OBJECT-SCAPING

Eine Methode zur digitalen Produktion von nonstandard Objekten

DIPLOMARBEIT

zur Erlangung des akademischen Grades eines

Diplom-Ingenieurs

Studienrichtung : Architektur

Markus Manahl

Technische Universität Graz

Erzherzog-Johann-Universität

Fakultät für Architektur

Betreuer: Univ.-Prof. Dipl.-Arch. Dr.sc.ETH Urs Leonhard Hirschberg

Institut für Architektur und Medien

Jänner 2010

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst,
andere als die angegebenen Quellen/Hilfsmittel nicht benutzt,
und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen
als solche kenntlich gemacht habe.

STATUTORY DECLARATION

I declare that I have authored this thesis independently,
that I have not used other than the declared sources / resources,
and that I have explicitly marked all material which has been quoted
either literally or by content from the used sources.

Graz, 11. Jänner 2010

Mein Dank gilt allen, ohne die diese Arbeit nicht möglich gewesen wäre,

im Besonderen Lore, Fini, Herbert und nicht zuletzt Professor Urs Hirschberg.

Inhaltsverzeichnis

1. Einführung

Der Einfluss der digitalen Technologien auf das architektonische Schaffen wurde spätestens in den 90er Jahren unübersehbar, als die damaligen CAD Anwendungen und 3D-Modellierprogramme die Welt der nichteuklidischen Geometrien erschlossen.

In der Vergangenheit waren es oft Entwicklungen von neuen Baumaterialien oder Konstruktionsweisen, die einen Wandel der architektonischen Formensprache auslösten, die Form richtete sich nach den Bedingungen ihrer Herstellung – nach den Eigenschaften verfügbarer Materialien und den handwerklichen oder maschinellen Verfahren, um diese Materialien zu bearbeiten. Mit dem Aufkommen der frei geformten Flächen in den 90er Jahren drehte sich der Prozess aber um. Die digitalen Entwurfswerkzeuge führten ein neue Formensprache in die architektonische Gestaltung ein, und es mussten erst geeignete Wege gefunden werden, die neuen Formen zu Materialisieren.

Heutige CAD Anwendungen machen die Manipulation von komplexen Geometrien am Bildschirm zum Kinderspiel. Diese immateriellen Formen dann aber in die Realität zu exportieren, ist eine ganz andere Geschichte.

„The man in the street still has no idea of this, but delineating the control points of a NURBS surface in order to generate a fluid surface is now within the range of any user after an apprenticeship of just half an hour. And that's how it should be. That on the other hand it may then be a question of controlling these surfaces, of modifying them by intervening on their coordinates, of giving them a thickness and of fabricating them, that's a whole new ballgame: namely, to shift the problems onto someone else while multiplying the budget." [1]

In den letzten 20 Jahren wurden eine Reihe von spektakulären Projekten realisiert, oft mit spektakulären Budgets. Aber in diesem Fall ist der Nonstandard in der Architektur nichts anderes, als die neueste Form der Distinktion einer Klientel, die die Möglichkeit hat, mehr als nur ein Standardbudget aufzubringen.

Die vorliegende Arbeit stellt den Prozess der Materialisierung digital ausgearbeiteter Formen an den Anfang. Die Grundlage der Arbeit ist eine Herstellungs-methode, die ich – in Anlehnung an ihre Inspirationsquelle – *Object-Scaping* nennen möchte. Object-Scaping ist eine Methode zur Fabrikation dreidimensionaler, geometrisch komplexer Objekte durch die Schichtung von Querschnitten, die mithilfe zweidimensional arbeitender, digitaler Fabrikationsmaschinen (zum Beispiel Laser- oder Waterjet-Cutter) hergestellt werden können.

Im Gegensatz zu anderen, formneutralen digitalen Produktionsmethoden, wie der Stereolithographie oder dem 3D-Druck, ist die hier präsentierte Methode ein spezifischer Prozess, dem eine eigene Formensprache eingeschrieben ist. Das dritte Kapitel dieser Arbeit ist der Vorstellung der Object-Scaping Methode gewidmet, sowie der Ausarbeitung einer prozessgerechten Formensprache, die sich aus den Bedingungen dieser Herstellungsmethode entwickelt.

Das Herzstück des Projekts ist eine parametrische Geometriebeschreibung, die den Produktionsprozess algorithmisch abbildet. Durch dieses parametrische Modell entsteht, wenn man so will, eine „intelligente" Geometrie, eine Form, die über die Bedingungen ihres eigenen Herstellungsprozesses bescheid „weiß" und sich selbständig entsprechend ihrer Produktionsbedingungen organisiert. Die Grundlagen des Geometriemodells werden ebenfalls im dritten Kapitel erläutert.

Auf der Basis des parametrischen Modells habe ich im Zuge dieser Arbeit ein Computerprogramm entwickelt, das zur Visualisierung und Bearbeitung von Objekten dient, die mit der vorgestellten Methode produziert werden sollen, und mit dem es möglich ist, die digitalen Daten für den Produktionsprozess zu generieren. Die Vorstellung der Computerapplikation bildet den Abschluss der Projektbeschreibung im vierten Kapitel.

1 Patrick Beaucé, Bernard Cache, *Towards a Non-standard Mode of Production*, in: Bernard Leupen, René Heijne, Jasper van Zwol (Hrsg.), *Time-based Architecture* (010 Publishers, Rotterdam, 2005), S. 116

2. Digitale Methoden in Entwurf und Produktion

Finally, any technology is anything but new. If we will understand technology at all, we will begin to see it as an uninterrupted and ubiquitous practice. All technologies have a long period of social, cultural, technical, and practical preparation. The habits of mind that underwrite one technology often influence successive technologies. In our mythical paradigm of progress, technical mastery, and paradigm shifts, terms such as "new" are merely rhetorical escalations.

Kiel Moe, *The Non-Standard, Un-Automatic Prehistory of Standardization and Automation in Architecture*[1]

2.1 Numerische Steuerung

Im Jahr 1949 beauftragte die amerikanische Air Force den Erfinder John T. Parsons mit dem Bau einer Werkzeugmaschine für die serielle Produktion hochpräziser Flugzeugbauteile. Vor der Entwicklung automatisch gesteuerter Produktionsverfahren wurden geometrisch komplexe Maschinenbauteile üblicherweise gefertigt, in dem eng gestaffelte Löcher entsprechend der gewünschten Form des Bauteils in ein Werkstück gebohrt und das überschüssige Material dann entweder maschinell oder sogar händisch weggefeilt wurde. Parsons entwickelte in Zusammenarbeit mit dem Servomechanisms Laboratory des Massachusetts Institute of Technology (MIT) eine dreiachsige Fräse, die von Servomotoren entlang ihrer Achsen bewegt wurde, und damit die automatisierte, präzise Fertigung von komplexen Bauteilen ermöglichte.[2]

Das Neue an dieser Maschine war ihre Steuerung durch numerische Daten. Sie war ein Werkzeug, das nicht mehr von dem neurologisch-muskulären Feedback-Loop eines Handwerkers geführt wurde, sondern von impuls-gesteuerten Motoren.[3]

1 Kiel Moe, *The Non-Standard, Un-Automatic Prehistory of Standardization and Automation in Architecture*, in: Proceedings of the 2007 ACSA National Conference (Associations of Collegiate Schools of Architecture, Washington, 2007)
2 David E. Weisberg, *The Engineering Design Revolution* (Ebook, Online unter http://www.cadhistory.net/, 2008), Kap. 3, S. 5f
3 Kiel Moe, *Automation Takes Command*, in: Robert Corser (Hrsg.), *Fabricating Architecture: Selected Readings in Digital Design and Manufacturing* (Princeton Architectural Press, New York, 2010), S. 154

Parsons Fräse gilt zu recht als der Urahn heutiger digitaler Produktionsmaschinen. Abbildung 2.1 zeigt eine schematische Darstellung der Maschine aus Parsons Patentantrag von 1958. Das Grundprinzip numerisch gesteuerter Fräsen ist bis heute dasselbe geblieben.

Die Programmierung der Maschine erfolgte damals jedoch noch über Lochstreifen, auf denen die einzelnen Arbeitsschritte der Maschine kodiert waren – ein Konzept das schon Joseph-Marie Jacquard bei seinem berühmten automatischen Webstuhl aus dem Jahr 1801 angewandt hatte.[4] Die Koordinatenangaben auf den Lochstreifen wurden von der Steuerungselektronik in Bewegungspfade übersetzt, aus denen die Impulsdaten für die Servomotoren erzeugt wurden, die dann den Fräskopf der Maschine bzw. das Werkstück selbst bewegten.

Die händische Programmierung jedes einzelnen Arbeitsschritts war allerdings ein äußerst aufwändiges Verfahren, das sich bald als größtes Manko des neuen Produktionsprozesses herausstellte. Ein neuer Weg musste gefunden werden, um die Maschine zu steuern.

Zur gleichen Zeit wurde am MIT unter dem Codenamen Whirlwind das erste Computersystem mit der Fähigkeit zur Echtzeitoperation entwickelt.[5] Whirlwind war eigentlich als Flugsimulator konzipiert worden, aber die Ingenieure des Automatic Programmed Tool (APT) Systems, das aus der Zusammenarbeit von Parsons mit dem Servomechanisms Laboratory hervorging, nutzten das System, um die Erstellung der Lochstreifen zu automatisieren, und entwickelten in der Konsequenz eine prozedurale Sprache zur Programmierung von numerisch gesteuerten Maschinen.[6] Die *numerical control* (NC) wurde mit Whirlwind und der APT-Sprache zur *computerized numerical control* (CNC).

4 Casey Reas, Chandler Williams, *Form + Code in Design, Art and Architecture* (Princeton Architectural Press, New York, 2010), S. 43
5 Weisberg, 2008, Kap. 3, S. 1
6 Ebd., Kap. 3, S. 6

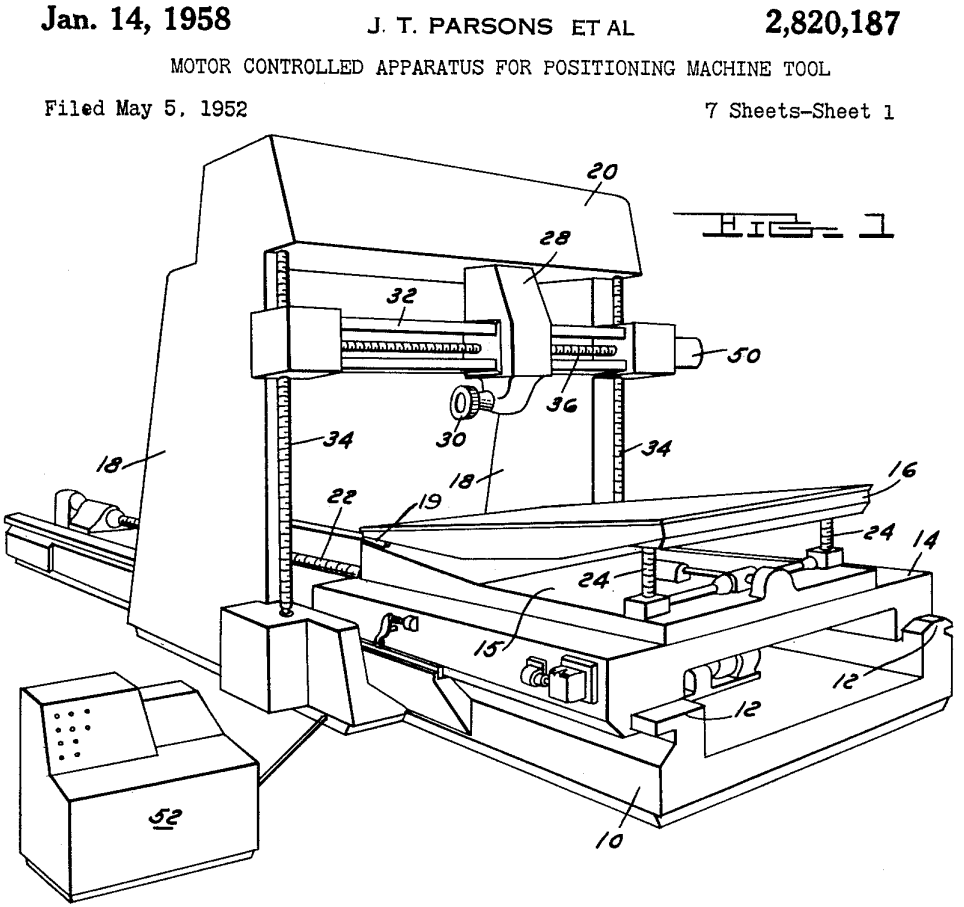


Abbildung 2.1: NC Fräse von John Parsons aus seinem Patentantrag von 1958

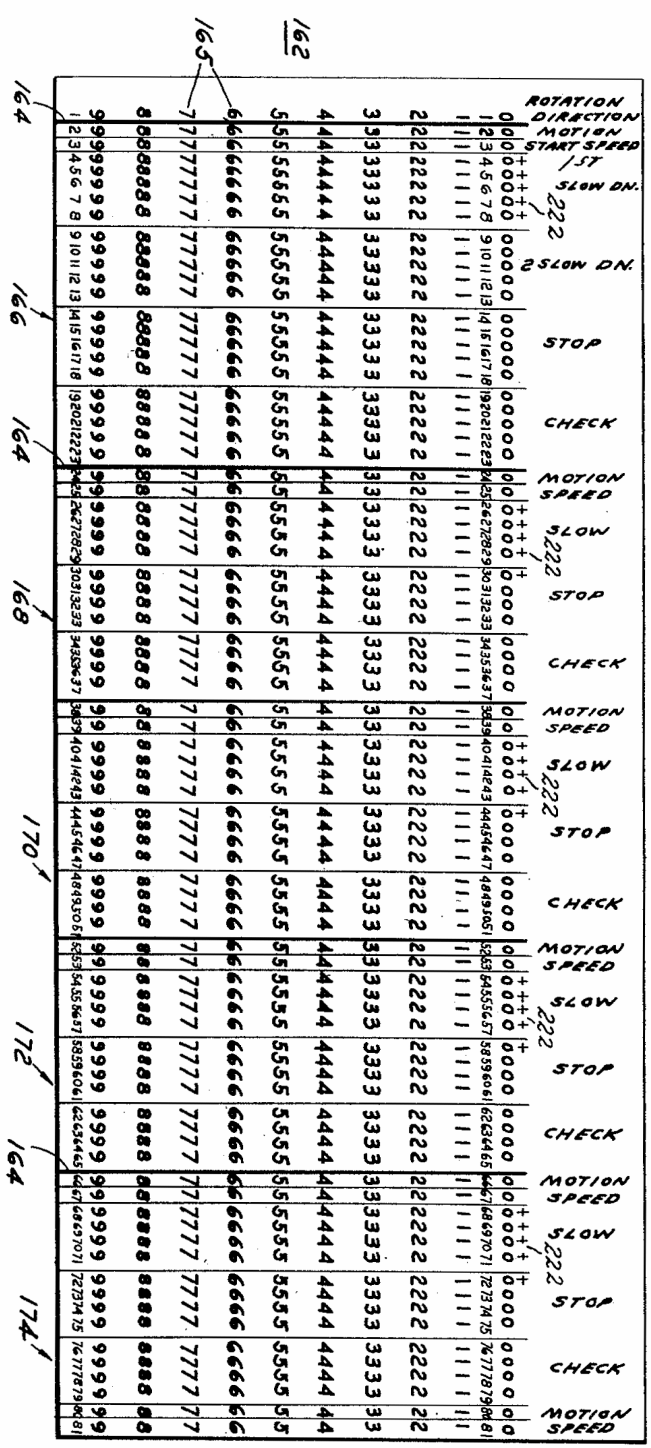


Abbildung 2.2: Lochstreifen zur Programmierung einer NC Fräse

2.2 Computer Aided Design

Schon in den 1960er Jahren erkannte die Industrie das Potential der neuen Werkzeuge. Es vor allem die Automobil- und Flugzeugindustrie, die die Entwicklung vorantrieben. Die Produktion von gebogenen Metallblechen für Autokarosserien oder Flugzeugrümpfe war ein zeitintensiver und kostspieliger Prozess: *„Sheet metal parts were manually designed using cross-section drawings from which templates were made. These were then used by patternmakers to produce a wood pattern that subsequently was used to machine stamping dies with a milling machine that copied the pattern. Many different people were involved in the process which was susceptible to error at each step.“* [7] Die Fertigung von Formteilen für die Produktion von Karosserie- oder Rumpfelementen konnte durch die neuen CNC Maschinen natürlich wesentlich vereinfacht werden, doch es fehlten vorerst noch die mathematischen Mittel, um die frei geformten Kurven und mehrfach gekrümmten Flächen dieser Bauteile mit der digitalen Sprache der neuen Maschinen zu beschreiben.

Viele der großen Auto- und Flugzeughersteller unterhielten in den 60er Jahren eigene Forschungsabteilungen, in denen zum Teil völlig isoliert von einander die Grundlagen für die mathematische Beschreibung von frei geformten Kurven und Flächen erarbeitet wurden. So entwickelte etwa James Ferguson, der für Boeing arbeitete, die Heremite-Kurve, Pierre Bézier bei Renault und Paul de Casteljau bei Citroën unabhängig von einander die Bézierkurve, und Carl de Boor bei General Motors schließlich die B-Spline Kurve[8] – die Grundlage der etwas später entwickelten Non Uniform Rational B-Splines (NURBS), also jener Freiformkurven und -flächen, die bis heute die zentrale Rolle bei der

7 Ebd., Kap. 2, S. 10

8 Claude Brezinski, Luc Wuytack, *Numerical analysis in the twentieth century*, in: Claude Brezinski, Luc Wuytack (Hrsg.), *Numerical analysis: historical developments in the 20th century* (Elsevier Science B. V., Amsterdam, 2001), S. 20f

Modellierung von gekrümmten Formen mit Hilfe des Computers spielen.

Die exakte mathematische Beschreibung beliebig gekrümmter Flächen war nun also möglich, und damit auch ihre Produktion mit numerisch gesteuerten Maschinen. Allerdings blieb noch ein Problem: die Bauteile wurden immer noch am Zeichenbrett entworfen. *„[A]ll relevant information was stored in the form of blueprints, and it was not clear how to communicate that information to the computer which was driving a milling machine. Digitizing points off the blueprints and fitting curves using familiar techniques such as Lagrange interpolation failed early on.“* [9]

Der Verlauf gekrümmter Bauteile wurde damals durch mehrere parallele Querschnittszeichnungen definiert, deren Kurven mithilfe von Schablonen oder Straklatten („splines“ in Englisch) konstruiert wurden. Doch die analogen, geometrischen Informationen dieser hand-gezeichneten Pläne waren nicht kompatibel mit den numerisch gesteuerten Herstellungsprozessen. Aus den Zeichnungen mussten erst digitale Daten gewonnen werden, mit denen dann die neuen Maschinen betrieben werden konnten.

1963 entwickelte Ivan Sutherland ein Programm namens Sketchpad[10], das nicht nur das erste Computerprogramm mit einem grafischen Userinterface war, sondern im Allgemeinen auch als das erste *computer aided design* (CAD) Progamm gilt. Sutherland entwickelte die Software auf dem TX-2 System des Lincoln Laboratory, das mit einer Kathodenstrahlröhre als Bildschirm und einem Lichtstift – dem Vorläufer der heutigen Computermaus – ausgestattet war. Sketchpad ermöglichte die Darstellung von Vektorgrafiken am Bildschirm und das Zeichnen und

9 Gerald E. Farin, *A History of Curves and Surfaces in CAGD*, in: Gerald E. Farin, Josef Hoschek, Myung-Soo Kim (Hrsg.), *Handbook of computer aided geometric design* (Elsevier Science B.V., Amsterdam, 2002) , S. 3

10 Weisberg, 2008, Kap. 3, S. 18ff

Abbildung 2.3: Arithmetische Einheit von Whirlwind (ca. ein Zehntel der gesamten Anlage). Das System hatte insgesamt 12 500 Vakuumröhren, konsumierte 150 000 Watt und beanspruchte eine Fläche von etwa 280 m².

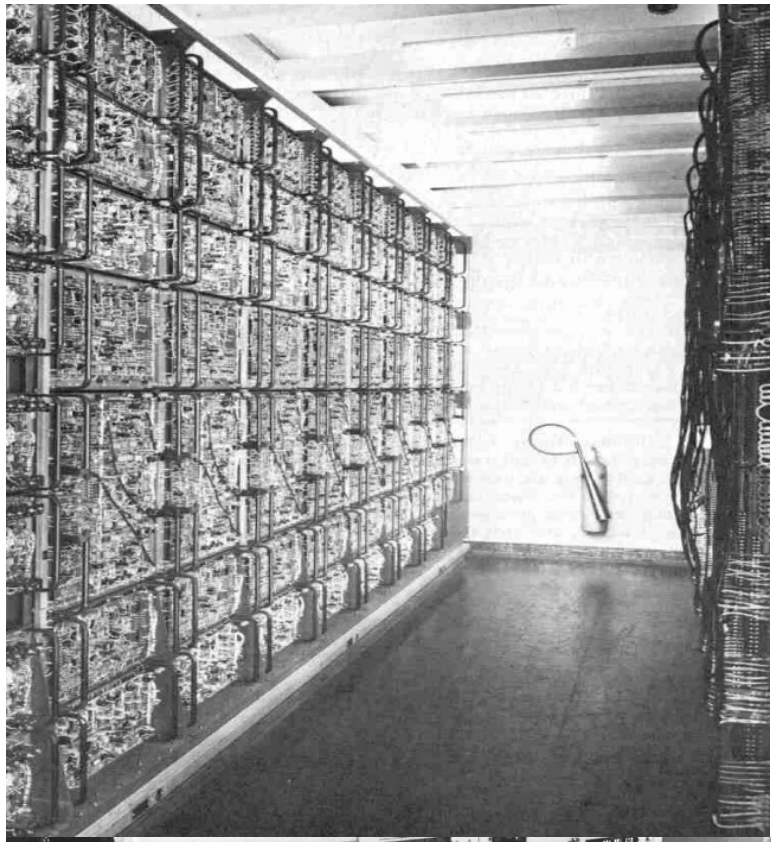


Abbildung 2.4: Ivan Sutherland bei der Präsentation seines Sketchpad Programmes auf dem TX-2 Rechner des MIT. Das System war mit einem CRT-Bildschirm, einem Lichtstift und einer Schalterkonsole für die Bedienung des Programmes ausgestattet.

Bearbeiten von zweidimensionalen Formen mithilfe des Lichtstiftes. Sutherlands Zeichenprogramm war in vielerlei Hinsicht wegweisend. Einige der verwendeten Konzepte, wie etwa das Kontextmenü oder die Gummibandlinie beim Zeichnen von Geraden, werden auch heute noch angewendet.

Mitte der 60er Jahre begannen mehrere große Auto- und Flugzeughersteller, Computerprogramme nach dem Vorbild von Sutherlands Sketchpad zu entwickeln, mit denen die Bauteilzeichnungen direkt grafisch am Rechner angefertigt werden konnten. Diese ersten industriellen CAD Programme waren aber mehr als nur Zeichenprogramme. Sie wurden nicht dafür gemacht, die Erstellung von Plangrafiken zu vereinfachen, sondern zur direkten Generierung von numerischen Daten für die neuen Fabrikationsprozesse. Die Kernfunktion der Programme war die Übersetzung von Informationen – von der analogen, kontinuierlichen Sprache der geometrischen Zeichnung in die digitale, diskrete Sprache der numerischen Formbeschreibung. Sie bildeten eine Brücke zwischen der sinnlich erfassbaren Welt der Plangrafik und der abstrakten Sprache der Fabrikationsmaschinen.

2.3 CAD in der Architektur

Im Laufe der 70er Jahre wurde die Weiterentwicklung der industriellen CAD Programme zunehmend ausgelagert. Es entstanden die ersten Unternehmen, die ausschließlich mit der Programmierung kommerzieller CAD Software beschäftigt waren und einige der einfachen hauseigenen Zeichenprogramme zu umfangreichen, kommerziell erfolgreichen CAD Paketen weiterentwickelten, wie etwa im Fall von Fords PDGS, Lockheeds CADAM oder Dassaults CATIA Software.[11]

Selbstverständlich wollten die spezialisierten Software-entwickler mit der Zeit ihren Absatzmarkt erweitern und begannen, ihre Programme auch Anwendergruppen außerhalb der Flugzeug- und Automobilindustrie schmackhaft zu machen. *„[CAD] systems were marketed predominately on the basis that they could reduce current operating costs. If you had a drafting department with 20 drafters, buy one of these systems, run it around the clock and you could get the same amount of work done with perhaps 10 or 12 people. In some cases, productivity improvements were truly spectacular, especially within organizations that did a lot of repetitive work.“*[12]

Voraussetzung für die Kommerzialisierung der CAD Programme waren natürlich die Fortschritte im Bereich der Computertechnologie. Die Rechner wurden – kurz gesagt – immer kompakter, billiger und gleichzeitig leistungsstärker, und mit der Verfügbarkeit von erschwinglichen grafischen Systemen wurden die Programme tatsächlich auch für Disziplinen interessant, die keinen direkten Bezug zu numerisch gesteuerten Fertigungsmethoden hatten.

Mit der Entwicklung der Mikroprozessoren und der Personal Computer Systeme fanden die CAD Programme auch in den Architekturbüros eine größere Verbreitung. Die Technologieadaption folgte dabei, wie in vielen anderen Arbeitsbereichen, der Logik des Ersatzes: wie zuvor schon die Schreibmaschinen durch Textverarbeitungsprogramme und die Spreadsheets der Buchhalter durch Tabellen-kalkulationsprogramme ersetzt wurden, wurden in der Architektur Lineal, Zirkel und das Zeichenbrett durch CAD Programme ersetzt.

Obwohl der Ersatz des Zeichenbretts in rein technologischer Hinsicht einen radikalen Bruch darstellte, änderte sich vorerst nichts Grundlegendes an der traditionellen Arbeitsweise. Die neuen Programme waren ja auch nicht dafür gemacht worden, den Arbeitsablauf zu revolutionieren oder gar den architektonischen Entwurfs-prozess zu beeinflussen – andernfalls wäre die CAD Technologie mit Sicherheit auf eine noch größere Opposition gestoßen.

Die Arbeit mit CAD Programmen war im Wesentlichen nichts anderes als die Fortsetzung der vormals analogen Zeichentätigkeit, nur diesmal nicht mit Stift und Zeichenbrett, sondern mit Maus und Computer. Das Ziel des CAD Zeichners blieb dasselbe, wie jenes seines analog arbeitenden Vorgängers, nämlich die Produktion von geometrischen Planungsdokumenten, von Grundrissen, Schnitten und Ansichten, und später auch von dreidimensionalen perspektivischen Darstellungen.

Man darf nicht vergessen, dass die Plangrafik spätestens seit der Renaissance, als sich das Berufsbild des entwerfenden Baukünstlers von jenem des ausführenden Baumeisters emanzipierte, *das* zentrale Medium architektonischer Entwürfe war, und es bis heute auch geblieben ist.

Bevor die Rolle des C*apomaestro* – des planenden und gleichzeitig ausführenden Baumeisters – zu Zeiten Leon Battista Albertis in Frage gestellt wurde, verlief der Bauvorgang noch *„weitgehend unabhängig von vorher festgelegten Projekten, in empirischer Auseinandersetzung mit dem Material und den im Lauf der Bauführung auftauchenden Einzelproblemen. Die architektonische Idee nahm also erst im emporwachsenden Bau selbst ihre endgültige Form an […]. Denn sowohl für die Raumkomposition wie für die Proportionen bestand in den meisten Fällen eine feste, in knappen Formeln auszudrückende Konvention. Eine zeichnerische Fixierung dieser Dinge war deshalb kaum erforderlich. Die Aufgaben der Architekturzeichnung begannen erst bei der Gestaltung der Teileinheiten, angefangen von der auf mehreren Pergamentstücken gezeichneten Fassade bis hinab zu einzelnen Pfeilern und Maßwerkmotiven, für die kleine Einzelblätter genügten, wenn man es nicht vorzog, solche*

11 Ebd., Kap. 13

12 Ebd., Kap. 2, S. 9

Bauteile unmittelbar in Originalgröße auf dem 'Reißboden' zu entwerfen.“ [13]

Alberti forderte Mitte des 15. Jahrhunderts in seiner Abhandlung *Über die Baukunst*, dass *„die Architekten keine Dinge machen, sondern dass sie Dinge entwerfen sollten“,* denn für Alberti war *„Bauen nämlich eine rein mechanische Operation, bar jeglichen intellektuellen Mehrwerts, deren einziger Zweck darin besteht, die Idee des Architekten zu materialisieren – sie auszudrucken, wie sie ist – in drei Dimensionen und im großen Maßstab“.*[14] In Albertis Vorstellung sollte es also eine klare Trennlinie zwischen Bauplanung und -ausführung geben. Die *Beschreibung* seiner Entwürfe sollte von nun an die Kernaufgabe des Architekten werden – er war der Autor von Planungsdokumenten, und die Ausführenden hatten das wortwörtlich autorisierte Werk möglichst genau umzusetzen. Damit kam der Plangrafik eine ungleich höhere Bedeutung zu, als zuvor noch bei der mittelalterlichen Arbeitsteilung. Gebaut wurde von nun an, was gezeichnet wurde (und was auch durch Zeichnungen repräsentiert werden konnte!), und gezeichnet wurde vornehmlich, was auch gebaut werden konnte. Die Zeichnung wurde, wie es Mario Carpo ausdrückt, zum *„unausweichlichen Flaschenhals, durch den die meisten Gebäude in der westlichen (Alberti'schen) Tradition erreicht werden sollten“.*[15]

Das Fundament der Planzeichnung war die Geometrie Euklids, die der bedeutende griechische Mathematiker bereits drei Jahrhunderte vor unserer Zeitrechnung in seinen 13 Büchern unter dem Titel *Stoicheia* („Elemente“) begründete, und die Instrumente der euklidischen Geometrie – Lineal und Zirkel – waren jahrhundertelang die wichtigsten Werkzeuge des Architekten.

„Euclid's Elements, as every schoolchild used to know, shows how to construct infinitely varied geometric compositions from points, straight lines, and arcs of circles. Parallels, perpendiculars, and congruencies figure prominently. And strictly speaking, the only instruments you need to explore this rigorously beautiful formal universe are a pencil, a straightedge, dividers, and compasses. Traditional drafting instruments comprise these simple, ancient tools, plus some more-modern inventions that save the trouble of explicitly executing the commoner Euclidean constructions. T-squares and parallel bars allow the ready production of parallels. Triangles facilitate the insertion of perpendiculars. Graph paper provides a modular framework of both parallels and perpendiculars. Graduated rulers and protractors simplify the subdivision of lines and angles. Tracing paper takes the tedious labor out of replicating shapes.“ [16]

Die Tradition der euklidischen Geometrie hatte die Werkzeuge der Architekten geformt, und die CAD Programme waren nun angetreten, diese traditionellen Zeichenwerkzeuge zu ersetzen, indem sie ihre Funktionen emulierten. Die Software war im Kern als Simulation des alten Zeichenbretts konzipiert. Ihre programmierten Funktionen waren nichts anderes als die Umsetzung der Möglichkeiten vormals analoger Zeichentätigkeiten innerhalb einer Computerapplikation. *„CAD systems were mostly employed as accurate and efficient replacements for traditional drafting instruments in the production of construction documents. They provided points, straight lines, arcs, and circles as basic graphic primitives, and their operations for inserting these primitives into drawings were directly based on fundamental theorems of Euclid. Grid and snap operations were the electronic equivalent of graph paper; copy operations enabled the rapid replication of existing shapes; and drawing 'layers' explicitly harked back to the days of transparent tracing-paper sheets.“* [17]

Die frühen CAD Applikationen waren vorerst rudimentäre zweidimensionale Zeichenprogramme mit bescheidener Funktionalität, aber die Weiterentwicklung der Programme durch unabhängige, spezialisierte Softwarefirmen sollte die Evolution der Applikationen allerdings wesentlich beschleunigen. Die Steigerung der Rechenleistung und die Weiterentwicklung der Grafikhardware erlaubte bald schon die räumliche Darstellung von Objekten, aber für die Bearbeitung von dreidimensionalen Körpern mussten erst noch neue Konzepte gefunden werden. Noch in den 70er Jahren wurde der Ansatz der Constructive Solid Geometry[18] entwickelt, mit dem komplexere Geometrien durch Kombination verschiedener einfacher Volumenkörper wie Kegel oder Würfel zusammengesetzt werden konnten. Andere Verfahren erlaubten die Erzeugung von dreidimensionalen Geometrien mit Konzepten, die von physischen Herstellungsverfahren inspiriert waren, wie etwa die Extrusion zweidimensionaler Polygone oder die Rotation von Querschnitten um eine Achse.

Der Aufbruch der CAD Programme in die dritte Dimension brachte aber alles andere als immersive Entwurfswerkzeuge hervor. Die umständlichen Modellierungskonzepte machten ein intuitives Arbeiten in drei Dimensionen unmöglich. Anstatt die Mittel des Entwerfers zu erweitern, wurden sie durch die Einschränkungen bei der Erzeugung und Bearbeitung dreidimensionaler Körper noch weiter limitiert.

Die anfängliche Skepsis vieler Architekten gegenüber der neuen Technologie war sehr wohl begründet, denn die CAD

13 Robert Oertel, *Wandmalerei und Zeichnung in Italien. Die Anfänge der Entwurfszeichnung und ihre monumentalen Vorstufen*, in: *Mitteilungen des Kunsthistorischen Instituts in Florenz 5* (1937/40), S. 217-234
Zitiert nach Christoph Schindler, *Die Standards des Nonstandards*, in: Urs Hirschberg, Daniel Gethmann, Ingrid Böck, Harald Kloft (Hrsg.), *Graz Architektur Magazin 06* (Springer Verlag, Wien, 2010), S. 185

14 Mario Carpo, *Das Digitale, „Mouvance“ und das Ende der Geschichte*, in: Urs Hirschberg, Daniel Gethmann, Ingrid Böck, Harald Kloft (Hrsg.), *Graz Architektur Magazin 06* (Springer Verlag, Wien, 2010), S. 25

15 Ebd., S. 25

16 William J. Mitchell, *Roll Over Euclid: How Frank Gehry Designs and Builds*, in: J. Fiona Ragheb (Hrsg.), *Frank Gehry, Architect* (Guggenheim Museum Publications, New York, 2001), S. 352f

17 Ebd., S. 354

18 Weisberg, 2008, Kap. 2, S. 13

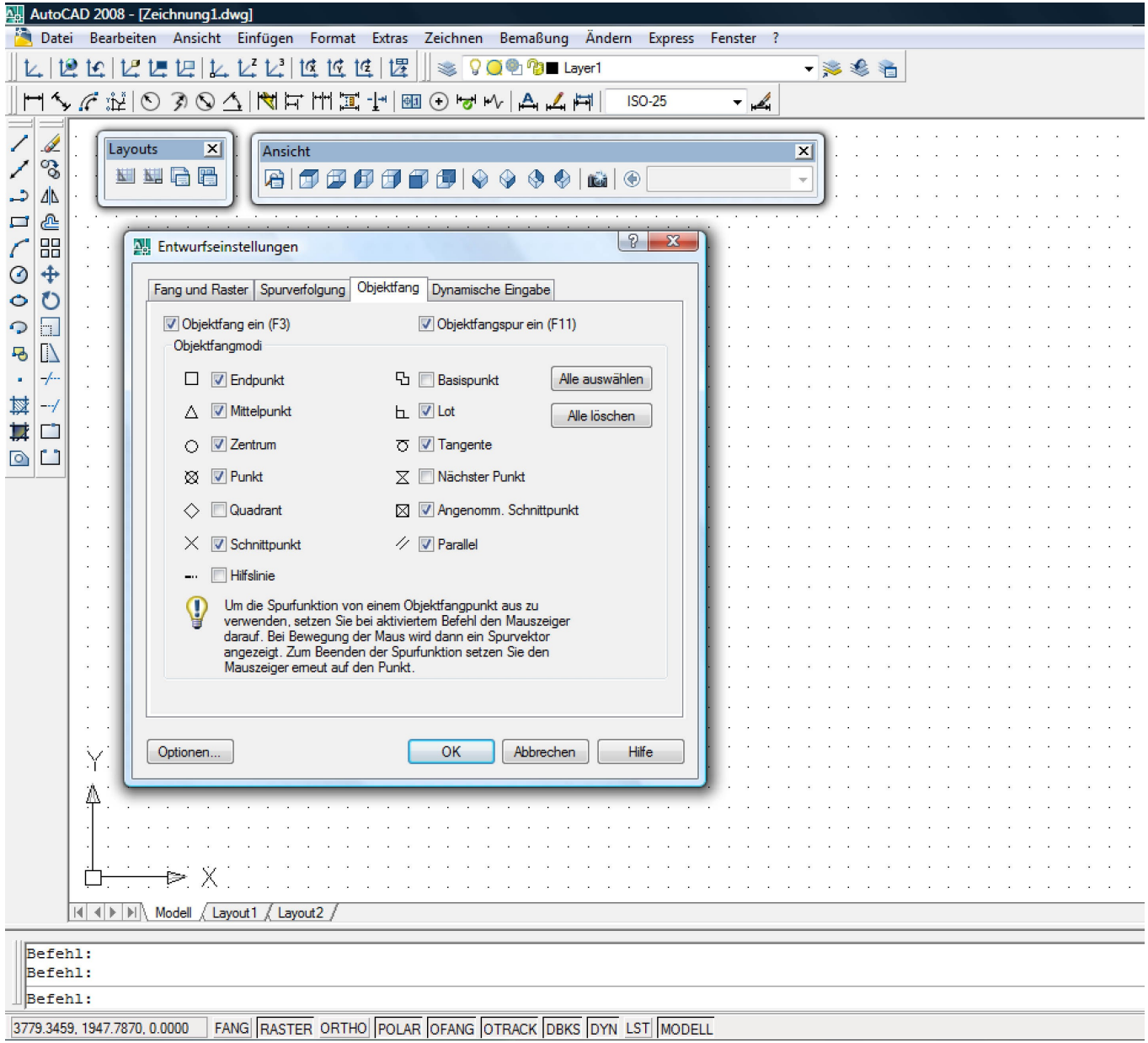


Abbildung 2.5: Das Erbe des Zeichenbretts und die Fortschreibung der euklidischen Tradition in den Werkzeugen von AutoDesks AutoCAD Software

Anwendungen boten lange Zeit keine großen Vorteile gegenüber der traditionellen Arbeitsweise, außer eben die Effizienzsteigerung bei der Erstellung von Planzeichnungen. Und es war genau diese Effizienzsteigerung traditioneller Methoden, die geraume Zeit die Suche nach alternativen Konzepten für den Einsatz digitaler Technologien in der Architektur marginalisierten.

2.4 Nonstandard Architektur

Die erste wirkliche Neuerung brachte die Integration nichteuklidischer Geometrien, allen voran die schon erwähnten NURBS Flächen, ins Formenrepertoire der CAD Programme. Bei der Visualisierung und Bearbeitung von unregelmäßigen, gekrümmten Flächen, die räumlich schwer zu erfassen und mit Lineal und Zirkel kaum zu konstruieren sind, zeigten sich die Vorzüge der neuen Technologie. Die interaktive Darstellung der Computerprogramme machte es möglich, die komplexen Formen von jedem beliebigen Blickwinkel aus zu beurteilen, was zuvor nur mit physischen Modellen möglich war. Die Manipulation der Geometrien durch das simple Verschieben von NURBS-Kontrollpunkten war kinderleicht, darüber hinaus war die interaktive Bearbeitung der Flächen in Echtzeit möglich, Änderungen an der Geometrie wurden in Sekundenbruchteilen am Bildschirm dargestellt. Die Integration der NURBS Flächen stellte zum ersten Mal eine echte Erweiterung der Möglichkeiten im Vergleich zu den traditionellen Arbeitsmethoden dar.

Interessanterweise waren es die Animationsprogramme der Filmindustrie wie 3D Studio Max, Maya oder Softimage (Abbildung 2.6), die diese Erweiterung noch vor den in der Architektur verwendeten CAD Anwendungen vollzogen. In den 90er Jahren begannen eine Reihe von Architekten mit diesen Programmen, die für einen völlig andere Disziplin konzipiert waren, zu experimentieren, was allerdings auch nicht unbedingt ungewöhnlich war, denn die meisten Büros

arbeiteten ohnehin nicht mit fachspezifischen Programmen.

Architekten waren ja nicht die Einzigen, deren Arbeitsgerät das Zeichenbrett gewesen war, das es zu ersetzen galt. So wurden in den 80er und 90er Jahren die Hoch- und Tiefbaubüros mit der selben Software bedient, wie die Fahrzeug- und Maschinenbauer, die Verkehrsplaner, Vermessungsbüros und Produktdesigner.

Im Gegensatz zu jenen CAD Programmen, die für die Erstellung zweidimensionaler Plangrafiken entwickelt worden waren, waren die Animationsprogramme von vornherein für die Arbeit in drei Dimensionen konzipiert. Und NURBS Flächen – die Grundlage der Modellierung organischer Formen – waren für die Programme der Filmindustrie natürlich ein unverzichtbarer Teil ihres geometrischen Repertoires.

Für einige der experimentierfreudigen Architekten bedeuteten die Möglichkeiten der digitalen Modellierwerkzeuge damals eine Versuchung, der nur schwer zu widerstehen war. *„And sure enough an extraordinary feeling of power consumes any architect to whom the modelers of CAO [Conception Assisteé par Ordinateur] give the means to generate surfaces that he or she generally cannot design with a ruler and compasses.“* [19]

Die Beliebtheit der geschwungenen Flächen ab den frühen 90er Jahren lässt sich aber nicht alleine durch die Verfügbarkeit der adäquaten Bearbeitungsmethoden begründen. *„[T]he empathy between digital technologies and round forms that characterized the early phases of the digital revolution in architecture was not a quirk of history […]. Architectural deconstructivism, which climaxed in the late eighties or early nineties, had fostered a fractured environment of dissonance and disjunction, parataxis and angularity. But, as art historians have known at least since*

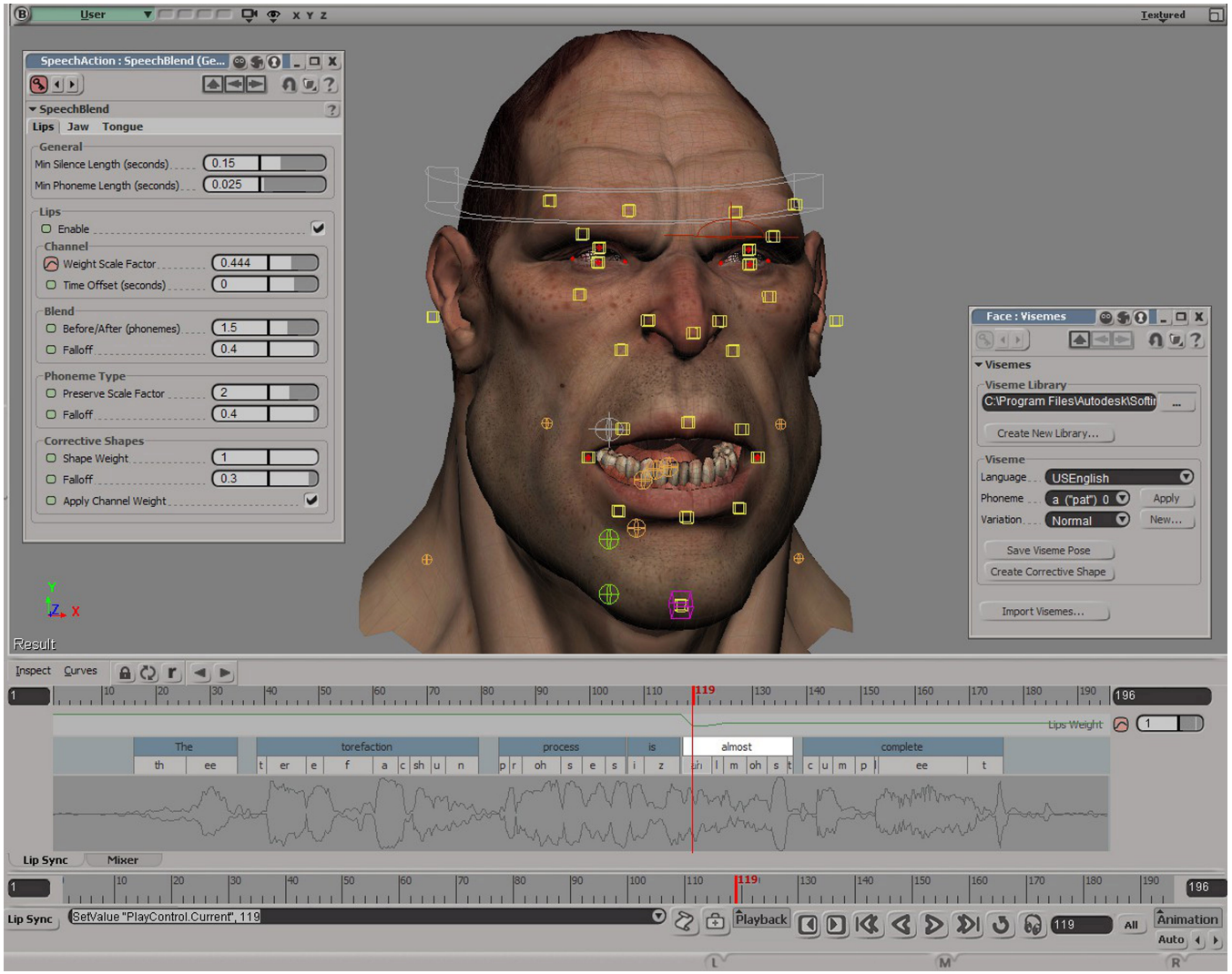


Abbildung 2.6: Virtuelle Trickfilmfigur in Softimage

19 Patrick Beaucé, Bernard Cache, *Towards a Non-standard Mode of Production*, in: Bernard Leupen, René Heijne, Jasper van Zwol (Hrsg.), *Time-based Architecture* (010 Publishers, Rotterdam, 2005), S. 116

Abbildung 2.7: „Friendly Alien“, Kunsthaus Graz (Peter Cook und Colin Fournier)

the time of Heinrich Wölfflin, forms have a well-known tendency to swing from the angular to the curvilinear, from parataxis to syntax. The nineties were no exception: after the excesses of decontructivist angularity, a rebound was inevitable.“ [20]

Und dieser „rebound“ wurde überschwänglich zelebriert. Die formale Freiheit, die die CAD Programme der 90er Jahre mit sich brachten, gebar tatsächlich einige eindrückliche Werke, die Meisten waren zumindest eine Zeit lang aufsehenerregend.

Doch was bedeutet die Freiheit der „freien“ Formen?

Der Prozess des Bauens war vor dem Beginn der Industrialisierung ein von handwerklichen Tätigkeiten geprägter Vorgang. Auf der Basis von handgezeichneten Plänen wurden Bauteile manuell hergestellt, die dann auf der Baustelle zusammengefügt wurden. Das Wissen über Produktions- und Konstruktionsmethoden wurde von Generation zu Generation weitergegeben und die tradierten Prozesse blieben immer wieder über längere Zeitperioden fast unverändert. Die konstruktive und formale Sprache von Steinbauten etwa *„hat sich über eine jahrtausendelange Baukultur entwickelt, immer vor dem Hintergrund der mangelnden Zugfestigkeit des Steins.“* [21]

Mit dem Beginn der Industrialisierung beschleunigten sich die Entwicklungen. Zuvor waren die in der Natur vorkommenden Werkstoffe Holz und Stein die wichtigsten Baumaterialien, nun wurde es möglich „*neue, so genannte künstliche Werkstoffe herzustellen und bereits bekannte Materialien wie Eisen durch verbesserte Herstellungs-*

20 Mario Carpo, *The Demise of the Identical - Architectural Standardization in the Age of Digital Reproducibility*, in: *Refresh - First International Conference on the Histories of Media Art, Science and Technology*, Conference Papers (Banff New Media Institute, http://www.banffcentre.ca/bnmi/programs/archives/2005/refresh/conference_docs.asp, 2005), S. 1f

21 Harald Kloft, *Logik oder Form*, in: Urs Hirschberg, Daniel Gethmann, Ingrid Böck, Harald Kloft (Hrsg.), *Graz Architektur Magazin 06* (Springer Verlag, Wien, 2010), S. 110

verfahren in technologisch neue Qualitäten zu überführen. Das Finden des synergetischen Zusammenwirkens von Material und Form war dabei immer ein iterativer Prozess.“ [22]

Die architektonische Form richtete sich seit jeher nach den Bedingungen ihrer Herstellung – den Eigenschaften verfügbarer Baumaterialien und den handwerklichen oder maschinellen Verfahren, um diese Materialien zu bearbeiten.

In der Vergangenheit waren es dann oft neue Materialentwicklungen oder Konstruktionsweisen, die einen Wandel der Formensprache auslösten, aber mit dem Aufkommen der frei geformten Flächen in den 90er Jahren drehte sich der Prozess um. Nun waren es die digitalen Entwurfswerkzeuge, die eine neue Formensprache in die architektonische Gestaltung einführten, aber es fehlten die geeigneten Mittel um die neuen Formen zu Materialisieren.

Dank der digitalen CAD Programme war es ein Kinderspiel auch die komplexeste Geometrie zu modellieren, aber diese immateriellen Formen dann aber in die Realität zu exportieren, war eine ganz andere Geschichte.

„The man in the street still has no idea of this, but delineating the control points of a NURBS surface in order to generate a fluid surface is now within the range of any user after an apprenticeship of just half an hour. And that's how it should be. That on the other hand it may then be a question of controlling these surfaces, of modifying them by intervening on their coordinates, of giving them a thickness and of fabricating them, that's a whole new ballgame: namely, to shift the problems onto someone else while multiplying the budget.“ [23]

Architektur, soll sie denn gebaut werden, kann nicht gänzlich frei sein, sie ist immer an die Bedingungen ihrer Materialisierung gebunden. Die formale Freiheit, die die

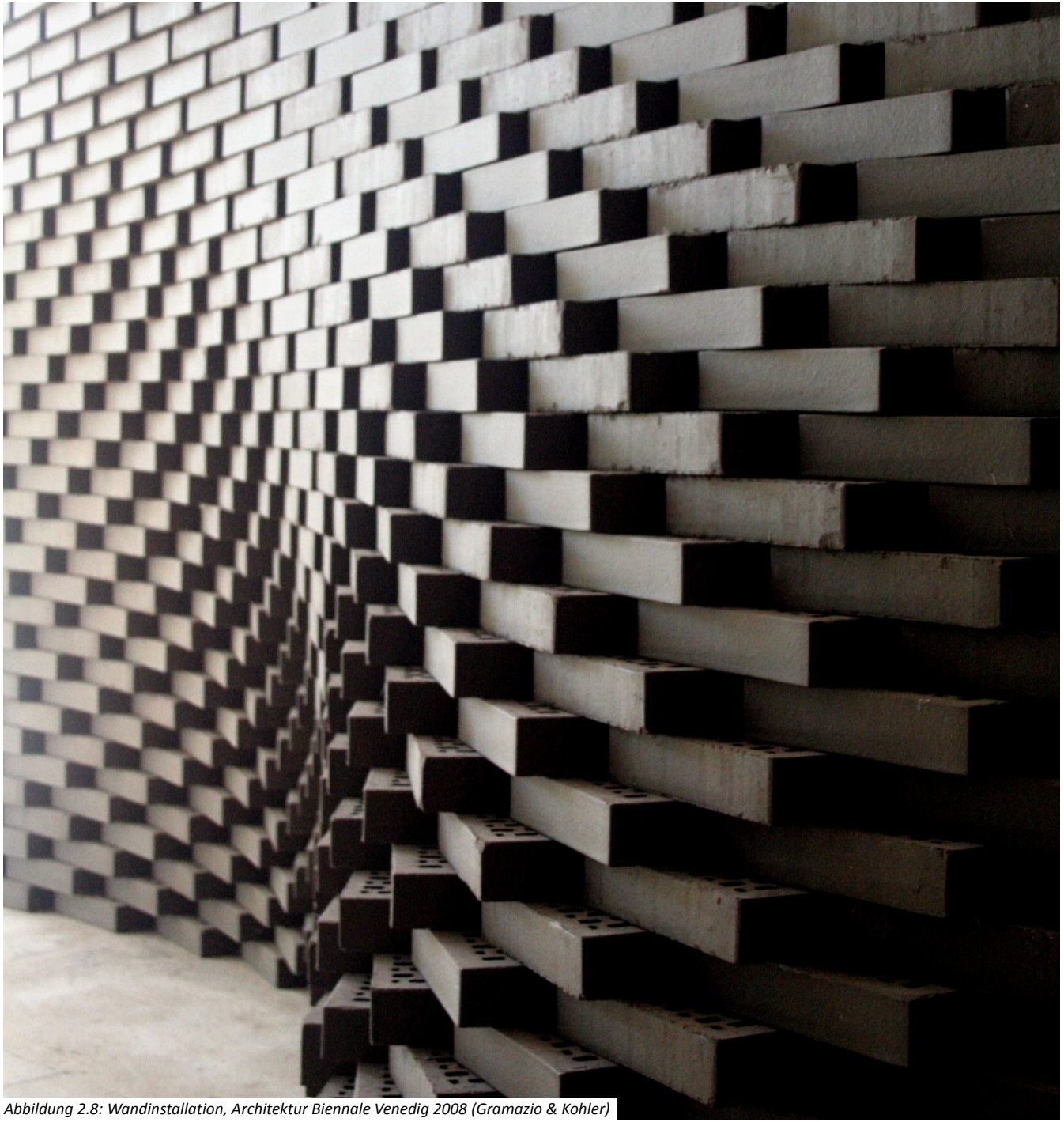


Abbildung 2.8: Wandinstallation, Architektur Biennale Venedig 2008 (Gramazio & Kohler)

22 Ebd.
23 Beaucé, Cache, 2005, S. 116

Abbildung 2.9: Der fünfachsige Roboterarm bei der Arbeit

digitalen Modellierungswerkzeuge mit sich brachten, hatte (und hat leider immer noch) einen hohen Preis, wenn sie in die Realität exportiert werden soll.

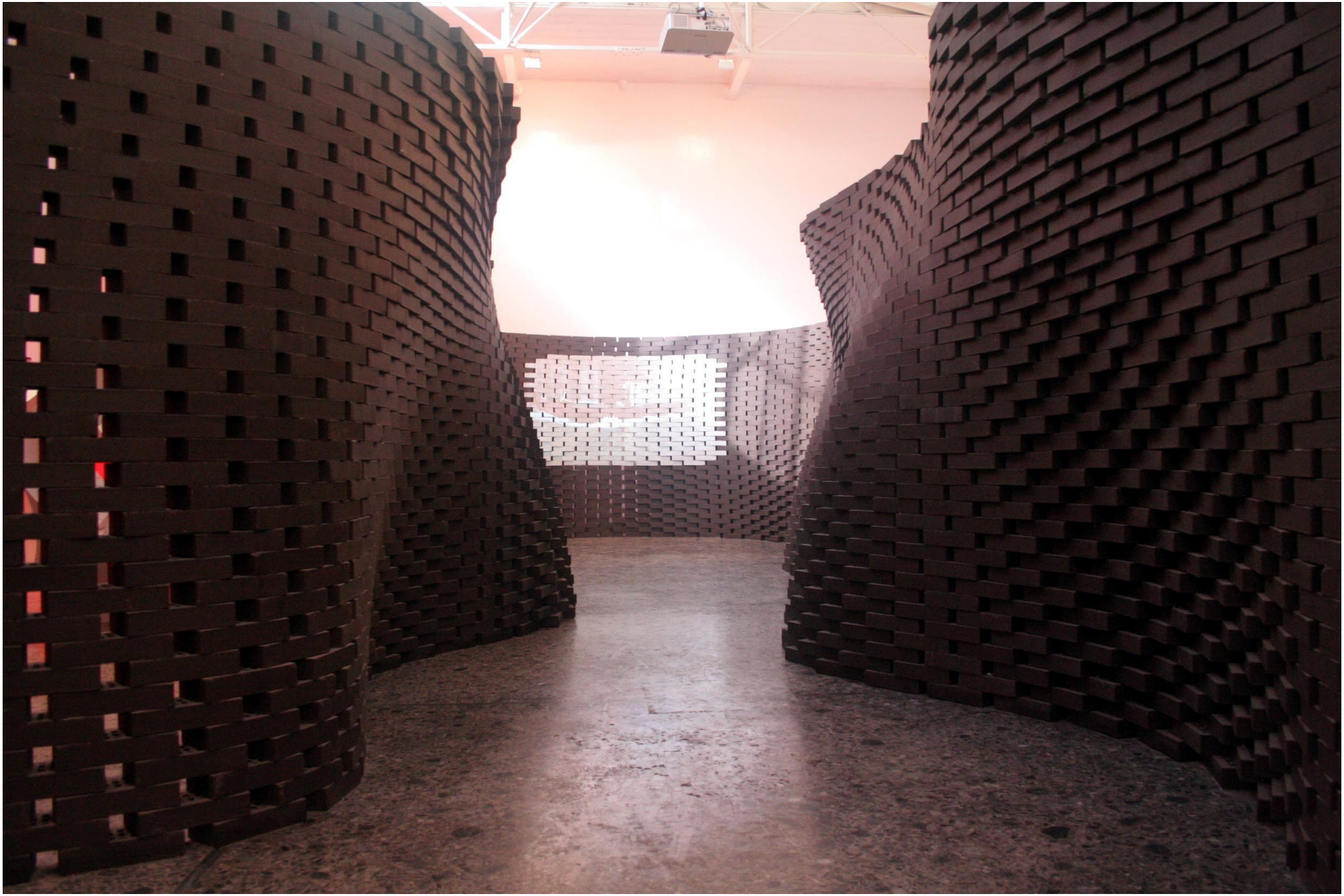
In den letzten 20 Jahren wurden tatsächlich eine Reihe von spektakulären Projekten realisiert, oft mit spektakulären Budgets. Aber in diesem Fall ist der Nonstandard in der Architektur nichts anderes, als die neueste Form der Distinktion einer Klientel, die die Möglichkeit hat, mehr als nur ein Standardbudget aufzubringen.

Dass es auch anders geht, zeigen digital arbeitende Büros wie jenes von Fabio Gramazio und Matthias Kohler, die nicht müde werden zu betonen, wie wichtig es ist, den digitalen Materialisierungsprozess von vornherein in ein Entwurfskonzept mit einzubeziehen: *„It is vital to incorporate the fabrication logic with the design process in order to fully benefit from the potential that lies within digital fabrication techniques. Digital fabrication does not only open up possibilities, but also lays constraints onto the design. Incorporated at the very beginning these constraints can act as guidance and lead to novel design strategies.“* [24]

Gramazio Kohler nehmen den Fabrikationsprozess als Ausgangspunkt ihrer Arbeiten. Bei ihren Projekten geht es nicht primär um Formen, sondern um den Weg der Formen zu ihrer Materialisierung.

Die Unvoreingenommenheit ihres Zuganges zum Thema zeigt sich in ihren Wandinstallationen, zum Beispiel jener von der Architektur Biennale 2008 in Venedig (Abbildung 2.8), für deren Errichtung sie einen Industrieroboterarm benutzen, der nach einem vorprogrammierten Schema die Ziegelsteine der geschwungenen Mauer positioniert. Dabei adaptieren sie die Konstruktionsmethode, die mehr als 10.000 Jahre alt ist, gemäß den Beschränkungen der

24 Tobias Bonwetsch, Ralph Bärtschi, Daniel Kobel, Fabio Gramazio, Matthias Kohler, *Digitally Fabricating Tilted Holes. Experiences in Tooling and Teaching Design*, in: eCAADe 25 Conference Proceedings (http://cumincad.scix.net, 2007), S. 798



Maschine. Denn der Roboterarm kann keinen Zement verarbeiten, stattdessen befestigt er die Ziegel mit Klebstoff aneinander. So entsteht eine Ziegelwand mit ganz neuen tektonischen Eigenschaften. Da die Ziegel durch den Klebstoff zugfest miteinander verbunden sind, müssen sie nicht, wie sonst üblich, in geraden Linien übereinander gelegt werden, sondern können versetzt übereinander angeordnet werden, ohne dass man befürchten muss, dass die Mauer durch einen seitlichen Stoß zusammenfallen würde.

Nun, da die erste Begeisterungswelle der formalen Überschwänglichkeit vorbei ist, scheint sich beim Einsatz digitaler Technologien allgemein eine neue, vielleicht etwas reifere Herangehensweise durchzusetzen, die die „Top-Down" Logik des Form-gebens durch eine „Bottom-Up" Logik des Form-findens ersetzt.

Es mag sicherlich formale Ähnlichkeiten zwischen den Arbeiten von Gramazio Kohler und jenen von zum Beispiel Frank Gehry geben, aber es gibt einen enormen Unterschied bei der Herangehensweise und der Art des Einsatzes digitaler Methoden. *„Gehry represents a more traditional, 'Postmodern' approach towards design, where the architect is perceived as the genius creator who imposes form on the world in a top-down process, and the primary role of the structural engineer is to make possible the fabrication of the designs of the master architect, as close as possible to his or her initial poetic expression."* [25]

Gramazio Kohler zeigen in ihren Arbeiten einerseits, dass in der Verbindung digitaler Fabrikationsprozesse mit vertrauten, traditionellen Konzepten völlig neue Qualitäten zu Tage treten können, vor allem aber zeigen sie, dass ihre „Bottom-Up" Logik, den Herstellungsprozess als Grundbedingung des digitalen Entwerfens anzusehen, vielversprechendere und interessantere Ergebnisse liefert, als der immer noch beliebte Ansatz, erst mit der digitalen Form zu beginnen, und das Problem ihrer Materialisierung dann anderen zu überlassen.

25 Neil Leach, *Digital Morphogenesis*, in: Architectural Design, *Theoratical Meltdown*, Vol. 79, Issue 1 (Wiley & Sons, London, 2009), S. 34

2.5 Code

Der wesentliche Unterschied zwischen dem Ansatz von Gramazio Kohler und der, wie Neil Leach schreibt, immer noch postmodernen Arbeitsweise Gehrys, liegt im Mittel der Architekturrepräsentation. Gehry arbeitet *„between the imaginary of sketches and the physicality of models. The computer is thus merely used as a notational device, translating these models from the physical into the digital realm"*.[26] Gehrys physische Modelle werden zwar digitalisiert, liegen dann aber als reine geometrische Information am Computer vor.

Gramazio Kohler dagegen arbeiten mit parametrischen Algorithmen, formuliert in einer Programmiersprache. Das Mittel ihrer Architekturbeschreibung ist der Sourcecode, ein performativer Text, wenn man so will.

Digitale Technologien stellen die dominante Rolle der rein geometrischen Entwurfsbeschreibung infrage. Eine Linie, auch wenn sie mit einer CAD Anwendung gezeichnet wurde und deshalb in digitaler Form in den Datenstrukturen des CAD Programms vorliegt, ist trotzdem nichts anderes als eine Linie – reine geometrische Repräsentation. Sie hat keine Information über ihren Kontext, sie könnte die Stufe einer Fluchttreppe repräsentieren oder das Blatt einer Türe. Normalerweise sind die einzigen Informationen, die sie hat, ihr Anfangspunkt, ihr Endpunkt und dass sie eine Linie ist.

Würde die Linie wissen, dass sie die Stufe einer Fluchttreppe repräsentiert, könnte sie vielleicht auch wissen, dass die Stufe eine gewisse Mindestbreite haben muss, und wenn sie diese Mindestbreite unterschreitet, könnte sie vielleicht auf diesen Missstand aufmerksam machen. Als Türblatt könnte sie sich bei der Größenänderung der Maueröffnung automatisch anpassen, oder umgekehrt, bei der Vergrößerung der Türe die Mauer über den erhöhten Platzbedarf informieren.

Das ist eine Form von Selbstorganisation, man nennt es assoziatives Verhalten. Die meisten CAD Programme sind mittlerweile mit assoziativen Objekten ausgestattet, man denke etwa an die Multilinienobjekte von AutoCAD oder assoziative Maßbeschriftungen, die sich bei Größenänderungen von Bauteilen automatisch aktualisieren, aber die Möglichkeiten von Selbstorganisation und assoziativem Verhalten bei programmierten Objekten gehen weit über diese einfachen Konzepte hinaus, was ich hoffe, mit der vorliegenden Arbeit beweisen zu können.

26 Ingeborg M. Rocker, *Architectures of the Digital Realm: Experimentations by Peter Eisenman / Frank O. Gehry*, in: Jörg H. Gleiter, Norbert Korrek, Gerd Zimmermann (Hrsg.), *Die Realität des Imaginären - Architektur und das digitale Bild* (Universitätsverlag der Bauhaus-Universität, Weimar, 2008), S. 257

3. Object-Scaping

[N]ew forms of architecture will not emerge as a result of the effects achieved by ever more pliant, fluid, complex, and heterogeneous shapes or architectural forms, but rather with the development of more pliant, complex, and heterogeneous forms of architectural practice – with architectural practices supple enough to be formed by what is outside or external to them, yet resilient enough to retain their coherence as architecture.

Michael Speaks, *Folding toward a New Architecture*[1]

Wie ich in der Einführung schon erwähnt habe, bildet ein digitialer Produktionsprozess, die Object-Scaping Methode, den Ausgangspunkt meiner Arbeit. Dieses Materialisierungskonzept möchte ich am Anfang dieses Kapitels vorstellen. Aus den Bedingung des Herstellungsprozesses heraus habe ich ein parametrisches Modell entwickelt, das in der zweiten Hälfte dieses Kapitels erläutert wird. Das Modell bildet die Grundlage für ein Computerprogramm, das erstens zur Visualisierung und Bearbeitung von Objekten dient, die mit der vorgestellten Methode produziert werden sollen, und zweitens die digitalen Daten für den Produktionsprozess generiert. Das Programm wird im folgenden Kapitel vorgestellt.

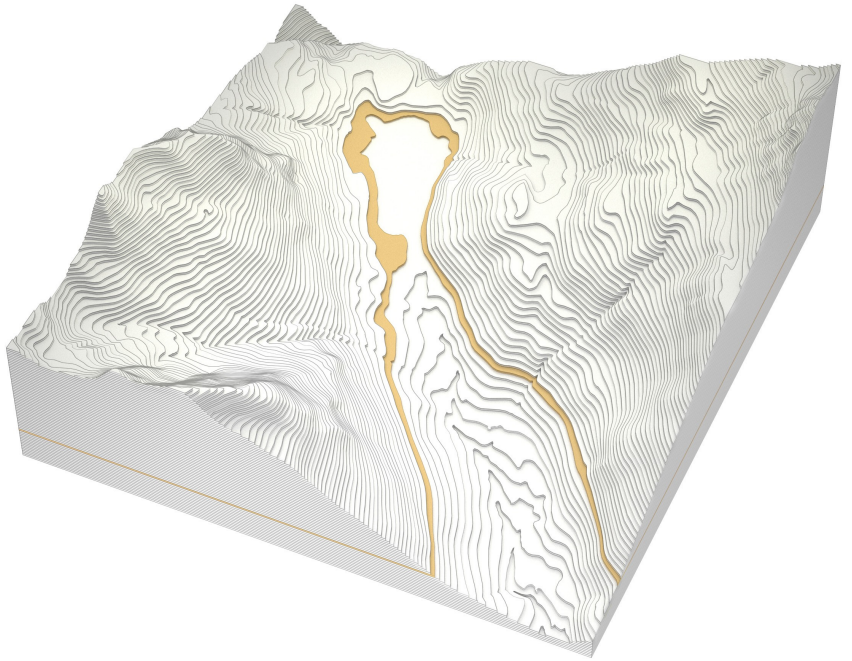
Um die Projektbeschreibung zu komplettieren, habe ich im Anhang die Umsetzung der wichtigsten Konzepte dieses und des folgenden Kapitels als Programmcodeauszüge beigefügt. Die Verweise auf die Sourcecodes sind an den entsprechenden Stellen im Text mit [SC] und der jeweiligen Nummer des Sourcecodeauszugs markiert.

3.1 Die Landschaftsmodellmethode

Wahrscheinlich kommen die meisten Architekturstudenten im Laufe ihrer Ausbildung einmal in die Verlegenheit, ein Landschaftsmodell bauen zu müssen. Diese Modelle sind ein wertvolles Instrument beim Entwurf von Gebäuden mit starkem Bezug zu ihrer natürlichen Umgebung. Mithilfe der

1 Michael Speaks, *Folding toward a New Architecture*, in: Bernard Cache, in: *Earth Moves. The Furnishing of Territories* (MIT Press, Cambridge, 1995), S. XVI

Abbildung 3.1: Landschaftsmodell massiv



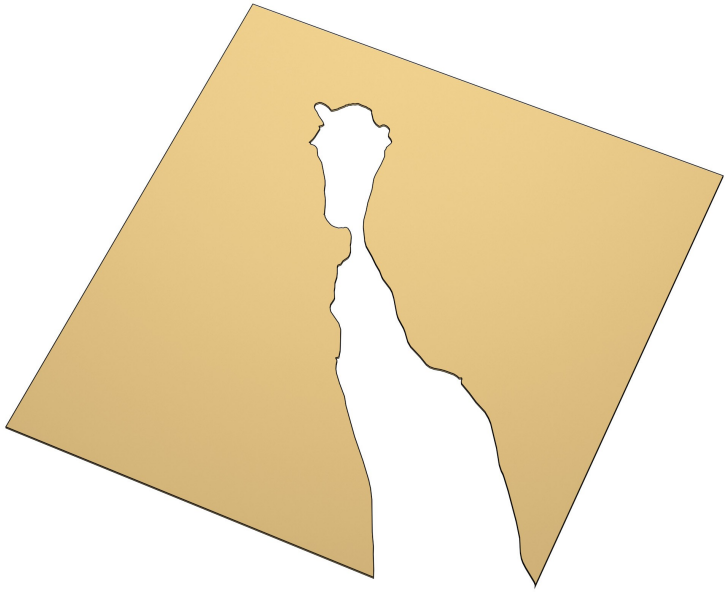


Abbildung 3.2: Eine geschnittene Höhenschichte für das Landschaftsmodell oben

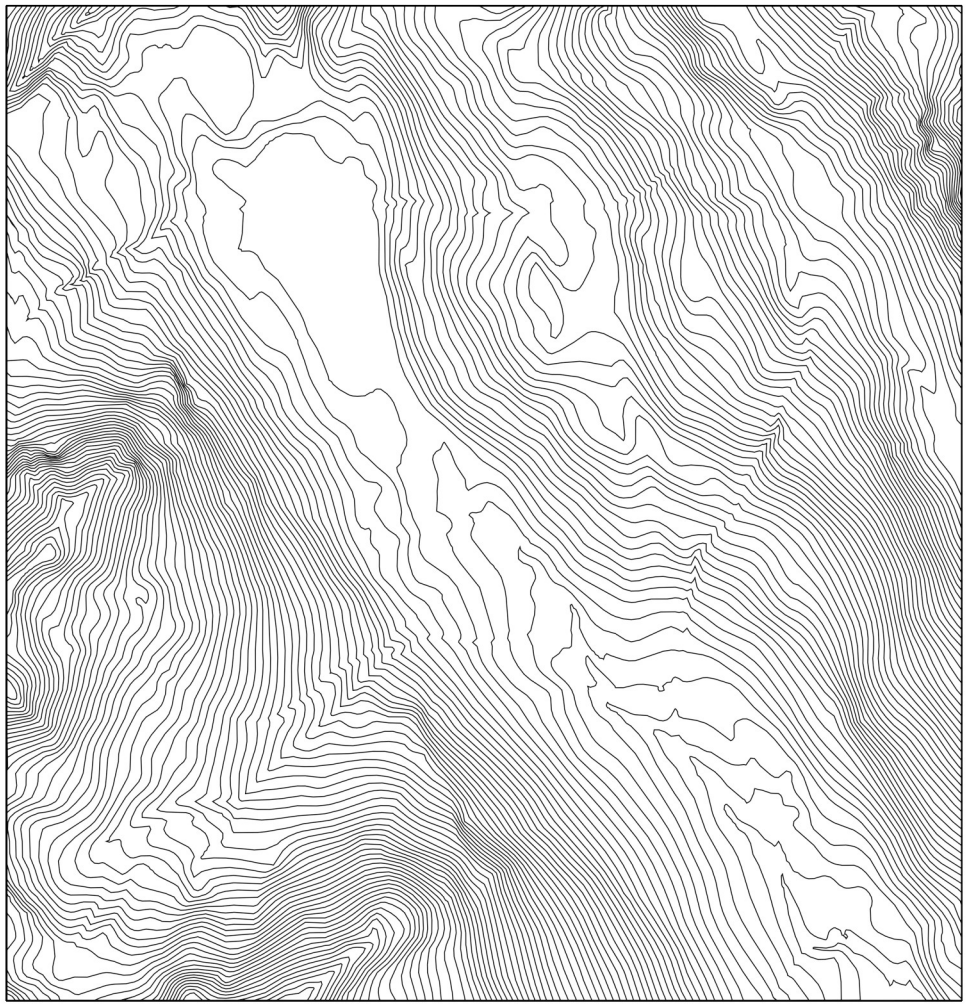
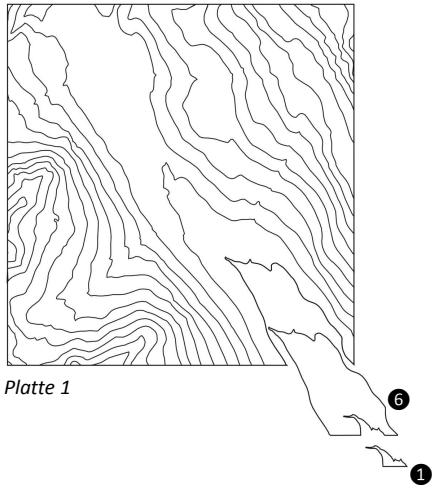


Abbildung 3.3: Höhenschichtlinien des Modells



Platte 1

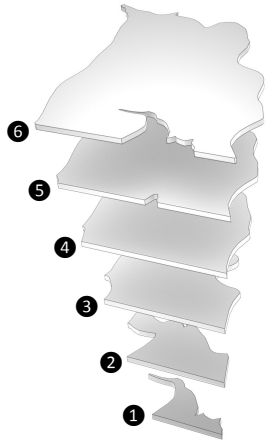
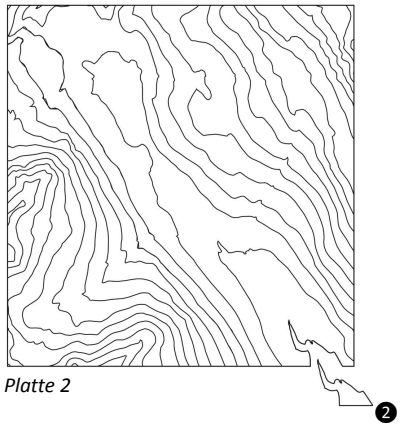
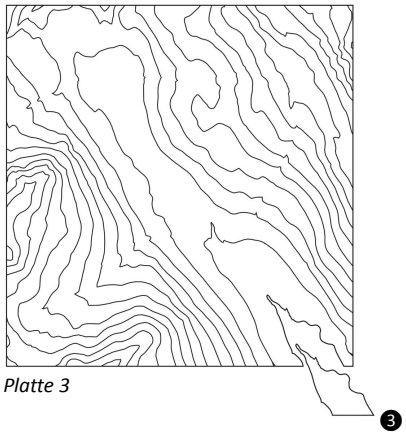


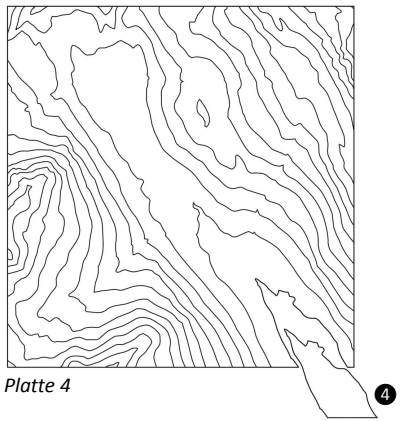
Abbildung 3.4: Anordnung der geschnitten Teile



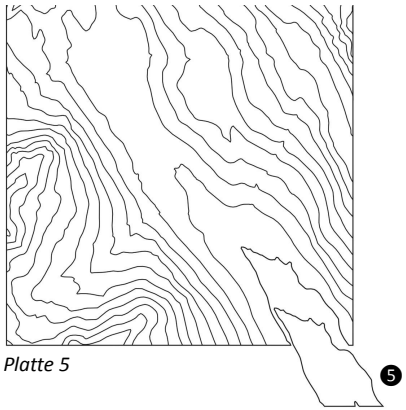
Platte 2



Platte 3



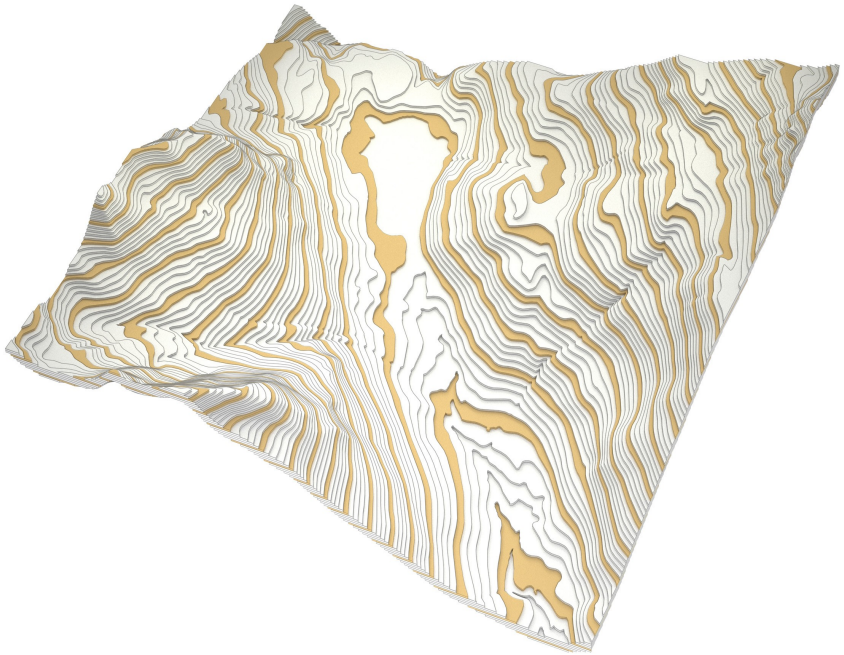
Platte 4

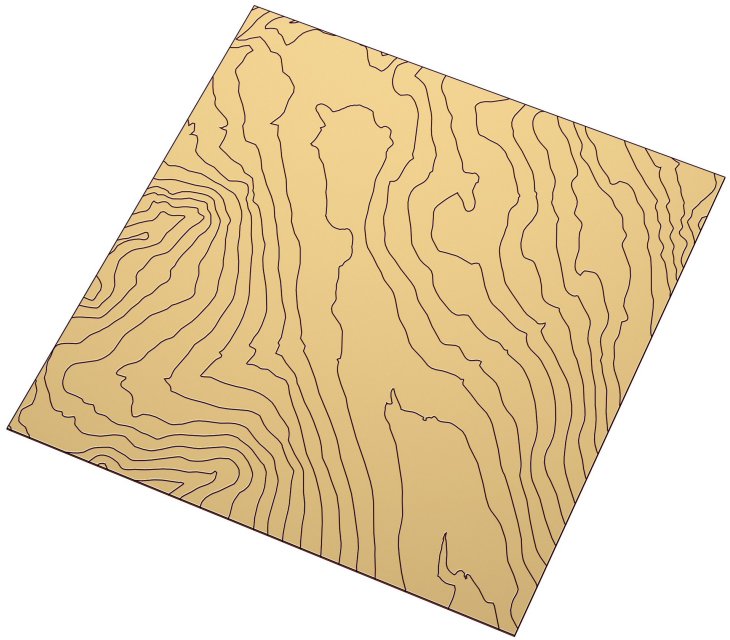


Platte 5

Abbildung 3.6: Schnittmuster der Modellbauplatten für das Landschaftsmodell in Abbildung 3.5

Abbildung 3.5: Landschaftsmodell aus 5 Platten





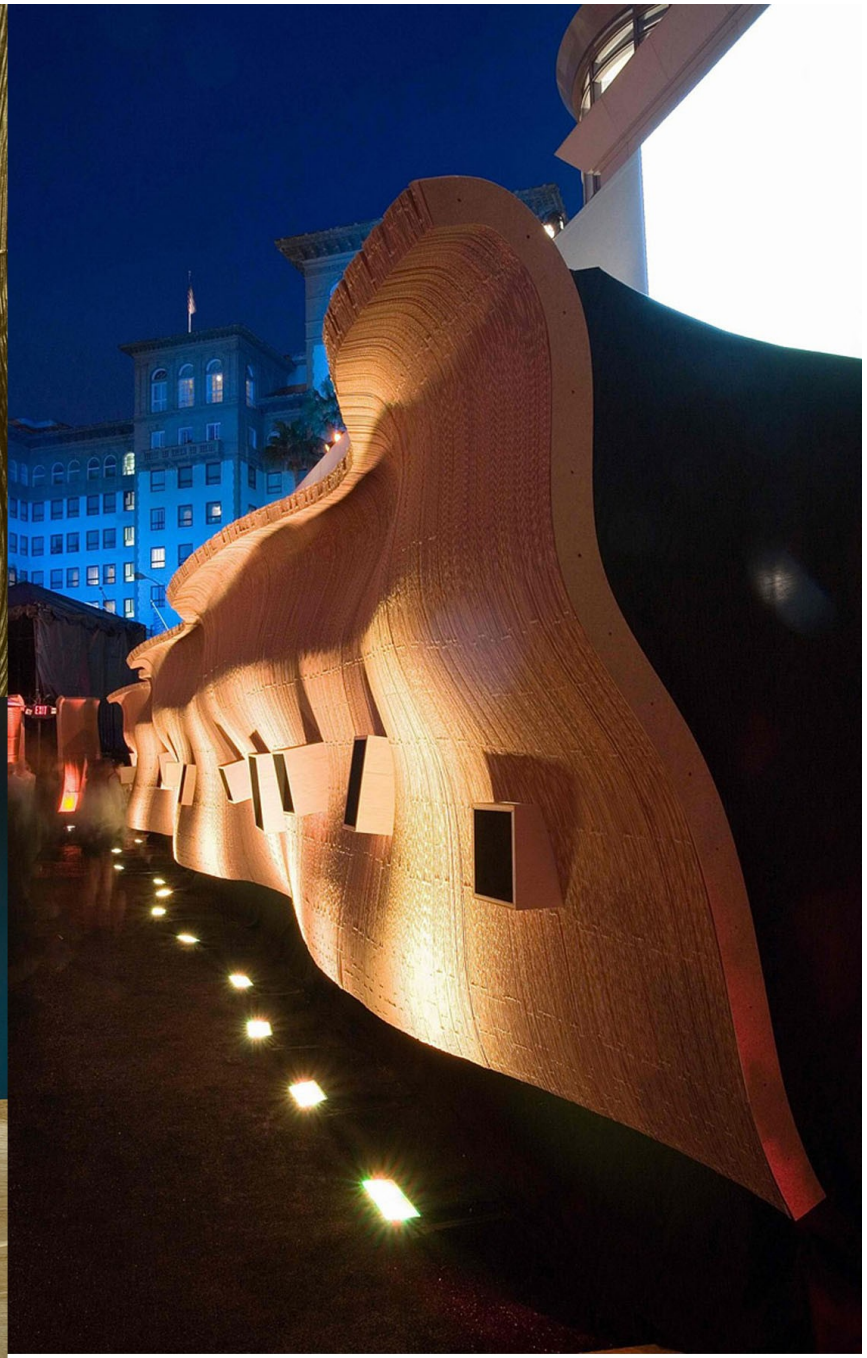


Abbildung 3.7: Rip-Curl Canyon (Bilder links) und Installation für Tiffany & Company Gehry Jewelery Launch Party (Bild oben) von Ball-Nogues Studio (maschinell geschnittene Wellpappe)

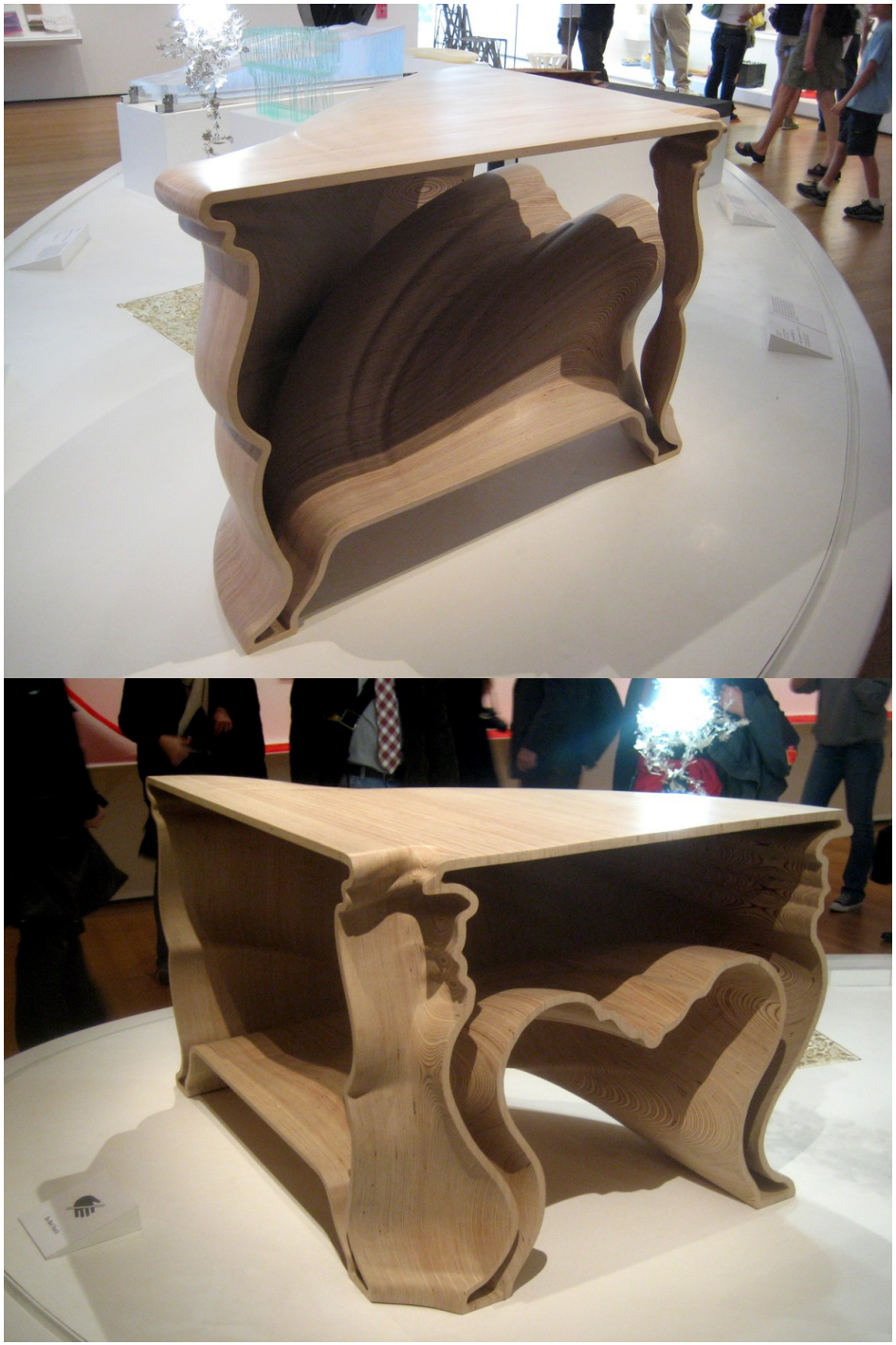


Abbildung 3.8: Cinderella Table von Jeroen Verhoeven (geschichtete Querschnitte aus Holzfurnier, händisch nachbearbeitet)

Modelle kann schon während der Planungsphase beurteilt werden, wie sich ein Bauwerk in die umgebende Landschaft einfügen wird.

Informationen zum Verlauf des umliegenden Geländes eines Bauplatzes liegen meistens in Form von Höhenschichtlinien vor, die im einfachsten Fall aus vorhandenem Kartenmaterial extrahiert werden (Abbildung 3.3). Aus diesem Grund werden Landschaftsmodelle traditionellerweise auch als Höhenschichtmodelle ausgeführt. Zum Bau eines solchen Höhenschichtmodells bieten sich zwei Varianten an: zum einen die naheliegende „massive“ Variante, bei der jede Höhenschichtlinie auf je eine Modellbauplatte aufgetragen und ausgeschnitten wird. Die ausgeschnittenen Schichten werden einfach übereinander gestapelt und verklebt. Für das Modell in Abbildung 3.1 werden dazu allerdings mindestens 50 Modellbauplatten benötigt, auch wenn kleinere Teile aus Verschnittstücken geschnitten werden.

Sofern sich die Höhenschichtlinien nicht überschneiden (was zum Beispiel bei großen überhängenden Felsvorsprüngen oder höhlenartigen Einschlüssen im Gelände passiert), kann man aber sehr viel material-effizienter vorgehen, indem mehrere Höhenschichtlinien pro Platte geschnitten werden. Für das Modell in Abbildung 3.5 werden insgesamt nur fünf Modellbauplatten benötigt. In diesem Beispiel wird dazu jede fünfte Höhenschichtlinie auf jeweils eine Platte aufgetragen, beginnend bei Schichtlinie 1 auf der ersten Platte, Schichtlinie 2 auf der zweiten, usw. Die sich dadurch ergebenden Teile werden wie in Abbildung 3.4 gemäß ihrer Reihenfolge übereinander gestapelt und befestigt.

Die fünf Modellbauplatten werden so ohne Verschnitt verarbeitet. Die Materialersparnis ist im Vergleich zum Modell in Abbildung 3.1 enorm, während der Arbeitsaufwand gleich bleibt, da bei beiden Varianten jede Höhenschichtlinie genau einmal geschnitten werden muss.

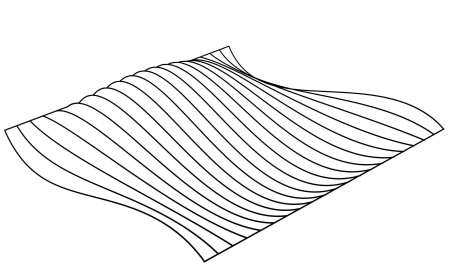
Selbstverständlich lassen sich auf diese Weise nicht nur Landschaftsmodelle herstellen. Jede beliebige dreidimensionale Form kann durch „Höhenschichtlinien“ beschrieben und dann durch Schichtung zweidimensionaler Querschnitte produziert werden. Es gibt eine Reihe von Projekten, die sich dieser einfachen Möglichkeit zur Fabrikation komplexer Formen bedienen (Abbildung 3.7 und 3.8). Allerdings ist den meisten dieser Projekte gemeinsam, dass sie keinerlei Bezug zu den Bedingungen ihrer Materialisierung entwickeln. Die Produktionsmethode ist dann reines Mittel zum Zweck, manchmal einfach nur ein materialineffizienter Ersatz für kostspieligere dreidimensionale Produktionsverfahren.

3.2 Analyse

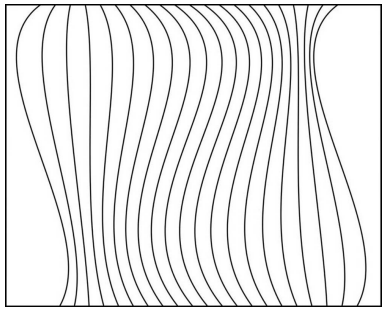
Angesichts der Bandbreite des Formenrepertoires, das uns durch den Einsatz von Computern beim Entwurf zur Verfügung steht, ist es nur konsequent, dass „formneutrale“ Produktionsmethoden, wie die Stereolithographie, Lasersintern oder der 3d-Druck, heute hoch im Kurs stehen. Bei der Fertigung mit einem 3d-Drucker macht es keinen großen Unterschied, ob man einen simplen Kubus oder ein Objekt mit einer hochkomplexen Geometrie produziert. Er wird beides mit der gleichen Geduld erledigen. Diese formale Freiheit ist aber teuer erkauft, besonders wenn man in einem größeren als einem Modellbaumaßstab produzieren möchte. Im Gegensatz dazu ist die hier präsentierte Methode ein spezifischer Prozess, dem eine eigene Formensprache „eingeschrieben“ ist. Manche Formen lassen sich damit sehr viel effizienter und vor allem materialsparsamer herstellen, als andere.

Der Schlüssel zur prozessgerechten Anwendung dieser Fertigungsmethode liegt in der Organisation der Schichtlinien in den Schnittmustern. Die Schichten für eine gekrümmte Fläche wie in Abbildung 3.9 lassen sich sehr leicht fertigen, indem die Profilkurven der Fläche im

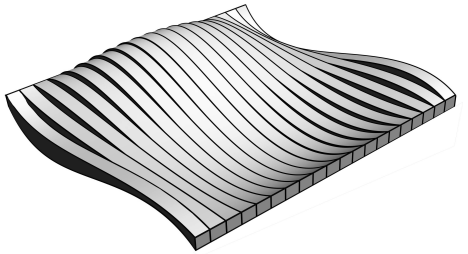
Abbildung 3.9: Herstellung einer gekrümmten Fläche durch Schichtung zweidimensionaler Elemente



Originale Fläche

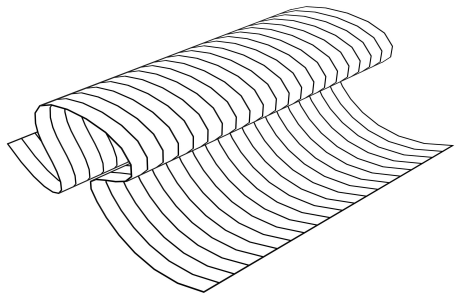


Schnittmuster mit den formdefinierenden Kuven der originalen Fläche auf einer Platte

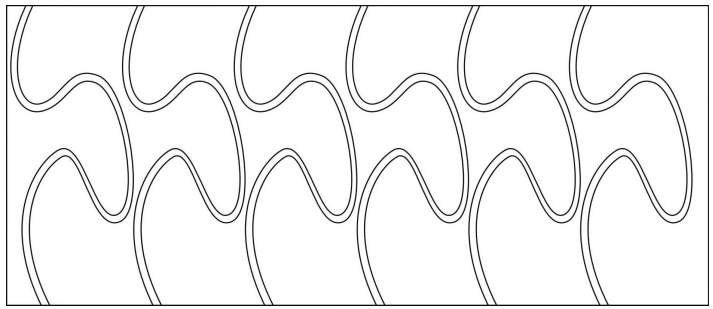


Die zweidimensionalen Elemete aus dem Schnittmuster aneinander gefügt

Abbildung 3.10: Eine ungünstige Form für die Herstellung mit der vorgestellten Methode



Originale Fläche



Schnittmuster

Abbildung 3.11: Pyramide, aus zwei Platten hergestellt

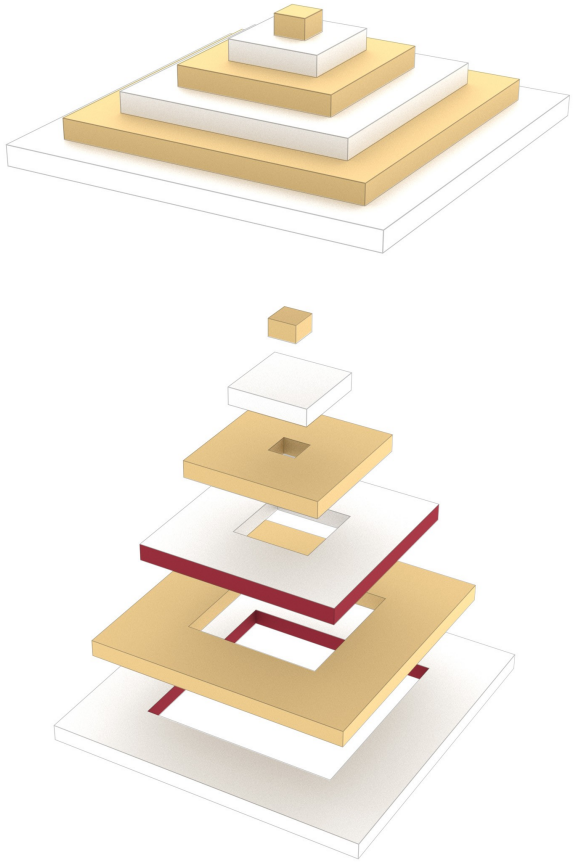
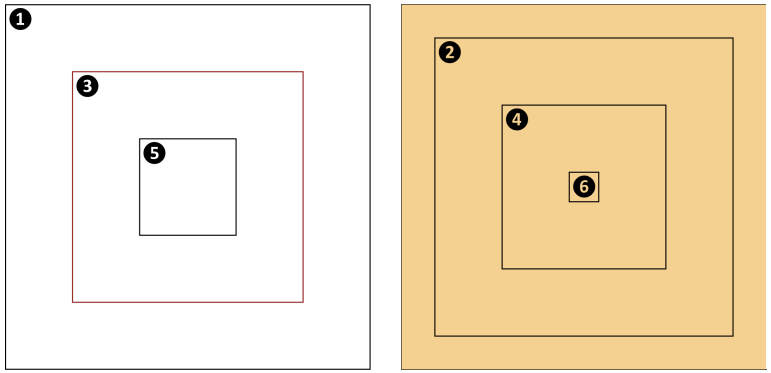


Abbildung 3.12: Schnittmuster der Pyramide auf zwei Platten



Schnittmuster ganz einfach nebeneinander angeordnet werden. Die 25 einzelnen Schichten werden dann mit nur 26 Schnittlinien erzeugt. Die Unterseite des Objektes bildet in diesem Fall die originale Fläche als Negativform ab.

Sehr viel schwieriger wird es bei der Fläche in Abbildung 3.10. Die einzelnen Schnittlinien können hier durch die überhängende Auskragung der Form nicht so platzsparsam parallel angeordnet werden. Die Folge ist, dass erstens jedes einzelne Teil zwei Schnittkanten benötigt, und zweitens, dass der Materialverbrauch vervielfacht wird und die Methode in diesem Fall gleich viel Verschnitt produziert, als würde man die Form mit einer 3d-Fräse fertigen.

Das selbe Konzept lässt sich auch auf Objektformen mit geschlossenen „Höhenschichtlinien“ umlegen. Abbildung 3.11 zeigt den sehr einfachen Fall einer Pyramide, die aus zwei Platten gefertigt wird. Die Schichtlinien werden dabei immer abwechselnd auf die beiden Platten verteilt (Abbildung 3.12). Der Trick dabei ist, dass die Schnittkanten auf der ersten Platte somit versetzt zu den Schnittlinien auf der zweiten Platte angeordnet sind. Durch diesen Versatz überlappen sich die einzelnen Lagen der Pyramide und können aufeinander gestapelt werden.

Die dritte „Höhenschichtlinie“ der Pyramide ist in der Abbildung rot markiert. Sie bildet gleichzeitig die Innenkante der ersten und die Außenkante der dritten Lage. Wenn die Schichtlinien geschnitten werden, erzeugen sie eine Aussparung in der benachbarten, weiter außen liegenden Schicht auf der selben Platte. Die dritte Lage der Pyramide besteht also eigentlich aus der Aussparung der ersten, die fünfte wiederum aus der Aussparung der dritten Schicht. Müssten die einzelnen Lagen nebeneinander angeordnet werden, wäre der Materialverbrauch um ein vielfaches größer.

Aus diesem System ergeben sich mehrere Konsequenzen: Erstens entsteht durch die ineinander liegende Anordnung

der Schnittlinien automatisch ein Hohlkörper - die Hülle der vorgegebenen Objektgeometrie. Die Innenseite der Hülle bildet dabei die Außenseite als Negativform ab. Zweitens ist es durch diese Anordnung auch möglich, geschlossene Flächen mit einer einzelnen Randkontur (z.B. eine Halbkugel) oder mit zwei Rändern („schlauchförmige" Objekte) materialeffizient zu fertigen.

Für die physische Qualität der Objekthülle sind zwei Parameter entscheidend: die Stärke der Hülle (also die Breite der einzelnen Schichten) und die Überlappung übereinanderliegender Schichten. Beide Parameter bestimmen die Belastbarkeit der Hülle und entscheiden letztendlich auch, ob das Objekt überhaupt herstellbar ist.

Eine ausreichende Überlappung ist natürlich wichtig, wenn die Schichten durch Kleben aneinander befestigt werden, aber auch, wenn später Aussparungen für Montagehilfen, wie etwa Stahlstifte zur Fixierung der Lage der einzelnen Schichten, in den Querschnitten untergebracht werden sollen.

Abbildung 3.13 zeigt den Einfluss der Schicht- und Plattenanzahl auf die Stärke der Hülle und die Überlappung der einzelnen Schichten. Der Anteil der überlappenden Fläche hängt alleine von der Anzahl der verwendeten Platten ab. Bei der Pyramide aus zwei Platten überlappen sich die Lagen genau bis zur Mitte der jeweiligen Schichten. Wird die selbe Pyramide aus drei Platten gefertigt, überlappen sich die Lagen zu zwei Drittel, bei vier Platten zu drei Viertel ihrer Fläche.

Die Stärke der Hülle hängt von der Anzahl der Platten und der Höhenschichten ab. Je mehr Platten verwendet werden, umso weniger Schichtlinien müssen pro Platte geschnitten werden. Das bedeutet, dass der Abstand zwischen der inneren und der äußeren Schnittkante jeder Schicht größer wird und die Lagen somit breiter werden. Die Gesamtzahl der Schichten ergibt sich aus der Objekthöhe (als Höhe -

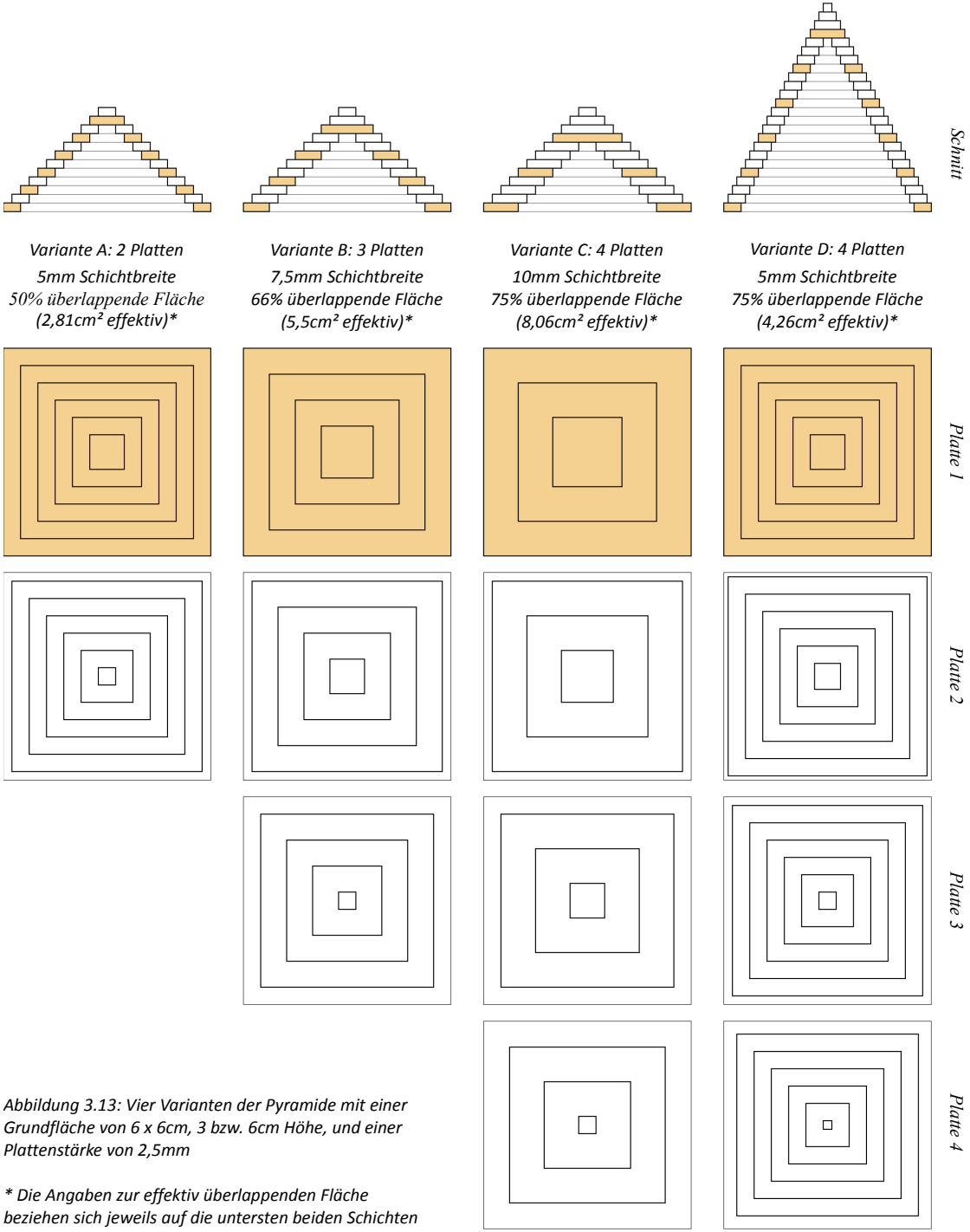


Abbildung 3.13: Vier Varianten der Pyramide mit einer Grundfläche von 6 x 6cm, 3 bzw. 6cm Höhe, und einer Plattenstärke von 2,5mm

** Die Angaben zur effektiv überlappenden Fläche beziehen sich jeweils auf die untersten beiden Schichten*

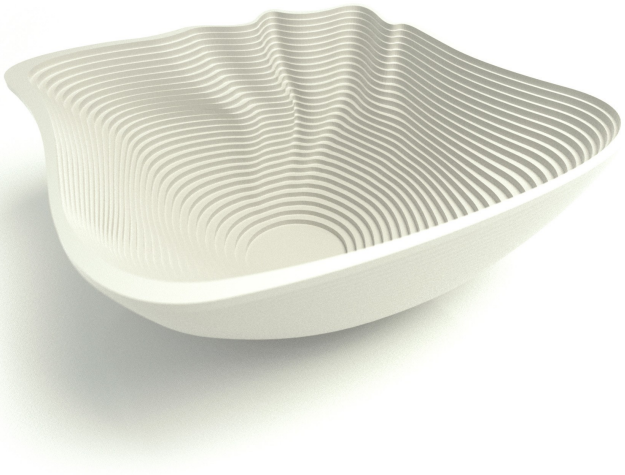


Abbildung 3.14: Entwurf einer Obstschale

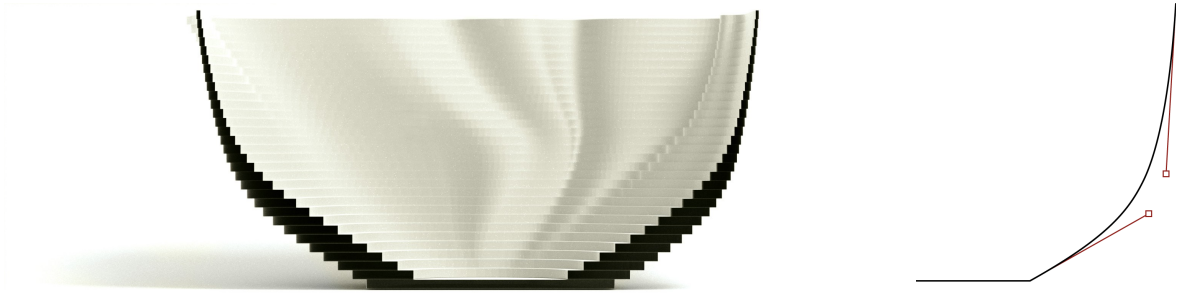


Abbildung 3.15: Schnitt durch die Schale

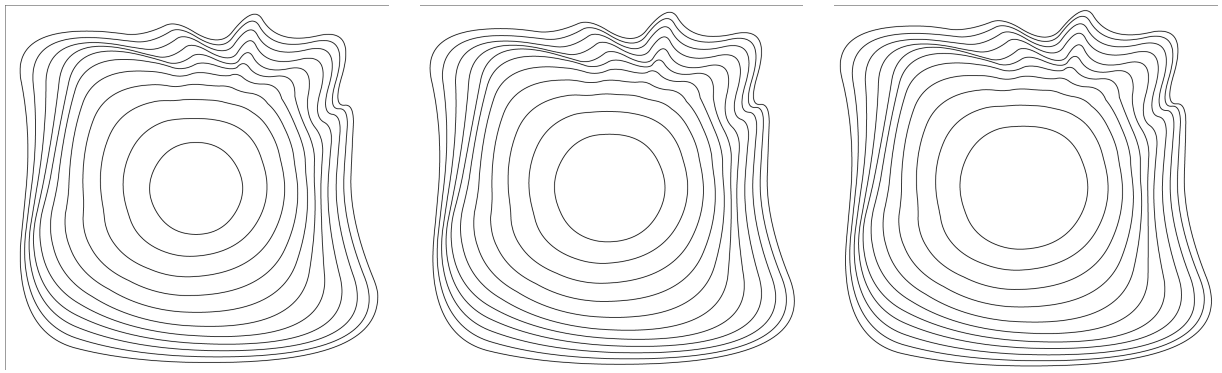


Abbildung 3.16: Schnittmuster

oder z-Achse – verstehe ich die Achse normal zur Ebene der Schichtlinien) geteilt durch die Stärke der Platten. Doppelte Objekthöhe bedeutet ganz einfach, dass doppelt so viele Höhenlagen geschnitten werden müssen. Die resultierenden Schichten werden entsprechend nur halb so breit (Varianten C und D).

Der Anteil der überlappenden Flächen steigt also mit der Anzahl der Platten. Die Stärke der Hülle steigt *generell* mit der Anzahl der Platten und verringert sich, je mehr Lagen geschnitten werden müssen. Der wichtigste Einflussfaktor auf die Breite der *einzelnen* Schichten ist aber die Höhenentwicklung des Objektes.

Die Höhenentwicklung einer Pyramide ist linear. Das bedeutet, dass die Entfernung der Schichtlinien zu ihren Nachbarn im Schnittmuster über den Höhenverlauf gleich bleibt. Alle regulären Schichten (also alle Schichten außer der innersten auf jeder Platte) haben folglich die gleiche Breite und den selben Anteil an überlappender Fläche. Wenn die Hülle des Objektes aber gekrümmt ist, verhalten sich die Schichten über den Höhenverlauf nichtlinear.

Abbildung 3.14 zeigt den Entwurf einer Obstschale, der diesen Effekt deutlich macht. Die Höhenentwicklung der Schale beschreibt im Schnitt (Abbildung 3.15) eine Kurve, die an ihrem Boden relativ flach beginnt und zum Rand hinauf immer steiler wird. Je steiler diese Kurve wird, umso näher liegen die „Höhenschichtlinien“ der Schale in den Schnittmustern beisammen, was in Abbildung 3.16 zu beobachten ist. Die einzelnen Schichten werden folglich immer schmäler, je steiler die Höhenentwicklung wird, in diesem Beispiel also je näher sie am Rand liegen.

Schmale Schichten verringern die Belastbarkeit der Hülle und können beim Zusammenbau der einzelnen Elemente zu Problemen führen, wenn die überlappende Fläche übereinander liegender Schichten zu klein wird, um sie aneinander befestigen zu können. Unter Umständen reicht

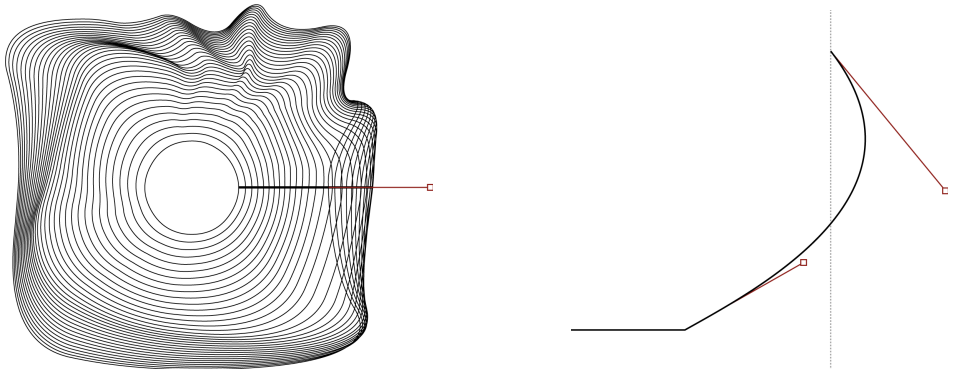
eine einzige zu schmale Schicht, um das gesamte Objekt unbrauchbar zu machen. Die Kontrolle der Schichtbreite über den Höhenverlauf ist bei dieser Produktionsmethode ebenso wichtig, wie sicherzustellen, dass sich die Schichtlinien nicht gegenseitig überschneiden.

Die Anfangs- und Endtangenten der Höhenverlaufskurve bieten eine gute Möglichkeit, die Höhenentwicklung des Objektes, und damit den Verlauf der Breite der einzelnen Schichten, einzuschätzen und zu kontrollieren. Diese Tangenten sind in Abbildung 3.15 rot eingezeichnet. Die Kurve beginnt am Boden der Schüssel in einem Winkel von etwa 30 Grad zur Horizontalen und endet am Rand der Schale schon nahezu senkrecht. Wäre die Endtangente noch steiler, würden die Schichten am Rand noch schmäler werden. Wenn sie, wie in Abbildung 3.17, sogar von der Schüssel weg zeigt, kann alleine aus der Lage der Tangente geschlossen werden, dass sich einige Höhenschichten überschneiden werden, da die Hülle in diesem Fall am Rand eine überhängende Form annimmt. Würde man Schichtlinien wie in Abbildung 3.17 dann tatsächlich aus einer Platte schneiden, hätte man ein Puzzle an Teilen vor sich, denn die Schnittlinien der überhängenden Elemente würden beim Schneiden die weiter innen liegenden Schichten zerstückeln.

3.3 Geometriemodell

Voraussetzung für die Darstellung und Bearbeitung von Objekten mit einem Computerprogramm ist die algorithmische Beschreibung ihrer Geometrie. Das Geometriemodell bildet hier den Vermittler zwischen den gestalterischen Vorstellungen des Entwerfers und der algorithmischen Weiterverarbeitung von Entwürfen, wie in der Berechnung der „Höhenschichtlinien“ und der automatischen Generierung der Schnittmuster für die Fabrikation der Objekte.

Abbildung 3.17: Änderung der Höhenverlaufskurve und die sich daraus ergebenden Schnittlinien



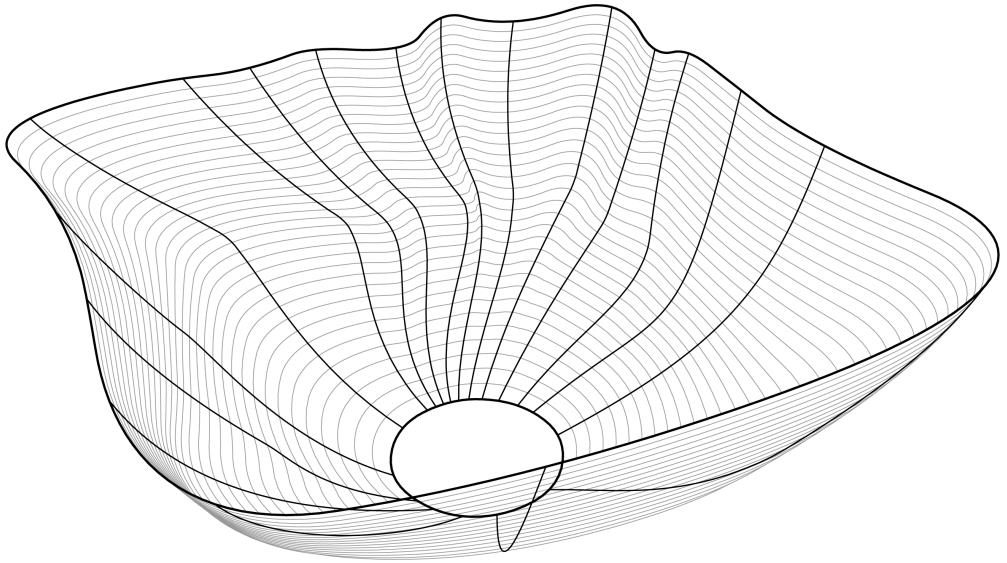


Abbildung 3.18: Querschnitte und Höhenverlaufskurven der Obstschale

Das erste Element des parametrischen Geometriemodells für dieses Projekt wurde eben schon vorgestellt: die Kurven des Höhenverlaufes. Der zweite Teil der Geometriebeschreibung sind die horizontalen Querschnitte der Ränder der Form. In Abbildung 3.18 sind die Querschnitte und Höhenverlaufskurven der Obstschale dargestellt. Wenn man sich die Schale ohne Boden vorstellt, beschreibt ihre Geometrie ein gekrümmtes Flächenband mit zwei geschlossenen Kurven als obere und untere Begrenzung. Der Rand der Schale bildet den oberen, äußeren Querschnitt der Form, ihr Boden den unteren, inneren Querschnitt. Alle Formen mit einer schlauchartigen Topologie lassen sich durch ihre zwei Randbedingungen und die Kurven ihres Höhenverlaufes definieren.

Die Grundlage des gesamten Geometriemodells sind Bézierkurven [SC-01]. Bézierkurven sind parametrisch modellierte Kurven, die in den 60er Jahren für die ersten CAD Anwendungen entwickelt wurden. Viele der großen Auto- und Flugzeughersteller unterhielten in dieser Zeit eigene Forschungsabteilungen, in denen zum Teil völlig isoliert von einander die Grundlagen für die mathematische Beschreibung von Freiformflächen erarbeitet wurden. So entwickelte etwa James Ferguson, der für Boeing arbeitete, die Heremite-Kurven, Pierre Bézier bei Renault und Paul de Casteljau bei Citroën unabhängig von einander die Bézierkurve, und Carl de Boor bei General Motors schließlich die B-Spline Kurve[2] - die Grundlage der Non Rational Uniform B-Splines (NURBS), also jener Freiformkurven und -flächen, die bis heute die zentrale Rolle bei der Modellierung von gekrümmten Formen mit Hilfe des Computers spielen.

Alle drei Ansätze basieren auf dem gleichen mathematischen Konzept: der Beschreibung von Kurven mit Hilfe von Polynomfunktionen. Das wesentliche Merkmal der Bézierkurven ist dabei die Vorhersehbarkeit ihrer Verformung beim Verschieben von Kontrollpunkten. Mit etwas Erfahrung lassen sich gewünschte Kurvenverläufe gezielt, ohne langes herumprobieren, herstellen. Das ist mit ein Grund warum Bézierkurven zum Standard in praktisch jedem Vektorgrafik- und CAD-Programm geworden sind.

„The breakthrough insight was to use control polygons (courbes à pôles), a technique that was never used before. Instead of defining a curve (or surface) through points on it, a control polygon utilizes points near it. Instead of changing the curve (surface) directly, one changes the control polygon, and the curve (surface) follows in a very intuitive way."[3]

Bézierkurven werden heute auf verschiedenste Weise in Computeranwendungen eingesetzt, nicht nur in CAD- und Grafikprogrammen, sondern zum Beispiel auch für Bewegungsanimationen in Trickfilmen, sanfte Überblendungen bei User-Interface Effekten, etc. Sogar die Buchstaben auf dieser Seite sind tatsächlich nichts anderes, als vom Drucker interpretierte Bézierkurven.

Um das Geometriemodell zu verstehen, ist ein wenig Hintergrundwissen über diesen Kurventyp notwendig. Bézierkurven bestehen aus einem Start- und einem Endpunkt, sowie einer variablen Anzahl von weiteren Kontrollpunkten, die den Verlauf der Kurve bestimmen, sie aber nicht berühren. Wird ein Kontrollpunkt bewegt, wird die Kurve im Bereich des entsprechenden Punktes in seine Richtung gelenkt. Die Kontrollpunkte können als eine Art Gewicht verstanden werden, das die Kurve verbiegt.

Die Abbildungen 3.19 bis 3.21 zeigen Bézierkurven 2., 3. und 10. Grades. Der Grad einer Kurve wird durch die Anzahl ihrer Kontrollpunkte bestimmt, genauer gesagt, er entspricht der Anzahl der Kontrollpunkte verringert um 1 (Start- und Endpunkte werden auch als Kontrollpunkte angesehen).

Eine Bézierkurve n-ten Grades für die gegebenen Kontrollpunkte P_i ist definiert[4] als

$$C(t)=\sum_{i=0}^{n} B_{i,n}(t)\,P_i \qquad (1)$$

Jeder Punkt einer Bézierkurve ist ein gewichtetes Mittel aus den gegebenen Kontrollpunkten. $B_{i,n}$ ist dabei die Gewichtsfunktion der einzelnen Kontrollpunkte.

$$B_{i,n}(t)=\binom{n}{i} t^i (1-t)^{n-i} \qquad (2)$$

Die konkrete Formel für eine quadratische Bézierkurve ergibt sich aus diesen beiden Gleichungen als

$$\begin{aligned} C(t) &= \sum_{i=0}^{2} \binom{2}{i} t^i (1-t)^{2-i} P_i \\ &= (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \end{aligned} \qquad (3)$$

P_0 ist dabei der Startpunkt der Kurve, P_1 der Kontrollpunkt, der den Verlauf der Kurve bestimmt, und P_2 ist ihr Endpunkt. Die Bedeutung des Parameters t lässt sich am besten Anhand der geometrischen Konstruktion eines beliebigen Kurvenpunktes nachvollziehen.

Abbildung 3.19 zeigt die Konstruktion des Kurvenpunktes für t=0.33 mithilfe des Algorithmus von de Casteljau. Dazu wird die Strecke von P_0 nach P_1 im Verhältnis t zu 1-t geteilt, hier also im Verhältnis 0.33 zu 0.67. Daraus ergibt sich der Hilfspunkt H_0. Die Strecke von P_1 nach P_2 wird ebenfalls im selben Verhältnis geteilt, was zum Punkt H_1 führt. Die Gerade durch H_0 und H_1 bildet die Tangente der Kurve am

2 Claude Brezinski, Luc Wuytack, *Numerical analysis in the twentieth century*, in: Claude Brezinski, Luc Wuytack (Hrsg.), *Numerical analysis: historical developments in the 20th century* (Elsevier Science B. V., Amsterdam, 2001), S. 20f

3 Gerald E. Farin, *A History of Curves and Surfaces in CAGD*, in: Gerald E. Farin, Josef Hoschek, Myung-Soo Kim (Hrsg.), *Handbook of computer aided geometric design* (Elsevier Science B.V., Amsterdam, 2002), S. 5

4 Hartmut Prautzsch, Wolfgang Boehm,Marco Paluszny, *Bézier and B-spline techniques* (Springer Verlag, Berlin, 2002)

gesuchten Punkt. Wenn man nun die Strecke von H_0 nach H_1 wiederum im Verhältnis 0.33 zu 0.67 teilt, erhält man schlussendlich den Kurvenpunkt für *t*=0.33.

Der Kurvenpunkt für *t*=0.32 würde folglich ein kleines Stückchen weiter links liegen, der Punkt für *t*=0.34 natürlich weiter rechts. Man könnte sagen, der Parameter *t* legt quasi die Stelle auf der Kurve fest, an dem ein Punkt berechnet werden soll. Die „Stelle" *t*=0 fällt dabei genau auf den Startpunkt der Kurve, was auch rechnerisch leicht nachzuvollziehen ist, wenn man in Gleichung (3) *t* gleich 0 setzt.

$$\begin{aligned} C(0) &= (1-0)^2 \cdot P_0 + 2 \cdot 0 \cdot (1-0) \cdot P_1 + 0 \cdot P_2 \\ &= 1 \cdot P_0 + 0 \cdot P_1 + 0 \cdot P_2 \end{aligned}$$

Das Gewicht von P_0 ist in diesem Fall 1, also 100%, das Gewicht der anderen Kontrollpunkte wird zu 0. Entsprechend erhält man den Endpunkt der Kurve, wenn t gleich 1 gesetzt wird:

$$\begin{aligned} C(1) &= (1-1)^2 \cdot P_0 + 2 \cdot 1 \cdot (1-1) \cdot P_1 + 1 \cdot P_2 \\ &= 0 \cdot P_0 + 0 \cdot P_1 + 1 \cdot P_2 \end{aligned}$$

Die Funktion erzeugt also Kurvenpunkte zwischen dem Anfangs- und dem Endpunkt, wenn der Parameter *t* größer als 0 und kleiner als 1 ist. Wenn man nun z.B. für die Darstellung der Kurve am Bildschirm neun Punkte der Kurve ermitteln möchte (auch neueste Grafikkarten können nur Linien zeichnen, Kurven müssen zur Darstellung in kurze Liniensegmente umgewandelt werden), wird Gleichung (3) für *t*=0.1, *t*=0.2, … bis *t*=0.9 berechnet [SC-01a].

Die Gerade durch H_1 und H_2 in Abbildung 3.19 bildet wie gesagt die Tangente der Kurve am Punkt C(0.33). Bei der Punktkonstruktion für *t*=0 würde der erste Hilfspunkt genau auf P_0 und der zweite genau auf P_1 fallen. Die Gerade durch P_0 und P_1 bildet also die Tangente der Kurve

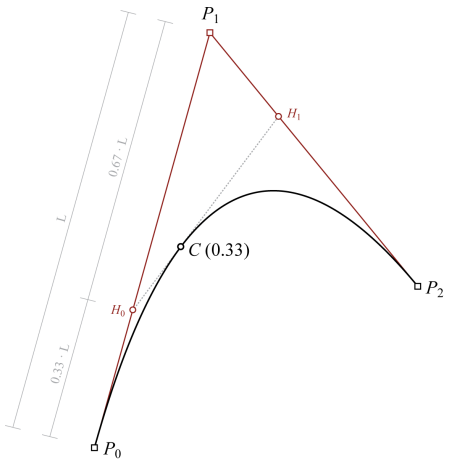


Abbildung 3.19: Quadratische Bézierkurve (2. Grades)

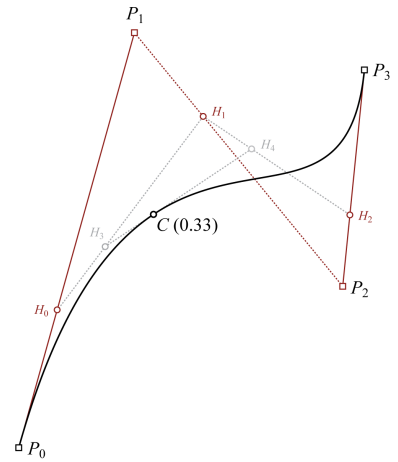


Abbildung 3.20: Kubische Bézierkurve (3. Grades)

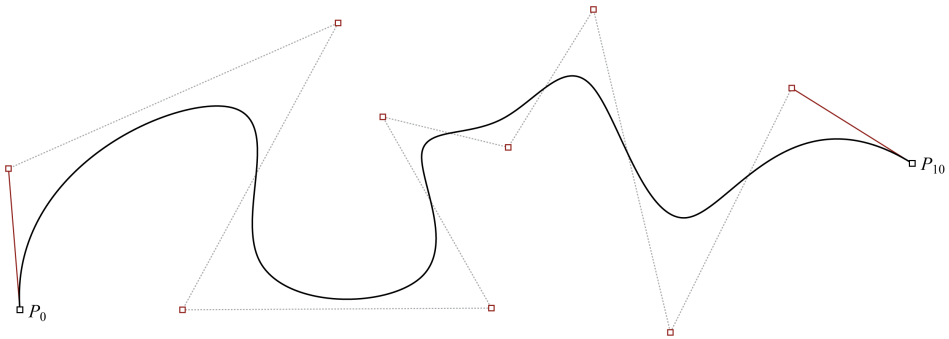


Abbildung 3.21: Bézierkurve 10. Grades

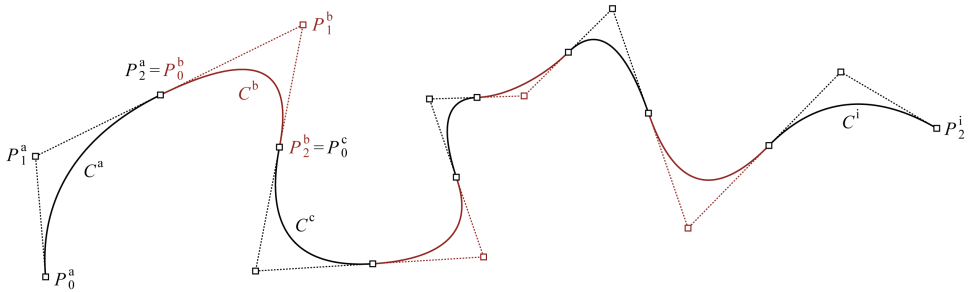


Abbildung 3.22: Annäherung der Kurve 10. Grades durch zusammengesetzte quadratische Bézierkurven

am Punkt P_0. Die Anfangstangente einer Bézierkurve ist immer durch ihre ersten beiden Kontrollpunkte definiert, die Endtangente entsprechend durch die letzten beiden. Quadratische Bézierkurven haben außer ihrem Anfangs- und Endpunkt aber nur noch einen weiteren Kontrollpunkt, der den Verlauf der Kurve bestimmt. Die Richtung beider Tangenten hängt hier also von dem einen Kontrollpunkt P_1 ab. Wenn jedoch die Anfangs- und Endbedingungen der Kurve, wie bei der Höhenverlaufskurve in den Abbildungen 3.15 und 3.17, getrennt voneinander kontrolliert werden sollen, ist zumindest eine Kurve dritten Grades notwendig.

Eine kubische Bézierkurve ergibt sich aus (1) und (2) als

$$C(t)=(1-t)^3 P_0+3t(1-t)^2 P_1 \\ +3t^2(1-t)P_2+t^3 P_3 \qquad (4)$$

Die Punktkonstruktion nach de Casteljau (Abbildung 3.20) funktioniert ähnlich wie bei der quadratischen Bézierkurve. Die Strecken von P_0 nach P_1, von P_1 nach P_2 und von P_2 nach P_3 werden wieder im Verhältnis t zu 1-t geteilt. Ebenso die Strecken von H_0 nach H_1 und von H_1 nach H_2. Die Teilung der sich daraus ergebende Strecke von H_3 nach H_4 im selben Verhältnis führt schlussendlich zum gesuchten Punkt der Kurve.

Die Linie vom Startpunkt zum zweiten Kontrollpunkt definiert die Anfangstangente der kubischen Bézierkurve, die Linie vom dritten Kontrollpunkt zum Endpunkt ihre Endtangente. Damit sind alle Voraussetzungen für die Beschreibung der Höhenverlaufskurve gegeben. Wie man sieht, steigt allerdings die Komplexität der Funktionen und damit die Zeit, die entsprechende Rechenoperationen erfordern, mit dem Grad der Kurven an [SC-2]. Zumindest die Berechnung von Kurvenpunkten kann aber mit einem kleinen Trick um einiges beschleunigt werden:

Sobald der Grad der Kurve bekannt ist, ist die einzige verbleibende Unbekannte der Gewichtsfunktion (2) der Parameter t. Wenn für die Darstellung von Kurven immer eine gleichbleibende Anzahl von Kurvenpunkten mit fixen t-Werten berechnet werden soll, können die Gewichte der einzelnen Kontrollpunkte z.B. für t=0.1, t=0.2, usw., vorkalkuliert und die Ergebnisse gespeichert werden [SC-01b]. Diese Ergebnisse können dann zur Berechnung von Punkten für jede Kurve gleichen Grades wiederverwendet werden, was die Kalkulationen wesentlich beschleunigt.

Wenn die Gewichtsfunktionen durch die vorkalkulierten Werte ersetzt werden, wird die Berechnung von Punkten der kubischen Kurve auf vier simple Additionen und Multiplikationen reduziert [SC-01c].

$$C(t)= B_{0,3}(t)\cdot P_0+B_{1,3}(t)\cdot P_1 \\ +B_{2,3}(t)\cdot P_2+B_{3,3}(t)\cdot P_3 \qquad (5)$$

Das gleiche Verfahren lässt sich auf Kurven beliebigen Grades anwenden, hilft aber leider nur bei der Berechnung von Kurvenpunkten und nicht bei komplexeren Operationen, wie sie im Folgenden noch vorgestellt werden.

Für die Beschreibung der Querschnittskurven, die in den meisten Fällen mehr als nur vier Kontrollpunkte benötigen werden, bieten sich zwei Möglichkeiten an. Zum einen können sie durch jeweils eine einzelne Kurve höheren Grades beschrieben werden, zum anderen durch zusammengesetzte Kurven niederen Grades, die in der Computerprogrammierung oft als Splines bezeichnet werden.

Splines sind komplexe Kurven, die aus einer Kette von mehreren miteinander verbundenen Teilkurven zusammengesetzt sind. Der Grad der Teilkurven ist dabei beliebig und kann auch innerhalb eines Splines variieren. Abbildung 3.22 zeigt einen Spline bestehend aus neun Teilkurven zweiten Grades, der annähernd die selbe Form wie die Kurve zehnten Grades in Abbildung 3.21 beschreibt. Wie man sieht, fällt der Endpunkt jeder Teilkurve des Splines mit dem Startpunkt der jeweils folgenden zusammen. Damit die einzelnen Kurven fließend ineinander übergehen, damit also kein Knick an ihren Endpunkten entsteht, müssen sich aufeinander folgende Kurven an ihrem Verbindungspunkt eine Tangente teilen. Das bedeutet, dass der vorletzte Kontrollpunkt, der gemeinsame Verbindungspunkt und der zweite Kontrollpunkt der folgenden Kurve auf einer Linie liegen müssen. Ist diese Bedingung erfüllt, spricht man von einer geometrisch stetigen Kurve.

Splines können über ihren gesamten Verlauf stetig sein, sie können aber auch an jedem Verbindungspunkt der Teilkurven eine Ecke ausbilden, was z.B. notwendig wäre, um die Form der Pyramide darzustellen. Tatsächlich kann auch eine Pyramide mit diesem Geometriemodell beschrieben werden. Wenn alle Kontrollpunkte einer Bézierkurve auf einer Linie liegen, wird aus der Kurve eine Gerade. Es ist also möglich, aus vier geraden Teilkurven die Grundfläche der Pyramide, die den unteren Querschnitt der Form bildet, zu modellieren. Der obere Querschnitt ist in diesem Fall auf einen Punkt reduziert, und die Kurven des Höhenverlaufs folgen den vier Kanten der Pyramide.

Zwei weitere Argumente sprechen für die Verwendung von Splines: erstens wirken die Kontrollpunkte von Bézierkurven global, das heißt das Verschieben eines einzelnen Kontrollpunktes einer Kurve höheren Grades verändert ihren gesamten Kurvenverlauf. Die Bearbeitung von Splines ist sehr viel intuitiver, da einzelne Kurvenabschnitte isoliert verändert werden können. Zweitens kann man sich bei zusammengesetzten Splines die speziellen Eigenschaften von Kurven niederen Grades zu nutze machen, um komplexe Operationen, wie z.B. die Detektierung von sich überschneidenden Kurven, zu vereinfachen.

3.4 Berechnung von Schichtlinien

Abbildung 3.23 zeigt den Entwurf eines Lampenschirms aus transluzentem Acrylglas. Die Querschnitte des Geometriemodells werden bei diesem Objekt von Splines mit ausschließlich quadratischen Teilkurven gebildet. Um die Schichtlinien berechnen zu können, wird jeder Kontrollpunkt auf dem oberen Querschnitt durch eine Höhenverlaufskurve mit einem Kontrollpunkt in entsprechender Lage auf dem unteren Querschnitt verbunden. An sich würde man für die Beschreibung des kreisrunden oberen Querschnitts nicht so viele Teilkurven benötigen, wie für den unteren Rand des Lampenschirms. Damit aber die Schichtlinien zwischen dem oberen und unteren Querschnitt berechnet werden können, müssen beide Kurven die gleiche Anzahl von Kontrollpunkten haben. In diesem Fall werden deshalb auf dem oberen Querschnitt weitere Kontrollpunkte erzeugt, indem die quadratischen Bézierkurven mithilfe des Algorithmus von de Casteljau geteilt werden [SC-01d].

Die Teilung von Kurven funktioniert nach dem selben Prinzip wie die Punktkonstruktion nach de Casteljau und ist anhand der Abbildung 3.25 leicht nachzuvollziehen. Wird eine Kurve geteilt, erhält man als Ergebnis zwei neue Teilkurven mit jeweils drei Kontrollpunkten, die zusammen die Form der originalen Kurve beschreiben. Wenn die Kurve in Abbildung 3.25 am Punkt $C(0.5)$ geteilt werden soll, beginnt die erste neue Teilkurve natürlich bei P_0 und endet im Punkt $C(0.5)$. Der erste und der letzte Kontrollpunkt sind also schon bekannt. Der zweite Kontrollpunkt einer quadratischen Bézierkurve definiert sowohl die Richtung der Anfangs- als auch der Endtangente der Kurve, oder anders gesagt, er liegt am Schnittpunkt beider Tangenten. Die Anfangstangente der ersten Teilkurve bleibt die Linie von P_0 nach P_1. Und die Endtangente durch den Punkt $C(0.5)$ ist mit der Geraden durch H_0 und H_1 ebenfalls schon bekannt. Der Schnittpunkt der Tangenten, und damit der

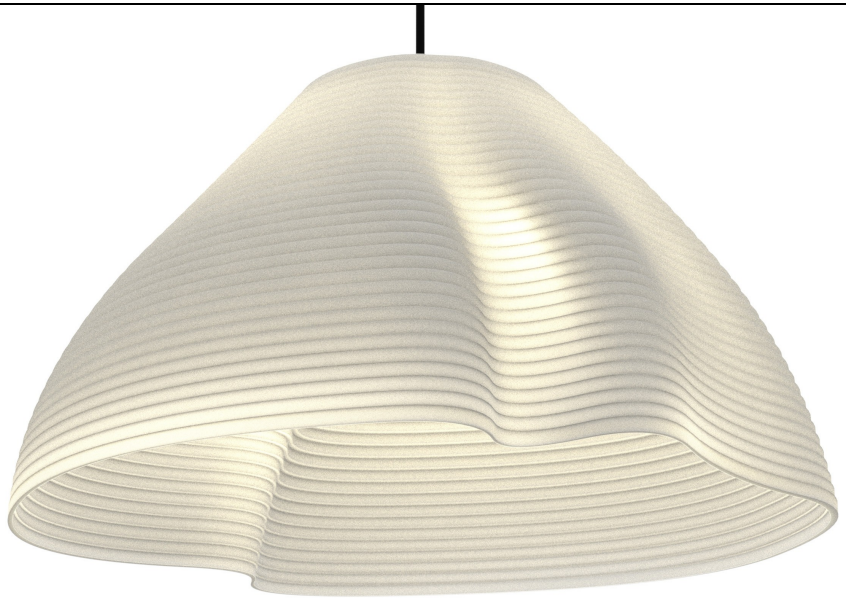


Abbildung 3.23: Lampenschirm aus transluzentem Acrylglas

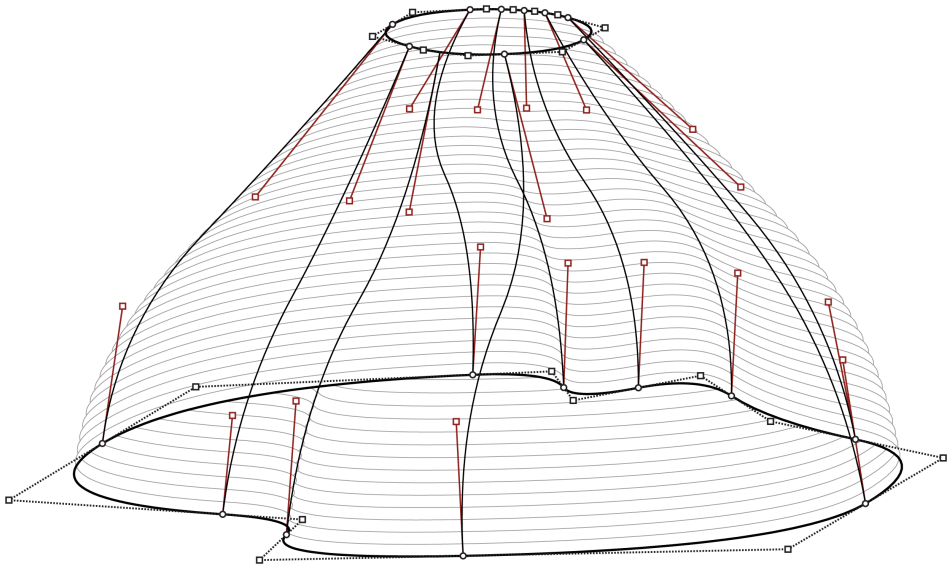


Abbildung 3.24: Geometriemodell des Lampenschirms

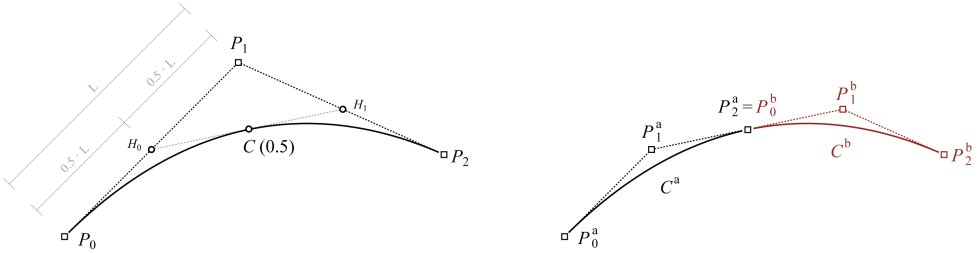


Abbildung 3.25: Teilung einer quadratischen Bézierkurve mithilfe des Algorithmus von de Casteljau

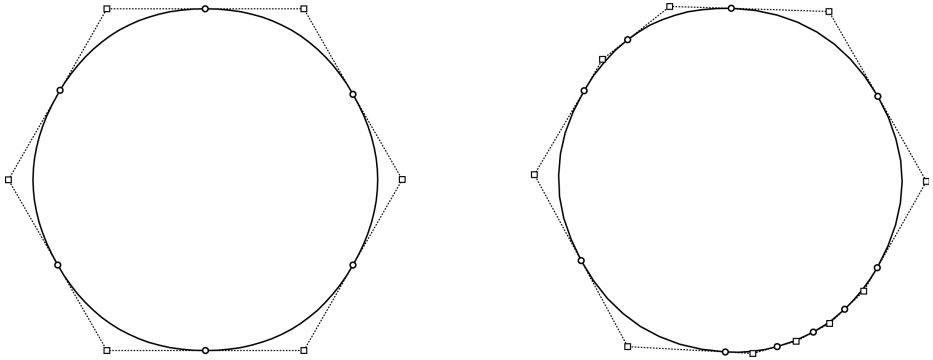


Abbildung 3.26: Oberer Querschnitt im Original (links) und mit ergänzten Kontrollpunkten (rechts)

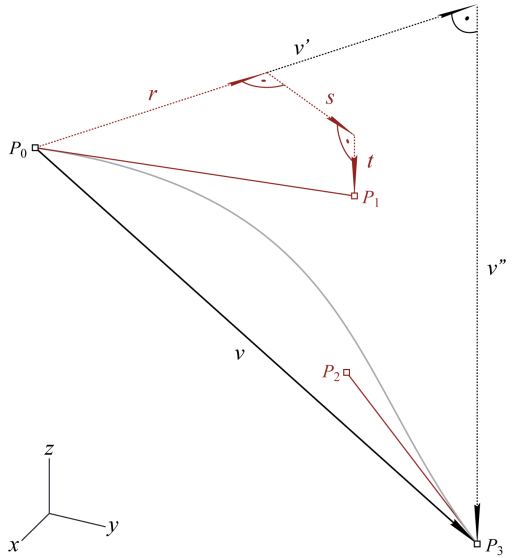


Abbildung 3.27: Beschreibung der Tangentenpunkte der Höhenverlaufskurven

mittlere Kontrollpunkt der ersten Teilkurve, ist also genau der Punkt H_0. Analog dazu sind die Kontrollpunkte der zweiten Teilkurve $C(0.5)$, H_1 und P_2.

Die Ergänzung von Kontrollpunkten ist intern für die Berechnung von Schichtlinien notwendig, der User soll aber nicht unnötig mit zu vielen parametrischen Punkten konfrontiert werden, um die er sich kümmern muss. Deshalb können die neu konstruierten Punkte auf Wunsch einfach ausgeblendet werden. Wenn die originalen Kurven verändert werden, berechnet das Programm die ergänzten Kontrollpunkte automatisch neu, ohne dass der Benutzer eingreifen muss.

Um das Bearbeiten von Formen noch leichter zu machen, ist es auch möglich, mehrere Höhenverlaufskurven durch ein einzelnes Tangentenpaar zu steuern. Zu diesem Zweck werden die Tangentenpunkte der Höhenverlaufskurven nicht durch ihre Raumkoordinaten beschrieben, sondern durch ihre relative Position zu den Kontrollpunkten der Querschnittskurven, die die jeweilige Höhenverlaufskurve verbindet. Diese beiden Kontrollpunkte bilden gleichzeitig den Anfangs- und Endpunkt der zugehörigen Höhenverlaufskurve und sind die Basis für die Positionsbestimmung von Tangentenpunkten. In Abbildung 3.27 ist die Verbindung beider Punkte als Vektor v bezeichnet, v' ist seine Projektion in die xy-Ebene und v'' seine z-Komponente. Um die Position des Tangentpunktes P_1 zu beschreiben, wird die Strecke von P_0 nach P_1 in die drei Vektoren r, s und t zerlegt. Vektor r beginnt bei jenem Endpunkt der Kurve, der in der Reihenfolge der Kontrollpunkte dem Tangentenpunkt am nächsten liegt. Er hat die selbe Richtung wie v'. Der Vektor s verläuft normal zu r in der xy-Ebene und t ist parallel zur z-Achse. Der Tangentenpunkt kann nun durch die drei Parameter R, S und T definiert werden, die jeweils das Längenverhältnis des Vektors v zu den Vektoren r, s bzw. t beschreiben. Diese RST-Werte können vom Tangentenpunktepaar einer Kurve

auf das einer anderen Kurve übertragen werden. Die Tangentenpunkte der anderen Kurve orientieren sich dann an deren Endpunkten, sie werden entsprechend der Position der Endpunkte rotiert und die Kurve wird automatisch gestaucht oder auseinander gezogen, je nachdem ob ihre Endpunkte näher beisammen oder weiter entfernt von einander liegen, als bei der originalen Kurve (Abbildung 3.28). Auf diese Weise können mehrere Kurven mit einem ähnlichen Verlauf erzeugt werden. Darüber hinaus ist es somit auch möglich, alle Höhenverlaufskurven eines Objekts mit nur einem Tangentenpunktepaar zu steuern, was sich gerade in der frühen Entwurfsphase, beim groben modellieren der Grundform eines Körpers, als sehr hilfreich erwiesen hat [SC-03].

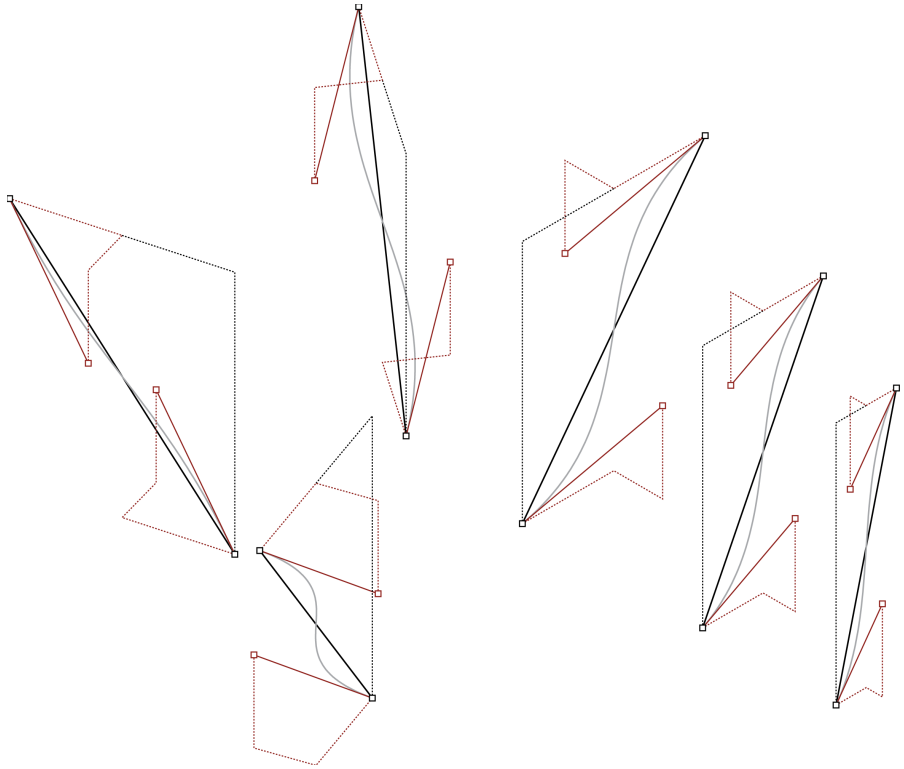
Wenn nun beide Querschnitte die gleiche Anzahl an Kontrollpunkten haben und alle Höhenverlaufskurven definiert sind, können die Schichtlinien des Objekts erzeugt werden, indem die Kontrollpolygone der Querschnittskurven entlang den Höhenverlaufskurven verschoben werden (ein Kontrollpolygon ist jenes Vieleck, das die Kontrollpunkte einer Kurve miteinander verbindet). Dazu wird auf jeder Höhenverlaufskurve eine festgelegte Anzahl von Punkten in einem regelmäßigen Höhenintervall berechnet (Abbildung 3.29).

Um einen Kurvenpunkt auf einer bestimmten Höhe zu finden, muss als erstes der entsprechende t-Wert dieses Punktes berechnet werden. Der Zusammenhang zwischen der Höhe eines Punktes und seinem t-Wert wird von der Kurvengleichung (4) hergestellt:

$$G_z = (1-t)^3 P_{0z} + 3t(1-t)^2 P_{1z} + 3t^2(1-t) P_{2z} + t^3 P_{3z} \qquad (6)$$

G_z ist die Höhe des gesuchten Punktes, P_{0z} bis P_{3z} sind die z-Koordinaten der gegebenen Kontrollpunkte. Der Parameter t ist damit die einzige Unbekannte in Gleichung

Abbildung 3.28: Gedrehte und skalierte Kurven, deren Tangentenpunkte identische RST-Werte haben



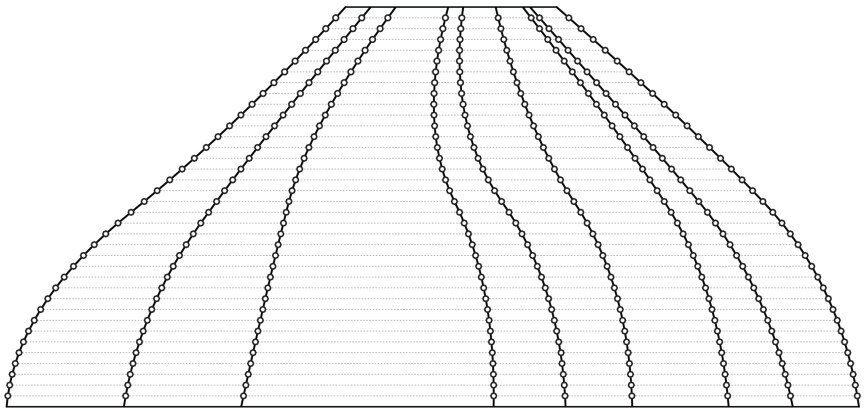


Abbildung 3.29: Neu berechnete Punkte auf den Höhenverlaufskurven

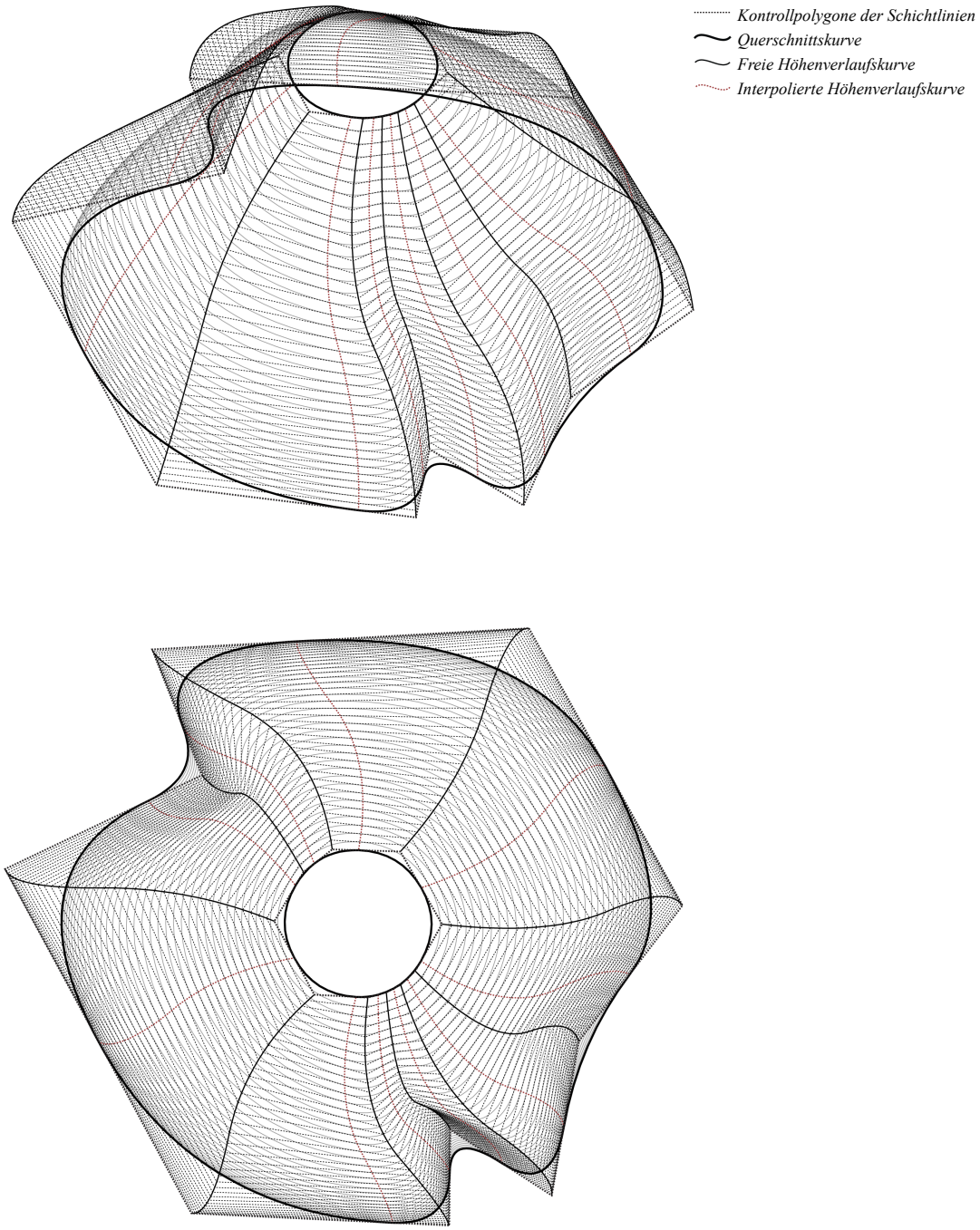


Abbildung 3.30: Verschieben der Kontrollpolygone der Querschnittskurven entlang der Höhenverlaufskurven

(6). Um ihn berechnen zu können, muss die Kurvenfunktion allerdings zuerst in eine Polynomgleichung der Form

$$a \cdot t^3 + b \cdot t^2 + c \cdot t + d = 0$$

gebracht werden. Die Umformung von (6) ergibt

$$\begin{aligned} &(-P_{0z} + 3P_{1z} - 3P_{2z} + P_{3z}) \cdot t^3 \\ &+ (3P_{0z} - 6P_{1z} + 3P_{2z}) \cdot t^2 \\ &+ (-3P_{0z} + 3P_{1z}) \cdot t + P_{0z} - G_z = 0 \end{aligned} \qquad (7)$$

Der t-Wert für den gesuchten Kurvenpunkt kann jetzt berechnet werden, indem die Nullstellen der kubischen Polynomgleichung (7) mithilfe der Cardanischen Formeln ermittelt werden[5] [SC-02c]. Die x- und y-Koordinaten des gesuchten Punktes lassen sich dann mit dem ermittelten t-Wert und der Kurvenfunktion (4) errechnen.

Wie in Abbildung 3.29 und 3.30 dargestellt, bilden alle Punkte auf gleicher Höhe jeweils ein neues Kontrollpolygon, jedes Kontrollpolygon definiert wiederum eine neue Schichtlinie des Objektes.

Wenn beide Querschnitte einen geometrisch stetigen Verlauf aufweisen, sollten auch die erzeugten Schichtlinien einen stetigen Verlauf haben. Da es so gut wie unmöglich wäre, händisch genau jene Höhenverlaufskurve zu formen, die über die gesamte Höhe keinen Knick am Verbindungspunkt zweier Teilkurven erzeugt, übernimmt das Programm diese Aufgabe. Beim Geometriemodell des Lampenschirms wird dazu jede zweite Höhenverlaufskurve interpoliert, indem die RST-Werte der benachbarten Kurven gemittelt und die Ergebnisse so korrigiert werden, dass die daraus resultierenden Schichtlinien geometrisch stetig verlaufen. Ob der Benutzer dabei die Höhenverlaufskurven der freien Kontrollpunkte oder der Verbindungspunkte der

5 Dietlinde Lau, *Algebra und Diskrete Mathematik 1*, 2. Auflage (Springer Verlag, Berlin, 2007), S. 265ff

Teilkurven kontrollieren möchte, bleibt ihm überlassen.

Zum Schluss bleibt noch ein letztes kleines Problem: wer die Schnittlinien des Modells in Abbildung 3.31 zählt, wird feststellen, dass es genau um drei mehr sind, als es überhaupt Schichten gibt. Logischerweise erzeugen die 13 Schnittlinien auf der ersten Platte 14 Einzelteile, davon sind zwei aber die inneren und äußeren Verschnittstücke. Es werden also mit den 13 Schnittlinien nur 12 Schichten aus der Platte geschnitten. Würden genau so viele Schnittlinien berechnet werden, wie es Schichten gibt, würde – je nach Sichtweise – die innere Schnittkante der innersten Schicht bzw. die äußere Schnittkante der äußersten Schicht auf jeder Platte fehlen. Pro verwendeter Platte muss also eine zusätzliche Schnittlinie gefunden werden, damit die gewünschte Anzahl an Schichten erzeugt wird.

Das Problem lässt sich am einfachsten lösen, indem das Höhenintervall der zu berechnenden Schichtlinien nicht entsprechend der Dicke der Platten gewählt wird, sondern so, dass sich die richtige Anzahl von Schnittkanten ergibt (Abbildung 3.32, Variante A). Die Schnittlinien werden auch hier in regelmäßigem Abstand über die Höhe der Form verteilt, das Höhenintervall ist aber genau um so viel reduziert, dass die benötigte Anzahl von zusätzlichen Schnittlinien erzeugt wird. Auf diese Weise entsteht garantiert ein Objekt mit einem vorhersehbaren stetigen Höhenverlauf, allerdings wird seine Form etwas von der vorgegebenen Geometrie abweichen.

Wenn die Form exakt nach Vorgabe erzeugt werden soll, müssen die Höhenverlaufskurven über ihre Endpunkte hinaus verlängert werden, damit die zusätzlichen Schnittlinien erzeugt werden können. Wenn die äußere Form der Objekthülle der vorgegebenen Geometrie entsprechen soll, werden die Kurven über den inneren Querschnitt hinaus verlängert (Abbildung 3.32, Variante B). Soll die innere Form exakt nach Vorgabe gefertigt werden, muss umgekehrt über den äußeren Querschnitt hinaus

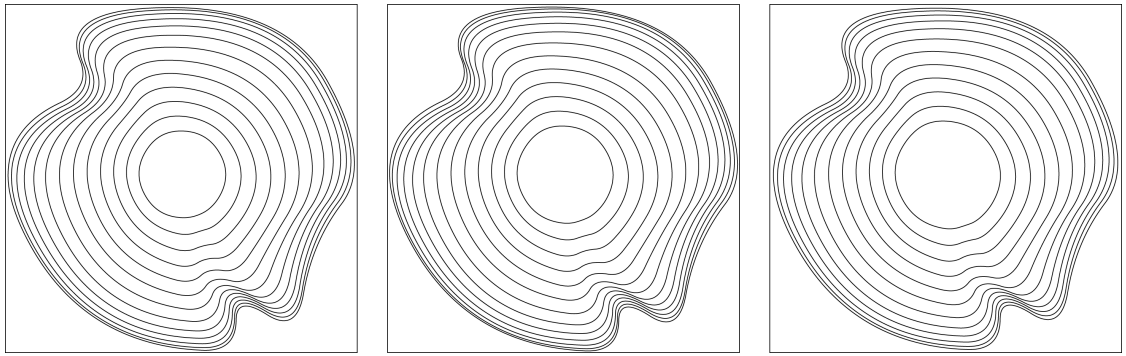


Abbildung 3.31: Schnittmuster des Lampenschirms

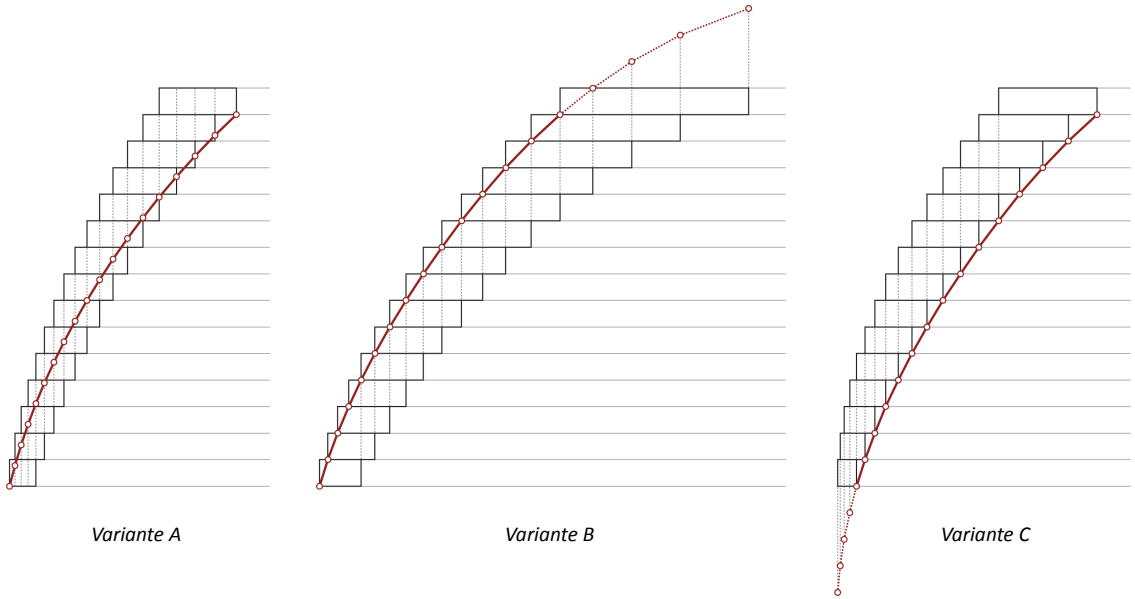


Abbildung 3.32: Varianten für die Erzeugung zusätzlicher Schnittlinien

verlängert werden (Abbildung 3.32, Variante C). Die lineare Fortsetzung der Kurven entlang ihrer Endtangenten erklärt sich von selbst. Es ist aber auch möglich, mithilfe der Kurvenfunktionen Punkte auf der Verlängerung einer Kurve zu berechnen, wenn ein t-Wert unter 0 bzw. über 1 gewählt wird. Allerdings kann das Verhalten der Kurven in ihrer Verlängerung unter Umständen zu unerwarteten Ergebnissen führen. Die Resultate dieser Methode sind weit weniger vorhersagbar, als bei Variante A. Die Entscheidung, welche Methode zur Erzeugung zusätzlicher Schichtlinien verwendet werden soll, bleibt aber dem Benutzer überlassen.

Damit ist der Weg von der Beschreibung einer Form durch Bézierkurven-Kontrollpunkte bis hin zur Berechnung der Schnittlinien zur Produktion des Objekts skizziert. Wie man sieht, ist das Geometriemodell für die Produktionsmethode maßgeschneidert. Das Modell soll die Komplexität der Erstellung und Bearbeitung von freien Formen auf wenige, leicht zu handhabende Parameter zu reduzieren und bildet damit die Grundlage für das Visualisierungs- und Entwurfsprogramm, das ich im folgenden Kapitel vorstellen möchte.

More recently architects are taking advantage of the new ability which the digital medium offers of redefining the customs and mechanisms that dictate their craft – that is, they can retool the tools that work on their thoughts

Ingeborg Rocker, *Architecture of the Digital Realm: Experimentations by Peter Eisenman & Frank O. Gehry*[1]

4. Das Programm

4.1 Wahl der Programmiersprache

Wie ich im zweiten Kapitel schon erwähnt habe, sind mittlerweile schon alle wichtigen CAD Programme mit einer Programmierschnittstelle ausgestattet, zumindest in Form einer eigenen Skriptsprache, wie etwa Mayas MEL (Maya Embedded Language), 3D Studios MaxScript oder Rhinoceros RhinoScript. Mithilfe der Programmier-schnittstellen können Geometrien erzeugt oder manipuliert werden, die dann mit dem entsprechenden Programm visualisiert und händisch weiterbearbeitet werden können.

Skriptsprachen sind bei digital arbeitenden Architekten momentan recht beliebt, sie haben aber zwei wesentliche Nachteile: erstens sind Skriptsprachen eigentlich nur für die Umsetzung einfacher Funktionen gedacht (wie etwa die Automatisierung repetitiver Tätigkeiten). Es ist, wenn auch möglich, so doch zumindest nicht empfehlenswert, mit einer Skriptsprache ein Programm vom Umfang des hier vorliegenden umzusetzen. Zweitens ist man bei der Verwendung einer programmspezifischen Skriptsprache an die jeweilige Applikation gebunden. Das bedeutet nicht nur, dass das aktuelle Projekt auf die Verwendung eines bestimmten CAD Programms eingeschränkt ist, sondern auch zukünftige Projekte, wenn man bereits geschriebenen Programmcode wiederverwenden will.

Für die Umsetzung des Programmes habe ich mich für die Programmiersprache Java entschieden. Ich bin überzeugt,

1 Ingeborg Rocker, *Architecture of the Digital Realm: Experimentations by Peter Eisenman & Frank O. Gehry*, in: Jörg H. Gleiter, Norbert Korrek, Gerd Zimmermann (Hrsg.), *Die Realität des Imaginären. Architektur und das digitale Bild* (Universitätsverlag der Bauhaus-Universität, Weimar, 2008), S. 249

dass es sich für digital arbeitende Architekten längerfristig lohnt, sich auch mit sogenannten „Hochsprachen“ zu beschäftigen.

Java ist eine objektorientierte Programmiersprache, die von Sun Microsystems entwickelt und 1995 erstmals der Öffentlichkeit vorgestellt wurde.[2] Die Grundidee objektorientierter Programmiersprachen ist, *„Daten und Funktionen, welche auf diese Daten angewandt werden können, möglichst eng in einem sogenannten Objekt zusammenzufassen und nach außen hin zu kapseln, so dass Methoden fremder Objekte diese Daten nicht versehentlich manipulieren können.“*[3] Die Vorteile der Objektorientierung hier darzulegen, würde den Rahmen dieser Arbeit sprengen, aber kurz gesagt sind Sourcecodes, die in objektorientierten Programmiersprachen verfasst wurden besser lesbar, leichter an neue Aufgaben anzupassen und damit leichter wiederzuverwenden.

In Java geschriebene Programme sind Plattform-unabhängig, *„das heißt sie laufen ohne weitere Anpassungen auf verschiedenen Rechnerarchitekturen und Betriebs-systemen mit entsprechender Laufzeitumgebung.“*[4] Möglich macht das die Java Virtual Machine, die den kompilierten Java Bytecode der Programme für das jeweils verwendete Betriebssystem sozusagen „übersetzt“. Java-Programme laufen also unter Windows genauso wie unter Linux oder auf dem Mac, sogar auf Smartphones oder Handys, ohne dass der Programmcode verändert werden muss.

In den letzten 15 Jahren hat sich Java zur populärsten Programmiersprache der Gegenwart entwickelt.[5] Auch wenn Popularität nicht zwangsläufig nicht auf Qualität schließen lässt, in einer Zeit in der Open Source Projekte eine immer größere Rolle spielen, ist die Verbreitung einer

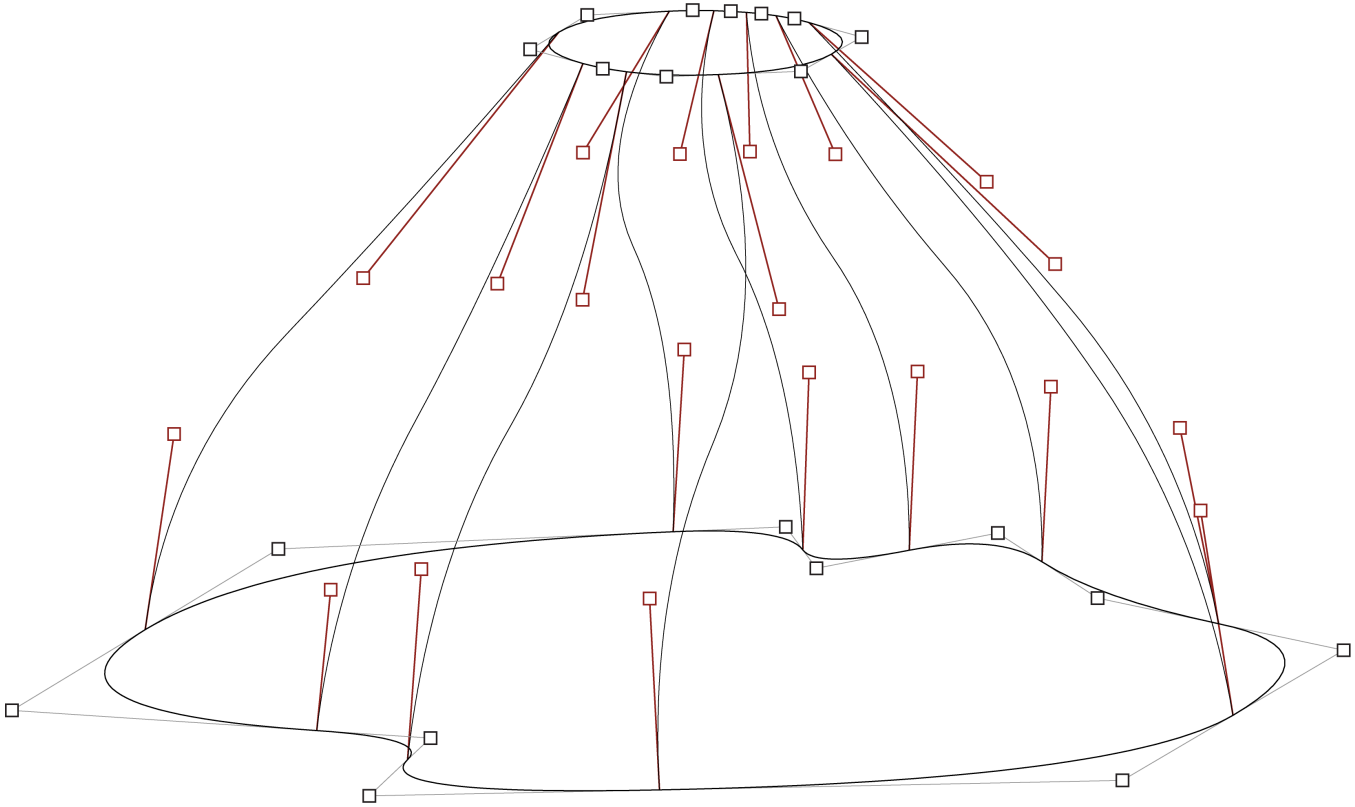


Abbildung 4.1: Geometriemodell

2 Todd Greanier, *Java foundations* (Sybex, Alameda, 2004), S. 2
3 http://de.wikipedia.org/wiki/Objektorientierte_Programmierung
4 http://de.wikipedia.org/wiki/Java_(Programmiersprache)
5 laut TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

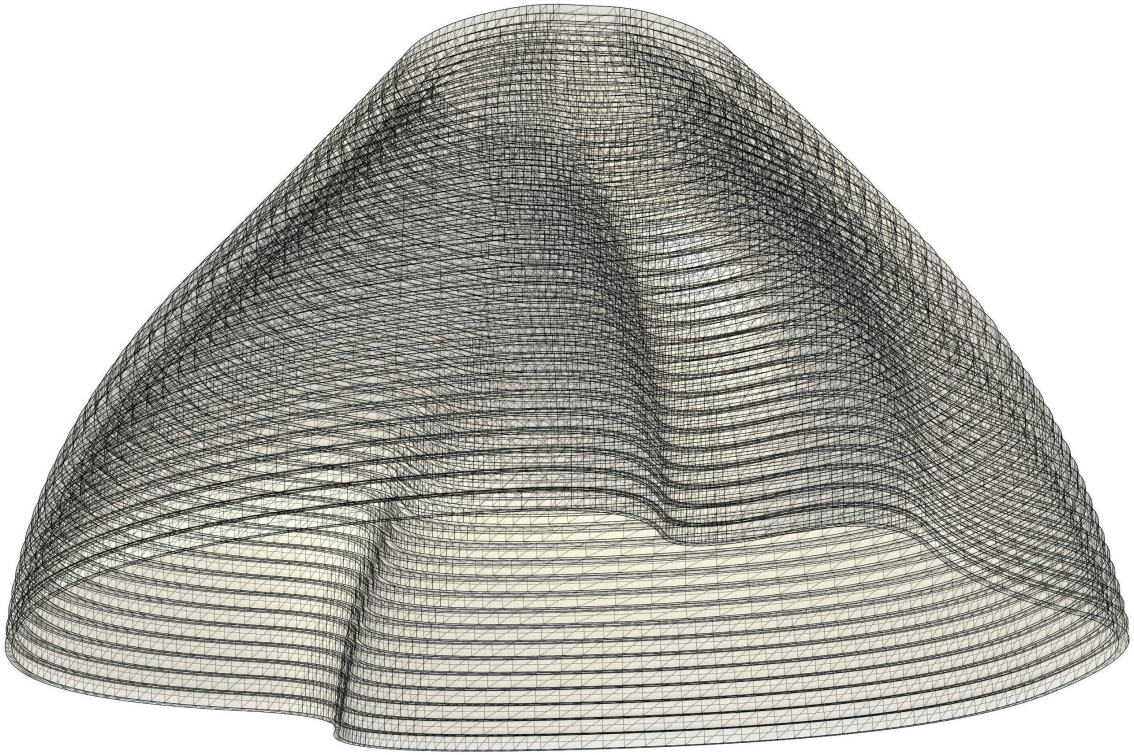


Abbildung 4.2: Das generierte dreidimensionale Polygonmodell

Programmiersprache kein zu unterschätzender Faktor.

4.2 Die Rolle des Programmes

Das Herzstück der Software ist die algorithmische Abbildung des im vorangegangen Kapitel beschriebenen Object-Scaping Methode. Mithilfe dieser algorithmischen Beschreibung entsteht, wenn man so will, eine „intelligente“ Geometrie, eine Form, die über die Bedingungen ihres eigenen Herstellungsprozesses bescheid „weiß“ und sich selbständig entsprechend ihrer Produktionsbedingungen organisiert.

Die gewünschte Form eines Objektes wird, wie in Abbildung 4.1 noch einmal dargestellt, durch die oberen und unteren Querschnitte und die Kurven des Höhenverlaufes festgelegt. Das im vorhergehenden Kapitel vorgestellte parametrische Modell der Geometrie nimmt die Kontrollpunkte (in der Abbildung als Quadrate markiert) der Kurven als seine Eingangsparameter, berechnet die einzelnen Höhenschichtlinien des Objektes und erzeugt automatisch die Schnittmuster für die Fabrikation des Objektes. Darüber hinaus generiert das parametrische Modell aus den Informationen der Schnittmuster die dreidimensionale Geometrie für die Visualisierung der entstehenden Form. Nur von den wenigen Kontrollpunkten der formerzeugenden Kurven als Parameter des Modells ausgehend, erzeugt der Algorithmus die in Abbildung 4.2 dargestellte, komplexe dreidimensionale Geometrie des herzustellenden Objektes.

Genau das ist es, was ein parametrisches Modell so wertvoll macht. Die Kontrolle auch hochkomplexer Vorgänge kann durch ein parametrisches Modell wesentlich vereifacht werden, indem die Steuerung der Prozesse durch das Modell auf eine überschaubare Anzahl von Parametern reduziert wird. Auf diese Weise wird Komplexität bewältigbar. Je weniger Parameter im Spiel sind, desto

leichter ist ein Modell zu bearbeiten. Die Berechnung der Pozesse, die mit den Parametern assoziert sind, verlaufen automatisch im Hintergrund, so kann sich der Entwerfer auf das Wesentliche konzentrieren: nämlich aufs Entwerfen.

Dabei ist die Reduktion der Steuerungsparameter aber nicht zu verwechseln mit der Abstraktion der Problemstellung, die mit dem parametrischen Modell bearbeitet wird.

„The mind of an architect or designer is trained to be receptive to the contextual shifts and cross-dependencies that drive the creative process. Abstraction plays a crucial role in reducing complexity in a meaningful way so that it becomes possible for the designer to process and interact with the evolving problem. But abstraction can be counter productive in those design problems that rely on the interdependency of multiple design constraints. In fact, abstraction might compromise some particular features critical to design.

Where the constraint relationships rely on precise, computationally intensive dependencies, a digital implementation has advantages over an abstracted one. This is where digital models can make a difference in design exploration. They can help to overcome the complexity barrier in design problems that can only be explored further by taking into account all cross dependencies and that would suffer from a loss of design relevant features through abstraction.“ [6]

Die Methode der zweidimensionalen Schichtung produziert Formen mit einem Reichtum an Details, deren Feinheiten sich aber, wie im vorangegangenen Kapitel beschrieben, aus einem komplexen Prozess heraus ergeben. Die sorgfälte Ausformulierung der Details ist nur in einem interaktiven Prozess möglich, in dem die Auswirkungen von Veränderungen am Entwurf sofort visualisiert werden.

Die Hauptaufgabe der Object-Scaper Software ist die interaktive Darstellung von programmierten oder von Hand modellierten Formen, und der schlussendliche Zweck des Programmes ist natürlich die Organisation der Schnittmuster und die automatische Generierung von digitalen Daten für den Fabrikationsprozess. Das Programm ist aber mehr als das. Es ist ein Werkzeug zur Form-findung, ein *Design Explorer*, um einen Begriff von Axel Kilian zu übernehmen.

„Computation can aid in the discovery of design solutions by modeling the constraints and the design representation into design explorers. Design explorers offer an alternative form of abstraction. Through the computationally based externalization of networks of dependency relations, design explorers capture and store states of the process for the designer to interact with. Through the interaction, additional layers of the constructed design model may be revealed. This externalized, computational representation offers a distinctly separate set of explorations aside from abstraction alone.“ [7]

Voraussetzung für das interaktive Gestalten ist das sofortige Feedback über Entwurfsentscheidungen, und damit eine performante Visualisierung von Entwürfen. Um eine ansprechende Darstellung von geschwungengen Formen zu erreichen, ist es durchaus möglich, dass 3D-Modelle komplexerer Entwürfe für die Darstellung in der Object-Scaper Software aus mehreren hundertausend Polygonen bestehen können. Eine hardwarebeschleunigte Visualisierung ist also unerläßlich.

Zu diesem Zweck nutze ich in Ergänzung zur *Java Standard Edition (SE) 6 Plattform*[8] die Software-Bibliothek *JOGL* (Java Binding for the OpenGL API)[9] für hardwarebeschleunigte grafische Darstellungen. Die *Open Graphics Library* (OpenGL) *„ist eine Spezifikation für eine plattform- und programmiersprachenunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafik. Der OpenGL-Standard beschreibt etwa 250 Befehle, die die Darstellung komplexer 3D-Szenen in Echtzeit erlauben.“* [10]

Mithilfe der OpenGL API ist es möglich, die Grafikhardware eines Computersystems direkt zu programmieren und damit die Darstellung von Grafiken wesentlich zu beschleunigen.

4.3 Das User-Interface

Abbildung 4.3 zeigt einen Screenshot des Object-Scaper Programms. Der größte Teil des Fensters nimmt die OpenGL-Darstellung (weißer Hintergrund) mit einer perspektiven Ansicht des entstehenden Objektes (rechts) und der Plandarstellung der Schnittmuster (links) in Anspruch. Um möglichst Effizient mit der zur Verfügung stehenden Bildschirmfläche umzugehen, kann die Plandarstellung mit der perspektiven Ansicht überlagert werden. Beide Darstellungen können beliebig verschoben und herangezoomt werden, die Perspektive kann zudem frei rotiert werden, damit das Objekt von jedem Blickpunkt aus betrachtet werden kann.

Am linken Fensterrand befindet sich eine Toolbar mit den wichtigsten Bearbeitungswerkzeugen für das Verschieben, Rotieren, Skalieren und Einfügen von Punkten. Am oberen Rand des Fensters findet sich das Programmmenü und am rechten Rand ein ausklappbares Bedienelement mit einer Listenansicht der einzelnen Modellelemente („Model explorer“) ganz oben, einem Dropdown-Menü, mit dem bestimmt werden kann, was in der Schnittmuster-darstellung angezeigt werden soll, und einem kontextbezogenen Bedienfeld darunter, bei dem zum Beispiel die Koordinaten von selektierten Punkten numerisch eingegeben werden können.

6 Axel Kilian, *Design Exploration through Bidirectional Modeling of Constraints* (http://dspace.mit.edu/handle/1721.1/33803, 2006), S. 23

7 Ebd.

8 http://www.oracle.com/technetwork/java/javase

9 http://www.kenai.com/projects/jogl

10 http://de.wikipedia.org/wiki/Opengl

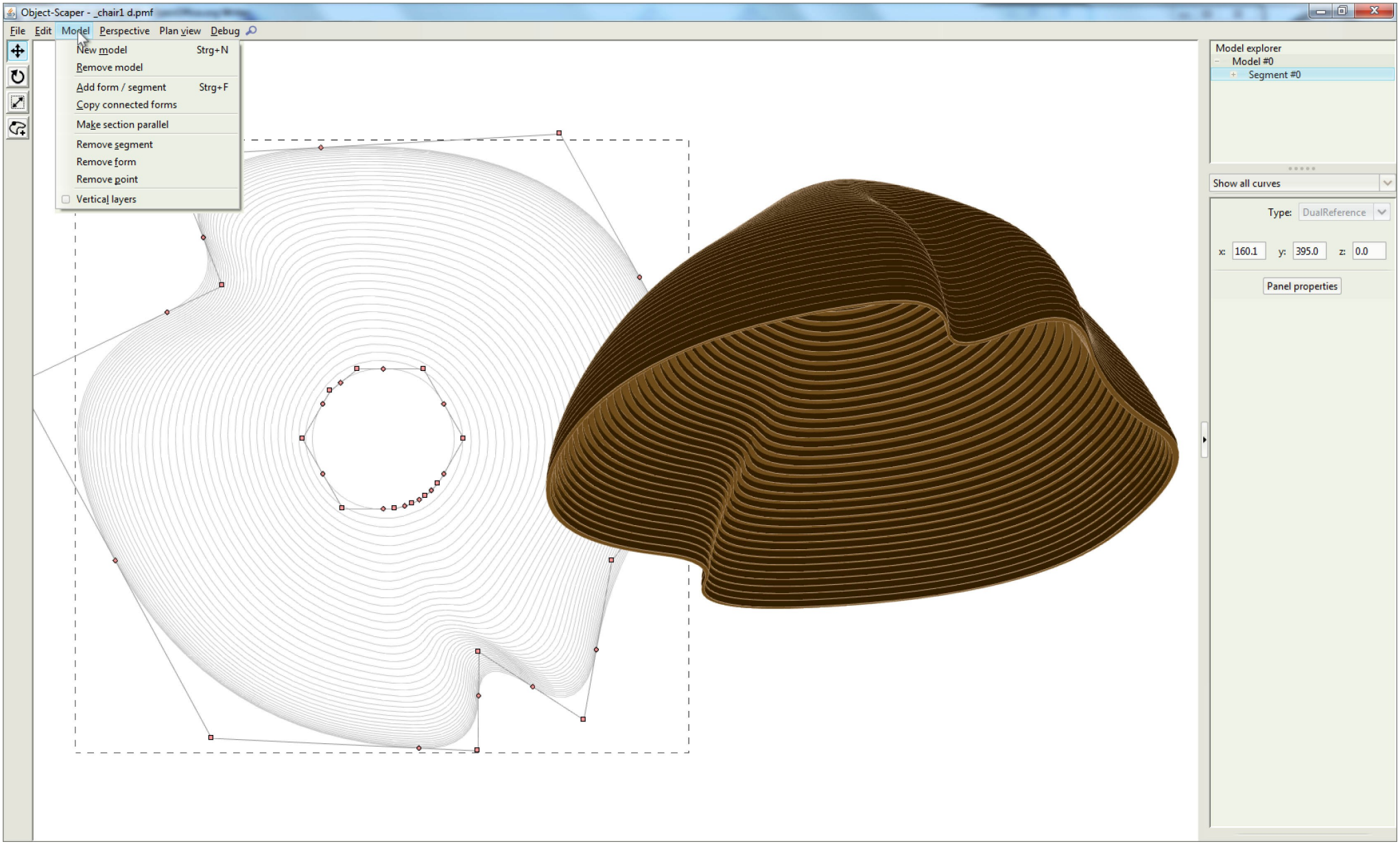


Abbildung 4.3: Screenshot des Hauptfensters

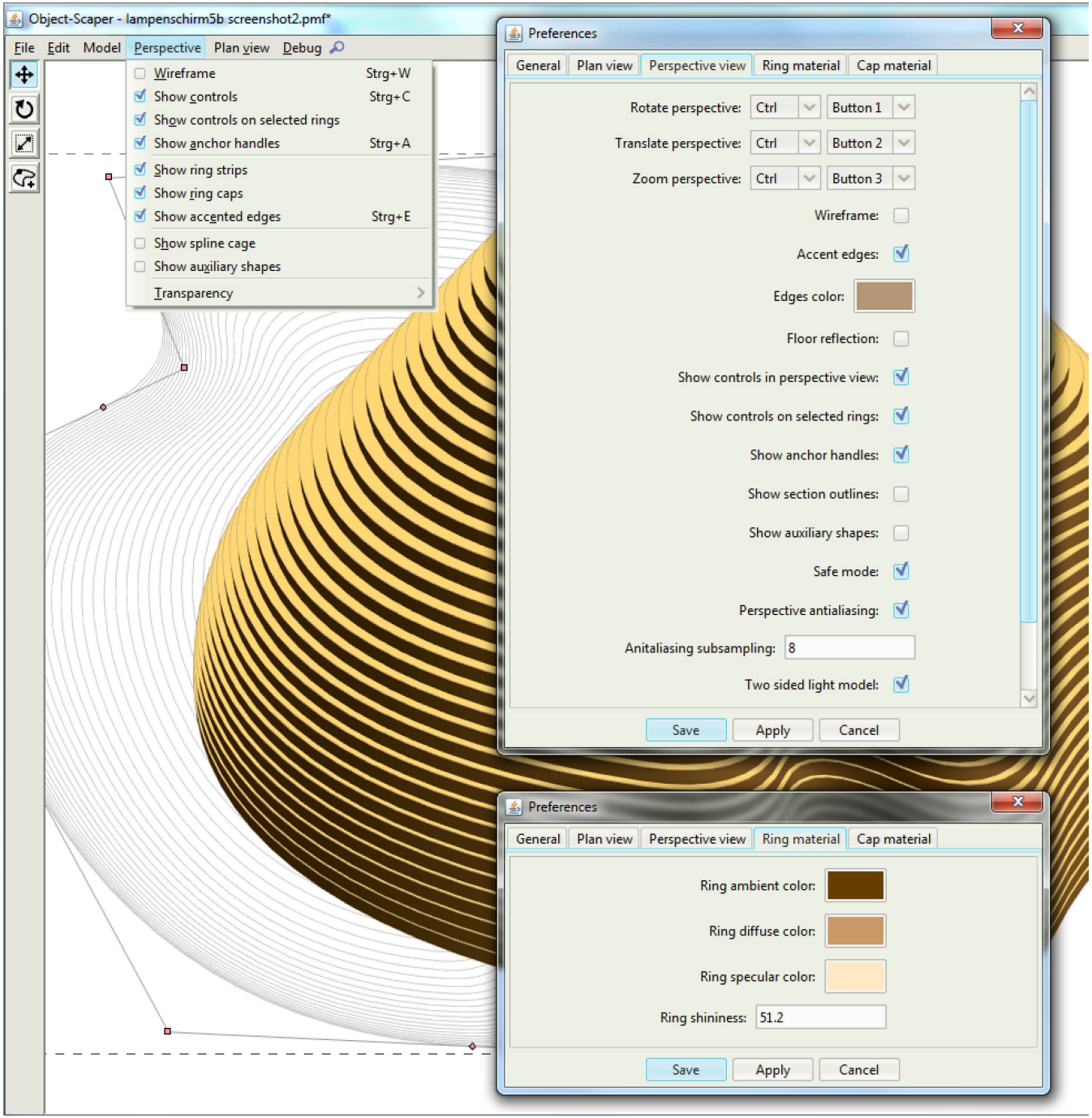


Abbildung 4.4: Anpassungsoptionen der perspektiven Ansicht

Für die dreidimensionale Darstellung stehen drei verschiedene Modi zur Verfügung: erstens eine schattierte perspektive Ansicht wie eben in Abbildung 4.3 dargestellt, zweitens eine Drahtmodell-Ansicht (Abbildung 4.7) und schlussendlich eine noch weiter abstrahierte Darstellung, die auf die formdefinierenden Kurven des Geometrie-modells reduziert ist (Abbildung 4.6).

Die schattierte Ansicht des Modells kann auf die unterschiedlichste Weise variiert werden (Abbildung 4.4). Die Anpassungsmöglichkeiten reichen von der getrennten Festlegung der Materialeigenschaften für die Schnitt- und Schichtoberflächen bis zur Selektion der in der perspektive angezeigten User-Interface Elementen, und von der Hervorhebung der Objektkonturen bis zur Festlegung der Antialiasing-Qualität.

Natürlich produziert die in Echtzeit berechnete OpenGL Darstellung keine qualitativ gleichwertigen Bilder, wie etwa ein Raytrace-Renderer. Zu diesem Zweck ist es aber möglich, die dreidimensionale Polygongeometrie des Objektes als DXF-Datei zu exportieren, um sie dann in einem beliebigen 3D-Modellierprogramm weiter zu bearbeiten.

Während die schattierte Ansicht eine exakte Abbildung der entstehenden Objektegeometrie darstellt, eigenen sich die beiden abstrakteren Drahtmodell-Ansichten besser zur Bearbeitung des Modells im Raum. Die Kontrollpunkte der Querschnitts- und Höhenverlaufskurven werden in der Perspektive als Kuben markiert, die mit der Maus entweder einzeln oder in Gruppen ausgewählt und dann manipuliert werden können. Wird ein Kontrollpunkt ausgewählt, färbt er sich, wie in Abbildung 4.6 dargestellt, dunkelrot und es erscheint ein Bearbeitungs-Widget zum Verschieben des selektierten Punktes oder der ausgewählten Punktgruppe entlang der x-, y- oder z-Achse.

Die Bearbeitung der Höhenverlaufskurven ist nur in der 3D-

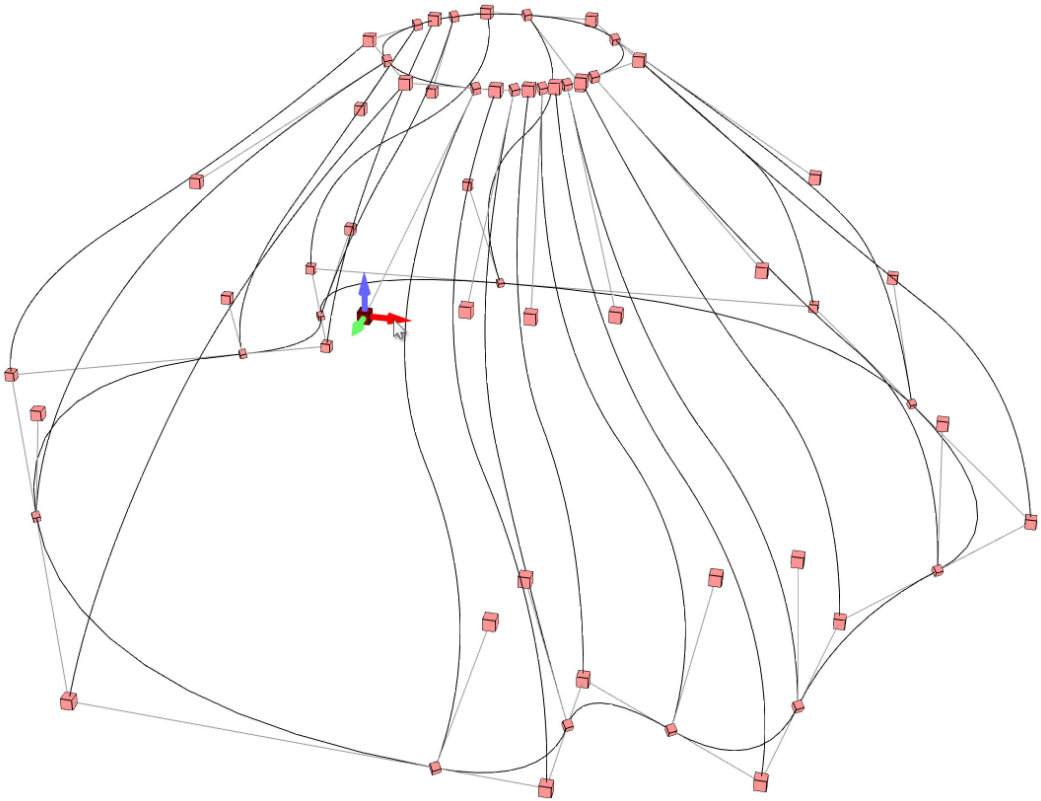


Abbildung 4.6: Darstellung und Bearbeitung der formdefinierenden Kurven in der perspektiven Ansicht

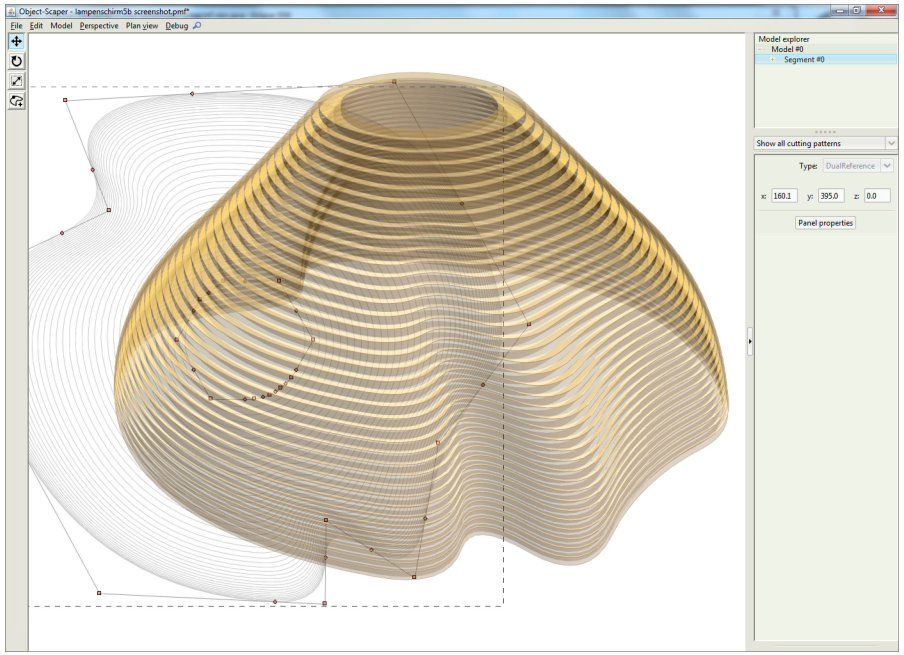


Abbildung 4.5: Transparente Perspektive

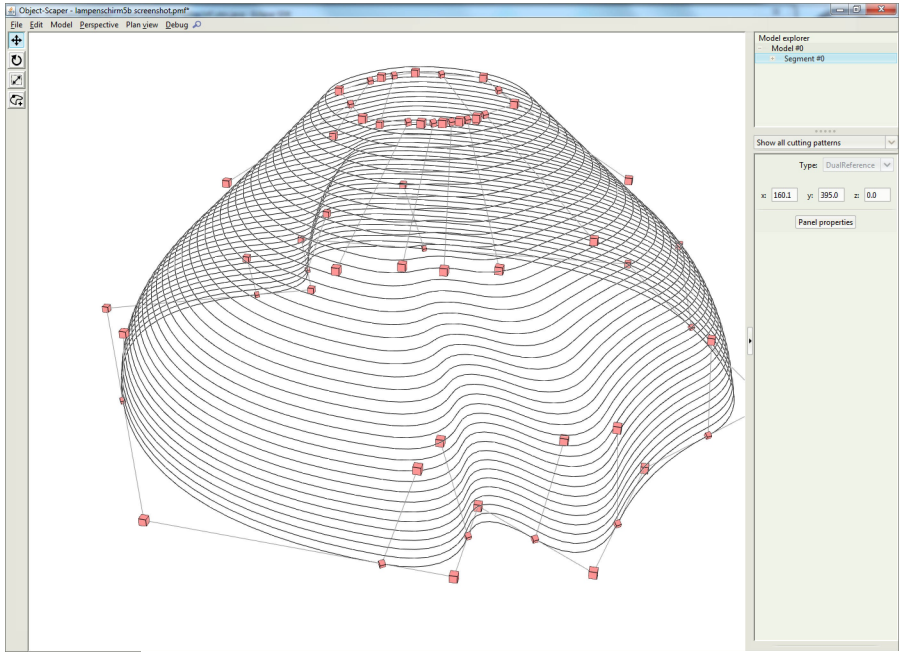


Abbildung 4.7: Wireframe-Darstellung

Ansicht sinnvoll. Dagegen können die Querschnittskurven sowohl in der Perspektive als auch in der zweidimensionalen Planansicht der Schnittmuster (Abbildung 4.8) editiert werden. Die Kontrollpunkte der Kurven werden in der Planansicht als Quadrate dargestellt und können wieder einzeln oder in Gruppen selektiert und dann verschoben, skaliert oder rotiert werden. Außerdem kann auch gleich eine Querschnittskurve als Ganzes ausgewählt und manipuliert werden. Selektierte Querschnittskurven werden in der Planansicht als rot gestrichelte Kurven angezeigt. Das schwarze gestrichelte Rechteck markiert den Rand der Platten, aus denen das Objekt gefertigt wird. Selbstverständlich kann das Plattenformat frei gewählt werden.

Mithilfe des Dropdown-Menüs im rechten Teil des Fensters kann festgelegt werden, welche Schnittlinien in der Planansicht dargestellt werden. Entweder können alle Schnittlinien übereinandergelegt oder aber auch nur die Schnittmuster einzelner Platten angezeigt werden (Abbildung 4.9).

Außer den Schnittlinien werden in der Planansicht auch noch die Kontrollpolygone der Querschnittskurven dargestellt. Wie man sieht, besteht die markierte Querschnittskurve in Abbildung 4.8 aus 10 einzelnen quadratischen Bézierkurven. Ihr Verlauf ist stetig, das heißt es gibt keinen Knick am Übergang der einzelnen Teilkurven. Damit die zusammengesetzte Kurve tatsächlich stetig verläuft, müssen die Endpunkte der Teilkurven (die um 45° gedrehten Quadrate in der Abbildung) auf der Verbindungslinie zwischen den freien Kontrollpunkten liegen. Um den stetigen Verlauf der Kurve zu garantieren, können die Endpunkte in diesem Fall von vornherein nur entlang der Verbindungsgeraden verschoben werden. Wird einer der freien Kontrollpunkte versetzt, verschieben sich auch die betroffenen Endpunkte automatisch so, dass der Übergang der Teilkurven stetig bleibt. Die Punkte sind in

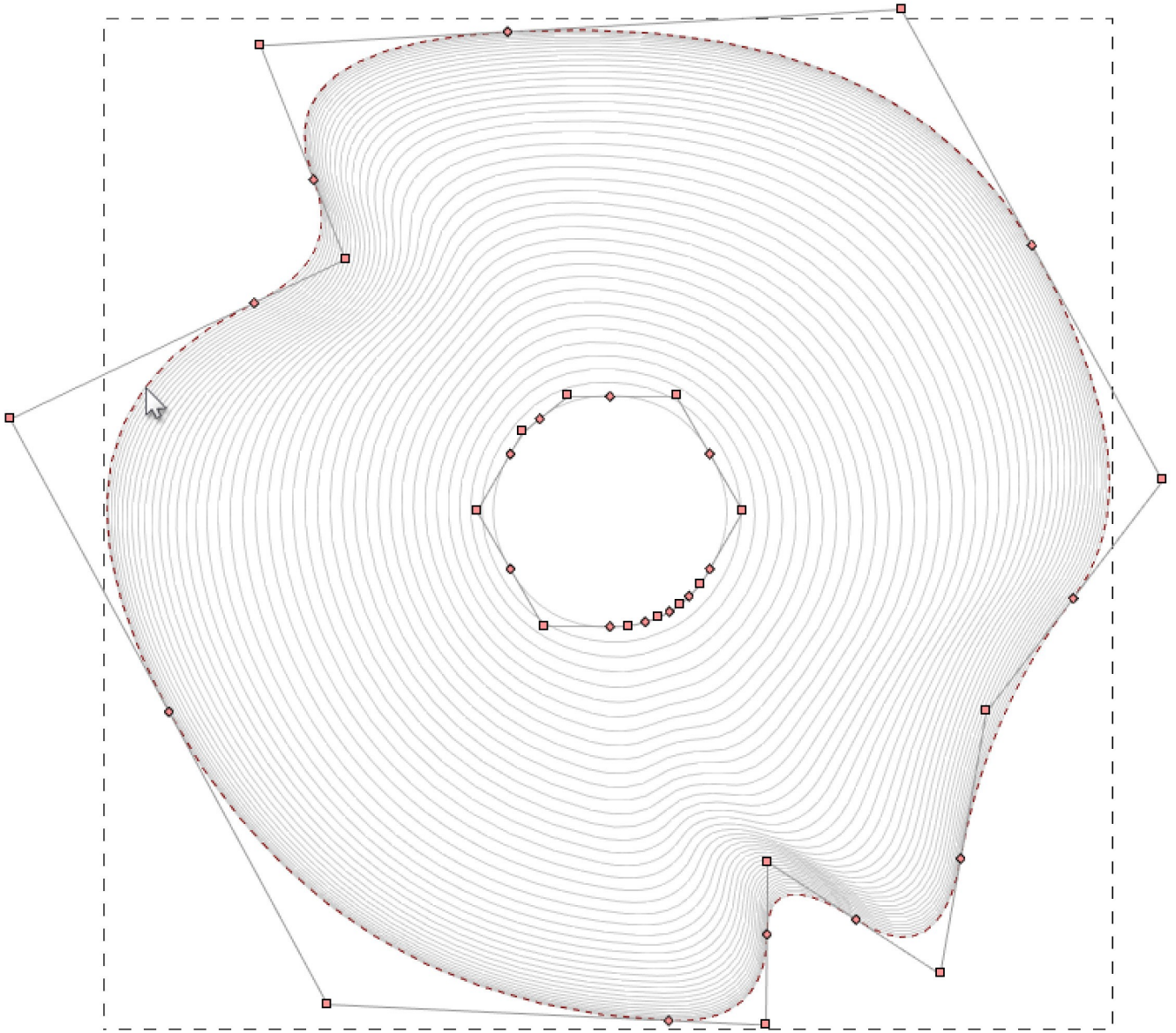
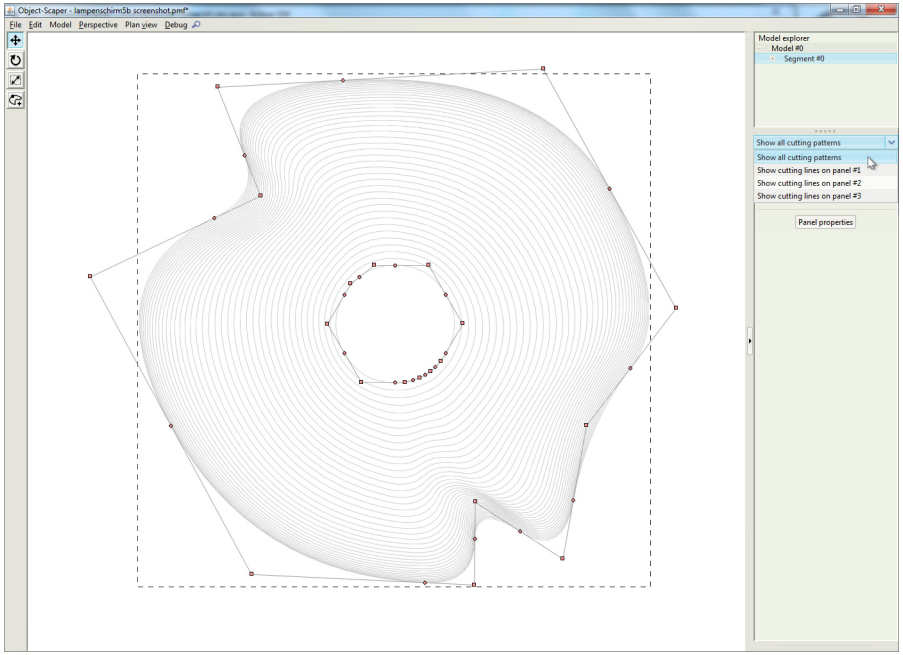


Abbildung 4.8: Bearbeitung der Querschnittskurven in der Planansicht der Schnittmuster



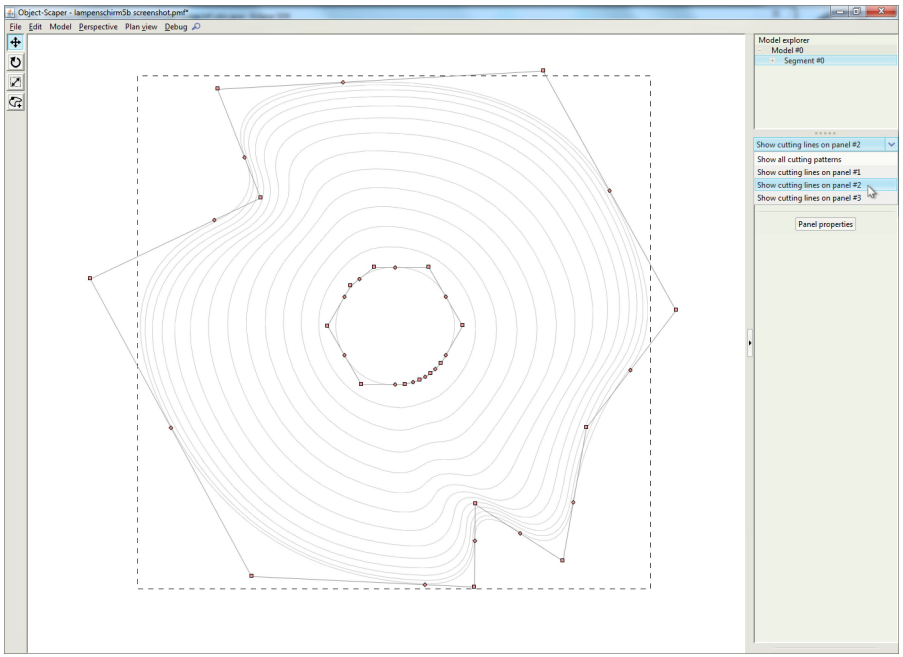


Abbildung 4.9: Wahl der darzustellenden Schnittlinien

diesem Fall mehr als nur Koordinatenangaben - sie „wissen“ sozusagen, dass sie auf der Verbindungsgeraden der benachbarten freien Kontrollpunkte liegen müssen, und berechnen ihre Position automatisch nach dieser Vorgabe.

Es gibt bei dieser algorithmischen Geometriebeschreibung also verschiedene „Arten“ von Punkten, die sich entsprechend ihrer Programmierung selbständig organisieren. Anhand der konkreten Implementierung der verschiedenen Punktetypen für dieses Projekt [SC-04 bis SC-09] kann gleichzeitig der vielleicht wesentlichste Vorteil parametrischer Modelle sowie eines der zentralen Konzepte objektorientierter Programmierung veranschaulicht werden, nämlich das assoziative Verhalten parametrischer Objekte und die Polymorphie objektorientierter Klassen.

Parametrisch-assoziatives Verhalten bedeutet nichts anderes, als dass die Änderung eines einzelnen Parameters automatisch auch eine Änderung der mit diesem Parameter assoziierten Strukturen des parametrischen Modells auslöst, dass also – um beim Beispiel der Teilkurvenend-punkte zu bleiben – nach dem Verschieben eines freien Kontrollpunktes automatisch die Position der benachbarten Kurvenendpunkte neu berechnet wird, so dass der stetige Übergang zwischen den Teilkurven erhalten bleibt, was dann wiederum Auswirkungen auf das gesamte restliche Geometriemodell hat, von der Schnittmusterorganisation bis zum OpenGL-Modell.

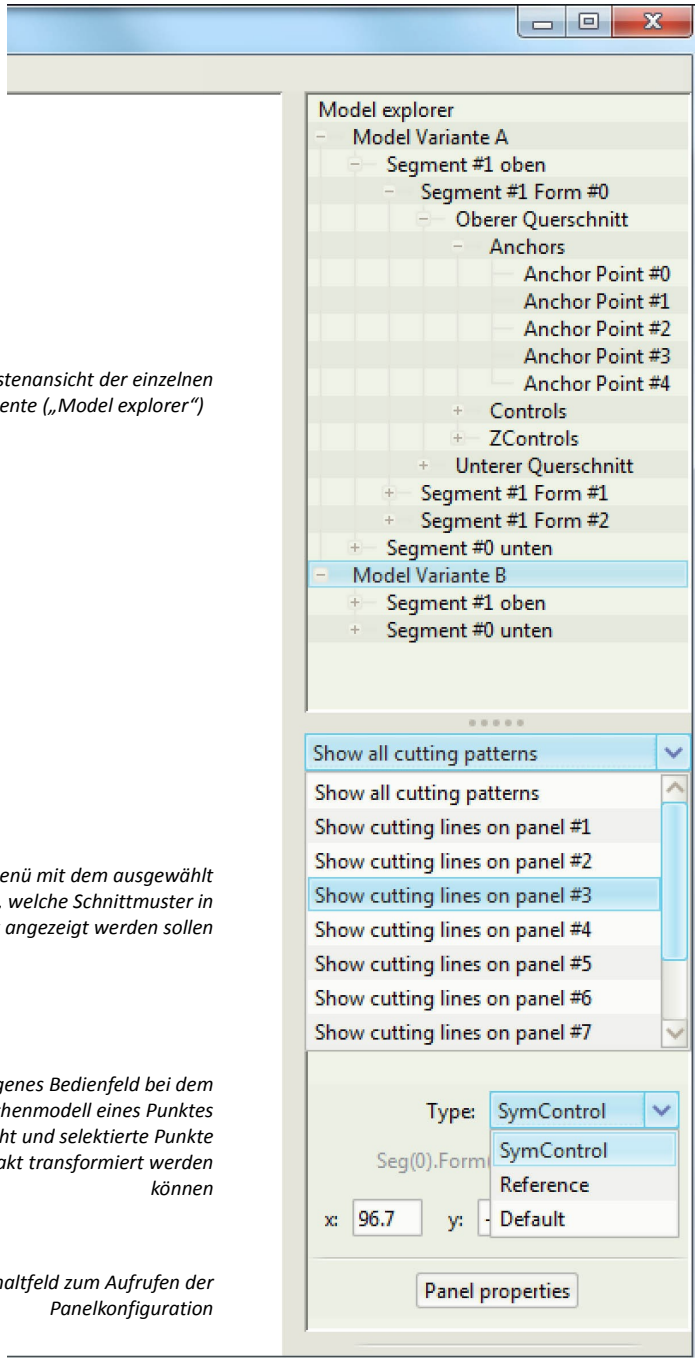
Die Basisklasse der Punktbeschreibung ist die Klasse *Point3D* [SC-04]. Jedes Point3D-Objekt hat ein eigenes ihm zugeteiltes Rechenmodell (*Point3dModel* [SC-05]), das festlegt, wie das Objekt die Koordinaten des von ihm repräsentierten Punktes zu errechnen hat. Im einfachsten Fall – bei den freien Kontrollpunkten etwa – speichert das Rechenmodell einfach nur die x-, y- und z-Koordinaten des Punktes und gibt sie bei Bedarf wieder aus [SC-07]. Bei den Teilkurvenendpunkten dagegen kalkuliert das Rechen-modell die Position des Punktes, indem es die Strecke

zwischen den beiden benachbarten freien Kontrollpunkten in einem festgelegten Verhältnis teilt [SC-09]. Da das Rechenmodell ein von der eigentlichen Point3D-Instanz getrenntes Objekt ist, kann es jederzeit ausgewechselt werden. Es ist also möglich, zur Laufzeit des Programmes das Rechenmodell jedes Punktes auszutauschen, ohne dass die internen Verknüpfungen des parametrischen Modells mit dem Punkt selbst verloren gehen. So kann mit einem einzelnen Mausklick beispielsweise der stetige Übergang zwischen zwei Teilkurven gebrochen und stattdessen ein geknickter Übergang geformt werden.

Um das Rechenmodell eines Punktes auszutauschen, steht in der Werkzeugleiste am rechten Fensterrand ein Bedienelement zur Verfügung (Abbildung 4.10 unten), mit dem es außerdem möglich ist, einzelne oder eine mehrere selektierte Punkte numerisch exakt zu transformieren.

Der „Model explorer“ oben listet die momentan geöffneten Modelle sowie ihre einzelnen Elemente auf. Um ein effizientes Versioning von Entwürfen zu ermöglichen, können mehrere Modelle gleichzeitig geöffnet und miteinander verknüpft werden. Der aktuell bearbeitete Entwurf kann mit einem Knopfdruck geklont und der Modellliste hinzugefügt werden. Damit wird es möglich, schnell und einfach verschiedene Varianten eines Entwurfs zu erstellen. In der Listenansicht kann augenblicklich von einem zum nächsten Modell gewechselt werden, um Entwurfsvarianten zu vergleichen oder Objektdifferenzierungen in programmierten, individualisierten Serien beurteilen zu können.

Außerdem können die einzelnen Elemente des Modells in der Explorer-Liste benannt und durch Anklicken gezielt selektiert werden, was die Bearbeitung von Entwürfen nochmal etwas übersichtlicher macht.



Listenansicht der einzelnen Modellelemente („Model explorer“)

Dropdown-Menü mit dem ausgewählt werden kann, welche Schnittmuster in der Planansicht angezeigt werden sollen

Kontextbezogenes Bedienfeld bei dem das Rechenmodell eines Punktes ausgetauscht und selektierte Punkte numerisch exakt transformiert werden können

Schaltfeld zum Aufrufen der Panelkonfiguration

Abbildung 4.10: Werkzeugleiste am rechten Fensterrand

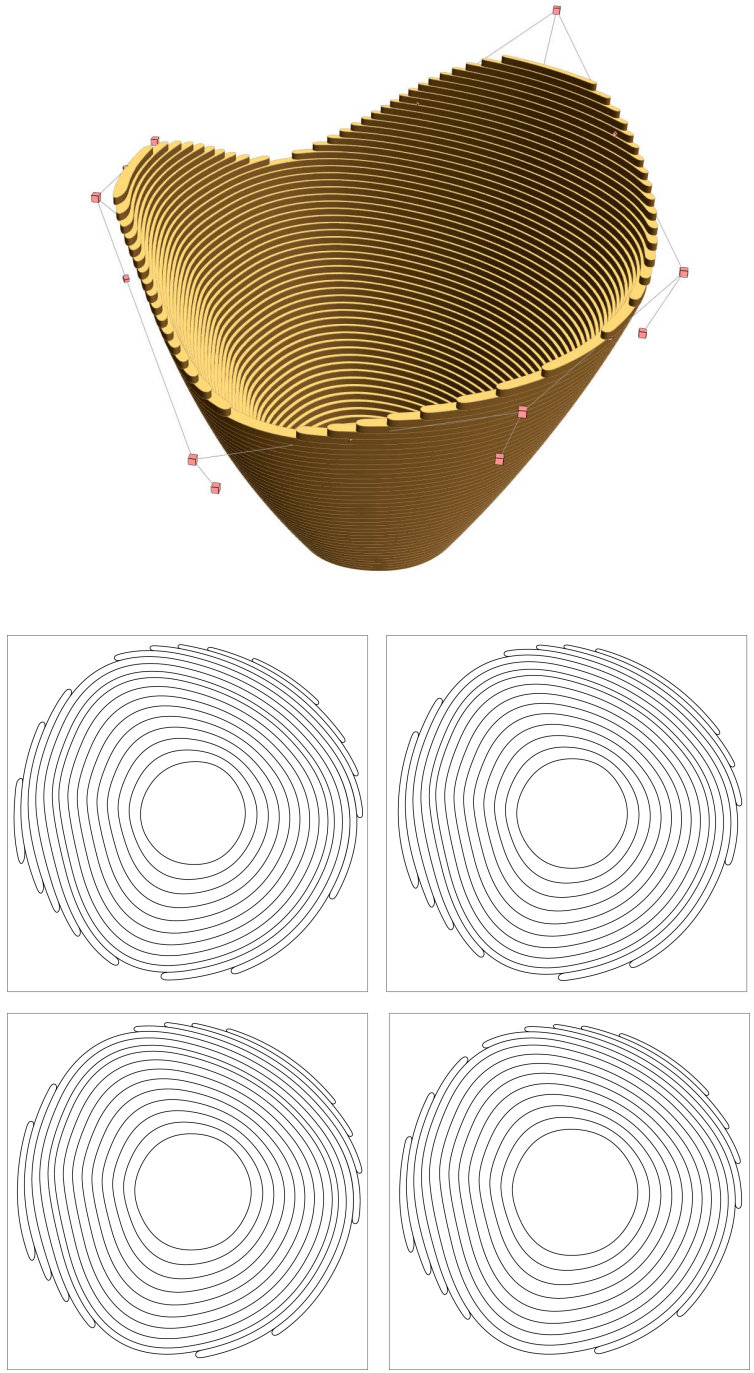


Abbildung 4.11: Objekt mit dreidimensionalen Randkurven

4.4 Erweiterung der Möglichkeiten

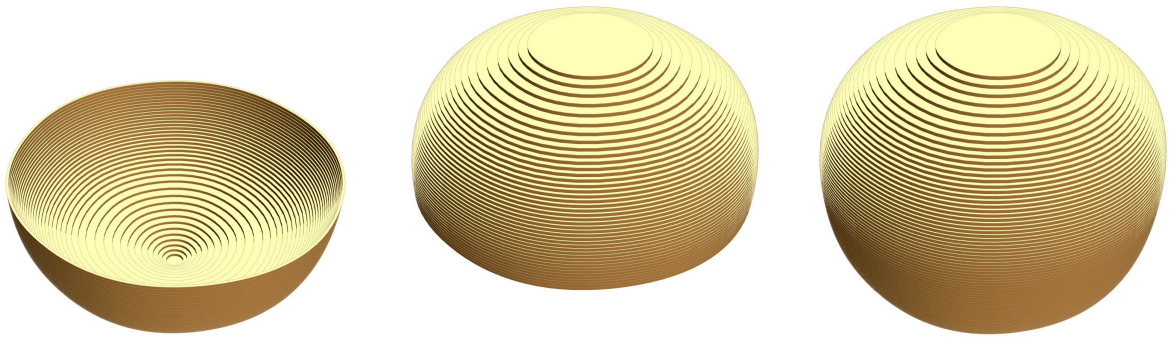
Mit dem im dritten Kapitel behandelten Geometriemodell lässt sich schon eine Fülle von Formen beschreiben, die mit der Object-Scaping Methode hergestellt werden können. Die Herstellungsmethode bietet aber noch mehr formale Möglichkeiten, die allerdings mit dem bis jetzt vorgestellten parametrischen Modell noch nicht beschrieben werden können.

Die erste Erweiterung der formalen Möglichkeiten stellen dreidimensionale Randkurven dar. Bis jetzt wurden nur Formen mit zweidimensionalen Querschnittskurven als Randbedingung vorgestellt, aber die Herstellungsmethode erlaubt auch Formen mit unebenen Rändern, wie bei dem Objekt in Abbildung 4.11.

Um die entsprechenden Schnittmuster zu erzeugen, werden die ebenen Höhenschichtlinien mit der dreidimensionalen Randkurve geschnitten. Dort wo sich die Randkurve mit einer Höhenschichtlinie überkreuzt, wird die jeweilige Schichtlinie auseinander geschnitten und mit der jeweils weiter innen liegenden Kurve im Schnittmuster verbunden. Die Verbindung der Schnittlinien kann wahlweise eckig oder, wie in Abbildung 4.11, abgerundet ausgeführt werden.

Die größte Einschränkung der Herstellungsmethode ist aber mit Sicherheit, dass sich keine Schichtlinien überschneiden dürfen und in der Konsequenz keine überhängenden Formen produziert werden können. Um eine solche Form herzustellen, können allerdings zwei oder mehr getrennt hergestellte Objekte zu einem Ganzen zusammengesetzt werden. Die einzige Bedingung ist, dass jedes Teilobjekt für sich keine überhängende Form aufweist. Auf diese Weise können nicht nur überhängende Formen erzeugt werden, sondern auch geschlossene Körper, wie die Kugel in Abbildung 4.12, die aus zwei einzeln hergestellten Segmenten besteht.

Abbildung 4.12: Aus zwei einzeln gefertigten Halbkugeln zusammengesetzte geschlossene Form



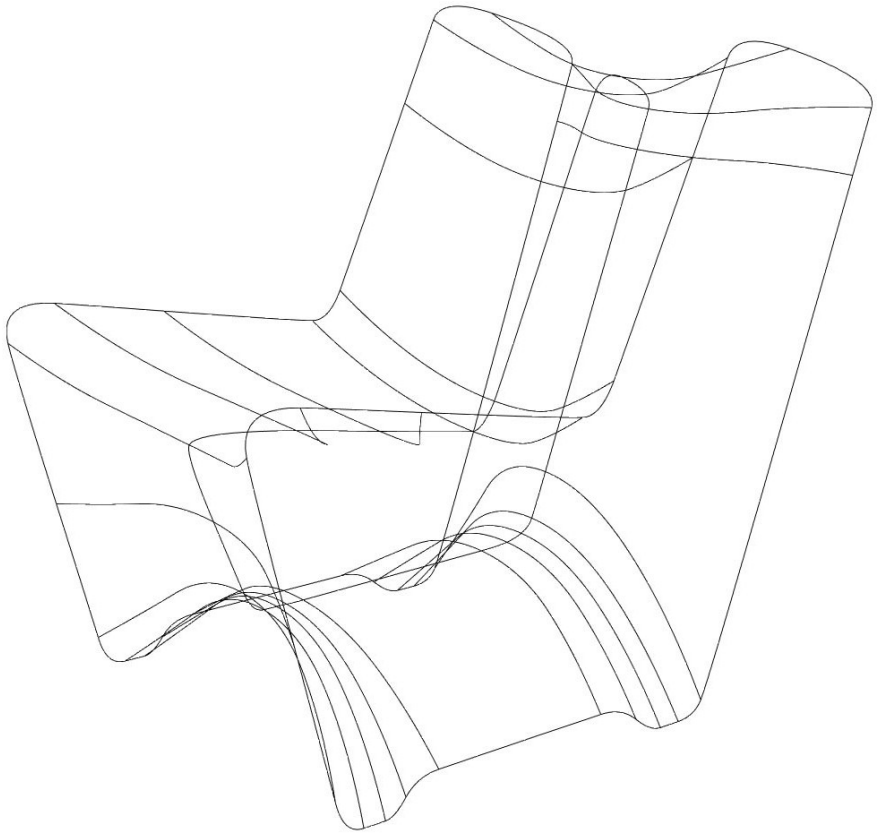


Abbildung 4.13: Querschnitts- und Höhenverlaufskurven des Sessels in Abbildung 4.14

Natürlich müssen die Schichten eines Objektes nicht zwangsläufig horizontal übereinander angeordnet werden. Abbildung 4.14 zeigt einen Entwurf eines Sessels, bei dem die „Höhenschichtlinien“ vertikal angeordnet sind. Auch dieses Objekt ist aus zwei Segmenten zusammengesetzt: die linke und die rechte Hälfte müssen mit zwei getrennten Schnittmustersystemen hergestellt werden, da sich sonst einzelne Schichten wieder überschneiden würden.

Die beiden Segmente teilen sich dabei genau in der Mitte des Sessels die selbe innere Querschnittskurve (Abbildung 4.13). Damit zwischen den beiden Segmenten ein sanfter Übergang und kein Knick entsteht, müssen die Höhenverlaufskurven der beiden Segmente (die in diesem Fall nicht vertikal sondern horizontal verlaufen) geometrisch stetig ineinander übergehen. Wenn sich zwei Segmente eine Querschnittskurve teilen, kümmert sich das parametrische Modell automatisch um den stetigen Übergang der Höhenverlaufskurven.

Bei vollkommen symmetrischen Segmenten können auch die äußeren Querschnittskurven miteinander Verknüpft werden, so dass Änderungen an der einen Randkurve automatisch auch bei der gespiegelten Querschnittskurve übernommen werden.

Dreidimensionale Randkurven, die vertikale Anordnung der Schichten und vor allem die Kombination zusammengesetzter, einzeln hergestellter Segmente erweitern die Möglichkeiten der Produktionsmethode noch einmal wesentlich. Obwohl das formale Repertoire durch die Bedingungen der Herstellungsmethode eingeschränkt ist, können die verschiedensten Objekte in diesem digitalen Workflow entworfen und produziert werden, und das mit den einfachsten, nämlich zweidimensional arbeitenden, digitalen Fabrikationsmaschinen.

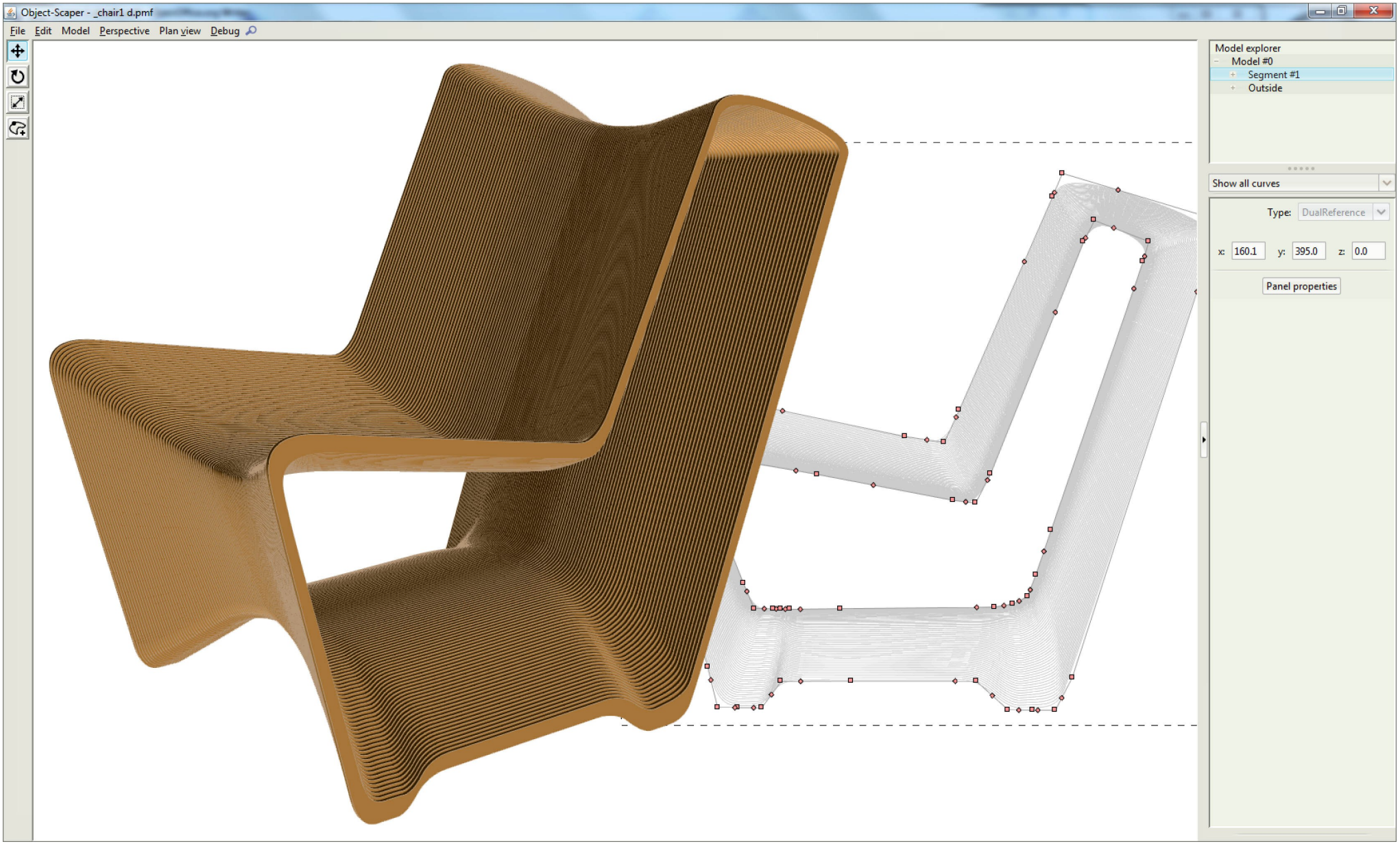


Abbildung 4.14: Sessel mit vertikalen Schichten, aus zwei Segmenten zusammengesetzt

5. Konklusion

Wollte man durch Massenproduktion kosten sparen, musste man identische Objekte produzieren. Das war die Logik der mechanischen, industriellen Herstellungsmethoden. *„In order to offer better and cheaper products to more people, we must offer the same product to all.“*[1] Henry Ford formulierte den fundamentalen Grundsatz der industriellen Produktion: *„Any customer can have a car painted any colour that he wants so long as it is black.“*[2] Die identische Serienproduktion war das ungeschriebene Gesetz der industriellen Revolution.

Die Industrialisierung war geprägt vom Ersatz menschlicher Arbeitskraft durch maschinelle. Heute, im Wandel der industriellen Gesellschaft zur Informationsgesellschaft erleben wir wieder einen Ersatzvorgang: dieses Mal werden die alten mechanischen Maschinen durch neue, digital gesteuerte Maschinen ersetzt. Diese numerisch gesteuerten Fabrikationsmaschinen machen es möglich, dass heute wieder individualisierte Objekte hergestellt werden können, so wie es vor der industriellen Revolution üblich war, als die handwerkliche Fertigung von Einzelstücken der Standard war. Und das ohne die hohen Kosten der handwerklichen Produktion – durchaus auch in einer maschinellen, seriellen Massenproduktion, aber ohne darauf festgelegt zu sein, identische Kopien produzieren zu müssen. Im besten Fall vereinigen digitale Herstellungsverfahren die Vorteile beider Welten, nämlich die Effizienz der maschinellen Fertigung und die Möglichkeit zur Individualisierung in der handwerklichen Produktion.

Der Gegenstand des digitalen, parametrischen Entwerfens ist nicht mehr nur das einzelne konkrete Objekt, sondern der Entwurf eines Systems, einer Serie von differenzierten Objekten. Die einfache Modifikation von Parametern erlaubt es, dass jedes Objekt einer Serie, das nicht mehr gezeichnet sondern berechnet wird, eine individuelle Form erhält. Allerdings bedeutet das bei den meisten Konzepten nur, dass die Objekte gerade einmal so weit variiert werden können, dass sie sich noch als Einzelstücke verkaufen lassen. Mit der Object-Scaping Methode ist dagegen ein Konzept entstanden, das einen Grad der Indeterminiertheit bietet, der es sogar erlaubt, funktionell differenzierte Objekte zu produzieren. Die Herstellungsmethode erweist sich trotz der formalen Einschränkungen, die sich aus einer prozessgerechten Formfindung ergeben, als äußerst vielseitig und flexibel einsetzbar.

Manche algorithmischen Methoden, wie rekursive oder randomisierte Funktionen, erwecken den Eindruck, dass digitale Entwurfsmethoden eher dazu taugen, Komplexität künstlich zu erzeugen. Und natürlich taugt der Computer auch dazu. Aber die wahre Stärke digitaler Methoden liegt in der Reduktion von Komplexität. Auch schwierigste Problemstellungen lassen sich durch die schrittweise Abstraktion innerhalb eines parametrischen Modells bewältigen. Assoziative Algorithmen und selbstorganisierte Objekte ermöglichen und erleichtern das Entwerfen auch mit komplexen Systemen.

Die Kombination von digitalen Methoden in der Gestaltung und in der Produktion kann dabei helfen, die jahrhundertealte Kluft zwischen Entwurf und Ausführung, die seit der Zeit Albertis besteht, zu überbrücken. Durch die numerisch gesteuerten Fabrikationsprozesse können Gestalter wieder einen direkten Bezug zur Materialisierung ihrer Ideen gewinnen und, mit der angemessenen Verwendung digitaler Technologien, eine Welt neuer Möglichkeiten erschließen. Voraussetzung dafür ist allerdings, die traditionellen Prozesse neu zu denken und die Bedingungen der Produktion von Anfang als Grundlage des digitalen Entwurfsprozesses zu akzeptieren.

1 Mario Carpo, *The Demise of the Identical - Architectural Standardization in the Age of Digital Reproducibility*, in: *Refresh - First International Conference on the Histories of Media Art, Science and Technology*, Conference Papers (Banff New Media Institute, http://www.banffcentre.ca/bnmi/programs/archives/2005/refresh/conference_docs.asp, 2005), S. 5

2 Henry Ford, *My Life and Work* (http://www.gutenberg.org/ebooks/7213, 1922), S. 40

Anhang: Sourcecodeauszüge

[SC-01]

```
package generalAlgorithm;

public class QuadBezier2D {

	private Point2D a1, a2, cp;

	public QuadBezier2D(Point2D a1, Point2D cp, Point2D a2) {
		this.a1 = a1;
		this.a2 = a2;
		this.cp = cp;
	}
```

[SC-01a]

```
	/**
	 *
	 * @param t curve parameter 0 <= t <= 1
	 * @return point on the spline
	 */
	public Point2D splinePoint(final double t) {
		final double b0 = ((1 - t) * (1 - t));
		final double b1 = (2 * t * (1 - t));
		final double b2 = (t * t);

		return splinePoint(b0, b1, b2);
	}
```

[SC-01b]

```
	private int curvePoints=10;

	private float b0=new float[curvePoints];
	private float b1=new float[curvePoints];
	private float b2=new float[curvePoints];

	public void calculateBernsteinPolynomials() {

		final float step=1f/(float)(curvePoints.get()-1);
		float t=0;
		for (int i=0; i<curvePoints; i++) {
			b0[i] = ((1 - t) * (1 - t));
			b1[i] = (2 * t * (1 - t));
			b2[i] = (t * t);
			t+=step;
		}
	}
```

[SC-01c]

```
	/**
	 *
	 * @param b0
	 * @param b1
	 * @param b2
	 * @return point on the spline based on the bernstein polynominals b0,
	 *               b1, b2
	 * @see Spline#splinePoint(double)
	 */
	public Point2D splinePoint(final double b0, final double b1,
			final double b2) {
		final double x = a1.x * b0 + cp.x * b1 + a2.x * b2;
		final double y = a1.y * b0 + cp.y * b1 + a2.y * b2;

		return (new Point2D(x, y));
	}
```

[SC-01d]

```
	/**
	 *
	 * Split the spline in two sections according to the De Casteljau
	 *      algorithm
	 * @return an array of two splines
	 */
	public Spline[] halveSpline() {
		Spline[] result = new Spline[2];
		Point2D newCP1 = a1.middle(cp);
		Point2D newCP2 = cp.middle(a2);
		Point2D newAnchor = newCP1.middle(newCP2);

		result[0] = new Spline(a1, newCP1, newAnchor);
		result[1] = new Spline(newAnchor, newCP2, a2);
		return (result);
	}

	/**
	 *
	 * Split the spline at t
	 *
	 * @param t
	 *            curve parameter 0 <= t <= 1
	 * @return an array with the two resulting splines
	 */
	public Spline[] splitSpline(final double t) {
		final Spline[] result = new Spline[2];
```

```
        double x = a1.x + (cp.x - a1.x) * t;
        double y = a1.y + (cp.y - a1.y) * t;
        final Point2D newCP1 = new Point2D(x, y);
        x = cp.x + (a2.x - cp.x) * t;
        y = cp.y + (a2.y - cp.y) * t;
        final Point2D newCP2 = new Point2D(x, y);
        x = newCP1.x + (newCP2.x - newCP1.x) * t;
        y = newCP1.y + (newCP2.y - newCP1.y) * t;
        final Point2D newAnchor = new Point2D(x, y);

        result[0] = new Spline(a1, newCP1, newAnchor);
        result[1] = new Spline(newAnchor, newCP2, a2);
        return (result);
    }

    /**
     *
     * @param t
     *            spline parameter t
     * @return A vector(!) of the tangent in splinePoint(t).
     *         splinePoint(t)+tangent would be a point on the actual
     *         tangent
     */
    public Point2D tangent(double t) {
        final double x = t * (2 * cp.x - a1.x - a2.x) + a1.x - cp.x;
        final double y = t * (2 * cp.y - a1.y - a2.y) + a1.y - cp.y;

        return (new Point2D(x, y));
    }
```

[SC-01e]

```
    private static double PRECISION = 0.001;

    private double closestPoint(final Point2D p, final double startT,
            final double endT, final int iterations) {
        final Spline s = splitSpline(startT, endT);
        final double dTest = PRECISION * 1 / (double) iterations;
        final double dist1 = p.distance(s.getAnchor1());
        final double dist2 = p.distance(s.getAnchor2());
        if (dist1 < dist2) {
            final Point2D testP1 = s.splinePoint(dTest);
            final double distTestP1 = p.distance(testP1);
            if (distTestP1 <= dist1 && iterations > 1) {
                return (closestPoint(p, startT, (startT +
                        endT) / 2, iterations - 1));
            } else {
                return (startT);
            }
        } else {
            final Point2D testP2 = s.splinePoint(1 - dTest);
            final double distTestP2 = p.distance(testP2);
            if (distTestP2 <= dist2 && iterations > 1) {
                return (closestPoint(p, (startT + endT) / 2,
                        endT, iterations - 1));
            } else {
                return (endT);
            }
        }
    }

    /**
     * @return Interpolates points of this spline and returns the length
     *         of the interpolated segments
     * @see GeomMath#B0s
     */
    public double getLength() {
        double lastX = a1.x;
        double lastY = a1.y;
        double x, y, dx, dy;
        double length = 0;
        for (int i = 1; i < GeomMath.B0.length; i++) {
            x = fastIteratePointX(GeomMath.B0[i], GeomMath.B1[i],
                    GeomMath.B2[i]);
            y = fastIteratePointY(GeomMath.B0[i], GeomMath.B1[i],
                    GeomMath.B2[i]);

            dx = x - lastX;
            dy = y - lastY;
            lastX = x;
            lastY = y;
            length += Math.sqrt(dx * dx + dy * dy);
        }
        return (length);
    }

    [...]

}
```

[SC-02]

```
package dataModel;

public class CubicBezier3d implements SplineInterface {

    private Point3d a1, cp1, cp2, a2;

    public CubicBezier3d(Point3d a1, Point3d cp1, Point3d cp2, Point3d a2)
    {
        super();
        this.a1 = a1;
        this.cp1 = cp1;
        this.cp2 = cp2;
        this.a2 = a2;
    }
```

[SC-02a]

```
    public Point3d splinePoint(double t) {
        final double b0 = (1 - t) * (1 - t) * (1 - t);
```

```
		final double b1 = 3 * t * (1 - t) * (1 - t);
		final double b2 = 3 * t * t * (1 - t);
		final double b3 = t * t * t;
		return (splinePoint(b0, b1, b2, b3));
	}

	public Point3d splinePoint(double b0, double b1, double b2, double b3)
{
		final double x = b0 * a1.getX() + b1 * cp1.getX() + b2 *
				cp2.getX() + b3 * a2.getX();
		final double y = b0 * a1.getY() + b1 * cp1.getY() + b2 *
				cp2.getY() + b3 * a2.getY();
		final double z = b0 * a1.getZ() + b1 * cp1.getZ() + b2 *
				cp2.getZ() + b3 * a2.getZ();
		return (new Point3d(x, y, z));
	}
```

[SC-02b]

```
	public Point3d tangent(double t) {
		final Point3d m = a1.between(cp1, t);
		final Point3d n = cp1.between(cp2, t);
		final Point3d o = cp2.between(a2, t);
		final Point3d t1 = m.between(n, t);
		final Point3d t2 = n.between(o, t);
		return (t2.subtract(t1));
	}
```

[SC-02c]

```
	/**
	 * Calculate the roots of a cubic function using Cardano's method. See
	 * Dietlinde Lau, Algebra und Diskrete Mathematik 1, 2. Auflage
	 * (Springer Verlag, Berlin, 2007), S. 265ff,
	 * http://de.wikipedia.org/wiki/Cardanische_Formeln,
	 * http://www.java-forum.org/mathematik/67531-kubische-funktion-
	 * extremwerte.html
	 *
	 * @return the roots of the cubic polynom a * x^3 + b * x^2 + c * x
	 *                                                                + d
	 */
	public static double[] cubicRoots(double a, double b, double c,
	                                  double d) {
		double[] roots;
		double oldA = a;

		a = b / oldA;
		b = c / oldA;
		c = d / oldA;

		double p = b - a * a / 3;
		double q = 2 * a * a * a / 27 - a * b / 3 + c;

		double D = q * q / 4 + p * p * p / 27;

		double u = Math.pow(-q / 2 + Math.pow(D, 1 / 2.0), 1 / 3);
		double v = Math.pow(-q / 2 - Math.pow(D, 1 / 2.0), 1 / 3);

		if (D > 0) {
			// eine lösung!
			roots = new double[1];

			double z = u + v;
			double x1 = z - a / 3;

			roots[0] = x1;

			return roots;
		}

		if (D == 0) {
			if (p == 0 && q == 0) {
				// eine lösung: determinante = 0, p = 0, q = 0
				// -> nullpunkt bei 0

				roots = new double[1];
				roots[0] = 0;
				return roots;
			}

			// zwei lösungen;
			roots = new double[2];

			double z1 = 3 * q / p;
			double z2 = -3 * q / 2 / p;

			double x1 = z1 - a / 3;
			double x2 = z2 - a / 3;

			roots[0] = x1;
			roots[1] = x2;

			return roots;
		}

		// System.out.println("drei lösungen!");
		roots = new double[3];

		double z2 = -Math.sqrt(-4 / 3 * p)
				* Math.cos(1 / 3
						* Math.acos(-q / 2 * Math.sqrt(-
						27 / p / p / p)) + Math.PI / 3);
		double z1 = Math.sqrt(-4 / 3 * p)
				* Math.cos(1 / 3 * Math.acos(-q / 2
						* Math.sqrt(-27 / p / p / p)));
		double z3 = -Math.sqrt(-4 / 3 * p)
				* Math.cos(1 / 3
						* Math.acos(-q / 2 * Math.sqrt(-
						27 / p / p / p)) - Math.PI / 3);

		double x1 = z1 - a / 3;
		double x2 = z2 - a / 3;
		double x3 = z3 - a / 3;

		roots[0] = x1;
		roots[1] = x2;
```

```
                roots[2] = x3;

                return roots;
        }

        [...]

}
```

[SC-03]

```
package dataModelPoints;

public class ModelSymControl extends ModelAbstract {

        //the point that contains the r, s, t values
        private Point3d refPoint;
        //the base control point on the closest section
        private Point3d base;
        //the control point on the opposite section
        private Point3d opposite;

        //this point's actual coords
        private double x, y, z;

        /**
         *
         * @param refPoint the reference point that stores r, s, t coordinates
         * @param base the point on the opposite section corresponding to base
         * @param opposite the reference point that stores r, s, t coordinates
         */
        public ModelSymControl(Point3d refPoint, Point3d base,
                               Point3d opposite) {

                this.refPoint = refPoint;
                this.base = base;
                this.opposite = opposite;

                recalc();
        }

        private void recalc() {
                final double vecX=opposite.getX()-base.getX();
                final double vecY=opposite.getY()-base.getY();

                double newX=base.getX()+vecX*refPoint.getX();
                double newY=base.getY()+vecY*refPoint.getX();

                final double nVecX=vecY;
                final double nVecY=-vecX;

                newX+=nVecX*refPoint.getY();
                newY+=nVecY*refPoint.getY();

                final double vecZ=opposite.getZ()-base.getZ();
                final double newZ=base.getZ()+vecZ*refPoint.getZ();

                //now done in ChangeListener!

                /*int what=0;
                if (getX()!=newX) what+=ChangeListener.POINT3D_X;
                if (getY()!=newY) what+=ChangeListener.POINT3D_Y;
                if (getZ()!=newZ) what+=ChangeListener.POINT3D_Z;

                if (what!=0) {
                        callListeners(what);
                }*/

                x=newX;
                y=newY;
                z=newZ;
        }

        @Override
        public void set(double x, double y, double z) {
                final double vecX=opposite.getX()-base.getX();
                final double vecY=opposite.getY()-base.getY();
                final double nVecX=vecY;
                final double nVecY=-vecX;

                final double r=-(-base.getY() * nVecX + base.getX() * nVecY -
                        nVecY * x + nVecX * y) / (nVecY * vecX - nVecX * vecY);
                final double s=-(base.getX() + r * vecX - x) / nVecX;

                final double vecZ=opposite.getZ()-base.getZ();
                final double t=(z-base.getZ())/vecZ;

                refPoint.set(r,s,t);

                this.x=x;
                this.y=y;
                this.z=z;

        }

        [...]

}
```

[SC-04]

```
package dataModelPoints;

public class Point3d  {

        private Point3dModel model = null;
        public List<ChangeListener> listeners = null;

        public ChangeListener modelChangeListener = new ChangeListener() {

                public void modelChange(Object source, int what) {
                        callListeners(what);
                }

                public String toString() {
                        return "dataModelPoints.Point3d.modelChangeListener@"
                                        + Integer.toHexString(hashCode());
```

```
        }
};

public Point3d() {
        setPointModel(new ModelDefault());
}

public Point3d(double x, double y, double z) {
        setPointModel(new ModelDefault(x, y, z));
}

public Point3d(Point3dModel model) {
        setPointModel(model);
}

public Point3d(double x, double y, double z, Point3dModel model) {
        model.set(x, y, z);
        setPointModel(model);
}

public Point3d(Point3d p) {
        setPointModel(p.getPointModel());
}

public double getX() {
        return model.getX();
}

public double getY() {
        return model.getY();
}

public double getZ() {
        return model.getZ();
}

public void set(Point3d p) {
        set(p.getX(), p.getY(), p.getZ());
}

public void set(double x, double y, double z) {
        model.set(x, y, z);
}

protected void callListeners(int what) {
        if (listeners != null) {
                for (ChangeListener l : listeners) {
                        l.modelChange(this, what);
                }
        }
}

public void addChangeListener(ChangeListener l) {

        if (listeners == null) {
                listeners = new ArrayList<ChangeListener>(3);
                listeners.add(l);
        } else if (!listeners.contains(l)) {
                if (listeners.size() == 0) {
                        // restoring the point -> force the point model
                        // to reattach
                        // connections to the rest of the DataModel
                        model.addChangeListener(modelChangeListener);
                }
                listeners.add(l);
        }
}

public void removeChangeListener(ChangeListener l) {
        if (listeners != null) {
                listeners.remove(l);
                if (listeners.size() == 0) {
                        // no more connections to this point -> allow
                        // the point model to
                        // remove all connections to the rest of the
                        // DataModel
                        model.removeChangeListener(modelChangeListener);
                }
        }
}

public void setPointModel(Point3dModel m) {
        if (this.model != null) {
                this.model.removeChangeListener(modelChangeListener);
        }
        this.model = m;
        this.model.addChangeListener(modelChangeListener);
        callListeners(ChangeListener.POINT_MODEL_CHANGE);
}

public Point3dModel getPointModel() {
        return model;
}

final public double distance(final Point3d p) {
        final double dx = getX() - p.getX();
        final double dy = getY() - p.getY();
        final double dz = getZ() - p.getZ();
        return (Math.sqrt(dx * dx + dy * dy + dz * dz));
}

final public double length() {
        return (Math.sqrt(getX() * getX() + getY() * getY() + getZ()
                        * getZ()));
}

final public Point3d add(final Point3d p) {
        return (new Point3d(getX() + p.getX(), getY() + p.getY(),
                        getZ() + p.getZ()));
}

final public Point3d subtract(final Point3d p) {
        return (new Point3d(getX() - p.getX(), getY() - p.getY(),
                        getZ() - p.getZ()));
}
```

```
	final public Point3d multiply(final double m) {
		return (new Point3d(getX() * m, getY() * m, getZ() * m));
	}

	final public Point3d middle(final Point3d p) {
		return (new Point3d((getX() + p.getX()) / 2,
				(getY() + p.getY()) / 2,
				(getZ() + p.getZ()) / 2));
	}

	final public Point3d between(final Point3d p, final double t) {
		final double x = this.getX() + (p.getX() - this.getX()) * t;
		final double y = this.getY() + (p.getY() - this.getY()) * t;
		final double z = this.getZ() + (p.getZ() - this.getZ()) * t;
		return (new Point3d(x, y, z));
	}

	public Point2D to2D() {
		return (new Point2D(getX(), getY()));
	}

	public Point3d copy() {
		final Point3d p = new Point3d(getX(), getY(), getZ());
		p.setName(getName());
		return (p);
	}

	public boolean equals(Object o) {
		if (o instanceof Point3d) {
			Point3d p = (Point3d) o;
			return (this.getX() == p.getX() &&
					this.getY() == p.getY() &&
					this.getZ() == p.getZ());
		}
		return (false);
	}

	[...]

}
```

[SC-05]

```
package dataModelPoints;

/**
 * Every Point3d has a model that determines how it's coordinates are
 * calculated. This makes it easier to create
 * points that depend on other point's positions etc.
 * @author mm
 *
 */
public interface Point3dModel {

	/**
	 * @return the x-coordinate of this point
	 */
	public double getX();

	/**
	 * @return the y-coordinate of this point
	 */
	public double getY();

	/**
	 * @return the z-coordinate of this point
	 */
	public double getZ();

	/**
	 * set this point's coordinates
	 */
	public void set(double x, double y, double z);

	/**
	 * Register a <code>ChangeListener</code> with this Point3dModel.
	 * Implementations must ensure that all
	 * ChangeListeners registered with this method are notified whenever
	 * this model's coordinates change, and
	 * that connections to the rest of the DataModel are restored when
	 * adding a <code>ChangeListener</code>
	 * after connections were cut by removing the last
	 * <code>ChangeListener</code> with
	 * <code>removeChangeListener()</code>.
	 * @see #removeChangeListener(ChangeListener)
	 * @param l the <code>ChangeListener</code> to add
	 */
	public void addChangeListener(ChangeListener l);

	/**
	 * Unregister a <code>ChangListerner</code>. Implementations must
	 * ensure that all connections to other
	 * points of the DataModel are cut when the last
	 * <code>ChangeListener</code> is removed.
	 * @param l the <code>ChangeListener</code> to remove
	 */
	public void removeChangeListener(ChangeListener l);
}
```

[SC-06]

```
package dataModelPoints;

public abstract class ModelAbstract  {

	private List<ChangeListener> listeners = null;

	public abstract double getX();

	public abstract double getY();

	public abstract double getZ();

	public abstract void set(double x, double y, double z);

	/**
```

```
	 * remove all connections (<code>ChangeListener</code>s) to other
	 * points
	 */
	public abstract void remove();

	/**
	 * undo remove()
	 */
	public abstract void restore();

	protected void callListeners(int what) {
		if (listeners != null) {
			for (ChangeListener l : listeners) {
				l.modelChange(this, what);
			}
		}
	}

	public void addChangeListener(ChangeListener l) {
		if (listeners == null) {
			listeners = new ArrayList<ChangeListener>(3);
			listeners.add(l);
		} else if (!listeners.contains(l)) {
			if (listeners.size() == 0) {
				restore();
			}
			listeners.add(l);
		}
	}

	public void removeChangeListener(ChangeListener l) {
		if (listeners != null) {
			listeners.remove(l);
			if (listeners.size() == 0) {
				remove();
			}
		}
	}
}
```

[SC-07]

```
package dataModelPoints;

public class ModelDefault extends ModelAbstract {

	private double x = 0, y = 0, z = 0;

	public ModelDefault() {
	}

	public ModelDefault(double x, double y, double z) {
		this.x = x;
		this.y = y;
		this.z = z;
	}

	public double getX() {
		return x;
	}

	public double getY() {
		return y;
	}

	public double getZ() {
		return z;
	}

	public void set(double x, double y, double z) {
		int what = 0;
		if (this.x != x)
			what += ChangeListener.POINT3D_X;
		if (this.y != y)
			what += ChangeListener.POINT3D_Y;
		if (this.z != z)
			what += ChangeListener.POINT3D_Z;

		if (what != 0) {
			this.x = x;
			this.y = y;
			this.z = z;
			callListeners(what);
		}
	}

	public void remove() {
	}

	public void restore() {
	}

}
```

[SC-08]

```
package dataModelPoints;

public class ModelReference extends ModelAbstract {

	private Point3d ref;
	private Point3d vec;

	private ChangeListener refPointChangeListener = new ChangeListener() {
		private static final long serialVersionUID = 1L;

		public void modelChange(Object source, int what) {
			callListeners(what);
		}
	};

	public ModelReference(double x, double y, double z, Point3d ref) {
		setRef(ref);
		setVec(new Point3d());
		set(x, y, z);
	}
```

```
	private void setRef(Point3d ref) {
		if (this.ref != null) {
			this.ref.removeChangeListener(refPointChangeListener);
		}
		this.ref = ref;
		this.ref.addChangeListener(refPointChangeListener);
	}

	private void setVec(Point3d vec) {
		if (this.vec != null) {
			this.vec.removeChangeListener(refPointChangeListener);
		}
		this.vec = vec;
		this.vec.addChangeListener(refPointChangeListener);
	}

	@Override
	public double getX() {
		return ref.getX() + vec.getX();
	}

	@Override
	public double getY() {
		return ref.getY() + vec.getY();
	}

	@Override
	public double getZ() {
		return ref.getZ() + vec.getZ();
	}

	@Override
	public void set(double x, double y, double z) {
		vec.set(x - ref.getX(), y - ref.getY(), z - ref.getZ());
	}

	/**
	 * @return the vector from this model's reference point to this
	 *         model's actual coordinates
	 */
	public Point3d getVec() {
		return vec;
	}

	/**
	 * @return the original point this model references
	 */
	public Point3d getReferencedPoint() {
		return ref;
	}

	@Override
	public void remove() {
		ref.removeChangeListener(refPointChangeListener);
	}

	@Override
	public void restore() {
		ref.addChangeListener(refPointChangeListener);
	}

}
```

[SC-09]

```
package dataModelPoints;

public class ModelDualReference extends ModelAbstract {

	private Point3d refA = null, refB = null;
	private double t = 0;

	private ChangeListener refPointChangeListener = new ChangeListener() {
		private static final long serialVersionUID = 1L;

		public void modelChange(Object source, int what) {
			callListeners(what);
		}
	};

	/**
	 * Create a Point3dModel that is moved whenever it's refence points
	 * are moved. This point is positioned between <code>refA</code> and
	 * <code>refB</code>.
	 *
	 * @param refA
	 *            first reference point
	 * @param refB
	 *            second reference point
	 * @param t
	 *            position of this point between <code>refA</code> and
	 *            <code>refB</code> so that this point will be placed at
	 *            <code>refA + (refB - refA) * t</code>
	 */
	public ModelDualReference(Point3d refA, Point3d refB, double t) {
		setRefA(refA);
		setRefB(refB);
		setT(t);
	}

	public ModelDualReference(ModelDualReference r) {
		setRefA(r.getRefA());
		setRefB(r.getRefB());
		setT(r.getT());
	}

	public double getX() {
		return (refA.getX() + (refB.getX() - refA.getX()) * t);
	}

	public double getY() {
		return (refA.getY() + (refB.getY() - refA.getY()) * t);
	}

	public double getZ() {
```

```
        return (refA.getZ() + (refB.getZ() - refA.getZ()) * t);
}

public void set(double x, double y, double z) {
        final double aX = refA.getX();
        final double aY = refA.getY();
        final double aZ = refA.getZ();
        final double vX = refB.getX() - aX;
        final double vY = refB.getY() - aY;
        final double vZ = refB.getZ() - aZ;

        double t = -(aX * vX - x * vX + aY * vY - y * vY + aZ * vZ - z
                        * vZ) / (vX * vX + vY * vY + vZ * vZ);

        if (t < 0)
                t = 0;
        if (t > 1)
                t = 1;

        setT(t);
}

public Point3d getRefA() {
        return refA;
}

public void setRefA(Point3d refA) {
        if (this.refA != null) {
                this.refA.removeChangeListener(refPointChangeListener);
        }
        this.refA = refA;
        this.refA.addChangeListener(refPointChangeListener);
}

public Point3d getRefB() {
        return refB;
}

public void setRefB(Point3d refB) {
        if (this.refB != null) {
                this.refB.removeChangeListener(refPointChangeListener);
        }
        this.refB = refB;
        this.refB.addChangeListener(refPointChangeListener);
}

public double getT() {
        return t;
}

public void setT(double t) {
        if (this.t != t) {
                final double oldX = getX();
                final double oldY = getY();
                final double oldZ = getZ();

                this.t = t;

                int what = 0;
                if (getX() != oldX)
                        what += ChangeListener.POINT3D_X;
                if (getY() != oldY)
                        what += ChangeListener.POINT3D_Y;
                if (getZ() != oldZ)
                        what += ChangeListener.POINT3D_Z;

                if (what != 0) {
                        callListeners(what);
                }
        }
}

public void remove() {
        refA.removeChangeListener(refPointChangeListener);
        refB.removeChangeListener(refPointChangeListener);
}

public void restore() {
        refA.addChangeListener(refPointChangeListener);
        refB.addChangeListener(refPointChangeListener);
}

}
```

Literaturverzeichnis

Beaucé, Patrick / Cache, Bernard, Towards a Non-standard Mode of Production, in: Bernard Leupen, René Heijne, Jasper van Zwol (Hrsg.), Time-based Architecture (010 Publishers, Rotterdam, 2005)

Bonwetsch, Tobias / Bärtschi, Ralph / Kobel, Daniel / Gramazio, Fabio / Kohler, Matthias, Digitally Fabricating Tilted Holes. Experiences in Tooling and Teaching Design, in: eCAADe 25 Conference Proceedings (http://cumincad.scix.net, 2007)

Brezinski, Claude / Wuytack, Luc, Numerical analysis in the twentieth century, in: Claude Brezinski, Luc Wuytack (Hrsg.), Numerical analysis: historical developments in the 20th century (Elsevier Science B. V., Amsterdam, 2001)

Carpo, Mario, Das Digitale, „Mouvance“ und das Ende der Geschichte, in: Urs Hirschberg, Daniel Gethmann, Ingrid Böck, Harald Kloft (Hrsg.), Graz Architektur Magazin 06 (Springer Verlag, Wien, 2010)

Carpo, Mario, The Demise of the Identical - Architectural Standardization in the Age of Digital Reproducibility, in: Refresh - First International Conference on the Histories of Media Art, Science and Technology, Conference Papers (Banff New Media Institute, http://www.banffcentre.ca/bnmi/programs/archives/2005/refresh/conference_docs.asp, 2005)

Farin , Gerald E., A History of Curves and Surfaces in CAGD, in: Gerald E. Farin, Josef Hoschek, Myung-Soo Kim (Hrsg.), Handbook of computer aided geometric design (Elsevier Science B.V., Amsterdam, 2002)

Greanier, Todd, Java foundations (Sybex, Alameda, 2004)

Kloft, Harald, Logik oder Form, in: Urs Hirschberg, Daniel Gethmann, Ingrid Böck, Harald Kloft (Hrsg.), Graz Architektur Magazin 06 (Springer Verlag, Wien, 2010)

Lau, Dietlinde, Algebra und Diskrete Mathematik 1, 2. Auflage (Springer Verlag, Berlin, 2007)

Leach, Neil, Digital Morphogenesis, in: Architectural Design, Theoratical Meltdown, Vol. 79, Issue 1 (Wiley & Sons, London, 2009)

Mitchell, William J., Roll Over Euclid: How Frank Gehry Designs and Builds, in: J. Fiona Ragheb (Hrsg.), Frank Gehry, Architect (Guggenheim Museum Publications, New York, 2001)

Moe, Kiel, Automation Takes Command, in: Robert Corser (Hrsg.), Fabricating Architecture: Selected Readings in Digital Design and Manufacturing (Princeton Architectural Press, New York, 2010)

Moe, Kiel, The Non-Standard, Un-Automatic Prehistory of Standardization and Automation in Architecture, in: Proceedings of the 2007 ACSA National Conference (Associations of Collegiate Schools of Architecture, Washington, 2007)

Prautzsch, Hartmut / Boehm, Wolfgang / Paluszny, Marco, Bézier and B-spline techniques (Springer Verlag, Berlin, 2002)

Reas, Casey / Williams, Chandler, Form + Code in Design, Art and Architecture (Princeton Architectural Press, New York, 2010)

Rocker, Ingeborg M., Architectures of the Digital Realm: Experimentations by Peter Eisenman / Frank O. Gehry, in: Jörg H. Gleiter, Norbert Korrek, Gerd Zimmermann (Hrsg.), Die Realität des Imaginären - Architektur und das digitale Bild (Universitätsverlag der Bauhaus-Universität, Weimar, 2008)

Schindler, Christoph, Die Standards des Nonstandards, in: Urs Hirschberg, Daniel Gethmann, Ingrid Böck, Harald Kloft (Hrsg.), Graz Architektur Magazin 06 (Springer Verlag, Wien, 2010)

Weisberg, David E., The Engineering Design Revolution (Ebook, Online unter http://www.cadhistory.net/, 2008)

Abbildungsverzeichnis