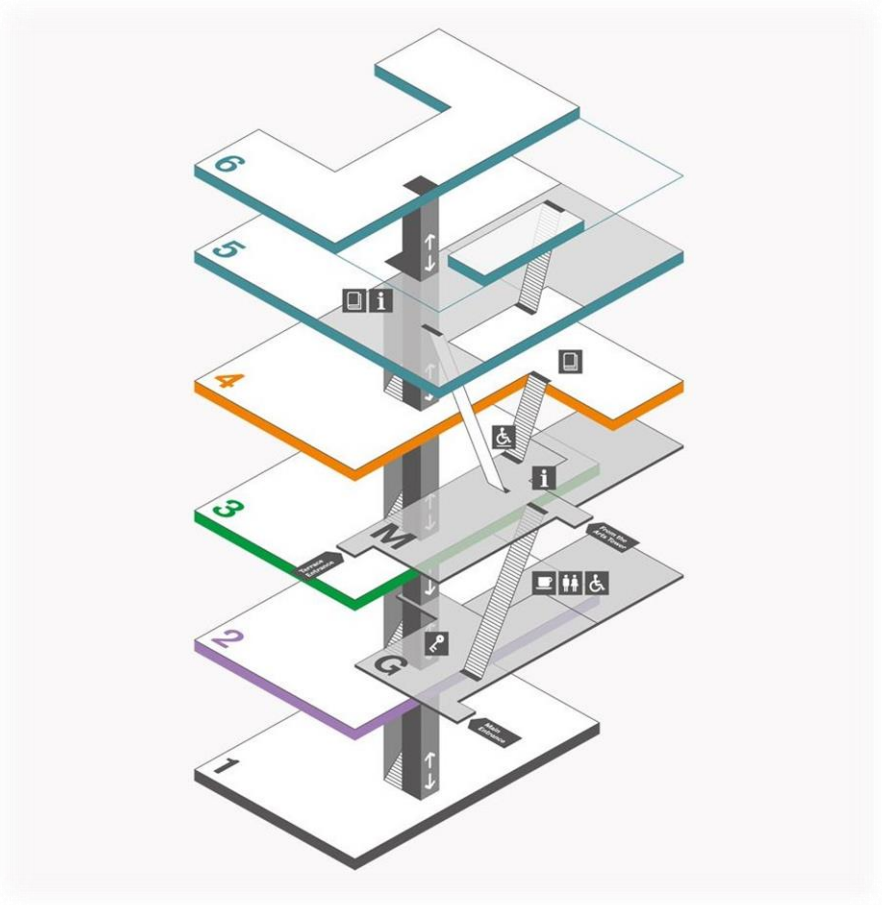
Genereren IMDF data

Afstudeerverslag



Naam	Diederik IJspeerd
Studentnummer	452705
Hogeschool	Saxion
Opleiding	HBO-ICT
Profiel	Software Engineering
Stagebedrijf	Baseflow
Stageperiode	08-02-2021 t/m 02-07-2021
Bedrijfsbegeleider	Gerwin Rupert
Stagebegeleider	Erik van der Arend
Datum	13-06-2021

Inhoudsopgave

Samenvatting

Baseflow maakt gebruik van indoor kaarten in IMDF formaat voor applicaties die zij voor hun klanten bouwen. Deze kaarten worden o.a. gebruikt voor indoor navigatie. De kaarten worden in IMDF formaat op aanvraag door een derde partij aangeleverd. De tijd tussen het aanvragen en ontvangen van een nieuwe kaart duurt lang: vaak meer dan drie weken. Dit is nadelig voor zowel Baseflow als de klanten van Baseflow die de kaarten in hun applicaties willen gebruiken.

Het doel van deze afstudeeropdracht is om de mogelijkheden van het genereren van IMDF te onderzoeken en hier een proof of concept (PoC) voor te bouwen. Het PoC moet daarbij schaalbaar zijn, zodat deze in de toekomst verder uitgebouwd kan worden en Baseflow idealiter geen derde partij meer hoeft in te huren voor het laten maken van kaarten in IMDF formaat.

Uit het onderzoek is naar boven gekomen dat kaarten in CAD en PDF formaat worden gebruikt voor het genereren van IMDF. Beide formaten zijn (indirect) om te zetten naar SVG. De SVG bestanden kunnen gebruikt worden om geometrische data uit te halen. Deze data wordt d.m.v. een set aan algoritmes eerst geoptimaliseerd en vervolgens omgezet naar lengte- en breedtegraad coördinaten waarna het als IMDF data te downloaden is. IMDF data kan dus deels geautomatiseerd gegenereerd worden. Er is echter altijd nog extra handmatig werk nodig. De output van de algoritmes is namelijk nooit perfect. Dit extra handmatige werk bestaat uit het weghalen/intekenen van lijnen en zal in de meeste gevallen aanzienlijk minder tijd kosten dan dat het normaal kost om IMDF data door de derde partij aangeleverd te krijgen.

Het gerealiseerde PoC bestaat uit een Vue.js frontend en een NestJS backend. De frontend is verantwoordelijk voor het renderen van de SVG en het mogelijk maken van gebruikersinteractie. De backend is verantwoordelijk voor het uitvoeren van alle (optimalisatie-)algoritmes en doet dus al het zware werk.

Het PoC kan in de toekomst uitgebreid worden door de algoritmes uit te breiden zodat ze met meer verschillende geometrische types om kunnen gaan. Daarnaast kan functionaliteit toegevoegd worden zodat meerdere kaarten tegelijk bewerkt en omgezet kunnen worden. Ook kunnen nog features worden ingebouwd die, naast de geautomatiseerde bewerkingen, ook handmatige bewerkingen van de kaart mogelijk maken. Voor een eventueel vervolgonderzoek is het interessant om te kijken wat er mogelijk is m.b.t. het optimaliseren van geometrie door het gebruik van machine learning/AI.

1 Inleiding

Dit document is het afstudeerverslag, onderdeel van het afstudeerproject van Diederik IJspeerd uitgevoerd bij het bedrijf Baseflow te Enschede. Dit afstudeerproject dient als afsluitende fase van de opleiding HBO-ICT met als profiel software engineering en is uitgevoerd van 8 februari t/m 2 juli 2021.

Baseflow maakt gebruik van indoor kaarten in IMDF formaat voor applicaties die zij voor hun klanten bouwen. Zo hebben ze bijvoorbeeld de applicatie “Digitaal Service Platform” ontwikkeld voor Asito. Deze applicatie maakt gebruik van een zulke kaarten om o.a. indoor navigatie mogelijk te maken voor de medewerkers van Asito op Schiphol. Baseflow werkt samen met een derde partij die deze kaarten aanleveren. De tijd tussen het aanvragen en ontvangen van een nieuwe kaart duur echter lang: vaak meer dan drie weken. Dit is nadelig voor zowel Baseflow als de klanten van Baseflow die de kaarten in hun applicaties willen gebruiken.

De probleemstelling luidt dan ook als volgt:
Het verkrijgen van indoor kaarten in IMDF formaat voor het gebruik in applicaties duurt lang, waardoor klanten van Baseflow na een aanvraag lang moeten wachten en is het ook niet mogelijk om snel een prototype te bouwen voor potentiële nieuwe klanten.

Dit afstudeerproject is beperkt tot het onderzoeken naar en het genereren van indoor kaarten in IMDF formaat. Outdoor kaarten komen dus niet aan de orde. Het doel van het onderzoek is daarbij om te achterhalen wat de meest efficiënte manier is om aan de benodigde IMDF data te komen. Daarom is gekeken naar de mogelijkheden om IMDF data te genereren uit verschillende bronnen. Hoewel de gegenereerde kaarten o.a. gebruikt zullen worden voor positionering en het aanpassen van onderdelen van de kaarten (zoals markers, waypoints etc.), is dit afstudeerproject beperkt tot het genereren van de kaarten.

Met dit afstudeerproject wordt de hoofdvraag: "Hoe kan Baseflow d.m.v. automatisering op een zo efficiënt mogelijke manier IMDF data genereren voor het renderen van indoor kaarten voor gebruik in haar applicaties?" beantwoord. Deelvragen die hierbij behandeld worden zijn:

1. Wat is IMDF, hoe werkt het en waar moet het aan voldoen?
2. Wat is het GeoJSON data formaat waar IMDF gebruik van maakt?
3. Welke features van het IMDF formaat zijn van belang voor het renderen van indoor kaarten?

4. Welke bestandsformaten van digitale kaarten worden momenteel gebruikt voor het laten maken van kaarten in IMDF formaat?
5. Welke mogelijkheden zijn er om geometrische informatie uit digitale kaarten om te zetten naar data?
6. Hoe kan de verkregen geometrische data omgezet worden naar data in het IMDF formaat?

De eerste twee deelvragen zijn beschrijvende deelvragen en worden uitgewerkt in hoofdstuk 2. De overige deelvragen worden besproken in hoofdstuk 4. Hoofdstuk 3 bevat een omschrijving over hoe het onderzoek en de implementatie is aangepakt. Hoofdstuk 5 en 6 beschrijven de ontwerp- en implementatiefase van het project en de keuzes die daar zijn gemaakt. Als laatst bevatten hoofdstuk 8, 9 en 9 respectievelijk de conclusie, discussie, aanbevelingen en reflectie.

2 Theoretisch kader

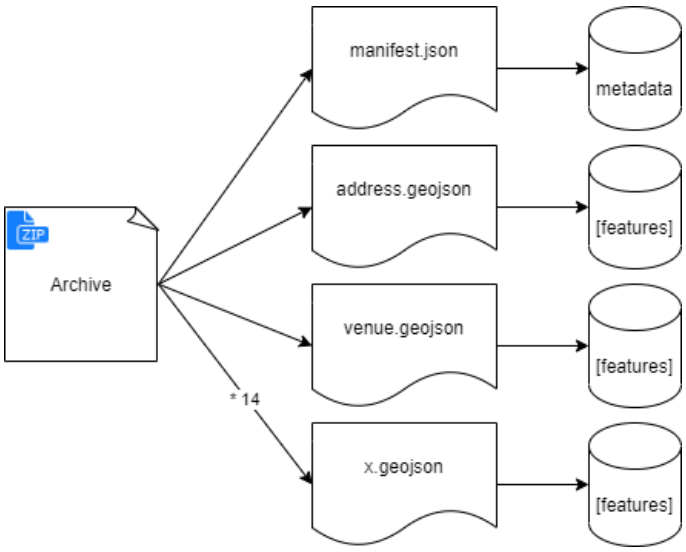
In dit hoofdstuk wordt een korte toelichting gegeven op de IMDF en GeoJSON specificaties. Dit beantwoordt tevens deelvraag 1 en 2. Voor een uitgebreide uitwerking van deze deelvragen, zie hoofdstuk 1.1 en 1.2 van het document Onderzoeksrapport (IJspeerd, 2021).

Het IMDF data formaat

IMDF staat voor Indoor Mapping Data Format en is een specifiek type gestandaardiseerd data model dat gebruikt wordt om indoor kaarten mee te renderen. Het formaat is ontwikkeld door Apple en kan gebruikt worden voor indoor positiebepaling en navigatie. IMDF kan op ieder apparaat, besturingssysteem en webbrowser gerenderd worden (Apple Inc., 2020).

In de basis is IMDF een verzameling GeoJSON bestanden. GeoJSON is op zijn beurt weer een data formaat waar op zijn minst geografische informatie in gedefinieerd staat. Daarnaast kan een GeoJSON bestand nog bestaan uit extra optionele, niet geografische data. Deze verzameling GeoJSON bestanden vormen samen met een manifest, dat metadata bevat, gezipt het IMDF formaat. Zo'n zip bestand wordt ook wel een archief genoemd.

Elk bestand in een archief bestaat uit een verzameling 'IMDF feature objects' (hierna: 'features') van hetzelfde type. Er zijn 16 verschillende typen features en elke feature beschrijft een fysiek of denkbeeldig element van een indoor kaart. De features Address en Venue zijn vereiste onderdelen van een archief. Alle andere features zijn optioneel. Figuur 1 laat een visuele weergave zien van hoe het IMDF formaat is opgebouwd.



Figuur 1 - Opbouw IMDF

Het GeoJSON data formaat

GeoJSON is een gestandaardiseerd data formaat gebaseerd op JSON en is ervoor gemaakt om geografische data in te omschrijven met daarnaast de mogelijk om niet-geografische data ('properties') er bij in op te nemen (IETF, 2016). De geografische data wordt omschreven door verschillende geometrische types: *Point*, *LineString*, *Polygon*, *MultiPoint*, *MultiLineString* en *MultiPolygon*.

GeoJSON gebruikt bij het definiëren van de coördinaten van deze geometrische types "decimal degrees" als eenheid. Decimal degrees is een specifieke notatie van het geografisch coördinaten systeem (GCS), waarmee verschillende posities op de aarde kunnen worden beschreven door de lengtegraad en breedtegraad uit te drukken in fracties van een graad. Elke lijst met coördinaten heeft op zijn minst de lengtegraad en breedtegraad met eventueel nog een hoogte als derde getal. Figuur 2 laat een voorbeeld zien van de inhoud van een GeoJSON bestand met enkel een *Point*.

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [6.94598, 52.21656]
  },
  "properties": {
    "name": "Baseflow Enschede"
  }
}
```

Figuur 2 - Een GeoJSON feature

3 Methodiek en proces

Onderzoeksfase
Het onderzoek in dit afstudeerproject betreft praktisch/toegepast, kwalitatief onderzoek. Voor het onderzoek zijn er dan ook geen hypotheses opgesteld. Het doel van het onderzoek is immers om een oplossing te vinden voor het probleem waar Baseflow mee zit.

Tijdens de onderzoeksfase is gebruik gemaakt van meerdere onderzoeksstrategieën uit het triangulatie framework (Turnhout et al., 2013). Bieb, Werkplaats en Lab zijn de drie voornaamste strategieën die zijn gebruikt. Bieb is voornamelijk gebruikt om d.m.v. literatuurstudie het onderzoek op te starten om kennis op te doen over IMDF en GeoJSON. Daarnaast is literatuurstudie gebruikt om kennis op te doen over verschillende (bronnen van) architectuurtekeningen en hun relatie tot indoor IMDF kaarten. Ook is gaandeweg gekeken naar eventuele (deel)oplossingen die al bestaan en is er gekeken naar de data die de derde partij in het verleden heeft opgeleverd.

De Werkplaats is de strategie die bij dit afstudeerproject het meest is gebruikt. Van werkplaats zijn de methodes prototyping en brainstorm het meest gebruikt. Deze methodes zijn niet alleen gebruikt bij het bouwen van het PoC in de implementatiefase, maar vooral ook tijdens de onderzoeksfase.

Lab is de strategie die gebruikt is bij het bouwen en uitproberen van mogelijke deeloplossingen waarbij continu getoetst is of het wel degelijk een passende deeloplossing is voor het gewenste PoC. Dit is gedaan door mijzelf tijdens het ontwikkelen ervan, maar ook bij de opdrachtgever, Gerwin Rupert en het team d.m.v. demo's en scrum meetings. Daarnaast zijn er unit tests geschreven.

Ontwerp- en implementatiefase
Na het onderzoek is in de ontwerp- en implementatiefase gebruik gemaakt van een agile werkwijze met daarbij (beperkte) scrummethodieken. M.a.w. er is op een iteratieve manier gewerkt, maar scrum is niet op de volledige manier toegepast, omdat dit project individueel is uitgevoerd en niet in teamverband. Daarom zijn er geen retrospectives en refinements geweest. Wel is er op iteratieve wijze in tweewekelijkse sprints gewerkt met aan iedere sprint voorafgaand een scrum meeting met de opdrachtgever. Deze scrum meetings waren een demo/sprint review en sprintplanning in één. Tijdens deze meetings is steeds de voortgang besproken, feedback gegeven en vooral ook de prioriteiten voor de opvolgende sprint besproken.

Algemeen
Ondanks dat de uitvoering van het afstudeerproject individueel was, ben ik wel onderdeel geweest van een van de teams om betrokken te blijven bij het bedrijf (en andersom) en het product waar het PoC voornamelijk voor is gebouwd. Met dit team zijn ook dagelijkse stand-ups gehouden en af en toe is er een demo gegeven van het prototype en PoC.

Keuzemoment
Ergens in de loop van het project is er een belangrijk keuzemoment geweest m.b.t. het verdere verloop van het project. Dit was een moment waarop besloten moest worden of er meer tijd in het optimaliseren van de tekeningen gestopt zou worden (zie hoofdstuk 4.3.4) of dat het belangrijker was om vanaf dat moment ervoor te zorgen dat de cirkel rond komt. M.a.w. dat het mogelijk is om, ongeacht de staat van de tekening, de tekening om te zetten naar IMDF en dit te downloaden. In samenspraak met de opdrachtgever is er toen voor gekozen om het optimaliseren te laten liggen en verder te gaan met het genereren en downloaden van IMDF. Hiermee is de tekening die de gegenereerde IMDF data representeert dus niet perfect, maar is het in ieder geval mogelijk om zowel de tekening zover mogelijk geautomatiseerd te optimaliseren als om te zetten naar IMDF.

4 Onderzoeksresultaten

In dit hoofdstuk wordt een samenvatting van de onderzoeksresultaten gepresenteerd, waarbij elke paragraaf een deelvraag beschrijft. De resultaten van de eerste twee deelvragen zijn beschreven in het hoofdstuk Theoretisch Kader. Dit hoofdstuk bevat de resultaten van de andere 4 deelvragen. Een uitvoeriger uitwerking van alle onderzoeksresultaten is terug te vinden in het document Onderzoeksrapport (IJspeerd, 2021).

In dit hoofdstuk wordt vaak gerefereerd naar 'keys', 'values' en 'key-value pairs'. Hiermee worden de keys, values en key-value pairs van JSON objecten bedoeld. Om de leesbaarheid te verhogen zijn alle beschreven keys en values in dit hoofdstuk schuingedrukt en beginnen de namen van alle IMDF features steeds met hoofdletters. Ook verschillende geometrische types zijn schuingedrukt.

4.1. Welke features van het IMDF formaat zijn van belang voor het renderen van indoor kaarten?

4.1.1. Huidige situatie

Ruwe IMDF data
Wat opvalt aan de IMDF data zoals deze door de derde partij aangeleverd wordt, is dat bij geen van de archieven alle mogelijke IMDF features gebruikt worden. Er worden meestal zo'n 6 à 7 features gebruikt, met op zijn minst de features Building, Level, Unit, Fixture en Opening. Daarnaast heeft elk archief een Points.geojson bestand, wat geen officiele IMDF features bevat. Ook valt op dat de data tussen verschillende archieven niet altijd even consistent is. Daarnaast komen sommige aspecten van de data niet overeen met de officiële IMDF specificatie.

Zo ontbreekt in de aangeleverde archieven in alle gevallen de vereiste Address feature, en ontbreken bij alle individuele features de vereiste keys *id* en *feature_type*. Bij features die *(Multi)Polygon* als geometrisch type hebben, valt op dat in veel gevallen de veelhoek meer coördinaten heeft dan nodig is. D.w.z. er zijn vaak punten waarvan de hoek tussen de aangrenzende lijnen 180 graden is en de aangrenzende lijnen samen dus een rechte lijn vormen. Dit maakt dat er overbodige data is en daarmee de grootte van de archieven groter is dan nodig. Ook zijn de features Anchor en Footprint, die vereist zijn bij gebruik van de Occupant en Building features, niet aanwezig.

Wat betreft geometrische types worden in sommige gevallen *MultiPolygon* en *MultiLineString* gebruikt waar ook gewoon respectievelijk *Polygon* en *LineString* gebruikt kunnen worden. Daarnaast worden bij de features niet altijd de juiste geometrische types gebruikt, zoals IMDF dat voorschrijft.

Optimalisatie IMDF data

Nadat de ruwe IMDF data aan Baseflow is aangeleverd, doet Baseflow nog het een en ander met de data voordat het in de applicaties van Baseflow gebruikt wordt. De IMDF data die in de applicaties gebruikt wordt voor het renderen van kaarten komt uit een database. Bij elk aangeleverd archief wordt de IMDF data dus in de database opgeslagen. Voordat dit gebeurt, worden er eerst bewerkingen op de IMDF data uitgevoerd. Dit gebeurt d.m.v. een script ontwikkeld door Baseflow en wordt gedaan om de dataset te ontdoen van data die niet gebruikt wordt in de applicatie en de dataset hiermee kleiner te maken. Daarnaast bundelt het script de data uit de GeoJSON bestanden en zet dit om naar een enkel JSON bestand.

Opvallend is dat alleen de bestanden met features van het type Building, Level, Unit en Fixture worden meegenomen bij deze bundeling. Alle mogelijke andere bestanden met andere features die worden aangeleverd worden dus niet gebruikt.

Het belangrijkste wat het script doet, naast het bundelen van de data uit alle losse GeoJSON bestanden, is het minder accuraat maken van alle coördinaten. De meeste coördinaten in de ruwe data hebben meer decimalen (tot 15 decimalen) dan nodig. Dit wordt d.m.v. het script teruggebracht naar 7 decimalen. Coördinaten met 7 decimalen bieden alsnog een precisie tot op de centimeter nauwkeurig, wat voor het gebruik in de applicaties meer dan voldoende is. Daarnaast filtert het script de *properties* van alle features. Alleen *level_id*, *category*, *ordinal* en *name* blijven daarbij over.

Gebruik IMDF in applicaties

De IMDF data wordt in de applicaties gebruikt om kaarten mee te renderen. Deze kaarten worden door klanten van Baseflow gebruikt voor o.a. indoor navigatie en het aanmaken van meldingen en points of interest (POI) op de kaart. In de applicaties wordt de geoptimaliseerde IMDF data zoals hierboven beschreven alleen gebruikt om de kaarten mee te renderen. De markers van alle points of interest worden namelijk als aparte laag er overheen gerenderd. De data voor deze POI's wordt uit de ruwe IMDF data gehaald met hetzelfde script dat wordt gebruikt voor optimalisatie. Hiervoor worden alle Points in Points.geojson (i.c.m. de bijbehorende Levels uit Levels.geojson) gebruikt.

Hierna wordt deze POI data apart in de database gezet. Hier is voor gekozen, omdat een van de belangrijke features van de applicaties is dat de gebruiker zelf POI's kan toevoegen, aanpassen en verwijderen. Met deze aanpak hoeft niet bij iedere wijziging door de gebruiker de IMDF data aangepast te worden en daarmee de hele map opnieuw gerenderd te worden. Daarnaast is een belangrijke feature van de applicatie dat gebruikers meldingen en taken kunnen aanmaken, waarmee ook markers worden aangemaakt. Deze markers worden in dezelfde laag gerenderd als de POI's.

Sommige IMDF features hebben in de huidige architectuur en bij het huidige gebruik van de app dus geen toegevoegde waarde. Dit zijn met name de features: Footprint, Section, Geofence, Detail, Opening en Relationship.

4.1.2. Gewenste situatie

Zoals in hoofdstuk 4.1.1 is beschreven, worden maar een beperkt aantal IMDF features gebruikt in de applicaties, namelijk Building, Level, Unit en Fixture. Deze features beschrijven samen de hele layout van de gebouwen die in kaart worden gebracht, wat benodigd is om de werkelijkheid realistisch genoeg in kaart te brengen voor het gebruik in de applicaties. Deze features zijn dan ook gewenst in het te genereren IMDF archief. Om aan het IMDF formaat te voldoen, zal ook voor elke feature de *feature_type* en een *id* worden meegenomen in de feature.

4.2. Welke bestandsformaten van digitale kaarten worden momenteel gebruikt voor het laten maken van kaarten in IMDF formaat?

In de huidige situatie worden de benodigde plattegronden door Baseflow opgevraagd bij de klant en doorgestuurd naar de derde partij. Om antwoord te kunnen geven op deze deelvraag is er contact gelegd met Asito, een klant van Baseflow die gebruik maakt van de functionaliteiten die Baseflow biedt m.b.t. indoor kaarten en navigatie. Asito gebruikt dit inmiddels voor meerdere locaties, waaronder Schiphol en de Technische Universiteit Eindhoven.

Uit dit contact is voortgekomen dat er tot op heden altijd plattegronden in CAD of PDF formaat worden aangeleverd. CAD staat voor ‘computer-aided design’ en omvat het gebruik van computers om ontwerpen mee te maken. Het wordt veel gebruikt bij het ontwerpen van elektronica en mechanica, maar ook bij het ontwerpen van gebouwen. PDF staat voor ‘portable document format’ en is simpelweg een bestandsformaat dat gebruikt wordt om documenten weer te geven en uit te wisselen. De weergave van een document in PDF formaat is altijd hetzelfde, ongeacht het systeem waar het op geopend wordt. PDF kan naast tekst zowel rasterafbeeldingen als vectorafbeeldingen bevatten. Voor dit project is PDF dat vector data bevat interessant (zie hoofdstuk 4.3).

Volgens Baseflow zijn CAD en SVG ook de formaten die de derde partij verwacht. Daarbij is het belangrijk dat er een plattegrond per verdieping wordt aangeleverd. Voor een voorbeeld van zo’n CAD tekening, zie Figuur 3.

4.3. Welke mogelijkheden zijn er om geometrische informatie uit digitale kaarten om te zetten naar data?

Om deze deelvraag te beantwoorden is er allereerst gekeken naar bestaande oplossingen. Er zijn een aantal bedrijven die aangeven IMDF data te kunnen genereren, zoals Mapxus (Mapxus, z.d.), Mappedin (Mappedin, z.d.) en Safe Software (Safe Software Inc., z.d.). Safe Software is daarbij de enige die aangeeft dat je met hun software bijvoorbeeld specifiek CAD tekeningen om kan zetten naar IMDF en je de mogelijkheid biedt om zelf hun software te proberen. Deze software is dan ook d.m.v. een gratis proefperiode uitgeprobeerd met de hoop wijzer te worden over de werkwijze van dergelijke software. De software is echter vrij specialistisch en daarnaast is de software ‘closed-source’ waardoor niet goed te achterhalen is hoe de software onder water werkt.

Na het onderzoeken van beschikbaar werk is er bij deze onderzoeksvraag een prototype gebouwd om te achterhalen wat de mogelijkheden zijn m.b.t. het extraheren en optimaliseren van geometrische data. Dit prototype is geschreven in JavaScript en maakt gebruik van SVG data. Zie hoofdstuk 4.3.4. Eerst worden CAD- en SVG kaarten afzonderlijk bekeken, alsook een vergelijk tussen de twee. Zie hoofdstuk 4.3.1 t/m 4.3.3.

4.3.1. CAD kaarten

CAD software maakt gebruik van vectoren (een verzameling punten met lijnen daartussen) en kan gebruikt worden voor zowel 2D als 3D ontwerpen. 2D CAD tekeningen bestaan altijd uit een verzameling van standaard vormen. Dit zijn: *Line*, *Polyline*, *Circle*, *Arc*, *Polygon* en *Ellipse*.

Autodesk AutoCAD (Autodesk, 2021), software waarmee CAD tekeningen gemaakt en ingelezen kan worden, heeft een functionaliteit waarmee verschillende data van CAD tekeningen geëxporteerd kan worden (Cohen, 2018) d.m.v. een 'Data Extraction Wizard'. Een van de dingen die je daarmee kan exporteren is de geometrische data van de vormen in het CAD bestand. Zo kan van *Lines* o.a. de start- en eindpositie in CSV (comma-seperated values) formaat geëxporteerd worden. Bij *Polyline*s kan alleen de totale lengte van de lijnen en de oppervlakte geëxporteerd worden.

Van een *Circle* kan de oppervlakte, straal (en diameter), omtrek en de center x- en center y coördinaten geëxporteerd worden. Van een *Arc* kan de lengte, oppervlakte, straal, start hoek, totale hoek en center x- en center y coördinaten geëxporteerd worden. Bij het maken van een *Polygon* kan gekozen worden hoeveel hoeken gewenst is. *Polygons* worden ook als *Polylines* geëxporteerd. Bij *Polygons* wordt dus ook alleen de totale lengte van de *Lines* en de oppervlakte geëxporteerd. Bij een *Ellipse* wordt alle ellipse-specifieke geometrie geëxporteerd, zoals center x/y, major axis vector x/y, minor axis vector x/y en major/minor radius.

Als laatste is er nog een 'hatch'. Een hatch is een patroon, zoals diagonale lijnen, dat een gesloten object opvult. Een hatch wordt niet als bijvoorbeeld losse lijnen geëxporteerd, maar gewoon als apart hatch object.

Behalve bij *Polylines* is dus van alle mogelijke 2D CAD vormen genoeg geometrische data te exporteren om deze vormen op een assenstelsel te recreëren. Voor *Polylines* is er wel een andere manier om aan de benodigde coördinaten te komen. Op *Polylines* kun je namelijk het commando 'list' uitvoeren wat in de console van AutoCAD een lijst geeft met alle coördinaten van alle hoekpunten (Autodesk, 2020). Het is ook mogelijk om deze output automatisch naar een log bestand te laten schrijven. Daarnaast is het mogelijk om *Polylines* eerst om te zetten naar *Lines*, d.m.v. het 'explode' commando (Autodesk, 2020).

4.3.2. SVG kaarten

De PDF bestanden zoals deze door de klant worden aangeleverd bevatten in de meeste gevallen vector tekeningen en zijn hoogstwaarschijnlijk naar PDF geëxporteerde CAD tekeningen. PDF tekeningen met raster data (i.t.t. vector data) worden in deze onderzoeksvraag niet meegenomen. De PDF bestanden met vector data zijn om te zetten naar SVG formaat (Scalable Vector Graphics). Dit kan bijvoorbeeld met Adobe Illustrator (Adobe, 2021) door het bestand simpelweg met de SVG extensie op te slaan. Vervolgens is de onderliggende XML data beschikbaar. Deze XML data bevat alle geometrie van de tekening.

Bij het omzetten van verschillende aangeleverde CAD en PDF tekeningen naar SVG valt op dat 4 van de 7 mogelijke SVG elementen het meest voorkomen en een betekenis hebben als het gaat om benodigde geometrie voor het renderen van een indoor kaart. Dit zijn: *Line*, *Polyline*, *Polygon* en *Path*.

Een SVG bestand kan bestaan uit 7 verschillende soorten vormen. Dit zijn *Line*, *Polyline*, *Rectangle*, *Circle*, *Ellipse*, *Polygon* en *Path*. Wat opvalt is dat de geometrie van deze vormen overeenkomt met de respectievelijke CAD vormen (zie hoofdstuk 4.3.1), maar dat de SVG vormen niet een-op-een overeenkomen met de CAD vormen. D.w.z. de geometrie van bijvoorbeeld een CAD *Polygon* is hetzelfde als die van een SVG *Polygon*, namelijk een aantal coördinaten met lijnen daartussen die samen een gesloten keten vormen. Maar, een CAD *Polygon* wordt niet per se omgezet in een SVG *Polygon* wanneer je de CAD tekening naar PDF exporteert en vervolgens omzet naar SVG formaat.

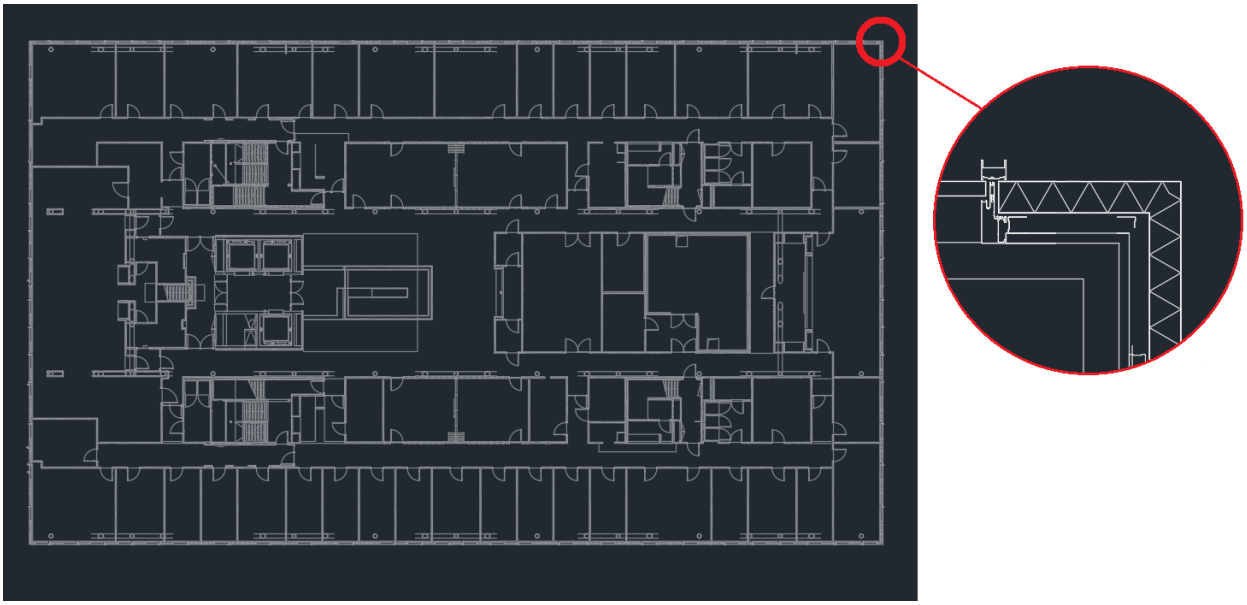
De SVG *Path* is een vreemde eend in de bijt. Met een *Path* kan ieder mogelijk te bedenken vorm getekend worden. Een path is een combinatie van rechte en gebogen lijnen. Een path kan dus een van de hierboven genoemde vormen representeren, maar ook vormen zoals bezier curves, quadratic curves etc. Daarnaast heeft een path niet een set met coördinaten, zoals alle andere SVG vormen, maar een set met instructies. Deze instructies geven samen aan wat en hoe de SVG vorm getekend moet worden.

Van alle vormen/vectoren in PDF tekeningen (en indirect CAD tekeningen) is er dus een snelle manier om de geometrische data in te zien door de PDF om te zetten naar SVG. Bij alle standaard SVG vormen is er voldoende geometrische data beschikbaar om deze op een assenstelsel te recreëren, en is rechttoe rechtaan.

De data van complexere vormen die zijn opgebouwd uit instructies (paths) i.p.v. alleen geometrie zijn om te zetten in een lijst met coördinaten waarlangs de vorm is getekend. Hiervoor kunnen de methodes *getTotalLength* en *getPointAtLength* van de *SVGPathElement* API (MDN Contributors, 2020) gebruikt worden. Hierbij maakt het dus ook niet uit welke (combinatie) van de 10 mogelijke instructies een path gebruikt.

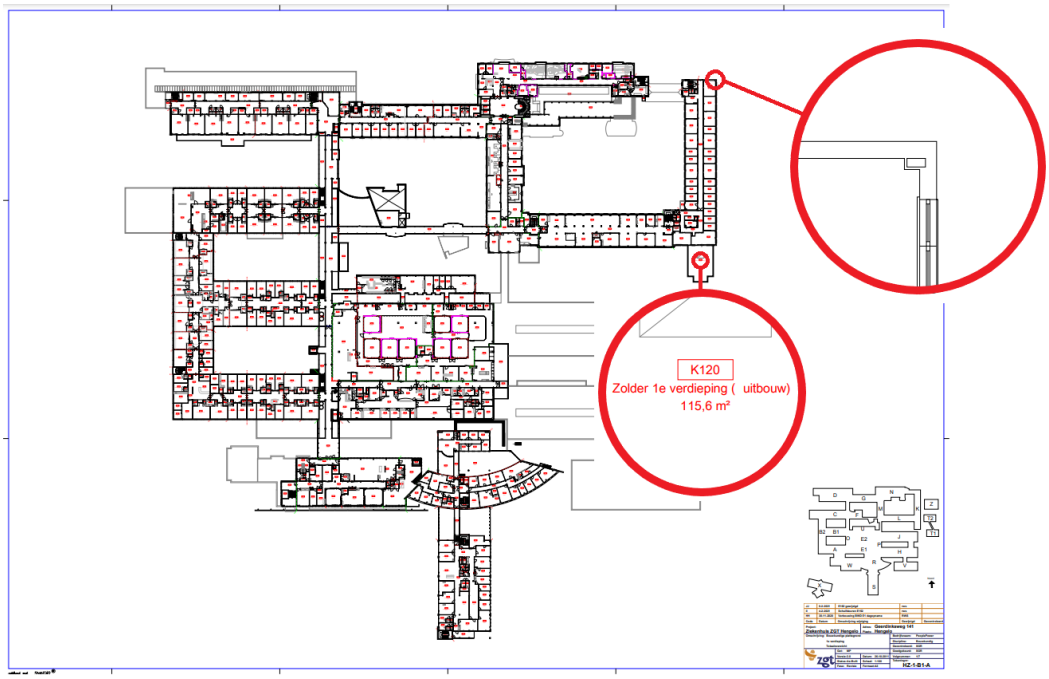
4.3.3. Vergelijking CAD- en PDF data

Een van de eerste problemen dat zich voordoet bij zowel CAD- als PDF tekeningen, is dat er veel onderdelen/geometrie van de tekeningen overbodig is voor het renderen van indoor kaarten. Figuur 3 laat een voorbeeld zien van een CAD tekening zoals deze is aangeleverd door een klant en geopend in Autodesk AutoCAD. Na het uitzetten van lagen die alleen maar overbodige geometrie bevatten, blijft dit over. Zoals te zien is in de close-up van een van de hoeken van de tekening, aan de rechter kant van Figuur 3, bestaat de tekening uit veel meer geometrie dan benodigd (en gewenst) is voor het IMDF formaat.



Figuur 3 - CAD tekening in Autodesk AutoCAD

De PDF tekeningen die aangeleverd worden door klanten lijken over het algemeen wat minder geometrie te bevatten. Zie de close-up rechtsboven in het voorbeeld in Figuur 4. Echter is dit nog steeds te veel. Daarnaast bevatten de meeste aangeleverde PDF tekeningen nog andere overbodige data. Dit zijn bijvoorbeeld aanduidingen van ruimtes, zoals te zien is in de onderste close-up in Figuur 4, rasters, title blocks, etc. Deze zijn in de PDF niet uit te zetten of weg te laten, zoals dat bij CAD mogelijk is.



Figuur 4 - PDF tekening

Nu duidelijk is hoe geometrische informatie uit verschillende soorten tekeningen gehaald kan worden, is het vinden van manieren om deze data te "versimpelen" de volgende stap. Anders is deze data nooit bruikbaar om omgezet te worden naar IMDF formaat en ook niet voor het gebruik in applicaties. En ook als dit wel omgezet zou kunnen worden, dan zou de IMDF data set veel te groot worden en de applicaties zouden zoveel data nooit (snel genoeg) kunnen inlezen en renderen. Tevens zou een gerenderde kaart met zoveel geometrie niet gebruiksvriendelijk zijn voor de eindgebruiker.

Een voordeel van SVG is dat deze te renderen is in een webbrowser. Hier is SVG in essentie ook voor gemaakt. Daarnaast kan de SVG data d.m.v. code gemanipuleerd worden. Hierna is het resultaat van de gemanipuleerde data direct weer zichtbaar te maken door het opnieuw te renderen. Om te kijken wat de mogelijkheden zijn in het versimpelen/optimaliseren van geometrie is er een HTML5 project opgezet dat een SVG bestand kan inlezen en renderen (zie hoofdstuk 4.3.4). Dit helpt ook om beter te kunnen zien uit welke SVG geometrie, en relaties daartussen, een naar SVG omgezette tekening nou bestaat. De vormen en geometrie in een SVG bestand zijn immers individueel goed leesbaar, maar in een bestand met duizenden regels is het onmogelijk om een verband te zien tussen geometrische objecten.

Hierbij is het ook goed om op te merken dat CAD tekeningen altijd om te zetten zijn naar PDF (en indirect dus ook naar SVG). Afhankelijk van de inhoud van de PDF, is dit andersom niet altijd even goed mogelijk. Software zoals AutoCAD heeft namelijk wel de mogelijkheid om PDF bestanden te importeren, maar als de PDF geen vector data bevat wordt deze omgezet naar een (raster) afbeelding. De geometrie uit de geïmporteerde data kan in dat geval ook niet in AutoCAD bewerkt worden (Autodesk, 2021).

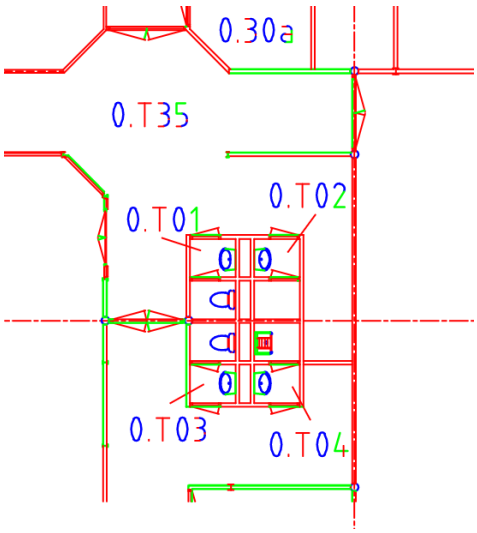
4.3.4. SVG script voor visualiseren en optimaliseren SVG data

Door gebruik te maken van SVG en de mogelijkheid om extra geometrie te tekenen in SVG in de webbrowser i.c.m. de DevTools debugger, is beter te achterhalen wat er precies gebeurt bij het handmatig testen/uitvoeren van verschillende algoritmes oftewel bewerkingen op de geometrische data.

4.3.4.1. Visualiseren SVG data

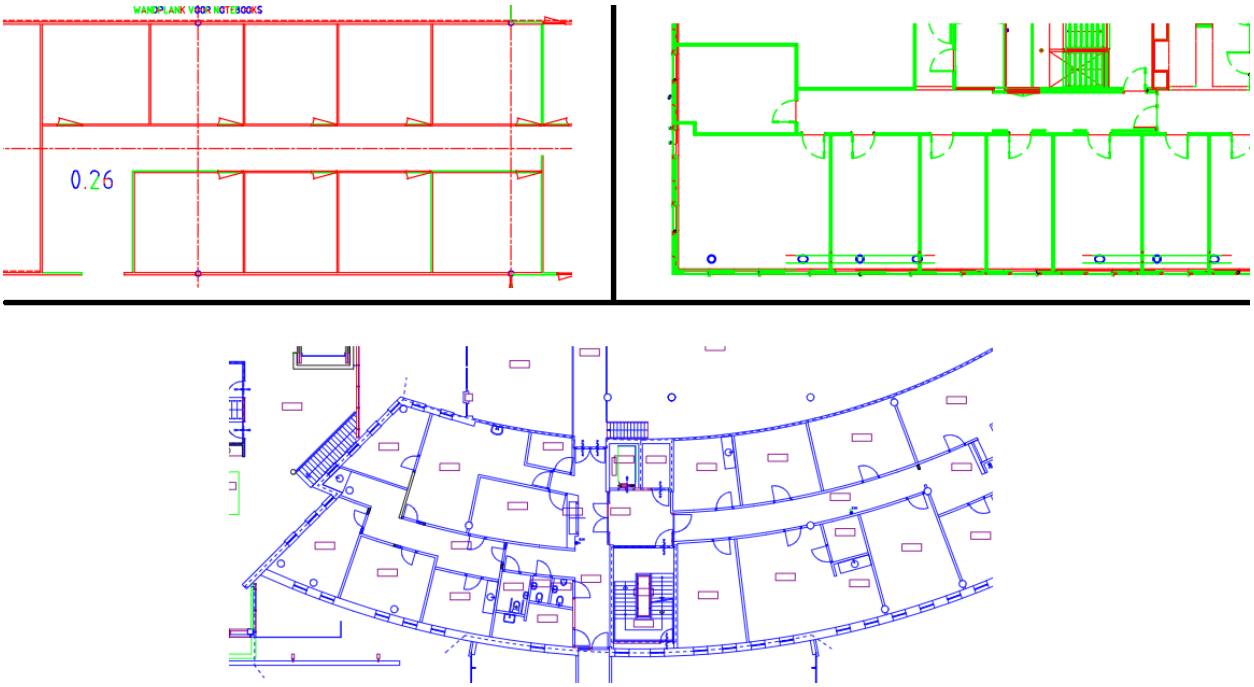
Allereerst is er begonnen aan het schrijven van een script (m.b.v. JavaScript) dat een SVG bestand kan inlezen en renderen. Om duidelijk de samenhang en relaties tussen verschillende geometrische types te visualiseren, is ervoor gezorgd dat elk geometrisch type met een druk op de knop los van elkaar, en met verschillende kleuren, gerenderd kan worden.

Hieruit bleek al gauw dat de relaties tussen verschillende geometrische types niet altijd logisch zijn. Een muur is bijvoorbeeld in sommige gevallen een *Line* en in andere gevallen een *Polyline*. Zie Figuur 5, waarbij lijnen rood zijn, *Polylines* groen en *Paths* blauw. Eén en dezelfde muur (binnen- en buitenmuur) kan dus al bestaan uit twee verschillende geometrische types.



Figuur 5 - "Gehusselde" geometrie in SVG

Daarnaast zitten er ook grote verschillen tussen tekeningen. Zo bestaat de ene tekening voor het grootste gedeelte uit *Lines*, de andere uit *Polylines*, en weer een andere uit *Paths*, zie Figuur 6. Omdat de meeste tekeningen veelal uit *Lines* en *Polylines* bestaan, zijn alle *Polylines* omgezet naar *Lines*, zodat de optimalisatie algoritmes (zie hoofdstuk 4.3.4.2) alleen met dit geometrisch type te maken hebben.



Figuur 6 – Delen van 3 verschillende SVG tekeningen

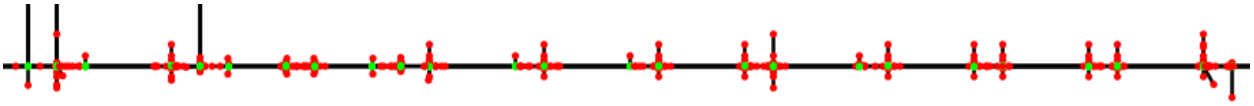
4.3.4.2. Optimaliseren data

Om de resultaten van de hieronder beschreven algoritmes te illustreren, zal gebruik gemaakt worden van een deel van een SVG tekening. De originele en onbewerkte tekening is te zien in bijlage 1.

De optimalisaties beginnen bij het in lijn brengen van alle SVG lijnen door ze te verschuiven naar een punt in een verzameling van vooraf gedefinieerde punten. Oftewel: een grid. Met deze aanpak zou er in theorie minder chaos van geometrie over moeten blijven (geometrie dat niet meer op elkaar aansluit, van elkaar verschoven is, etc.), omdat lijnen dan over elkaar komen te liggen. Het idee om met zo'n grid te werken is tijdens een brainstorm meeting naar boven gekomen nadat verschillende andere aanpakken geprobeerd waren (zie document Onderzoeksrapport hoofdstuk 1.5.4.).

Het gridsysteem is gemaakt d.m.v. een twee dimensionale array waarbij elke array in de array een rij met punten bevat. Met een druk op de knop worden alle lijnen verschoven naar het dichtstbijzijnde gridpunt. Om dit voor elkaar te krijgen kijkt het algoritme naar welk punt van de grid het dichtst bij de start van een lijn (punt x1/y1) komt.
Vervolgens krijgt dat punt van de lijn dezelfde coördinaten als die van het gridpunt. Het tweede punt van de lijn (punt x2/y2) wordt met dezelfde afstand en in dezelfde richting verschoven als het eerste punt, zodat de lijn 'intact' blijft.
De gehele lijn verschuift dus als het ware en hiermee komen de beginpunten van alle lijnen op de grid te liggen. Zie bijlage 2.

Een volgende logische stap was, nu lijnen boven op elkaar liggen en oorsprong hebben in dezelfde punten, om duplicaten te verwijderen. Hiervoor is een algoritme geschreven dat lijnen die dezelfde start- en eindpunten hebben als een andere lijn te verwijderen. Na het uitvoeren hiervan bleek dat er ook vóór het snappen naar de grid veel duplicaten zijn. Daarnaast bleek dat, ondanks dat duplicaten gefilterd zijn, er veel lijnen zijn die elkaar overlappen. Zie Figuur 7, waarbij de start- en eindpunten van de lijnen respectievelijk groen en rood zijn gemaakt.



Figuur 7 – Close-up overlappende lijnen

Voordat andere optimalisatie algoritmes uitgevoerd kunnen worden, moeten deze overbodige overlappende lijnen eerst weg zijn. Ook hier is een algoritme voor geschreven. Van elke lijn worden alle lijnen verzameld die dezelfde oriëntatie hebben en die (deels) met de lijn overlappen. Vervolgens wordt bepaald wat de kleinste en grootste punten zijn. D.w.z. bijvoorbeeld in het geval van horizontale lijnen wordt gekeken naar alle x-coördinaten van alle lijnen en wordt daarvan de laagste waarde en hoogste waarde gepakt. Deze twee punten worden gebruikt om een nieuwe lijn te maken. Als laatst worden alle verzamelde lijnen en de originele lijn verwijderd. Op deze manier worden de lijn en alle lijnen die er mee overlappen gemerged.

De nieuwe gecombineerde lijnen kunnen ook weer overlappen met andere lijnen. Daarom moet het filteren van overlappende lijnen een aantal keer gebeuren, totdat alle overlappende lijnen samengevoegd zijn. Door overlappende lijnen te combineren, veranderden de relaties tussen punten. Hierdoor komt het dus ook voor dat de nieuwe startpunten van sommige nieuwe (gecombineerde) lijnen niet meer op de grid liggen. Hierdoor moet ook het snap to grid algoritme opnieuw gedraaid worden. Omdat hiermee lijnen weer verschuiven, kunnen er dus ook weer lijnen gaan overlappen. Afhankelijk van de tekening moet het proces van overlappende lijnen verwijderen en het snappen naar een grid dus meerdere malen gebeuren.

Vooral bij grotere en complexere tekeningen valt op dat er na het snappen naar een grid, en zelfs na het combineren van overlappende lijnen, veel korte en overbodige lijntjes over blijven. Een ander algoritme zorgt ervoor dat deze lijnen verwijderd worden. D.m.v. een invoerveld kan worden aangegeven bij welke drempelwaarden dit moet gebeuren, waarbij er voor elk type oriëntatie (horizontaal, verticaal, diagonaal) een aparte waarde kan worden opgegeven. Zo kan ervoor gezorgd worden dat niet te veel lijnen worden verwijderd wanneer lijnen met een bepaalde oriëntatie een hogere drempelwaarde vereisen om gefilterd te worden.

Bijlage 3 laat het resultaat zien na het samenvoegen van overlappende lijnen en het filteren van korte lijnen. Deze tekening heeft nu aanzienlijk minder lijnen en sommige hoeken sluiten nu beter op elkaar aan.

Hoewel veel lijnen nu wel op elkaar aansluiten, zijn er zijn nog steeds lijnen die ‘door schieten’ m.b.t. de grenzen van de tekening. Dit zijn allemaal lijnen die een eindpunt buiten de ingestelde grid hebben; de lijnen worden immers aan de hand van het startpunt op de grid gesnapt. De volgende stap was dan ook om deze eindpunten ook op de grid te zetten, en daarmee de lijnen als het ware in te korten. Het resultaat van dit algoritme is te zien in bijlage 4. Alle hoeken sluiten nu goed op elkaar aan en alle andere lijnen die eerder doorschoten sluiten nu netjes op andere lijnen aan.

De volgende stap in het onderzoeken van optimalisatiemogelijkheden was het opvullen van (overbodige) gaten. Hiervoor is een algoritme geschreven waarmee, wederom met gebruikersinput voor de drempelwaardes, lijnen die in het verlengde van elkaar liggen op elkaar aan worden gesloten. Dit algoritme kijkt naar de 2 x- of y-coördinaten van een gat van lijnen die in het verlengde van elkaar liggen. Als de afstand daartussen onder de opgegeven drempelwaarde valt, dan worden de 2 lijnen verwijderd en komt er een nieuwe lijn voor in de plaats die loopt vanaf het uiterste punt van lijn A naar het uiterste punt van lijn B. Zie bijlage 5 voor het resultaat.

Na het uitvoeren van alle bovenstaande algoritmes in verschillende volgordes en met verschillende invoer, is het resultaat zoals in bijlage 5 te zien is vele malen beter dan de originele tekening in de figuur in bijlage 1. Deze specifieke tekening gaat van 2375 naar 16 lijnen en met minimaal handmatig werk zou de tekening nog wat verder geoptimaliseerd kunnen worden.
Echter, bij grotere en complexere tekeningen laat het resultaat nog veel handmatig werk over. De figuur in bijlage 6 laat het resultaat zien van het uitvoeren van de eerder genoemde algoritmes op de tekening uit Figuur 3[1].

Zoals te zien is zijn de buitenmuren wederom prima. Ze sluiten goed op elkaar aan en alle andere geometrie ligt erbinnen. Er blijven echter nog veel lijnen over die overbodig zijn en er blijven ook veel lijnen over die niet goed op elkaar aansluiten. Dit laatste zijn lijnen die te kort zijn waardoor ze niet aansluiten of juist te lang zijn waardoor ze door schieten. Om te bedenken hoe een nog beter resultaat behaald kan worden is een brainstormsessie met het team gepland. Hier kwamen in hoofdlijnen twee dingen uit, waarvan één na implementatie van een nieuw algoritme een positief resultaat gaf. Dit is namelijk dat een deel van de overbodige lijnen wat met elkaar gemeen hebben. Ze sluiten namelijk nergens op aan en ‘zweven’ dus.

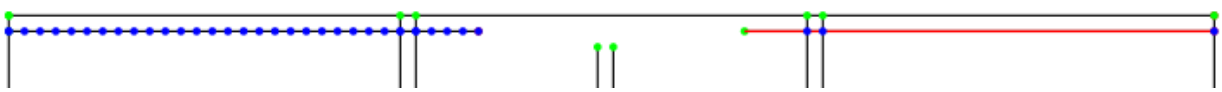
[1] Vanaf nu zullen illustraties m.b.t. optimalisaties de resultaten laten zien van het uitvoeren van algoritmes op deze complexere tekening

Met zwevende lijnen worden lijnen bedoeld waarvan het begin- en eindpunt vrij ligt. M.a.w. deze punten liggen niet in het verloop van een andere lijn. Deze lijnen sluiten dus nergens op aan. Het filteren van deze lijnen zou sowieso een beter resultaat opleveren. In het voorbeeld van bijlage 6 zijn alle zwevende lijnen immers overbodige lijnen. Hier is dan ook een algoritme voor geschreven. Dit algoritme kijkt naar alle gridpunten die in het verloop van een lijn liggen. Vervolgens wordt er gekeken of en welke lijnen er zijn die beginnen of eindigen op een van deze gridpunten. De lijnen die hier gevonden worden, blijven bewaard. Op deze manier blijven op het eind dus alleen maar lijnen over die wel ergens op aansluiten en zijn alle zwevende lijnen eruit gefilterd. Het resultaat is te zien in bijlage 7.

Direct is te zien dat sommige lijnen die eerder aansloten op een zwevende lijn, nu ‘half zweven’. D.w.z. alleen het begin- of eindpunt sluit aan op een andere lijn. Veel van deze lijnen moeten iets verlengd worden om op de juiste plek op een andere lijn aan te sluiten. Ze sloten voor het filteren van zwevende lijnen dus op een verkeerde lijn aan, namelijk een overbodige zwevende lijn. Zie cirkels 1 en 2 in bijlage 7. Dit bracht het idee om deze lijnen te verlengen om hiermee aan te sluiten op de dichtstbijzijnde lijn. Daarbij kunnen de lijnen de half zwevende lijnen die te lang zijn (doorschieten), zoals de lijnen in cirkel 3, ingekort worden om hiermee ook aan te sluiten op de dichtstbijzijnde lijn. Sommige overbodige lijnen zouden daardoor ook verdwijnen, zoals de lijn aangegeven met cirkel 4, omdat het beginpunt van de lijn samen komt te liggen met het eindpunt.

Om dit voor elkaar te krijgen is een algoritme geschreven die gebruik maakt van alle gridpunten waar lijnen overheen lopen. Allereerst worden al deze gridpunten per lijn verzameld. Vervolgens wordt er van een lijn gekeken hoe vaak de gridpunten van het begin- en eindpunt van de lijn voorkomen. Als dit maar 1 keer is, dan betekent het dat dat punt van de lijn nergens op aan sluit. Hiermee wordt dus bekend welke lijnen half (of nog volledig) zweven.

Vervolgens wordt voor elk van deze lijnen gekeken welke van de verzamelde gridpunten (de gridpunten waar alle andere lijnen overheen lopen) in het verlengde liggen van de lijn. Figuur 8 illustreert dit. De rode lijn is de lijn die bekeken wordt. Het beginpunt van deze lijn (zie groene punt) zweeft namelijk. De blauwe punten zijn alle gridpunten waar andere lijnen overheen lopen én in het verlengde liggen van de zwevende lijn.



Figuur 8 - Algoritme Illustratie aansluiten zwevende lijnen

Van deze gridpunten wordt berekend welk punt het dichtst bij het zwevende punt ligt, waarna de zwevende lijn verlengd of ingekort wordt zodat het zwevende punt op het dichtstbijzijnde gridpunt komt te liggen. De rode lijn uit Figuur 8 wordt dus ingekort. Bijlage 8 laat het resultaat zien van dit algoritme.

4.3.4.3. Latere inzichten SVG optimalisaties

In de realisatiefase van het project zijn er nog een aantal dingen naar boven gekomen die invloed hebben op het resultaat van de optimalisaties. Zo bleek dat sommige algoritmes langer duurden dan nodig, omdat er veel lijnen zijn die uit een punt bestaan. D.w.z. lijnen waarbij het begin- en eindpunt op elkaar liggen. Het is dan ook het beste deze lijnen te filteren alvorens alle andere algoritmes uit te voeren. Een ander algoritme maakt dit mogelijk.

Daarnaast kon veel code minder complex gemaakt worden als lijnen eerst ‘in dezelfde richting’ gelegd worden. M.a.w. als alle horizontale lijnen het startpunt links van het eindpunt hebben liggen, en alle verticale lijnen het startpunt boven het eindpunt hebben liggen. Dit slaat vooral op de functies/algoritmes die benodigd zijn voor unit herkenning (zie hoofdstuk 4.3.5).

4.3.5. Script voor unit herkenning

Een minimale extra stap voordat de geometrische data omgezet kan worden naar geografische data dat zinnig is voor het IMDF formaat, is het omzetten van alle lijnen naar units. In de IMDF data wil je immers geen geografische informatie hebben van honderden of zelfs duizenden losse lijnen, maar gegroepeerde geografische data van bijvoorbeeld ruimtes. Dus, lijnen moeten gecombineerd en omgezet kunnen worden naar polygonen. Er is dan ook een algoritme geschreven die dit geautomatiseerd doet. Dit algoritme maakt gebruik van de kennis over welke gridpunten gerelateerd zijn aan alle lijnen. Alle lijnen liggen nu immers op een grid en sluiten op elkaar aan.
Voordat dit algoritme uitgevoerd kan worden, zorgt een ander algoritme er eerst voor dat alle lijnen opgesplitst worden waar deze aansluiten op andere lijnen en dus een kruising vormen. Stel je hebt 2 lijnen. Lijn A sluit in het midden aan op lijn B. Dan wordt lijn B opgesplitst daar waar lijn A aansluit, en zijn er nu 3 lijnen. Dit is nodig, omdat het algoritme voor unit herkenning als het ware lijnen in een bepaalde richting gaat "tracken". Om deze reden moet er nog een ander algoritme eerst uitgevoerd worden. Dit is een algoritme die alle lijnen in dezelfde richting legt (zie hoofdstuk 4.3.4.3).

Het algoritme begint het tracken steeds bij het begin van een willekeurige lijn. Dan kijkt het naar de eerste andere lijn die haaks op de lijn staat die getrackt wordt. Het tracken vervolgt zich nu in die betreffende lijn, en de richting die genomen is (d.w.z. met de klok mee of tegen de klok in) wordt onthouden.

Vervolgens volgt het steeds de eerstvolgende lijn die haaks op de lijn staat die dan getrackt wordt, mits het dezelfde richting volgt als de eerste "afslag" die genomen is. In de gevallen dat er geen lijn haaks aansluit op het eindpunt van de lijn die getrackt wordt (d.w.z. er sluit alleen een lijn op aan die in het verlengde ligt), dan wordt die lijn getrackt. Bij het toevoegen van zo'n lijn aan de unit wordt deze gemerged met de vorige lijn. Dit voorkomt dat er in de vervolgstap polygonen met overbodig veel punten gegenereerd worden.

Op deze manier eindigt het tracken bij het startpunt van de lijn waar het tracken begonnen is, of het nou om een rechthoek of veelhoek gaat. Is het tracken terug bij af, dan is er een unit gevonden. Zie bijlage 9. Het algoritme begint nu opnieuw met tracken bij een volgende willekeurige lijn. Dit resulteert in sommige gevallen in een unit dat al eerder gevonden is. In dat geval wordt de gevonden unit niet opnieuw opgenomen in de lijst met herkende units. De uiteindelijke gevonden units bestaan elk uit een verzameling van 4 of meer lijnen.

4.3.6. Conclusie

Geometrische data kan dus het best uit digitale kaarten worden gehaald wanneer deze in SVG formaat aan te leveren zijn. Dit is mogelijk bij zowel CAD als PDF kaarten door deze om te zetten naar SVG. Door het gebruik van een set aan algoritmes is deze SVG data te optimaliseren waarmee de dataset drastisch kleiner wordt.

4.4. Hoe kan de verkregen geometrische data omgezet worden naar data in het IMDF formaat?

De verkregen SVG coördinaten (zie hoofdstuk 4.3.4) zijn coördinaten die behoren tot een cartesisch coördinatenstelsel. Coördinaten in het IMDF formaat moeten geografische coördinaten zijn in lengte- en breedtegraad notatie (zie hoofdstuk 2). De cartesische coördinaten moeten dan ook omgezet worden naar geografische coördinaten. Dit proces wordt georeferencing genoemd (GIS Geography, 2021). De geografische coördinaten die hier uit voortkomen zijn de basis voor de te genereren IMDF data.

Om georeferencing toe te passen moet een tekening eerst de juiste transformaties doorstaan zodat de locatie, rotatie en schaalverdeling overeenkomen met de situering van een gebouw in de werkelijkheid. Hier komt hoe dan ook handmatig werk bij kijken. Het is bijvoorbeeld mogelijk om bij CAD kaarten hier Google Earth (zie Onderzoeksrapport hoofdstuk 1.6.1) of gespecialiseerde software zoals ArcMap (Esri, 2020) voor te gebruiken. Met deze oplossingen worden er echter afhankelijkheden gecreëerd, zowel van externe software als een specifiek formaat (CAD in bovenstaande voorbeelden).

De snelste en meest onafhankelijke manier van georeferencing, zonder gebruik van externe software, is dan ook om de gebruiker de mogelijkheid te geven om een tekening, zoals de bewerkte SVG (zie hoofdstuk 4.3.4), over een satelliet kaart heen te leggen en de nodige transformaties uit te voeren.

Transformeren tekening: roteren, schalen en transleren
Om transformaties mogelijk te maken is het SVG script (zie hoofdstuk 4.3.4) uitgebreid. De overgebleven set aan geometrie kan d.m.v. sliders geroteerd en geschaald worden. Hierbij wordt alle geometrie d.m.v. standaard wiskunde individueel getransformeerd. Eerst wordt het middelpunt van de tekening berekend, waarna alle geometrie zodanig getransleerd wordt dat het middelpunt van de tekening in de oorsprong van het SVG canvas komt te liggen. Waar dit bij standaard assenstelsels linksonder is, is dit bij SVG linksboven.

Vervolgens worden de benodigde bewerkingen uitgevoerd: roteren of schalen. Dit roteren en schalen gebeurt steeds met respectievelijk een relatieve hoek of factor t.o.v. de eerder opgegeven waardes. Op deze manier hoeft niet steeds de tekening teruggebracht te worden naar de originele rotatie of schaalgrootte alvorens deze naar de ingestelde waarde geroteerd of geschaald kan worden. Als laatst worden alle lijnen weer zodanig getransleerd zodat ze weer op de originele positie op het canvas komen. Translatie van de tekening wordt gedaan d.m.v. drag-and-drop met de muis vanaf het middelpunt van de tekening. Het bijbehorend algoritme transleert hierbij constant de tekening naar de nieuwe positie van de muis.

Toepassen juiste transformaties d.m.v. satelliet kaart
De volgende stap was het implementeren van een interactieve satelliet kaart waar de SVG tekening overheen gelegd kan worden zodat de gebruiker de juiste transformaties kan uitvoeren. Hierbij is Leaflet gekozen, omdat dit een JavaScript library is, het open-source is, en vooral ook omdat het de mogelijkheid geeft om een 'SVG Overlay' te gebruiken (Agafonkin, z.d.).

De kaart i.c.m. SVG Overlay is op zo'n manier geïmplementeerd dat de gebruiker na het inladen van de kaart twee hoekpunten kan aanklikken waartussen het SVG canvas gerenderd zal worden. Vervolgens kan de gebruiker de benodigde transformaties uitvoeren om de tekening precies over het juiste gebouw te leggen.

Omzetten cartesische coördinaten naar geografische coördinaten
De volgende en laatste stap is het omzetten van de SVG coördinaten naar geografische coördinaten. Na onderzoek is er een aanpak gevonden om dit voor elkaar te krijgen. Leaflet heeft de mogelijkheid om pixel coördinaten om te zetten naar geografische coördinaten d.m.v. de *containerPointToLatLng* methode. Deze methode verwacht een Leaflet Point, een punt met x/y-coördinaten.

Als de pixel locaties van alle punten van de geometrie op de Leaflet kaart bekend zijn, kan dit gebruikt worden om daarmee de geografische coördinaten te bepalen. Er is een algoritme geschreven die deze pixel locaties achterhaald. In essentie kijkt dit algoritme naar de hoekpunten van de denkbeeldige rechthoek dat een geometrisch object omvat d.m.v. de HTML *getBoundingClientRect* methode.

5 Ontwerp

Bij dit project is ontwerpen op verschillende manieren aan bod gekomen. Tijdens de onderzoeksfase is gaandeweg een groot deel van de algoritmes ontworpen die in de uiteindelijke PoC zijn gebruikt. Zie hoofdstuk 4.3.4. Nadat het onderzoek grotendeels was afgerond, was het tijd om te bouwen aan het PoC. Voordat aan de implementatie hiervan begonnen is, is er nagedacht over het ontwerp voor zowel de softwarearchitectuur alsook de voorkant van de applicatie waar de gebruiker interactie mee heeft. Dit hoofdstuk zal ingaan op het ontwerp van het PoC.

Voor aanvang van de ontwerp- en implementatiefase is er geen requirementsanalyse uitgevoerd. Op verzoek van de opdrachtgever zijn er in plaats daarvan bij elke scrum meeting (zie hoofdstuk 3) in samenspraak prioriteiten/requirements vastgesteld voor elke sprint. Tijdens de eerste scrum meeting zijn de belangrijkste requirements/systeemeisen besproken:

- het PoC moet een SVG bestand als invoer accepteren (must);
- de uitvoer moet IMDF data zijn dat te downloaden is (must);
- de IMDF data moet geoptimaliseerde data bevatten, waardoor de grootte van de dataset beperkt blijft (must);
- er moet een standaard pipeline (zie hoofdstuk 5.2) beschikbaar zijn (must) en het moet mogelijk zijn om een pipeline samen te stellen (should);
- de applicatie moet gebruiksvriendelijk zijn (must).

5.1. High-level ontwerp softwarearchitectuur

Het prototype dat tijdens de onderzoeksfase gaandeweg is gebouwd bestaat uit simpelweg een enkel JavaScript/HTML project. Voor het op te leveren PoC is ervoor gekozen om een splitsing te maken tussen front- en backend. De backend is in grote lijnen verantwoordelijk voor het uitvoeren van alle bewerkingen/algoritmes. De frontend is verantwoordelijk voor de datapresentatie, gebruikersinteractie en georeferencing. De splitsing heeft bij de use-case van dit project een aantal voordelen:

- De client wordt ontlast doordat alle ‘zware’ algoritmes en reken logica niet in de browser uitgevoerd worden;
- Alle belangrijke logica/algoritmes zijn door derden niet zomaar uit de frontend te halen;
- De applicatie is (potentieel) sneller, wanneer de backend op een relatief snelle server wordt gedraaid en er eventueel aandacht is besteed aan load balancing;

- Toekomstige integraties met andere (interne) systemen is mogelijk. Zo kan de gegenereerde IMDF data bijvoorbeeld rechtstreeks geüpload worden naar het systeem dat er gebruik van gaat maken;
- Het genereren van de IMDF data (het archief) in de backend en het downloaden ervan vanuit de frontend zorgt ervoor dat het archief consistent is over verschillende systemen.

Communicatie met een server betekent echter ook dat er veel data heen en weer gestuurd wordt. Om die hoeveelheid data te verkleinen, wordt elke response body gecomprimeerd voordat het over de lijn gestuurd wordt. Daarnaast worden attributen van classes die alleen in de backend benodigd zijn voor de algoritmes uitgesloten van de responses d.m.v. serialisatie.

De algoritmes die gaandeweg het onderzoek ontworpen en ontwikkeld zijn, zijn geschreven in JavaScript. Dit was tijdens het onderzoek de beste keuze, omdat de resultaten van de algoritmes snel inzichtelijk gemaakt konden worden door de combinatie van HTML voor het renderen van de SVG na het uitvoeren van bewerkingen op de SVG data en JavaScript voor het uitvoeren van deze bewerkingen. Hoewel het discutabel is of JavaScript de beste taal is voor de use-case van dit project, is ervoor gekozen om het PoC niet in een radicaal andere taal te schrijven. Een groot deel van de algoritmes en logica uit het prototype moesten namelijk ook in het PoC komen.

Wel is ervoor gekozen om de backend in TypeScript (superset van JavaScript) te schrijven i.p.v. JavaScript. Door gebruik te maken van de ‘explicit typing’ dat TypeScript biedt is de code robuuster, minder foutgevoelig, beter leesbaar en beter schaalbaar. Als backend framework is gekozen voor NestJS, dat TypeScript volledig ondersteund. Daarnaast biedt het een combinatie van object oriented programming en functional programming, wat voor het PoC een ideale combinatie is.

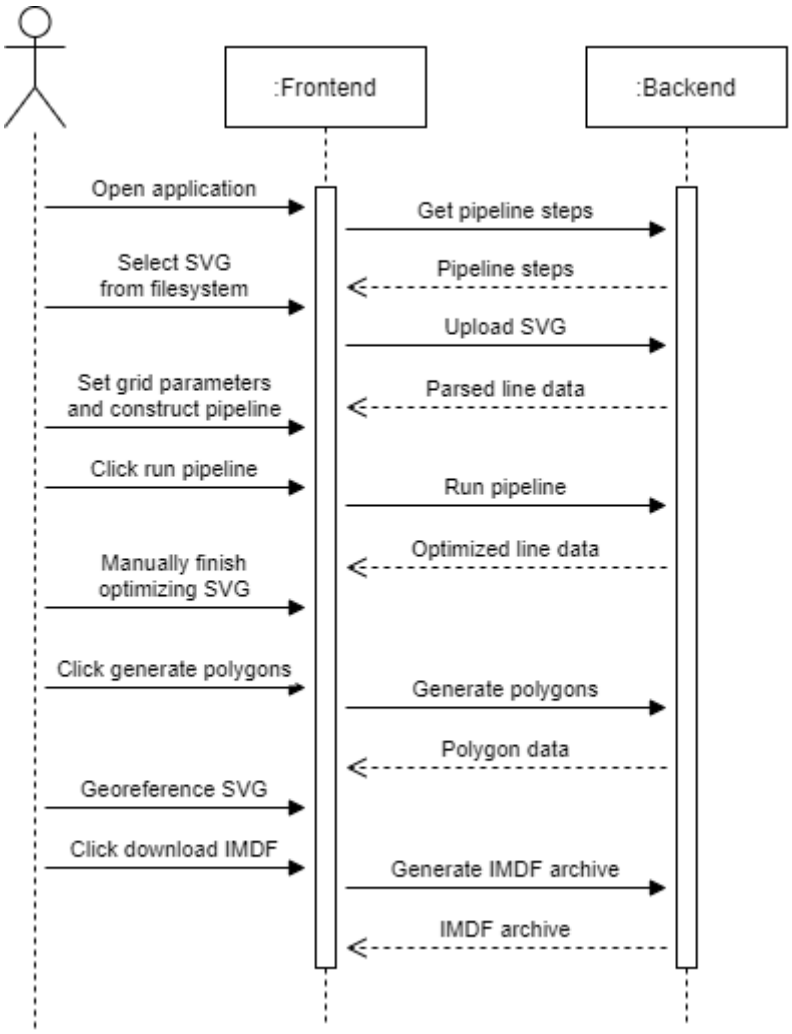
5.2. Workflow

Het PoC maakt gebruik van een pipeline. Deze pipeline is een opeenvolging van stappen, waarbij elke stap een algoritme (zie hoofdstuk 4.3.4.2) representeert. De pipeline bestaat initieel uit een vast aantal stappen in een vaste volgorde, maar is configureerbaar zodat de gebruiker per situatie een andere pipeline kan samenstellen.

Figuur 9 laat een sequence diagram zien van het systeem. Hier is de gebruikersinteractie met de frontend en de communicatie tussen frontend en backend te zien. Daarnaast geeft dit een high-level overview van de workflow van het systeem, van het openen van de applicatie tot het downloaden van IMDF.

De gebruiker moet om te beginnen een SVG bestand uploaden. De backend leest het geüploade bestand uit en parsed de inhoud. D.w.z. voor alle vectordata in het bestand worden objecten aangemaakt. Deze objecten zijn in de algoritmes immers makkelijker te manipuleren. De backend filtert daarna eerst alle duplicaten en lijnen die een punt representeren (lijnen waarvan het eindpunt hetzelfde is als het beginpunt). Hierdoor wordt de data set een stuk kleiner. Daarna stuurt de backend deze geparste data terug waarna de frontend de tekening rendert.

Vervolgens kan de gebruiker de grid naar wens instellen, alsook de pipeline samenstellen met de gewenste (volgorde van) stappen en de benodigde parameters per stap. Daarna kan de pipeline uitgevoerd worden. De backend voert voor alle stappen in de pipeline de bijbehorende algoritmes uit, waarna het de bewerkte vectordata terug stuurt naar de frontend. De gebruiker kan nu nog wat handmatige bewerkingen op de tekening uitvoeren, namelijk het verwijderen van overgebleven overbodige lijnen. Als laatst kan de gebruiker de tekening op de juiste manier op de kaart leggen en met een druk op de knop de IMDF data downloaden.



Figuur 9 - High-level sequence diagram systeem

5.3. Backend

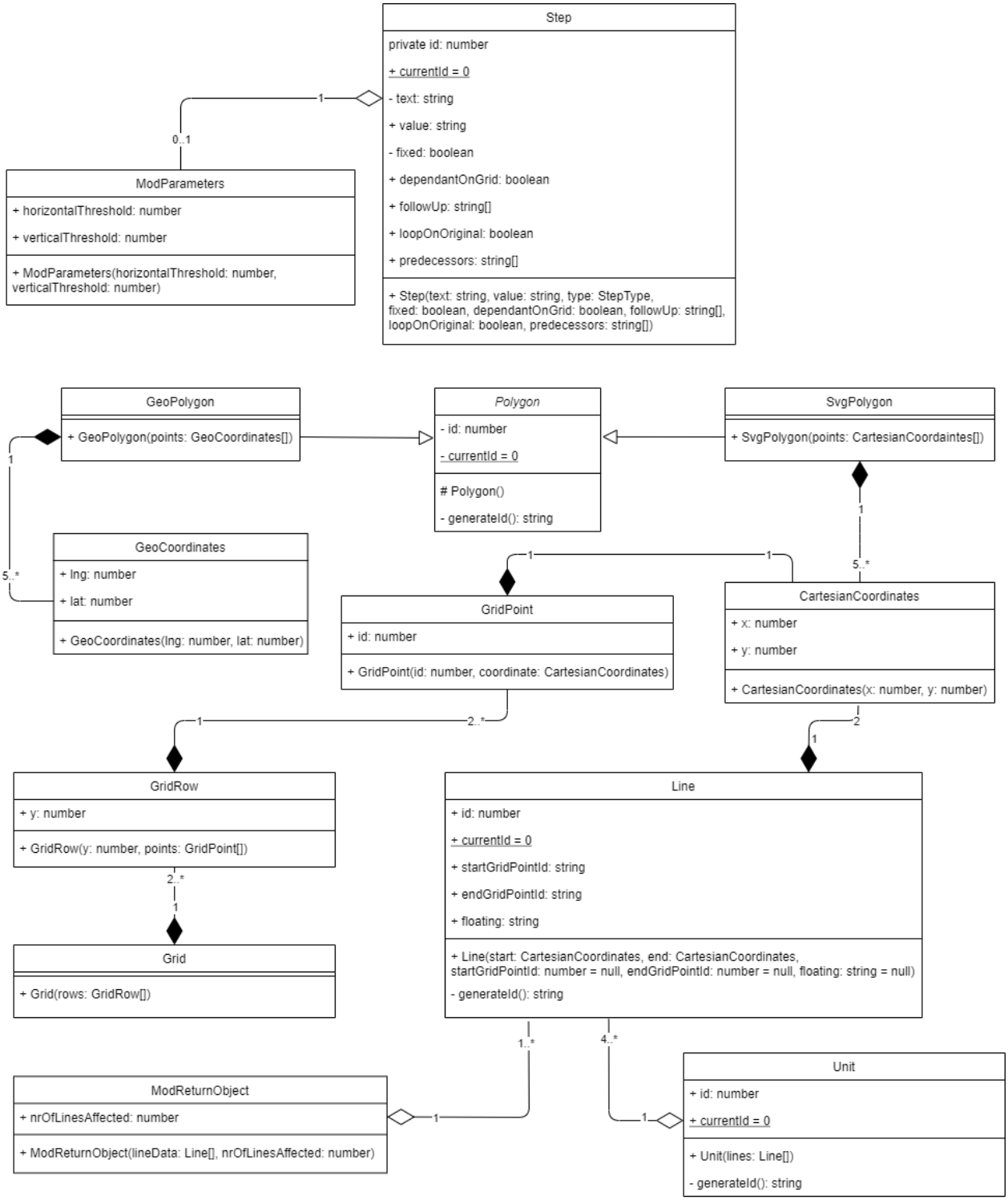
De backend bestaat uit 1 controller en een vijftal endpoints. Hoe deze endpoints zijn opgebouwd is te zien in bijlage 10. Deze bijlage bevat screenshots van Swagger Editor, waar het OpenAPI contract voor deze backend in is opgesteld. Het bijbehorende yaml bestand is terug te vinden in het project.

De backend is stateless. Er wordt in de backend namelijk op geen enkele manier gebruik gemaakt van persistentie of authenticatie en autorisatie. Indien gewenst is dit in de toekomst natuurlijk wel toe te voegen. De backend wordt puur en alleen gebruikt voor het ontvangen van requests, het bewerken van de inkomende data en het sturen van de bewerkte data als response. Security issues zoals SQL injection of cross-site request forgery zijn dan ook niet aan de orde.

CORS is momenteel zo ingesteld dat alleen localhost:8080 (de Vue.js development server) de backend kan benaderen. Wanneer het systeem ergens gehost gaat worden moet deze configuratie aangepast en/of uitgebreid worden. Daarnaast is het in dat geval wellicht verstandig om de backend te beveiligen tegen bijvoorbeeld DoS aanvallen. De backend maakt gebruik van Helmet, een Express library dat de backend (deels) beschermt tegen een aantal ‘standaard’ potentiële security issues door het instellen van bepaalde HTTP headers (NestJS, z.d.).

Alle mogelijke stappen die in een pipeline voor kunnen komen staan gedefinieerd in een config (JSON) bestand. In dit bestand zijn ook afhankelijkheden opgenomen. D.w.z. een stap in de config heeft properties die aangeven welke stappen vooraf moeten gaan aan een stap en welke stappen er uitgevoerd moeten worden na de stap. Onder water bestaan sommige stappen in een pipeline dus uit meerdere stappen/algoritmes. Bijvoorbeeld: de stap die zwevende lijnen aansluit (zie hoofdstuk 4.3.4.2), moet voorafgegaan worden door de algoritmes die de lijnen naar de grid snappen, omdat dit algoritme gebruik maakt van de gridpunten waar lijnen overheen lopen. Daarnaast zijn andere eigenschappen hier configureerbaar, zoals de tekst die de stappen in de frontend krijgen en welke stappen wel en niet aan de pipeline in de frontend toegevoegd mogen worden.

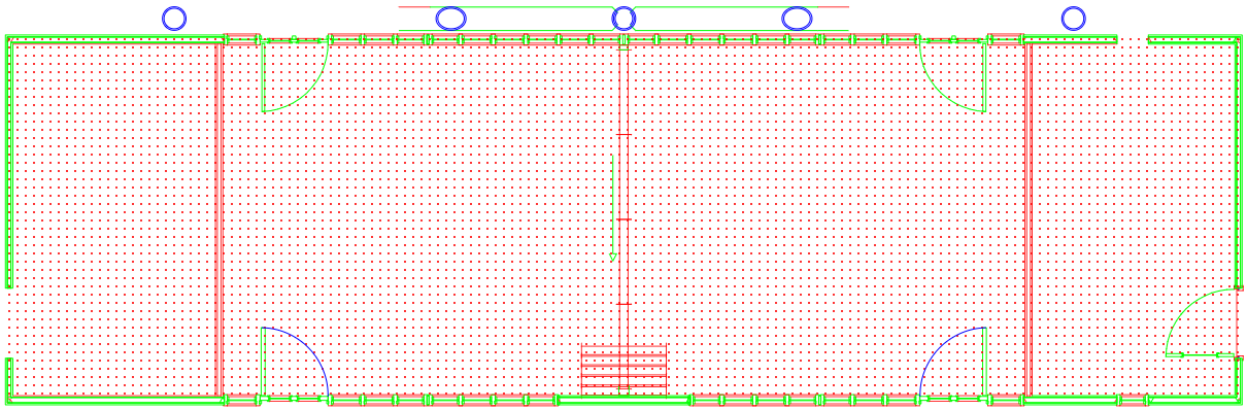
Figuur 10 laat een class diagram zien van alle modelclasses die relaties tot elkaar hebben. Dit diagram bevat om de leesbaarheid te vergroten niet alle classes.



Figuur 10 - Class diagram modelclasses

De S*tep* class representeert een stap in de pipeline en wordt gebruikt voor het aanroepen van de juiste algoritmes (zie hoofdstuk 6.2). Een stap kan een instantie van *ModParameters* hebben wanneer het bijbehorende algoritme verticale- en horizontale drempelwaardes vereist. Als het PoC later uitgebreid wordt om ook met diagonale lijnen te werken, dan kan hiervoor nog een extra variabele aan toegevoegd worden.

De grid, die gebruikt wordt bij het uitvoeren van verschillende optimalisatie algoritmes, is opgebouwd uit 3 classes. De class *Grid* bestaat uit een verzameling rijen, *GridRow*, dat op zijn beurt weer uit een verzameling gridpunten, *GridPoint* bestaat. Gevisualiseerd ziet dat er uit zoals weergegeven in Figuur 11, waarbij alle rode punten samen de grid vormen.



Figuur 11 - Screenshot SVG tekening met grid

Ieder gridpunt heeft een instantie van de class *CartesianCoordinates*. Deze class representeert coördinaten op het SVG canvas met een x- en y waarde. Ook de *Line* en de *SvgPolygon* classes maken hier gebruik van. De *SvgPolygon* is een subclass van de *Polygon* class, omdat er nog een ander type polygon is: de *GeoPolygon*. Het verschil tussen de *SvgPolygon* en *GeoPolygon* classes is dat *SvgPolygon* gebruikt wordt voor het renderen van de SVG en *GeoPolygon* gebruikt wordt bij het georeferencen en het genereren van de IMDF data. *SvgPolygon* maakt gebruik van *CartesianCoordinates* en *GeoPolygon* maakt gebruik van *GeoCoordinates*. De *GeoCoordinates* class bevat geen x- en y-coördinaten, maar lengte- en breedtegraad coördinaten.

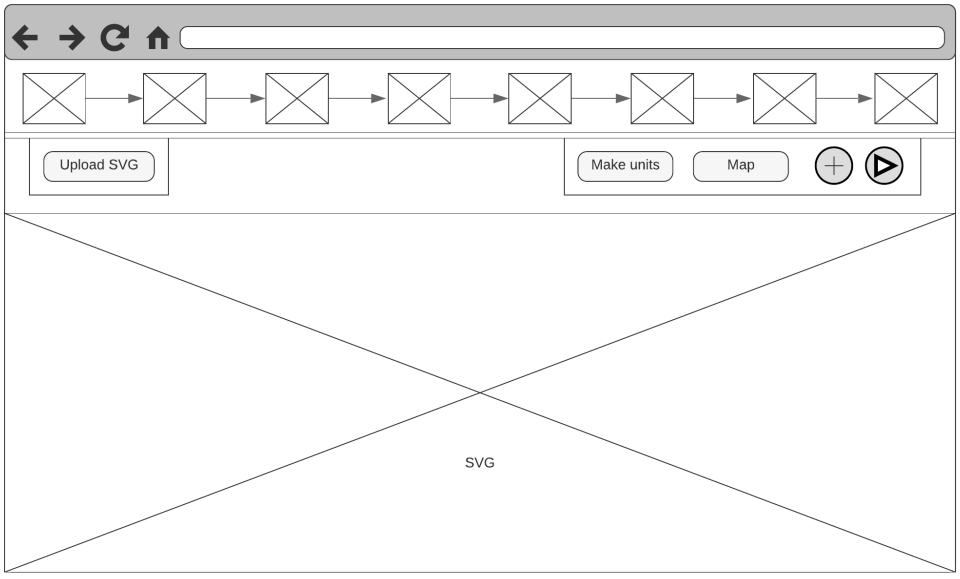
De class *Unit* wordt gebruikt bij het algoritme dat verantwoordelijk is voor de unit herkenning (zie hoofdstuk 4.3.5). Een herkende unit bestaat uit een verzameling instanties van de *Line* class. Nadat alle units zijn herkend worden deze omgezet naar instanties van *SvgPolygon* die op zijn beurt tijdens het proces van georeferencen weer worden omgezet in *GeoPolygon* instanties.

Alle optimalisatie algoritmes geven een *ModReturnObject* terug. *ModReturnObject* bestaat uit de bewerkte lijn data met daarbij nog het aantal lijnen dat is aangepast. Dit is bij het uitvoeren van de pipeline nodig om te weten of een stap nogmaals uitgevoerd moet worden. Zie ook hoofdstuk 6.2.

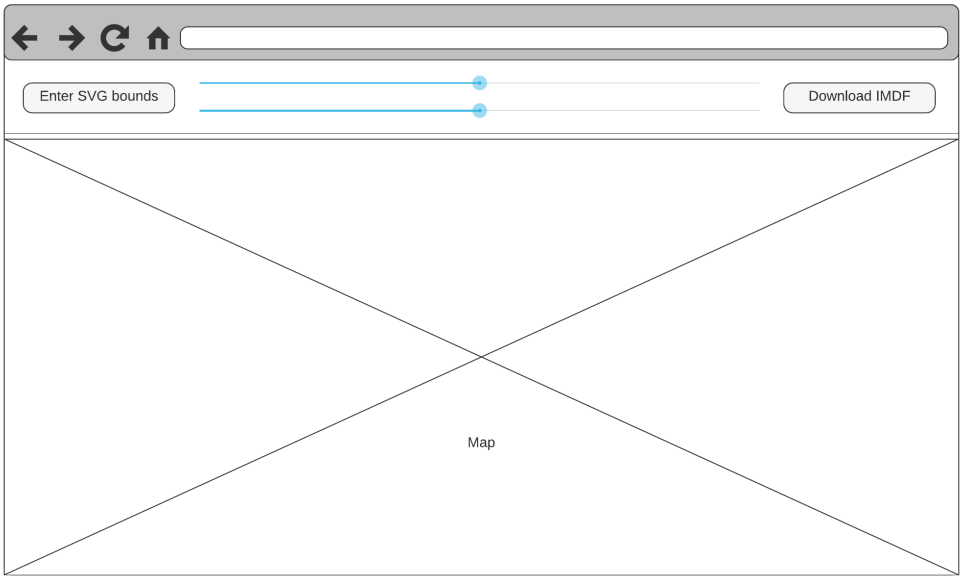
5.4. Frontend

Voor de frontend wordt het Vue.js framework gebruikt. Vue.js is het frontend framework waar ik de meeste ervaring mee heb en dus ook het snelst een kwalitatief goede voorkant van het PoC mee heb kunnen bouwen. Voor de UI is gebruik gemaakt van Vuetify. Vuetify is een UI framework dat gebruikt maakt van de material design specificatie (Google LLC, z.d.) en biedt een verzameling aan UI componenten.

Bij dit project is voor de voorkant de UX belangrijker dan de UI. M.a.w. het is, naast dat de applicatie natuurlijk goed moet werken (hangt nu het meest af van de backend), belangrijk dat de usability van de applicatie goed en intuïtief is. Figuur 12 en Figuur 13 laten de initiële wireframes van de applicatie zien.



Figuur 12 - Wireframe hoofdpagina



Figuur 13 - Wireframe map pagina

Figuur 12 bevat de startpagina van de applicatie dat bestaat uit 3 onderdelen. Bovenin is de pipeline te zien met direct daar onder een aantal actions. De rest van de pagina beslaat de SVG tekening. Figuur 13 is de pagina waar georeferencing plaatsvindt en bestaat uit 2 onderdelen. Bovenin zijn de sliders en buttons voor het renderen en transformeren van de SVG en het downloaden van IMDF te vinden. De rest van de pagina bevat de map.

Na de eerste sprint zijn deze designs herzien en zijn er aanpassingen gedaan om de usability verder te verbeteren. Hiervoor is ook een meeting geweest met iemand uit het design team van Baseflow. Het moest namelijk duidelijker zijn wat de workflow is en daarmee wat er wanneer mogelijk is. Naast wat kleine aanpassingen is de grootste verandering dat er voor een soort 'wizard' aanpak is gekozen. D.w.z. de gebruiker krijgt stap voor stap te zien wat op ieder moment de volgende stap in de workflow is. Hoe de frontend er uiteindelijk uit is komen te zien, is te zien in de screenshots van het PoC in bijlage 11 t/m 16. Om de inhoud van de pagina's beter weer te geven zijn deze screenshots uitvergroot en bevatten dus niet de hele webpagina’s.

Zo begint de applicatie met alleen een veld om een SVG tekening te uploaden. Zie bijlage 11. Nadat een tekening is gekozen, wordt de tekening gerenderd en worden de invoervelden getoond waarmee de grid gemaakt kan worden. Zie bijlage 1. Nadat de grid parameters zijn ingesteld wordt de knop ‘save grid’ klikbaar en kan de grid opgeslagen worden. Hierna komt pas de pipeline tevoorschijn. Zie bijlage 13.

De pipeline bestaat initieel uit een set van vooraf gedefinieerde stappen. Indien gewenst kan dit aangepast worden door stappen te verwijderen en toe te voegen. Het toevoegen van een stap gebeurt door op de ‘+’-knop te klikken. Zie bijlage 14. De ‘play’ knop van de pipeline wordt pas klikbaar zodra alle parameters van de pipeline stappen zijn ingevuld. In de figuur in bijlage 13 is te zien dat dit nog niet het geval is. De stappen in de pipeline die parameters bevatten hebben een blauw edit-icoontje om aan te geven dat er parameters in te stellen zijn. Wanneer een stap parameters bevat die nog niet ingesteld zijn, dan krijgt de stap een rood alert-icoontje. Bijlage 13 laat beide scenario's zien.

De parameters van een stap zijn aan te passen door op de stap te klikken. Zie bijlage 15. Hier kan een stap ook verwijderd worden door op ‘delete step’ te klikken. De stappen die geen parameters bevatten zijn te verwijderen door er met de muis overheen te zweven, waarna een prullenbak icoontje het nummer van de stap vervangt. Als op zo'n stap geklikt wordt, wordt deze verwijderd.

Nadat de pipeline is uitgevoerd door op de play knop te klikken, wordt het mogelijk om de tekening om te zetten naar polygonen. Hiervoor wordt de knop ‘generate polygons’ klikbaar. Daarna wordt het mogelijk om op de ‘georeference’ knop te klikken. De gebruiker wordt dan naar de pagina gebracht waar de georeferencing plaatsvindt. Zie bijlage 16.

Op deze pagina zijn de sliders (voor rotatie en scaling) en de ‘download IMDF’ knop uitgegrijsd totdat de gebruiker gekozen heeft waar de SVG canvas gerenderd moet worden. Nadat de georeferencing is voltooid kan de gebruiker de IMDF data downloaden door op de knop ‘download IMDF data’ te klikken. Na het downloaden is het natuurlijk mogelijk om terug te navigeren naar de pagina met de SVG render en pipeline. Hier kan de gebruiker indien gewenst een nieuwe SVG tekening uploaden en het proces herhalen. Het uploaden van een ander SVG bestand gebeurt door op het paperclip-icoontje links in de pipeline te klikken (zie bijlage 13).

6 Implementatie en realisatie

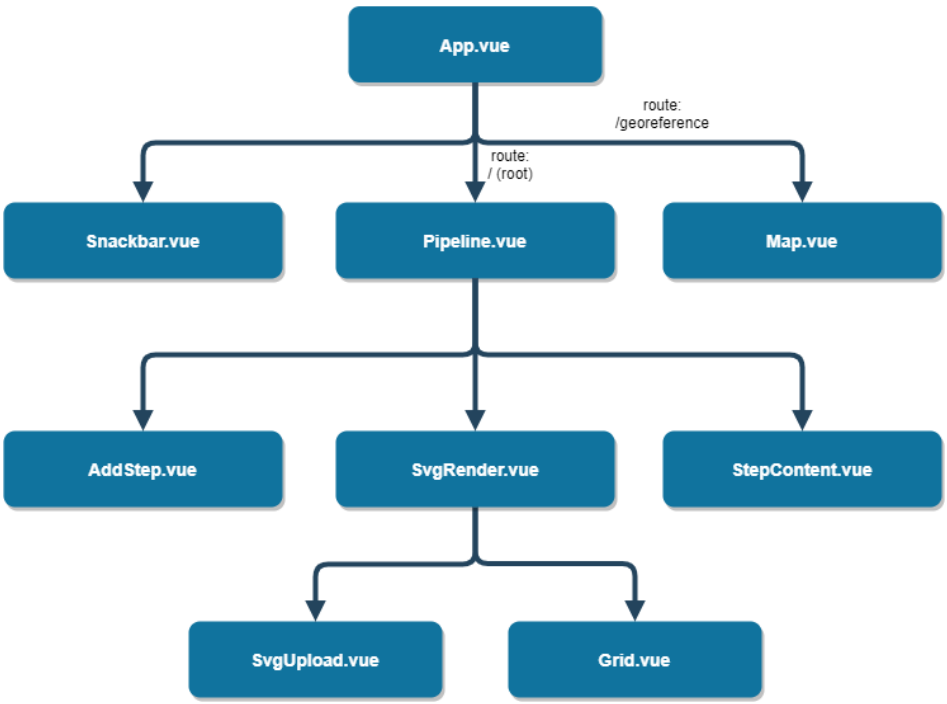
Ook implementatie is bij dit project op verschillende momenten en manieren aan bod gekomen. Tijdens de onderzoeksfase is gaandeweg een groot deel van de algoritmes geschreven die in de uiteindelijke PoC zijn gebruikt. Zie hoofdstuk 4.3.4. Dit hoofdstuk zal ingaan op zowel de implementatie van sommige algoritmes als de implementatie van het PoC.

6.1. Software implementatie keuzes

De backend maakt gebruik van een combinatie van een object georiënteerde programmeerstijl en een functionele programmeerstijl. Om bepaalde types af te dwingen en de code minder foutgevoelig te maken wordt gebruik gemaakt van classes en objecten. Alle algoritmes die verantwoordelijk zijn voor SVG optimalisatie en unit herkenning zijn gegroepeerd in een drietal losse bestanden. Deze algoritmes zijn opgesplitst in functies en werken op een functionele manier. D.w.z. elk van die functies verwacht als input een set geometrische data en heeft als output de bewerkte/omgezette data.

Deze algoritmes zijn in de toekomst als het nodig is dus ook gemakkelijk uit het project te halen om bijvoorbeeld naar een ander project te migreren (eventueel na loskoppelen van expliciete typeringen). Om de koppeling tussen de model classes en algoritmes losjes te houden is ervoor gekozen om geen functies in de classes op te nemen die specifiek zijn voor de classes, maar deze in de algoritmes te laten.

De frontend bestaat uit een aantal ‘single file components’, componenten die bestaan uit een HTML template sectie, een JavaScript script sectie en een (scoped) CSS sectie. Deze componenten zijn verantwoordelijk voor het renderen van de pagina's en algemene gebruikersinteractie hiermee. Figuur 14 laat een overzicht zien van deze componenten en de relaties ertussen. *Pipeline.vue* wordt ingeladen wanneer de applicatie gestart wordt en *Map.vue* wordt ingeladen op route */georeference*. Aangezien het grootste gedeelte van de applicatie een wizard structuur kent (zie hoofdstuk 5.4) worden de meeste componenten (indirect) in *Pipeline.vue* gebruikt. Afhankelijk van de staat van de applicatie worden een of meerdere componenten dan wel of niet getoond. Het component *Snackbar.vue* bevat een snackbar dat wordt gebruikt om op verschillende plekken feedback te geven aan de gebruiker en wordt rechtstreeks in *App.vue* gebruikt zodat deze in de hele applicatie beschikbaar is.



Figuur 14 - Component structuur

Daarnaast zijn er een aantal losse JavaScript bestanden die verantwoordelijk zijn voor het renderen van de SVG en de gebruikersinteractie met de SVG. Dit zijn:

- renderer.js voor het aanmaken en renderen van SVG elementen;
- interaction.js voor het mogelijk maken van gebruikersinteractie met de SVG;
- transform.js voor het transformeren van de SVG (transleren, scalen, roteren).

Ook deze logica en algoritmes zijn dus ook gemakkelijk los te halen van de rest van het project.

Het Vue.js project maakt gebruik van VueX, een store dat het mogelijk maakt om de staat van de applicatie over de verschillende componenten te delen. Hier is onder andere de geometrische data beschikbaar. Dit zorgt ervoor dat de geometrische data op ieder moment overal hetzelfde is en up-to-date is.

6.2. Implementatie pipeline executie

Zoals in hoofdstuk 5.3 genoemd is, staan alle mogelijke stappen van de pipeline gedefinieerd in een config bestand waarbij ook afhankelijkheden zijn opgenomen. Deze stappen worden in hetzelfde formaat meegestuurd wanneer de pipeline uitgevoerd wordt d.m.v. de ‘optimize’ endpoint (zie bijlage 10). Figuur 15 laat de functie zien die in deze endpoint wordt aangeroepen en dus verantwoordelijk is voor het uitvoeren van de pipeline. Deze functie maakt gebruik van twee andere functies: *runStepPredecessors* en *runPipelineStep*. Deze functies worden verderop toegelicht.

Allereerst wordt de grid aangemaakt, omdat deze benodigd is voor een aantal stappen. Vervolgens wordt het config bestand ingelezen, omdat de stappen hierin benodigd zijn voor het uitvoeren van eventuele stappen waar een stap in de pipeline afhankelijk van is (voorafgaande en opvolgende stappen). Daarna wordt er een loop uitgevoerd waarbij over de meegestuurde pipeline stappen geïtereerd wordt. In elke iteratie wordt er eerst een functie aangeroepen, *runStepPredecessors*, die verantwoordelijk is voor het uitvoeren van de stappen die voorafgaan (*step.predecessors*) aan de desbetreffende stap. Vervolgens wordt de daadwerkelijke stap (*step*) van de iteratie uitgevoerd d.m.v. de *runPipelineStep* functie. Dan worden nog van alle opvolgende stappen (*step.followUp*) zowel de voorgaande stappen uitgevoerd als de opvolgende stappen zelf. Als laatst wordt er gekeken of de iteratie opnieuw uitgevoerd moet worden. Dit wordt gedaan als één de twee volgende condities waar is:

- De stap in de huidige iteratie heeft *loopOnOriginal* op true staan en het uitvoeren van deze stap zelf heeft geresulteerd in meer dan 0 bewerkte lijnen;
- De stap in de huidige iteratie heeft *loopOnFollowUp* op true staan en het uitvoeren van de laatste stap in *step.followUp* heeft geresulteerd in meer dan 0 bewerkte lijnen.

```
async runPipeline(
  lineData: Line[],
  gridParams: GridParameters,
  steps: Step[],
): Promise<Line[]> {
  const grid: Grid = createGrid(gridParams);
  const allSteps: Step[] = JSON.parse(
    await this.fs.promises.readFile('src/config/steps.config.json', 'utf-8'),
  );

  for (let i = 0; i < steps.length; i++) {
    const currentStep: Step = steps[i];

    lineData = this.runStepPredecessors(
      currentStep,
      allSteps,
      lineData,
      grid,
    );

    const result: ModReturnObject = this.runPipelineStep(
      lineData,
      grid,
      currentStep,
    );
    lineData = result.lineData;

    let nrOfAffectedLinesFromFollowUp = 0;
    for (const followUpStepValue of currentStep.followUp) {
      const followUpStep: Step = allSteps.find(
        (step: Step) => step.value === followUpStepValue,
      );
      lineData = this.runStepPredecessors(
        followUpStep,
        allSteps,
        lineData,
        grid,
      );
      const resultFromFollowUp: ModReturnObject = this.runPipelineStep(
        lineData,
        grid,
        followUpStep,
      );
      lineData = resultFromFollowUp.lineData;
      nrOfAffectedLinesFromFollowUp = resultFromFollowUp.nrOfAffectedLines;
    }
    if (
      (currentStep.loopOnFollowUp && nrOfAffectedLinesFromFollowUp > 0) ||
      (currentStep.loopOnOriginal && result.nrOfAffectedLines > 0)
    ) {
      i -= 1;
    }
  }
  return lineData;
}
```

Figuur 15 - Code snippet pipeline executie (runPipeline)

Figuur 16 laat de functie *runStepPredecessors* zien. Dit is een recursieve functie die alle voorafgaande stappen (*step.predecessors*) en de daarvan voorafgaande stappen (etc.) uitvoert. Bij elke iteratie wordt dus eerst gekeken of de stap in de iteratie voorafgaande stappen heeft, zo ja, dan roept de functie zichzelf aan. Zodra dat klaar is wordt de stap in de iteratie zelf uitgevoerd (*runPipelineStep*). Mocht er hierdoor een eindeloze loop ontstaan, dan wordt het proces afgebroken en wordt er een error gegooid.

```
private runStepPredecessors(
  step: Step,
  allSteps: Step[],
  lineData: Line[],
  grid: Grid,
): Line[] {
  step.predecessors.forEach((precedingStepValue: string) => {
    const precedingStep = allSteps.find(
      (step: Step) => step.value === precedingStepValue,
    );
    precedingStep.predecessors.forEach(
      (predecessorPrecedingStepValue: string) => {
        if (predecessorPrecedingStepValue === step.value) {
          throw new BadRequestException(
            HttpStatus.INTERNAL_SERVER_ERROR,
            'Circular reference in pipeline detected',
          );
        }
      },
    );
    if (precedingStep.predecessors.length > 0) {
      lineData = this.runStepPredecessors(
        precedingStep,
        allSteps,
        lineData,
        grid,
      );
    }
    lineData = this.runPipelineStep(lineData, grid, precedingStep).lineData;
  });
  return lineData;
}
```

Figuur 16 - Code snippet pipeline executie (runPipelinePredecessors)

De functie *runPipelineStep*, die zowel in *runPipeline* als *runStepPredecessors* wordt aangeroepen, is te zien in Figuur 17. Dit is de meest low-level functie van de drie en is verantwoordelijk voor het daadwerkelijk uitvoeren van een stap door het bijbehorende algoritme aan te roepen. Eerst wordt van een stap het juiste algoritme erbij gepakt. Dit gebeurt door de *step.value* van een stap te gebruiken. Vervolgens wordt er gekeken of het algoritme aangeroepen moet worden met grid als parameter, met *ModParameters* (zie hoofdstuk 5.3) als parameter of zonder parameters. Hierna wordt algoritme daadwerkelijk aangeroepen.

```
runPipelineStep(lineData: Line[], grid: Grid, step: Step): ModReturnObject {
  const functionToExecute = modFunctions[step.value];
  let result: ModReturnObject;
  if (step.dependantOnGrid) {
    result = functionToExecute(lineData, grid);
  } else if (step.parameters) {
    result = functionToExecute(lineData, step.parameters);
  } else {
    result = functionToExecute(lineData);
  }
  return result;
}
```

Figuur 17 - Code snippet pipeline execution (runPipelineStep)

6.3. Opbouw IMDF data

Voor het genereren van de GeoJSON bestanden worden template files gebruikt. Dit zijn JSON bestanden waarin de opbouw van een dergelijk GeoJSON bestand beschreven staat. Deze bestanden worden ingelezen, waarna de benodigde properties gevuld worden met data en worden vervolgens weggeschreven naar de GeoJSON bestanden benodigd voor het archief.

Het IMDF archief bestaat uit 3 bestanden. Dit zijn unit.geojson, venue.geojson en het manifest.json. Het bestand unit.geojson is hier het belangrijkst. Het bevat de coördinaten van alle polygonen die bij de unit herkenning naar boven zijn gekomen. Dit bestand bevat dus vooralsnog alle belangrijke data voor het renderen van een IMDF kaart. Het manifest bevat wat metadata, zoals het moment waarop het archief is aangemaakt en de versie van IMDF waar het archief op is gebaseerd. Verder wordt er nog de venue.geojson aangemaakt. Dit bestand bevat een rechthoek die de geografische grenzen van de hele tekening beschrijft.

Alle coördinaten in het gegenereerde IMDF archief hebben een precisie van zo'n 15 decimalen. Het script dat uitgevoerd wordt op de IMDF data voordat deze in de database wordt gezet (zie hoofdstuk 4.1.1) brengt dit terug naar 7 decimalen.

6.4. Implementatie algoritmes

Het algoritme verantwoordelijk voor de unit herkenning (functie *makeUnits*) is een van de complexere algoritmes en maakt gebruik van twee andere algoritmes: *orientateLines* en *splitLines*. Hoofdstuk 4.3.5 beschrijft in grote lijnen hoe het *makeUnits* algoritme te werk gaat. Hier zal een gedetailleerdere beschrijving van deze 4 algoritmes volgen. Zie bijlage 17 voor een flowchart ter ondersteuning van onderstaande beschrijving.

Function orientateLines
Zodra de *makeUnits* functie aangeroepen wordt, wordt direct de functie *orientateLines* aangeroepen. De functies *makeUnits* verwacht namelijk dat alle lijnen dezelfde oriëntatie hebben. Deze functie kijkt voor elke lijn of het beginpunt van de lijn dichter bij de oorsprong van het canvas ligt dan het eindpunt. Zo niet, dan worden de coördinaten van het begin- en eindpunt omgedraaid/geflipt. Nu hebben alle lijnen dezelfde oriëntatie.

Function splitLines
Deze functie wordt direct na *orientateLines* aangeroepen en zorgt ervoor dat alle lijnen worden opgesplitst op de punten waar de lijnen kruisigingen vormen met andere lijnen. Dit is nodig, omdat het algoritme voor unit herkenning als het ware lijnen in een bepaalde richting gaat "tracken". Als de lijnen niet opgesplitst worden, zullen niet alle units herkend worden.

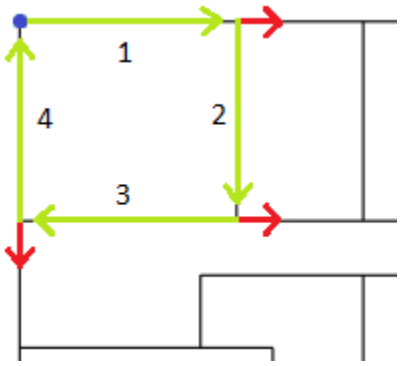
Voor alle lijnen wordt allereerst de functie *findGridPointsInCourseOfLine* aangeroepen. Om het diagram en deze beschrijving leesbaar te houden is deze functie een black box. Deze functie geeft van elke lijn alle gridpunten terug die in het verloop van de lijn liggen, inclusief de gridpunten van het start- en eindpunt van de lijn. Deze data (elke lijn met zijn gridpunten) wordt in een variabele opgeslagen om later te vergelijken met andere lijnen.

Vervolgens begint de iteratie over alle lijnen (zie bijlage 17, *orientateLines* box - For every line). Van de huidige lijn in de iteratie wordt wederom de functie *findGridPointsInCourseOfLine* aangeroepen. Ditmaal geeft de functie echter alle gridpunten in het verloop van de lijn terug, exclusief de gridpunten van het start- en eindpunt van de lijn. Een lijn hoeft immers nooit bij het begin- of eindpunt opgesplitst te worden.

Nu wordt elke gridpunt van de huidige lijn vergeleken met alle gridpunten van alle andere lijnen. Zodra er een match is, is er een kruising tussen de twee lijnen die vergeleken worden en worden de coördinaten van de kruising in een variabele opgeslagen. Zodra van de huidige lijn alle vergelijkingen met andere lijnen klaar zijn, is het tijd om de lijn aan de hand van deze coördinaten op te splitsen in meerdere lijnen. Dit proces herhaalt zich vervolgens voor alle lijnen die er zijn.

makeUnits

Nu alle lijnen georiënteerd en opgesplitst zijn, is het tijd om de unit herkenning uit te voeren. Om alle mogelijke units te achterhalen wordt voor elke lijn de unit herkenning uitgevoerd. Alle logica in de functie wordt dus iteratief uitgevoerd (zie bijlage 17, *makeUnits* box - For every line). Voor elke iteratie wordt er een nieuwe unit aangemaakt, waarvan de huidige lijn in de iteratie het begin van de unit is.
Zie Figuur 18 ter illustratie, waar lijn 1 de huidige lijn is. Ook wordt het eindpunt van de flow (*endPointInFlow*) steeds bijgehouden. Dit is het punt waar het tracken van de lijnen in een unit steeds geëindigd is, en kan dus zowel het begin- als eindpunt van een lijn zijn. Bij de eerste lijn in een nieuwe unit is dit altijd het eindpunt van de lijn, omdat het tracken altijd bij het beginpunt (zie blauwe punt Figuur 18) van de lijn begint.



Figuur 18 - Illustratie unit herkenning

De eerste stap is het vinden van de eerste lijn die haaks op de lijn staat die bekeken wordt. Dit is in het geval van de illustratie lijn 2. Nu wordt ook de richting in de flow bepaald die de startlijn en eerste haakse lijn ten opzichte van elkaar hebben. Hiervoor wordt de functie *findDirectionInFlow* aangeroepen, wat wederom een black box is. In dit geval is dit de richting *clockWise*. De eerste haakse lijn, lijn 2, is nu de nieuwe huidige lijn. Vervolgens wordt er zolang de unit nog niet afgesloten is (zie bijlage 17, *makeUnits* box – While unit not enclosed) steeds een lijn gezocht dat voldoet aan de volgende voorwaarden:

- De lijn is aangesloten op het eindpunt in de flow;
- De lijn volgt de vastgestelde richting: met de klok mee of tegen de klok in.

In de illustratie zullen dus vervolgens lijn 3 en lijn 4 getrackt worden. Als er maar één lijn is aangesloten op het eindpunt in de flow dan gelden bovenstaande voorwaarden niet en wordt de aangesloten lijn per definitie de nieuwe huidige lijn in de flow.

Zodra het eindpunt in de flow van de laatst gevonden lijn aansluit op het beginpunt van de eerste lijn in de unit, dan is de unit volledig afgesloten en dus voltooid. Nu moet er alleen nog gekeken worden of de gevonden unit al eerder gevonden is. Als dit niet het geval is dan wordt de unit toegevoegd aan de lijst met gevonden units. Dit proces herhaalt zich vervolgens voor alle lijnen die er zijn. Aan het eind van de iteraties zijn alle units gevonden.

7 Testen

Het belangrijkste gedeelte van het PoC zijn de algoritmes voor SVG optimalisaties en unit herkenning. Om ervoor te zorgen dat deze algoritmes doen (en blijven doen) wat er van ze verwacht wordt, zijn er voor deze algoritmes geïsoleerde unit tests geschreven. Ook voor de helper functies zijn unit tests geschreven. Deze unit tests zijn geschreven met het Jest test framework. Het doel van deze tests is het bevestigen dat de output van de functies is wat er verwacht wordt bij het meegeven van specifieke input. Daarnaast is er beoogd een zo hoog mogelijke line coverage te halen.

De tests die algoritmes testen waarbij geometrische data gemanipuleerd wordt, beginnen steeds met het inlezen van een SVG bestand. Deze SVG bestanden bevatten mock data waar in de tests de algoritmes op uitgevoerd worden. Voor elk van deze tests is een apart SVG bestand aangemaakt. Op deze manier is te voorkomen dat bij het toevoegen en aanpassen van tests in de toekomst bepaalde tests onverwachts falen. Daarnaast bevatten deze bestanden mock data dat met de hand is geschreven, i.t.t. ‘kant- en klare’ SVG tekeningen. Hierdoor zijn de bestanden klein, overzichtelijk en zijn ze makkelijk aan te passen. Daarnaast is hierdoor beter te voorspellen wat de uitkomst van de tests zou moeten zijn (i.t.t. wanneer kant- en klare SVG bestanden met honderden of duizenden lijnen worden gebruikt) en zijn tests makkelijker te debuggen.

Naast bovengenoemde unit tests zijn er ook 2 unit integration tests geschreven. Deze testen respectievelijk het uitvoeren van de gehele pipeline en het herkennen van units en genereren van polygonen. Deze tests maken gebruik van een kant- en klare SVG tekening met meer dan 2000 lijnen.

In totaal zijn er in 3 test suites 32 test cases geschreven. Hierbij wordt gebruik gemaakt van 12 bestanden die mock data bevatten. De uiteindelijke line coverage van alle bovengenoemde algoritmes en helper functies is 100%. Zie bijlage 18 voor het test rapport, waarbij de bestanden *mods.ts*, *geometry.helper.ts* en *grid.helper.ts* de optimalisatie algoritmes en helper functies bevatten.

In plaats van een acceptatietest aan het einde van de afstudeerperiode is op verschillende momenten gedurende de implementatiefase gepeild of de opdrachtgever tevreden was met het (tussen)resultaat. Dit is voornamelijk gedaan tijdens de scrum meetings met de opdrachtgever, waarin de voortgang en prioriteiten steeds besproken zijn. De requirements (zie hoofdstuk 5) zijn behaald en de opdrachtgever is blij met het uiteindelijk opgeleverde PoC.

8 Conclusie

De hoofdvraag van dit project luidt: "Hoe kan Baseflow d.m.v. automatisering op een zo efficiënt mogelijke manier IMDF data genereren voor het renderen van indoor kaarten voor gebruik in haar applicaties?". Om hier antwoord op te geven zal eerst antwoord worden gegeven op de deelvragen (m.u.v. de beschrijvende deelvragen).

Er zijn geen specifieke IMDF features die per se van belang zijn voor het renderen van indoor kaarten. Zolang de gebruikte features de juiste geografische coördinaten bevatten kan het namelijk gebruikt worden om een kaart mee te renderen. Wel schrijft het IMDF formaat voor welke features (en onderdelen van features) verplicht zijn om aan het IMDF formaat te voldoen. Bij Baseflow worden er van de 16 mogelijke features 4 gebruikt: Building, Level, Unit en Fixture.

Het type kaarten die worden gebruikt voor het genereren van IMDF zijn CAD en PDF. Beide formaten zijn (indirect) om te zetten naar SVG. De SVG kan vervolgens gebruikt worden om geometrische data uit te halen. Deze data bevat veel overbodige geometrie en moet eerst geoptimaliseerd worden voordat het omgezet kan worden naar IMDF.

De verkregen geometrische data moet ook omgezet worden naar lengte- en breedtegraad coördinaten. Dit kan gedaan worden door de SVG tekening over een wereldkaart te renderen en zodanig te transformeren dat het op de juiste plek op de kaart komt te liggen. Het resultaat hiervan is uiterst nauwkeurig. Vervolgens kan de data naar GeoJSON bestanden worden geschreven waarna het gezipt en gedownload kan worden.

IMDF data kan dus deels geautomatiseerd gegenereerd worden door het gebruik van een set aan algoritmes die bewerkingen uitvoeren op de vectordata, mits het SVG data als invoer heeft. Om deze data verder te verfijnen en/of om of de data semantisch te verrijken (toevoegen properties) is er altijd nog extra handmatig werk nodig. Dit extra handmatige werk zal in de meeste gevallen echter aanzienlijk minder tijd kosten dan dat het normaal kost om IMDF data door de derde partij aangeleverd te krijgen.

Het uiteindelijke PoC laat zien dat het inderdaad mogelijk is om IMDF data grotendeels geautomatiseerd te laten genereren. De klant is tevreden met dit eindresultaat en heeft ook aangegeven dat er potentie in zit om het project in de toekomst verder op te pakken. Ook wordt niet uitgesloten dat, wanneer het project verder is doorontwikkeld, het project in de toekomst wellicht open-source aangeboden kan worden.

9 Discussie en aanbevelingen

Er zijn een aantal dingen die geïmplementeerd kunnen worden om het PoC verder uit te breiden. Zo werkt het PoC momenteel alleen met lines en polylines. Sommige SVG kaarten bevatten echter nog andere cruciale SVG geometrie, zoals paths. Het PoC zou dan ook uitgebreid kunnen worden zodat het ook met deze andere geometrie om kan gaan. Daarnaast zijn de algoritmes in het PoC gericht op het bewerken van horizontale en verticale lijnen. Ook dit kan uitgebreid worden zodat het ook met diagonale lijnen om kan gaan. Ook is het een goed idee om bij de georeferencing alle geometrie tijdelijk om te zetten naar een enkel SVG path alvorens alle transformaties toe te passen. Dit zal ervoor zorgen dat het transformeren soepeler en sneller gaat, omdat hiermee minder geometrie aangepast hoeft te worden (zie document Onderzoeksrapport hoofdstuk 1.6.2).

Daarnaast werkt het PoC nu nog met enkele SVG bestanden. M.a.w. de applicatie moet per SVG bestand, en dus per verdieping van een gebouw doorlopen worden. Aangezien in de meeste gevallen verdiepingen van een gebouw recht boven elkaar liggen is het ook mogelijk om het PoC zodanig uit te breiden dat meerdere SVG tekeningen tegelijk (in één sessie) geüpload en bewerkt kunnen worden waarna ook maar eenmalig georeferencing hoeft plaats te vinden. Dit zorgt niet alleen voor minder handmatig werk, maar is ook gewenst omdat meerdere verdiepingen in hetzelfde archief horen. Ook is het wenselijk om hiermee de Level features voor het archief te laten genereren.

Om verdere optimalisatie van tekeningen mogelijk te maken zouden functionaliteiten ingebouwd kunnen worden die het mogelijk maken om de SVG handmatig verder te bewerken, zoals het toevoegen/intekenen van geometrie. Ook is het verstandig om bewerkte SVG bestanden tussentijds op te kunnen slaan, zij het aan de achterkant, zij het aan de voorkant door bijvoorbeeld de bewerkte SVG te kunnen downloaden. Mochten er om wat voor reden dan ook problemen ontstaan, bijvoorbeeld doordat de API verantwoordelijk voor het leveren van de tiles in Leaflet niet reageert, dan is het altijd mogelijk om een eerdere state van de tekening terug te halen. Als laatst is het een goed idee om in de backend logging in te bouwen om bij te houden welke algoritmes er in welke volgorde worden uitgevoerd en om bijvoorbeeld terug te kunnen zien of er eventuele problemen of onregelmatigheden zijn opgetreden.

Voor een eventueel vervolgonderzoek is het interessant om te kijken wat er mogelijk is m.b.t. het optimaliseren van geometrie door het gebruik van machine learning/AI.

10 Reflectie

Bijna de gehele afstudeerperiode is vanwege overheidsmaatregelen m.b.t. de corona pandemie vanuit huis uitgevoerd. Afstuderen is per definitie natuurlijk al een solistische aangelegenheid, maar het thuis uitvoeren ervan maakte dat het afstuderen nog individualistischer was. Dit heeft in mijn geval zijn voor- en nadelen gehad. Het grootste voordeel was dat ik mij over langere periodes beter heb kunnen concentreren. Het grootste nadeel was echter dat het thuiswerken mijn creativiteit niet ten goede is gekomen.

Vooral tijdens meetings (zowel met de opdrachtgever als met het team) is het thuis uitvoeren van het afstudeerproject niet ideaal geweest. Fysiek overleg voeren m.b.v. een whiteboard e.d. is namelijk gemakkelijker dan een video conference bij te wonen. Daarnaast is er op kantoor meer ruimte voor bijvoorbeeld onverwachte brainstorm sessies of spontane momenten waarop je met collega's over allerlei zaken kunt sparren. Deze tekortkomingen zijn echter weer deels gecompenseerd door de mogelijkheid om zelf te kunnen kiezen wanneer je werkt i.p.v. gebonden te zijn aan vaste kantoortijden.

Een van de potentiële 'struikelblokken' van dit project was het gebrek aan beschikbaar werk en informatie m.b.t. het onderwerp. Bij het doen van onderzoek ben ik gewend dat er relatief veel informatie over een bepaald onderwerp beschikbaar is zodat daar op verder geborduurd kan worden. Dit was tijdens de onderzoeksfase soms even wennen. Desalniettemin heeft dit risico geen negatieve impact gehad op het project en de resultaten daarvan.

Tijdens de implementatiefase is er elke sprint een vast moment geweest waarop een meeting met de opdrachtgever gepland stond. Tijdens de onderzoeksfase was hier echter geen vast moment voor ingepland, maar zijn de meetings steeds ingepland wanneer dat nodig was. Dit zorgde er echter voor dat het soms even duurde voordat de meeting plaats kon vinden. Het was beter geweest om ook tijdens deze fase, naast de spontane meetings, een aantal vaste momenten in te plannen om de voortgang, eventuele tegenslagen en prioriteiten te bespreken.

Tijdens de gehele afstudeerperiode zijn er een aantal meetings met het team ingepland voor het geven van demo's. Deze momenten zijn erg waardevol geweest voor het verdere verloop van het project, omdat deze meetings ook zijn gebruikt voor het uitwisselen van ideeën en feedback.

De HBO-i competenties analyseren, ontwerpen en realiseren (HBO-i stichting, 2021) zijn met dit afstudeerproject op niveau 3 aangetoond d.m.v. de drie verschillende fases van het project: onderzoek (zie hoofdstuk 4), ontwerp (zie hoofdstuk 5) en implementatie (zie hoofdstuk 6).

Literatuurlijst

Internet Engineering Task Force (IETF). (2016, augustus). RFC 7946 - The GeoJSON Format. IETF Tools. https://tools.ietf.org/html/rfc7946

Cohen, S. (2018, mei 3). Data Extraction: Exploring the Features and Benefits of AutoCAD. Autodesk Blogs. https://blogs.autodesk.com/autocad/data-extraction-exploring-features-benefits-autocad/

Autodesk. (2020, augustus 13). LIST (Command). Autodesk Knowledge Network. https://knowledge.autodesk.com/support/autocad/learn-explore/caas/CloudHelp/cloudhelp/2021/ENU/AutoCAD-Core/files/GUID-88FFCF22-5F25-48D9-BD43-4F248EFFCE17-htm.html#:%7E:text=You%20can%20use%20LIST%20to,model%20space%20or%20paper%20space.

GIS Geography. (2021, januari 3). What is Georeferencing? How to Georeference Anything. https://gisgeography.com/georeferencing/

Agafonkin, V. (z.d.). Leaflet - a JavaScript library for interactive maps. Leaflet. https://leafletjs.com/reference-1.7.1.html#svgoverlay

PROJ — PROJ 8.0.0 documentation. (2021, maart 29). PROJ. https://proj.org/

Esri. (2018, november 12). World UTM Grid. ArcGIS Hub. https://hub.arcgis.com/datasets/esri::world-utm-grid

Turnhout, K., Craenmehr, S., Holwerda, R., Menijn, M., Bakker, R., & J.P.Z. (2013, april). Triangulatie: een basis voor de onderzoeksleerlijn in ICT- en mediaonderwijs. ResearchGate. https://www.researchgate.net/publication/272348221_Triangulatie_een_basis_voor_de_onderzo eksleerlijn_in_ICT-_en_mediaonderwijs

Adobe. (2021). Adobe Illustrator. https://www.adobe.com/nl/products/illustrator.html?sdid=88X75SL2&mv=search&ef_id=Cj0KCQiA7NKBBhDBARIsAHbXCB6Oud7L6DW-iDLdVGmjFaB-MQWI1aCctMcwhcB_FF_JVPqIOUIW-2QaApZ-EALw_wcB:G:s&s_kwcid=AL!3085!3!341238701945!e!!g!!adobe%20illustrator!1479062

547!59972730129&gclid=Cj0KCQiA7NKBBhDBARIsAHbXCB6Oud7L6DW-iDLdVGmjFaB-MQWI1aCctMcwhcB_FF_JVPqIOUIW-2QaApZ-EALw_wcB#

MDN Contributors. (2021, februari 19). SVG: Scalable Vector Graphics | MDN. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/SVG/Element/svg

MDN Contributors. (2020, december 21). SVGPathElement - Web APIs | MDN. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/SVGPathElement

Autodesk. (2021b, maart 23). AutoCAD Software. https://www.autodesk.com/products/autocad/overview?term=1-YEAR

Mapxus. (z.d.). Mapxus: IMDF. https://www.mapxus.com/imdf

Mappedin. (z.d.). Indoor Mapping Data Format (IMDF). https://www.mappedin.com/mapping/imdf-export/

Safe Software Inc. (z.d.). FME | Data Integration Platform. Safe Software. https://www.safe.com/fme/

Autodesk. (2021, mei 6). How to convert a PDF to a DWG in AutoCAD. AutoCAD | Autodesk Knowledge Network. https://knowledge.autodesk.com/support/autocad/learn-explore/caas/sfdcarticles/sfdcarticles/How-to-convert-a-PDF-to-a-DWG-in-AutoCAD.html

Esri. (z.d.). About georeferencing CAD datasets—ArcMap | Documentation. ArcGIS Desktop. https://desktop.arcgis.com/en/arcmap/latest/manage-data/cad/about-georeferencing-cad-datasets.htm

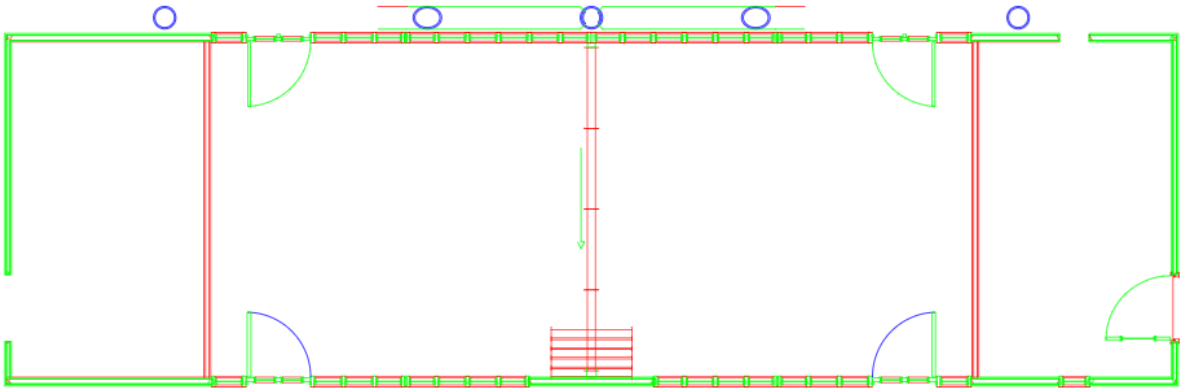
NestJS. (z.d.). Helmet. https://docs.nestjs.com/security/helmet

HBO-i stichting. (2021, juni 10). Domeinbeschrijving. HBO-i. https://www.hbo-i.nl/publicaties-domeinbeschrijving/

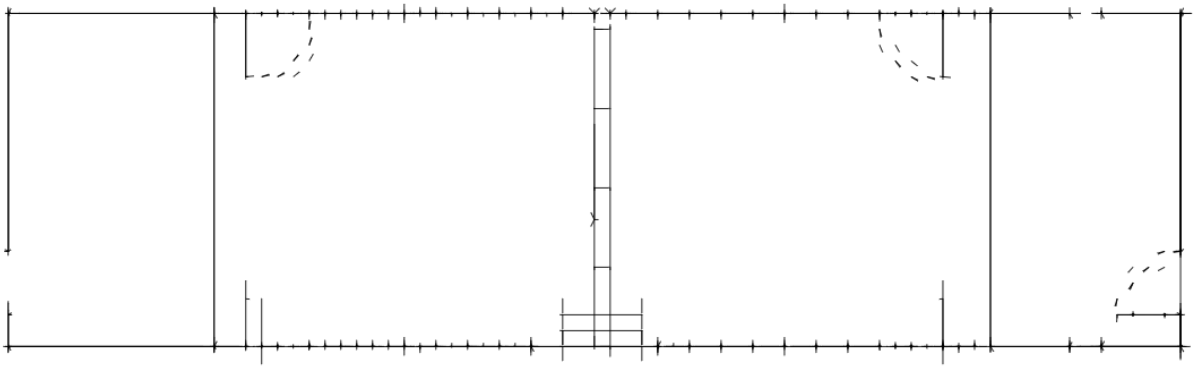
Google LLC. (z.d.). Material Design. Material. https://material.io/design/introduction

Bijlagen

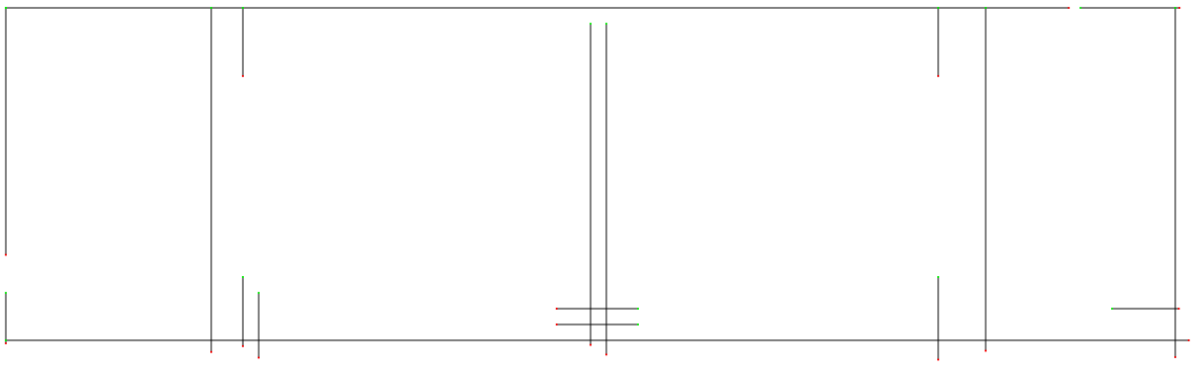
1 SVG tekening voor uitvoeren algoritmes



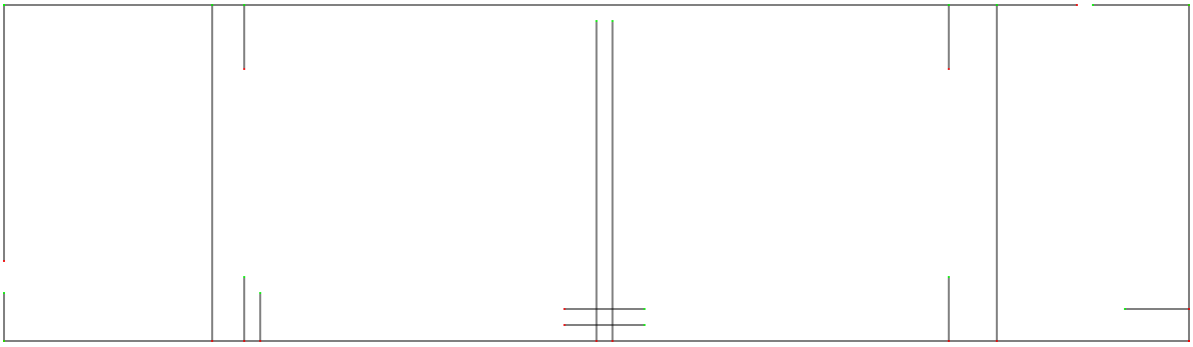
2 Snap to grid



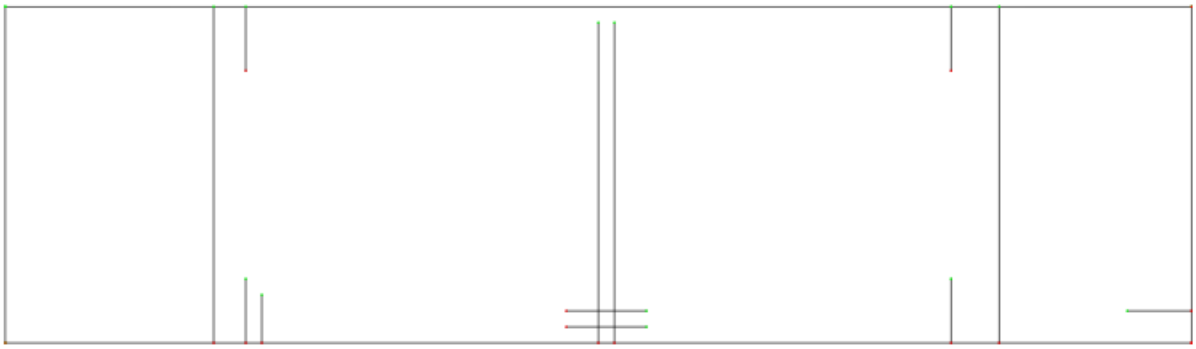
3 Filteren overlappende en korte lijnen



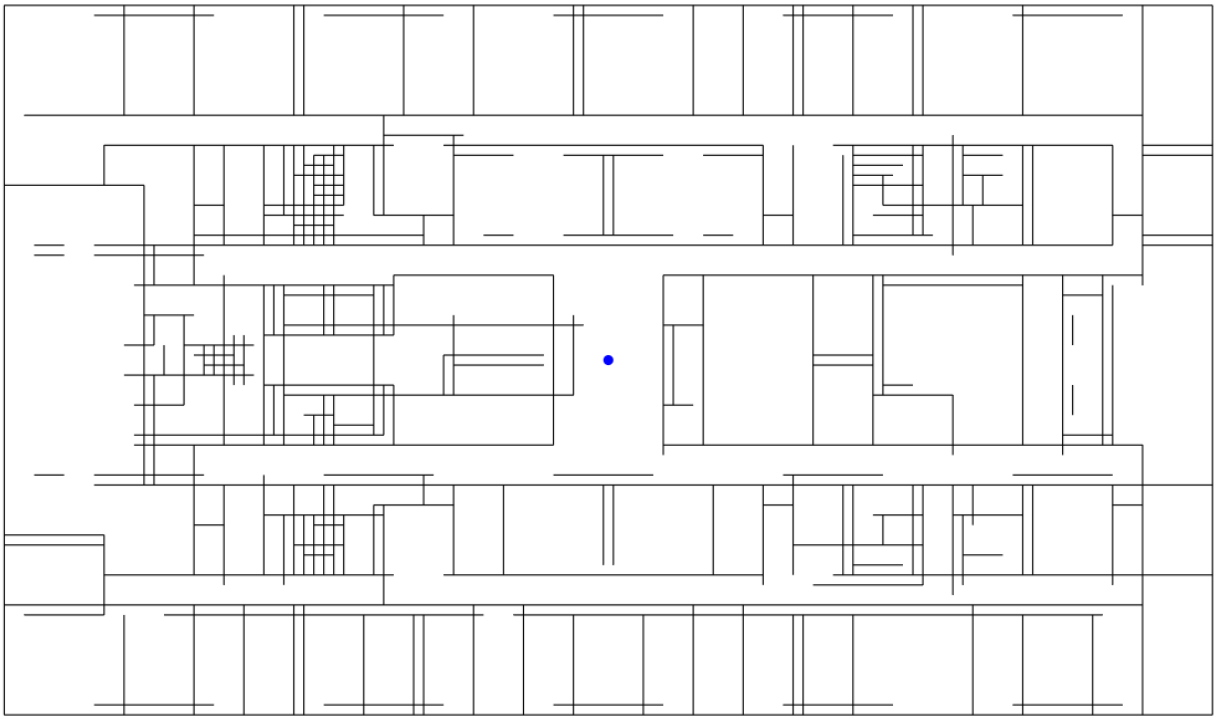
4 Snap end to grid



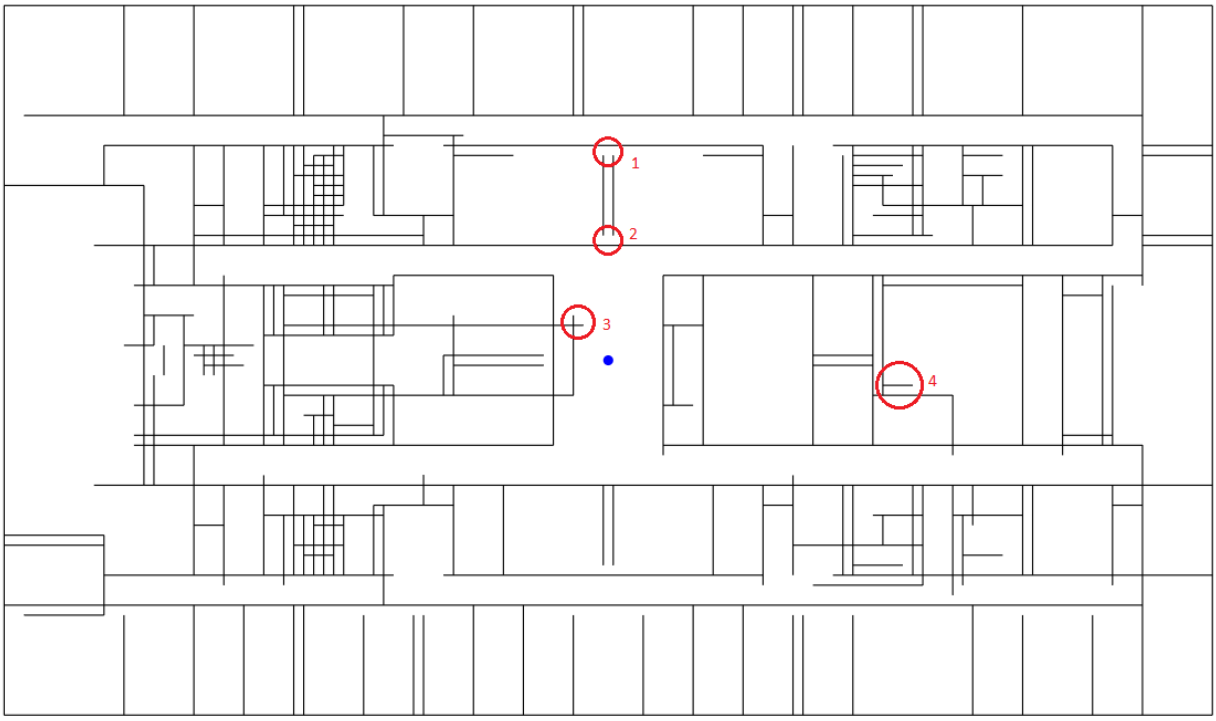
5 Opvullen gaten



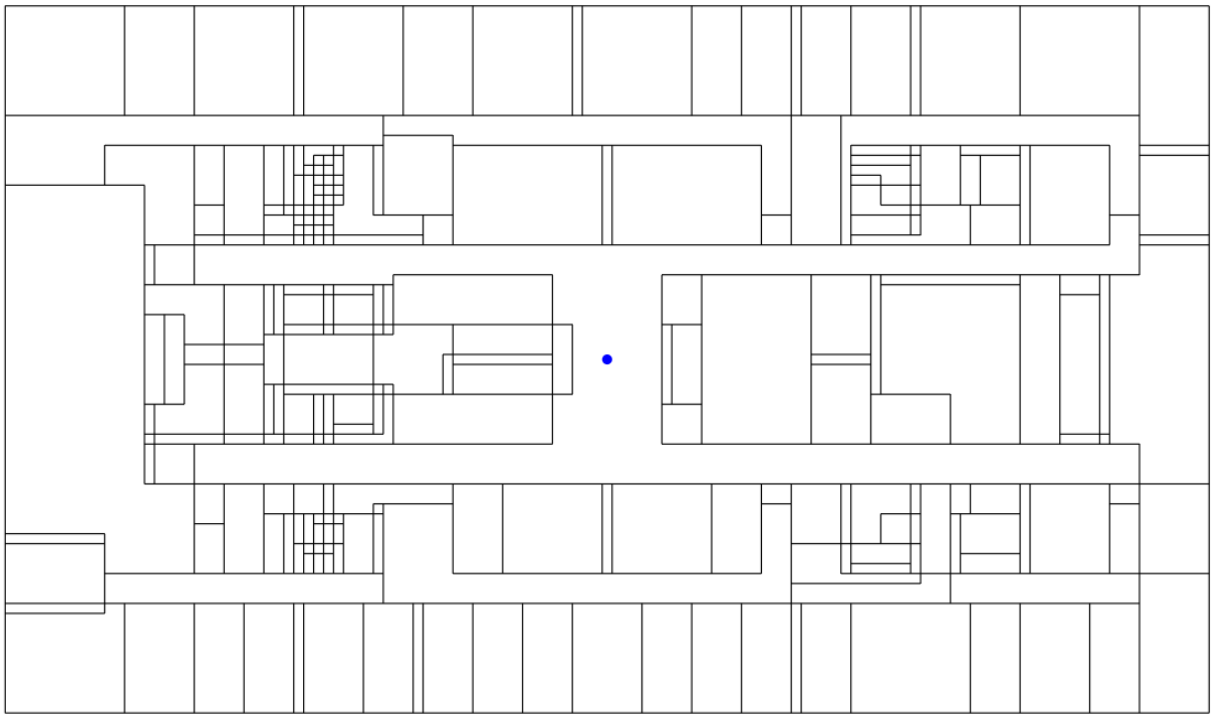
6 Tussenresultaat complexere tekening



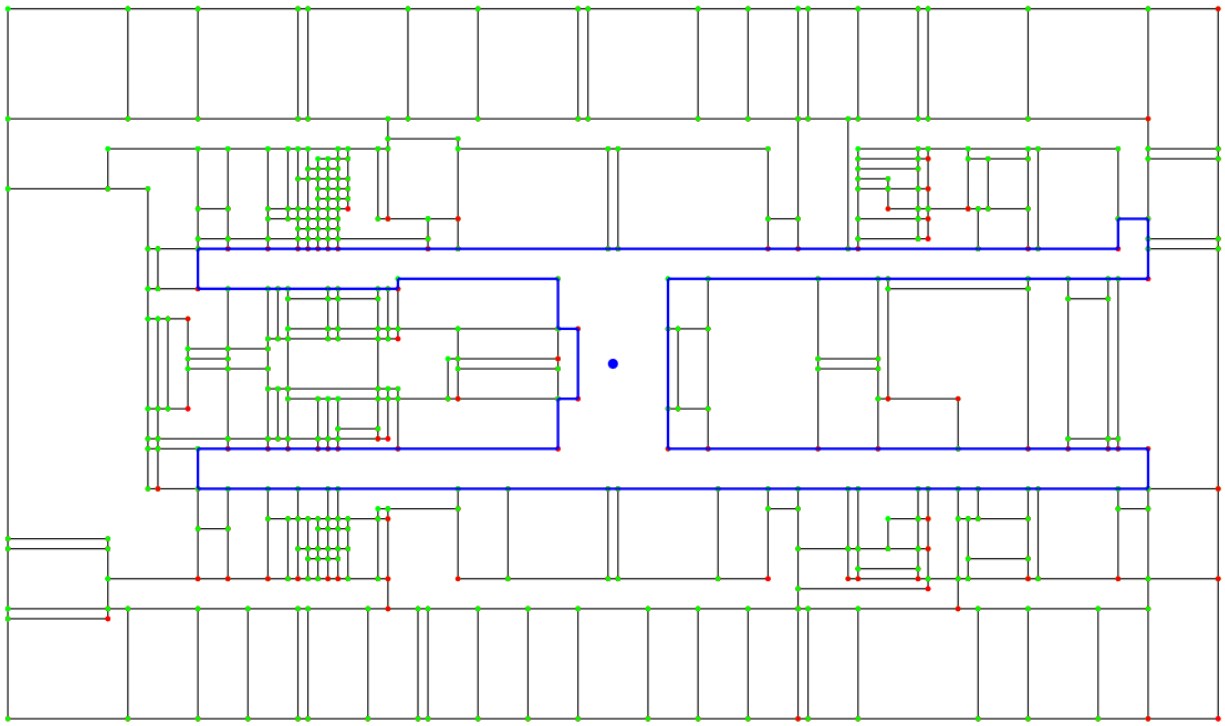
7 Filteren zwevende lijnen



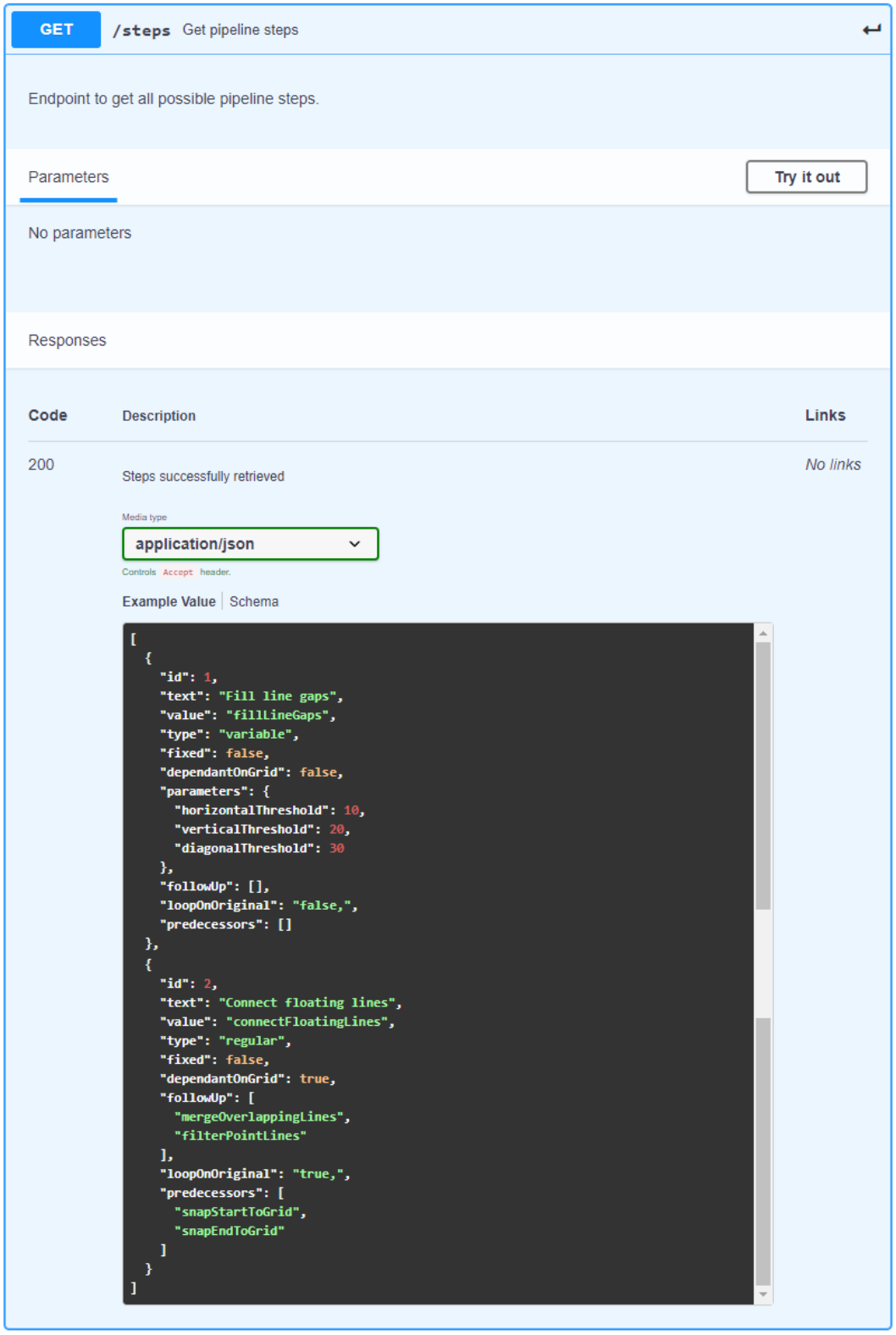
8 Aansluiten (half)-zwevende lijnen



9 Unit herkenning



10 API endpoints (screenshots Swagger UI)



POST `/upload` Upload SVG file

Endpoint to upload SVG file. Endpoint reads file, parses the SVG content, filters duplicates and point lines and returns parsed line data.

Parameters | Try it out

No parameters

Request body required — multipart/form-data

SVG file that needs to be read and parsed

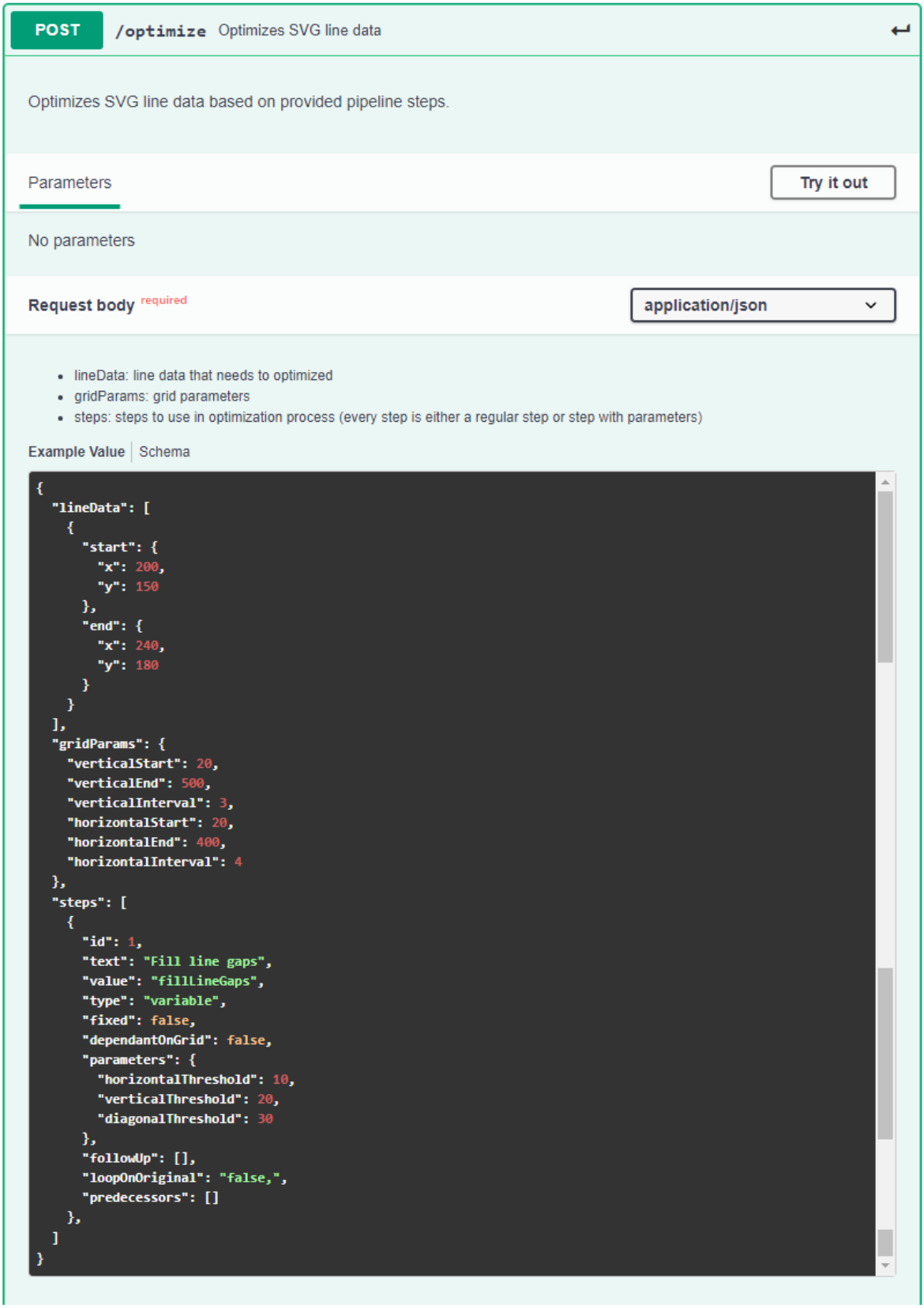
file

`string($binary)`

Responses

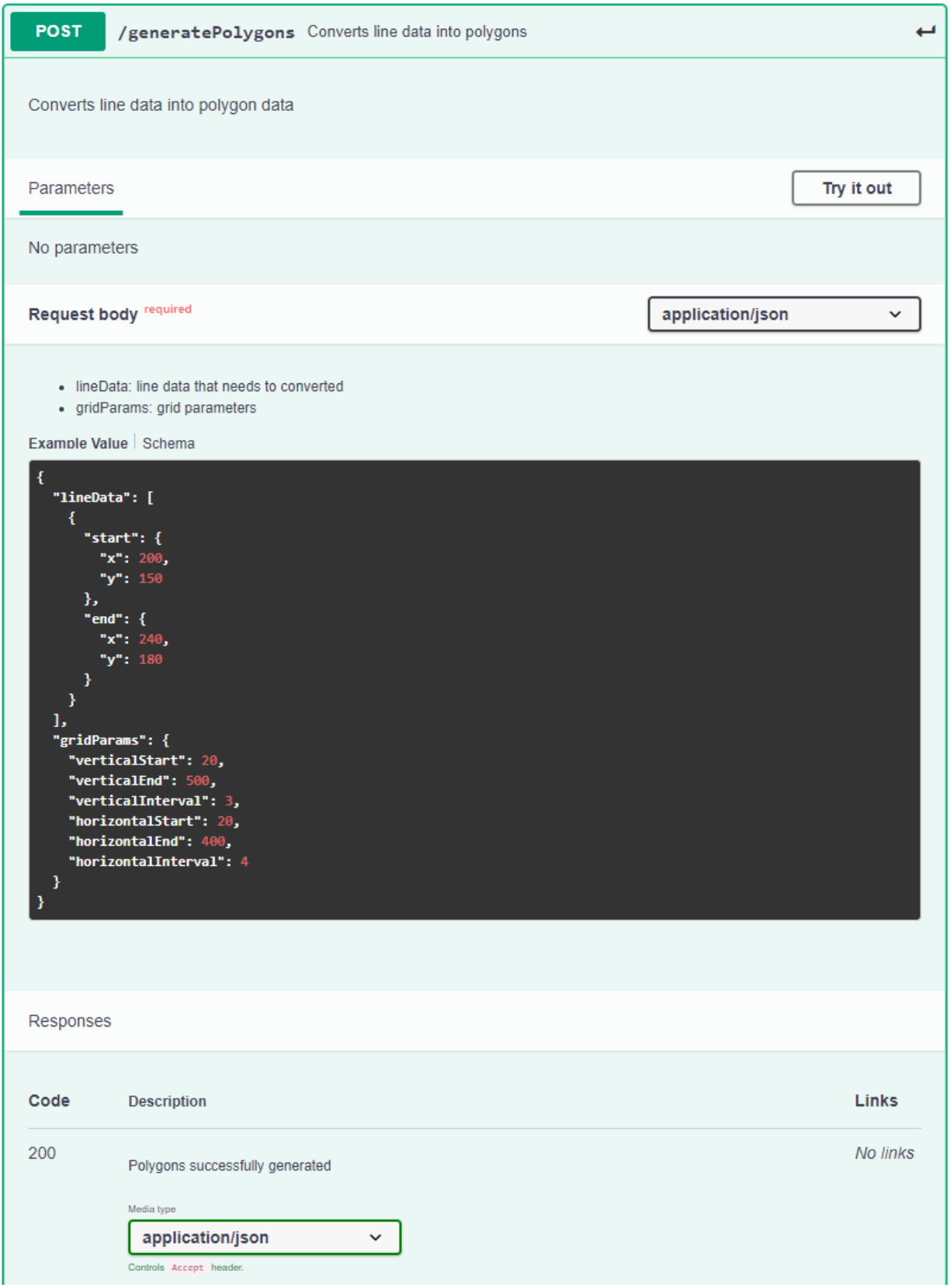
Code	Description	Links
200	File successfully uploaded. Response contains parsed line data and viewbox of the SVG drawing Media type: application/json Controls Accept header. Example Value | Schema	*No links*
400	SVG file is missing	*No links*
422	File doesn't contain group tag	*No links*

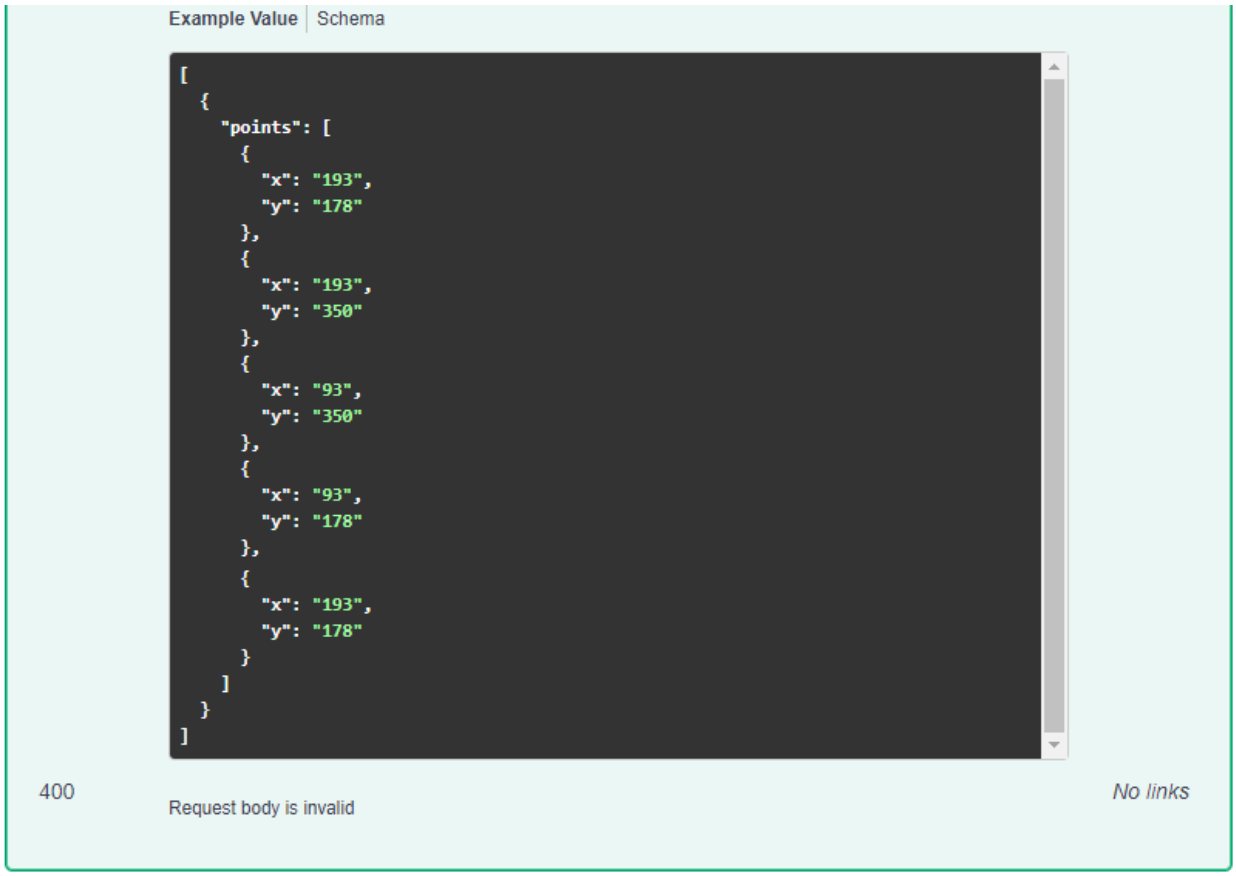
```
{
  "lineData": [
    {
      "start": {
        "x": 200,
        "y": 150
      },
      "end": {
        "x": 240,
        "y": 180
      }
    }
  ],
  "viewBox": {
    "viewBox": "0 0 842 595"
  }
}
```

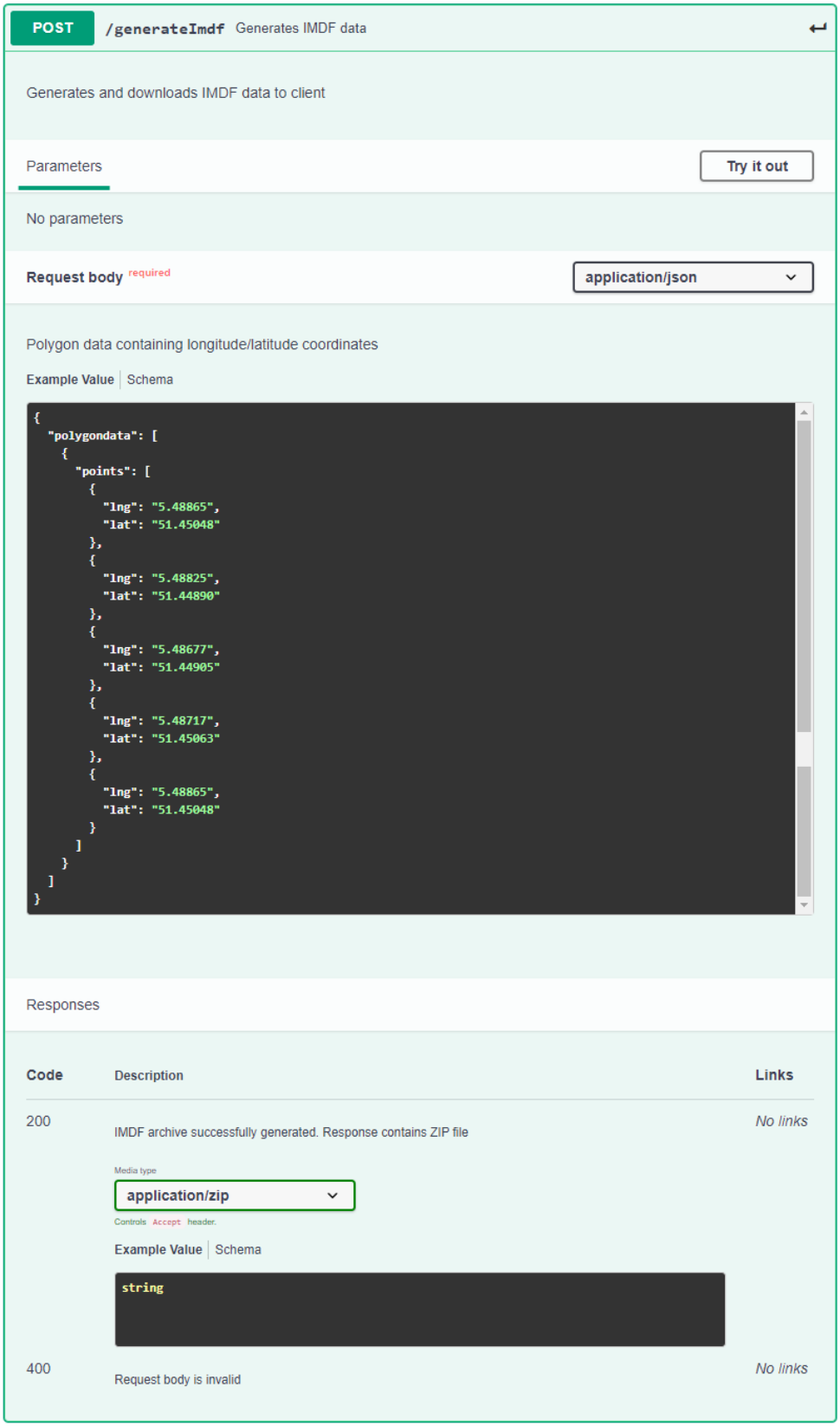

POST
/optimize
Optimizes SVG line data
Optimizes SVG line data based on provided pipeline steps.
Parameters
Try it out
No parameters
Request body
required
application/json
lineData: line data that needs to optimized
gridParams: grid parameters
steps: steps to use in optimization process (every step is either a regular step or step with parameters)
Example Value
Schema
{
"lineData": [
{
"start": {
"x": 200,
"y": 150
},
"end": {
"x": 240,
"y": 180
}
}
],
"gridParams": {
"verticalStart": 20,
"verticalEnd": 500,
"verticalInterval": 3,
"horizontalStart": 20,
"horizontalEnd": 400,
"horizontalInterval": 4
},
"steps": [
{
"id": 1,
"text": "Fill line gaps",
"value": "fillLineGaps",
"type": "variable",
"fixed": false,
"dependantOnGrid": false,
"parameters": {
"horizontalThreshold": 10,
"verticalThreshold": 20,
"diagonalThreshold": 30
},
"followUp": [],
"loopOnOriginal": "false,",
"predecessors": []
},
]
}

Responses

Code	Description	Links
200	Line data successfully optimized Media type application/json Controls Accept header. **Example Value** | Schema	*No links*
400	Request body is invalid	*No links*
500	Circular reference in pipeline detected	*No links*


POST
/generatePolygons
Converts line data into polygons
Converts line data into polygon data
Parameters
Try it out
No parameters
Request body
required
application/json
lineData: line data that needs to converted
gridParams: grid parameters
Example Value
Schema
{
"lineData": [
{
"start": {
"x": 200,
"y": 150
},
"end": {
"x": 240,
"y": 180
}
}
],
"gridParams": {
"verticalStart": 20,
"verticalEnd": 500,
"verticalInterval": 3,
"horizontalStart": 20,
"horizontalEnd": 400,
"horizontalInterval": 4
}
}
Responses
Code
Description
Links
200
Polygons successfully generated
No links
Media type
application/json
Controls Accept header.


Example Value
Schema
[
{
"points": [
{
"x": "193",
"y": "178"
},
{
"x": "193",
"y": "350"
},
{
"x": "93",
"y": "350"
},
{
"x": "93",
"y": "178"
},
{
"x": "193",
"y": "178"
}
]
}
]
400
Request body is invalid
No links

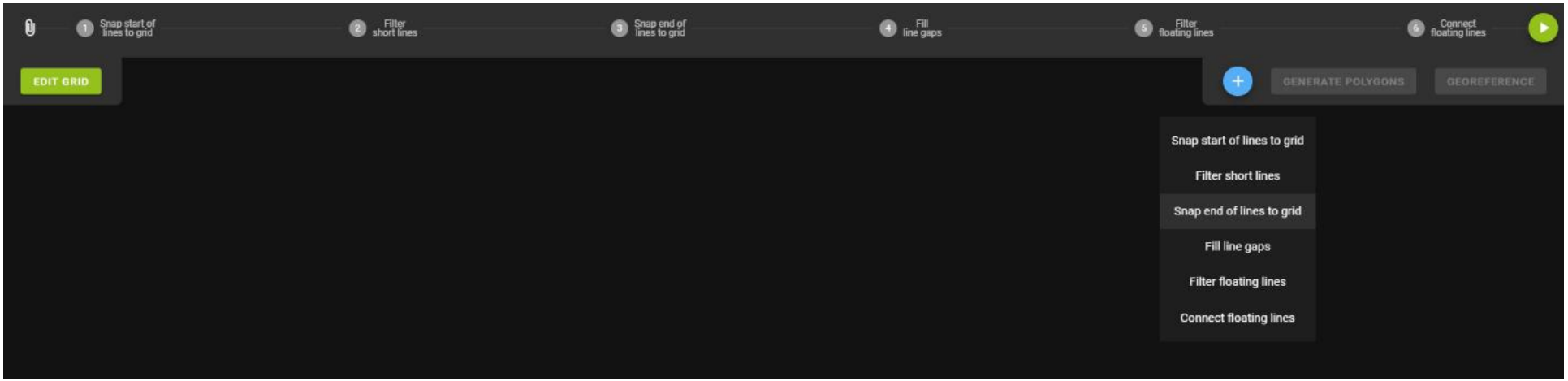

POST
/generateImdf
Generates IMDF data
Generates and downloads IMDF data to client
Parameters
Try it out
No parameters
Request body
required
application/json
Polygon data containing longitude/latitude coordinates
Example Value
Schema
{
"polygondata": [
{
"points": [
{
"lng": "5.48865",
"lat": "51.45048"
},
{
"lng": "5.48825",
"lat": "51.44890"
},
{
"lng": "5.48677",
"lat": "51.44905"
},
{
"lng": "5.48717",
"lat": "51.45063"
},
{
"lng": "5.48865",
"lat": "51.45048"
}
]
}
]
}
Responses
Code
Description
Links
200
IMDF archive successfully generated. Response contains ZIP file
No links
Media type
application/zip
Controls Accept header.
Example Value
Schema
string
400
Request body is invalid
No links

11 Screenshot SVG upload

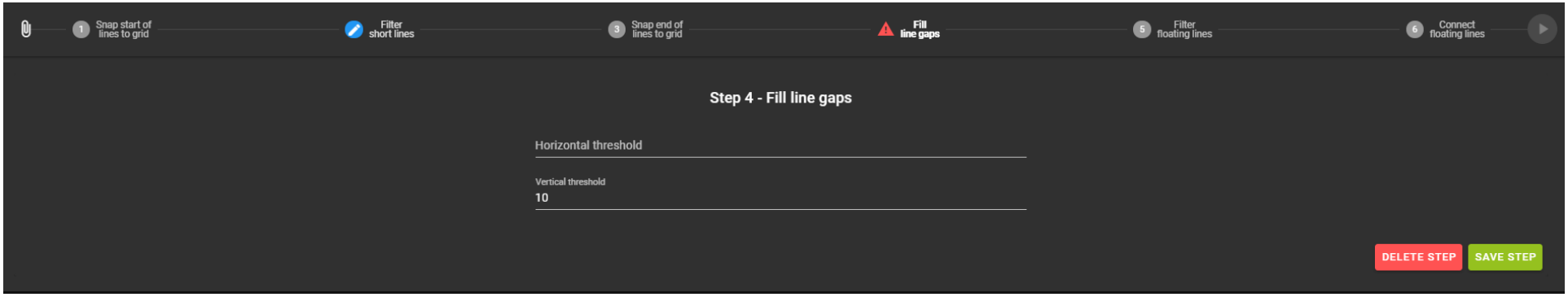
12 Screenshot grid

13 Screenshot pipeline

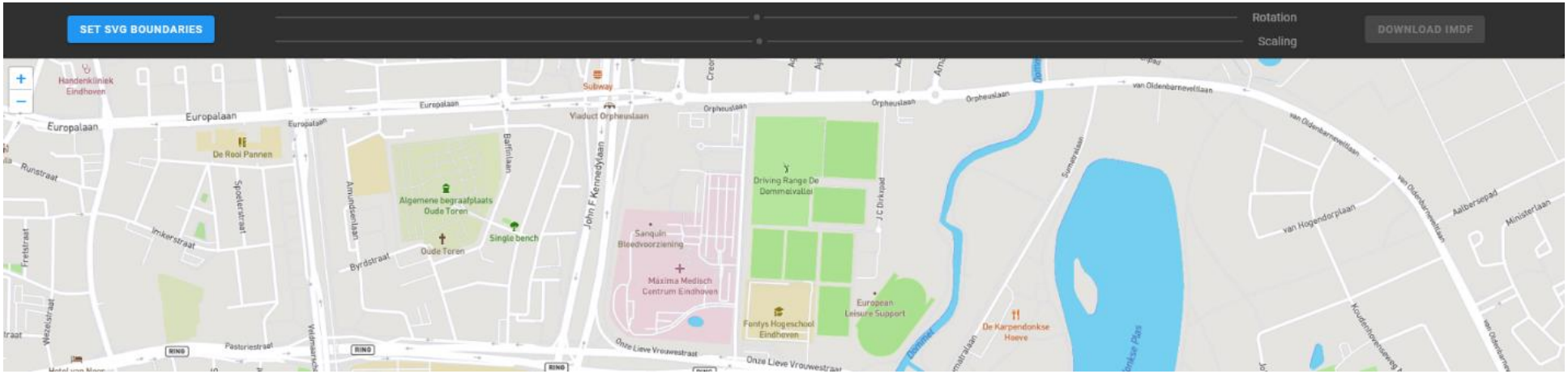
14 Screenshot toevoegen pipeline stap



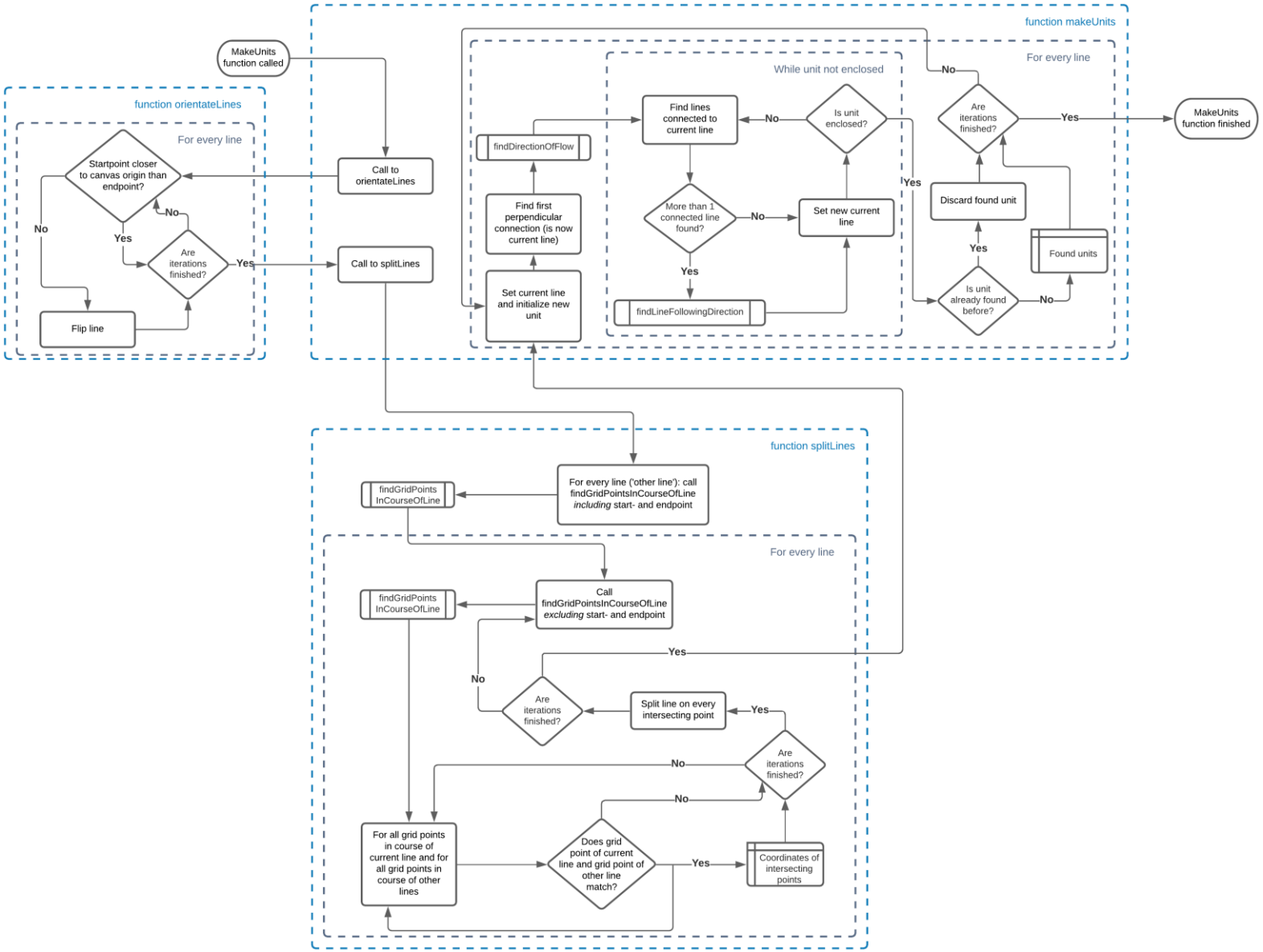
15 Screenshot pipeline stap parameters



16 Screenshot map



17 Flowchart algoritme unit herkenning



18 Test rapport unit tests

All files src/geometry

100% Statements 217/217 **92.31%** Branches 156/169 **100%** Functions 43/43 **100%** Lines 210/210

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
cartesianCoordinates.model.ts		100%	3/3	100%	0/0	100%	1/1	100%	2/2
direction.ts		100%	3/3	100%	2/2	100%	1/1	100%	3/3
endPointInFlow.ts		100%	3/3	100%	2/2	100%	1/1	100%	3/3
geoCoordinates.model.ts		100%	3/3	100%	0/0	100%	1/1	100%	2/2
geoPolygon.model.ts		100%	4/4	100%	0/0	100%	1/1	100%	4/4
geometry.helper.ts		100%	162/162	91.77%	145/158	100%	29/29	100%	158/158
line.model.ts		100%	15/15	100%	3/3	100%	2/2	100%	15/15
orientation.ts		100%	3/3	100%	2/2	100%	1/1	100%	3/3
polygon.model.ts		100%	7/7	100%	0/0	100%	2/2	100%	7/7
svgGeometryType.ts		100%	3/3	100%	2/2	100%	1/1	100%	3/3
svgPolygon.model.ts		100%	4/4	100%	0/0	100%	1/1	100%	4/4
unit.model.ts		100%	7/7	100%	0/0	100%	2/2	100%	6/6

All files src/grid

100% Statements 58/58 **93.75%** Branches 15/16 **100%** Functions 18/18 **100%** Lines 55/55

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
grid.helper.ts		100%	43/43	93.75%	15/16	100%	14/14	100%	42/42
grid.model.ts		100%	2/2	100%	0/0	100%	1/1	100%	2/2
gridParameters.model.ts		100%	7/7	100%	0/0	100%	1/1	100%	7/7
gridPoint.model.ts		100%	3/3	100%	0/0	100%	1/1	100%	2/2
gridRow.model.ts		100%	3/3	100%	0/0	100%	1/1	100%	2/2

All files src/optimization

84.24% Statements 524/622 **82.18%** Branches 166/202 **75%** Functions 81/108 **84.28%** Lines 504/598

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
modParameters.model.ts		100%	3/3	100%	0/0	100%	1/1	100%	3/3
modReturnObject.model.ts		100%	3/3	100%	0/0	100%	1/1	100%	2/2
mods.ts		100%	410/410	93.71%	149/159	100%	57/57	100%	394/394
optimization.controller.ts		0%	0/31	0%	0/13	0%	0/7	0%	0/29
optimization.module.ts		0%	0/6	100%	0/0	100%	0/0	0%	0/4
optimization.service.ts		60.9%	95/156	56.67%	17/30	51.22%	21/41	60.13%	92/153
step.model.ts		100%	13/13	100%	0/0	100%	1/1	100%	13/13

Versiebericht

Versie	*Opmerkingen*	*Datum*
0.1	Concept versie	30-05-2021
1.0	Definitieve versie	13-06-2021