Hochschule Konstanz
Department of Computer Science

Eingereicht von
Nik Steinbrügge
Matrikelnummer 297033

nik.steinbruegge@htwg-konstanz.de

Bachelor Thesis

Entwicklung einer Software zur 2D-Modellierung von Räumen und Berechnung einer Delaunay-Triangulierung

Konstanz, 17 August 2021

Bachelor Thesis

Entwicklung einer Software zur 2D-Modellierung von Räumen und Berechnung einer Delaunay-Triangulierung

von

Nik Steinbrügge

Bachelor of Science
in Angewandter Informatik

an der Hochschule Konstanz - Technik, Wirtschaft und Gestaltung,

Matrikelnummer:	297033
Abgabedatum:	17 August 2021
Prüfer:	**Prof. Dr. Rebekka Axthelm**
Zweitprüfer:	**Prof. Dr. Georg Umlauf**

Eidesstattliche Erklärung

Hiermit erkläre ich, Nik Steinbrügge, geboren am 03.09.1998 in Lüneburg,

(1) dass ich meine Bachelorarbeit mit dem Titel:

Entwicklung einer Software zur 2D-Modellierung von Räumen und Berechnung einer Delaunay-Triangulierung

in der Fakultät Informatik unter Anleitung von Professorin Rebekka Axthelm selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angeführten Hilfen benutzt habe,

(2) dass ich die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

(3) dass die eingereichten Abgabe-Exemplare in Papierform und im PDF-Format vollständig übereinstimmen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, den 17 August 2021

Ort, Datum Nik Steinbrügge

Abstract

In dieser Arbeit wurde ein Delauny-Mesh-Generator in Python entwickelt. Zum Generieren eines Meshs wurde die Bibliothek pyGIMLi verwendet. Anschließen wurde ein Performancetest zwischen zwei Implementierungen einer zeitaufwendigen Funktion durchgeführt. Diese wurde in C und in Python implementiert, um exemplarisch die Performance einer Implementierung von pFlow in Python hervorzusagen. Bei gleichem Algorithmus hat die Python-Version der Funktion ungefähr 65-mal länger gebraucht als ihre C-Variante.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1

Einleitung

Notsituationen und Evakuierungen von Menschen können überall und jeder Zeit auftreten. Deshalb ist es von großer Bedeutung, dass Gebäude in solchen Situationen ausreichend Notausgänge und Fluchtwege besitzen. Diese sollten schon beim Entwerfen und Planen von Gebäuden berücksichtigt werden. Diese Planung wird zunehmend schwerer, wenn die Anzahl der zu Evakuierenden steigt, wie zum Beispiel auf Großveranstaltungen. Es ist komplex, solche Abläufe für Notfälle für eine große Menschenmenge zu planen. Daher gibt es Software, die sich spezialisiert, Bewegungen von Menschen zu simulieren, sodass Personenströme in Notsituation berechnet werden können. Diese Simulationen ermöglichen das Prognostizieren und Zeit messen einer vollständigen Evakuierung. Schließlich können mit diesen Ergebnissen Änderungen vorgenommen werden. Ein Gebäude könnte zusätzliche Fluchtwege benötigen oder die Menschenanzahl eines Stadtfestes könnte beschränkt werden.

2

pFlow

pFlow ist eine Software zum Simulieren von Personenströmen und -bewegungen. pFlow simuliert dabei Laufwege von Personen in vordefinierten Orten. In diesen Orten werden eine definierbare Menge an Personen zufällig verteilt, die sich als Ziel zu einem ebenfalls definierbaren Ausgang bewegen und damit die Simulation verlassen. Die Simulation endet, wenn alle Personen den Ausgang erreicht haben. Während der Simulation werden die Personen und ihre Bewegungen zum Ausgang in Echtzeit dargestellt. Dabei werden zusätzlich Personendichte und Richtungsvektoren der Laufbewegung berechnet und protokolliert, sodass die Simulation umfangreich ausgewertet werden kann.

Eine Personenflusssimulation von pFlow ist von Bedeutung, wenn prognostiziert werden soll, ob sich Räumlichkeiten für eine bestimmte Anzahl an Personen eignen, sodass in Notsituationen das Risiko einer Massenpanik oder Ähnlichem so niedrig wie möglich ist. Flaschenhälse eines Grundrisses für den Personenfluss können von pFlow prognostiziert und hervorgehoben werden. Architekten könnten pFlow beim Entwerfen von Gebäuden benutzen und prognostizieren, ob ein geplanter Grundriss mit einer geplanten Menge an Personen kompatibel ist. Dies bedeutet, dass herausgefunden wird, ob Räume, Türen, Gänge und Fluchtwege breit genug sind, um eine geplante Anzahl an Personen in einer Notsituation beim Verlassen des Gebäudes nicht zu behindern. Aufgrund der guten Skalierbarkeit von pFlow findet es außerdem Einsatz in der Eventplanung. Großveranstaltungen wie Stadtfeste oder ähnliche große Festlichkeiten können im Voraus hinsichtlich des Risikos auf Massenpaniken evaluiert werden.

2.1. Funktionsweise

Der Grundbaustein jeder Personenflusssimulation ist der Ort der Simulation. Im Folgenden werden Orte, wenn sie pFlow als Basis für eine Simulation vorliegen, als Karten bezeichnet. Zunächst ist eine Karte eine zweidimensionale Ansicht eines Ortes aus der Vogelperspektive. Dabei wird der Ort durch Polygonzüge modelliert. Durch die Darstellung mithilfe von Polygonzügen lassen sich komplexere Strukturen von Orten wie zum Beispiel ganze Stockwerke, problemlos realisieren. Der signifikante Unterschied zwischen Ort und Karte ist, dass die Fläche einer Karte in einer triangulierten Form vorliegt. Dies bedeutet, dass die gegebene Fläche mit einer Menge von Dreiecken gefüllt ist. Die Fläche wird also nicht mehr durch Polygonzüge beschrieben, sondern durch ein Netz aus Dreiecken, welches die Fläche idealerweise exakt ausfüllt.

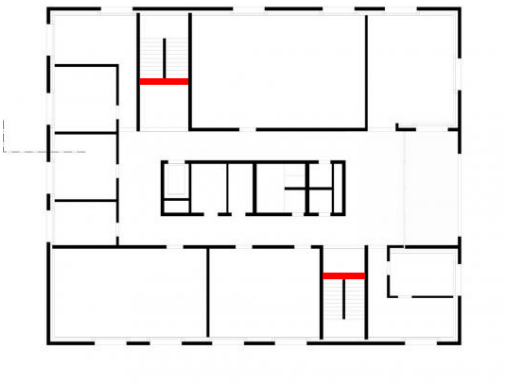

Abbildung 2.1: Grundriss mit Ausgängen [Gmb17]

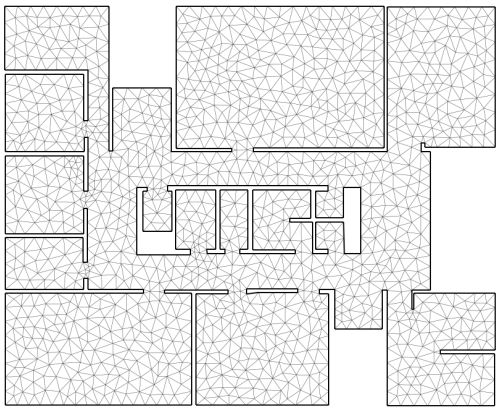

Abbildung 2.2: Triangulierter Grundriss

In Abbildung 2.1 ist ein Grundriss des zweiten Obergeschosses des O-Gebäudes der HTWG-Konstanz [Gmb17] dargestellt. Dieser Grundriss wird im Verlauf der Arbeit benutzt, um verschiedene Sachverhalte darzustellen. Hier soll dieser als Beispiel für einen möglichen Ort einer Simulation fungieren. Dieses Stockwerk ist in Abbildung 2.2 als eine Karte dargestellt. Wie zu sehen ist, unterscheiden sich beide Abbildungen darin, dass die Fläche der Karte (2.2) als Dreiecksgitter dargestellt ist. Außerdem ist auffällig, dass die Außenwände in Abbildung 2.1 Löcher für Fenster haben, die für eine pFlow-Simulation irrelevant ist und daher in der Karte nicht vorhanden sind. Trotz der Unterschiede der beiden Abbildungen ist es wichtig, dass die Größenverhältnisse des Grundrisses (2.1) und der Karte (2.2) exakt dieselben sind. Dies ist für die Genauigkeit der Simulation wichtig, da sich vom Ergebnis einer Simulation auf einer verfälschten Karte nur schwer Schlussfolgerungen auf den eigentlichen Ort ziehen lassen, da sich

Ort und Karte unterscheiden.

Die Beschreibung einer Fläche durch ein Dreiecksgitter ist für pFlow von Relevanz, da das Gitter als Struktur für Berechnungen benutzt wird. pFlow benutzt die Ecken der Dreiecke der Triangulierung als Punkte für Berechnungen. Beispielsweise wird auf ihnen die Personendichte und Bewegungsvektoren der Personen berechnet. Demnach ist es auf die Dichte an berechneten Daten ausschlaggebend, wie viele Dreiecke und damit auch Dreiecksecken es in der triangulierten Fläche gibt. Falls eine hohe Dichte an berechneten Daten erwünscht ist, wird folglich ein Gitter mit einer größeren Menge an Dreiecken benötigt, wobei diese Dreiecke bei steigender Anzahl kleiner werden. Hierbei wird auch von einer feinen Triangulierung gesprochen [She12].

Um eine Karte für eine Simulation zu finalisieren, wird die Definition von mindestens einem Ausgang benötigt. Der Ausgang repräsentiert während der Simulation ein Ziel für die Personen, dem sie sich nähern und beim Erreichen die Simulation verlassen. Außerdem besteht die Option, bestimmte Flächen der Karte als Attraktoren zu kennzeichnen. Attraktoren können mit einem Faktor definiert werden, der bestimmt, wie stark diese Fläche für Personen in der Simulation anziehend wirkt. Durch einen negativen Wert kann ein Attraktor auch abstoßend wirken.

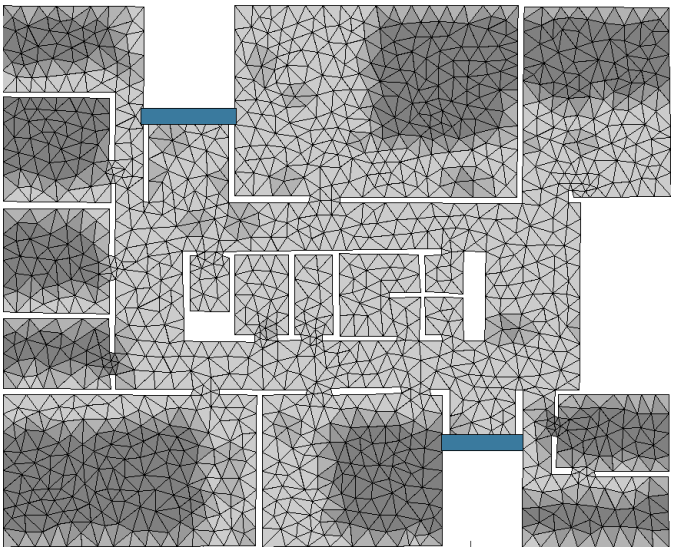

Abbildung 2.3: Triangulierter Grundriss mit zwei Ausgängen

Zunächst lässt sich in der Abbildung 2.1 sehen, dass zwei Stellen rot markiert sind. Diese markierten Stellen zeigen die Eingänge der Treppenhäuser, also zwei Stellen an denen das Stockwerk verlassen werden kann. Diese Ausgänge sollen nun auch für

die Simulation benutzt werden. Dafür müssen sie in die vorhandene Karte eingetragen werden. Folglich entsteht die überarbeitete Karte in Abbildung 2.3 mit blau markierten Ausgängen. Die Färbung des Hintergrunds in verschiedenen Graustufen kann dabei ignoriert werden.

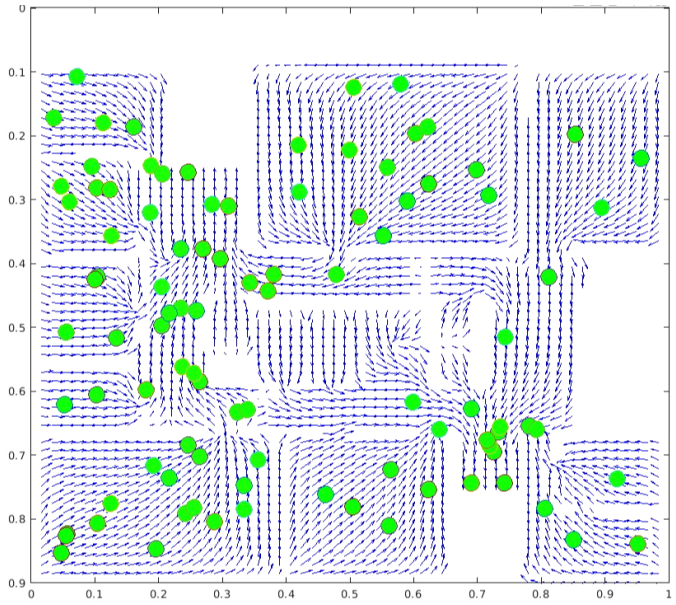

Abbildung 2.4: Screenshot einer Simulation nach 5 Sekunden

Die Karte mit den eingetragenen Ausgängen ist bereit, simuliert zu werden. Abbildung 2.4 zeigt einen Screenshot einer Simulation nach ungefähr fünf Sekunden nach Simulationsstart. Dabei stellen die grünen Kreise Personen dar und die Bewegungsvektoren sind durch kleine blaue Pfeile dargestellt. Da Abbildung 2.4 einen Zeitpunkt kurz nach Simulationsstart zeigt, lässt sich noch sehr gut die anfängliche zufällige Verteilung der Personen auf der Karte erkennen. Dabei ist zu sehen, dass es vermehrt Personen gibt, die sich kurz vor dem Ausgang befinden und in naher Zukunft die Simulation verlassen werden. Außerdem zeigen die Bewegungsvektoren sehr gut die Richtungen des Personenflusses an, sodass gut zu erkennen ist, in welche Richtung und zu welchem Ausgang sich die einzelnen Personen bewegen werden.

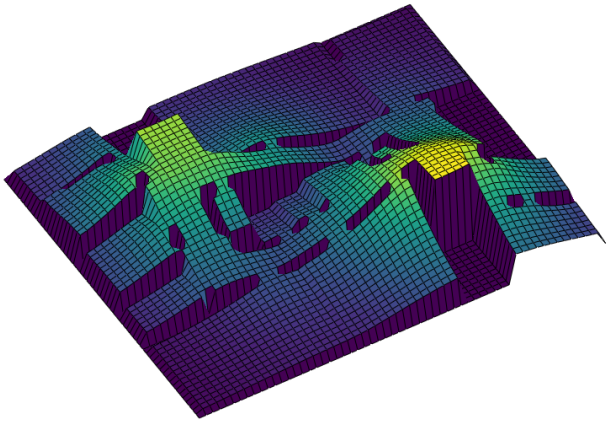

Abbildung 2.5: Dichteverteilung nach 5 Sekunden

Schließlich lässt sich in Abbildung 2.5 die Dichteverteilung der Personen der Simulation zum selben Zeitpunkt von Abbildung 2.4 sehen. Es lässt sich erkennen, dass schon kurz nach dem Simulationsstart die Personendichte zu den beiden Ausgängen steigt und direkt an den Ausgängen Maxima erreicht. Dies war zu erwarten, da die Ausgänge ein eindeutiger Flaschenhals für den Personenfluss sind, weil jede Person in der Simulation einen der Ausgänge als Ziel hat. Außerdem fällt auf, dass die Dichte innerhalb der Räume meistens sehr gering ist, aber diese plötzlich in den Bereichen der Türen zu den Fluren ansteigt. Diese Anstiege sind im Vergleich zu den Dichten an den Ausgängen klein, aber trotzdem sollten diese kleineren Flaschenhälse des Personenflusses nicht ignoriert werden. Andere Flaschenhälse mit einer hohen Personendichte würden in Abbildung 2.5 klar zu erkennen sein, aber abgesehen von den Ausgängen gibt es in diesem Beispiel keine.

Diese Beispiele sollen veranschaulichen, welche Bestandteile für eine pFlow Simulation von Relevanz sind, welche Daten berechnet werden und wie aus diesen Daten Rückschlüsse gezogen werden kann.

Zuletzt fällt auf, dass es in den oben benutzen Grundrissen und Karten keine Hindernisse in den Räumen gibt. Im Beispiel sind alle Räume komplett leer und das ist sehr unwahrscheinlich, da dies Büro- und Vorlesungsräume sind, in denen immer Tische stehen. Da die Tische, um für die Simulation berücksichtigt zu werden, im Gitternetz (2.2) als Löcher eingetragen werden müssen, wird eine komplett neue Triangulierung des Grundrisses benötigt. Daher ist es hilfreich, wenn schon triangulierte Flächen im Nachhinein editiert werden können. So könnten zum Beispiel verschiedene Anordnungen von Tischen in Räumen simuliert werden, sodass die Tische für eine Vorlesung

und für eine Evakuierung optimal platziert sind und sichergestellt werden kann, dass die Tische kein Labyrinth vor den Ausgängen sind.

3

pFlow-Map-Generator

Der wichtigste Bestandteil jeder Simulation von pFlow ist die Karte, auf der die Simulation durchgeführt wird. Da pFlow selbst keine Funktion besitzt, Karten zu erstellen, musste vorher auf andere Programme zurückgegriffen werden, um Karten für pFlow erstellen zu können. Diese Lücke in pFlows Workflow wird durch den pFlow-Map-Generator (pFMG) geschlossen. Ziel ist es, dass Karten für pFlow auf eine einfache Art modelliert, editiert und trianguliert werden können.

3.1. Idee

Grundsätzlich lassen sich die Aufgaben zum Erstellen einer Karte für pFlow, in zwei Teile gliedern, das Modellieren und das Triangulieren eines Grundrisses. Dabei umfasst das Modellieren den gesamten Prozess, in dem der Grundriss gezeichnet wird. Dabei soll eine Menge an Features gegeben sein, die diesen Prozess so einfach und intuitiv wie möglich gestalten sollen. Das Triangulieren beinhaltet die Generierung eines Dreiecksgitters auf der vorher modellierten Fläche und das Speichern in einem Dateiformat, welches mit pFlow kompatibel ist.

3.1.1. Das Modellieren

Das Modellieren einer Karte wird durch einen Polygonzug mit einer beliebigen Anzahl an Punkten realisiert. Dadurch können zum einen Grundrisse von kleinen Räumen schnell erstellt werden. Zum anderen sind durch die beliebige Anzahl an Punkten auch

komplexere Strukturen wie ganze Stockwerke möglich zu modellieren. Auch Rundungen in Wänden lassen sich durch das Setzen von beliebig vielen Punkten im Polygonzug näherungsweise beschreiben. Grundsätzlich sind also alle zweidimensionalen Formen möglich, zumindest approximiert zu modellieren.

Neben dem Zeichnen vom Grundriss ist es möglich, weitere Polygonzüge hinzuzufügen, die als Hindernisse bzw. Löcher im Grundriss fungieren. Dies könnten zum einen weitere Wände in der Mitte eines Stockwerkes wie in Abbildung 2.2, oder Tische und Regale in einem Büro sein.

Weiterhin sollen Punkte, nachdem sie gezeichnet worden sind, einzeln verschiebbar sein, um Korrekturen nach dem Setzen von Punkten zu ermöglichen. Außerdem ist es hilfreich, ganze Polygonzüge zu verschieben, sodass ihre Form gleich bleibt. Dies wäre vergleichbar mit dem Verschieben von Tischen in einem Raum.

3.1.2. Das Triangulieren

Durch das Modellieren wurde eine Fläche erstellt, die trianguliert werden soll. Da das Triangulieren einer Fläche eine komplexe Aufgabe darstellt und es viele Bibliotheken gibt, die das Triangulieren übernehmen könnten, wäre es praktisch dafür eine Bibliothek zu verwenden. Folglich sollte bei der Wahl der Programmiersprache für die Implementierung des pFlow-Map-Generators darauf geachtet werden, dass es ein geeignetes Framework in derselben Programmiersprache für das Triangulieren gibt.

Nach dem Triangulieren der Fläche muss das Dreiecksgitter als Datei für pFlow exportiert werden. Dabei werden die Daten des Gitters ausgelesen und in einer definierten Struktur gespeichert, sodass diese mit pFlow kompatibel ist.

Zusätzlich sind die Formen der Dreiecke der Triangulierung von Bedeutung. Ihre Form darf nicht vollständig beliebig sein. Besonders verformte Dreiecke mit sehr spitzen oder stumpfen Winkeln können in pFlow zu Rechenproblemen führen. Daher ist es wichtig, dass die Dreiecke des Gitters eine möglichst gleichmäßige Form haben. Die Delaunay Triangulierung fokussiert sich beim Triangulieren genau darauf, weswegen sie für den pFlow-Map-Generator benutzt wird.

3.2. Delaunay Triangulierung

Die Delaunay Triangulierung ist eine von vielen Arten eine gegebene Fläche in eine endliche Menge an Dreiecken zu teilen und so diese Fläche approximiert darzustellen. Triangulierungen finden vermehrt Einsatz in der Finiten-Elementen-Methode (FEM). Für verschiedenste Simulationen werden Flächen im zwei- und dreidimensionalen Raum mithilfe der FEM durch eine begrenzte - finite - Anzahl von Elementen dargestellt. Dies ermöglicht es, dass näherungsweise Berechnungen auf diesen finiten Elementen gemacht werden können, welches vorher durch die unendliche Anzahl an Punkten in einer Fläche nicht möglich war. Im $\mathbb{R}^2$ füllt die FEM Flächen mit Drei- oder Vierecken. Da eine Karte für pFlow in einer triangulierten Form, also mit Dreiecken gefüllt, vorliegen muss, wird hier das Füllen einer Fläche mit Vierecken vernachlässigt. Die FEM findet auch Anwendung im $\mathbb{R}^3$ und dabei werden dreidimensionale Formen in Tetraeder oder Quader geteilt [Zwe20], aber dies wird ebenfalls vernachlässigt, da sich pFlow auf den $\mathbb{R}^2$ begrenzt.

Dreiecke mit besonders spitzen oder stumpfen Winkeln können bei Berechnungen von pFlow zu numerischen Problemen führen. Daher wird für die Triangulierung von pFlow-Karten die Delaunay Triangulierung verwendet. Diese Art der Triangulierung fokussiert sich darauf, dass gegebene Flächen in möglichst gleichmäßige Dreiecke ohne extreme Winkel geteilt werden. Beim Erstellen der Dreiecke achtet der Delaunay-Algorithmus auf das Maximieren des minimalen Innenwinkels der Dreiecke [She12]. Daraus folgt, dass die optimale Dreiecksform drei gleiche Winkel besitzt und damit ein gleichseitiges Dreieck ist. Dies bedeutet, dass diese Form die präferierte Dreiecksform der Delaunay Triangulierung ist und es größtenteils vermieden wird, Dreiecke mit besonders spitzen oder stumpfen Winkeln zu generieren.

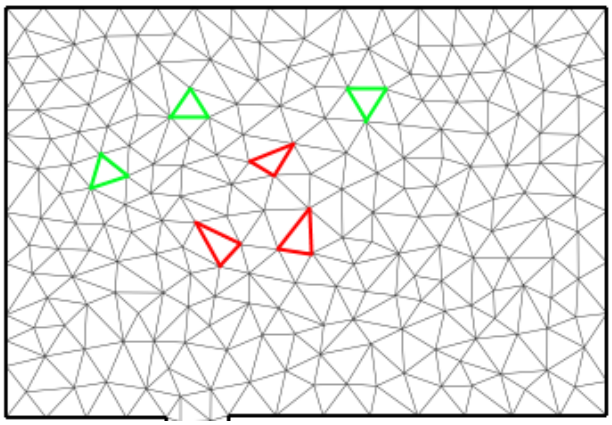

Abbildung 3.1: Ein Teil der Triangulierung aus Abbildung 2.2

Abbildung 3.1 zeigt einen Ausschnitt aus der Triangulierung aus Abbildung 2.2. In dieser Abbildung sind vereinzelt Dreiecke grün und rot markiert. Die grün markierten Dreiecke sind gleichseitig oder nahezu gleichseitig. In der Theorie bevorzugt die Delaunay Triangulierung diese Form. Diese Aussage wird damit bestätigt, dass im Beispiel viele Dreiecke zu sehen sind, die nahezu der bevorzugten Form entsprechen. Weiterhin sind auch drei Dreiecke rot markiert, die zeigen sollen, dass nicht alle Dreiecke in dieser Triangulierung gleichseitig sind. Es können auch leicht verformte Dreiecke auftreten, deren Innenwinkel ein wenig spitzer bzw. stumpfer sind, als die der gleichseitigen Dreiecke.

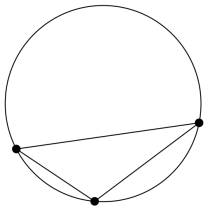

Abbildung 3.2: Ein Umkreis eines Dreiecks [Ber14]

Der Algorithmus der Delaunay Triangulierung maximiert, wie zuvor schon erwähnt, den minimalen Innenwinkel der Dreiecke. Die Gültigkeit von Dreiecken während einer Triangulierung wird dabei von der sogenannten Umkreisbedingung überprüft. Der Umkreis eines Dreiecks ist der eindeutige Kreis, der durch jede Ecke des Dreiecks verläuft. In Abbildung 3.2 ist der Umkreis eines Dreiecks zur Veranschaulichung dargestellt. Die Umkreisbedingung einer Triangulierung ist dann erfüllt, wenn jeder Umkreis nur die Punkte des eigenen Dreiecks berührt bzw. beinhaltet [Ber14].

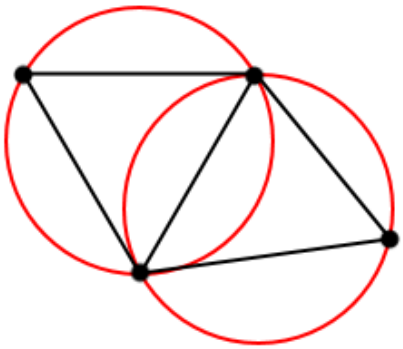

Abbildung 3.3: Zwei Dreiecke mit Umkreisen, Umkreisbedingung erfüllt

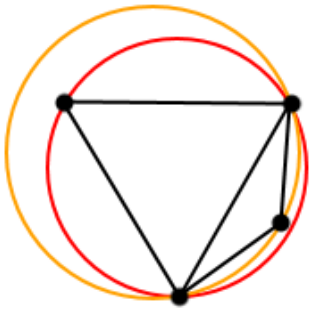

Abbildung 3.4: Zwei Dreiecke mit Umkreisen, Umkreisbedingung nicht erfüllt

Abbildung 3.3 zeigt zwei Dreiecke mit ihren Umkreisen. Wie zu sehen ist liegen in beiden Umkreisen nur die Ecken des eigenen Dreiecks. Die rechte Ecke des rechten Dreiecks liegt außerhalb des Umkreises des linken Dreiecks. Dies ist dasselbe für das rechte Dreieck und daher erfüllen diese beiden Dreiecke die Umkreisbedingung. Dabei ist anzumerken, dass beide Dreiecke eine vergleichbare Form eines gleichseitigen Dreiecks vorweisen und deswegen die Umkreise verhältnismäßig klein ausfallen.

In Abbildung 3.4 sind erneut zwei Dreiecke und ihre Umkreise dargestellt. Wie zu sehen ist, wurde das rechte Dreieck im Vergleich zu Abbildung 3.3 zu einem stumpfwinkligen Dreieck gestaucht. Durch diese Veränderung hat sich auch der Umkreis verändert, sodass der orangefarbene Umkreis entsteht. Durch den veränderten Umkreis ist zu sehen, dass beide Umkreise nicht nur die eigenen Dreiecksecken beinhalten, sondern auch jeweils eine Ecke des anderen Dreiecks. Da folglich beide Umkreise mehr als nur die eigenen Ecken enthalten, erfüllt diese Triangulierung nicht die Umkreisbedingung und ist als Delaunay Triangulierung ungültig. Auffällig dabei ist, dass obwohl die beiden Umkreise eine ähnliche Größe haben, das stumpfwinklige Dreieck auf der rechten Seite (in 3.4) sehr viel kleiner ist als das links anliegende Nachbardreieck. Dadurch wird deutlich, dass für eine einheitliche Struktur einer Triangulierung das Verhältnis zwischen Größe des Umkreises und Größe des Dreiecks minimiert werden soll. Bei gleichseitigen Dreiecken ist dies der Fall, wie in Abbildung 3.3 zu sehen ist. Aus diesen Gründen ist das gleichseitige Dreieck die präferierte Form von Dreiecken in einer Delaunay Triangulierung.

Diese Beispiele sollen veranschaulichen, welche Forderungen es an eine Delaunay Triangulierung gibt und wie diese mithilfe der Umkreisbedingung erfüllt werden. In Ergänzung dazu soll gezeigt werden, wieso das gleichseitige Dreieck die bevorzugte

Form für eine Delaunay Triangulierung ist.

3.3. Implementierung des pFlow-Map-Generators

Der pFlow-Map-Generator (pFMG) soll das Erstellen von Karten für pFlow so einfach und intuitiv wie möglich gestalten. Dabei soll ein Set an Funktionen gegeben sein, welche dem Benutzer helfen, viele verschiedene Sachen umzusetzen und ihn dabei nicht behindern. Weiterhin gibt es mit Hinsicht auf die Entwicklung Anforderung an den pFMG. Anfangs werden auf die Forderungen aus Entwicklersicht und danach auf die wichtigsten Features für die Benutzung eingegangen.

Zunächst soll es eine Version des pFMG geben, die mit allen gängigen Betriebssystemen kompatibel ist. Das macht die Entwicklung und Weiterentwicklung einfacher, wenn es nur eine Version der Anwendung gibt, an der Fehler behoben werden müssen. Andernfalls würde es verschiedene Ausführungen des Programmes für verschiedene Betriebssysteme geben, in denen plattformspezifische Fehler behoben werden müssen, was ein großer Aufwand ist. Um eine Plattformunabhängigkeit zu bieten, muss eine eine Programmiersprache gewählt werden, die entweder eine eigene plattformunabhängige virtuelle Maschine (VM) oder einen Interpreter als Laufzeitumgebung besitzt [J B06]. Beispielsweise würden sich Scala und Java als Sprache mit eigener VM und Python als Interpretersprache eignen.

Neben der Plattformunabhängigkeit soll der pFMG leichtgewichtige Merkmale haben. Dies bedeutet, dass alle Aktionen zeitlich performant ausgeführt werden sollen, um längere Wartezeiten für den Benutzer zu vermeiden bzw. zu minimieren. Damit ist zum Beispiel gemeint, dass der Map-Generator schnell startet, sich schnell alte Speicherstände laden lassen und Triangulierungen über komplexe Polygonzüge schnell erstellt werden können.

Schließlich wurde entschieden, den pFlow-Map-Generator in Python3 zu implementieren. Dies ist damit begründet, dass es für Python viele verschiedene und frei zugängliche Bibliotheken gibt, die sich mit der finiten Elemente Methode befassen. Außerdem lassen sich in Python leicht simple Benutzeroberflächen (englisch: User Interfaces, kurz: UIs) erstellen. Zuletzt legt Python einen großen Wert auf die einfache Implementierung. Zusammenfassend wird aus den oben genannten Gründen deutlich, dass sich Python als Implementierungssprache eignet.

3.3.1. Das Modellieren

Thema dieses Kapitels ist die UI und der Prozess, wie Grundrisse durch Polygone realisiert werden. Außerdem wird beschrieben, wie diese beiden Themen in der UI umgesetzt sind.

Die gesamte Benutzeroberfläche ist mit dem GUI-Toolkit (GUI = Graphical User Interface) Tkinter [Fou21] realisiert. Tkinter ist in der Standard-Bibliothek von Python enthalten und besitzt viele verschiedene Komponenten für eine simple GUI-Programmierung. Da für den pFMG nur eine einfache GUI geplant ist und außergewöhnliche Komponenten nicht benötigt werden, reicht Tkinter für die Realisierung der pFMG-GUI vollkommen aus.

Das Modellieren von Grundrissen ist über das Zeichnen von Polygonzügen realisiert. Dies bedeutet, dass einzelne Punkte definiert werden, welche mit Linien verbunden werden. Dadurch ist es möglich Formen von Räumen, Stockwerken oder anderen komplexen Orten zu erstellen.

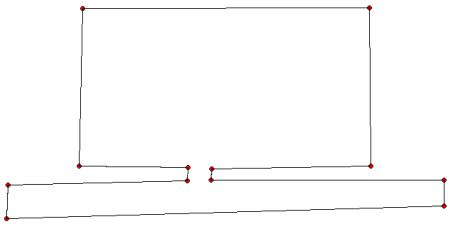

Abbildung 3.5: Einfacher Grundriss mit schrägen Wänden

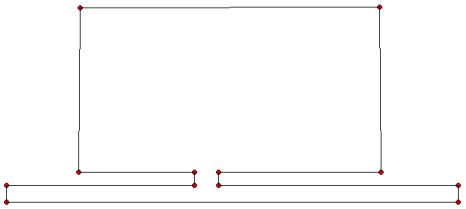

Abbildung 3.6: Selber Grundriss mit geraden Wänden

In Abbildung 3.5 lässt sich ein simpler Grundriss sehen, der einen Raum mit einer Verbindung zu einem Flur darstellt. Dieser Grundriss wurde im pFMG als Polygonzug modelliert. Die Punkte, die der Benutzer durch Klicken auf einer Leinwand hinzufügt, sind klein und rot gefärbt. Die Linien, die hier als Wände fungieren, werden automatisch zwischen Punkten direkt beim Einfügen der Punkte gezeichnet. Die Ausnahme dabei ist die Linie, zwischen dem ersten und letzten Punkt. Sie wird gezeichnet, wenn vom Benutzer bestätigt wird, dass der Polygonzug vollständig ist. Abbildung 3.5 zeigt deutlich, dass es schwierig sein kann, beim ersten Setzen der Punkte, diese so zu setzen, dass beispielsweise die Winkel der Ecken des Raumes exakt sind. Deshalb gibt es die Möglichkeit einzelne Punkte zu verschieben, um Ungenauigkeiten auszubessern. In Abbildung 3.6 ist beispielsweise der Grundriss aus Abbildung 3.5 zu sehen, nachdem

einzelne Punkte korrigiert wurden.

Der nachbearbeitete Grundriss in Abbildung 3.6 ist bis jetzt noch für eine pFlow Simulation sehr uninteressant. Das liegt daran, da der Raum bis jetzt noch komplett leer ist und keine Hindernisse enthält. Um dies zu ändern, werden dem Grundriss Hindernisse hinzugefügt. Die Hindernisse sollen Tische darstellen, sodass der Raum dem Aufbau eines Klassenzimmer in einer Schule ähnlich ist.

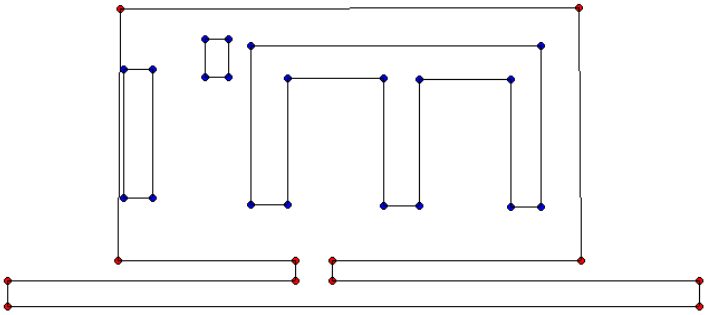

Abbildung 3.7: Grundriss eines Klassenzimmers mit Tischen

Abbildung 3.7 zeigt den modellierten Grundriss des oben erwähnten Klassenzimmers. Es fällt auf, dass drei zusätzliche Polygonzüge hinzugefügt wurden, deren Punkte blau und nicht rot gefärbt sind. Die blau gefärbten Punkte zeigen, dass diese ein Polygonzug für ein Loch in der Triangulierung darstellen. Dies bedeutet, dass diese Flächen keine Dreiecke nach der Triangulierung enthalten werden und deshalb ein unpassierbares Hindernis repräsentieren.

Der Aufbau des Klassenzimmers ist so zu interpretieren, dass auf der linken Seite direkt an der Wand eine Tafel oder Whiteboard steht und der einzelne Tisch der Tisch des Lehrers ist. Die große zusammenhängende Form stellt die zusammengestellten Tische für die Schüler dar. Auf diesem Grundriss könnte eine Simulation erstellt werden, um sicherzustellen, dass die Tische den Weg zum Ausgang nicht blockieren. Außerdem könnten verschiedene Anordnungen von Tischen getestet werden. Beispielsweise könnte das Klassenzimmer so umgestellt werden, dass sich die Tafel mit dem Tisch des Lehrers auf der rechten Seite des Zimmers befindet und ob diese Veränderung eine Auswirkung auf den Personenfluss in diesem Zimmer hat.

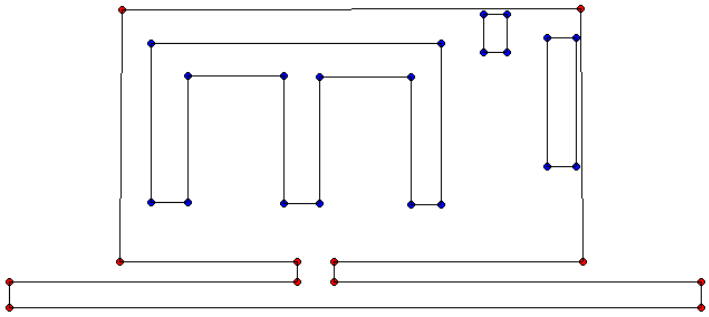

Abbildung 3.8: Klassenzimmer mit anderer Anordnung von Tischen

Polygonzüge lassen sich auch im Ganzen verschieben, sodass ihre Formen erhalten bleiben und nicht alle Punkte einzeln verschoben werden müssen. Die oben genannte Anordnung des Klassenzimmer mit der Tafel auf der rechten Seite lässt sich in Abbildung 3.8 sehen. Durch dieses einfache Verschieben von Polygonzügen lassen sich Karten mit verschiedenen Anordnungen von Hindernissen ohne viel Arbeitsaufwand erstellen. Schließlich kann pFlow so verschiedene Karten testen, um eine optimale Stellung von Hindernissen in Bezug auf den Personenfluss zu finden.

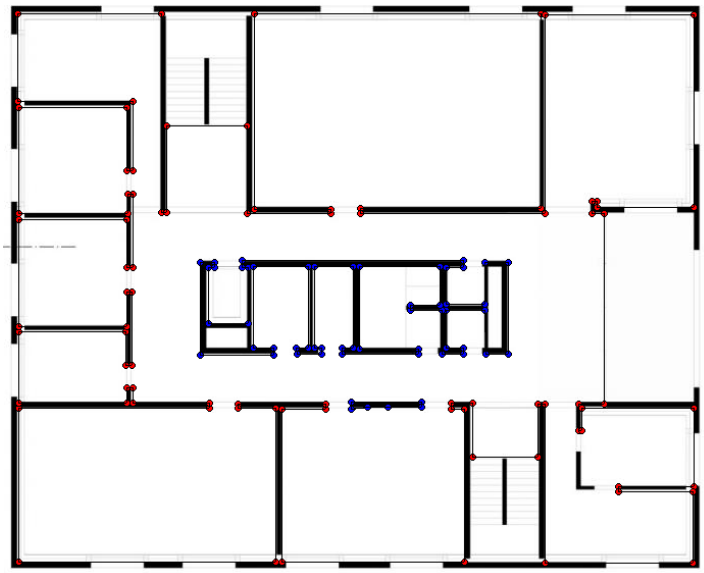

Abbildung 3.9: Auf einen Hintergrund [Gmb17] gezeichneter Polygonzug

Schließlich können für eine Modellierungen Grundrisse in den Hintergrund geladen werden. Dadurch ist das Platzieren der Punkte in die vorgegebenen Positionen sehr stark vereinfacht. In Abbildung 3.9 ist zu sehen, dass der Polygonzug für die Triangulierung aus Abbildung 2.2 auf den Grundriss von Abbildung 2.1 gezeichnet ist.

3.3.2. Das Triangulieren

Das Erstellen einer Triangulierung eines Grundrisses erfolgt mit der Bibliothek pyGIMLi. pyGIMLi ist eine open-source Python Bibliothek für Modellierungen in der Geophysik und enthält unter anderem auch Lösungen für die FEM im zwei- und dreidimensionalen Raum [Tea21]. Weiterhin ist es möglich, Delaunay Triangulierungen von Flächen eines Polygonzugs zu erstellen und dabei Löcher zu definieren, was pyGIMLi zur optimalen Bibliothek für das Triangulieren des pFMG macht.

```
1  main_poly = draw_polygon_by_polygon(points, _, _)
2
3  to_be_meshed = [_, main_poly]
4
5  for hole in holes:
6      drawn_hole = draw_polygon_by_polygon(hole, _, _, hole=True)
7      to_be_meshed.append(drawn_hole)
8
9  mesh = mt.createMesh(to_be_meshed)
```

Listing 3.1: Codestelle aus dem pFMG

In Listing 3.1 ist eine kurze Codestelle zu sehen, in der die Triangulierung gestartet wird. Für das Beispiel wurde der Code für bessere Verständnis angepasst, indem manche Variablen umbenannt, entfernt oder durch Unterstriche ersetzt wurden. Es werden die gezeichneten Polygonzüge umgewandelt, sodass pyGIMLi diese verarbeiten kann. Dies wird von der Methode $draw\_polygon\_by\_polygon$ in Zeile 1 für ersten Polygonzug gemacht. In Zeile 6 wird dies für jedes Hindernis-Polygon noch mal gemacht und diese werden anschließend in Zeile 7 in eine Liste eingefügt. Diese Liste beinhaltet Strukturen, die für die Triangulierung berücksichtigt werden sollen. Schließlich wird in Zeile 9 diese Liste als Parameter pyGIMLi übergeben, um eine Triangulierung zu erstellen.

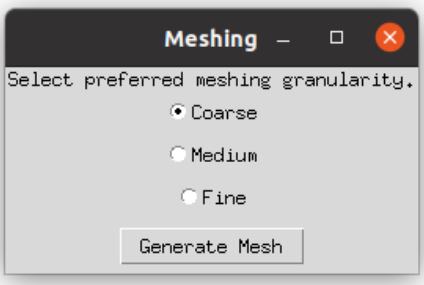

Abbildung 3.10: Fenster zum Einstellen der Feinheit

Zusätzlich kann die Größe der Dreiecke in einer Triangulierung durch pyGIMLi de-

finiert werden. Es kann eine Maximalgröße der Dreiecke definiert werden, sodass der Feinheitsgrad der Triangulierung verändert werden kann. Dafür wurde ein eigener grafischer Dialog entwickelt, der in Abbildung 3.10 zu sehen ist. Der Dialog bietet drei verschiedene Optionen. Abhängig von der Breite und Höhe des Polygonzugs wird eine maximale Dreiecksfläche berechnet. Dieser Maximalwert hängt zusätzlich von der gewählten Option ab und ist daher bei einer groben Einstellung ($Coarse$) größer als bei einer feinen ($Medium$ oder $Fine$).

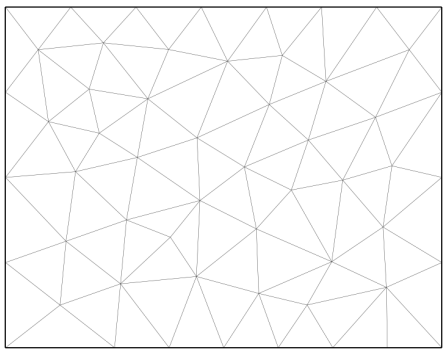

Abbildung 3.11: Eine grobe Triangulierung ($Coarse$)

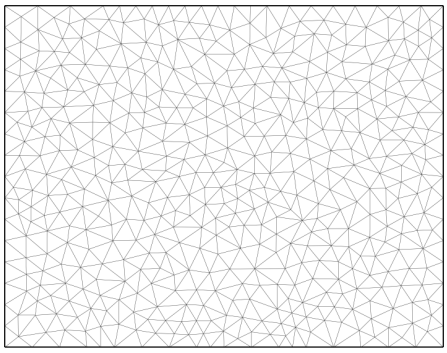

Abbildung 3.12: Eine mittlere Triangulierung ($Medium$)

Zum Vergleich dieser verschiedenen Einstellungen wurde ein Rechteck im pFMG gezeichnet und mit zwei verschiedenen Einstellungen trianguliert. In Abbildung 3.11 und Abbildung 3.12 sind die beiden Ergebnisse zu sehen. Die Triangulierung in Abbildung 3.11 wurde mit der Einstellung $Coarse$ erstellt und die in Abbildung 3.12 mit $Medium$. Dabei fällt auf, dass die Dreiecke in der linken Abbildung deutlich größer sind, als in der rechten. Falls eine noch feinere Triangulierung gewünscht ist, gibt es noch eine dritte Option ($Fine$), wie auch in Abbildung 3.10 zu sehen, mit der sehr feine Gitter erstellt werden können. Falls beispielsweise einen Grundriss mit schmalen Gängen trianguliert werden sollte, wäre es sinnvoll den Feinheitsgrad für diesen zu erhöhen, sodass die Gänge mit mehr als einer Reihe an Dreiecken gefüllt sind.

Schließlich wird das generierte Gitternetz nach dem Erstellen dem Benutzer gezeigt. Dadurch ist es möglich zu kontrollieren, ob das Gitter wie erwartet generiert wurde.

3.4. Aufgetretene Probleme

Während der Entwicklung des pFlow-Map-Generators sind vereinzelt Probleme aufgetreten, die ein unerwartetes Verhalten hervorrufen konnten oder umständlich umgangen werden mussten. Die zwei größten Probleme sollen im Folgenden mit ihrer Lösung erläutert werden.

Export in eine Datei

Nachdem ein Gitternetz über einen Grundriss erstellt wird, soll dieses Netz als Datei gespeichert werden können. Anschließend soll diese Datei von pFlow als Basis einer Simulation eingelesen werden können. pyGIMLi hat eingebaute Funktionen, um das erstellte Gitter in Dateien verschiedenen Typs zu speichern, wobei keiner dieser Datentypen mit pFlow kompatibel ist. Folglich müssen die Daten des Gitters anders ausgelesen werden und in einer korrekten Struktur abgespeichert werden.

Die Implementierung von pyGIMLi ist zwei geteilt. Die eine Hälfte ist in Python und die andere in C++ implementiert. Daraus folgt, dass bei manchen Funktionen, die in Python aufgerufen werden, ein C++-Datentyp zurückgegeben wird. Auf diesem Datentyp lassen sich mit Python nur eine begrenzte Menge an Funktionen ausführen. Das Gitter einer Triangulierung wird als C++ Datentyp zurückgegeben, der nicht die benötigten Funktionen unterstützt, sodass die benötigten Daten nicht ausgelesen werden können. Um dennoch an die Daten zu gelangen, muss ein umständlicher Weg gegangen werden, indem eine oben genannte Funktion benutzt wird, um das Gitter in einer Datei zu speichern. Das Dateiformat und pFlow sind nicht kompatibel, daher wird diese Datei nochmals vom pFMG eingelesen. Da die Datei nur ein Teil der Daten enthält, musste ein Algorithmus entwickelt werden, der die benötigten Daten aus den gegebenen Daten berechnet. Dies kann bei sehr feinen Triangulierungen bis zu 15 Minuten dauern. Diese Laufzeit ist sehr lang und fällt besonders auf, da die Laufzeiten der restlichen Funktionen vom pFMG so kurz gehalten sind, dass sie für den Benutzer fast nicht bemerkbar sind. Außerdem ist das ein riesiger Zeitaufwand, um Daten zu gewinnen, die im C++-Datentyp enthalten sind, aber nicht ausgelesen werden können.

Fehlerhafte Triangulierungen

Außerdem ist ein unerwünschtes Verhalten der pyGIMLi Bibliothek aufgefallen, bei dem fehlerhafte Triangulierungen erzeugt werden. Diese werden erzeugt, wenn Teile von Hindernissen außerhalb des Grundrisses liegen.

Abbildung 3.13 zeigt einen leicht veränderten Grundriss von Abbildung 3.7. Es fällt auf, dass der Polygonzug in Abbildung 3.13 auf der linken Seite zum Teil außerhalb

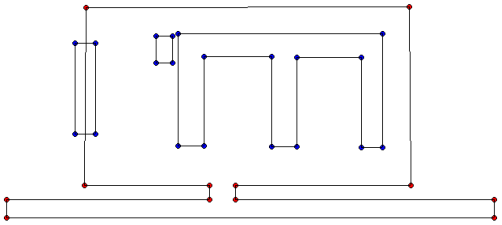

Abbildung 3.13: Grundriss des Klassenzimmers mit verschobenem Loch

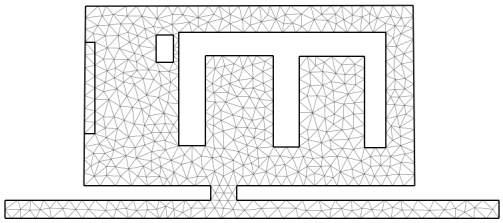

Abbildung 3.14: Fehlerhafte Triangulierung des Klassenzimmers

des Grundrisses liegt. Eigentlich sollte sich dadurch die Triangulierung nicht verändern und der außen liegende Teil ignoriert werden. In Abbildung 3.14 ist die Triangulierung von Abbildung 3.13 zu sehen. Man sieht, dass die Fläche des linken Polygonzugs nicht mehr als Loch in der Triangulierung behandelt wird. Der Polygonzug ist markiert, wird aber nicht mehr als Loch behandelt.

Abbildung 3.13 soll ein übertriebenes Beispiel dafür sein, was passieren könnte wenn sich aus Versehen des Benutzers Polygonzüge überschneiden. Dabei ist es egal, wie stark sich die Flächen überschneiden, da der Fehler auch bei einer minimalen Überschneidung auftritt. Beispielsweise könnte dieser Fehler auftreten, wenn versucht wird, im Beispiel des Klassenraums, die Tafel so nah wie möglich an die Wand zu platzieren. Da durch diesen Fehler Löcher nicht als Löcher trianguliert werden, ist die Triangulierung fehlerhaft.

3.5. Outlook

Der pFlow-Map-Generator enthält ein Menge an Features, die eine vollkommene Modellierung von Karten für pFlow Simulationen möglich macht. Dennoch gibt es viele Möglichkeiten, den pFMG weiter zu entwickeln. Es könnte weitere implizite Hilfen für den Benutzer geben, aber auch weitere Features, die das Modellieren und Triangulieren optimieren. Im Folgenden werden mögliche Weiterentwicklungen vorgestellt und begründet, wieso diese relevant sind.

3.5.1. Undo, Redo und Kopieren und Einfügen

Zwei klassische Feature-Paare, die in vielen Programmen implementiert sind, fehlen beide im pFMG. Deren Implementierung hat beim Entwickeln eine niedrige Priorität bekommen, da sie beim Benutzen anderer Features behilflich sein können, aber es Features mit höherer Priorität gab.

Viele Programme besitzen das klassische Undo und Redo Feature. Das Undo Feature ermöglicht es, dass jede Aktion rückgängig gemacht werden kann. Redo erlaubt außerdem das Widerrufen eines Undos. Dieses Paar an Funktionen findet großen Wert, wenn Benutzer einen Fehler machen. Wenn beispielsweise in einem Texteditor zu viel Text gelöscht wird und die Datei nicht gespeichert ist, wird durch das Undo der gelöschte Text gerettet und wieder eingefügt werden. Beim pFMG könnten irrtümlich Punkte verschoben werden, sodass der Polygonzug verformt wird. Und um dabei nicht den Punkt manuell wieder zurechtrücken zu müssen, wäre ein Undo sehr nützlich. Falls vom Benutzer ungewollt ein Undo ausgeführt wird, ist es zusätzlich sinnvoll ein Redo Feature zu haben.

Das zweite wichtige Feature-Paar, welches noch nicht implementiert wurde, ist das Kopieren und Einfügen. Falls Objekte in verschiedensten Programmen mehrfach gebraucht werden, ist es sinnvoll, diese nur einmal zu entwickeln und durch kopieren zu duplizieren. Beispielsweise treten bestimmte Codezeilen beim Programmieren vermehrt auf, sodass diese kopiert werden, anstatt sie noch einmal zu tippen. Falls Räume im pFMG mit vielen einzelnen, aber gleich großen Tischen gefüllt werden sollen, steht es nahe, diese zu kopieren. Im Beispiel mit dem Klassenraum (Abbildung 3.7) wurden die Tische der Schüler durch einen einzelnen großen Polygonzug beschrieben. Für eine Darstellung mit einzelnen Tischen wäre das Kopieren einzelner Tische nützlich.

3.5.2. Nachträgliches Hinzufügen und Löschen von Punkten

Polygonzüge können, nachdem sie vollendet sind, verändert werden, indem die einzelnen Punkte oder das ganze Polygon verschoben werden kann. Nachträglich Punkte eines Polygonzugs hinzufügen oder löschen geht dabei nicht. Daraus folgt, dass beim Zeichen eines Polygons genügend Punkte benutzt werden müssen, sodass nachträgliche Verformungen des Polygons, die ggf. mehr Punkte benötigen, möglich sind. Außerdem ist es nicht möglich, gezeichnete Löcher zu löschen, da keine Polygonzüge gelöscht werden können. Es besteht die Möglichkeit, diese vor der Triangulierung komplett aus dem Grundriss zu ziehen, da sie so beim Triangulieren ignoriert werden. Aber dies stellt

eher ein Workaround für ein wichtiges Feature dar.

3.5.3. Zoom

Das genau Modellieren ist auch mit einem Grundriss im Hintergrund als Vorlage schwierig. Mit dieser Variante können Punkte relativ genau gesetzt werden, wobei dies nicht genau genug sein kann. Daher sollte es eine Zoom-Funktion geben, mit der bestimmte Stellen vergrößert werden können, sodass die Punkte zielgenau platziert werden können.

3.5.4. Linien korrigieren

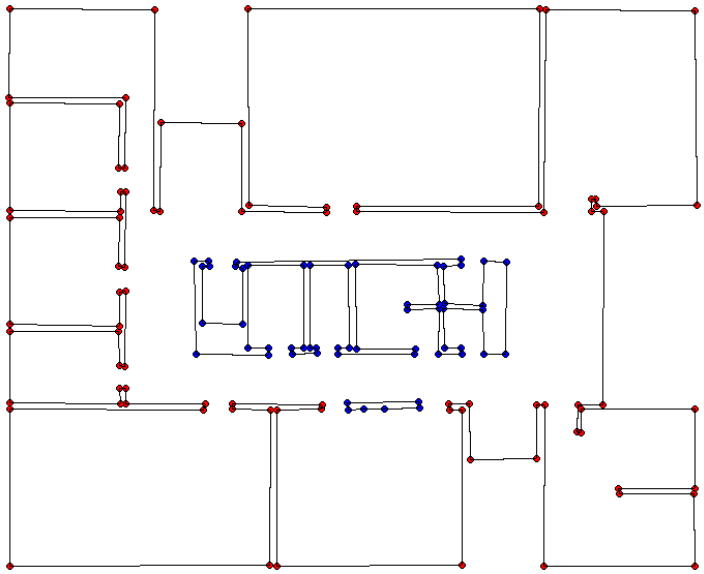

Abbildung 3.15: Modellierung des Stockwerks des O-Gebäudes

Schließlich fällt auf, dass es schwierig sein kann, Punkte so zu setzen, dass die Linien bzw. Wände exakt horizontal oder vertikal verlaufen. In Abbildung 3.15 ist ein Grundriss ohne jegliche Korrekturen nach dem Setzen der Punkte zu sehen. Auffällig dabei ist, dass viele Linien nicht genau horizontal oder vertikal verlaufen, obwohl sie es sollten. Viele Wände sind nicht parallel, sodass schnell deutlich wird, dass dieses Modell noch bearbeitet werden muss. Bei einem Grundriss mit dieser Anzahl von Punkten ist es sehr aufwendig, einzelne Punkte zu verschieben, sodass alle Wände und Winkel stimmen. Daher wäre es sinnvoll, wenn es eine Funktion gäbe, die automatisch erkennt, welche Wände wie verlaufen sollen und die Punkte entsprechend korrigiert.

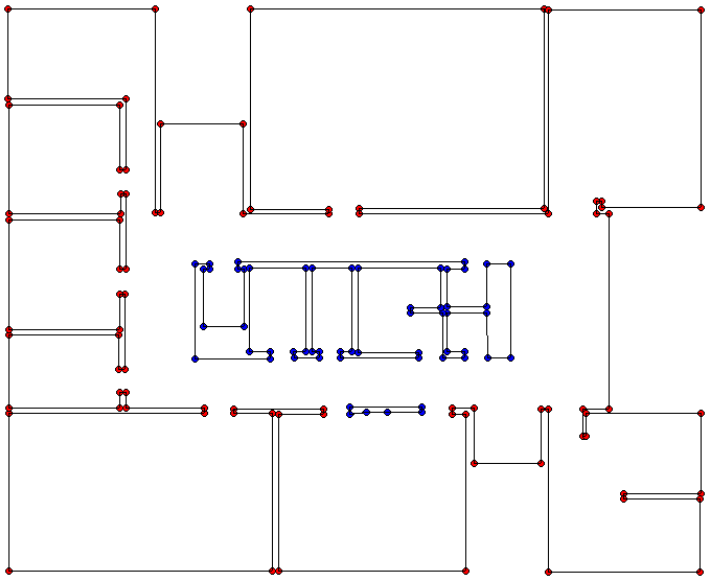

Abbildung 3.16: Korrigierter Grundriss

Abbildung 3.16 zeigt, wie eine Korrektur von Abbildung 3.15 aussehen könnte. Diese Korrektur wurde manuell durchgeführt, indem zeitaufwendig die Koordinaten in einer gespeicherten Datei verändert wurden. Dieses zu Automatisieren würde dem Benutzer sehr viel Zeit beim genauen Modellieren abnehmen.

4

Performance Test

pFlow ermöglicht eine Modifikation, dass eine Simulation mit Echtzeitdaten versorgt werden kann und so während Großveranstaltungen Prognosen über den Personenverlauf machen kann. Dabei ist es sehr bedeutend, dass diese Simulation schneller als Echtzeit ausgeführt wird, da sonst keine Prognose getroffen werden kann. pFlows Rechenzeit für Simulationen ist stark von der Komplexität der Triangulierung abhängig. Bei sehr feinen Triangulierungen ist es möglich, dass pFlow nicht schneller als Echtzeit Berechnungen machen kann. Deshalb sind Optimierungen für pFlow, die sich nur minimal auf die Laufzeit auswirken, schon von großer Bedeutung, da dadurch Prognosen schneller getroffen werden können. Aus diesen Performancegründen ist pFlow in C implementiert. C-Programme haben die Eigenschaft, dass sie eine schnelle Ausführzeit haben [Bri20], aber benötigen dafür eine manuelle Speicherverwaltung des Programms beim Programmieren. Fraglich ist dabei, ob pFlow nicht in einer höheren Programmiersprache implementiert werden kann, um das Programmieren bei ähnlicher Performance zu vereinfachen. Höhere Programmiersprachen haben meist eine höhere Ausführungszeit als C [Bri20], aber könnten ggf. trotzdem Simulationen schneller als Echtzeit berechnen. Um dies zu untersuchen, soll eine zeitintensive Funktion von pFlow in Python und C implementiert werden, um die Laufzeiten zu vergleichen.

Durch vorherige Recherche wurde deutlich, dass C gleiche Anweisungen schneller ausführt als Python [Unb]. Dennoch stellt sich die Frage, ob eine Implementierung in Python auch Simulationen schneller als Echtzeit berechnen kann, da eine leicht schlechtere Performance bei einfacherer Implementierung in Kauf genommen werden könnte.

4.1. Getestete Funktion

Zunächst muss eine Funktion ausgewählt werden, mit der der Performancetest durchgeführt wird. Dafür wurde eine Analyse von pFlow während einer Simulation erstellt, in dem sich die Ausführungszeiten pro Funktion ablesen lassen. Dabei ist eine Funktion aufgefallen, die einen größeren Teil der gesamten Laufzeit beansprucht, obwohl ihre Aufgabe weniger komplex ist. Die Aufgabe dieser Funktion ist es für einen gegeben Punkt, das dazugehörige Dreieck in der Triangulierung zu finden. Bei Triangulierungen mit einer geringen Anzahl an Dreiecken stellt diese Aufgabe kein Problem dar. Bei einer sehr feinen Triangulierung kann dies sehr viel Zeit kosten.

Damit diese beiden Funktionen vergleichbar sind, wurde der Algorithmus der Funktion vorher festgelegt. Dieser wurde dann in beiden Programmiersprachen implementiert, sodass der einzige Unterschied die Sprache der Implementierung ist. Auf mögliche Optimierungen des Algorithmus wurde hier verzichtet, da diese in beiden Programmiersprachen möglich sind. Außerdem ist der Vergleich der Ausführungszeit der Programmiersprachen einfacher, wenn die Algorithmen so einfach wie möglich sind.

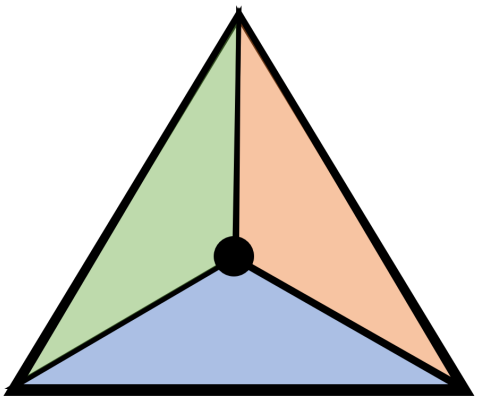

Abbildung 4.1: Aufteilung in 3 Dreiecke mit Punkt innerhalb des Dreiecks

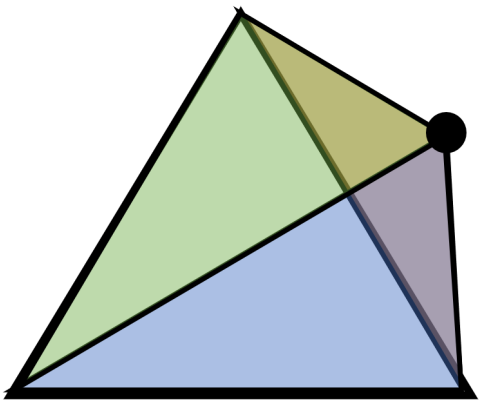

Abbildung 4.2: Aufteilung in 3 Dreiecke mit Punkt außerhalb des Dreiecks

Der Algorithmus durchläuft für die Prüfung pro Punkt alle Dreiecke des Gitternetzes. Dabei wird pro Dreieck überprüft, ob sich der gegebene Punkt innerhalb dieses Dreiecks befindet. Dies wird durch Berechnung von Teilflächen überprüft. Mithilfe des Punktes und den Ecken des Dreiecks werden drei Flächen definiert. Diese Flächen werden jeweils durch zwei Eckpunkte des Dreiecks und dem gegeben Punkt definiert. So eine Aufteilung ist in Abbildung 4.1, wenn sich der Punkt innerhalb des Dreiecks befindet. Gegenteilig dazu zeigt Abbildung 4.2, wie eine Aufteilung aussähe, wenn der

Punkt sich außerhalb des Dreiecks befände. Um zu prüfen, ob sich der Punkt innerhalb des zu testenden Dreiecks befindet, werden die Flächeninhalte der drei Teilflächen berechnet und aufsummiert und mit der Fläche des großen Dreiecks verglichen. Falls die Flächeninhalte gleich groß sind, liegt der Punkt innerhalb des Dreiecks. Beim Berechnen der Flächeninhalte können Ungenauigkeiten auftreten, sodass die Flächeninhalte nicht exakt gleich sind, obwohl der Punkt innerhalb des Dreiecks liegt. Deshalb wurde festgelegt, dass Flächen sich um 10^{-7} unterscheiden dürfen und trotzdem als gleich groß gelten.

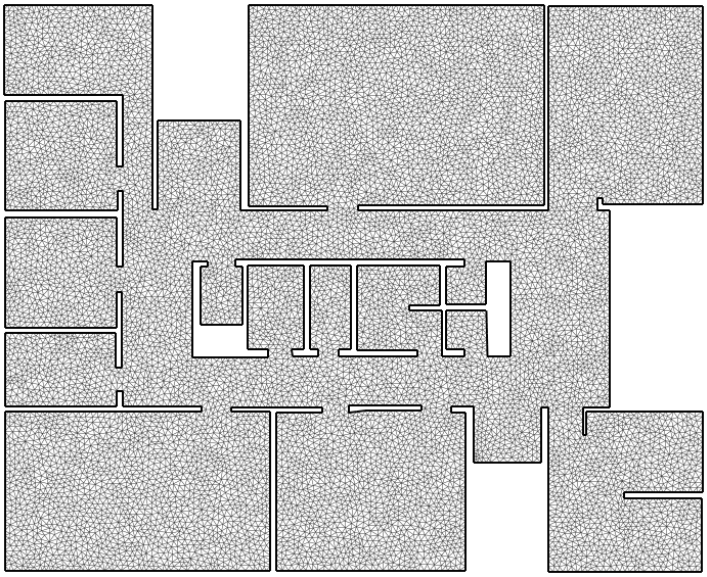

Abbildung 4.3: Feine Triangulierung als Basis für den Performancetest

Als Gitter für den Benchmark wird eine feinere Triangulierung des Grundrisses aus Abbildung 2.1 benutzt, die in Abbildung 4.3 dargestellt ist. Dieses Gitter besteht aus circa 13.000 Punkten und 25.000 Dreiecken. Vor dem Test berechnen beide Implementierungen die Mittelpunkte aller Dreiecke. Für diese Punkte werden im Benchmark die passenden Dreiecke gesucht.

Damit die Laufzeiten beider Implementierungen vergleichbar sind, werden sie auf demselben System nacheinander ausgeführt. Um weiterhin Ungenauigkeiten zu eliminieren, werden die Laufzeiten beider Funktionen mehrmals gemessen.

4.2. Ergebnisse

In Tabelle 4.1 sind die Laufzeiten der beiden Implementierungen in Sekunden zu sehen. Jede Implementierung wurde fünfmal ausgeführt und die Zeiten sind für eine bessere Übersicht innerhalb einer Spalte aufsteigend sortiert. Die C-Funktion benötigt im Durchschnitt circa 7,4 Sekunden, um für 25.000 Punkte das passende Dreieck zu suchen, während die Python-Variante im Durchschnitt 499,9 Sekunden für dieselbe Aufgabe braucht. Damit ist die Implementierung in Python ungefähr 68-mal langsamer.

C	Python
7,23	473,869
7,26	503,521
7,36	503,611
7,49	504,439
7,69	514,247

Tabelle 4.1: Laufzeiten in Sekunden in den verschiedenen Programmiersprachen

4.3. Optimierungen

Der Algorithmus, dem beide Implementierungen nachempfunden sind, ist mit Absicht sehr simpel gehalten. Weiterhin interessant ist, wie sich mögliche Optimierungen auf die einzelnen Laufzeiten auswirken. Deshalb werden beide Implementierungen mit jeweils einer Optimierung verändert. Alle Optimierungen des Algorithmus könnten auch in der jeweils anderen Implementierung benutzt werden. Hier soll gezeigt werden, wie sich eine einfache Optimierung auf die Laufzeit der Python-Funktion auswirkt und wie die C-Funktion für die Benutzung innerhalb von pFlow optimiert wurde.

Innerhalb der Python-Implementierung wird die meiste Rechenzeit dafür benötigt, um die verschiedenen Flächen der Dreiecke zu berechnen und dadurch zu bestimmen,

ob der Punkt innerhalb dieses Dreiecks liegt. Dieses könnte optimiert werden, sodass die Flächen nicht jedes Mal berechnet werden müssen, sondern nur dann, wenn der Punkt auch in der Nähe des Dreiecks liegt.

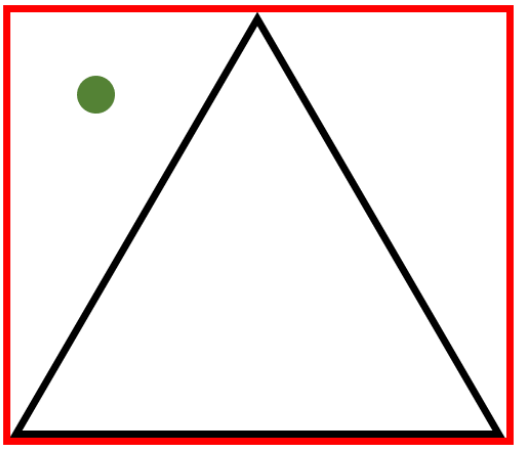

Abbildung 4.4: Visualisierung der Optimierung

Um diese Idee zu realisieren, wird um ein Dreieck ein Rechteck gezeichnet, welches dieselbe Höhe und Breite wie das Dreieck besitzt, wie es in Abbildung 4.4 gezeigt ist. Falls der gegebene Punkt nicht innerhalb dieses Rechtecks liegt, kann er auch nicht innerhalb des Dreiecks liegen. Folglich überspringt der Algorithmus in diesem Fall eine erhebliche Menge an Berechnungen und spart damit Rechenzeit. Der Fall, dass ein Punkt innerhalb des Rechtecks, aber außerhalb des Dreiecks liegt, ist in Abbildung 4.4 dargestellt. In diesem Fall würde der Algorithmus die Flächen zur Überprüfung berechnen und herausfinden, dass der Punkt nicht innerhalb des Dreiecks liegt. Schließlich sollten mit dieser Optimierung unnötige Flächenberechnungen übersprungen werden, sodass eine große Verringerung der Rechenzeit erwartet werden kann.

Python	Python optimiert
473,869	229,877
503,521	232,890
503,611	240,485
504,439	248,161
514,247	249,718

Tabelle 4.2: Gegenüberstellung der Laufzeiten in Sekunden der optimierten Python-Variante mit der unoptimierten

In Tabelle 4.2 ist eine Gegenüberstellung der Laufzeiten zwischen optimiertem und nicht optimiertem Algorithmus in Python zu sehen. Dabei fällt auf, dass der optimierte Algorithmus doppelt so schnell, wie der nicht optimierte ist. Eine Halbierung der Laufzeit ist eine enorme Steigerung der Schnelligkeit. Aber dennoch ist die optimierte Implementierung in Python 32-mal langsamer als die nicht optimierte Implementierung in C, wodurch nochmals der enorme Unterschied der Laufzeiten zwischen C und Python deutlich gemacht wird. Außerdem könnte dieselbe Optimierung auch in C implementiert werden und es würde eine ähnliche Verbesserung der Performance erwartet werden.

Schließlich ist die von pFlow genutzte Implementierung sehr stark optimiert. Die Optimierung besteht darin, dass ein Gitter aus Quadraten zusätzlich zu der Triangulierung erstellt wird und diese übereinandergelegt werden. Dabei werden Zusammenhänge zwischen einzelnen Quadraten und Dreiecken, die mit dem Quadrat eine Schnittfläche besitzen, gebildet. Dadurch wird für einen Punkt erst das Quadrat gesucht, in dem er liegt, sodass der Punkt in einem der Dreiecke liegen muss, die eine Schnittfläche mit dem Quadrat bilden. Anschließend ist die Anzahl an Flächen, die berechnet werden muss, um das Dreieck zu dem Punkt zu lokalisieren, sehr niedrig. Dadurch, dass nur noch wenige Fläche berechnet werden müssen, spart diese Optimierung sehr viel Rechenzeit. Dies wird außerdem noch verstärkt, weil durch das Schließen von einem Quadrat auf eine Menge an Dreiecken eine Iteration über die gesamte Anzahl an Dreiecken ausgeschlossen ist.

C	C optimiert
7,23	0,026
7,26	0,027
7,36	0,028
7,49	0,029
7,69	0,037

Tabelle 4.3: Gegenüberstellung der Laufzeiten in Sekunden der optimierten C-Variante mit der unoptimierten

In Tabelle 4.3 sind die Laufzeiten der Optimierung mit dem nicht optimierten Algorithmus gegenüber gestellt. Die Zeiten der optimierten Variante sind im Vergleich zu der nicht optimierten sehr niedrig. Die Optimierung ist um circa 252-mal schneller als die ursprüngliche Variante.

4.4. Auswertung

Durch diesen Benchmark sollte untersucht werden, ob eine Implementierung von pFlow in Python mit leichter Verschlechterung der Performance möglich wäre. Durch den stichprobenartigen Test der Implementierung einer Funktion in die beiden Programmiersprachen wurde gezeigt, dass Python zum Ausführen derselben Funktion 68-mal so lange benötigt, wie das Gegenstück in C. Selbst eine optimierte Variante der Implementierung in Python ist noch langsamer, als die Basisimplementierung in C.

Demzufolge könnte prognostiziert werden, dass eine komplette Implementierung von pFlow in Python um denselben Faktor langsamer wäre. Eine Implementierung in Python wäre für den Programmierer zwar einfacher, da zum Beispiel die Speicherverwaltung automatisiert ist. Aber da für pFlow die Performanz von höchster Bedeutung ist, würde sich diese aufgrund der Implementierung in Python sehr stark verschlechtern.

5

Fazit

Der pFlow-Map-Generator, der in dieser Arbeit entwickelt wurde, füllt eine entscheidende Lücke im Workflow von pFlow. Es lassen sich einfach Grundrisse in Karten umwandeln, sodass sie nahtlos für Simulationen in pFlow genutzt werden können. Dafür stehen dem Benutzer die wichtigsten Features zur Verfügung, sodass viele verschiedene Formen von Karten ohne Probleme erstellt werden können.

Außerdem wurde stichprobenartig untersucht, ob pFlow bei gleicher Performance in Python implementiert werden könnte. In dieser Untersuchung wurde deutlich, wie viel schneller die Programmiersprache C Operationen und Funktionen ausführen kann als Python. Die Beispielfunktion in C wurde 68-mal schneller ausgeführt, als die Implementierung in Python. Der Faktor 68 ist so hoch, dass sicher gesagt werden kann, dass eine pFlow Implementierung in Python nicht den Ansprüchen auf eine schnelle Ausführungszeit gerecht werden kann.

Schließlich kann hinterfragt werden, ob der pFMG wirklich hätte in Python implementiert werden soll. Python-Code ist zwar sehr einfach zu schreiben und lesen, aber ab einer gewissen Komplexität ist Python-Code sehr unübersichtlich, sodass dieser sehr gut kommentiert und dokumentiert werden muss. Python verzichtet auf statisches Typsystem, welches das Übergeben und Benutzen von Variablen erschwert. Im Code ist es schnell zu übersehen, welche Variablen welchen Typ haben und wohin diese übergeben werden sollen. Java oder Scala könnten dabei Abhilfe schaffen, da Datentypen zur Compile-Zeit kontrolliert werden.

Außerdem hätte eine Bibliothek zum Triangulieren selbst entwickelt werden können, sodass die Wartezeit beim Exportieren einer Triangulierung in eine Datei vermieden werden könnte. Dies wäre sehr nützlich, hätte aber den Arbeitsaufwand einer Bachelorarbeit übertroffen. Zusammenfassend konnte ein großer Zeitaufwand beim Entwickeln durch Python gespart werden, indem der Code einfach zu formulieren ist und es eine Vielzahl an kostenlosen Bibliotheken gibt, die genutzt werden können. Letztlich ist es eine gute Wahl gewesen, den pFMG in Python zu implementieren. Hätte man eine andere Programmiersprache zum Implementieren gewählt, wären wahrscheinlich andere programmiersprachenspezifische Probleme aufgetreten.

Literatur

[Unb] Unbekannt. “Which programs are fastest?” In: (). Abgerufen: 15.08.2021. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html.

[Ber14] Emo Welzl Bernd Gärtner Michael Hoffmann. “Chapter 6 - Delaunay Triangulations”. In: *Computational Geometry* (10.01.2014). Abgerufen: 11.08.2021. URL: https://www.ti.inf.ethz.ch/ew/courses/CG13/lecture/Chapter%206.pdf.

[Bri20] Gunavaran Brihadiswaran. “A Performance Comparison Between C, Java, and Python”. In: (23.07.2020). Abgerufen: 15.08.2021. URL: https://medium.com/swlh/a-performance-comparison-between-c-java-and-python-df3890545f6d.

[Fou21] Python Software Foundation. “tkinter — Python interface to Tcl/Tk”. In: (12.08.2021). Abgerufen: 13.08.2021. URL: https://docs.python.org/3/library/tkinter.html.

[Gmb17] MGF Architekten GmbH. “Loggia zum See”. In: (29.03.2017). Abgerufen: 02.08.2021. URL: https://www.german-architects.com/de/architecture-news/reviews/loggia-zum-see.

[J B06] N. Horspool J. Bishop. “Cross-Platform Development: Software that Lasts”. In: (9.10.2006). Abgerufen: 13.08.2021. URL: https://ieeexplore.ieee.org/abstract/document/1707631.

[She12] Jonathan Richard Shewchuk. “Lecture Notes on Delaunay Mesh Generation”. In: (5.02.2012). Abgerufen: 02.08.2021. URL: https://people.eecs.berkeley.edu/~jrs/meshpapers/delnotes.pdf.

[Tea21] GIMLi Development Team. “pyGIMLi - Geophysical Inversion & Modelling Library”. In: (12.07.2021). Abgerufen: 14.08.2021. URL: https://www.pygimli.org/about.html#sec-gimli.

[Zwe20] Monika Zwettler. “Was ist eigentlich FEM?” In: (13.10.2020). Abgerufen: 11.08.2021. URL: https://www.konstruktionspraxis.vogel.de/was-ist-eigentlich-fem-a-969326.