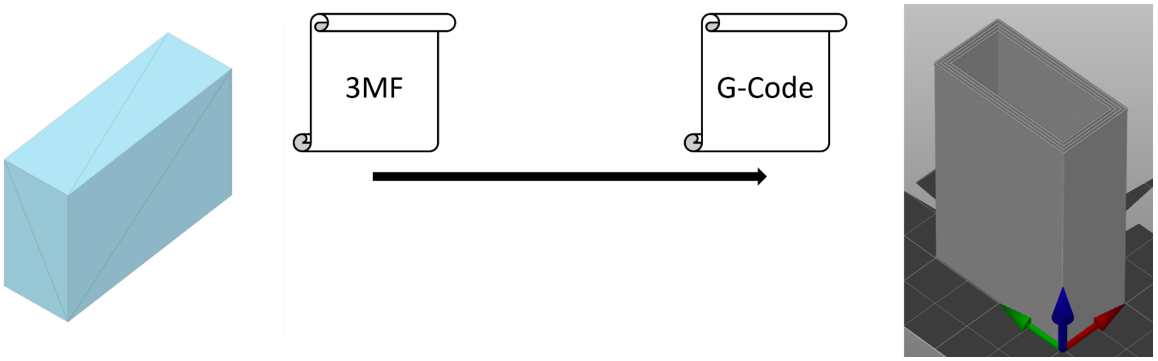
λslicer

Programmieren eines Slicers für den 3D-Druck



Abschlussprojekt im Modul “Fortgeschrittene Funktionale Programmierung”

Ludwig Dinter & Katja Frey

Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. David Sabel

20.03.2022

1. Das Projekt

Das Ziel des Projekts war es, einen Slicer für den 3D-Druck zu entwerfen. Das entstandene Programm liest einen 3D-Körper ein, dessen Oberfläche durch ein Dreiecks-Mesh im 3MF-Standard des 3MF-Konsortiums beschrieben ist (vgl. https://3mf.io/). Der 3D-Körper wird an der Z-Achse in konstanten Abständen (Standardwert 0.2mm) horizontal (XY-Ebene) zerschnitten ("gesliced"). Anhand des Dreiecks-Mesh werden die Konturen berechnet, die den Umriss des Körpers je Schnitthöhe beschreiben. Die berechnete Kontur wird n-mal (Standardwert n=4) nach innen versetzt ("geoffsetted"), um die Füllung des Körpers zu einem gewissen Grad zu realisieren. Anschließend werden für die Konturen und Offsets weitere Größen berechnet wie beispielsweise die Menge an Material, die auf einem Pfad-Stück aus der Düse herausgedrückt werden soll. Das Ergebnis wird schließlich in eine Datei im "G-Code" Format geschrieben. Wie in der Abbildung auf dem Deckblatt zu erkennen ist, bleibt bei unserem Programm zwangsläufig ein Teil des Körpers hohl. Gebräuchliche 3D-Slicer berechnen zusätzlich raumfüllende Pfade, um den Hohlraum zu schließen. Dies hätte den zeitlichen Rahmen unseres Projekts jedoch gesprengt.
Unser Programm kann in der Konsole ausgeführt werden, wobei die Nutzer*in den Pfad der 3MF-Datei sowie den Zielpfad der "G-Code"-Datei angeben muss. Zusätzlich können optional die Standardeinstellungen verändert werden; die verfügbaren Optionen lassen sich mit "--help" anzeigen. Ein Aufruf des Programms könnte daher folgendermaßen Aussehen:

```
stack exec -- hslicer-exe -s 0.2 -c 4 -o "./gcodefile.gcode"
"./resources/example_3mfs/Box/3D/3dmodel.model" +RTS -N
```

Abb1. Beispielaufruf für unser Programm, wobei -s die Slice-Höhe und -c die Anzahl der Offsets je Kontur angibt.

2. Eingebrachte Vertiefende Themen

Zur Umsetzung des Projekts verwendeten wir folgende vertiefende Themen:
Anwendungsprogrammierung: Einerseits wird die Bibliothek xml-conduit für das Parsen der 3MF-Datei eingebunden. Andererseits wird die Bibliothek hspec verwendet, um die eigenen Funktionen testen zu können.
Parallelität: Die Berechnung der einzelnen Slices (Konturen und Offsets) wird unter Verwendung von Strategien parallelisiert und die Performance mit einer nicht-parallelisierten Version des Programms verglichen (s. Abschnitt Tests zur Parallelität).
Funktionale Referenzen (Linsen): Für alle selbst-definierten Datenstrukturen werden Linsen erzeugt (teilweise automatisch, teilweise manuell). Verschiedene Berechnungen, die diese Datenstrukturen verändern, kombinieren Linsen miteinander und machen Gebrauch von Operatoren der Linsen-Bibliothek lens.

3. Vorgehensweise

Über die gesamte Projektlaufzeit hinweg arbeiteten wir sowohl einzeln als auch in der Gruppe. Wöchentlich fanden 1-2 online Gruppentreffen statt, für die wir Kollaborations-Tools wie Bildschirmübertragung (via Zoom) oder Pair Programming Extensions (VSC LiveShare) nutzten. Während der Treffen fügten wir Code-Stücke zusammen, unterstützten uns gegenseitig bei Problemen, programmierten gemeinsam und berieten über Datenstrukturen, die Programmstrukturierung und die Aufgabenteilung bis zum nächsten Treffen. Zwischen den Gruppentreffen hatte jeder eine Aufgabe, die bis zum nächsten Treffen gelöst werden sollte. Ludwigs Schwerpunkte waren dabei die Berechnung der Offsets der Konturen sowie die Übersetzung in GCode. Katjas Schwerpunkte waren das Einlesen der 3MF-Datei sowie die Berechnung der Konturen.

Die vertiefenden Themen wurden in den gemeinsamen Gruppentreffen erarbeitet und umgesetzt. In einem abschließendem Gruppentreffen verfassten wir diese Dokumentation. Insgesamt trugen beide Gruppenmitglieder in gleichen Teilen zur Umsetzung des Projekts bei.

4. Schwierigkeiten und Herausforderungen

Fließkomma-Ungenauigkeiten: Wir entschieden uns bewusst dazu, alle Funktionen und Datenstrukturen, die etwas mit Computergrafik zu tun haben, selbst zu entwickeln und nicht auf vorhandene Bibliotheken zuzugreifen. Dabei verwendeten wir den Fließkomma-Datentyp “Double”. Mit der Zeit fiel uns auf, dass verschiedene Grafikfunktionen unpräzise Ergebnisse lieferten. Nach einer kurzen Recherche wurde deutlich, dass dies auf mangelnde Präzision des Fließkomma-Datentyps zurückzuführen war. Eine Abhilfe wäre, auf einen Datentyp zu setzen, der rationale Zahlen darstellt, etwa “Data.Ratio.Rational”.

Innere und äußere Konturen unterscheiden: Um die Kontur-Offsets zu berechnen, ist es notwendig zu wissen, welche Seite einer Kontur die Innenseite eines 3D-Objekts ist. Ein 3D-Objekt, durch das ein Loch in der Mitte führt, hat sowohl eine innere Kontur als auch eine äußere. Dementsprechend wird bei einer inneren Kontur der Offset nach “außen” durchgeführt, und bei einer äußeren Kontur nach “innen” (s. Abb2., eine orange Kontur ist “außen” und die andere “innen”). Zur Unterscheidung definierten wir die Datentypen “InnerContour” und “OuterContour”, die jeweils eine “[Vertex]” kapseln.

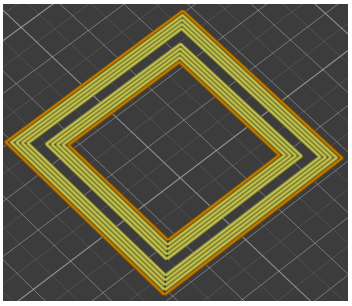


Abb2.

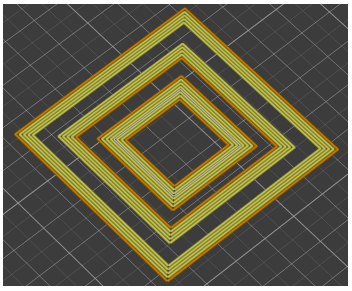


Abb3.

Schwierigkeiten ergaben sich, als wir mehrfach verschachtelte Konturen als “InnerContour” oder “OuterContour” typisieren wollten (s. Abb3.). Unser implementierter Algorithmus kann dies nicht berücksichtigen, da er nur prüft, ob eine Kontur von einer anderen Kontur umschlossen ist. Ist dies der Fall, wird sie als “InnerContour” typisiert und somit wären alle verschachtelten Konturen fälschlicherweise vom Typ “InnerContour”. Um dieses Szenario abzudecken, müssten wir eine andere Klassifizierungsstrategie entwickeln.

Aufgabenverteilung: Anfangs hatten wir Schwierigkeiten, eine Schnittstelle zu finden, um zu gewährleisten, dass jeder unabhängig mit klar definierten Anforderungen bis zum nächsten Treffen arbeiten könnte. Es stellte sich schnell heraus, dass es sinnvoll ist, Datenstrukturen und Typsignaturen von spezifischen Funktionen gemeinsam im Treffen festzulegen - besonders, wenn es sich nicht vermeiden ließ, dass wir parallel an eng verzahnten Programmteilen arbeiten.

5. Tests zur Parallelität

Um Parallelität als vertiefendes Thema einzubringen, experimentierten wir damit, unser Programm an verschiedenen Stellen mit verschiedenen Strategien zu parallelisieren. Für alle Analysen verwendeten wir einen Zylinder, dessen Kontur ein 500-Punkte Polygon ist. Alle parallelisierten Funktionen befinden sich im Modul LibHslicer.PlanarSlice. In unseren Performance Tests verglichen wir die Strategien r0, rseq und rdeepseq. In Abbildung 6 ist eine Übersicht der Ergebnisse dargestellt. Eine Analyse nach den ersten Parallelisierungs-Versuchen ergab, dass unser Programm mit 5 Kernen am schnellsten zu laufen scheint (vgl. Abb4.). Die folgenden Performance-Analysen wurden daher mit 5 Kernen durchgeführt (vgl. Abb5.).

```
for f in 1 2 3 4 5 6 7 8; do printf $f; printf ": "; stack exec -- hslicer-exe
"./test/Lib3mfSpec_res/Polygon/3D/3dmodel.model"  "./gcodefile.gcode"  +RTS  -N$f
-s 2>&1 | sed -n "/Total/p";done
1:   Total   time   12.328s  ( 12.758s elapsed)
2:   Total   time   14.578s  (  7.771s elapsed)
3:   Total   time   18.755s  (  7.015s elapsed)
4:   Total   time   22.427s  (  6.540s elapsed)
5:   Total   time   26.688s  (  6.466s elapsed)
6:   Total   time   31.429s  (  7.633s elapsed)
7:   Total   time   33.508s  (  7.668s elapsed)
8:   Total   time   34.839s  (  8.101s elapsed)
```

Abb4. Laufzeiten bei Manipulation der verwendeten Kerne.

Zunächst parallelisierten wir die Funktion sliceContours. Dies brachte einen erheblichen Performance-Vorteil gegenüber keiner Parallelisierung. Der Unterschied zwischen den Strategien rseq und rdeepseq war dagegen nur sehr gering. Wir vermuten, dass die Funktionen, welche von sliceContours aufgerufen werden, die Elemente schon fast bis zur Normalform auswerten. Anschließend parallelisierten wir zusätzlich die Funktion calcMultiOffset. Zwar werden dadurch alle Kerne über die gesamte Laufzeit benutzt, allerdings wirkt sich das kaum auf die Laufzeit aus. Der Grund liegt darin, dass nur die Berechnung der Offsets für eine Kontur parallelisiert wird, nicht jedoch die Offset-Berechnung je Slice-Ebene. Letzteres gelang uns nicht, da die Berechnung der Konturen je Slice bereits parallel erfolgt. Hierfür müsste vmtl. eine eigene Auswertungsstrategie entwickelt werden.

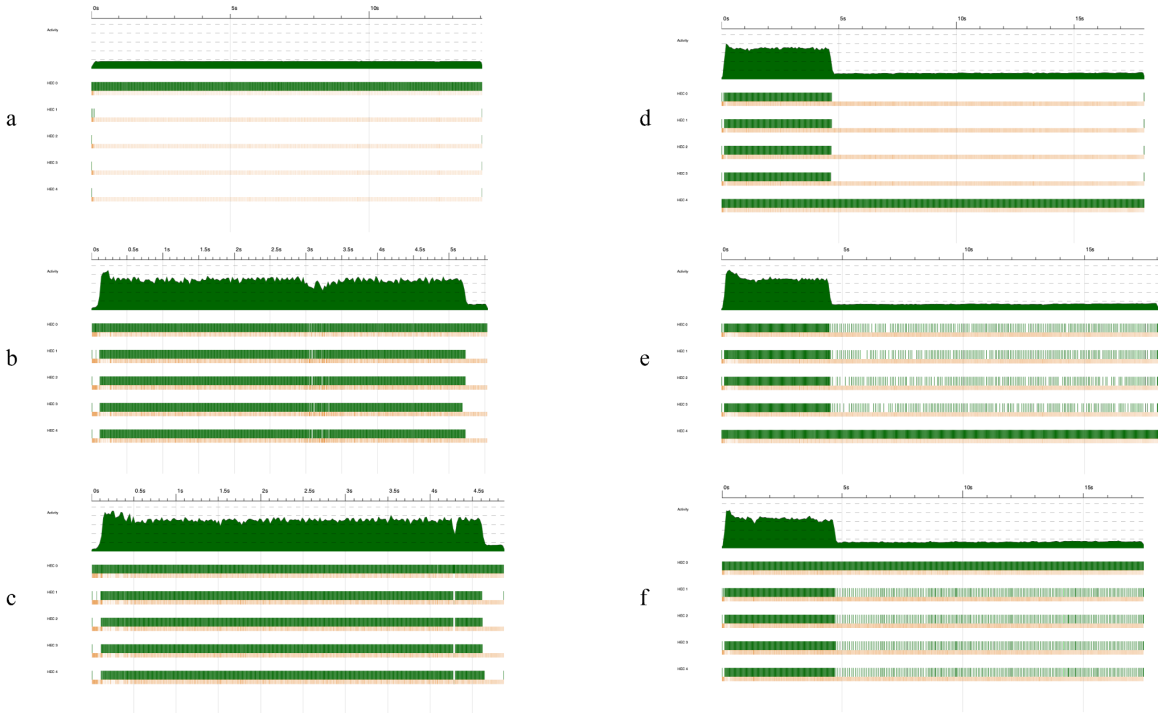


Abb5. Gegenüberstellung verschiedener Parallelisierungen bei Verwendung von 5 Kernen je Ausführung.
a keine Parallelisierung, 1 Offset, Laufzeit 14,1s.
b Parallelisierung von sliceContours (rseq), 1 Offset, Laufzeit 5,54s.
c Parallelisierung von sliceContours (rdeepseq), 1 Offset, Laufzeit 4,89s.
d Parallelisierung von sliceContours (rdeepseq), 4 Offsets, Laufzeit 17,98s.
e Parallelisierung von sliceContours (rdeepseq) und calcMultiOffset (rseq), 4 Offsets, Laufzeit 18,1s.
f Parallelisierung von sliceContours (rdeepseq) und calcMultiOffset (rdeepseq), 4 Offsets, Laufzeit 17,49s.