehörigkeit zu den einzelnen Charts farblich kennzeichnet. Standardmäßig ist der Wert aus Performancegründen auf Null gesetzt. Möchte man jedoch verfolgen, wie der Algorithmus arbeitet, kann der Wert natürlich erhöht werden.
Die letzte Checkbox *Zufallstartpunkt* (Random Seed) in dieser Sektion bestimmt den Anfangsvertex, bei dem der Algorithmus ansetzt. Standardmäßig beginnt der Algorithmus beim ersten Vertex im Netz und arbeitet sich dann voran. Dies ist sinnvoll, um reproduzierbare Ergebnisse zu erzeugen. Da die Auswahl des Startpunktes aber Auswirkungen auf die Qualität der Segmentierung hat, kann durch Einschalten der Checkbox ein zufälliger Startknoten ausgewählt werden, der interessante Resultate hervorbringen kann.

3. **Mesh View Parameters**: Innerhalb dieses Box werden nur die graphischen Parameter eingestellt, die zum Anzeigen des Netzes benötigt werden. Darunter fallen die Größe des Canvas in *Höhe* (Height) und *Weite* (Width), die Erstellung des Frustum für die Kamera durch *Blickwinkel* (View angle), die *Nah- und Fernebene* (Near, Far), der Startwert der Kamera auf der Z-Achse (Start Z) sowie drei Paramter (ID, Name, Target), die das Erstellen des Canvas und sein Einbinden in den DOM zur Laufzeit ermöglichen.

4. **Chart View Parameters**: Dieser Menüpunkt verhält sich gleich zu den Mesh View Parameters und zeigt die gewonnen Teilstücke der Segmentierung planar nebeneinander im 3D-Raum in einem seperaten Canvas Element an.

5. **Texture Parameters**: Neben den Darstellungseinstellungen für die Erzeugung und das Einfügen des Canvas in das DOM (Width, Height, ID, Name, Target) erlauben die Texturparamter auch weitere graphische Aspekte zu regulieren.
Die *feste Texturgröße* (Fixed Texture Size) bestimmt, ob nach der Ausführung des Packing-Algorithmus die Texturwerte von Höhe und Breite verändert werden dürfen, um eine Textur mit möglichst geringer Fläche zu erzeugen.
Längen-/Breitentausch (Swap Width/Height) erlaubt es, vor der Anwendung des Packing-Algorithmus die einzelnen Charts so zu drehen, dass die Höhe stets der längsten Seite entspricht.
Mit *Kanten zeichnen* (Draw Edges) werden die Kanten der einzelnen Dreiecksflächen mit eingezeichnet und mit *Chart-Rahmen zeichnen* (Draw Charts' Frames) wird die benutzte Bounding-Box um die einzelnen Charts sichtbar gemacht.

6. **Packing Algorithm**: Hier kann der Benutzer auswählen, nach welcher Strategie die einzelnen Charts auf dem Atlas angeordnet werden.

5.3 Programmaufbau und -ablauf

Im Folgenden soll nun die Funktionsweise des Programms erläutert werden. Dazu wird eine Übersicht über die einzelnen Module des Programms gegeben und deren Wechselwirken untereinander beschrieben. Zentrale Codeabschnitte werden genauer erläutert und zum besseren Verständnis graphisch dargestellt. Der gesamte Quellcode ist kommentiert und lässt sich in dem mitgelieferten Verzeichnis nachlesen.
Der grobe Aufbau lässt sich wie folgt beschreiben: Über die Eingabemaske, die als HTML-File unter dem Namen *index.html* vorliegt, wählt der Benutzer ein zu bearbeitendes Netz aus. Dabei kann der Benutzer auf Netze auf seiner lokalen Festplatte zurückgreifen oder eines der mitgelieferten Netze im Drop-Down Menü auswählen. Das Programm unterstützt dreieckflächige Netze im *Object File Format* (.off) [1].

5.3.1 Einlesen mittels reader.js

Die Controller-Funktion der Eingabemaske übernimmt die Datei *main.js*. Die Auswahl eines Netzes wird dabei von ihr registriert und führt zum Aufruf des Reader-Moduls (*reader.js*). Dort wird die ausgewählte Datei eingelesen und in einem ersten Schritt ein Mesh erstellt, das die Bibliothek three.js benötigt, um das Netz graphisch darstellen zu können. Darüber hinaus werden, sofern das Netz von der lokalen Festplatte geladen wurde, weitere Meta-Information wie Filename, Erstellungsdatum und Größe in einem Meta-Objekt gespeichert.
Ein erfolgreiches Einlesen und Verarbeiten der Datei führt zu einem Event-Aufruf mit dem Namen *doneReading*, der als Parameter die Metainformationen und das erstellte Mesh an das Controller-Modul zurückgibt. Die Kontrollfunktion reagiert auf das *doneReading*-Event mit der Ausführung der Initialisierungsfunktion der Applikation, deren Aufgabe es ist, eine funktionierende Halbkantenstruktur zu erzeugen und diese in dem *meshObject* zu speichern.

5.3.2 Initialisierung und Erstellung der Halbkanten-Datenstruktur

Das Applikation-Objekt (*application.js*) legt vier Funktionen offen, auf die die Controller-Funktion aus *main.js* zugreifen kann und besitzt eine noch leere private Variable *meshObject*, das Herzstück der Anwendung, in der die Ergebnisse der einzelnen Operationen gespeichert werden und die es nun in der Initialisierungsphase zu füllen gilt. Der erste Schritt in der Initialisierungsphase des Programms besteht darin, eine Halbkanten-Datenstruktur aus den Vertizes und Flächen des Eingabenetzes zu erzeugen. Dazu wird das eigens dafür geschriebene Modul HEDS

1 `http://shape.cs.princeton.edu/benchmark/documentation/off_format.html`

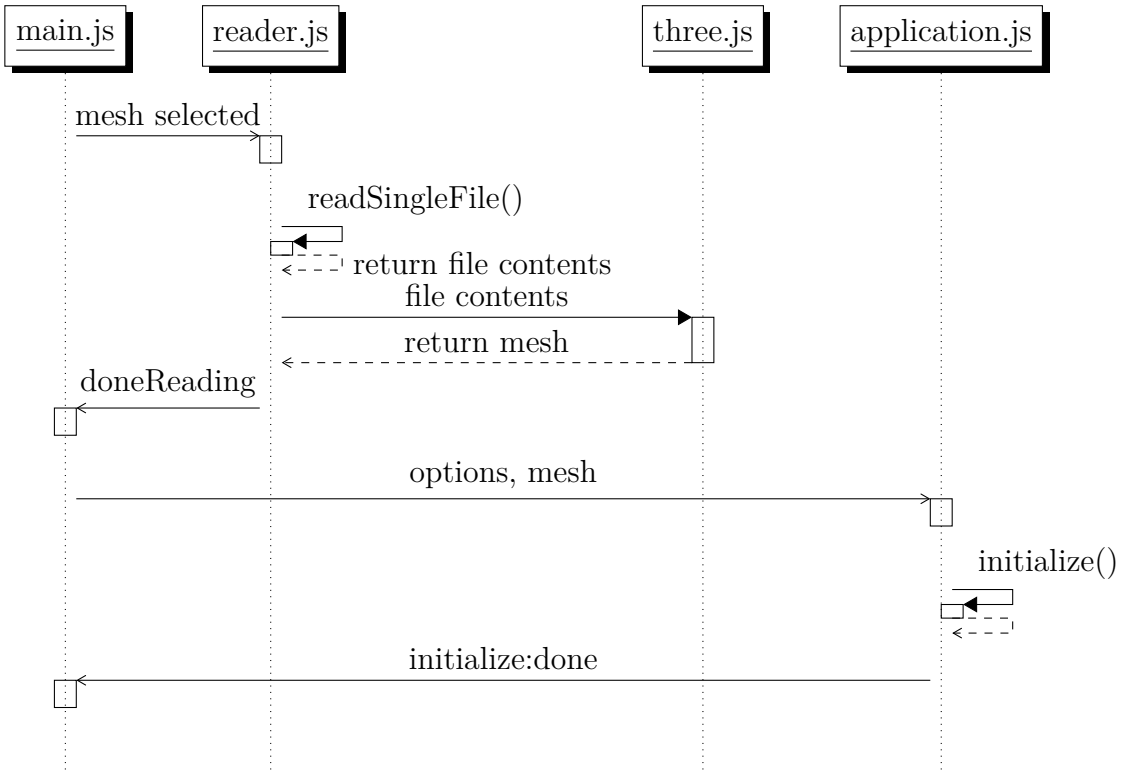

Abbildung 5.2: Initialisierung: Einlesen einer Datei und Erstellung des meshObjekts

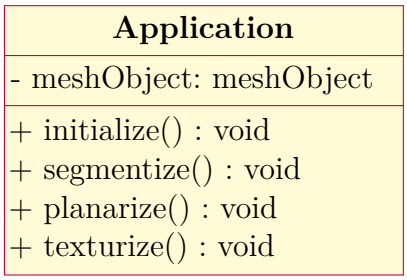

Abbildung 5.3: Application Klasse

(*HalfEdge-Data-Structure*) aufgerufen und die darin enthaltene Funktion *buildHEDS* ausgeführt. Die Schritte zur Erzeugung der Datenstruktur sehen wie folgt aus:

1. Erstelle für jeden Knoten in der Eingabedatei ein Vertex-Objekt.

2. Erstelle nun für jede Fläche der Eingabedatei ein Oberflächen-Objekt (*Face*), dieses enthält unter anderen eine Liste mit den drei Vertex-Objekten, aus denen die Fläche zusammengesetzt ist. Gleichzeitig wird die Face-Liste eines jeden Vertex für jedes Face aufgefüllt, an dessen Konstruktion er beteiligt ist.

3. Erzeuge die Halbkanten zwischen den drei Vertizes einer Dreiecksfläche. Halb-

kanten besitzen unter anderen Referenzen auf vorhergehende Halbkanten, nachfolgende Halbkanten, die Dreiecksfläche zu der sie gehören und einen Richtungsvektor.

4. Als Zwischenschritt werden die Attribute der Vertizes erweitert, so dass diese nun auch jeweils eine Liste an eingehenden und ausgehenden Halbkanten besitzen. Sofern eine Halbkante keine Außenkante des Netzes ist, kann so nun auch die Zwillingshalbkante einer Halbkante bestimmt werden.

5. Anschließend erfolgt die Erzeugung von Kanten, die im Gegensatz zu den Halbkanten ungerichtet sind. Dabei bilden bis zu zwei Halbkanten eine Kante.

6. Abschließend gilt es noch die letzten Beziehungen zueinander einzufügen: Kanten werden passend in die jeweiligen Kanten-Arrays der Face-, Halbkanten- und Vertexobjekte eingefügt. Außerdem lassen sich nun die Nachbarflächen einer jeden Fläche und die Nachbarvertizes eines jeden Knotens bestimmen. Diese werden in die entsprechenden Felder eingetragen und die Größen der einzelnen Flächenstücke können errechnet werden.

Da Three.js in der Version R61 selbst noch keine Wiremeshes von Objekten erzeugen kann, wird des Weiteren noch die Funktion **buildWireframe** aufgerufen, die als Funktionsparameter die Liste der erzeugten Halbkanten übernimmt und daraus ein eigenes Mesh erzeugt. Jedes einzelne erzeugte Element kommt darüber hinaus in eine eigene Elementliste. Zusammen werden diese mit den Parametern aus der Eingabemaske, die der Benutzer ausgefüllt hat, an den Konstruktor der Klasse **MeshObject** übergeben, der das komplett fertig erzeugte Objekt zurückgibt, das anschließend von *application.js* verwendet wird. Die Initialisierungsphase wird mit einem weiteren Event mit dem Namen *initialize:done* an die Controller-Funktion beendet. Ab diesem Zeitpunkt sind alle Voraussetzungen für eine Segmentierung erfüllt, was durch die Aktivierung des großen *Execute*-Knopfes ersichtlich wird. Gleichzeitig steht dem Benutzer zu diesem Zeitpunkt schon eine einfache graphische Darstellung des Meshes zur Verfügung, bei der die Flächen als auch die Kanten monochrom gekennzeichnet sind.
Ein Bestätigen des *Execute*-Knopfes führt schließlich zur Ausführung der anderen drei Funkionen des Application-Objekts und zur Erzeugung des Textur-Atlas.

5.3.3 Segmentierung

Die Funktion *segmentize* der Applikation ruft die *createCharts*-Funktion der eigenen *meshObject*-Variable auf, deren Aufgabe das Einteilen des Netzes in Charts und das Erzeugen der Chart-Struktur ist. Dabei geht die Funktion in folgenden fünf Schritten vor, die anschließend näher erläutert werden:

1. **Creation**: Erzeuge eine Initialaufteilung des Netzes in Charts.

MeshObject
- sceneOptions: object + vertices: array<Vertex> + faces: array<Face> + edges: array<Edge> + halfedges: array<Halfedge> + charts: array<Chart> + submeshes: array<Three.Mesh> + options: object + meta: object + faces2D: array<object> + vertices2D: array<object> + sceneObjects: array<Scene>
- buildScene(submeshes, options) : sceneObject - _randomHex(): String - sortAreaAscending(): boolean + startAllAnimations(): void + createCharts(): void + createPlanarRepresentation(): void + mergeCharts(): void + removeEmptyCharts(): void + _drawFaces(): void + refineCharts(): void + createTexture(): void

Abbildung 5.4: Auszug aus MeshObject-Klasse

2. **Update**: Update die Nachbarschaftsbeziehungen der Charts zueinander.
3. **Merging**: Löse Charts auf, die den vom Benutzer eingegebenen Kriterien nicht entsprechen und füge sie anderen Charts zu.
4. **Refine**: Bessere die Außenkanten der Charts aus.
5. **Draw**: Zeichne die Charts.

In der Creation-Phase wird eine Ersterzeugung der Charts vorgenommen. Dabei wandert der Algorithmus von dem ersten Oberflächendreieck (Face) aus in einem *Breath-First Search*-Ansatz über das Eingabenetz: Es wird über die Liste der Faces iteriert und jedes Element darauf hingehend untersucht, ob es schon abgearbeitet wurde. Wenn dies nicht der Fall ist, landet das Element in einer *queue*. Aus dieser wird stets das erste Element entfernt (*qHead*), das im Folgenden das Start-Face für einen neuen Chart darstellt. Von dort aus wird nach und nach über den 1-Ring,

2-Ring, . . .n-Ring der Nachbar-Faces iteriert und diese dem Chart hinzugefügt, wie sie den Eingabekriterium des Benutzers entsprechen. Dieses ist definiert durch den maximalen Winkelunterschied ϕ, den eine neue Oberfläche zum Ursprungs-Face haben darf, wobei $0 \leq \phi < 90°$ gilt. Jede hinzugefügte Oberfläche wird als bearbeitet markiert und die jeweiligen Attribute des Faces und des Charts aktualisiert. Können

Algorithm 1 Creation-Phase

```
1:  MAD := 45                          ▷ maximal angle difference in degree, e.g. 45°
2:  for f ∈ Faces do
3:      if f is not marked then
4:          queue := [f]
5:          neighbours := f.neighbours
6:          chart := new Chart()
7:          cNormal := f.Normal
8:
9:          while queue is not empty do
10:             qHead ← queue.pop
11:             if qHead is not marked then
12:                 n1 ← qHead.normal      ▷ normalized normal of current element
13:                 n2 ← cNormal                    ▷ normalized normal of chart
14:                 angle ← arccos(n1 · n2)
15:                 if angle ≤ MAD then
16:                     chart.add(f)                          ▷ add face to chart
17:                     qHead.mark()
18:                     for nF ∈ neighbours do
19:                         queue.push(nF)                ▷ add new faces to queue
```

keine neuen Oberflächen zu einem Chart mehr hinzugefügt werden, gilt dieser als geschlossen. In der Liste aller Faces wird das nächste unbearbeitete Element herausgesucht, das als Start-Face für den nächsten Chart benutzt wird und die oberen Schritte erneut ausgeführt.
Nach Beendigung der Creation-Phase ist jede Oberfläche genau einem Chart zugewiesen. Da für die Merging-Phase die Relationen der Charts zueinander bekannt sein müssen, werden nun die Nachbarschaftsbeziehung der Charts zueinander herausgearbeitet. Dazu werden von jedem Face des Charts die Zugehörigkeit der Nachbaroberflächen des Faces untersucht. Gehört so ein Nachbar-Face zu einem anderen Chart, wird dieser neue Chart in einer Liste vermerkt, aus der am Ende sämtliche Duplikate entfernt werden. Abhängig von der Eingabe des Benutzers und der Topologie des Netzes kann es vorkommen, dass in der Creation-Phase sehr viele kleine Charts erzeugt werden. Darum erfolgt nun die Merging-Phase.

Algorithm 2 Dissolving a chart

```
1:  procedure DISSOLVE(chart)
2:      queue := chart.Faces

3:      while queue is not empty do
4:          qHead ← queue.pop

5:          if qHead is boundary face then
6:              nCharts = [ ]

7:              for nF ∈ qHead.Faces do              ▷ get neighbour faces of qHead
8:                  if nF.Chart ≠ chart then
9:                      nCharts.push(nF.Chart)       ▷ collect neighbour charts

10:             nCharts ← unique(nCharts)            ▷ remove duplicates

11:             minAngle := 360
12:             bestChart := nCharts[0]
13:             for c ∈ nCharts do
14:                 n1 ← c.Normal      ▷ normalized normal of the neighbour chart
15:                 n2 ← chart.Normal              ▷ normalized normal of the chart

16:                 angle ← arccos(n1 · n2)

17:                 if angle < minAngle then
18:                     minAngle ← angle
19:                     bestChart ← c

20:             chart.remove(f)
21:             bestChart.add(f)                   ▷ insert face to best fitting chart

22:             for nF ∈ qHead.Faces do              ▷ get neighbour faces of qHead
23:                 updateBoundary(nF)       ▷ check if face is now a boundary face
24:         else
25:             queue.push(qHead)
```

Algorithm 3 Update-Phase

```
1: chartList := [ ]

2: for f ∈ Chart do                                  ▷ Each face of the chart
3:     for n ∈ f do                                  ▷ Each neighbour face of the face
4:         if n.Chart ≠ Chart then
5:             chartList.push(nChart)

6: chartList ← unique(chartList)
7: Chart.Charts ← chartList
```

Hierbei betrachtet der Algorithmus nun die erzeugten Charts und wendet dabei die vom Benutzer eingestellten Kriterien zur Chartgröße und -anzahl an. Es werden alle Charts herausgefiltert, die im Vergleich zum größten Chart nicht die eingegebene relative Größe aufweisen. Jeder so ausgewiesene Chart wird mittels der Funktion *dissolve* aufgelöst. Dies geschieht, indem alle ihm zugehörigen Faces nacheinander von außen nach innen dem zu ihnen am Besten passenden Nachbarchart zugewiesen werden. Ein so geleerter Chart wird mitsamt aller Relationen zu anderen Charts aus dem *meshObject* und den betroffenen Nachbarcharts entfernt, das Objekt selbst gelöscht. Dieser Vorgang wird solange durchgeführt, bis die Chartgröße und -anzahl den Eingabeparametern entspricht. Bevor die Charts dann in der Draw-Phase gezeichnet

Algorithm 4 Merging-Phase

```
1: maxChartNumber := 5
2: tooSmall := true
3: minChartSize := 0.9 · largestChart

4: while tooSmall and charts.length > maxChartNumber do
5:     tooSmall ← false
6:     queue := [ ]

7:     for c ∈ Charts do
8:         if c.size < minChartSize and charts.length > maxChartNumber
   then
9:             tooSmall := true
10:            queue.push(c)

11:    queue ← queue.sort()                          ▷ In ascending order
12:    qHead ← queue.pop
13:    qHead.dissolve()
```

werden, werden ihre Grenzen zueinander ausgebessert. Dazu dient die Refine-Phase.

Hierzu werden die äußeren Oberflächen, die Boundary-Faces, untersucht. Da alle Flächen Dreiecke sind, können folgende Annahmen getroffen werden:

1. Jedes Boundary-Face (Face am Rand des Charts) hat genau drei Seiten, an denen maximal eine benachbarte Oberfläche angrenzt.

2. Wenn ein Chart aus mehr als einer Oberfläche besteht, gilt, dass von den drei Nachbaroberflächen eines Boundary-Face mindestens eine zum selben Chart gehört.

3. Wenn dies der Fall ist und nur genau eine Nachbaroberfläche zum gleichen Chart gehört, bedeutet dies, dass das Dreieck in mindestens einen anderen Chart hereinsticht.

4. Sticht dieses Boundary-Face wirklich genau nur in einen einzigen Nachbarchart hinein, bedeutet dies außerdem, dass zwei der drei Kanten des Dreiecks benötigt werden, um die Grenzen der beiden Charts festzulegen.

Gerade der letzte Punkt bedeutet, dass die Charträndер besonders gezackt aussehen. Dieses Verhalten kann ausgebessert werden, wenn das betreffende Dreieck aus seinem jetzigen Chart entfernt wird und in den benachbarten eingefügt wird. Nach Beendigung der fünf Phasen wird ein weiteres Event mit dem Namen *segmentize:done* von der Applikation gesendet. Dies hat zur Folge, dass die Main-Methode in eine neue Ansicht springt und das verarbeitete Netz anzeigt.
Gleichzeitig beginnt die dritte Stufe der Applikation durch den Aufruf der Funktion *planarize*.

5.3.4 Erstellung der 2D Repräsentation

Das Ziel dieser Funktion ist es, aus dem dreidimensionalen Charts eine zweidimensionale Repräsentation zu erzeugen, mit der schlussendlich die Textur auf dem Canvas-Element hergestellt werden kann. Dies erfolgt in drei Schritten:

1. Erzeuge eine Kopie eines jeden Charts.

2. Rotiere und projiziere die neuen Charts solange, bis der z-Wert jedes Vertex Null ist

3. Generiere anschießend daraus 2D-Objekte, mit denen das Canvas Element arbeiten und die Textur erstellen kann.

Die Funktion *planarize* in der Applikation ruft als erstes die *createPlanarRepresentation*-Methode des Mesh-Objekts auf, welche im Gegenzug die Instanzmethoden der Vertizes und Faces aufruft, die eine Instanzvariable für die 2D-Repräsentation in dem jeweiligen Objekt erzeugen und diese mit einer Kopie der Three.js-Darstellung des Objekts füllen. Gleichzeitig werden Referenzen zwischen neuem und altem Objekt

Algorithm 5 Refine-Phase

```
1: for c in Charts do
2:     hasChanged ← true

3:     while hasChanged is true do
4:         hasChanged ← false

5:         for finChart.Faces do
6:             if f is boundary face then
7:                 nCharts := [ ]
8:                 for nFinf.Faces do        ▷ for all neighbour faces of current face
9:                     if nF.Chart ≠ c then
10:                        nCharts.push(nF.Chart)

11:                differentCharts ← unique(nCharts)        ▷ number of different
   charts

12:                if nCharts.length = 2 and differentCharts = 1 then
13:                    c.remove(f)
14:                    nF.Chart.add(f)
15:                    hasChanged ← true

16:                    for nF ∈ f.Faces do  ▷ get neighbour faces of current Face f
17:                        updateBoundary(nF)     ▷ check if face is now a boundary
   face
```

sowie zwischen den neuen Objekten geschaffen mit der Folge, dass die Halbkantendatenstruktur für die 2D-Repräsentation nicht verloren geht. Alle so neu erzeugten Objekte, die der 2D-Darstellung dienen, werden außerdem in einer entsprechenden Instanz-Variable des jeweiligen Charts gespeichert, sodass dieser ebenfalls auf seine eigene 2D-Repräsentation zugreifen kann.
Die Parametrisierung der 3D-Vertizes erfolgt über ein einfaches Projektion-Rotations-Verfahren. Jeder Chart besitzt eine globale Normale, die als Durchschnitt aller Normalen seiner Oberflächen angesehen werden kann. Damit die Z-Komponenten für alle Vertizes Null wird, muss eine Rotation des Charts erfolgen, sodass seine Normale mit der Richtung der Z-Achse im 3D-Raum übereinstimmt. Dazu wird für jeden Chart eine Rotationsmatrix mithilfe der Rodrigues-Formel [1] erzeugt und diese

1 `http://mathworld.wolfram.com/RodriguesRotationFormula.html`[19.07.2014]

auf alle Vertizes des Charts angewendet:

$$R_\phi = I + \sin(\phi) \cdot A + (1 - \cos(\phi)) \cdot A^2.$$

Die Rotationsmatrix beschreibt dabei die Drehung ϕ um eine Rotationsachse N. Diese Achse ist das Ergebnis des Kreuzprodukts der Normalen des Charts N_{chart} und der Z-Achse N_{zAxis},

$$N = \frac{N_{chart} \times N_{zAxis}}{\|N_{chart} \times N_{zAxis}\|},$$

wobei A die antisymmetrische Matrix aus der Rotationsachse beschreibt:

$$A = \begin{pmatrix} 0 & -N_z & N_y \\ N_z & 0 & -N_x \\ -N_y & N_x & 0 \end{pmatrix}.$$

Durch die Anwendung der Rotationsmatrix auf die Elemente des Charts zeigen nun der Durchschnitt aller Normalen entlang der Z-Achse, sodass nun nur noch eine einfach Translation der einzelnen Vertizes erfolgen muss, damit der gesamte Chart auf der X-Y-Ebene liegt.
Dieses Verfahren ist relativ schnell und erzeugt stabile Ergebnisse, leidet jedoch an Präzision: Durch die simple Projektion eines 3D-Körpers oder Ausschnitt eines solchen auf eine Fläche, die dann gedreht wird, geht die Topologie des Teilnetzes in den 2D-Raum gänzlich verloren, was zu einem Abbildungsfehler führt, wenn die erzeugte Textur wieder zurück auf das Netz übertragen wird. Des Weiteren können durch eine schlechte Segmentierung des Netzes Teile des gleichen Charts hintereinander projiziert werden und gehen damit bei der Einbettung verloren.
Am Ende der *planarize*-Funktion steht dem Programm sowohl ein weiteres Three-Mesh zur Verfügung, das die einzelnen Teil-Charts auf der X-Y-Ebene darstellt, als auch eine interne 2D-Repräsentation der Vertizes und der Oberflächen, die nun für den letzten Schritt, dem Packing, benötigt werden.

5.3.5 Packing

Der Packing-Algorithmus behandelt die einzelnen Charts wie Rechtecke, so wie es in Kapitel 4 beschrieben wird. Dazu werden als erstes die Extrema der einzelnen Charts über die Funktion *findObjectExtrema* ermittelt. Für jeden Chart, der jetzt durch ein Objekt repräsentiert wird, kann mit dieser Funktion sein Zentrum, als auch die vier Außenpunkte ermittelt werden, die nötig sind, um eine rechteckige Box, die Boundary-Box, um den Chart zu legen und das Chart-Objekt korrekt im Atlas zu platzieren.
Je nach Benutzereinstellungen werden die einzelnen Rechtecke noch einmal betrachtet und um 90 Grad gedreht, sodass die längere Seite der Rechtecke stets die Höhe der Box ist. Anschließend wird das vom Benutzer ausgewählte Verfahren, *Best-Fit*

Decreasing Height, *Next-Fit Decreasing Height* oder *First-Fit Decreasing Height* auf die einzelnen Objekte angewandt.
Sollte das so enstandene Packing nicht mehr auf den Textur-Atlas passen, skaliert der Algorithmus die einzelnen Charts solange, bis diese auf die ausgewählte Fläche passen.
Ein Container, wie er in Kapitel vorgestellt wird, besteht hierbei nur aus einem Array von Levels, welche wiederum sehr klein gehaltene Objekte sind, auf denen der Texturgenerator arbeiten kann: Dabei entspricht das y-Offset eines Levels dem y-Offset des vorherigen Levels plus dessen Höhe oder ist, wie im Falle des erstens Level, direkt Null. Das x-Offset gibt dagegen die Position innerhalb des Levels an, an der das nächste Objekt eingefügt werden kann. Außerdem dient es zur Bestimmung, ob ein neues Objekt überhaupt eingesetzt werden kann oder ein neues Level geöffnet werden muss.
Das height-Attribute gibt die Höhe des jeweiligen Levels an und wird verwendet, um später zu überprüfen, ob das Packing valide ist, das heißt innerhalb der festgelegten Texturgrößen passt.
Darüber hinaus enthält ein Level noch eine Liste aller Element, die sich innerhalb des Levels befinden.
Eine erfolgreiche Ausführung des Texturgenerators erzeugt ein Canvas-Element, in dem die einzelnen Elemente nach dem ausgewählten Packing-Verfahren angeordnet sind. Die Applikation ist mit diesem Schritt abgeschlossen.

5.3.6 Auswertung

Das Erstellen eines voll funktionsfähigen Textur-Atlas Generator stellte den zweiten großen Schwerpunkt dieser Arbeit da. Schon in der Planungsphase war zu erkennen, dass die Komplexität des Programms sehr stark mit dem Funktionsumfang skalieren würde, weswegen früh Annahmen, Abwägungen und Ausschlüsse getroffen werden mussten.
Dies fängt schon bei der Wahl des Formats der Eingabenetze an. In dieser Arbeit wurde das **Open-File Format** (.off) gewählt, da seine Codierung sehr einfach gehalten ist und ein schnelles Lesen des Eingabenetzes ermöglicht. Nichtsdestotrotz unterstützt das OF-Format auch die Erweiterungen COFF, NOFF und deren Kom-

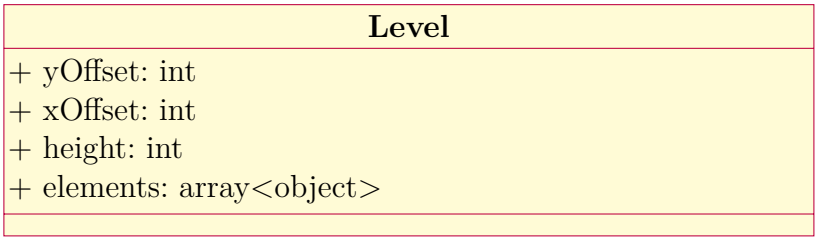

Abbildung 5.5: Level Objekt

bination CNOFF, die Informationen über die Farben (Color) und die Normalen (Normals) oder gar beides enthalten. Alle drei können verarbeitet werden, Farben und Normalen werden jedoch übersprungen, da das Programm durch die Generierung der Halbkanten-Datenstruktur diese selbst wieder anlegt. Die Auswahl der weiteren Netztypen, gerade auch proprietärer Modelltypen wie .blend, .obj, .c4d und .3dm ist durchaus möglich, aber würde schnell den Rahmen dieser Studienarbeit durch Beachtung vieler Einzel- und Sonderheiten der einzelnen Formate sprengen.
Dies führt auch gleich zu einer weiteren Herausforderung: die Umwandlung und Darstellung der Eingabenetze. Zu Beginn dieser Arbeit bot es sich an, die Halbkanten-Darstellung mittels OpenMesh[1] zu erzeugen. In der damaligen Version 2.3 wurde dies aber aufgrund von vielen Unzulänglichkeiten und der fehlenden Plattform-Unabhängigkeit schnell wieder verworfen, darüber hinaus schien das Projekt eingestellt worden zu sein. Parallel zu dieser Studienarbeit wurden die Arbeiten an OpenMesh jedoch wieder aufgenommen und so steht seit dem 18. Juli die neueste Version 3.2 auf der Website zur Verfügung, weshalb die Verwendung von OpenMesh für zukünftige Projekte oder gar der Erweiterung dieses Projektes durchaus wieder in Betracht fallen sollte.
Durch die Erzeugung einer eigenen Halbkanten-Datenstruktur wuchs die Komplexität weiter an, neu auftauchende Herausforderungen verhinderten ein Fokussierung auf die eigentliche Aufgabenstellung.
Dabei stellte das größte Problem die fehlenden Nebenläufigkeitsmöglichkeiten in JavaScript dar. JavaScript unterstützt nicht wie Java die Erstellung neuer Threads zur parallelen Verarbeitung von Prozessen. Das Betriebssystem weißt dem Browser oder jedem Tab im Browser einen Thread zu, den das jeweilige JavaScript-Programm dort mitbenutzt und von dessen Ressourcen zerrt. Die graphische Darstellung über das Canvas-Element mit einem 3D-Kontext erlaubt es, viele Berechnungen auf die GPU der Grafikkarte zu verlagern, das Erzeugen der Halbkanten-Datenstruktur und das Anordnen der Charts auf der Textur finden jedoch noch speziell auf der CPU statt, was deutlich spürbar wird, je höher die Komplexität des Eingabenetzes ist. Eine Lösung für zukünftige Erweiterung dieses Programms könnte das Verwenden der Funktionen *setTimeout()*, *setInterval()* und *XMLHttpRequest()* sein, um Nebenläufigkeit vorzutäuschen oder viele der Funktionen und Prozesse in Zukunft durch Webworker [2], eine neue API von HTML5, sobald deren Implementierung abgeschlossen ist, zu ersetzen. Im jetzigen Zustand muss jedoch bei der Wahl eines Eingabenetzes in Kauf genommen werden, dass die Gefahr besteht, dass die Länge der Ausführung des Programms zu einem Kontext-Verlust des Browsers seitens des Betriebssystems führt, wenn dessen Abarbeitung zum Beispiel über 6 Sekunden dauert.
Ein weiterer Punkt ist die Programmiersprache JavaScript selbst. Ohne strikte

1 `http://www.openmesh.org`
2 `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html`

Typisierung und mithilfe von Vererbung über Prototyping lassen sich schnell und unkompliziert Klassen und Klassenhierarchien konstruieren. Als ein größeres Problem während der Programmierung stellte sich das Präzisionsverhalten der Float-Zahlen heraus. Das sehr ungenaue Verhalten von JavaScript im Bezug auf Nachkommastellen, gerade bei trigonometrischen Funktionen, führte öfters zu ungewollten Ergebnissen und Verlassen des eigentlichen Wertebereichs, was ungewollte Nebenwirkungen bei Folge-Berechnungen hatte. Das Auffinden solcher Fehler gestaltete sich aufgrund der sehr begrenzten Debugging-Funktion der Skriptsprache als besonders zeitaufwendig. Ein letzter großer Punkt stellte das Austauschen von Informationen zwischen den einzelnen Bibliotheken dar. So verwendet die eingeführte Datenstruktur eigene Klassen und Klassenhierarchien, um Daten intern darzustellen, kapselt aber Vektoren und Face-Indizes aus der ThreeJS-Bibliothek ein, um ein gleichzeitiges Rendern und Rechnen zu erlauben. Des Weiteren benötigt das einfache Canvas-Element eine eigene Datenstruktur, da die 2D-Ebene auf dem Canvas ihren Ursprung standardmäßig in der linken oberen Ecke hat. Steigende Y-Werte werden durch Punkte, die tiefer auf dem Canvas-Element liegen dargestellt. Es lohnte sich daher, anstatt bei jeder Berechnung eine Spiegelung an der X-Achse vorzunehmen, die Positionselemente jedes Charts einmalig neu für das Canvas zu generieren. Da diese Operation jedoch auf der CPU ausgeführt wird, stellt dies, wie oben schon erwähnt, gerade bei größeren Netzen ein Problem im Bezug auf Geschwindigkeit und Antwortverhalten des Browsers dar.

KAPITEL 6

Zusammenfassung

Die vorliegende Arbeit liefert einen Überblick über die vielfältig Abläufe der Textur-Atlas Generierung. Es werden der generelle Ablauf der Generierung, sowie ausgewählte Verfahren der Segmentierung, Parametrisierung und des Packings erläutert. Für jeden Teilschritt werden verschiedene Varianten und deren Anwendungsfälle beschrieben, Beispiele gezeigt und ihre angewendeten Strategien, Heuristiken und die damit einhergehenden Vor- und Nachteile aufgezeigt.
Mit dem beigelegten Programm werden darüber hinaus einerseits das dargestellte Wissen praktisch reproduziert, andererseits durch die spezielle Auswahl der Programmiersprache JavaScript und der Ausführung im eigenen Browserfenster, eine Machbarkeitsstudie angelegt, die die Portabilität dieses generellen Problems darstellt. Gerade die technische Ausarbeitung des Programms zeigt jedoch auch, dass viele der vorgestellten Methoden und Verfahren gewisse Grundlagen benötigen, die nicht immer als gegeben vorausgesetzt werden und gewisse Vorüberlegungen getroffen werden müssen.
Darunter fallen in erster Linie die Beschreibung des gewählten Netzes und die Erreichbarkeit der einzelnen Komponenten des Netzes, wie Knoten, Flächen und deren Relationen zueinander. Ohne eine gut strukturierte Topologie-Beschreibung kann eine Atlas-Generierung nicht stattfinden. Mit der Entscheidung, die Datenstruktur schließlich selbst zu implementieren, gewinnt diese Arbeit zusätzliche Tiefe und rückt weitere Aspekte bezüglich Laufzeitverhalten, Speicherplatz und Nebenläufigkeit im Browser sowie Eingabenetzkomplexität von gewählten Netzen in den Vordergrund.
Letztendlich kann diese Arbeit als eine Art Denkanstoß für weitere Verbesserung eines browser-internen Atlas-Erzeugungsprogramms angesehen werden. Die oben genannten Performance-Probleme, eine besseren Klassenstruktur der Komponenten oder ein pixelgenauer Packing-Algorithmus sind nur ein paar Bereiche, die sich optimieren lassen und können als mögliche Anknüpfungspunkte für Folgearbeiten angesehen werden.

Literaturverzeichnis

[Bak80] Baker, B. S.; Jr., E. G. Coffman und Rivest, R. L.: Orthogonal packing in two dimensions. *SIAM J. Computing* (1980), Bd. 9(4):S. 846–855

[Cof80] Coffman, Edward G.; Garey, M. R.; Johnson, David S. und Tarjan, Robert Endre: Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms. *SIAM J. Comput.* (1980), Bd. 9(4):S. 808–826

[Des02] Desbrun, Mathieu; Meyer, Mark und Alliez, Pierre: Intrinsic Parameterizations of Surface Meshes (2002)

[Fek00] Fekete, Sandor P. und Schepers, Jörg: On more-dimensional packing I: Modeling (2000)

[Flo01] Floater, Michael und Hormann, Kai: Parameterization of triangulations and unorganized points, in: *Principles of Multiresolution in Geometric Modelling*, Springer, S. 127–154

[Flo03] Floater, Michael S.: Mean value coordinates. *Computer Aided Geometric Design* (2003), Bd. 20

[Flo05] Floater, Michael S. und Hormann, Kai: Surface Parameterization: a Tutorial and Survey, in: *Advances in multiresolution for geometric modelling*, Springer Verlag (2005), S. 157–186

[Gil61] Gilmore, P.C. und Gomory, R.E.: A linear programming approach to the cutting stock problem, Operations Research 9 (1961)

[Gil63] Gilmore, P.C. und Gomory, R.E.: A linear programming approach to the cutting stock problem - part II, Operations Research 11 (1963)

[Gil65] Gilmore, P.C. und Gomory, R.E.: Multistage cutting problems of two and more dimensions, Operations Research 13 (1965)

[Gu02] Gu, Xianfeng; Gortler, Steven J. und Hoppe, Hugues: Geometry Images, Bd. 21 (3), Association for Computing Machinery, San Antonio, TX, S. 355–361

[Hü01] Hübli, Andreas und Gross, Markus: Multiresolution Feature Extraction from Unstructured Meshes, in: *Proceedings of IEEE Visualization Conference 2001*

[Jul05] Julius, Dan; Kraevoy, Vladislav und Sheffer, Alla: D-Charts: Quasi-Developable Mesh Segmentation, in: *Computer Graphics Forum, Proceedings of Eurographics 2005*, Bd. 24, Eurographics, Blackwell, S. 581–590

[Lod02] Lodi, Andrea; Martello, Silvano und Monaci, Michele: Two-dimensional packing problems: A survey. *European Journal of Operational Research* (2002), Bd. 141(2)

[Lé02] Lévy, B.; S. Petitjean, N. Ray und Maillot, J.: Least Squares Conformal Maps for Automatic Texture Atlas Generation, in: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, ACM, S. 362–271

[Nö11] Nöll, T. und Stricker, D.: Efficient Packing of Arbitrary Shaped Charts for Autotmatic Texture Atlas Generation, Bd. 30 (2011)

[Oka02] Okano, H.: A scanline-based algorithm for the 2D free-from bin packing problem, Journal of the Operations Research (2002)

[Oli93] Oliveira, J. F. und Ferreira, J. S.: Algorithms for nesting problems, Applied Simulated Annealing (1993)

[Pin93] Pinkall, Ulrich und Polthier, Konrad: Computing Discrete Minimal Surfaces and Their Conjugates. *Experimental Mathematics* (1993), Bd. 2:S. 15–36

[San01] Sander, P.; Snyder, J.; Gortler, S. J. und Hoppe, H.: Texture mapping progressive meshes, in: *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, S. 409–416

[Sch03] Schall, Oliver: Automatische Erzeugung von Textur-Atlanten für mannigfaltige Flächen, Diplomarbeit der TH Aachen (2003)

[She02] Sheffer, Alla und Hart, John C.: Seamster: Inconspicuous Low-Distortion Texture Seam Layout, in: *IEEE Visualization*, S. 291–298

[She06] Sheffer, Alla; Praun, Emil und Rose, Kenneth: Mesh Parameterization Methods and Their Applications. *Foundations and Trends in Computer Graphics and Vision* (2006), Bd. 2(2)

[Tut63] Tutte, W. T.: How to draw a graph, Proceedings of the London Mathematical Society (1963)

Abbildungsverzeichnis

Tabellenverzeichnis