HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT DRESDEN
UNIVERSITY OF APPLIED SCIENCES

Modul G384 – 3D-Stadtmodelle

Automatische Dachflächengenerierung

mittels gewichtetem Straight-Skeleton-Algorithmus

Theodor Rieche

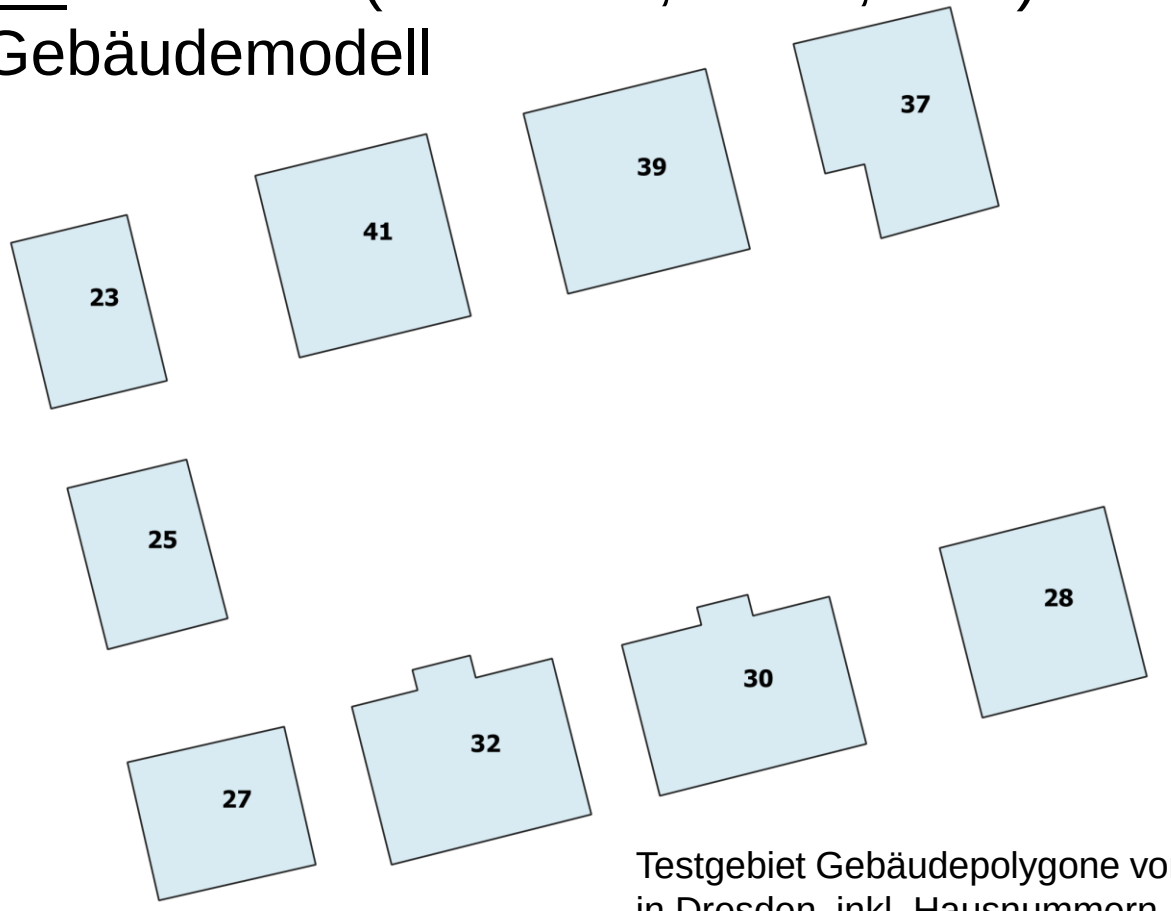
Fakultät Geoinformation
HTW Dresden
Masterstudiengang Geoinformatik / Management

Gliederung

Motivation

Motivation

gegeben: 2D Hausumringe/ Grundrisse
gesucht: Dachform (Firstlinien, Kehle, Grat) für 3D Gebäudemodell



Testgebiet Gebäudepolygone von OpenStreetMap in Dresden, inkl. Hausnummern

Motivation

- Ergebnis: eine Skeleton-Geometrie

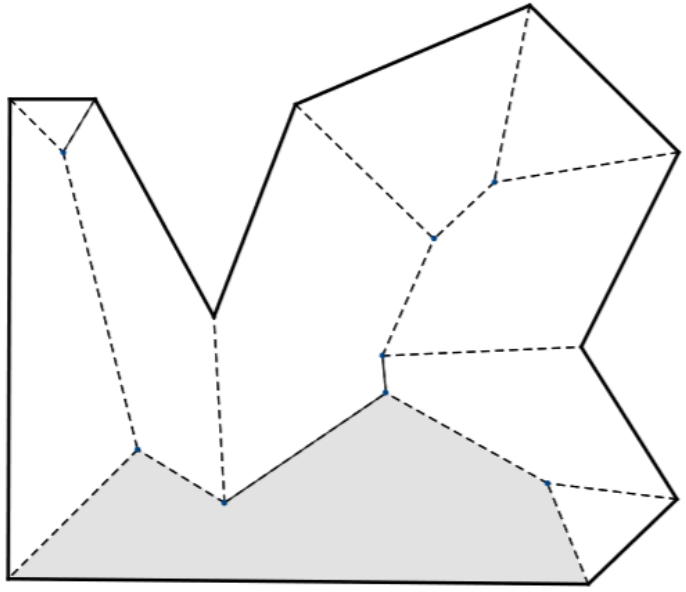
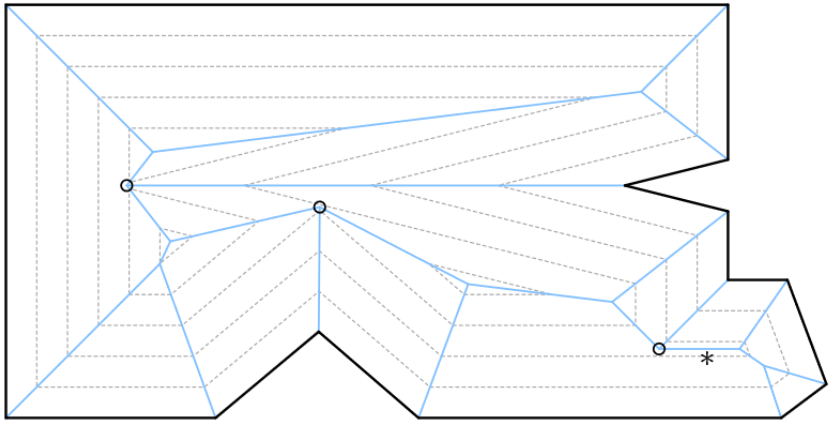


Abbildung 2.2: Straight Skeleton

Quelle [3]

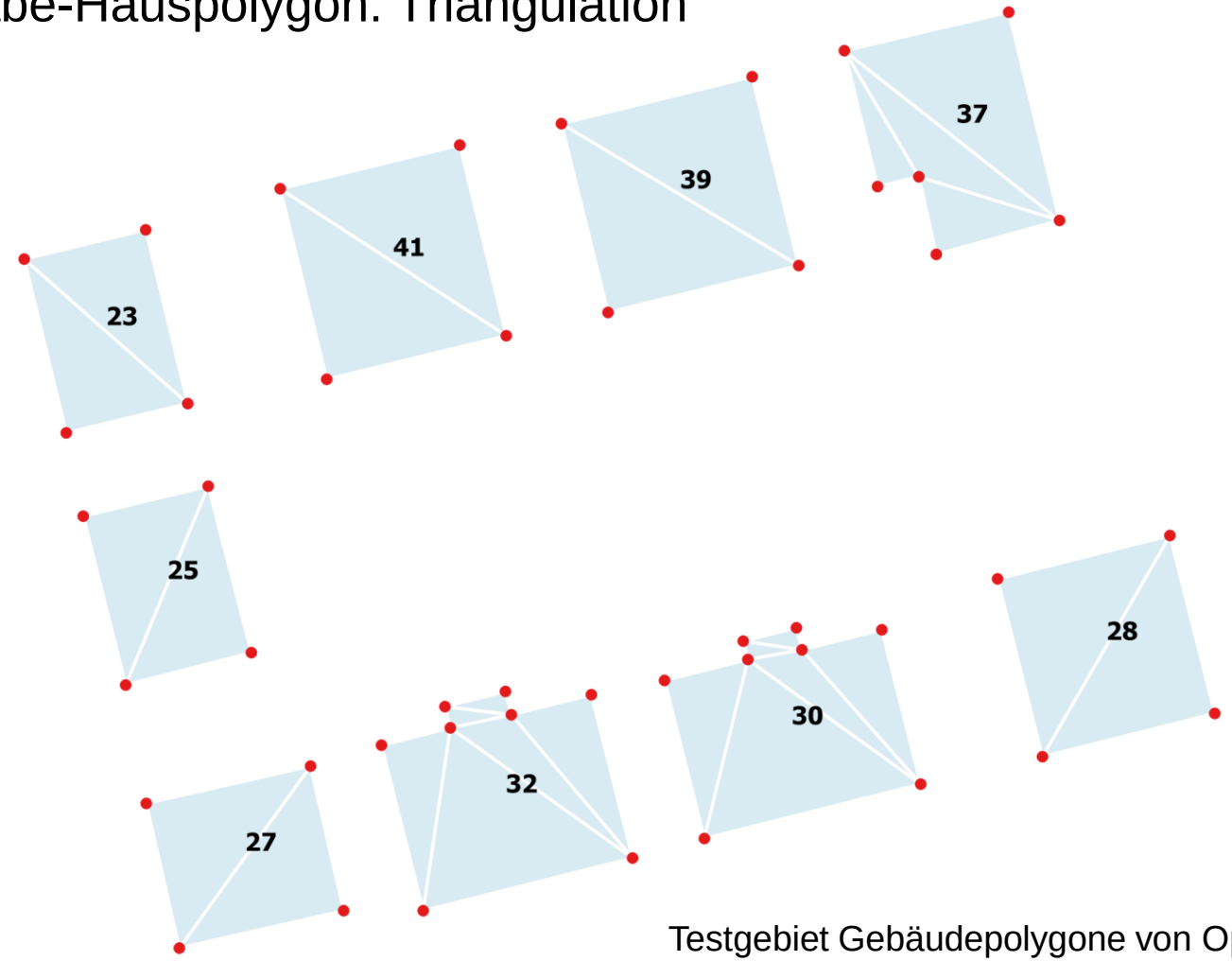


Quelle [5]

Algorithmus

Algorithmus „Straigh Skeleton“

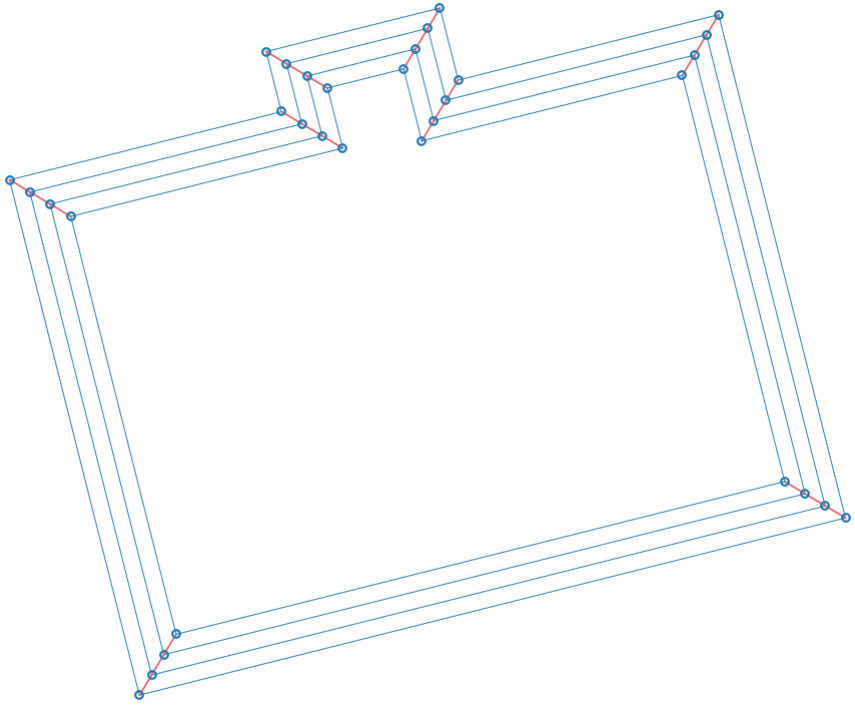
- Eingabe-Hauspolygon: Triangulation



Testgebiet Gebäudepolygone von OpenStreetMap in Dresden, inkl. Hausnummern & Triangulation

Algorithmus

- Alle Kanten (Wavefront-Edges) wanden parallel in Richtung Inneres des Polygons
  - **Ungewichtet**: für alle Kanten gleiche Geschwindigkeit/ Gewicht = 1
  - **Gewichtet**: Gewicht repräsentiert Geschwindigkeit, 0 entspricht keine Bewegung
- Schrumpfungsprozess, Wavefronts durchlaufen Polygon



Algorithmus

- Im Schrumpfungsprozess: Events verursacht
- Angezeigt durch zusammenklappendes Dreieck der Triangulation (Wann ist Fläche des Dreieckes gleich null?)

EDGE-Event: eine Wavefront-Kante schrumpft auf Länge Null

FLIP-Event: ein Knoten trifft auf eine Dreiecks-Seite (spoke)

SPLIT-Event: ein Knoten trifft auf eine Wavefront-Kante (teilt diese)

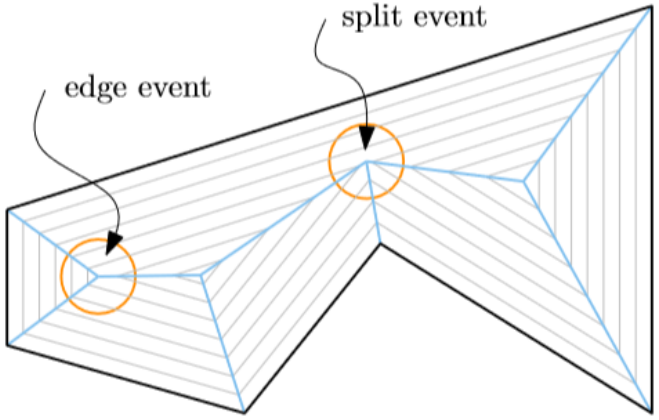


Figure 2: Edge and split events.

Quelle [4]

Algorithmus

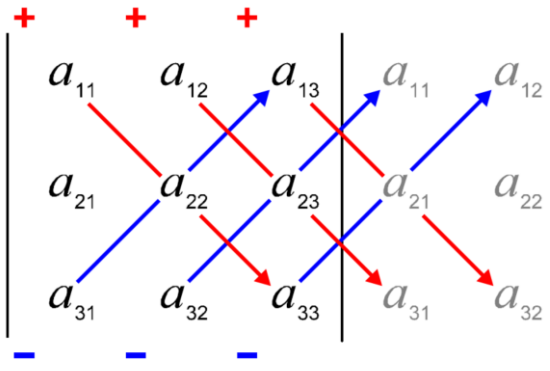
Berechne initiale Bewegungsvektoren der Knoten anhand Kantengewichten

1. für alle aktiven Dreiecke (Triangulation) berechne Collapse Time
2. Nimm das Dreieck mit der kleinsten Collapse Time
3. Führe den Schrumpfungsprozess um diese Zeit durch/ Bewege Dreiecke, Knoten und Kanten nach Innen um Zeit *min(Collapse Time)*
4. Analysiere und bearbeite das Event
5. Passe gegebenenfalls benachbarte Dreiecke, Kanten, Knoten an, berechne bei EDGE und SPLIT neuen Bewegungsvektor

Solange Summe aller aktiven Dreieck-Flächeninhalte > 0,
wiederhole Schritte 1. - 5.

Mathematische Grundlagen

- Vektorrechnung/ Matrizen-Rechnung
- Normalenvektor
- Triangulation eines Polygons
- Orientierung eines Polygons/ Liste von Punkten (Quelle [1])
  - *Summe über alle Kanten: (x2-x1)(y2+y1). Wenn Ergebnis positiv → im Uhrzeigersinn. Wenn negativ → gegen UZS*
- Flächeninhalt Dreieck: Satz des Heron
- Strecke: Satz des Pythagoras
- Determinante 3x3 Matrix: Regel von Sarrus



Quelle [wikipedia]

Implementierung

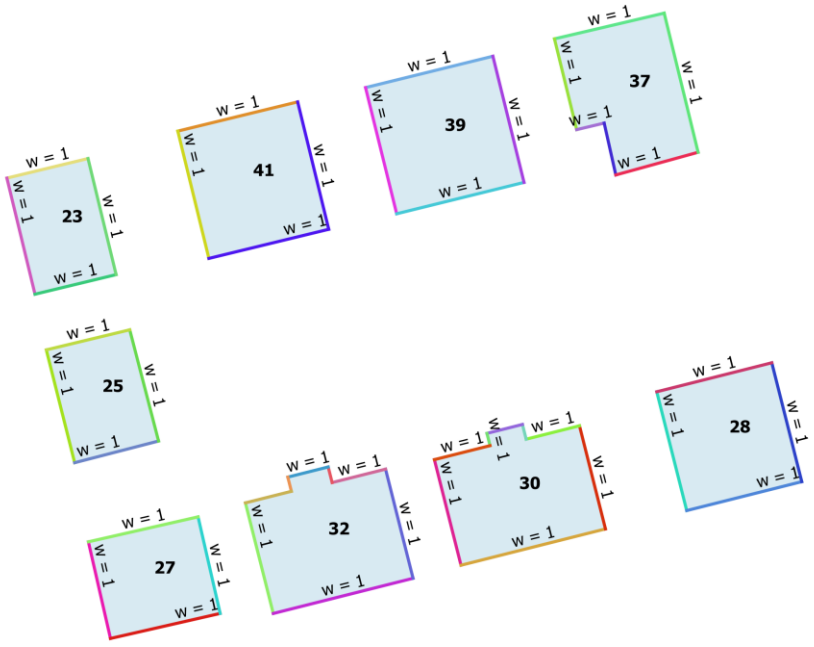
Implementierung

- Umgesetzt in Python 2.7
- OpenSource-Ansatz
- Software-Einsatz: PyScripter, QGIS, notepad++, Excel, Online JSON Viewer
- 1.740 Zeilen erstellt (inkl. Kommentare und Leer-Zeilen)
- Triangulation wurde verwendet von: John Burkardt, 2016, unter Lizenz: GNU LGPL license (Quelle: [2])
- Import von GeoJSON in kartesischen metrischen CRS
  - Attribut ID erforderlich
- Export als GeoJSON
- Importierte Bibliotheken:

```
import os
import sys
import math
import json
import csv
import copy

import numpy as np


import polygon_triangulate as ptri

import time
from time import gmtime, strftime
```

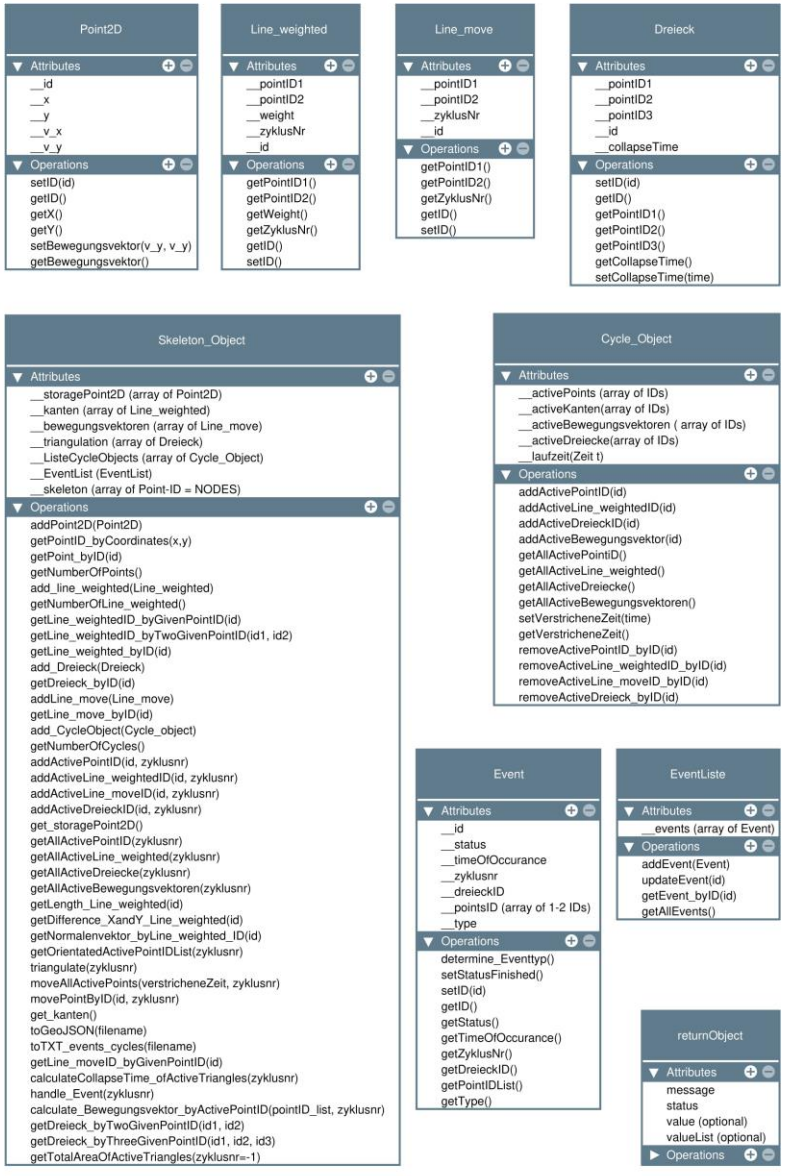
Implementierung

- Objekt-orientiert
- Abgebildet in 9 Objekten und 2 separaten Funktionen:
- Pre-Prozessierung Polygon zu Polylines
  def **GeoJSON_Polygon2LineSegment**(inputFile, outputFile):
- Linien erhalten Standard-Gewicht = 1 → manuell editieren



- Start des Programmes
  def **straight_skeleton_weighted**(inputFile):

Implementierung



- Objekt „**Skeleton_Object()**“ als Rahmen
- Objekt „**Cycle_Object()**“ pro Zyklus / Schrumpfungsschritt zwischen zwei Events → alle aktiven Geometrien
- Geometrien als „**Point2D()**“, „**Line_weighted()**“, „**Line_move()**“, „**Dreieck()**“
- Events in „**Event()**“ in „**EventListe()**“
- „**returnObject()**“ zur Kommunikation zwischen Objekten/ Resultate

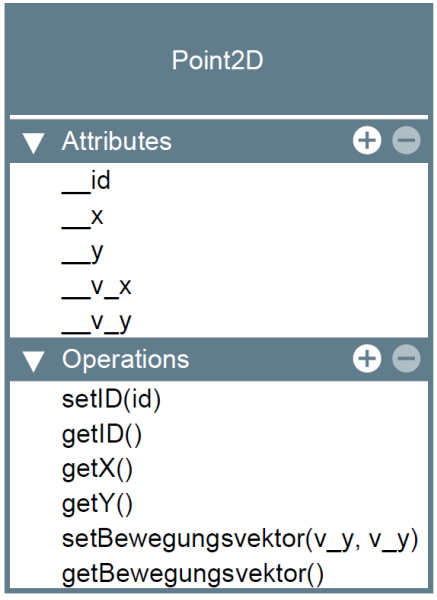
- Keine Vererbung zwischen Line_weighted() und Line_move()

Implementierung

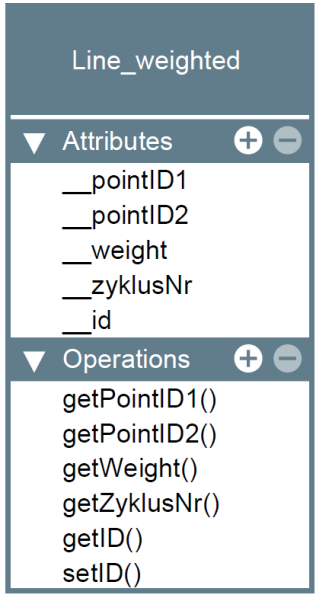
Modellierung / Repräsentation der Geometrie

- Objekt-basiert, topologisch strukturiert (Lagebeziehungen)
- Einzig Point2D(x,y) hält Koordinaten
- Explizite Verknüpfung zu Linien und Dreiecken via IDs

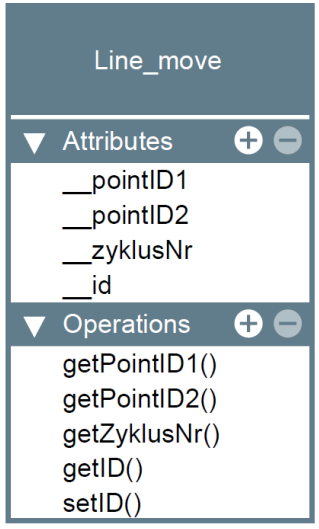
Punkt/ Point/ Point2D



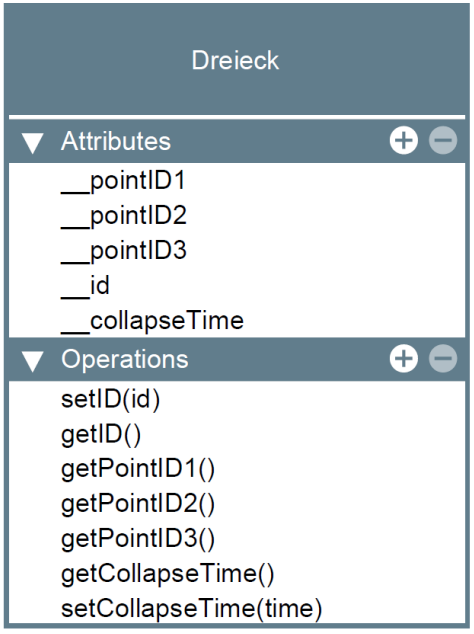
Kante/ Line_weighted/ gew. Kante/ WaveFront



Bewegungsvektor/ Line_move



Dreieck/ Triangle



Ansatz zur vereinfachten Auswertung eines Events

- Ermittle zu dem zusammengeklappten Dreieck und seiner Nachbarschaft alle relevanten Daten
- Alle Angabe in eine Matrix schreiben
- Durch Vergleichen der Matrix, evt. auch mit Wildcards/ * oder >/ <

{kante1_ID}	{kante2_ID}	{kante3_ID}
{kantentyp1}	{kantentyp2}	{kantentyp3}
{kante1_laenge}	{kante2_laenge}	{kante3_laenge}
{punktID_gegenueberliegend}	{punktID_gegenueberliegend}	{punktID_gegenueberliegend}
{dreickID_benachbart}	{dreickID_benachbart}	{dreickID_benachbart}

Kantentyp *-1 = unbekannt, 1= spoke, 2 = Wavefront-Kante*

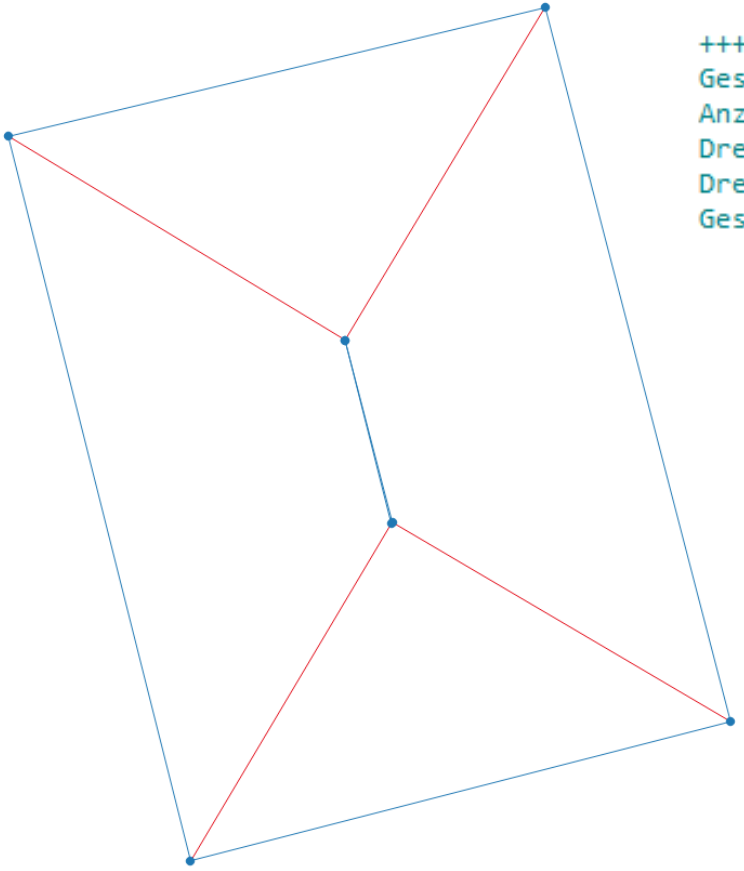
Vereinfachte Detektion möglich:

Welche Kante ist auf Länge Null geschrumpft?

Welches Event ist es?

Wie muss die Geometrie nun aktualisiert werden?

```
Analyse Matrix fuer Event:
[[  6.          -1.           5.        ]
 [  2.           1.           2.        ]
 [  0.          17.39172233  17.39172233]
 [  4.           5.           5.        ]
 [  2.          -1.          -1.        ]]
++++++++++++++++++++++++++++++++
```



Terminiert wenn:

```
++++++++++++++++++++++++++++++++
Gesamtflaeche (aller Dreiecke) berechnen fuer Zyklus Nr.: 1
Anzahl aktiver Dreiecke: 2
Dreieckflaeche fuer ID: 2 betraegt: 0.0
Dreieckflaeche fuer ID: 3 betraegt: 0.0
Gesamtflaeche: 0.0
```

Herausforderungen

Herausforderungen

Ist Python objektorientiert? → Jain

- Objekte müssen laut OOP gekapselt sein, nur ***Public*** Eigenschaften bzw. Methoden dürfen von Außen ansprechbar sein

ABER

- get-Methode getAllActivePointID() liefert Liste aller IDs aus
- Ohne eine Kopie wird die Liste unmittelbar ausgegeben
- Jede Änderung dort wird auch im Objekt selbst vollzogen
- Man vergibt aus Versehen

Schreibrechte in einer get-Methode!

Lösung: copy.deepcopy()

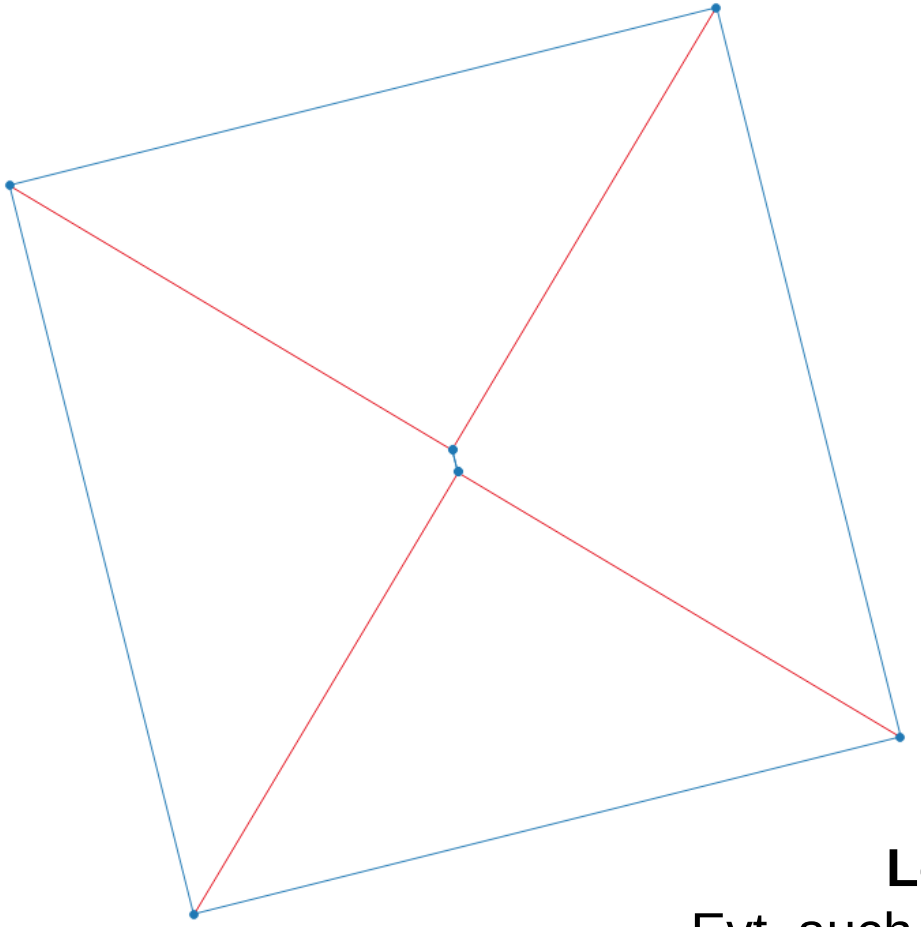
(Python 2) → Kopie erstellen

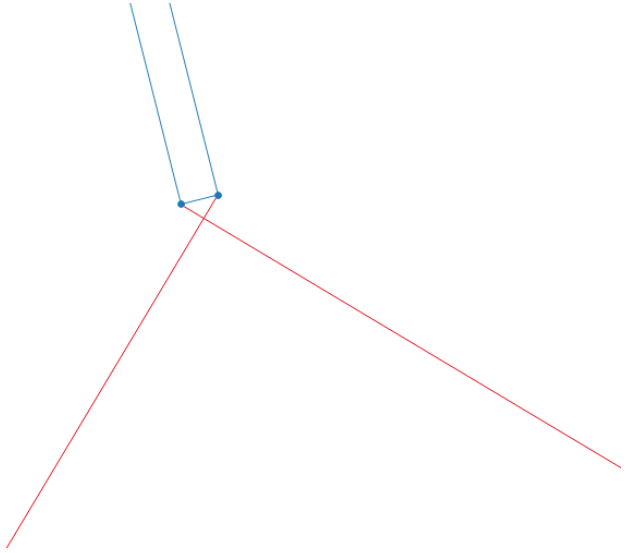
```
class Cycle_Object:
    def __init__(self):

        self.__activePoints = []

        self.__activeKanten = []

        self.__activeBewegungsvektoren = []

        self.__activeDreiecke = []

        self.__laufzeit = 0.0

    def getAllActivePointID(self):

        return copy.deepcopy(self.__activePoints)
```

Herausforderungen

Probleme durch Rundung der errechneten Collaps Time in Variable

- Erster Schrumpfungs-Schritt bis 1. Event





Neu-Punkt Berechnung
durch Rundung 0,15 mm daneben!
Lösung → Objektfang mit 1mm Toleranz
Evt. auch numpy.float128() anstelle Python float()

Fazit & Ausblick

Fazit

- Programmier-Fertigkeiten verbessert, besonders Objekt-orientiert
- Algorithmus funktioniert im Allgemeinen
- Punkt-Fang mit 1mm Toleranz deutliche Verbesserung → saubere Geometrien entstehen, keine Sliver-Polygone…
- Optimierungsbedarf und Ausbaubar
- Gute Mischung zwischen Konzeption und Implementierung ist wichtig
- Hoch und Tiefs bei Implementierung
- Laufzeit-Verhalten des Algorithmus wurde nicht untersucht
- Fehler in wissenschaftlichen Papers

Ausblick

- Gewichte der Kanten auch als (Treppen-) Funktion der Zeit t $w_{Kante}(t)$ für weitere Dachformen (Krüppelwalmdach) → **Speed-Change-Event** kommt dazu
- Weitere Dateiformate (zB Shapefile) , CRS + sphärisch?, …
- Sonderfälle / simultan auftretende Events abhandeln
- Geometrien vorher auf Parallelität/ Orthogonalität prüfen?
- Weitere Dachformen durch Gewichte-Templates abrufbar? (Satteldach: Gewichte 1-0-1-0 usw.)
- ALKIS Dachformtyp-Attribut auswerten und Gewichte festlegen
- Weitere Pre-Analysen um Gewichte zu berechnen/ optimieren, zum Beispiel dass möglichst wenig Dachflächen (Anzahl) generiert werden…

Quelle

[1] Algorithmus zur Orientierung von Polygonen: https://stackoverflow.com/questions/1165647/how-to-determine-if-a-list-of-polygon-points-are-in-clockwise-order/1165943#1165943

[2] Implementierung einer Triangulation von John Burkardt https://people.sc.fsu.edu/~jburkardt/py_src/polygon_triangulate/polygon_triangulate.html

[3] Weighted Straight Skeleton - Grundlagen und Implementierung – Masterarbeit Gerhild Grinschgi, TU Graz, Oktober 2016

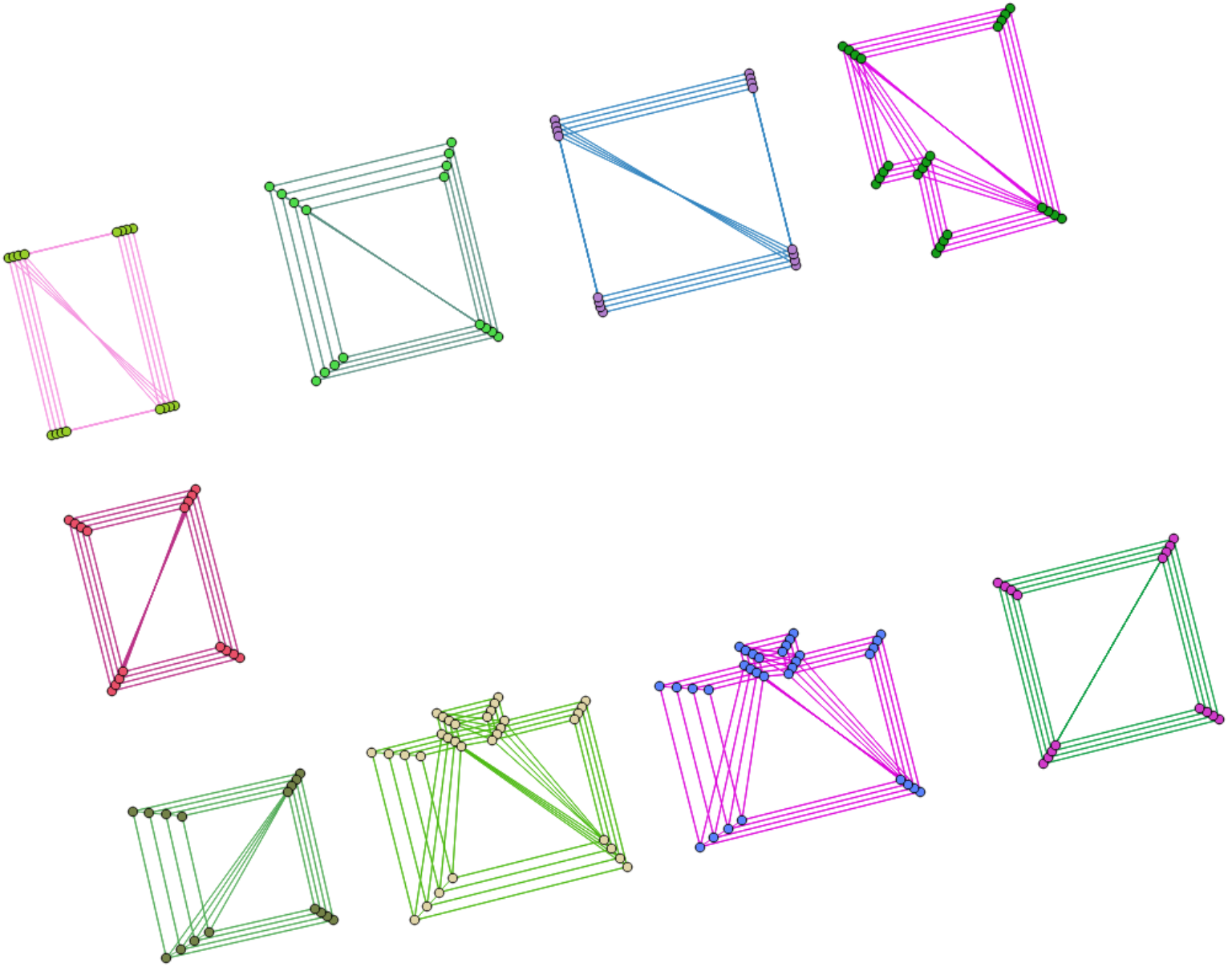
[4] Computing Straight Skeletons by Means of Kinetic Triangulations – Master´s Thesis Peter Palfrader, Universität Salzburg, September 2013

[5] Straight Skeletons with Additive and Multiplicative Weights and Their Application to the Algorithmic Generation of Roofs and Terrains – Martin Held, Peter Palfrader, Universität Salzburg, April 2016

Alle Grafiken und Screenshots ohne Quelle sind von Theodor Rieche

Vielen Dank für Ihre Aufmerksamkeit

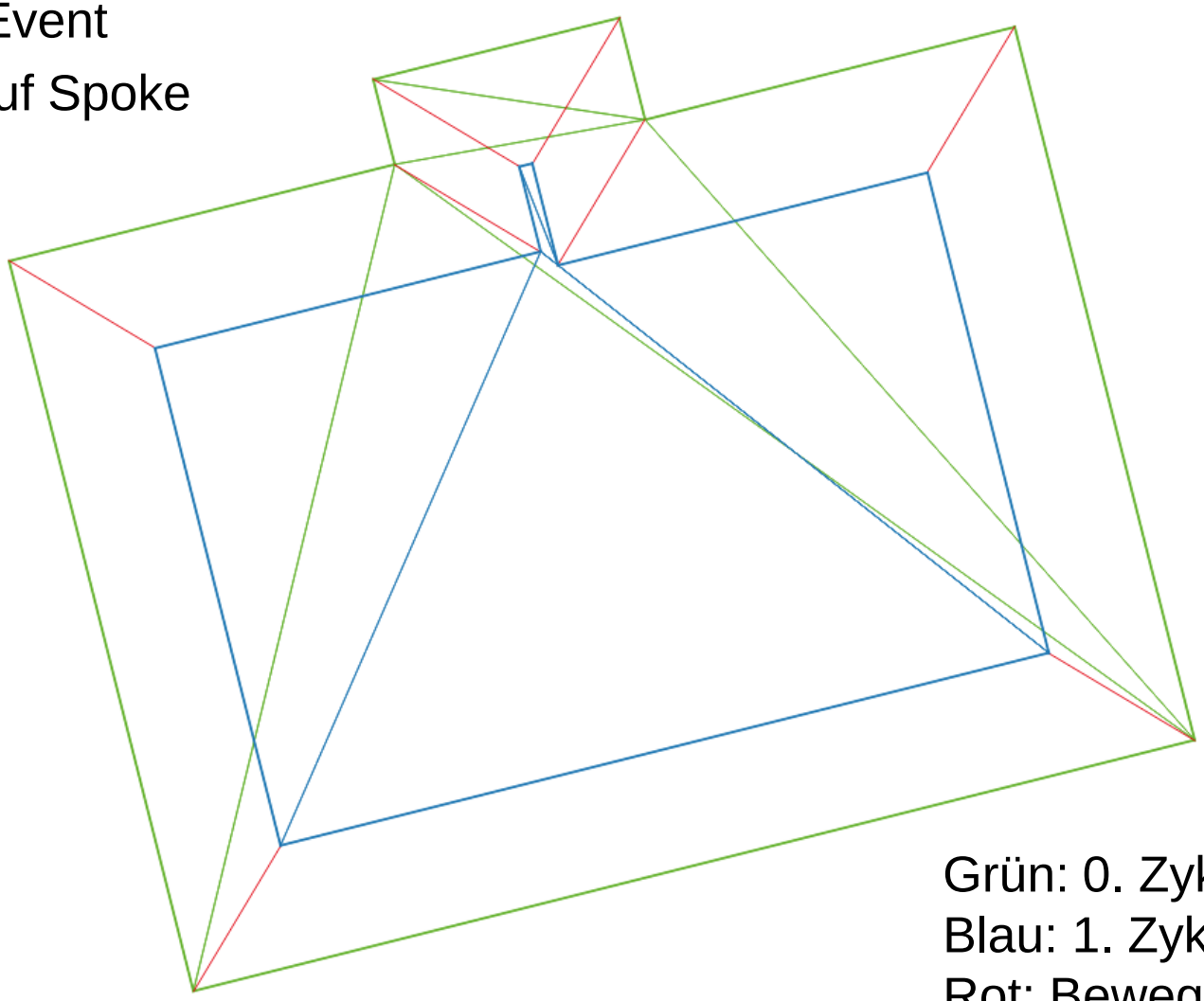
Versuche mit unterschiedlichen Gewichten



Implementierung

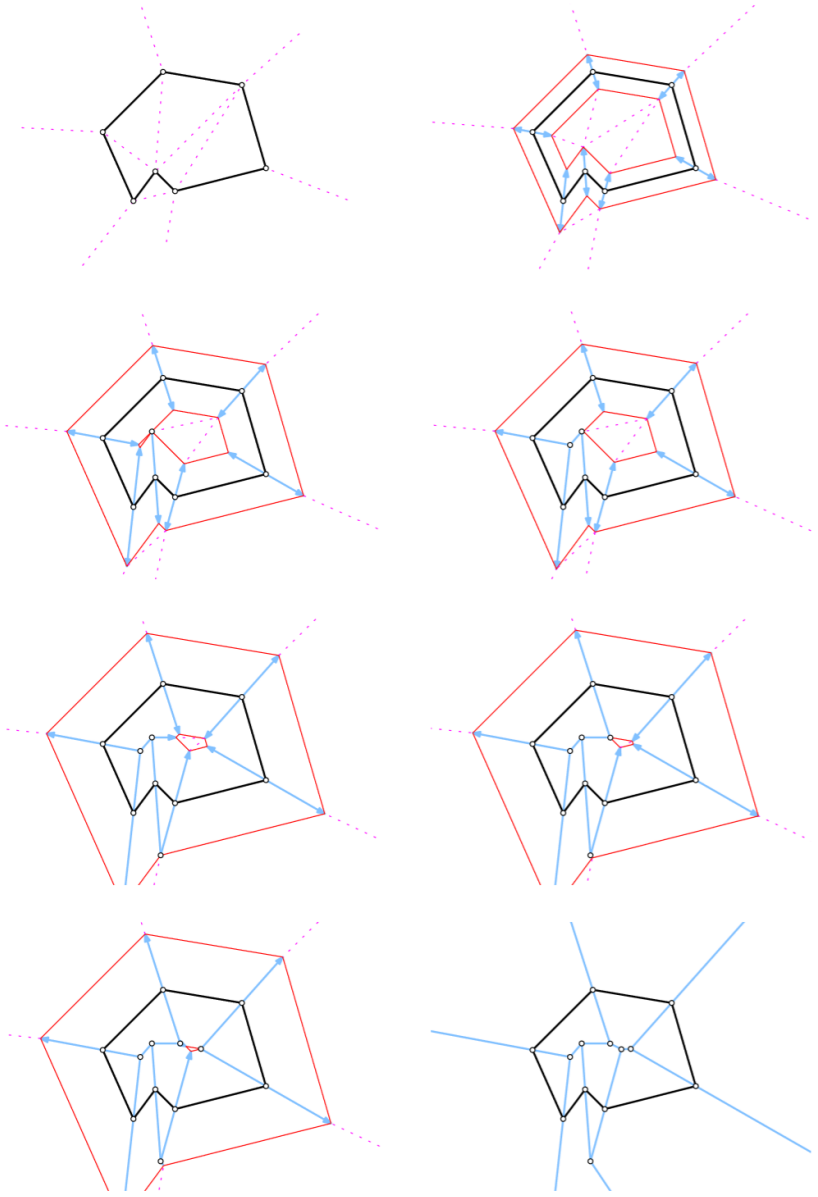
- FLIP Event

Knoten auf Spoke



Grün: 0. Zyklus
Blau: 1. Zyklus
Rot: Bewegungslinien

- Schrumpfungsprozess

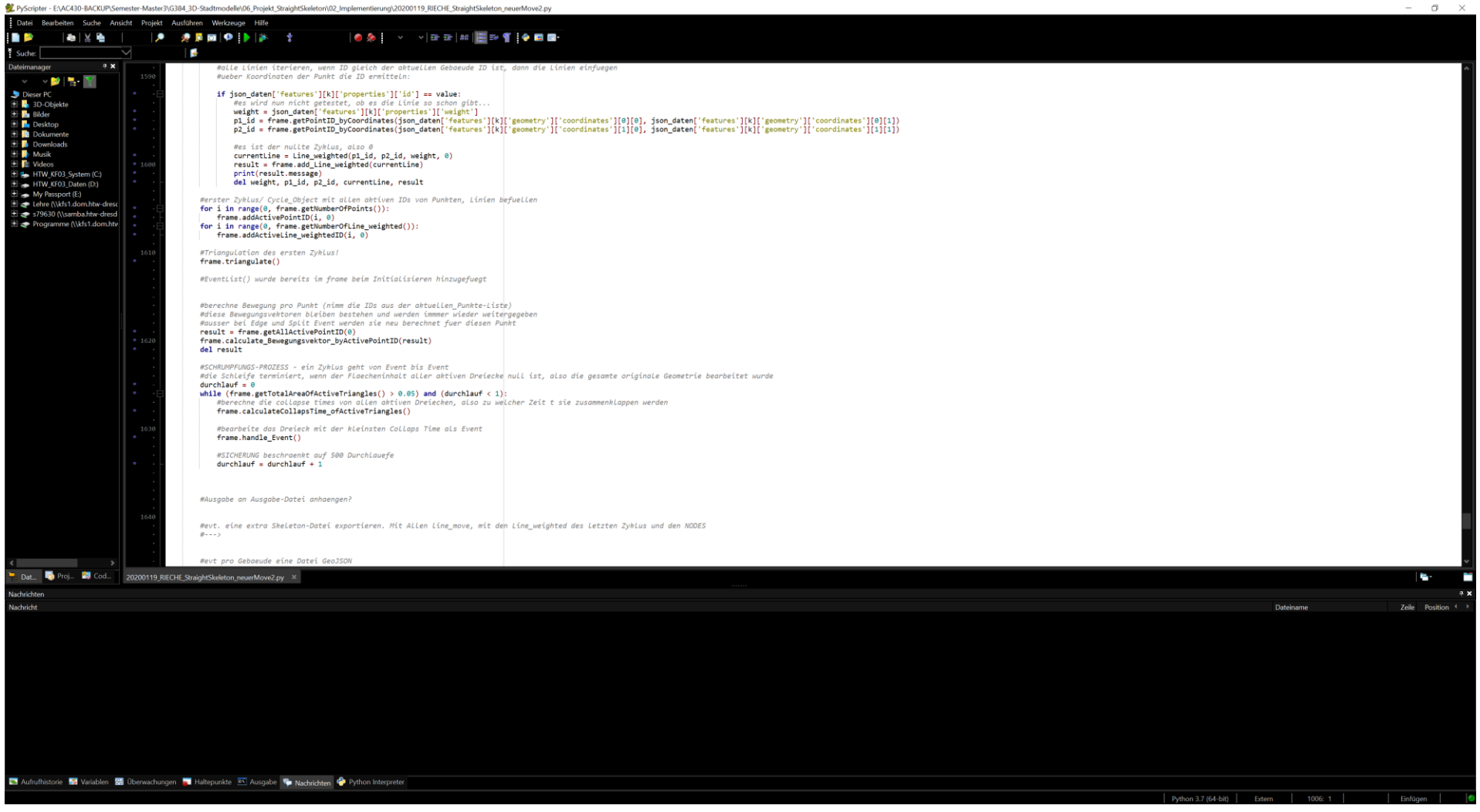


Quelle [4]

Einschränkungen

- Kartesisches metrisches CRS
- Dateiformat GeoJSON
- Keine innere Ringe/ Löcher
- Toleranzen:
  - Strecke: < 1 mm → 0
  - Fläche: < 0.05 qm → 0 (bei Summe Flächeninhalt aller akt. Dreiecke)

- Mehrere Events gleichzeitig


PyScripter - E:\AC430-BACKUP\Semester-Master3\G384_3D-Stadtmodelle\06_Projekt_StraightSkeleton\02_Implementierung\20200119_RIECHE_StraightSkeleton_neuerMove2.py
Datei Bearbeiten Suche Ansicht Projekt Ausführen Werkzeuge Hilfe
Dateimanager
Dieser PC
3D-Objekte
Bilder
Desktop
Dokumente
Downloads
Musik
Videos
HTW_KF03_System (C:)
HTW_KF03_Daten (D:)
My Passport (E:)
Lehre (\\kfs1.dom.htw-dresd
s79630 (\\samba.htw-dresd
Programme (\\kfs1.dom.htw
#alle Linien iterieren, wenn ID gleich der aktuellen Gebaeude ID ist, dann die Linien einfuegen
#ueber Koordinaten der Punkt die ID ermitteln:
if json_daten['features'][k]['properties']['id'] == value:
#es wird nun nicht getestet, ob es die Linie so schon gibt...
weight = json_daten['features'][k]['properties']['weight']
p1_id = frame.getPointID_byCoordinates(json_daten['features'][k]['geometry']['coordinates'][0][0], json_daten['features'][k]['geometry']['coordinates'][0][1])
p2_id = frame.getPointID_byCoordinates(json_daten['features'][k]['geometry']['coordinates'][1][0], json_daten['features'][k]['geometry']['coordinates'][1][1])
#es ist der nullte Zyklus, also 0
currentLine = Line_weighted(p1_id, p2_id, weight, 0)
result = frame.add_Line_weighted(currentLine)
print(result.message)
del weight, p1_id, p2_id, currentLine, result
#erster Zyklus/ Cycle_Object mit allen aktiven IDs von Punkten, Linien befuellen
for i in range(0, frame.getNumberOfPoints()):
frame.addActivePointID(i, 0)
for i in range(0, frame.getNumberOfLine_weighted()):
frame.addActiveLine_weightedID(i, 0)
#Triangulation des ersten Zyklus!
frame.triangulate()
#EventList() wurde bereits im frame beim Initialisieren hinzugefuegt
#berechne Bewegung pro Punkt (nimm die IDs aus der aktuellen_Punkte-Liste)
#diese Bewegungsvektoren bleiben bestehen und werden immmer wieder weitergegeben
#ausser bei Edge und Split Event werden sie neu berechnet fuer diesen Punkt
result = frame.getAllActivePointID(0)
frame.calculate_Bewegungsvektor_byActivePointID(result)
del result
#SCHRUMPFUNGS-PROZESS - ein Zyklus geht von Event bis Event
#die Schleife terminiert, wenn der Flaecheninhalt aller aktiven Dreiecke null ist, also die gesamte originale Geometrie bearbeitet wurde
durchlauf = 0
while (frame.getTotalAreaOfActiveTriangles() > 0.05) and (durchlauf < 1):
#berechne die collapse times von allen aktiven Dreiecken, also zu welcher Zeit t sie zusammenklappen werden
frame.calculateCollapsTime_ofActiveTriangles()
#bearbeite das Dreieck mit der kleinsten CoLlaps Time als Event
frame.handle_Event()
#SICHERUNG beschraenkt auf 500 Durchlauefe
durchlauf = durchlauf + 1
#Ausgabe an Ausgabe-Datei anhaengen?
#evt. eine extra Skeleton-Datei exportieren. Mit ALLen Line_move, mit den Line_weighted des letzten Zyklus und den NODES
#--->
#evt pro Gebaeude eine Datei GeoJSON
20200119_RIECHE_StraightSkeleton_neuerMove2.py
Nachrichten
Nachricht
Dateiname
Zeile
Position
Aufrufhistorie
Variablen
Überwachungen
Haltepunkte
Ausgabe
Nachrichten
Python Interpreter
Python 3.7 (64-bit)
Extern
1006: 1
Einfügen

```
def calculateCollapsTime_ofActiveTriangles(self, zyklusnr=-1):
    #wenn die Zyklusnr. fuer den die Collapse Times berechnet werden soll NICHT angegeben wurde, nimm die zuletzt hinzugefuegte
    if zyklusnr==-1:
        zyklusnr=len(self.__ListeCycleObjects)-1

    print('++++++++++++++++++++++++++++++++')
    print('berechne Collapse Times fuer Zyklus Nr.: ' + str(zyklusnr))

    #fuer alle aktiven Dreiecke des gegebenen Zyklus: berechne jeweils die CollapsTime t, also wann das Dreieck zusammenklappen wird/ Area = gleich Null/ Kollinearitaet der Punkte
    activeTriangles = self.getAllActiveDreiecke(zyklusnr)
    #gehe die aktiven Dreiecke durch
    for s in range (0, len(activeTriangles)):
        dreieck_aktuell = self.getDreieck_byID(activeTriangles[s])
        #hole dir die Punkte
        punkt_o1 = self.getPoint_byID(dreieck_aktuell.getPointID1())
        punkt_o2 = self.getPoint_byID(dreieck_aktuell.getPointID2())
        punkt_o3 = self.getPoint_byID(dreieck_aktuell.getPointID3())

        o1_x = punkt_o1.getX()
        o1_y = punkt_o1.getY()
        o2_x = punkt_o2.getX()
        o2_y = punkt_o2.getY()
        o3_x = punkt_o3.getX()
        o3_y = punkt_o3.getY()

        #sektiere jeweils pro Punkt die passenden Bewegungsvektoren aus dem aktuellen Zyklus
        s1_x, s1_y = punkt_o1.getBewegungsvektor()
        s2_x, s2_y = punkt_o2.getBewegungsvektor()
        s3_x, s3_y = punkt_o3.getBewegungsvektor()

        #berechne die Zeit t
        #es kann eine, zwei oder gar keine Loesungen geben

        #umsetzen der P-Q-Formel fuer Quadratische Formeln
#         faktor_Grad2 = 1.0 * (s1_y * s2_x + s1_x * s2_y + s1_y * s3_x - s2_y * s3_x - s1_x * s3_y + s2_x * s3_y)
#         faktor_Grad1 = 1.0 * (o2_y * s1_x - o3_y * s1_x - o2_x * s1_y + o3_x * s1_y - o1_y * s2_x + o3_y * s2_x + o1_x * s2_y - o3_x * s2_y + o1_y * s3_x - o2_y * s3_x - o1_x * s3_y + o2_x * s3_y)
#         faktor_Grad0 = 1.0 * (-1.0 * o1_y * o2_x + o1_x * o2_y + o1_y * o3_x - o2_y * o3_x - o1_x * o3_y + o2_x * o3_y)

        faktor_Grad2 = 1.0 * (s2_x*s3_y + s1_x*s2_y + s1_y*s3_x - s2_x*s1_y - s3_x*s2_y - s3_y*s1_x)
        faktor_Grad1 = 1.0 * (s2_x*o3_y + s3_y*o2_x + s1_x*o2_y + s2_y*o1_x + s1_y*o3_x + s3_x*o1_y - s2_x*o1_y - s1_y*o2_x - s3_x*o2_y - s2_y*o3_x - s3_y*o1_x - s1_x*o3_y)
        faktor_Grad0 = 1.0 * (o2_x*o3_y + o1_x*o2_y + o1_y*o3_x - o2_x*o1_y - o3_x*o2_y - o3_y*o1_x)

        #in die Normalform bringen (bei x^2 kein Faktor mehr stehen)
        p = faktor_Grad1 / faktor_Grad2
        q = faktor_Grad0 / faktor_Grad2
        radikant = (p * 0.5) * (p * 0.5) - q

        #Loesen und Anzahl Loesungen verarbeiten!
        zeit = 0.0
        #Anzahl Loesungen
        if radikant < 0:
            zeit = -1.0
            #keine Loesungen
        if radikant == 0:
            #eine Loesung
            zeit = -1.0 * p * 0.5
        if radikant > 0:
            #bei zwei Loesungen
            t1 = -1.0 * p * 0.5 + math.sqrt(radikant)
            t2 = -1.0 * p * 0.5 - math.sqrt(radikant)
            if (min(t1, t2) < 0) and (max(t1, t2) >= 0):
                zeit = max(t1, t2)
            if (t1 >= 0) and (t2 >= 0):
                zeit = min(t1, t2)
            if (t1 < 0) and (t2 < 0):
                zeit = -1.0
            del t1, t2
        print('fuer Dreieck ID: ' + str(dreieck_aktuell.getID()) + ' Collapes Time t = ' + str(zeit))
        #es reicht die kuerzere Zeit bei zwei Loesungen zu nehmen. Wenn es nicht zusammenklappt, dann -1.0 also unendlich!

        #setze die Zeit t beim Dreieck!
        self.__triangulation[activeTriangles[s]].setCollapseTime(zeit)

        del zeit, dreieck_aktuell, punkt_o1, punkt_o2, punkt_o3, o1_x, o1_y, o2_x, o2_y, o3_x, o3_y
        del s1_x, s1_y, s2_x, s2_y, s3_x, s3_y, faktor_Grad0, faktor_Grad1, faktor_Grad2, p, q, radikant

    return
```

```
def calculate_Bewegungsvektor_byActivePointID(self, pointID_list, zyklusnr=-1):

    #hier wird fuer alle gegebenen Punkte aus der pointID_list der entsprechende Bewegungsvektor berechnet
    #der Bewegungsvektor steht fuer die Strecke, die dieser aktive Punkt in einer Zeiteinheit t=1 zurueecklegen wuerde...

    #dieser wird dann mit setBewegungsvektor diesem Punkt uebergeben...
    #aufgrund der Liste kann auch nur ein einzelner Punkt unter Umstaenden einen neuen Vektor bekommen (nach einem entsprechenden Event)

    if zyklusnr==-1:
        zyklusnr=len(self.__ListeCycleObjects)-1

    print('++++++++++++++++++++++++++++++++')
    print('Bewegungsvektoren berechnen fuer Zyklus Nr.: ' + str(zyklusnr))
    print('fuer die Punkte ID List: ' + str(pointID_list))

    #hole die gegen den Uhrzeigersinn orientierte sortie Punkt-Liste aller aktiven Punkte im angefragten Zyklus
    #da auch mehrere Polygone zurueckgegeben werden koennen, muessen diese mit einer for-schleife durchlaufen werden...
    result3 = self.getOrientatedActivePointIDList(zyklusnr)
    orientatedPoints = result3.valueList
    print('orientierte Punkte: ' + str(orientatedPoints))
    del result3
    print('Anzahl an Linienzuegen/ Polygonen: ' + str(len(orientatedPoints)))

    #gehe alle Linienzuege durch /es kann nach einem SPlit-Event mehr als einen geben!
    for v in range(0, len(orientatedPoints)):

        #Loop over active points
        for i in range(0, len(orientatedPoints[v])):
            #pruefe, ob dieser Punkt einer der in der PointIDListe angegeben Punkte ist!

            if pointID_list.count(orientatedPoints[v][i]) > 0:
                #der Punkt soll einen neuen Bewegungsvektor bekommen!

                print('berechne Bew-Vektor fuer Punkt ID: ' + str(orientatedPoints[v][i]))

                #bestimme die 3 Punkte - ermittle Vorgaenger und Nachfolger
                id1 = 0
                id2 = 0
                if i == 0:
                    id1 = len(orientatedPoints[v])-1
                    id2 = i+1
                if i == len(orientatedPoints[v])-1:
                    id1 = i-1
                    id2 = 0
                if (i > 0) and (i < len(orientatedPoints[v])-1):
                    id1 = i-1
                    id2 = i+1

                punkt_aktuell = self.getPoint_byID(orientatedPoints[v][i])
                punkt_vorgaenger = self.getPoint_byID(orientatedPoints[v][id1])
                punkt_nachfolger = self.getPoint_byID(orientatedPoints[v][id2])
                del id1, id2

                #Linie 1/ Kante1 liegt zwischen aktuellem Punkt und dem Vorgaenger
                kante1 = self.getLine_weighted_byID(self.getLine_weightedID_byTwoGivenPointID(punkt_vorgaenger.getID(), punkt_aktuell.getID()))
                #Linie 2/ Kante2 liegt zwischen aktuellem Punkt und dem Nachfolger
                kante2 = self.getLine_weighted_byID(self.getLine_weightedID_byTwoGivenPointID(punkt_aktuell.getID(), punkt_nachfolger.getID()))

                #x,y Differenzen der Kanten berechnen fuer Linie l1 und Linie l2
                l1_x = punkt_aktuell.getX() - punkt_vorgaenger.getX()
                l1_y = punkt_aktuell.getY() - punkt_vorgaenger.getY()
                l2_x = punkt_nachfolger.getX() - punkt_aktuell.getX()
                l2_y = punkt_nachfolger.getY() - punkt_aktuell.getY()

                #Normalen-Vektoren berechnen fuer die beiden Kanten (nach links in Kantenrichtung zeigend, da gegen den UZS orientiert = nach Innen zeigend!)
                n1_x = -1 * l1_y
                n1_y = l1_x
                n2_x = -1 * l2_y
                n2_y = l2_x

                #normalen-Vektor normieren und mit Kantengewichten als Skalar multiplizieren
                betrag_n1 = math.sqrt(n1_x**2 + n1_y**2)
                betrag_n2 = math.sqrt(n2_x**2 + n2_y**2)
```

- Zweiter Teil: def **calculate_Bewegungsvektor_byActivePointID**(self, pointID_list, zyklusnr=-1):

```
            normal1_x = 0.0
            normal1_y = 0.0
            if betrag_n1 != 0:
                normal1_x = kante1.getWeight() * (1 / betrag_n1) * n1_x
                normal1_y = kante1.getWeight() * (1 / betrag_n1) * n1_y
            else:
                print('ACHTUNG: division by zero, Bewegungsvektor/ Normal1 fuer Point ID: ' + str(orientatedPoints[v][i]))

            normal2_x = 0.0
            normal2_y = 0.0
            if betrag_n2 != 0:
                normal2_x = kante2.getWeight() * (1 / betrag_n2) * n2_x
                normal2_y = kante2.getWeight() * (1 / betrag_n2) * n2_y
            else:
                print('ACHTUNG: division by zero, Bewegungsvektor/ Normal2 fuer Point ID: ' + str(orientatedPoints[v][i]))

            #die beiden neuen Geraden aufstellen, gleichsetzen und den gemeinsamen Schnittpunkt finden
            #SCHNITTPUNKT zweier Geraden

             theta = (punkt_vorgaenger.getX() + normal1_x - punkt_nachfolger.getX() - normal2_x) / (punkt_nachfolger.getX() + punkt_vorgaenger.getX() - 2.0 * punkt_aktuell.getX())

            #SUBSTITUTION mit a,b,c,d,e,f durchfuehren
            a = punkt_nachfolger.getX() + normal2_x - punkt_vorgaenger.getX() - normal1_x
            b = l2_x
            c = l1_y
            d = punkt_nachfolger.getY() + normal2_y - punkt_vorgaenger.getY() - normal1_y
            e = l2_y
            f = l1_x
            #damit Beta ausrechnen
            beta = (d * f - a * c) / (b * c - e * f)

            #Schnittpunkt anhand von dem Beta ausrechnen (wichtig: in gerade2 also kante 2 einsetzen!)
            schnitt_x = punkt_nachfolger.getX() + normal2_x + beta * l2_x
            schnitt_y = punkt_nachfolger.getY() + normal2_y + beta * l2_y

            #bewegungsvektor berechnen, zwischen dem aktuellen Punkt und dem neu berechneten Schnittpunkt
            v_x = schnitt_x - punkt_aktuell.getX()
            v_y = schnitt_y - punkt_aktuell.getY()

            #diesen Vektor nun mit set-Methode dem entsprechenden Punkt mitgeben!
            self.__storagePoint2D[orientatedPoints[v][i]].setBewegungsvektor(v_x, v_y)

            del kante1, kante2, l1_x, l1_y, l2_x, l2_y, n1_x, n1_y, n2_x, n2_y, v_x, v_y
            del normal1_x, normal1_y, normal2_x, normal2_y
            del punkt_aktuell, punkt_nachfolger, punkt_vorgaenger
            del beta, a, b, c, d, e, f

    return
```