Analyse und Implementierung verschiedener Sliceralgorithmen für den 3D-Druck

Schweizer Jugend forscht 2016

Autor:
Kevin De Keyser

Betreuer SJf:
Daniel Eisenbarth

Betreuer Maturaarbeit:
Christoph Wildfeuer

Abstract

Kurzfassung: Die Arbeit testet die Ideen effizienter Sliceralgorithmen und diskutiert dessen Probleme. Es vergleicht die asymptotische Laufzeit mit anderen langsameren Ansätzen und bietet generell Einblicke in den 3D-Druck.

Beim 3D-Druck muss zuerst ein digitales 3-dimensionales Objekt (hier STL) in Maschinencode (hier G-Code) umgewandelt werden. Der Maschinencode enthält Informationen für den 3D-Drucker, etwa wie sich der Druckkopf bewegen muss, die Bodenplatte erhitzt werden sollte, die Fördergeschwindigkeit eingestellt werden muss, etc. Für diese Umwandlung wird eine Slicersoftware verwendet. 'Slicing' ist dabei der Prozess ein Modell in druckbare Stücke zu schneiden, zuerst in übereinander liegende Polygone, welche danach wiederum in Segmente geschnitten werden. Solche Segmente kann man dann mit einem 3D-Drucker ausdrucken. In aktuellen Slicern geht dieses 'Slicen' ziemlich lange, vor allem bei grösseren Objekten kann das Slicen schnell einmal 5-10 Minuten dauern. Diese Arbeit implementiert einen Slicer in C++98 und entwickelt Methoden die Dauer des Slicen erheblich zu verkürzen.

Die Arbeit ist dabei chronologisch strukturiert, zuerst werden die jeweiligen Konzepte besprochen, darauf folgen jeweils die Implementierungen, die Fehlerbehebungen und schliesslich die Optimierungen. Auch der Prozess wie der Sliceralgorithmus zu Stande gekommen ist und welche physikalischen Probleme man beachten muss ist in der Arbeit notiert. Am Ende der Arbeit ist der Source-Code des Projekts aufgelistet um die Resultate zu reproduzieren. Diese Implementation wurde auch mit industriegängigen Slicer auf Geschwindigkeit unter verschiedenster Bedingungen verglichen, wobei der vorgeschlagene Slicer besonders schnell abschnitt.

Motivation

Über mein Interesse, mittels Algorithmen komplexe Aufgaben zu lösen, fand ich die Freude an der Software-Entwicklung. Dieses Hobby wurde zu meiner Leidenschaft, weswegen ich ein Grossteil meiner Freizeit mit Weiterbildung verbringe. 2014 und 2015 nahm ich an der Schweizer Informatikolympiade teil, wo ich jeweils unter den ersten 10 in die Ränge kam und habe 2015 ein Buch über theoretische Informatik veröffentlicht.
Für die anstehende Maturaarbeit wollte ich mehr über die praktischen Anwendungen der Mathematik und der Informatik lernen. Zur Zeit der Themenwahl kaufte die Kantonsschule Sursee einen 3D-Drucker. Fasziniert von den Objekten, welche der 3D-Drucker scheinbar aus der Luft formte, fragte ich Dr. Christoph Wildfeuer ob ich meine anstehende Maturaarbeit in diesem Bereich machen durfte. Dank ihm ergab sich für mich die Möglichkeit meine Softwarekenntnisse mit der physikalischen / praktischen Welt zu verknüpfen.
Der 3D-Drucker verfügte über einen Slicer (Drucksoftware): Cura, welcher die Bewegungen des Druckkopfes entwickelte. Für die Maturaarbeit habe ich mir dann vorgenommen einen eigenen Sliceralgorithmus zu entwickeln und diesen zu analysieren.
Zuerst habe ich mich mit den Komponenten des 3D-Druckers und den Protokollen (Dateien) des Slicers vertraut gemacht. Dabei sind mir Sliceralgorithmen eingefallen, die ich bewusst analysieren und implementieren wollte. In dieser Dokumentation ist die Vorgehensweise der Implementierung chronologisch beschrieben. So steht auch welche Komplikationen passierten und welche Optimierungen gemacht werden mussten, um das Programm funktionsfähig zu machen. Dabei wurde die Dokumentation sehr bildreich. Ebenfalls habe ich noch andere Vorgehensweisen theoretisch analysiert, die mir im Laufe der Arbeit eingefallen sind.
Ich habe sehr viel Erfahrung gemacht im Software/Hardware Bereich. So mussten zum Beispiel verschiedene Programme entwickelt/benutzt werden um den 3D-Drucker zu simulieren und um Fehler schnell testen zu können, da ein Druck mehrere Stunden in Anspruch nahm. Auch in anderen Bereichen habe ich viele Erfahrungen gemacht, so auch im Rapid-Prototyping. Verschiedene Objekte mussten modelliert, konvertiert und gedruckt werden um den Slicer zu testen. Dabei konnte ich mir neue Fähigkeiten aneignen, ich habe zum Beispiel ein Brettspiel modelliert und kreiert.

Heute läuft das Programm so weit fehlerfrei und ich bin sehr zufrieden mit der geleisteten Arbeit.

Danksagung

Diese Arbeit wäre ohne die folgenden Beteiligten nicht möglich gewesen und ich möchte mich bei Ihnen auch schriftlich bedanken.
Schweizer Jugend forscht: Gäbe es diesen Wettbewerb nicht, würde diese Arbeit irgendwo in einem Schubladenfach verstauben. Ihr habt mich motiviert weiter an der Arbeit zu tüfteln und bin dadurch auch auf völlig neue Ideen gestossen.
Daniel Eisenbarth: Ihnen möchte ich besonders danken, da sie mir die notwendigen Tipps geben konnten, meine Arbeit besser zu vermitteln. Sie konnten mich immer motivieren, an Teile der Arbeit zu feilen und konnten mir Problemstellungen zeigen an denen sie arbeiteten, was mich sehr interessierte. Ich bin sehr dankbar, dass sie sich die Zeit genommen haben mich zu betreuen und hoffe, dass wir weiterhin in Kontakt bleiben.
Kantonsschule Sursee: Ich bin sehr dankbar, dass ich mein Maturajahr an der Kantonsschule Sursee absolvieren konnte. Dank dieser Schule bekam ich die Möglichkeit eine Arbeit in dem Gebiet des 3D-Drucks zu machen, was schliesslich dazu geführt hat, dass diese Arbeit überhaupt existiert.
Hr. Dr. Christoph Wildfeuer: Zuerst möchte ich Ihnen danken für ihre wichtige Rolle als Betreuer. Ich hatte immer die Möglichkeit Emails mit Ihnen auszutauschen und Fragen bezüglich der Arbeit, aber auch bezüglich formellen und Privatinteressen. Sie konnten mich immer wieder für Neues begeistern, unter anderem für den 3D-Drucker. Ich bin sehr dankbar, Sie kennen zu dürfen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Code - Ausschnitte

0 Geschichte des 3D-Drucks

Charles (Chuck) W. Hull (geboren 12. Mai 1939) ist die Vaterfigur der Stereolithographie[1], welche er 1986 erfand. Die Stereolithographie ist die herkömmliche Art und Weise ein 3-dimensionales Objekt zu drucken, bei welcher schichtenweise ein Material aufgetragen wird. Schon früher wurde mit CNC (Computerized Numerical Control) Maschinen digitale Objekte ins Leben gerufen, dort wird jedoch z.B. ein Plastikwürfel in ein Objekt umgewandelt, in dem, mithilfe einer Strahltechnik, um das Objekt herum geschnitten wird. Mit solchen Schnittmaschinen kann man jedoch keine Hohlräume erzeugen und es geht viel Restmaterial bei der Beschneidung verloren. Hull hat dieses Marktlücke entdeckt und hat sein ursprüngliches Verfahren patentieren lassen (EP0171069). In seinem Patent beschreibt er einen Prozess bei dem eine mit Photopolymer beschichtete Platte von UV-Licht bestrahlt wird. Das Photopolymer verhärtet (polymerisiert) sich auf der Stelle, um so den Schichtendruck zu ermöglichen.

2014 gewann Hull den 'European Inventor Award' in der 'Non-European countries' Kategorie[2] für seine Erfindung.

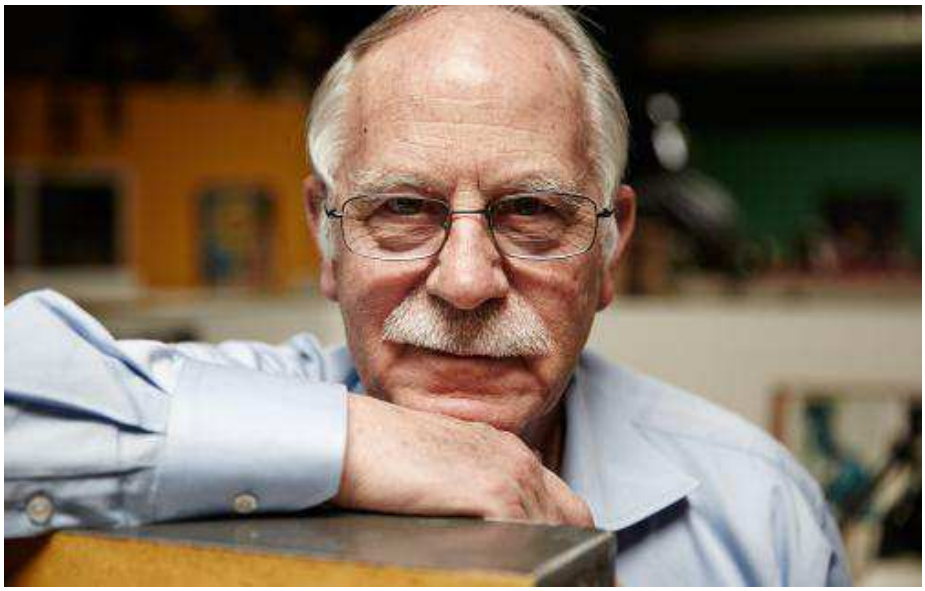


Abbildung 1: Charles (Chuck) W. Hull[12]

Heute werden ganz verschiedene Druckmaterialien verwendet, am häufigsten Plastik (PLA, ABS), Metall und Sandstein. Auch mit menschlichem Gewebe und Schokolade wird heute gedruckt. Diese Arbeit beschäftigt sich mit Plastikdruckern. 3D-Drucker werden heute nicht nur in der Industrie genützt, sondern auch an ein paar anderen Orten:
In der Medizin werden sie verwendet um Gipsgebisse, Knochentransplantate, billige bionische Arme und sogar Organe zu drucken.
In der Mathematik kann man erstmals komplizierte Fraktale, Hilbertskurven und Kleinsche Flaschen ausdrucken.
Die Firma YHBM in Shanghai druckt heute schon sehr billige, verwendbare

Häuser. Es ist wirklich faszinierend wie einfach es ist einen 3D-Drucker aufzubauen, wenn sogar Häuser damit gedruckt werden.
Meiner Meinung nach ist der faszinierendste Bereich jedoch die Hobbymodellierung, denn man kann jegliche Figur bequem von zu Hause modellieren und ausdrucken. 3D-Drucker sind schon heute kommerziell erhältlich und die Slicer gratis verfügbar. In der Zukunft denke ich, gewinnt der 3D-Druck immer mehr an Bedeutung. So werden Resin-Printer schneller und billiger, während die Software immer schneller und einfacher zu bedienen wird. Wenn bis dahin nicht alles virtuell abläuft, denke ich werden in 10 Jahren die meisten einen solchen 3D-Drucker besitzen.

1 Einführung in den 3D-Druck

1.1 Der Slicer

Eine Slicersoftware wandelt digitale 3-dimensionale Objekte (CAD) in Maschinencode um, welcher angibt wie sich der Druckerkopf bewegen muss um ein Objekt auszudrucken. Bei dem praktischen Teil dieser Arbeit verwende ich die industriegängigen Formate, STL (Surface Tesselation Language) als digitaler Input der 3D-Objekte und G-code (oder RS-274) als Maschinencode. In dieser Arbeit werden verschiedene Algorithmen vorgestellt um diesen zu erzeugen. Praktisch implementiert sind diese in C++98 und mit der Cross-Plattform Library OpenFrameworks[11].

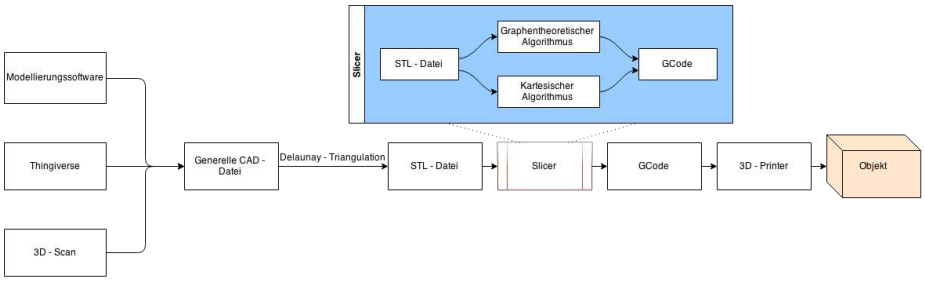


Abbildung 2: Slicer Flowchart

1.2 STL-Format

Es gibt zwei Typen von STL-Dateien, die binäre STL Datei und die ASCII STL Datei. Die ASCII Version ist nicht so kompakt wie die binäre Datei und ist lesbar von blossem Auge. Das STL-Format wird anhand der ASCII-, der Klartextversion vorgestellt.
STL Dateien enthalten folgende Informationen: Name des Objekts, Dreieckskoordinaten und den Normalenvektor der Dreiecke. Das ganze wird sehr ersichtlich mit der Syntax der ASCII-STL Datei. Um ein 3-dimensionales Objekt darzustellen braucht man nur Dreiecke, welche aneinander gereiht werden. Runde Oberflächen sind nur durch Annäherungen erreichbar, so kann man aus einer Pyramide ein Viereck machen, aus diesem ein Dodecahedron und so weiter, bis man näherungsweise eine Kugel erreicht hat.
Um ein beliebiges Polyhedron in Dreiecke zu unterteilen muss man für jeder dessen Fläche eine Delaunay-Triangulation[3][4] durchführen. Dieses Verfahren ist nicht Teil dieser Arbeit.

1.2.1 ASCII-STL Syntax

```
solid[Name]
```

So beginnt jede ASCII-STL Datei, mit dem Begriff *solid* und dem Namen.
Danach folgen N Dreiecke. Diese Dreiecke werden alle mit demselben Syntax beschrieben:

```
facet normal nX nY nZ
outer loop
  vertex pX1 pY1 pZ1
  vertex pX2 pY2 pZ2
  vertex pX3 pY3 pZ3
endloop
endfacet
```

Zuerst wird mit dem Begriff *facet* angekündigt, dass ein neues Dreieck beschrieben wird. Nach *normal* folgen drei 32-Bit Floats (Gleitkommazahlen), den Normalenvektor des Dreiecks. Den Normalenvektor kann man auch selber berechnen und wird eigentlich bei dieser Arbeit nicht benötigt:

```
nX = (pY1 * pZ2) - (pY2 * pZ1)
nY = (pZ1 * pX2) - (pZ2 * pX1)
nZ = (pX1 * pY2) - (pX2 * pY1)
```

Zwischen *outer loop* und *endloop* folgen die drei Koordinaten des Dreiecks. Jedes Dreieck hat 3 Eckpunkte, die mit dem englischen Begriff *vertex* bezeichnet werden. Darauf folgen jeweils wieder drei 32-Bit Floats (Gleitkommazahlen), die entsprechenden X, Y und Z-Koordinaten des Objekts. Dabei einigt man sich nicht auf eine Masseinheit, sondern man beschränkt sich auf relative Koordinaten, die wir später beim ausdrucken sowieso skalieren können müssen. Die Datei endet mit *endsolid*.

```
endsolid [Name]
```

1.3 G-Code

G-Codes[5] werden zur Lenkung einer Maschine verwendet, in dieser Arbeit also des Druckkopfes. Diese Arbeit haltet sich an die G-Code Definitionen des MakerBot Druckers. Der resultierende G-Code sollte jedoch universell funktionieren.
Der G-Code besteht aus M-Codes und G-Codes, welche einfache Low-Level Funktionen sind.

```
[G/M][N] [param1] [param2] [param3]... [paramN]
```

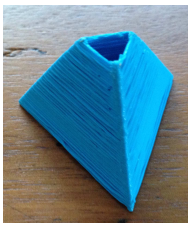
Zwei Beispiele:

```
M106 S0
G0 X20 Y20 Z0.3
```

Der G-Code lässt sich in 3 Schichten unterteilen. Es gibt den Initialisierungscode, den Druckcode und den Endcode. Der Initialisierungscode und der Endcode sind bei jeder Anwendung gleich und stellen sicher, dass der Druckkopf am richtigen Ort ist, der Lüfter eingeschaltet wird, die Bodenplatte richtig erhitzt wird, etc.
M-Codes stehen für 'miscellaneous codes'. Meist handelt es sich hier um maschinenspezifische Kommandos, sowie den Lüfter ein- und auszuschalten. Sie sind eigentlich nur im Initialisierungscode und im Endcode vorhanden. Der Slicer verwendet die Folgenden:

```
M25 ; Pausiert das SD drucken
M106 ST ; Der Lüfter wird mit T PMW verwendet
```

(a) Gedrucktes Objekt ohne Lüfter



(b) Gedrucktes Objekt mit Lüfter

Abbildung 3: Miscellaneous Codes in Aktion

Diese M-Codes zu optimieren stellt sich als recht schwierig dar. Das linke Objekt a) in Abbildung 2 wurde ohne Lüfter, 3x langsamer und mit 50% mehr Druckmaterial gedruckt, wie das optimierte Objekt b) auf der rechten Seite. Der eigentliche G-Code ist für die Maschinensteuerung gedacht. Der Slicer verwendet die Folgenden (Einheiten in mm):

```
G0 X00 Y00 Z00 ; Bewegt den Druckkopf zum Punkt X/Y/Z
G1 F0000 ; Förderrate F pro Minute
G1 X00 Y00 Z00 E00 ; Druckt mit E Filament zum Punkt X/Y/Z
G10 ; Schliesst den Druckkopf
G11 ; Öffnet den Druckkopf
```

Der Initialisierungscode und Endcode wurden von den Druckeinstellungen des MakerBots und von Cura übernommen.
Initialisierungscode:

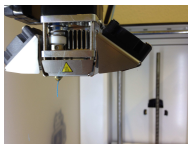
```
;FLAVOR:UltiGCode
;TIME:0
;MATERIAL:0
;MATERIAL2:0

;Layer count:0
M106 S255
G0 F9000 X20 Y20 Z0.3
G1 F3600 X180 Y20 Z0.3 E16.1208
G1 F3600 X180 Y180 Z0.3 E34.0609
G1 F3600 X20 Y180 Z0.3 E58.0992
G1 F3600 X20 Y20 Z0.3 E60.0392
```

Endcode:

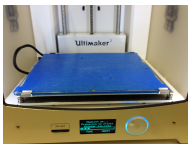
```
G10
M106 S0
M25
```

Im letzten Abschnitt des Initialisierungscode wird noch eine Schürze um das Objekt gezeichnet. Das wird verwendet, damit am Anfang nicht versehentlich Plastik auf das Objekt fällt und die Druckspur fest haften kann.

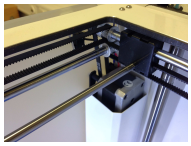
1.4 Hardware



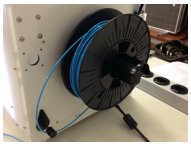
(a) Druckerkopf



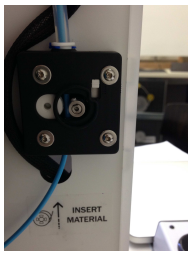
(b) Druckplatte



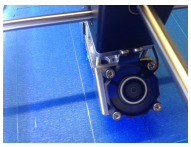
(c) Gewinde



(d) Filament



(e) Förderrotor



(f) Lüfter

Abbildung 4: Hardware Teile

Um überhaupt drucken zu können, muss man den 3D-Drucker verwendbar machen. Die Druckplatte muss kalibriert werden und ein Filament muss eingespannt werden. Dabei wird das Filament durch den Förderrotor gezogen, welcher dann das Filamentband hochzieht. Während dem Drucken ist der Druckkopf erhitzt um das Plastik zu schmelzen, damit nachher das Plastik aufgetragen werden kann. Die Gewinde bewegen sich wie im G-Code angegeben. Der Lüfter muss auch während dem Drucken laufen. Der Lüfter verhärtet eine Plastikschicht, damit die darüberliegende Schicht festen halt bekommt. Der vorgestellte Slicer schaltet den Lüfter während des ganzen Drucks auf seine maximale Leistung ein.

2 Begriffe & Variabeln

Bei der Erklärung der Algorithmen und in der Komplexitätsanalyse werden die folgenden Variablen verwendet:
N : Die Anzahl Dreiecke des Graphen. Die Approximation N = Anzahl Knoten des Graphen, wird angenommen.
Q : Die Breite der Druckspur. Sie entspricht auch der Anzahl Schichten des ausgedruckten Objekts.
D : Die Anzahl Dimensionen des Graphen (Modells). Normalerweise handelt es sich um 3-dimensionale Objekte.

K : Beschreibt die Anzahl Dimensionen eines spezifischen Graphens.
L : Durchschnittsgrösse der Kanten in Q-Einheiten.
$\mathcal{O}()$: Ist der Landau-Operator für die worst-case asymptotischer Laufzeit des Teilprogrammes. So würde $\mathcal{O}(N * N)$ bedeuten, dass man zum Beispiel im schlimmsten Fall eine quadratische Anzahl vergleiche der Dreiecke machen würde.
$\Theta()$: Ist der Landau-Operator welche die durchschnittliche asymptotische Laufzeit beschreibt. Er wird in dieser Arbeit verwendet um die Laufzeitanalyse für dünne Graphen zu berechnen, also für Graphen dessen Knoten durchschnittlich nicht mehr wie $\log N$ Kanten besitzen. Fast jedes druckbare Objekt ist

3 Graphentheoretischer Algorithmus

3.1 Theorie

Der graphentheoretische Algorithmus verwendet ein Divide & Conquer Verfahren [1] . Es reduziert pro Schritt einen D-dimensionalen Graphen in Q-Verschiedene (D-1)-dimensionale Graphen. Eine Reduktion kann in $\mathcal{O}(Q * N + N \log N)$ erreicht werden. Mithilfe eines Scanline-Algorithmus[6] kann eine Reduktion sogar in $\Theta(Q * \log N + N \log N)$ erreicht werden, bei einem dünnen Graphen. Erreicht das Objekt eine 1-dimensionale Struktur wird ein Raycasting-Algorithmus[7] angewendet um ihn in ein G-Code umzuwandeln. Die Laufzeit des Ray-Casting Algorithmus könnte schlimmstenfalls $O(N)$ betragen, doch durchschnittlich bei einem dünnen Graphen ist die Komplexität fast konstant $\Theta(1)$. Die Laufzeit des gesamten Algorithmus beträgt also im schlimmsten Fall $\mathcal{Q} * \mathcal{Q} * \mathcal{Q} * \mathcal{N} + \mathcal{N} \log \mathcal{N}$ und durchschnittlich $\Theta(Q * Q * \log N + N \log N)$. Der vorgestellte Algorithmus hat somit allgemein eine asymptotische Laufzeit von $\Theta(Q^(D-1) * \log N + D * N \log N)$

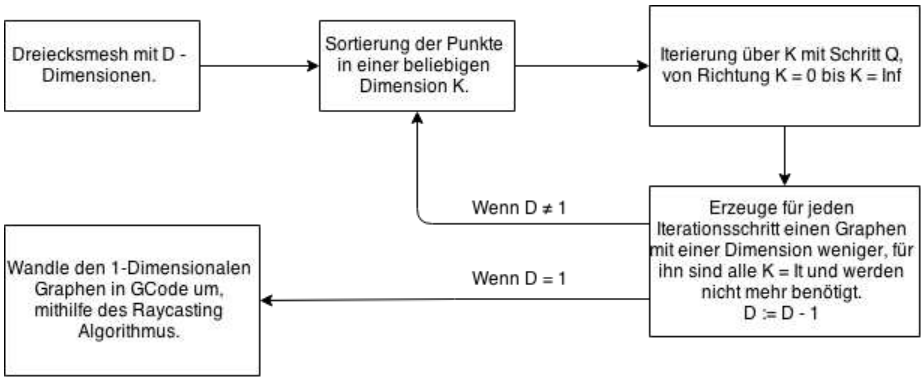


Abbildung 5: Graphentheoretischer Algorithmus Flowchart

[1] Mit dem Divide & Conquer Verfahren ist ein Lösungsansatz gemeint. Dabei wird ein Problem in Unterprobleme aufgeteilt (in diesem Fall in Raumdimensionen) um das Problem erleichtert zu bewältigen (in diesem Fall den 1-dimensionalen Streifen zu drucken).

3.2 STL Format einlesen

Nach dem einlesen einer ASCII-STL Datei mithilfe von ofFileDialogResult, muss ein ASCII-STL Parser geschrieben werden, namentlich um das Dokument verwenden zu können, um die Koordinaten aufzubereiten.

```
void parseAsciiSTL(vector<string> &input) {
  //Tokenizer
  //Parse Commands
  //Create Triangle List
  //Translation and Scaling of the coordinates
  //Rounding of the coordinates
  //Mesh or Field Generation
}
```

Code - Ausschnitt 1: Einlesen einer STL Datei

3.2.1 Tokenizer

Der Tokenizer (auch Lexer) wird die Eingabedatei in einen String-Container abspeichern, der die einzelnen Werte gekürzt vorbereitet. In der split Funktion werden die Leer- und Enter (\n) -zeichen gekürzt. Zuvor wird noch mit ::tolower die Eingabe kleingeschrieben.

```
vector<string> lexedInput;

vector<string> split (string &s, vector<char> &symbols) {
    vector<string> output;
    string temp = "";
    for(int i = 0; i < s.length(); ++i) {
        bool hasSplitted = false;
        for(int j = 0; j < symbols.size(); ++j) {
            if(s.at(i) == symbols[j]) {
                if(temp.length() != 0) output.push_back(temp);
                temp = "";
                hasSplitted = true;
            }
        }
        if(not hasSplitted) temp += s.at(i);
    }
    if(temp.length() != 0) output.push_back(temp);
    return output;
}

void parseAsciiSTL(vector<string> &input) {
    //Tokenizer
    lexedInput.clear();
    for(string &currentLine : input) {
        transform(currentLine.begin(), currentLine.end(), currentLine.
            begin(), ::tolower); //to lower case

        toSplit.push_back('\n');
        toSplit.push_back('␣');
        vector<string> splitted = split(currentLine, toSplit);
        lexedInput.insert(lexedInput.end(), splitted.begin(), splitted.
            end());
    }
    ...
```

```
}
```

Code - Ausschnitt 2: Aufbereitung der ASCII-STL Datei

3.2.2 Parser

Der Parser speichert jedes Dreieck in einen Vektor-Container. Jedes Dreieck besteht aus den Koordinaten des Dreiecks und des Normalenvektors, der Container triangleList ist also ein 2-dimensionales Array mit Grösse: N * 4.

```
void parseAsciiSTL(vector<string> &input) {
  ...
  //Parser
  int it = 0;
    vector<Vector> preLoadedTriangle (4, Vector(0, 0, 0));
    vector< vector<Vector> > triangleList;

    int currentPoint = 0;

    while(it < lexedInput.size())
    {
        if(lexedInput[it] == "solid")
        {
            while(lexedInput[++it] != "facet")
            {
                name += lexedInput[it];
            }
        }
        else if(lexedInput[it] == "normal")
        {
            preLoadedTriangle[3].x = ::atof( lexedInput[++it].c_str() )
                ;
            preLoadedTriangle[3].y = ::atof( lexedInput[++it].c_str() )
                ;
            preLoadedTriangle[3].z = ::atof( lexedInput[++it].c_str() )
                ;
        }
        else if(lexedInput[it] == "vertex")
        {
            preLoadedTriangle[currentPoint].x = ::atof ( lexedInput[++
                it].c_str() );
            preLoadedTriangle[currentPoint].y = ::atof ( lexedInput[++
                it].c_str() );
            preLoadedTriangle[currentPoint].z = ::atof ( lexedInput[++
                it].c_str() );
            ++currentPoint;
        }
        else if(lexedInput[it] == "endfacet")
        {
            triangleList.push_back(preLoadedTriangle);
            currentPoint = 0;
            ++it;
        }
        else ++it;
    }
    ...
}
```

Code - Ausschnitt 3: Konvertierung der STL Datei zu Vektoren

Die Koordinaten werden vorher noch verschoben und auf die gewünschte Grösse skaliert. Hiervon ist kein Code-Beispiel notwendig. Danach müssen die Koordinaten gerundet werden, da die Dreiecke teilweise die selben Eckpunktkoordinaten haben und diese nicht wegen einer Gleitkommazahl-Fehlpräzision verloren werden dürfen. Diese gerundeten Koordinaten werden noch wichtig sein für die Aussenlinie. Nach der Skalierung wird die setPrecision() Funktion aufgerufen um die Koordinaten gleich zu stellen.

```
struct Vector
{
    float x, y, z;

    Vector() : x(0), y(0), z(0) {}
    Vector(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}

    void setPrecision(int precision)
    {
        x = round(x * precision) / precision;
        y = round(y * precision) / precision;
        z = round(z * precision) / precision;
    }
};
```

Code - Ausschnitt 4: Rundung der Vektoren

3.3 Algorithmus 1 - Graphentheoretischer Algorithmus

3.3.1 Dreiecke abspeichern

Die Dreiecke werden nun in einer Klasse abgespeichert, wobei sichergestellt wird, dass ref1 der tiefste Punkt ist und ref3 der höchste Punkt.

```
struct Triangle {
    Vector ref1, ref2, ref3;
    Triangle(Vector& _ref1, Vector& _ref2, Vector& _ref3) :
    ref1(_ref1), ref2(_ref2), ref3(_ref3) {
        if(ref3 < ref2) swap(ref2, ref3);
        if(ref2 < ref1) swap(ref1, ref2);
        if(ref3 < ref2) swap(ref2, ref3);
    }
};

void parseAsciiSTL(vector<string>& input)
 ...
   vector<Triangle> slowTriangles;
    for(int i = 0; i < triangleList.size(); ++i) {
        Triangle tri (triangleList[i][0], triangleList[i][1],
            triangleList[i][2]);
        slowTriangles.push_back(tri);
    }
    slowAlgorithm(slowTriangles);
}
```

Code - Ausschnitt 5: Dreiecksdatenstruktur

3.3.2 Reduzierung der Dimensionen

Anhand des Source-Codes sieht man nun schön, wie der Algorithmus zuerst Dreiecke zu Segmenten umwandelt und diese später in Punkte umgewandelt werden. Die Laufzeit lässt sich berechnen, wenn man die For-Schleifen genauer analysiert. $\mathcal{O}(Q*N+Q^2*N+Q^2*N*logN) = \mathcal{O}(Q^2*NlogN)$. Die jeweiligen Dimensionsreduktionen werden weiter unten im Detail erklärt.

```
void slowAlgorithm(vector<Triangle> triangles)
{
    for(float y = 0; y <= settings.sizeY; y += settings.layerThickness)
         {
        vector<Segment> segments;
        for(Triangle& triangle : triangles) {
            if(triangle.ref3.y - triangle.ref1.y > 0.1 &&
               y >= triangle.ref1.y && y < triangle.ref3.y) {
                Segment segment = getCrossLine(triangle.ref1, triangle.
                    ref2, triangle.ref3, y);
                segments.push_back(segment);
            }
        }

        for(float x = 0; x <= settings.sizeX; x += settings.
            layerThickness) {
            vector<Vector> points;
            for(Segment& segment : segments) {
                if(segment.ref2.x < segment.ref1.x) swap(segment.ref1,
                    segment.ref2);
                if(segment.ref2.x - segment.ref1.x > 0.1 && x >=
                    segment.ref1.x && x < segment.ref2.x) {
                    Vector point = crossPointX(segment.ref1, segment.
                        ref2, x);
                    points.push_back(point);
                }
            }

            sort(points.begin(), points.end(), zComp);
            for(int i = 0; points.size() > 0 && i < points.size() - 1;
                i += 2) {
                moveTo(points[i]);
                printTo(points[i + 1]);
            }
        }
    }
}
```

Code - Ausschnitt 6: Kernalgorithmus durch Reduzierung der Dimensionen

3.3.3 Dreiecke in Segmente umwandeln

Jedes Dreieck müsste schlimmstenfalls Q-Mal an den Y-Koordinaten geschnitten werden. Um ein beliebiges Dreieck am Punkt y zu schneiden, muss man mit 4 Möglichkeiten rechnen.

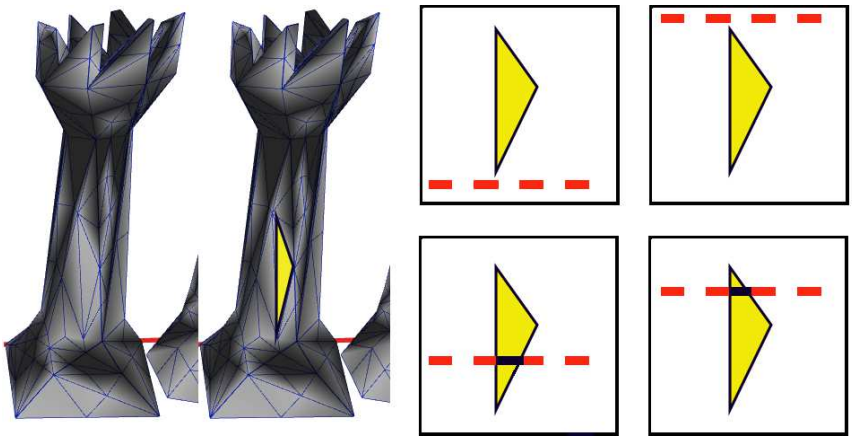


Abbildung 6: 4 Möglichkeiten des Slicers anhand einer Schachfigur

An den Stellen, wo die Y-Koordinate unter oder über dem Dreieck platziert ist, kontrollieren wir einfach ob der tiefste Punkt über der Y-Koordinate platziert ist oder der höchste Punkt unter der Y-Koordinate platziert ist. Gleichzeitig werden Dreiecke welche horizontal stehen ignoriert.

```
if(triangle.ref3.y - triangle.ref1.y > 0.1 &&
   y >= triangle.ref1.y && y < triangle.ref3.y) {
     Segment segment = getCrossLine(triangle.ref1, triangle.ref2,
         triangle.ref3, y);
     segments.push_back(segment);
}
```

Code - Ausschnitt 7: Aussortierung horizontaler Dreiecke

Das Segment der betroffenen Fälle wird in der getCrossLine Funktion berechnet. Auch hier muss zuerst überprüft werden, ob die Y-Koordinate oberhalb oder unterhalb des mittleren Punktes des Dreiecks steht. Wenn also die betroffene Y-Koordinate kleiner ist, wie die Y-Koordinate des mittleren Punktes, wird der Schnittpunkt des Segmentes aus dem unteren und dem mittleren Punkt berechnet, wenn nicht zwischen dem mittleren und oberen Punkt. Liegen der untere und der mittlere Punkt auf der gleichen Y-Koordinate, muss das Segment zwischen dem mittleren und dem oberen Punkt geschnitten werden. Liegen der obere und der mittlere Punkt auf der gleichen Y-Koordinate, muss das Segment zwischen dem mittleren und dem unteren Punkt geschnitten werden. Der Schnittpunkt zwischen dem Segment des unteren und oberen Punktes muss immer berechnet werden. Auf der Abbildung der Schachfigur wird es noch deutlicher. Die beiden Schnittpunkte bilden nun das Schnittsegment des Dreiecks an der gewählten Y-Koordinate.

```
Segment getCrossLine(Vector& lower, Vector& middle, Vector& upper,
    float yPos) {
    Vector first = crossPointY(lower, upper, yPos);
    Vector second;
    if(yPos < middle.y) {
        //lower & middle
        if(middle.y - lower.y < 0.2) second = middle;
        else second = crossPointY(lower, middle, yPos);
```

```
    }
    else {
        //middle & upper
        if(upper.y - middle.y < 0.2) second = middle;
        second = crossPointY(middle, upper, yPos);
    }

    Segment output(first, second);
    return output;
}
```

Code - Ausschnitt 8: Signifikante Strecken zur Schnittsegmentberechnung wählen

Um den Schnittpunkt eines Segments an einer Y-Koordinate zu berechnen speisen wir die beiden Punkte in die crossPointY Funktion. Diese berechnet die fehlenden X und Z Koordinaten des Dreiecks. Dabei wird das Verhältnis zwischen der Y-Differenz des unteren und oberen Punktes und der Y-Differenz des unteren Punktes und der geschnittenen Y-Koordinate berechnet. Dieses Verhältnis kann dann auch auf die X und Z-Koordinate angewendet werden. Eine Division durch null kann nicht passieren, da horizontale Dreiecke vorher schon heraus gesiebt worden sind.

```
Vector crossPointY(Vector& lower, Vector& upper, float yPos) {
    Vector output;
    float diff = (yPos - lower.y) / (upper.y - lower.y);
    output.y = yPos;
    output.x = (diff * (upper.x - lower.x)) + lower.x;
    output.z = (diff * (upper.z - lower.z)) + lower.z;
    return output;
}
```

Code - Ausschnitt 9: Schnittsegmentberechnung

3.3.4 Segmente in Punkte umwandeln

Ähnlich wie bei der ersten Reduzierung werden nun diese Schnittsegmente in Punkte an den jeweiligen X-Koordinaten schlimmstenfalls Q-mal geschnitten. Auch hier werden die Segmente aussortiert die sich über- oder unterhalb der Scanline befinden.

```
if(segment.ref2.x - segment.ref1.x > 0.1 && x >= segment.ref1.x && x <
      segment.ref2.x)
```

Code - Ausschnitt 10: Aussortierung paralleler Segmente zur x-Achse

Da es nur einen Schnittpunkt gibt, kann man direkt zum berechnen der Z-Koordinate relativ zur X-Koordinate übergehen, dafür sorgt die Funktion $crossPointX(Vector\&, Vector\&, float)$. Sie funktioniert nach dem gleichen Prinzip, wie die Schnittpunkt-Berechnung der Y-Koordinate.

```
Vector crossPointX(Vector& lower, Vector& upper, float xPos) {
    Vector output;
    float diff = (xPos - lower.x) / (upper.x - lower.x);
    output.x = xPos;
    output.y = lower.y; //previously determined
    output.z = (diff * (upper.z - lower.z)) + lower.z;
```

```
    return output;
}
```

Code - Ausschnitt 11: Schnittpunktberechnung

3.3.5 Punkte in G-Code umwandeln

Für jeden Q-ten Schnitt der Y-Koordinate, haben wir Q Schnitte parallel zur X-Koordinate und zu jedem Schnitt auf der X-Achse haben wir einen Satz von Punkten, welcher ein mehrfaches von 2 Punkten beinhalten muss. Genauer, beginnt man von einer Seite zu zählen, sind die ungeraden Punkte immer am Anfang einer Druckspur und die geraden am Ende einer Druckspur. Dieses Prinzip macht es ein Ray-Casting Algorithmus.

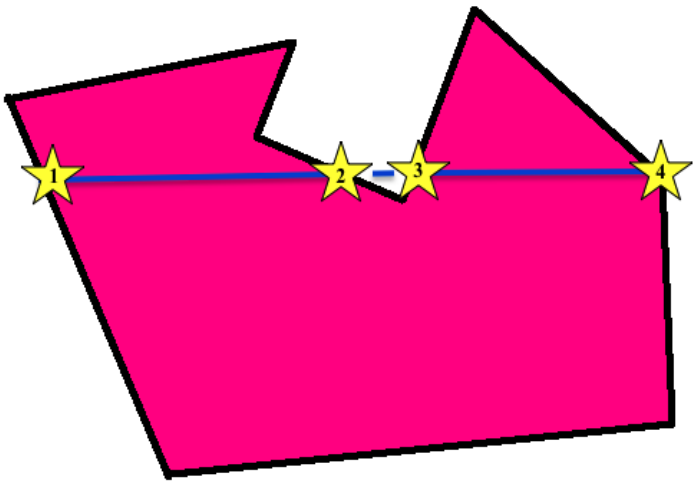


Abbildung 7: Ray-Casting Algorithmus

Jetzt muss man nur noch von jeder geraden Anzahl (wenn man von 0 beginnt zu zählen) zu jeder ungeraden Anzahl eine Linie zeichnen.

```
sort(points.begin(), points.end(), zComp);
for(int i = 0; points.size() > 0 && i < points.size() - 1; i += 2) {
    moveTo(points[i]);
    printTo(points[i + 1]);
}
```

Code - Ausschnitt 12: Ray-Casting Algorithmus

3.3.6 Bewegungen des Druckkopfes

In den beiden Funktionen printTo und moveTo werden die Druckbewegungen kontrolliert. Die Bewegung ohne genau zu Drucken läuft über G0 und wird auch verwendet um etwas auszudrucken. Die Syntax der G0 Funktion läuft wie folgt: $G0FXZY$
F ist die Feedrate, also die Vorschubgeschwindigkeit, und X, Z und Y sind die Destinationskoordinaten. Die Koordinaten Y und Z müssen vertauscht werden, da sie im STL-Format umgekehrt aufgeschrieben sind. Die Geschwindigkeiten des Bewegens und des Druckens sind normalerweise unterschiedlich und deswegen werden auch zwei verschiedene Einstellungen verwendet: settings.moveRate und settings.printRate.

```
void moveTo(float x, float y, float z)
{
    z += settings.pointAbovePlate;
    x += transX;
    y += transZ;

    //SAFETY JUST IN CASE
    if(x < 0 || y < 0 || z < 0 || x > settings.sizeX || y > settings.
        sizeY || z > settings.sizeZ) {
        cout << "Warning, moves to: " << x << " " << y << " " << z <<
            endl;
    }

    string output = "G0 F" + settings.moveRate + " X" + ofToString(x) +
         " Y" + ofToString(y) + " Z" + ofToString(z);
    gcode.push_back(output);

    previousX = x;
    previousY = y;
    previousZ = z;
}

void moveTo(Vector &temp)
{
    //Swap Z & Y Axis;
    moveTo(temp.x, temp.z, temp.y);
}
```

Code - Ausschnitt 13: G0-Code Kreierung

Die Koordinaten previousX, previousY und previousZ sind die aktuellen Koordinaten des Druckkopfes und erlauben eine Berechnung der Distanz. Auch die die Fülldichte E (Extrusion amount) muss hier angegeben werden und mit der vorherigen Fülldichte summiert werden, für welche man die Distanz benötigt.

```
  void printTo(float x, float y, float z)
{
    z += settings.pointAbovePlate;
    x += transX;
    y += transZ;

    //SAFETY JUST IN CASE
    if(x < 0 || y < 0 || z < 0 || x > settings.sizeX || y > settings.
        sizeY || z > settings.sizeZ) {
```

```
      cout << "Warning, moves to: " << x << " " << y << " " << z <<
          endl;
    }

    float extrusion = 0;

    float distX = abs(x - previousX);
    float distY = abs(y - previousZ);
    float distZ = abs(z - previousZ);

    float dist =  sqrt(distX * distX + distY * distY + distZ * distZ);

    extrusion = dist * settings.fillThickness;

    string output = "G1 F" + settings.fillRate + " X" + ofToString(x) +
         " Y"  + ofToString(y) + " Z" + ofToString(z) + " E" +
        ofToString(extrusion + previousExtrusion);
    gcode.push_back(output);

    previousX = x;
    previousY = y;
    previousZ = z;
    previousExtrusion += extrusion;

}

void printTo(Vector &temp)
{
    //Swap Z & Y Axis;
    printTo(temp.x, temp.z, temp.y);
}
```

Code - Ausschnitt 14: G1-Code Kreierung

3.4 Optimierungen

3.4.1 Optimierung der Einstellungen

Einstellungen wie Q (die Druckspurbreite), Füllrate, Geschwindigkeit des Druckerkopfes und der Umdrehung des Lüfters, Fülldichte, etc. brauchten eine lange Zeit um richtig eingestellt zu werden. Hier sind die aktuellen Einstellungen des Slicers:

```
struct Settings {
    float precision = 1; //1 = 1 mm, 2 = 0.1 mm, 3 = 0.01 mm, etc.
    float sizeX = 200, sizeY = 200, sizeZ = 200;
    float layerThickness = 0.2, pointAbovePlate = 0.1;
    float fillThickness = 0.1;
    float fillSpace = 5.0;
    int fillAlgorithm = 1;
    bool fanOn = true;
    bool addRetraction = true;
    float retractionDist = 5.0;
    bool wireHotspotFilter = false;
    string fillRate = "3600"; //36000 (2400)
    string moveRate = "9000"; //90000
} settings;
```

Code - Ausschnitt 15: Einstellungen des Slicers

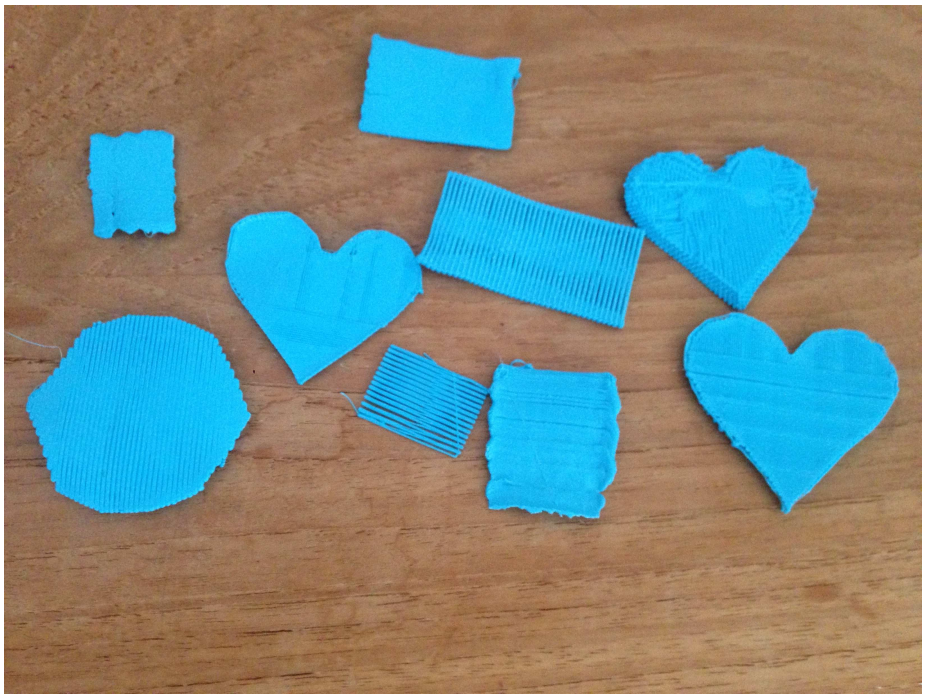


Abbildung 8: Verschieden Tests um die Einstellungen richtig hinzubekommen

3.4.2 Einsaugoptimierung

Obwohl bei der Bewegung des Druckkopfes mit G0 eigentlich kein Druckfilament aufgetragen werden sollte, schmiert der erhitzte Druckkopf trotzdem eine Spur hinter sich her. So kommt es zu einem diagonalen Muster anstelle reihenhafter 'Würste'.

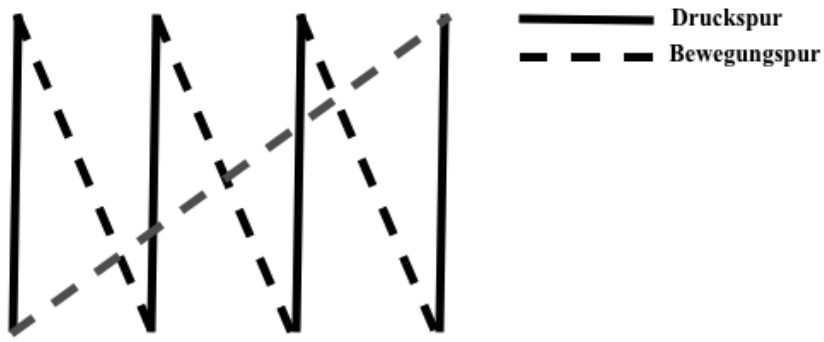


Abbildung 9: Schmierdiagonale

Das Problem lässt sich mit den Kommandos G10 und G11 lösen, welche das Filament kurz zurückziehen, respektive wieder herauslassen. Wenn nun die Bewegungsdistanz einen gewissen Wert übersteigt wird das Filament eingesaugt und wieder ausgesaugt. Eine einfache Modifikation in der moveTo Funktion und das Problem scheint behoben zu sein.

```
if(settings.addRetraction && settings.retractionDEist <= approxDist)
    gcode.push_back("G10");
gcode.push_back(output);
if(settings.addRetraction && settings.retractionDist <= approxDist)
    gcode.push_back("G11");
```

Code - Ausschnitt 16: Einsaugoptimierung

(a) Ohne Optimierung (b) Mit Optimierung

Abbildung 10: Einsaugoptimierung

3.4.3 Einfräsung des Förderrads

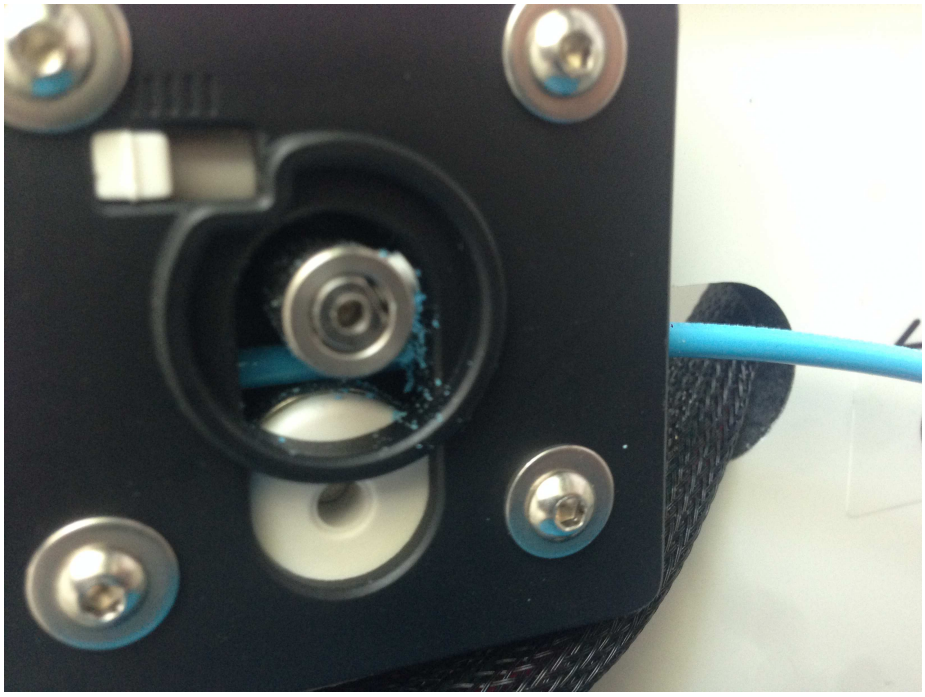


Abbildung 11: Einfräsung des Plastiks

Wenn nun das Filament jedes Mal die zuvor genannte Bewegungsdistanz übersteigt, wird das Filament viel zu oft hineingesogen. So bohrt sich das Förderrad in das Filament ein. Um diesen Hardware Fehler zu beheben, kann man ständig, pro Druckspur die Richtung ändern und somit eine Form von gewolltem Zig-Zag zu erreichen. Gleich nach der Sortierung der Punkte, muss man nur noch eine globale boolesche Variable wählen und diese jedes Mal invertieren.

```
sort(points.begin(), points.end(), zComp);
if(direction) reverse(points.begin(), points.end());
direction = !direction;
```

Code - Ausschnitt 17: Zig-Zag Optimierung

Die vorherige Optimierung bleibt jedoch implementiert für komplexere Bewegungen.

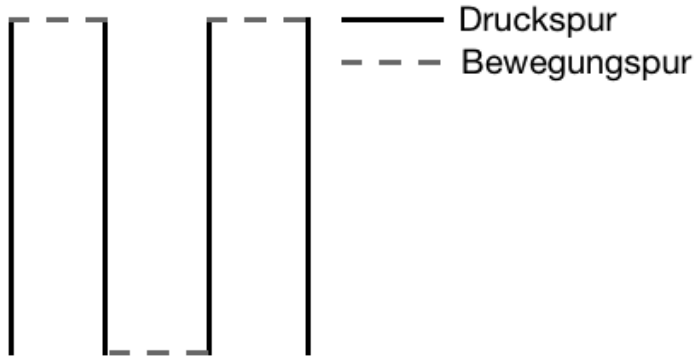


Abbildung 12: Zig-Zag Spur

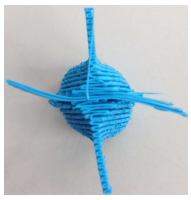
3.4.4 Gitterförmige Darstellung

Ein Problem der Füllmethode von vorhin ist die unsymmetrische Darstellung, zumindest von oben betrachtet. Wechselt man nun aber pro Y-Schicht die Richtung des Druckens, wird das Produkt viel symmetrischer und statisch stabiler. Es wird auch später beim Material sparen verwendet. Der Füllalgorithmus wird in eine separate Funktion gepackt um den Programmiercode übersichtlicher zu behalten. Abwechslungsweise wird nun entweder die X-Koordinate oder die Z-Koordinate geschnitten.

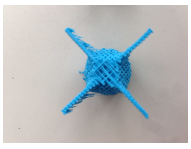
```
bool xOrZ = false;
void fillAlgorithm1(vector<Segment>& segments, float newThickness) {
    bool direction = false;
    if(xOrZ) {
        for(float x = 0; x <= settings.sizeX; x += newThickness) {
            vector<Vector> points;
            for(Segment& segment : segments) {
                if(segment.ref2.x < segment.ref1.x) swap(segment.ref1,
                    segment.ref2);
                if(segment.ref2.x - segment.ref1.x > 0.1 && x >=
                    segment.ref1.x && x < segment.ref2.x) {
                    Vector point = crossPointX(segment.ref1, segment.
                        ref2, x);
                    points.push_back(point);
                }
            }

            sort(points.begin(), points.end(), zComp);
            if(direction) reverse(points.begin(), points.end());
            direction = !direction;
            for(int i = 0; points.size() > 0 && i < points.size() - 1;
                i += 2) {
                if(!direction) {
                    points[i].z -= settings.layerThickness * 3;
                    points[i+1].z += settings.layerThickness * 3;
```

```
                }
                else {
                    points[i].z += settings.layerThickness * 3;
                    points[i+1].z -= settings.layerThickness * 3;
                }
                moveTo(points[i]);
                printTo(points[i + 1]);
            }
        }
    }
    else {
        for(float z = 0; z <= settings.sizeZ; z += newThickness) {
            vector<Vector> points;
            for(Segment& segment : segments) {
                if(segment.ref2.z < segment.ref1.z) swap(segment.ref1,
                    segment.ref2);
                if(segment.ref2.z - segment.ref1.z > 0.1 && z >=
                    segment.ref1.z && z < segment.ref2.z) {
                    Vector point = crossPointZ(segment.ref1, segment.
                        ref2, z);
                    points.push_back(point);
                }
            }

            sort(points.begin(), points.end(), xComp);
            if(direction) reverse(points.begin(), points.end());
            direction = !direction;
            for(int i = 0; points.size() > 0 && i < points.size() - 1;
                i += 2) {
                if(!direction) {
                    points[i].x -= settings.layerThickness * 3;
                    points[i+1].x += settings.layerThickness * 3;
                }
                else {
                    points[i].x += settings.layerThickness * 3;
                    points[i+1].x -= settings.layerThickness * 3;
                }
                moveTo(points[i]);
                printTo(points[i + 1]);
            }
        }
    }
}
```

Code - Ausschnitt 18: Gitteroptimierung



(a) Ohne Abwechslung



(b) Mit Abwechslung

Abbildung 13: Unsymmetrische Darstellungen

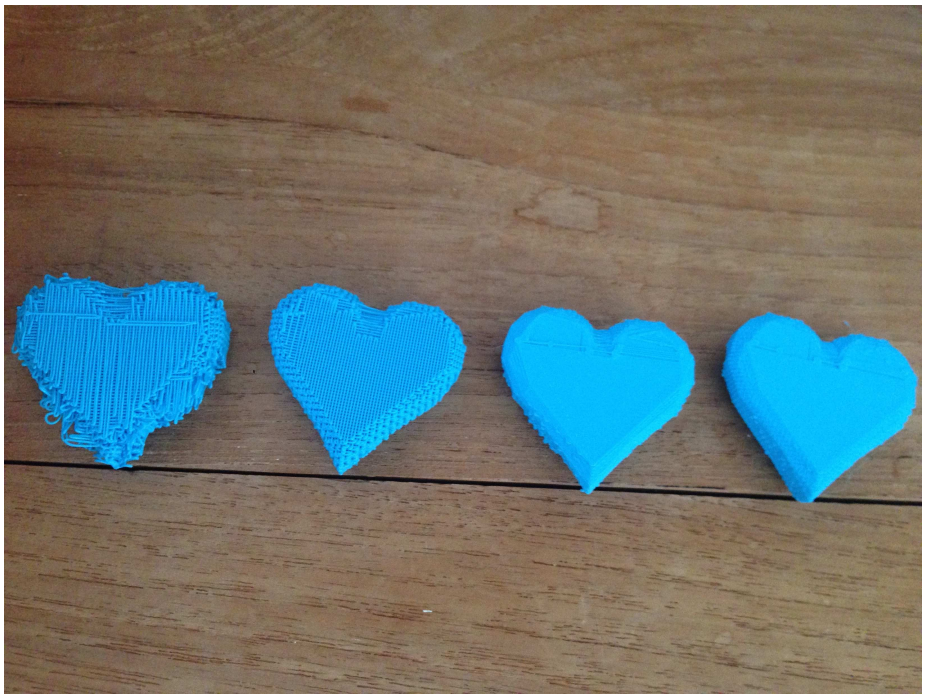


Abbildung 14: Verschiedene Gittereinstellungen (Dichte der Linien, Häufigkeit der Abwechslung)

3.4.5 Aussenlinie

Abbildung 15: Herz nur mit Aussenlinie gedruckt

Eine Figur ist zwar druckbar aber sieht noch sehr verpixelt aus. Um die Struktur zu verfeinern kann man eine Hülle um das Objekt zeichnen. Die vorher berechneten Schnittsegmente sind genau diese Hülle, man muss jedoch noch die Reihenfolge des Druckens berechnen. Um die Hülle zu zeichnen verwen-

det man den Greedy-Algorithmus[2] des Problems: So bewegt man sich immer zum nächstgelegenen, noch ungedruckten Segment, des vorherig gedruckten gewählten Segments. Man bewegt sich sogar immer zum nächstgelegen Segmentpunkt des vorherigen Segmentpunkts.

```
void createShell(vector<Segment>& segments, float scale)
{
    if(segments.size() > 0) {
        vector<bool> used(segments.size(), 0);

        bool done = false;
        bool position = false;
        int i = 0;
        //antipickel mode
        Vector positionFrom(0, 0, 0);
        while(!done) {

            float dist = MAX_FLOAT;
            for(int j = 0; j < segments.size(); ++j) {
                if(!used[j]) {
                    if( distanceY(positionFrom, segments[j].ref1) <
                        dist) {
                        dist = distanceY(positionFrom, segments[j].ref1
                            );
                        position = true;
                        i = j;
                    }
                    if( distanceY(positionFrom, segments[j].ref2) <
                        dist) {
                        dist = distanceY(positionFrom, segments[j].ref2
                            );
                        position = false;
                        i = j;
                    }
                }
                else if(j == segments.size() - 1 && dist == MAX_FLOAT)
                    done = true;
            }
            used[i] = true;
            if(position) {
                moveTo(segments[i].ref1);
                printTo(segments[i].ref2);
                positionFrom = segments[i].ref2;
            }
            else {
                moveTo(segments[i].ref2);
                printTo(segments[i].ref1);
                positionFrom = segments[i].ref1;
            }
        }
    }
}
```

Code - Ausschnitt 19: Erzeugung der Aussenlinie

[2]Mit einem Greedy-Algorithmus ist ein Lösungsverfahren gemeint, welches immer nach der nächstbesten intuitiven Lösung sucht.

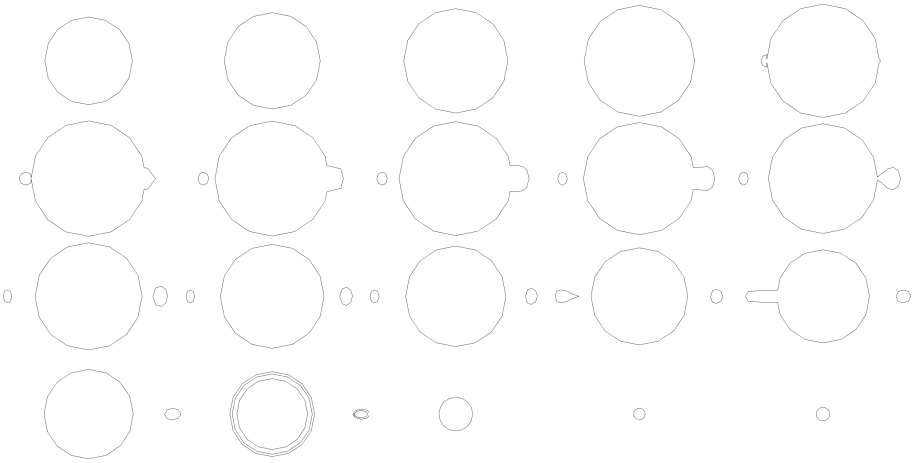
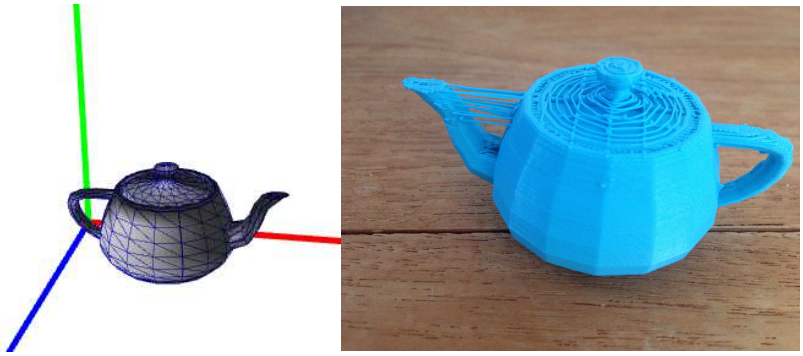


Abbildung 16: Die Aussenlinien einer Teekanne



(a) Teekanne im Slicer (b) Teekanne ausgedruckt

Abbildung 17: Teekanne Digital & Ausgedruckt

3.4.6 Pickeleffekt

Bei jeder Schicht muss der 3D Drucker an einer Position beginnen. An dieser Stelle hinterlässt er einen kleinen Plastiktupfer. Um diesen Pickeleffekt vorzubeugen wählt man bei der Hüllenkreierung einen Punkt (0,0), welcher für den Algorithmus den vorherigen Druckpunkt ist. So beginnt der Slicer immer an der Strecke, welche am nächsten bei (0,0) ist und hinterlässt keine Spuren, wenn er zu drucken beginnt. Auch Cura bekam ein Update mit dem 'Joris mode', der die gleiche Lösung zu diesem Problem fand.

Abbildung 18: Zylinder nach Pickeloptimierung

3.4.7 Aussenlinie vs. Fülllinie

Wenn man nun eine Aussenlinie um ein Objekt herum zeichnet, entsteht ein Konflikt mit der Fülllinie, da beide Linien dieselben Punkte besuchen. Man kann nun die Punkte nach aussen verschieben, um die Aussenlinie nicht zu streifen, doch dass wäre ziemlich rechenintensiv und bräuchte mehr Programmiercode. Die Alternative ist ein kleiner Trick, entweder man verschiebt die Aussenlinie oder man verschiebt die Fülllinie. Die Fülllinie muss einfach bei einer geraden Kreuzung mit der Aussenlinie ein wenig nach vorne verschoben werden, während bei einer ungeraden Kreuzung mit der Aussenlinie, die Fülllinie nach innen verschoben werden sollte.

```
for(int i = 0; points.size() > 0 && i < points.size() - 1; i += 2) {
    moveTo(points[i]) += settings.layerThickness * 3;
    printTo(points[i + 1]) -= settings.layerThickness * 3;
}
```

Code - Ausschnitt 20: Aussenlinie vs. Fülllinie

3.4.8 Materialeinsparung

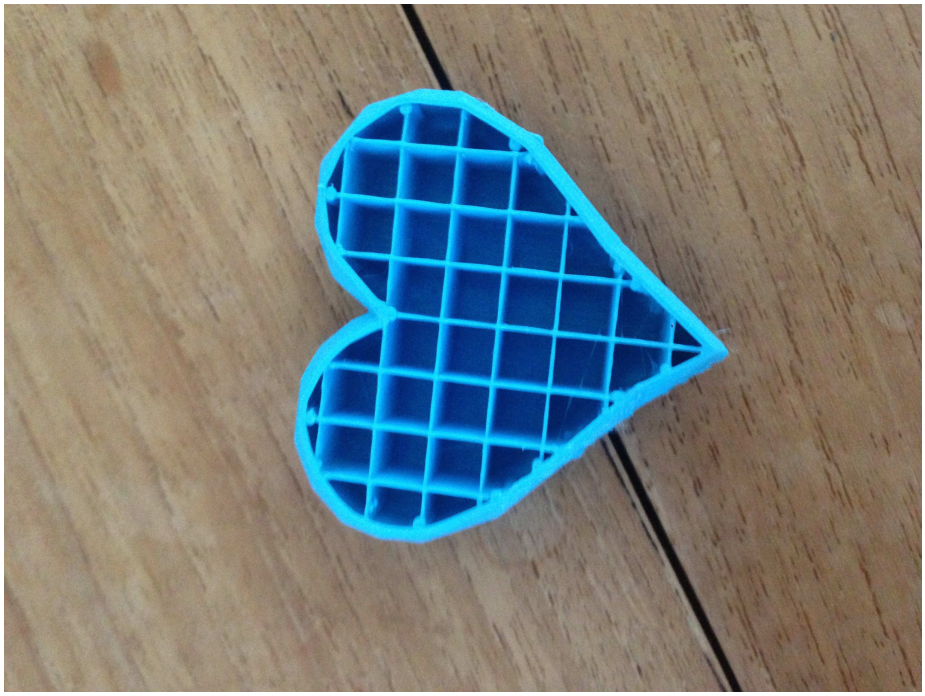


Abbildung 19: Fülldichte des Slicers zum Materialsparen

Anstelle das Objekt mit Plastik zu füllen, kann man es auch nur teilweise mit Plastik füllen. Der vorhin beschriebene Füllalgorithmus wurde in eine Funktion verpackt, um nun Abstände zwischen den Linien zu machen. Das resultiert in eine Quadratstruktur (Siehe Bild). Es gibt verschiedene Füllstrukturen, es gibt zum Beispiel Honigwaben, um ein Objekt stabiler zu machen bei diagonaler Kraftauslastung oder noch viel kompliziertere 3-dimensionale Strukturen so wie ein Tetrakaidecahedron, um von jeder möglichen Seite das Objekt stabilisieren zu können. Man kann den Füllalgorithmus auch ganz weglassen und stattdessen die Aussenlinie nach innen verschieben, eine sogenannte konzentrische Füllung. Bei allen Füllungen entsteht aber das gleiche Problem: Es gibt Stellen die trotzdem vollständig gefüllt werden müssen, denn sonst fehlt jegliche horizontale Fläche. Um das Problem zu beheben kann man alle Schichten, welche eine horizontale Fläche besitzen, mit enger Fülldichte füllen, so wie am Anfang.

```
bool verticalHack = false;
[...]
 if(triangle.ref3.y - triangle.ref1.y <= 0.1 && abs(y - triangle.ref1.y
     ) < 0.8) {
                verticalHack = true;
}
[...]
if(verticalHack) fillAlgorithm1(segments, settings.layerThickness * 2);
else fillAlgorithm1(segments, settings.fillSpace);
```

Code - Ausschnitt 21: Hackvariante

Eine elegantere Variante, welche mehr Material spart und nicht die ganze Schicht füllt, würde den Füllalgorithmus ein zweites Mal anwenden, für die Stellen, welche eine horizontale Fläche besitzen. Da aber mehrere horizontale Dreiecke pro horizontale Fläche auftreten können, müssen Streckenduplikate aussortiert werden. (Zwei horizontale Dreiecke könnten sich ja nebeneinander befinden und so werden Streckenduplikate entfernt). Streckenduplikate können mit einem Set, einer sortierten Datenstruktur (ein Baum) aussortiert werden.

```
for(float y = -0.02; y <= settings.sizeY; y += settings.layerThickness)
     {
    [...]
    set<Segment> edgeSegments; //delete all duplicate lines
    vector<Segment> edgeSegmentsVector;
    for(Triangle& triangle : triangles) {
        [...]
        if(triangle.ref3.y - triangle.ref1.y <= 0.1 && abs(y - triangle
            .ref1.y) < 0.8) {
            Segment a (triangle.ref1, triangle.ref2);
            a.ref1.y = y; a.ref2.y = y;
            if(edgeSegments.count(a) != 0) edgeSegments.erase(a);
            else edgeSegments.insert(a);
            Segment b (triangle.ref2, triangle.ref3);
            b.ref1.y = y; b.ref2.y = y;
            if(edgeSegments.count(b) != 0) edgeSegments.erase(b);
            else edgeSegments.insert(b);
            Segment c (triangle.ref1, triangle.ref3);
            c.ref1.y = y; c.ref2.y = y;
            if(edgeSegments.count(c) != 0) edgeSegments.erase(c);
            else edgeSegments.insert(c);
        }
    }
    [...]
    copy(edgeSegments.begin(), edgeSegments.end(), back_inserter(
        edgeSegmentsVector));
    fillAlgorithm1(edgeSegmentsVector, settings.layerThickness * 2);
}
```

Code - Ausschnitt 22: Elegante Variante

3.4.9 Pizzaiolo Optimierung

Diese Technik habe ich mir direkt bei Cura abgeschaut, als es eine kleine Fläche drucken musste. Cura hat bei ganz kleinen Flächen nicht gefüllt, sondern stattdessen die Aussenlinie 2-mal aufgetragen. Die createShell Funktion muss also einfach 2x aufgerufen werden.

3.4.10 Low-Poly Optimierung

Ein weiteres Problem kann die Grösse der Dreiecke darstellen. Möchte man z.B. einen 1 (cm) x 1 (cm) x 1 (cm) Rubrik's Würfel drucken, wären die Dreiecke kleiner als die zu druckenden Schichten. In diesem Fall funktioniert der Slicer schlicht und einfach nicht. Das gleiche Problem taucht auf, wenn man ein Modell in sehr hoher Auflösung einscannt. Dann muss man vorher das STL-File mit einem Algorithmus, wie der Garland-Heckbert Algorithmus [10], bearbeiten.

3.4.11 Umfallen der Figur

Ein mysteriöses Problem war das Umfallen von grösseren Objekten die gedruckt wurden. Zuerst war die Vermutung, dass der Untergrund nicht mehr gut klebte und die Objekte nicht mehr fest hafteten, doch beim zweiten Mal wurde das Umfallen gefilmt und es schien nicht nur am Boden zu liegen. Anscheinend wird je nach Objekt die Gewichtsverlagerung so stark, dass die Figur umfällt. Um dieses Problem zu beheben, kann man einfach die ersten 2 Zentimeter vollgedruckt füllen, so bleibt es stabil am Boden stehen.

```
if(y < 2) fillAlgorithm(segments, settings.layerThickness * 2); //Get
    fundament
```

Code - Ausschnitt 23: Bodenständigkeit

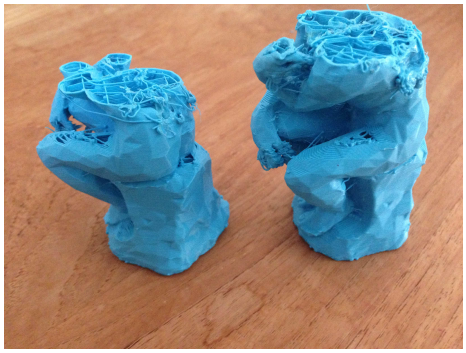


Abbildung 20: Umgefallene Figuren

3.4.12 Weitere Probleme

Ein weiteres Problem ist das Abnehmen einer Figur. Die Druckplatte sollte beim Entfernen erhitzt und die Figur sollte abgekühlt werden. Auch wenn diese Vorkehrungen getroffen sind, sind manche Objekte immer noch sehr schwer zu entfernen. Beim Bildbeispiel ist nicht nur die Bodenplatte beim wegnehmen verbogen, sondern auch eine Stütze abgebrochen.

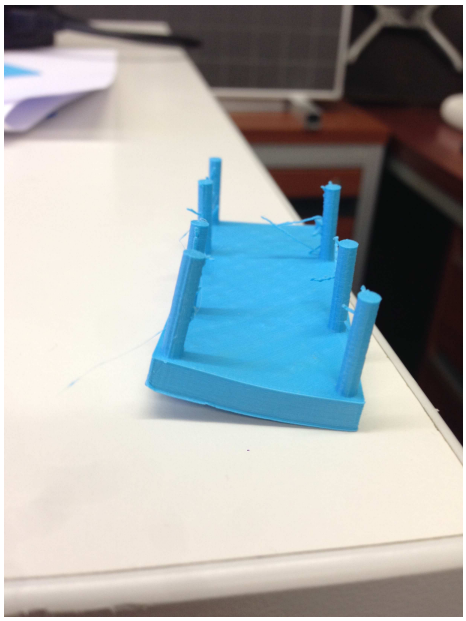


Abbildung 21: Durch das Entfernen verbogene Figur

Im Endcode darf nicht vergessen werden, nach der letzten Druckbewegung das Druckfilament ruckartig einzusaugen, da sonst das ganze Objekt noch mit einem Strich gekrönt wird.

Das Einfräsen der Motoren war immer noch ein Problem, nicht wegen dem Einsaugen, sondern wegen der Druckspule, die sich verknotete. Danach musste man das Plastik entwirren, denn ab und zu werden Plastikstriche gezogen, die nicht hätten gezogen werden müssen.

3.5 Laufzeitoptimierung

3.5.1 Das Hüllenproblem

Der vorherige Greedy-Algorithmus der Aussenlinie wurde in $O(N^2 * Q)$ implementiert. Er könnte mit einer *map* in $O(N \log N * Q)$ implementiert werden, bei dem einfach jeder Punkt in ein Array gespeichert wird, welches keine Duplikate zulässt (Mittels einer *unordered_map* könnte man auch $O(N * Q)$ erreichen). In diesem Array werden ausserdem noch die Segmente abgespeichert, sodass der Slicer in linearer Laufzeit den Punkten folgen kann.

3.5.2 Scanline Algorithmus

Die Laufzeit des bisherigen Algorithmus liegt immer noch bei $\Theta(Q * Q * N)$ (mit Radix-Sort und einer *unordered_map*). Die folgende Optimierung reduziert die Komplexität des Algorithmus auf $O(Q * Q * logN + N * logN)$ bei dünnen Graphen, also wenn pro Knoten wenig Kanten ausgehen, um genauer zu sein,

wenn pro Knoten durchschnittlich nur $logN$ Kanten ausgehen. Der Trick dabei ist, gewisse Fälle im voraus heraus zu filtern, um so nur die betroffenen Fälle zu behandeln. So wird für jedes Dreieck die untere Ecke und die obere Ecke notiert. Danach werden diese Punkte anhand ihrer Y-Koordinate sortiert. Haben zwei Punkte die gleiche Y-Koordinate, werden Punkte dessen Dreiecke hinzugefügt werden vorgerückt, als ob sie eine kleinere Y-Koordinate hätten (Siehe Tabelle 1). Wenn nun der Slicer das Objekt in 2-dimensionale Graphen zu schneiden beginnt, kann man in der sortierten Datenstruktur auf die aktuell-verwendeten Dreiecke zugreifen.

Y-Koordinate	0.0	0.0	0.0	1.2	1.2	1.5	1.5	2.0	2.0	2.0
Betroffenes Dreieck	A	B	C	D	B	E	A	C	D	E
Hinzufügen / Löschen	H	H	H	H	L	H	L	L	L	L
Aktuelle Dreiecke	A	A, B	A, B, C	A, B, C, D	A, C, D	A, C, D, E	C, D, E	D, E	E	-

Tabelle 1: Datenstruktur der Scanlinie

```
struct scanlineNode
{
    int triangleReference;
    float sortPosition;
    bool insertRemove; //true = insert; false = remove
    scanlineNode() {}

    scanlineNode(int _triangleReference, bool _insertRemove, float
        _sortPosition) : triangleReference(_triangleReference),
        insertRemove(_insertRemove), sortPosition(_sortPosition) {}
};

bool operator <(const scanlineNode& lhs, const scanlineNode& rhs)
{
    if(lhs.sortPosition == rhs.sortPosition) return lhs.insertRemove;
        //Insertion is always more superior
    else return lhs.sortPosition < rhs.sortPosition;
}
```

Code - Ausschnitt 24: Datenstruktur der Scanlinie

Die Implementation der Datenstruktur wurde so gewählt, dass man die gleiche Implementation nicht nur für Dreiecke entlang der Y-Achse verwenden kann, sondern auch für Segmente entlang der X (oder Z) - Achse und sogar für Tetraeder auf einer 4-ten Achse.

```
void scanline(vector<Triangle> allTriangles)
{
     vector<scanlineNode> scanTriangles;
    //Create scanTriangles
    for(int i = 0; i < allTriangles.size(); ++i) {
        scanlineNode adding(i, true, allTriangles[i].ref1.y);
        scanlineNode removing(i, false, allTriangles[i].ref3.y);
        scanTriangles.push_back(adding);
        scanTriangles.push_back(removing);
    }
    sort(scanTriangles.begin(), scanTriangles.end());
    [...]
}
```

Code - Ausschnitt 25: Erzeugung der Scanlinie für Dreiecke

Wie gewohnt wird von unten nach oben mit Schritt Q und Startposition 0 iteriert. Bei jedem Iterationsschritt werden die betreffenden Dreiecke herausgefiltert, die in dem Y-Wert des Iterationsschrittes liegen. Dem Baum 'triangles' werden nun die neuen Dreiecke hinzugefügt. Die Dreiecke im Baum werden dann wie gewohnt in Segmente aufgeteilt und gefüllt. Am Ende des Iterationsschritts werden aufhörende Dreiecke heraus sortiert. So erreichen wir beim herauslesen der Dreiecke nur $\mathcal{O}(Q + N \log N)$ und nicht $\mathcal{O}(Q * N)$ Schritte bei einem durchschnittlichen Modell.

```
[...]
int scanlinePosition = 0;
set<Triangle> triangles;
for(float y = -0.02; y <= settings.sizeY; y += settings.layerThickness)
     {
    int previousPosition = scanlinePosition;
    //First add all new triangles:

    while (scanlinePosition < scanTriangles.size() && scanTriangles[
        scanlinePosition].sortPosition <= y)
    {
        if(scanTriangles[scanlinePosition].insertRemove == true)
            triangles.insert(allTriangles[ scanTriangles[
            scanlinePosition].reference ] );
        ++scanlinePosition;
    }
    [...]

    //Remove old triangles
    while (previousPosition < scanTriangles.size() && scanTriangles[
        previousPosition].sortPosition <= y) {
            if(scanTriangles[previousPosition].insertRemove == false)
                triangles.erase(allTriangles [scanTriangles[
                previousPosition].reference] );
            ++previousPosition;
        }
    createShell(segments, 1);
    [...]
}
```

Code - Ausschnitt 26: Hinzufügen / Löschen neuer Dreiecke

Die Scanlinie für Segmente kann genau gleich implementiert werden. (Siehe im Source-Code am Ende der Arbeit.)

4 Andere Ansätze

4.1 Konzentrischer Algorithmus

Der konzentrische Algorithmus ist identisch mit dem graphentheoretischen Algorithmus mit Ausnahme des Füllens. In diesem Fall wird einfach die Aussenlinie, je nach Fülldichte, nach innen verschoben. Bei horizontalen Ebenen wird die Fülldichte gänzlich klein. Der Algorithmus ist langsamer: $\Theta(Q * Q * \log N * \log \log N)$ und $\mathcal{O}(Q * Q * N * \log N)$ oder genauer $\Theta(Q^{D-1} * \log N * \log \log N)$ und $\mathcal{O}(Q^{D-1} * N * \log N)$. Anstelle der Scanlinie der Segmente, wird hier um Q-

Schichten (Breite des Objekts) jeder Punkt einmal verschoben und durchfahren. Die Berechnung der Punktverschiebung läuft auf $\mathcal{O}(N)$.

4.2 Kartesischer Algorithmus

Der Kartesische Algorithmus erzeugt zuerst einen Q*Q*Q (oder genauer $\mathcal{Q}^D$) Container. Für jedes Dreieck wird nun die längste Kante gewählt und dessen L Punkte mithilfe des Bresenham-Algorithmus[9] eingetragen. Jeder dieser L Punkte wird nun mit dem Punkt gegenüber der längsten Kante verbunden, wieder mithilfe des Bresenham-Algorithmus. Die resultierenden Punkte werden in den Q*Q*Q Container abgespeichert mit dem Zahlenwert 1. Danach wird das Innere des Objekts mit 2 gefüllt. So kann man für Elemente mit der Zahl 2 einen Füllalgorithmus verwenden und mit den Elementen der Zahl 1 eine Hülle konstruieren. Den Rest kann man ganz normal, wie beim graphentheoretischen Algorithmus, ohne Optimierung implementieren. Dieser Algorithmus ist der Langsamste, aber für mich der Verständlichste. Die asymptotische Laufzeit beträgt: $\mathcal{O}(Q * Q * Q + N * L)$ oder genauer $\mathcal{O}(Q^D + N * L^{D-2})$.
Q^D ist das kartesische Gitter und $N * L^{D-2}$ ist die Anzahl von Linien die mit dem Bresenham-Algorithmus gezeichnet werden müssen.

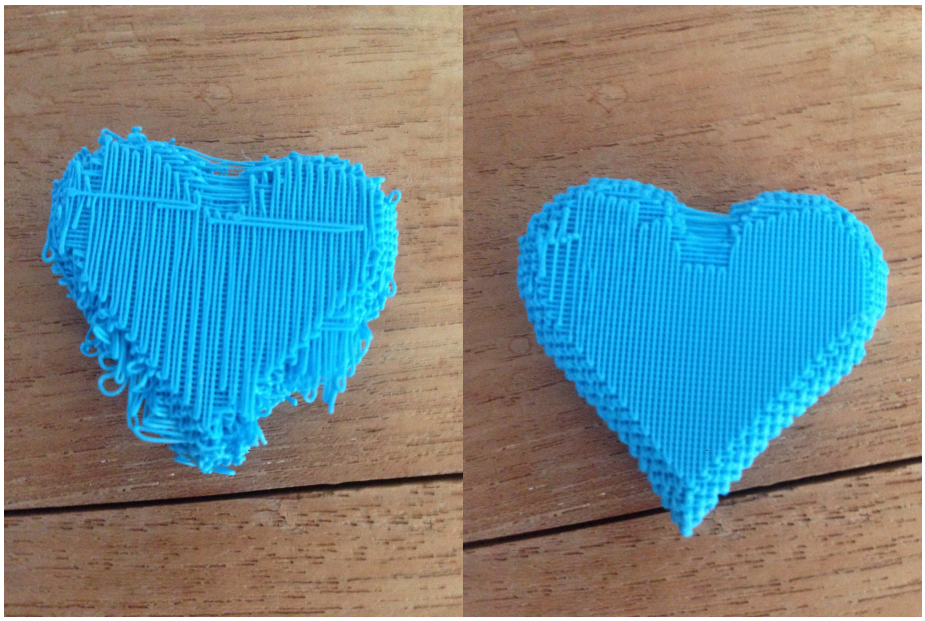
4.3 Hyperdimensionale Drucker

Ein 2D-Drucker verwendet Segmente, ein 3D-Drucker verwendet Dreiecke, ein 4D-Drucker verwendet Tetraeder als Eingabedatei. Alle besprochenen Algorithmen können für D-dimensionale Drucker erweitert werden.
Das physikalische Problem der 3D-Drucker auf der Erde, ist die Gravitationskraft. Überhängende Schichten sind nicht mit einem Plastikdrucker druckbar, da das Plastik beim auftragen noch flüssig ist und auf den Boden tropft. Auf der ISS gibt es bereits einen 3D-Drucker, welcher nicht mit dieser Einschränkung leben muss. Ohne physikalische Störkräfte, lassen sich Plastikdrucker problemlos in N-dimensionalen Welten bauen.

5 Erfahrungen & Anwendung

Was habe ich aus dieser praktischen Arbeit mitgenommen? Hauptsächlich das Fehlertesten (und Fehler gibt es fast immer bei einem Projekt dieser Grösse) mit Hardware sehr schwierig ist. Man konnte nur eine Veränderung am Tag vornehmen, da es sehr lange dauerte zum Drucken. So kam es, dass man mit sehr wenig Information einen Fehler beheben musste. Zum Beispiel habe ich in einem Fall die Koordinaten eines vorherigen Punktes übernommen:

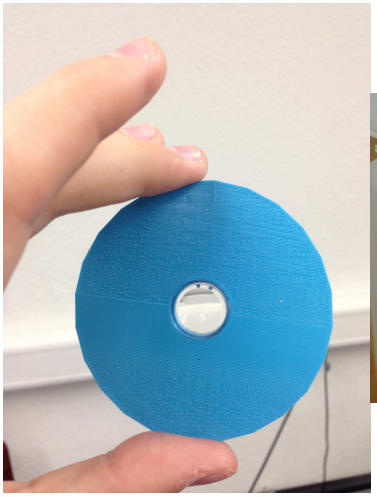
```
(p2.x, p2.y, p1.z)
```



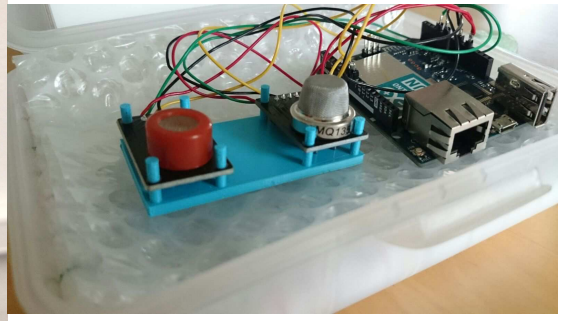
(a) Herz mit Fehler (b) Herz ohne Fehler

Abbildung 22: Fehlerbehebung des Punktevertauschs

So hat ein verkehrtes Zeichen in diesen 1'000 Zeilen grosse Konsequenzen.
Zusätzlich sind mir Vorteile eines eigenen Slicers aufgefallen. Viele spezielle Fälle, wie überhängende Stellen, kann man mit einem eigenen Slicer separat behandeln. Wenn der G-Code einmal stimmt, läuft er immer gleich und man kann eine Figur massenweise produzieren. In der Zukunft, wenn 3D-Drucker in der Produktion verwendet werden, kann ich nur empfehlen, einen Spezialisten anzuheuern um den G-Code zu designen.
Der Slicer wurde auch für nützliche Projekte eingesetzt, so wurde zum Beispiel ein Linsenhalter für Hr. Tino Lorenz entwickelt, eine Stütze für Sensoren eines Mikrocontrollers und die Hülle eines Raspberry Pi's.



(a) Linsenhalter



(b) Sensorstütze

Abbildung 23: Linsenhalter & Sensorstütze

Literatur

[1] https://en.wikipedia.org/wiki/Chuck_Hull

[2] http://www.epo.org/learning-events/european-inventor/finalists/2014/hull.html

[3] Delaunay, Boris: Sur la sphère vide. A la mémoire de Georges Voronoï. 1934

[4] P. Su and R.L.S. Drysdale: A Comparison of Sequential Delaunay Triangulation Algorithms. 1996

[5] EIA Standard RS-274-D Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines, 2001 Eye Street, NW, Washington, D.C. 20006: Electronic Industries Association, February 1979

[6] Bouknight W.J, An Improved Procedure for Generation of Half-tone Computer Graphics Representation. 1969

[7] The Raycasting-Algorithm presented is based on the Jordan curve[8] theorem and has no real inventor, since it is termed obvious.

[8] Jordan, Camille: Cours d'analyse. 1887

[9] J. E. Bresenham: Algorithm for computer control of a digital plotter. 1965

[10] M. Garland and P. Heckbert: Surface Simplification Using Quadric Error Metrics. In Proceedings of SIGGRAPH 97.

[11] C++ Library: OpenFrameworks (http://www.openframeworks.cc)

[12] Dieses Bild wurde von diesem Artikel genommen: http://www.3ders.org/articles/20140506-charles-hull-nominated-for-the-european-inventor-award.html Der genaue Ort des Bildes ist: http://www.3ders.org/images/european-invention-award-2.jpg

[13] Das Struktogramm wurde mithilfe folgender Webseite generiert: http://stukimania.hu/en/

[14] Clipper, eine Polygon-Clipping Bibliothek für C++: http://www.angusj.com/delphi/clipper.php basiert auf Vatti's clipping Algorithmus O(N log N).

A Schweizer Jugend forscht

Abbildung 24: Schweizer Jugend Forscht 2016

Die ganze Arbeit oberhalb dieser Sektion war Teil meiner Maturaarbeit, in den folgenden Sektionen werden neue Optimierungen besprochen.
Ich hatte das Glück, dass mir Daniel Eisenbarth als Betreuer für SJf zugeteilt wurde. Nach ersten Emails, haben wir uns schliesslich an der Vorrunde der SJf persönlich kennengelernt, wo wir die Arbeit besprachen und das weitere Vorgehen besprachen. Ich habe mich auch mit den Personen von Slic3r verständigt und sie haben mich auf eine Stelle in ihrem Programmcode hingewiesen, welche die Umwandlung von 3-dimensionalen Meshs auf Q 2-dimensionale Graphen in linearer Laufzeit fertigten. Ihre Idee habe ich später weitergesponnen und auch auf die 1-dimensionalen Slices angewendet mit welcher mein Programm nun eine asymptotische Laufzeitkomplexität von der Anzahl Druckbewegungen besitzt.
Daniel Eisenbarth hat mir später Verbesserungsvorschläge gemailt, zum einen sollte ich den Algorithmus grafisch vereinfacht darstellen, ich habe mich für ein Nassi-Schneider Diagramm und ein Flowchart entschieden. Weiter sollte ich verschiedene, möglichst unterschiedliche Objekte slicen um die Laufzeit der Programme Slic3r und KISSlicer mit meinem zu vergleichen. Diese Diagramme und Laufzeiten belaufen sich nun auf das neue Programm.
Des weiteren habe ich die Parameter geändert um noch genauer drucken zu können, die Eingabe / Ausgabe beschleunigt und den Programmcode generell verkürzt.

A.1 Der neue Sliceralgorithmus

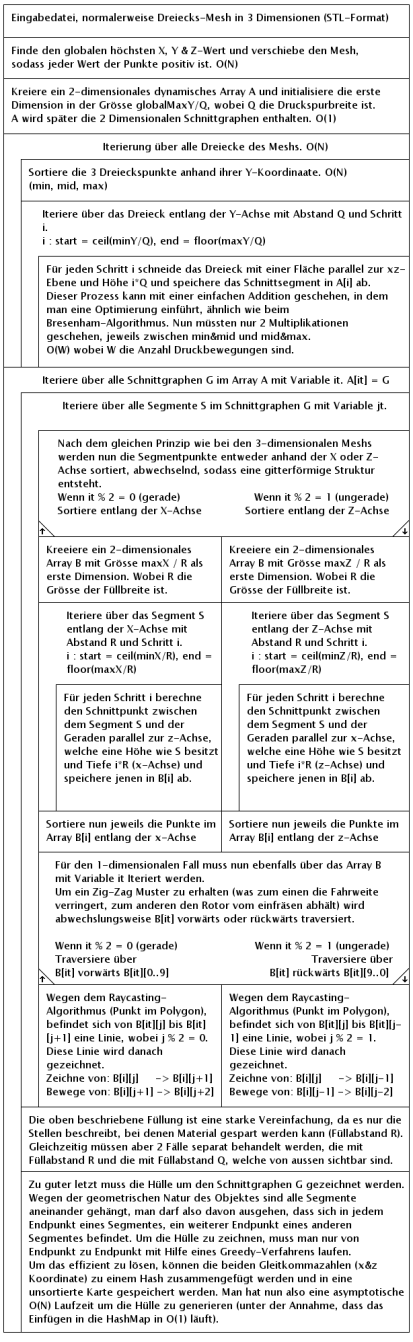


Abbildung 25: Nassi-Shneider Diagramm[13] (auch Struktogramm genannt)

A.1.1 Slicen der Dreiecke in linearer Laufzeit

Als erstes muss immer der Dreiecks-Mesh in Q Schnittebenen aufgeteilt werden, wobei Q für die Druckbreite steht (der Abstand zwischen den Schnittebenen). Die Schnittebenen lassen sich als 2-dimensionalen planaren Graphen darstellen. Im neuen Algorithmus wird hierbei der Array, welcher später die planaren Graphen beinhalten wird, schon initialisiert mit Grösse $globalMaxY/Q$, wobei $globalMaxY$ der höchste Punkt des Meshs darstellt.
Nun wird zuerst durch alle Dreiecke iteriert und bei jedem Dreieck die Schnittebene bestimmt, welche das Dreieck am tiefsten schneidet (mit der kleinsten Y-Koordinate). Diese Y-Koordinate ist $ceil(lowY/Q)$. Nun kann das Schnittsegment zum planaren Graphen mit der selben Höhe hinzugefügt werden.

```
pair<vector<bool>, vector<vector<Segment> > > triangleSlicing(vector<
    Triangle> triangles) {
    //from 0 to settings.sizeY
    //settings.layerThickness
    settings.layerThickness *= 1.000011111; //This weird constant is
        used in order to avoid edges
    vector<vector<Segment> > segments (settings.sizeY / settings.
        layerThickness);
    vector<bool> flat(settings.sizeY / settings.layerThickness, false);
    for(Triangle triangle : triangles) {
        int slicePosition = triangle.ref1.y / settings.layerThickness;
        if(fmod(triangle.ref1.y, settings.layerThickness) != 0.) ++
            slicePosition;

        for(int i = slicePosition; settings.layerThickness * i <
            triangle.ref3.y; ++i)
            segments[i].push_back(getCrossLine(triangle.ref1, triangle.
                ref2, triangle.ref3, settings.layerThickness * i));

        float maxX = max(max(triangle.ref1.x, triangle.ref2.x),
            triangle.ref3.x);
        float minX = min(min(triangle.ref1.x, triangle.ref2.x),
            triangle.ref3.x);
        float maxZ = max(max(triangle.ref1.z, triangle.ref2.z),
            triangle.ref3.z);
        float minZ = min(min(triangle.ref1.z, triangle.ref2.z),
            triangle.ref3.z);


        if((triangle.ref3.y - triangle.ref1.y) / max(maxX - minX, maxZ
            - minZ) < settings.angleForFilling) {
            if(slicePosition >= 1) flat[slicePosition-1] = true;
            flat[slicePosition] = true;
            if(slicePosition + 1 < flat.size()) flat[slicePosition+1] =
                 true;
        }
    }
    float yPos = settings.pointAbovePlate;
    for(auto & segList : segments) {
        for(auto & segment : segList) {
            segment.ref1.y = yPos;
            segment.ref2.y = yPos;
        }
        yPos += settings.layerThickness;
    }
    pair<vector<bool>, vector<vector<Segment> > > output;
    output.first = flat;
```

```
    output.second = segments;
    return output;
}
```

Code - Ausschnitt 27: Dreieckslicer mit linearer Laufzeit

Um die Division/Multiplikation bei der Bestimmung des Schnittsegments vorzubeugen, könnte die Tatsache genutzt werden, das die Grösse des Schnittsegments linear zu- (lowY bis midY), bzw. abnimmt (midY bis topY), sowie es der Bresenham-Algorithmus [9] bei der Zeichnung von Linien macht. Dies habe ich aber nicht implementiert, da es nur theoretisch von Interesse ist. (Ich vertraue darauf, dass die Multiplikationen zweier Gleitkommazahlen in konstanter Laufzeit implementiert wurde). Unter dieser Annahme sollte die Laufzeit identisch mit der Grösse des Arrays sein, welche diese planaren Graphen speichert, da keine zusätzlichen Operationen verwendet wurden (also eigentlich optimal).
In meinem Programm musste ich noch die Steigungswinkel der Dreiecke berücksichtigen und flache Dreiecke (mit geringem Winkel zur xz-Ebene) speziell behandeln.

A.1.2 Slicen der Segmente in linearer Laufzeit

Für jeden planaren Graph mussten nun die Schnittpunkte mit der gitterförmigen Struktur berechnet werden. Slic3r verwendet hierzu eine Polygon-Clipping Bibliothek namens Clipper [14], welcher auf Vatti's Clipping Algorithmus basiert, welcher eigentlich nur eine kleine Veränderung zum Bentley-Ottmann Algorithmus ist (Laufzeit $O(MlogM)$), wobei M die Anzahl Kanten im Graphen addiert mit der Anzahl Gitterkanten ist.
Um hier wieder eine lineare Laufzeit hinzubekommen wird wie beim Dreieck-Slicen durch jedes Segment iteriert und die Schnittstelle mit einer Geraden parallel zur x oder z-Achse (und gleicher Höhe wie das zu schneidende Segment) in einen 2-dimensionales Array abgespeichert. Die Punklisten in dem Array werden danach in die z oder x-Achse sortiert und man kann jeweils vom Element $A[i][j]$ nach $A[i][j+1]$ eine Linie mit dem Drucker zeichnen, wenn j gerade ist. Zusätzlich muss abwechslungsweise rückwärts oder vorwärts iteriert werden um die Zig-Zag Optimierung zu behalten.

```
void segmentSlicingAndPrinting(vector<Segment> segments, bool xSort,
    float fillDist) {
    bool orientationOfSlicer = false;
    if(xSort) {
        vector<vector<Vector> > pointList (settings.sizeX / fillDist);
        for(Segment seg : segments) {
            if(seg.ref2.x < seg.ref1.x) swap(seg.ref1, seg.ref2);
            int slicePosition = seg.ref1.x / fillDist;
            if(fmod(seg.ref1.x, fillDist) != 0.) ++slicePosition;
            for(int i = slicePosition; fillDist * i < seg.ref2.x; ++i)
                pointList[i].push_back(crossPointX(seg.ref1, seg.ref2,
                    i * fillDist));
        }
        for(vector<Vector> points : pointList) {
            sort(points.begin(), points.end(), zComp);
            if(orientationOfSlicer) reverse(points.begin(), points.end
                ());
            for (int i = 0; i < points.size() && i < points.size() - 1;
                 i += 2) {
```

```
                if(orientationOfSlicer) {
                    points[i].z -= settings.layerThickness * 3; //
                        ERFAHRUNGSWERT
                    points[i+1].z += settings.layerThickness * 3;
                }
                else {
                    points[i].z += settings.layerThickness * 3;
                    points[i+1].z -= settings.layerThickness * 3;
                }
                moveTo(points[i]);
                printTo(points[i+1]);
            }
            orientationOfSlicer = !orientationOfSlicer;
        }
    } else {
        vector<vector<Vector> > pointList (settings.sizeZ / fillDist);
        for(Segment seg : segments) {
            if(seg.ref2.z < seg.ref1.z) swap(seg.ref1, seg.ref2);
            int slicePosition = seg.ref1.z / fillDist;
            if(fmod(seg.ref1.z, fillDist) != 0.) ++slicePosition;
            for(int i = slicePosition; fillDist * i < seg.ref2.z; ++i)
                pointList[i].push_back(crossPointZ(seg.ref1, seg.ref2,
                    i * fillDist));
        }
        for(vector<Vector> points : pointList) {
            sort(points.begin(), points.end(), xComp);
            if(orientationOfSlicer) reverse(points.begin(), points.end
                ());
            for (int i = 0; i < points.size() && i < points.size() - 1;
                 i += 2) {
                if(orientationOfSlicer) {
                    points[i].x -= settings.layerThickness * 3;
                    points[i+1].x += settings.layerThickness * 3;
                }
                else {
                    points[i].x += settings.layerThickness * 3;
                    points[i+1].x -= settings.layerThickness * 3;
                }
                if(abs(points[i].x - points[i+1].x) + abs(points[i].z -
                     points[i+1].z) >= settings.layerThickness * 7) {
                moveTo(points[i]);
                printTo(points[i+1]);
                }
            }
            orientationOfSlicer = !orientationOfSlicer;
        }
    }
}
```

Code - Ausschnitt 28: Segmentslicer in linearer Laufzeit

A.1.3 Hüllenkreeierung in linearer Laufzeit

Nun fehlt nur noch das lineare Zeichnen der Aussenhülle. Dabei muss einem eigentlich nur eine kleineres Lemma auffallen. Die Segmente des planaren Graphen sind aneinander gebunden, wobei es auch separate Zyklen geben kann. Jeder Endpunkt eines Segments teilt also diesen Endpunkt mit genau einem anderen Segment. Nun kann man eine intuitive lineare Greedy-Suche benutzen um ein Segment nach dem Nächsten zu drucken, also man beginnt bei einem

Startsegment (ich wähle das Segment, welches die kleinste X-Koordinate besitzt, wegen dem Pickeleffekt) und wandere dem Pfad der Segmente herunter. Sind alle Segmentkoordinaten exakt berechnet, könnte man hierfür eine Hash-Map verwenden mit durchschnittlicher konstanter Laufzeit zum Einfügen (also total $O(N)$), doch wegen Rundungsfehlern verwende ich hier eine gewöhnliche Map mit $O(\log N)$ Laufzeit zum Einfügen.
Das einfache Problem das nächste Segment zu finden, kann mit Rundungsfehlern auf einmal ein Nearest Neighbor Problem werden, doch auch dort gibt es die Methode des Locality-Sensitive Hashing. Ich habe mich aber wie oben erwähnt auf eine einfache Map gestützt und mit ihr immer auf das nächst höhere Objekt (in x, dann z Richtung) gestützt und es hat für mich in der Praxis nie Probleme gegeben.

```
void createLinearTimeShell(vector<Segment> segments)
{
    if(segments.size() > 2) {
        string originalFillRate = settings.fillRate;
        string originalMoveRate = settings.moveRate;
        settings.fillRate = settings.fillRateHull;
        settings.moveRate = settings.moveRateHull;

        set<pair<Vector, Vector> > points;
        pair<Vector, Vector> currentSegment(pair<Vector, Vector>(Vector
            (settings.sizeX * 2, 0, 0), Vector(0,0,0)));
        for(auto s : segments) {
            //Every point has one connection for greedy algorithm
            points.insert(pair<Vector, Vector>(s.ref1, s.ref2));
            points.insert(pair<Vector, Vector>(s.ref2, s.ref1));
            //Pickle effect => therefor find point with smallest x axis
            if(s.ref1.x < currentSegment.first.x) {
                currentSegment.first = s.ref1;
                currentSegment.second = s.ref2;
            }
            if(s.ref2.x < currentSegment.first.x) {
                currentSegment.first = s.ref2;
                currentSegment.second = s.ref1;
            }
        }

        Vector smallestVector(-settings.sizeX, currentSegment.first.y,
            -settings.sizeZ);
        while(points.size() != 0) {
            set<pair<Vector, Vector> >::iterator it = points.
                lower_bound(currentSegment);
            swap(currentSegment.first, currentSegment.second);
            moveTo(it->first);
            printTo(it->second);
            points.erase(it);
            points.erase(currentSegment);
            set<pair<Vector, Vector> >::iterator nextIt = points.
                lower_bound(pair<Vector, Vector>(currentSegment.first,
                smallestVector));
            if(nextIt == points.end()) nextIt = points.begin();

            currentSegment = *nextIt;
        }

        settings.fillRate = originalFillRate; //3600
        settings.moveRate = originalMoveRate;
    }
```

}

Code - Ausschnitt 29: Hüllenkreeierung in linearer Laufzeit

A.1.4 Mögliche Erweiterungen

Einige kommerzielle Slicerprogramme haben die Möglichkeit de Grösse der Eingabedatei zu schrumpfen, in dem sie Gebiete mit besonders kleinen Dreiecken zu grösseren Dreiecken vereinen, zum Beispiel mit dem Garland-Heckbert Algorithmus[10]. Dies kann vor allem beim 2D-Slicen deutlich Zeit einsparen.
Auch die Generalisierung des neuen Algorithmus im hyperdimensionalen Raum ist möglich, ähnlich wie bei dem Scanline-Algorithmus (Siehe 4.3). Auch hier muss nur das Slicen auf eine beliebige Dimension angewandt werden.

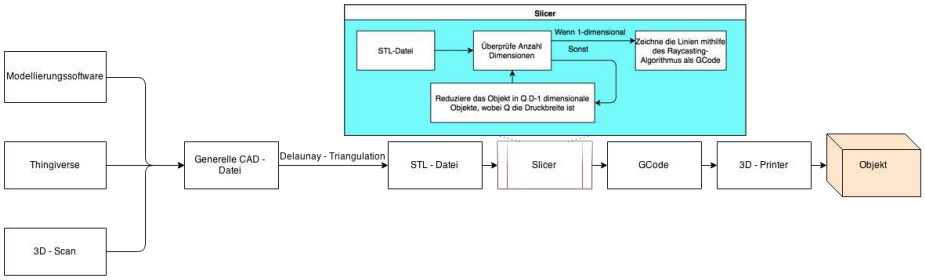


Abbildung 26: Flowchart über die Funktion des Slicers

A.2 Benchmarking

Für das Benchmarking habe ich mein Programm nicht mit Debug Unterstützung (ohne gdb z.B.) kompiliert um eine fairen Vergleich zu machen, dies fördert schliesslich auch die Geschwindigkeit. Um die Laufzeit zu testen waren verschiedene STL-Dateien vonnöten, welche ich von thingiverse `http://www.thingiverse.com` (Eine Plattform um 3D Objekte mit der Welt zu teilen) heruntergeladen habe. Die benutzten Objekte sind jeweils verlinkt, doch da die meisten binäre STL-Dateien verwendeten und ich mich für ASCII-STL Dateien entschieden habe(Siehe 1.2), habe ich die Objekte schnell mit einem Ruby-Skript von GitHub konvertiert `https://github.com/cmpolis/convertSTL`. Die Programme liefen alle unter den selben Bedingungen, namentlich auf einem Mac Book Air(13-Inch, Mid 2013) mit 1.3 GHz und 8 GB RAM. Auf dem Betriebssystem liefen keine weiteren Programme. Die Zeit musste ich für Slic3r von Hand messen und bei meinem Programm konnte ich den exakten Wert bestimmen. Das ich es von Hand gemessen habe, ist aber nicht so schlimm, da ein paar Drucke über eine Minute dauerten.
Für den 3D-Druck habe ich das Rectilinear Füllsequenz genommen, da mein Slic3r bis jetzt keine andere genutzt hat. Als Füllabstand habe ich 5 mm gewählt, dies entspricht der Fülldichte von etwa 25% bei Slicer unter diesen Einstellungen. Anhand des Ladebalkens konnte ich sehen wie weit sowohl Slic3r als auch KISSlicer waren und konnte so bestimmen in welchem Stadium die Slic3r sich

befanden (im Import, Slicen oder Export). Dabei habe ich folgende Einstellungen verwendet für alle Slic3r:

```
settings.layerThickness = 0.2;
settings.fillThickness = 0.1; //usually half of settings.
    layerThickness
settings.countOfPerimeters = 2;
```

Code - Ausschnitt 30: Mittlere Qualität

A.2.1 Testobjekt 1 10cm gross

Voronoi Experiment No. 5 http://www.thingiverse.com/thing:241367 hat folgende Spezifikationen:

- Dimensionen: (56.38mm, 99.98mm, 53.31mm)(x/y/z)
- Anzahl Dreiecke: 197'832
- Anzahl Knoten: 98'29

Voronoi Experiment No. 5 (10cm)	Laufzeit ohne Output	Laufzeit mit Output
Slicer für SJf	18.17912(s)	22.5363(s)
Slic3r 1.2.9	1:02 (min)	6:06 (min)
KISSlicer 1.4.5.10	56(s)	1:04 (min)

A.2.2 Testobjekt 1 22cm gross

Voronoi Experiment No. 5 http://www.thingiverse.com/thing:241367 hat folgende Spezifikationen:

- Dimensionen: (124.034mm, 219.999mm, 117.28mm)(x/y/z)
- Anzahl Dreiecke: 197'832
- Anzahl Knoten: 98'29

Voronoi Experiment No. 5 (22cm)	Laufzeit ohne Output	Laufzeit mit Output
Slicer für SJf	35.23242(s)	43.8335(s)
Slic3r 1.2.9	2:29 (min)	14:24 (min)
KISSlicer 1.4.5.10	2:27(min)	2:45 (min)

A.2.3 Testobjekt 2 10cm gross

Natürlich musste ich auch die berühmte Utah Teekanne drucken, ich habe mich für eine Teekanne mit niedriger Auflösung entschieden um auch einmal ein kleine Datei zu testen.
Little tea pot http://www.thingiverse.com/thing:460243 hat folgende Spezifikationen:

- Dimensionen: (124.034mm, 219.999mm, 117.28mm)(x/y/z)
- Anzahl Dreiecke: 1'076

- Anzahl Knoten: 538

Utah Teekanne (10cm)	Laufzeit ohne Output	Laufzeit mit Output
Slicer für SJf	0.625937(s)	0.491979(s)
Slic3r 1.2.9	3(s)	7(s)
KISSlicer 1.4.5.10	9(s)	10(s)

A.2.4 Testobjekt 2 10cm gross

Natürlich musste ich auch die berühmte Utah Teekanne drucken, ich habe mich für eine Teekanne mit niedriger Auflösung entschieden um auch einmal ein kleine Datei zu testen.
Little tea pot http://www.thingiverse.com/thing:460243 hat folgende Spezifikationen:

- Dimensionen: (220.001mm, 102.669mm, 136.904mm)(x/y/z)
- Anzahl Dreiecke: 1'076
- Anzahl Knoten: 538

Utah Teekanne (22cm)	Laufzeit ohne Output	Laufzeit mit Output
Slicer für SJf	1.296694(s)	1.72101(s)
Slic3r 1.2.9	5(s)	12(s)
KISSlicer 1.4.5.10	1:01(min)	1:05(min)

A.2.5 Testobjekt 3 22cm

Das letzte Objekt habe ich absichtlich sehr gross gewählt, es handelt sich um die eingescannte Figur Nike von Samothrake http://www.thingiverse.com/thing:196038. Der eigentliche Algorithmus hat hier nur 1:20 Minuten gedauert, doch das Einlesen einer solch grossen Datei hat noch viel mehr Zeit gekostet. Die konvertierte ASCII-STL Datei hatte immerhin eine Grösse von 500 MB!

- Dimensionen: (142.826mm, 219.999mm, 159.834mm)(x/y/z)
- Anzahl Dreiecke: 1'960'102
- Anzahl Knoten: 980'051

Nike von Samothrake (22cm)	Laufzeit ohne Output	Laufzeit mit Output
Slicer für SJf	2:43.0179(min)	3:04.993 (min)
Slic3r 1.2.9	5:53(min)	zu Lange
KISSlicer 1.4.5.10	7:58(min)	8:45(min)

A.2.6 Analyse der Daten

Die anderen Programme sind bei diesem Test natürlich im Nachteil, obwohl ich versucht habe die Tests so fair wie nur möglich zu machen. Auch Slic3r ist in

C++ programmiert, die Codebasis von KISSlicer konnte ich nicht nachschauen (Close-Source), aber sie verwenden beide Rep-Rap Entwicklertools zum slicen.
Sehr erstaunlich war es, dass Slic3r laut seinem Ladebalken hatte er immer ungefähr so lange Zeit zum slicen, wie es KISSlicer brauchte, doch für den Output brauchte es immer immens lange. Ich spekuliere, dass dies einer Fehlursache war und habe deswegen beide Zeiten angegeben, sowohl mit Ausgabe, als auch ohne Ausgabe. Erstaunlicherweise war die Ausgabe von Slic3r immer halb so gross oder noch kleiner wie der Output meines Slicers und dem von KISSlicer, vielleicht hat er noch irgend ein Kompressionsverfahren verwendet, obwohl ich nichts dergleichen gesehen habe.
Etwas weiteres was ich mir nicht erklären kann, ist warum der KISSlicer für die Teekanne ungefähr 1 Minute braucht zum slicen, 10x länger als das vorherige halb so grosse Objekt.
Übrigens stellt sowohl meine Slicersoftware, als auch die anderen Programme das Objekt in 3D dar, dass ist in der Laufzeit also vermerkt.

B Source-Code

In diesem Abschnitt ist der gesamte Source-Code noch einmal aufgelistet, inklusive der graphischen Oberfläche und dessen Einstellungen. Um das Projekt selber kompilieren zu können braucht man OpenFrameworks und einen Compiler, welcher C++ 98 unterstützt. Wenn die Schweizer Jugend forscht bestanden ist, sollte der Source-Code auf: http://teegabel.com verfügbar sein.

```
//
//   Created by De Keyser Kevin.
//

//OPENFRAMEWORKS
#include "ofMain.h"

class ofApp : public ofBaseApp{

public:
    void setup();
    void update();
    void draw();

    void keyPressed(int key);
    void keyReleased(int key);
    void mouseMoved(int x, int y );
    void mouseDragged(int x, int y, int button);
    void mousePressed(int x, int y, int button);
    void mouseReleased(int x, int y, int button);
    void windowResized(int w, int h);
    void dragEvent(ofDragInfo dragInfo);
    void gotMessage(ofMessage msg);

};

#define MAX_INT 999999
#define MAX_FLOAT 999999999999999

//LIBRARIES
#include <algorithm>
#include <cctype>
#include <cmath>
#include <fstream>
#include <functional>
#include <iostream>
#include <locale>
#include <set>
#include <string>
#include <vector>
#include <map>

using namespace std;

//GLOBALS
ofTrueTypeFont font;
enum MODES { FILE_SELECT, DRAW };
MODES mode = FILE_SELECT;
ofEasyCam easyCam;

ofMesh graphicsMesh;
ofLight lightSource;

vector<string> gcode;

float previousX = 0, previousY = 0, previousZ = 0, previousExtrusion = 0;
float transX = 20, transZ = 20;

string name = "";

struct Settings {
    float sizeX = 240, sizeY = 240, sizeZ = 240;
```

```
    /*
     FOR HIGH QUALITY:
     float layerThickness = 0.1;
     float fillThickness = 0.05;
     UND BLAUER STICKER DRAUF!! -> WICHTIG, sonst pointAbovePlate funktioniert
          nicht
     */
    /*
     FOR NORMAL QUALITY:
     float layerThickness = 0.2;
     float fillThickness = 0.1;
     UND BLAUER STICKER DRAUF!! -> WICHTIG, sonst pointAbovePlate funktioniert
          nicht
     */
    bool doFill = true;
    bool safeMaterial = true;
    bool displayApproximateTimeConsumption = true;
    bool quadraticTimeSearch = false; //This is for files, that have floating
         point errors.
    int amountOfLinearSearches = 5;
    int localitySensitivityHashing = 3; // Mindestens grösse 3
    float layerThickness = 0.2; //0.2 for Ultimaker2, 10 for imaging
    float fillThickness = 0.1; //= 0.05;
    float pointAbovePlate = -0.05;
    float fillSpace = 5.0;
    int fillAlgorithm = 1;
    float baseHeight = 1.0;
    bool fanOn = true;
    bool addRetraction = true;
    float retractionDist = 4.0; //INCREADIBLY IMPORTANT
    bool wireHotspotFilter = false;
    float angleForFilling = 1./5.;
    string fillRate = "2400"; //3600 (2400)
    string moveRate = "9000"; //9000
    string fillRateHull = "1200";
    string moveRateHull = "4800";
} settings;

void drawStringCenter(string input, int x, int y) {
    font.drawString(input, x - (font.stringWidth(input) / 2), y /*- (font.
         stringHeight(input) / 2)*/ );
}

struct Vector
{
    float x, y, z;

    Vector() : x(0), y(0), z(0)
    {
        //EMPTY: Make x y z = 0
        //Calculate Normal
    }
    Vector(float _x, float _y, float _z) : x(_x), y(_y), z(_z)
    {

    }

    void setPrecision(int precision) //global precision
    {
        x = round(x * precision) / precision;
        y = round(y * precision) / precision;
        z = round(z * precision) / precision;
    }
};

bool operator<(const Vector& vec1, const Vector& vec2) {
    if(vec1.y != vec2.y) return vec1.y < vec2.y; //First sort by y coordinate, due
         to bottom up printing.
    else if(vec1.x != vec2.x) return vec1.x < vec2.x;
    return vec1.z < vec2.z;
}

bool xComp(const Vector& vec1, const Vector& vec2) {
    if(vec1.x != vec2.x) return vec1.x < vec2.x;
```

```
    if(vec1.y != vec2.y) return vec1.y < vec2.y;
    return vec1.z < vec2.z;
}

bool zComp(const Vector& vec1, const Vector& vec2) {
    if(vec1.z != vec2.z) return vec1.z < vec2.z;
    if(vec1.y != vec2.y) return vec1.y < vec2.y;
    return vec1.x < vec2.x;
}

bool operator!=(const Vector& vec1, const Vector& vec2) {
    return ! ( vec1 < vec2 && vec2 < vec1 );
}

Vector operator+(const Vector& vec1, const Vector& vec2) {
    Vector toReturn = vec1;
    toReturn.x += vec2.x;
    toReturn.y += vec2.y;
    toReturn.z += vec2.z;
    return toReturn;
}

struct Segment {
    Vector ref1, ref2;
    Segment(Vector& _ref1, Vector& _ref2) : ref1(_ref1), ref2(_ref2) {
        if(ref2 < ref1) swap(ref1, ref2);
    }
};

bool operator<(const Segment& seg1, const Segment& seg2) {
    if(seg1.ref1 < seg2.ref1) return seg1.ref1 < seg2.ref1;
    if(seg2.ref1 < seg1.ref1) return seg1.ref1 < seg2.ref1;
    return seg1.ref2 < seg2.ref2;

}

struct TriangleRef {
    int ref1, ref2, ref3;
    TriangleRef() : ref1(0), ref2(0), ref3(0) {}
    TriangleRef(int _ref1, int _ref2, int _ref3) : ref1(_ref1), ref2(_ref2), ref3(
        _ref3) {}
};

struct Triangle {
    Vector ref1, ref2, ref3;
    Triangle(Vector& _ref1, Vector& _ref2, Vector& _ref3) :
    ref1(_ref1), ref2(_ref2), ref3(_ref3) {
        if(ref3 < ref2) swap(ref2, ref3);
        if(ref2 < ref1) swap(ref1, ref2);
        if(ref3 < ref2) swap(ref2, ref3);
    }
};

bool operator<(const Triangle& tri1, const Triangle& tri2) {
    if(tri1.ref2 != tri2.ref2) return tri1.ref2 < tri2.ref2;
    if(tri1.ref1 != tri2.ref1) return tri1.ref1 < tri2.ref1;
    return tri1.ref3 < tri2.ref3;
}

struct Mesh
{
    //Creation of mesh is n log n  ==> Could be n using unordered_map with
        handcrafted hash function... BUT CAN BE IGNORED
    //Memory O(3n) => f(x) E O(n)
    map<Vector, int> isContained; // log(n)
    vector<Vector> points;
    vector<TriangleRef> triangles;

    Mesh() {}

    int addPoint (Vector&);
    void addTriangle (Vector&, Vector&, Vector&);
    void scale(float);
    void translate(Vector);
```

```
    pair<Vector, Vector> getCrossLine(int, float);
    //Segment getCrossLineNS(Triangle&, float);
    ofMesh parseGraphicsMesh();

} dataMesh;


int Mesh::addPoint(Vector &input)
{
    if (isContained.count(input) == 0)
    {
        isContained[input] = points.size();
        points.push_back(input);
        return points.size() - 1;
    }
    else
    {
        return isContained[input];
    }
}

void Mesh::addTriangle (Vector &a, Vector &b, Vector &c)
{
    TriangleRef toAdd;
    toAdd.ref1 = addPoint(a);
    toAdd.ref2 = addPoint(b);
    toAdd.ref3 = addPoint(c);

    //Sorting points
    if(points[toAdd.ref2] < points[toAdd.ref1]) swap(toAdd.ref1, toAdd.ref2);
    if(points[toAdd.ref3] < points[toAdd.ref2]) swap(toAdd.ref2, toAdd.ref3);
    if(points[toAdd.ref2] < points[toAdd.ref1]) swap(toAdd.ref1, toAdd.ref2);

    triangles.push_back(toAdd);
}

void Mesh::scale(float pos)
{
    for(Vector &vertices : points)
    {
        vertices.x *= pos;
        vertices.y *= pos;
        vertices.z *= pos;
    }
}

void Mesh::translate(Vector pos)
{
    for(int i = 0; i < points.size(); i++)
    {
        points[i].x += pos.x;
        points[i].y += pos.y;
        points[i].z += pos.z;
    }
}

//! ATTENTION FOR DIVISION BY ZERO. CHECK IF CORRECT
Vector crossPoint(Vector&lower, Vector &upper, float yPos)
{
    Vector output;
    if(upper.y < lower.y) cout << "ERROR:␣136" << endl;
    if(upper.y - lower.y < 0.1) {return lower; //Important!
        //cout << "Anomaly: 123 " << yPos << endl;
    }
    float diff = (yPos - lower.y) / (upper.y - lower.y);
    output.y = yPos;
    output.x = (diff * (upper.x - lower.x)) + lower.x;
    output.z = (diff * (upper.z - lower.z)) + lower.z;
    return output;
}

pair<Vector, Vector> Mesh::getCrossLine(int reference, float yPos)
{
    //ref1 & ref3
```

```
    pair<Vector, Vector> output;
    output.first = crossPoint(points[ triangles[reference].ref1 ], points[
         triangles[reference].ref3 ], yPos);

    if(yPos < points[ triangles[reference].ref2 ].y) {
        output.second = crossPoint(points[ triangles[reference].ref1 ], points[
            triangles[reference].ref2 ], yPos);
    }
    else {
        output.second = crossPoint(points[ triangles[reference].ref2 ], points[
            triangles[reference].ref3], yPos);
    }
    return output;
}

void calcNormals(ofMesh &mesh);
ofMesh Mesh::parseGraphicsMesh()
{
    ofMesh toReturn;

    if(settings.wireHotspotFilter)
    {
        //Hotspot Wireframe Filtering
        vector<int> colorFilter (isContained.size(), 0);

        for(TriangleRef &triangle : triangles)
        {
            colorFilter[triangle.ref1]++;
            colorFilter[triangle.ref2]++;
            colorFilter[triangle.ref3]++;
        }

        for(int i = 0; i < points.size(); i++)
        {
            toReturn.addVertex(ofPoint(points[i].x, points[i].y, points[i].z));
            ofColor toFill;
            toFill.set(min(colorFilter[i] * 20, 255), min(colorFilter[i] * 20,
                255), max(255 - (colorFilter[i] * 20), 0));
            toReturn.addColor(toFill);
        }

    }
    else {
        //No Filtering
        for(Vector &vertices : points)
        {
            toReturn.addVertex(ofPoint(vertices.x, vertices.y, vertices.z));
        }
    }

    for(TriangleRef &triangle : triangles)
    {
        toReturn.addIndex(triangle.ref1);
        toReturn.addIndex(triangle.ref2);
        toReturn.addIndex(triangle.ref3);
    }
    calcNormals(toReturn);
    return toReturn;
}


//From Zach ofxMeshUtils
//ofZach/ofxMeshUtils/blob/master/src/ofxMeshUtils.cpp#L32-L58
void calcNormals(ofMesh &mesh) {
    //cout << triangles[0].ref1;
    for( int i=0; i < mesh.getVertices().size(); i++ ) mesh.addNormal(ofPoint
         (0,0,0));

    for( int i=0; i < mesh.getIndices().size(); i+=3 ){
        const int ia = mesh.getIndices()[i];
        const int ib = mesh.getIndices()[i+1];
        const int ic = mesh.getIndices()[i+2];

        ofVec3f e1 = mesh.getVertices()[ia] - mesh.getVertices()[ib];
```

```
        ofVec3f e2 = mesh.getVertices()[ic] - mesh.getVertices()[ib];
        ofVec3f no = e2.cross( e1 );

        // depending on your clockwise / winding order, you might want to reverse
             the e2 / e1 above if your normals are flipped.

        mesh.getNormals()[ia] += no;
        mesh.getNormals()[ib] += no;
        mesh.getNormals()[ic] += no;
    }
}

void moveTo(float x, float y, float z)
{
    z += settings.pointAbovePlate;
    x += transX;
    y += transZ;

    float distX = abs(x - previousX);
    float distY = abs(y - previousY);
    float distZ = abs(z - previousZ);

    float approxDist = distX + distY + distZ;

    //SAFETY JUST IN CASE
    if(x < 0 || y < 0 || z < 0 || x > settings.sizeX || y > settings.sizeY || z >
         settings.sizeZ)
    {
        //cout << "Warning, moves to: " << x << " " << y << " " << z << endl;
    }

    string output = "G0␣F" + settings.moveRate + "␣X" + ofToString(x) + "␣Y" +
         ofToString(y) + "␣Z" + ofToString(z);

    if(settings.addRetraction && settings.retractionDist <= approxDist) gcode.
         push_back("G10");
    gcode.push_back(output);
    if(settings.addRetraction && settings.retractionDist <= approxDist) gcode.
         push_back("G11");

    previousX = x;
    previousY = y;
    previousZ = z;
}

void addPrintQueue(float x, float y, float z) {
    moveTo(x, y, z);
    //similar to line optimization
}

void moveTo(Vector temp)
{
    //Swap Z & Y Axis;
    addPrintQueue(temp.x, temp.z, temp.y);
}


void printTo(float x, float y, float z)
{
    z += settings.pointAbovePlate;
    x += transX;
    y += transZ;

    //SAFETY JUST IN CASE
    if(x < 0 || y < 0 || z < 0 || x > settings.sizeX || y > settings.sizeY || z >
         settings.sizeZ) {
        //cout << "Warning, moves to: " << x << " " << y << " " << z << endl;
    }


    float extrusion = 0;

    float distX = abs(x - previousX);
    float distY = abs(y - previousY);
```

```
    float distZ = abs(z - previousZ);
    float distXY = sqrt(distX * distX + distY * distY);
    float dist =  sqrt(distXY * distXY + distZ * distZ);

    extrusion = dist * settings.fillThickness;

    string output = "G1 F" + settings.fillRate + " X" + ofToString(x) + " Y"  +
        ofToString(y) + " Z" + ofToString(z) + " E" + ofToString(extrusion +
        previousExtrusion);
    gcode.push_back(output);

    previousX = x;
    previousY = y;
    previousZ = z;
    previousExtrusion += extrusion;

}

void printTo(Vector temp)
{
    //Swap Z & Y Axis;
    printTo(temp.x, temp.z, temp.y);
}

void startCodes()
{
    if(gcode.size() != 0) cout << "Warning, output hasn't been cleared." << endl;
    gcode.push_back(";FLAVOR:UltiGCode");
    gcode.push_back(";TIME:0");
    gcode.push_back(";MATERIAL:9999999999999999");
    gcode.push_back(";MATERIAL2:0");
    gcode.push_back("");
    gcode.push_back(";Layer count:0");
    if(settings.fanOn) gcode.push_back("M106 S255");
    else gcode.push_back("M106 S0");

    moveTo(20 - transX,   20 - transZ,  0);
    printTo(settings.sizeX - 20 - transX, 20 - transZ,  0);
    printTo(settings.sizeX - 20 - transX, settings.sizeY - 20 - transZ, 0);
    printTo(20 - transX, settings.sizeY - 20 - transZ, 0);
    printTo(20 - transX,  20 - transZ,  settings.pointAbovePlate);
}

void endCodes()
{
    gcode.push_back("G10");
    gcode.push_back("M106 S0");
    gcode.push_back("M25");
}

float distanceY(Vector& a, Vector& b) {
    //apporximate
    float distX = abs(a.x - b.x);
    float distZ = abs(a.z - b.z);
    return sqrt(distX * distX + distZ * distZ);
}


Vector crossPointY(Vector& lower, Vector& upper, float yPos) {
    Vector output;
    float diff = (yPos - lower.y) / (upper.y - lower.y);
    output.y = yPos;
    output.x = (diff * (upper.x - lower.x)) + lower.x;
    output.z = (diff * (upper.z - lower.z)) + lower.z;
    return output;
}

Vector crossPointX(Vector& lower, Vector& upper, float xPos) {
    Vector output;
    float diff = (xPos - lower.x) / (upper.x - lower.x);
    output.x = xPos;
    output.y = lower.y; //previously determined
    output.z = (diff * (upper.z - lower.z)) + lower.z;
    return output;
```

```
}

Vector crossPointZ(Vector& lower, Vector& upper, float zPos) {
    Vector output;
    float diff = (zPos - lower.z) / (upper.z - lower.z);
    output.z = zPos;
    output.y = lower.y; //previously determined
    output.x = (diff * (upper.x - lower.x)) + lower.x;
    return output;
}

Segment getCrossLine(Vector& lower, Vector& middle, Vector& upper, float yPos) {
    Vector first = crossPointY(lower, upper, yPos);
    Vector second;
    if(yPos < middle.y) {
        //lower & middle
        if(middle.y - lower.y < 0.2) second = middle;
        else second = crossPointY(lower, middle, yPos);
    }
    else {
        //middle & upper
        if(upper.y - middle.y < 0.2) second = middle;
        second = crossPointY(middle, upper, yPos);
    }

    Segment output(first, second);
    return output;
}

//OLD QUADRATIC CODE

bool imaging = false;
int imageCounter = 0;
void createShell(vector<Segment>& segments, float scale)
{
    settings.fillRate = "1200";
    settings.moveRate = "4800";

    if(scale == 1 && segments.size() > 0) {
        vector<bool> used(segments.size(), 0);

        //OH NO O(N^2)
        bool done = false;
        bool position = false;
        int i = 0;
        //antipickel mode
        Vector positionFrom(settings.sizeX/2, 0, 0);
        while(!done) {

            float dist = MAX_FLOAT;
            for(int j = 0; j < segments.size(); ++j) {
                if(!used[j]) {
                    if(distanceY(positionFrom, segments[j].ref1) < dist) {
                        dist = distanceY(positionFrom, segments[j].ref1);
                        position = true;
                        i = j;
                    }
                    if(distanceY(positionFrom, segments[j].ref2) < dist) {
                        dist = distanceY(positionFrom, segments[j].ref2);
                        position = false;
                        i = j;
                    }
                }
                else if(j == segments.size() - 1 && dist == MAX_FLOAT) done = true
                    ;
            }
            used[i] = true;
            if(imaging) ofLine(segments[i].ref1.x*10, segments[i].ref1.z*10,
                 segments[i].ref2.x*10, segments[i].ref2.z*10);
            if(position) {
                moveTo(segments[i].ref1);
                printTo(segments[i].ref2);
                positionFrom = segments[i].ref2;
            }
```

```
            else {
                moveTo(segments[i].ref2);
                printTo(segments[i].ref1);
                positionFrom = segments[i].ref1;
            }
        }

        if(imaging) {
            ofImage entity;
            entity.grabScreen(0, 0, 1024, 768);
            entity.saveImage(ofToString(imageCounter) + "-print.png");
            ofBackground(255, 255, 255);
            for(Segment& segment : segments) ofLine(segment.ref1.x*10, segment.
                 ref1.z*10, segment.ref2.x*10, segment.ref2.z*10);

            entity.grabScreen(0, 0, 1024, 768);
            entity.saveImage(ofToString(imageCounter) + "-img.png");
            imageCounter++;
            ofBackground(255, 255, 255);
        }

    }
    settings.fillRate = "2400"; //3600
    settings.moveRate = "9000";
}


void createLinearTimeShell(vector<Segment> segments)
{
    if(segments.size() > 2) {
        string originalFillRate = settings.fillRate;
        string originalMoveRate = settings.moveRate;
        settings.fillRate = settings.fillRateHull;
        settings.moveRate = settings.moveRateHull;

        set<pair<Vector, Vector> > points;
        pair<Vector, Vector> currentSegment(pair<Vector, Vector>(Vector(settings.
             sizeX * 2, 0, 0), Vector(0,0,0)));
        for(auto s : segments) {
            //Every point has one connection for greedy algorithm
            points.insert(pair<Vector, Vector>(s.ref1, s.ref2));
            points.insert(pair<Vector, Vector>(s.ref2, s.ref1));
            //Pickle effect => therefor find point with smallest x axis
            if(s.ref1.x < currentSegment.first.x) {
                currentSegment.first = s.ref1;
                currentSegment.second = s.ref2;
            }
            if(s.ref2.x < currentSegment.first.x) {
                currentSegment.first = s.ref2;
                currentSegment.second = s.ref1;
            }
        }

        //set<pair<Vector, Vector> >::iterator jt, it = points.find({startVector,
             nextVector});
        Vector smallestVector(-settings.sizeX, currentSegment.first.y, -settings.
             sizeZ);
        while(points.size() != 0) {
            set<pair<Vector, Vector> >::iterator it = points.lower_bound(
                 currentSegment);
            swap(currentSegment.first, currentSegment.second);
            moveTo(it->first);
            printTo(it->second);
            points.erase(it);
            points.erase(currentSegment);
            set<pair<Vector, Vector> >::iterator nextIt = points.lower_bound(pair<
                 Vector, Vector>(currentSegment.first, smallestVector));
            if(nextIt == points.end()) nextIt = points.begin();

            currentSegment = *nextIt;
        }

        settings.fillRate = originalFillRate; //3600
        settings.moveRate = originalMoveRate;
```

```
    }
}



pair<vector<bool>, vector<vector<Segment> > > triangleSlicing(vector<Triangle>
     triangles) {
    //from 0 to settings.sizeY
    settings.layerThickness *= 1.000011111; //This weird constant is used in order
          to avoid edges
    vector<vector<Segment> > segments (settings.sizeY / settings.layerThickness);
    vector<bool> flat(settings.sizeY / settings.layerThickness, false);
    for(Triangle triangle : triangles) {
        int slicePosition = triangle.ref1.y / settings.layerThickness;
        if(fmod(triangle.ref1.y, settings.layerThickness) != 0.) ++slicePosition;

        for(int i = slicePosition; settings.layerThickness * i < triangle.ref3.y;
             ++i)
            segments[i].push_back(getCrossLine(triangle.ref1, triangle.ref2,
                 triangle.ref3, settings.layerThickness * i));

        float maxX = max(max(triangle.ref1.x, triangle.ref2.x), triangle.ref3.x);
        float minX = min(min(triangle.ref1.x, triangle.ref2.x), triangle.ref3.x);
        float maxZ = max(max(triangle.ref1.z, triangle.ref2.z), triangle.ref3.z);
        float minZ = min(min(triangle.ref1.z, triangle.ref2.z), triangle.ref3.z);


        if((triangle.ref3.y - triangle.ref1.y) / max(maxX - minX, maxZ - minZ) <
             settings.angleForFilling) {
            if(slicePosition >= 2) flat[slicePosition-2] = true;
            if(slicePosition >= 1) flat[slicePosition-1] = true;
            flat[slicePosition] = true;
            if(slicePosition + 1 < flat.size()) flat[slicePosition+1] = true;
        }
    }
    float yPos = settings.pointAbovePlate;
    for(auto & segList : segments) {
        for(auto & segment : segList) {
            segment.ref1.y = yPos;
            segment.ref2.y = yPos;
        }
        yPos += settings.layerThickness;
    }
    pair<vector<bool>, vector<vector<Segment> > > output;
    output.first = flat;
    output.second = segments;
    return output;
}

void segmentSlicingAndPrinting(vector<Segment> segments, bool xSort, float
     fillDist) {
    bool orientationOfSlicer = false;
    if(xSort) {
        vector<vector<Vector> > pointList (settings.sizeX / fillDist);
        for(Segment seg : segments) {
            if(seg.ref2.x < seg.ref1.x) swap(seg.ref1, seg.ref2);
            int slicePosition = seg.ref1.x / fillDist;
            if(fmod(seg.ref1.x, fillDist) != 0.) ++slicePosition;
            for(int i = slicePosition; fillDist * i < seg.ref2.x; ++i)
                pointList[i].push_back(crossPointX(seg.ref1, seg.ref2, i *
                     fillDist));
        }
        for(vector<Vector> points : pointList) {
            sort(points.begin(), points.end(), zComp);
            if(orientationOfSlicer) reverse(points.begin(), points.end());
            for (int i = 0; i < points.size() && i < points.size() - 1; i += 2) {
                if(orientationOfSlicer) {
                    points[i].z -= settings.layerThickness * 3; //ERFAHRUNGSWERT
                    points[i+1].z += settings.layerThickness * 3;
                }
                else {
                    points[i].z += settings.layerThickness * 3;
                    points[i+1].z -= settings.layerThickness * 3;
```

```
                }
                moveTo(points[i]);
                printTo(points[i+1]);
            }
            orientationOfSlicer = !orientationOfSlicer;
        }
    } else {
        vector<vector<Vector> > pointList (settings.sizeZ / fillDist);
        for(Segment seg : segments) {
            if(seg.ref2.z < seg.ref1.z) swap(seg.ref1, seg.ref2);
            int slicePosition = seg.ref1.z / fillDist;
            if(fmod(seg.ref1.z, fillDist) != 0.) ++slicePosition;
            for(int i = slicePosition; fillDist * i < seg.ref2.z; ++i)
                pointList[i].push_back(crossPointZ(seg.ref1, seg.ref2, i *
                     fillDist));
        }
        for(vector<Vector> points : pointList) {
            sort(points.begin(), points.end(), xComp);
            if(orientationOfSlicer) reverse(points.begin(), points.end());
            for (int i = 0; i < points.size() && i < points.size() - 1; i += 2) {
                if(orientationOfSlicer) {
                    points[i].x -= settings.layerThickness * 3;
                    points[i+1].x += settings.layerThickness * 3;
                }
                else {
                    points[i].x += settings.layerThickness * 3;
                    points[i+1].x -= settings.layerThickness * 3;
                }
                if(abs(points[i].x - points[i+1].x) + abs(points[i].z - points[i
                     +1].z) >= settings.layerThickness * 7) {
                    moveTo(points[i]);
                    printTo(points[i+1]);
                }
            }
            orientationOfSlicer = !orientationOfSlicer;
        }
    }
}



vector<string> lexedInput;

vector<string> loadFile(string fileLocation) {
    ifstream fin; //declare a file stream
    fin.open( ofToDataPath(fileLocation).c_str() ); //open your text file
    vector<string> output; //declare a vector of strings to store data

    while(fin != NULL) //as long as theres still text to be read
    {
        string str; //declare a string for storage
        getline(fin, str); //get a line from the file, put it in the string
        output.push_back(str); //push the string onto a vector of strings
    }
    return output;
}

vector<string> split (string &s, vector<char> &symbols) {
    vector<string> output;
    string temp = "";
    for(int i = 0; i < s.length(); ++i)
    {
        bool hasSplitted = false;
        for(int j = 0; j < symbols.size(); ++j)
        {
            if(s.at(i) == symbols[j]) {
                if(temp.length() != 0) output.push_back(temp);
                temp = "";
                hasSplitted = true;
            }
        }
        if(not hasSplitted) temp += s.at(i);
    }
    if(temp.length() != 0) output.push_back(temp);
```

```
    return output;
}
string getFileContents(const char *filename) {
    ifstream in(filename, ios::in | ios::binary);
    if (in) {
        string contents;
        in.seekg(0, ios::end);
        contents.resize(in.tellg());
        in.seekg(0, ios::beg);
        in.read(&contents[0], contents.size());
        in.close();
        return(contents);
    }
    throw(errno);
}

vector<string> splitASCIISTL (string &s) {
    vector<string> output;
    string currentString;
    for(int i = 0; i < s.size(); ++i) {
        if(s[i] < 33 && currentString.size() != 0) {
            output.push_back(currentString);
            currentString.clear();
        }
        else if(s[i] > 32) {
            //ONLY USE LOWER-CASE
            if(s[i] >= 65 && s[i] <= 90) currentString.push_back(s[i] + 32);
            else currentString.push_back(s[i]);
        }
    }
    return output;
}

void parseAsciiSTL(string &input) {
    lexedInput.clear();
    const clock_t totalTime = clock();

    if(settings.displayApproximateTimeConsumption) cout << "Lexing␣input␣file:␣"
         << endl;

    lexedInput = splitASCIISTL(input);
    int it = 0;
    vector<Vector> preLoadedTriangle (4, Vector(0, 0, 0)); //0 = Point 1, 1 =
         Point 2, 2 = Point 3, 3 = normal
    vector< vector<Vector> > triangleList;

    int currentPoint = 0;
    int howLongItWillTake = lexedInput.size() / 100;
    for(it = 0; lexedInput[it] != "facet"; ++it) {}

    while(it < lexedInput.size())
    {
        if(settings.displayApproximateTimeConsumption && it % howLongItWillTake ==
             0) cout << "Loading␣file:␣" <<  it / howLongItWillTake << "%\n";
        if(lexedInput[it] == "solid")
        {
            while(lexedInput[++it] != "facet")
            {
                name += lexedInput[it];
            }
        }
        else if(lexedInput[it] == "normal")
        {
            preLoadedTriangle[3].x = ::atof( lexedInput[++it].c_str() );
            preLoadedTriangle[3].y = ::atof( lexedInput[++it].c_str() );
            preLoadedTriangle[3].z = ::atof( lexedInput[++it].c_str() );
        }
        else if(lexedInput[it] == "vertex")
        {
            preLoadedTriangle[currentPoint].x = ::atof ( lexedInput[++it].c_str()
                 );
            preLoadedTriangle[currentPoint].y = ::atof ( lexedInput[++it].c_str()
                 );
```

```
        preLoadedTriangle[currentPoint].z = ::atof ( lexedInput[++it].c_str()
            );
        ++currentPoint;
    }
    else if(lexedInput[it] == "endfacet")
    {
        //meshData.addTriangle(preLoadedTriangle[0], preLoadedTriangle[1],
            preLoadedTriangle[2]);
        triangleList.push_back(preLoadedTriangle);
        currentPoint = 0;
        ++it;
    }
    else ++it;
}

//Inverting Y and Z axis for OF
for(int i = 0; i < triangleList.size(); ++i)
{
    for(int j = 0; j < 3; ++j)
    {
        swap(triangleList[i][j].y, triangleList[i][j].z);
    }
}

//Now for the approx sizes
float maxX = triangleList[0][0].x, maxY = triangleList[0][0].y, maxZ =
    triangleList[0][0].z;
float minX = triangleList[0][0].x, minY = triangleList[0][0].y, minZ =
    triangleList[0][0].z;
for(int i = 0; i < triangleList.size(); ++i)
{
    for(int j = 0; j < 3; ++j)
    {
        if(triangleList[i][j].x > maxX) maxX = triangleList[i][j].x;
        if(triangleList[i][j].y > maxY) maxY = triangleList[i][j].y;
        if(triangleList[i][j].z > maxZ) maxZ = triangleList[i][j].z;
        if(triangleList[i][j].x < minX) minX = triangleList[i][j].x;
        if(triangleList[i][j].y < minY) minY = triangleList[i][j].y;
        if(triangleList[i][j].z < minZ) minZ = triangleList[i][j].z;
    }
}
//Translate to 0 0

for(int i = 0; i < triangleList.size(); i++)
{
    for(int j = 0; j < 3; j++)
    {
        triangleList[i][j].x -= minX;
        triangleList[i][j].y -= minY;
        triangleList[i][j].z -= minZ;
    }
}
maxX -= minX;
maxY -= minY;
maxZ -= minZ;
minX = 0;
minY = 0;
minZ = 0;

if(settings.displayApproximateTimeConsumption) cout << "Reading␣in␣took␣this␣
    long:␣" << float( clock () - totalTime ) /  CLOCKS_PER_SEC << endl;


cout << "Original␣dimensions:␣"<< endl;
cout << "X:␣" << maxX << "mm" << endl;
cout << "Y:␣" << maxY << "mm" << endl;
cout << "Z:␣" << maxZ << "mm" << endl;


//Scale to max
float xScale = settings.sizeX / maxX; //1; //settings.sizeX / maxX;
float yScale = settings.sizeY / maxY; //1; //settings.sizeY / maxY;
float zScale = settings.sizeZ / maxZ; //1; //settings.sizeZ / maxZ;
float minScale = 1; // min(xScale, min(yScale, zScale)); //1
```

```
//Now scaling to sc

//Meta scaling
float scaleNow = 1; //0.4; //0.135;//1; //0.135;
minScale *= scaleNow;
for(int i = 0; i < triangleList.size(); i++)
{
    for(int j = 0; j < 3; j++)
    {
        triangleList[i][j].x *= minScale;
        triangleList[i][j].y *= minScale;
        triangleList[i][j].z *= minScale;
        //swap(triangleList[i][j].y, triangleList[i][j].z);
    }
}
maxX *= minScale;
maxY *= minScale;
maxZ *= minScale;
cout << "Scaled␣dimensions:␣"<< endl;
cout << "X:␣" << maxX << "mm" << endl;
cout << "Y:␣" << maxY << "mm" << endl;
cout << "Z:␣" << maxZ << "mm" << endl;


//Add triangles to ofMesh
for(int i = 0; i < triangleList.size(); i++) {
    dataMesh.addTriangle(triangleList[i][0], triangleList[i][1], triangleList[
        i][2]);
}
//Vector centerTrans((settings.sizeX / 2) - (maxX / 2), 0, (settings.sizeZ /
    2)  - (maxX / 2));
//dataMesh.translate(centerTrans);
//dataMesh.scale(scaleNow);

ofVec3f centerOfRotation ((maxX / 2) * scaleNow, (maxY / 2) * scaleNow, (maxZ
    / 2) * scaleNow);
easyCam.setTarget(centerOfRotation);

transX = (settings.sizeX / 2) - (maxX / 2);
transZ = (settings.sizeZ / 2)  - (maxX / 2);

startCodes();

//slow algorithm
vector<Triangle> fastTriangles;
for(int i = 0; i < triangleList.size(); ++i) {
    Triangle tri (triangleList[i][0], triangleList[i][1], triangleList[i][2]);
    fastTriangles.push_back(tri);
}

const clock_t gcodeTimeCreation = clock();
cout << "Started␣algorithm␣at␣time:␣" << gcodeTimeCreation << endl;

pair<vector<bool>, vector<vector<Segment> > > generated = triangleSlicing(
    fastTriangles);
bool xSort = true;
int debugInformation = generated.first.size() / 100;
for(int i = 1; i < generated.first.size(); ++i) { //skip first segment
    if(settings.displayApproximateTimeConsumption && i % debugInformation ==
        0) cout << i / debugInformation << "%" << endl;
    if(settings.doFill) {
        if(i * settings.layerThickness < settings.baseHeight || generated.
            first[i] || settings.safeMaterial == false) {
            //Twice the shell
            segmentSlicingAndPrinting(generated.second[i], xSort, settings.
                layerThickness * 2);
            if(settings.quadraticTimeSearch) createShell(generated.second[i],
                1.0);
            else createLinearTimeShell(generated.second[i]);
            if(settings.quadraticTimeSearch) createShell(generated.second[i],
                1.0);
            else createLinearTimeShell(generated.second[i]);

            //createShell(generated.second[i], 1.0);
```

```
                //createShell(generated.second[i], 1.0);
            }
            else {
                //Twice the grid
                segmentSlicingAndPrinting(generated.second[i], xSort, settings.
                    fillSpace);
                xSort = !xSort;
                segmentSlicingAndPrinting(generated.second[i], xSort, settings.
                    fillSpace);
                //createShell(generated.second[i], 1.0);
                if(settings.quadraticTimeSearch) createShell(generated.second[i],
                    1.0);
                else createLinearTimeShell(generated.second[i]);
            }
            xSort = !xSort;
        }
        else {
            if(settings.quadraticTimeSearch) createShell(generated.second[i], 1.0)
                ;
            else createLinearTimeShell(generated.second[i]);
        }
    }

    //slowalgorithm(slowTriangles);
    //scanline(slowTriangles);
    if(settings.displayApproximateTimeConsumption) cout << "Took␣this␣long:␣" <<
         float( clock () - gcodeTimeCreation ) /  CLOCKS_PER_SEC << endl;


    endCodes();

    const clock_t outputTime = clock();

    cout << "Triangles:␣" << fastTriangles.size() << endl;
    cout << "Vertices:␣" << dataMesh.points.size() << endl;


    ofstream p;
    p.open("/Users/Desktop/outputhi.gcode");

    for(string temp : gcode) p << temp << endl;
    p.close();
    if(settings.displayApproximateTimeConsumption) cout << "Output␣took␣this␣long:
        ␣" << float( clock () - outputTime ) /  CLOCKS_PER_SEC << endl;
    if(settings.displayApproximateTimeConsumption) cout << "Output␣took␣this␣long:
        ␣" << float( clock () - totalTime ) /  CLOCKS_PER_SEC << endl;
}


//GRAPHICS
void displayLines()
{
    ofSetLineWidth(5);
    ofSetColor(ofColor::red);
    ofLine(0, 0, 0, 10000, 0, 0);
    ofSetColor(ofColor::green);
    ofLine(0, 0, 0, 0, 10000, 0);
    ofSetColor(ofColor::blue);
    ofLine(0, 0, 0, 0, 0, 10000);
    ofSetLineWidth(1);
    //model.drawFaces();
}

void displayObject()
{
    ofEnableLighting();
    ofEnableDepthTest();

    ofSetColor(ofColor::blue);
    graphicsMesh.drawWireframe();
    ofSetColor(ofColor::white);
    graphicsMesh.drawFaces();

    ofDisableDepthTest();
```

```
    ofDisableLighting();
    //spotlight.disable();
    //ofDisableLighting();
}

//Pre-Processor macros
#define DEB(x) cerr << x << endl;


//--------------------------------------------------------------
void ofApp::setup(){
    //ofEnableDepthTest();
    ofSetVerticalSync(true);
    //glEnable(GL_DEPTH_TEST);
    font.loadFont("Timber.ttf", 16);
    ofSetBackgroundColor(ofColor::white);
    ofBackground(ofColor::white);
    ofColor color;
    color.setBrightness(150);
    lightSource.setDiffuseColor(color);
    lightSource.enable();
}

//--------------------------------------------------------------
void ofApp::update(){

}
//--------------------------------------------------------------
void ofApp::draw(){
    switch (mode) {
        case FILE_SELECT:
            ofSetRectMode(OF_RECTMODE_CENTER);
            ofSetColor(ofColor::black);
            if(mouseX > (ofGetWindowWidth() / 2) - 100 && mouseX < (
                ofGetWindowWidth() / 2) + 100 && mouseY > (ofGetWindowHeight() /
                2) - 25 && mouseY < (ofGetWindowHeight() / 2) + 25) ofSetColor(
                ofColor::gray);
            ofRect(ofGetWindowWidth() / 2, ofGetWindowHeight() / 2, 200, 50);
            ofSetColor(ofColor::white);
            drawStringCenter("Load␣File...", ofGetWindowWidth() / 2,
                ofGetWindowHeight() / 2);
            ofSetColor(ofColor::gray);
            drawStringCenter("ASCII␣STL␣files␣supported", ofGetWindowWidth() / 2,
                 ofGetWindowHeight() / 2 + 50);
            break;

        case DRAW:
            easyCam.begin();
            displayLines();
            displayObject();
            easyCam.end();
            break;

        default:
            cout << "Unknown␣mode␣" << mode << endl;
            break;
    }
    //cursor
    ofSetColor(ofColor::black);
    ofLine(mouseX - 10, mouseY, mouseX + 10, mouseY);
    ofLine(mouseX, mouseY - 10, mouseX, mouseY + 10);
}

//--------------------------------------------------------------
void ofApp::keyPressed(int key){

}

//--------------------------------------------------------------
void ofApp::keyReleased(int key){

}

//--------------------------------------------------------------
```

```
void ofApp::mouseMoved(int x, int y ){

}

//--------------------------------------------------------------
void ofApp::mouseDragged(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mousePressed(int x, int y, int button){

}

//--------------------------------------------------------------
void ofApp::mouseReleased(int x, int y, int button){
    if (mode == FILE_SELECT) {
        //Load file box
        if(x > (ofGetWindowWidth() / 2) - 100 && x < (ofGetWindowWidth() / 2) +
             100 && y > (ofGetWindowHeight() / 2) - 25 && y < (ofGetWindowHeight()
              / 2) + 25) {
            ofFileDialogResult result = ofSystemLoadDialog( "Select␣3D␣Model" ,
                 false);
            string path = result.getPath();
            if(path.size() > 0) {
                cout << "Loading:␣" << path << endl;
                string fileInformation = getFileContents(path.c_str());
                parseAsciiSTL(fileInformation);
                graphicsMesh = dataMesh.parseGraphicsMesh();
                mode = DRAW;
            }
        }
    }
}

//--------------------------------------------------------------
void ofApp::windowResized(int w, int h){

}

//--------------------------------------------------------------
void ofApp::gotMessage(ofMessage msg){

}

//--------------------------------------------------------------
void ofApp::dragEvent(ofDragInfo dragInfo){

}
```