Dit is het gedeelte van het verslag dat ik heb geschreven, waarin in ik een beeld geef van wat ik heb kunnen realiseren tijdens het project.

1 VOGELS OP DE KAART TONEN

Om vogels te kunnen spotten is het natuurlijk belangrijk om te weten waar deze vogels te vinden zijn. Deze informatie wordt getoond aan de hand van zones en punten op de kaart die bepaald worden door middel van doorgegeven spottingen van deze vogel. Dit zorgt ervoor dat de gegevens zich dynamisch uitbreiden naarmate het aantal spottingen van de vogel toeneemt. Aangezien de gebruiker deze punten zelf kan ingeven wordt erop gerekend dat de gebruiker er geen misbruikt van maakt. Het nagaan van de authenticiteit is daardoor moeilijk. Bij verdere uitwerkingen zou wel gezocht kunnen worden op een methode voor het tegengaan van spammen om op die manier grootschalige fouten te vermeiden.

De kaart in kwestie wordt aangemaakt via de Mapbox API. Dit is een gratis API die vaak gebruikt wordt om kaarten te tekenen. Om deze te kunnen gebruiken is het nodig om een acces token te bezitten. Deze tokens geven de gebruiker bepaalde rechten tot het gebruik van deze API en moet bij het aanmaken van een kaart meegegeven worden. Deze API maakt gebruik van GeoJSON objecten om gegevens op de kaart te kunnen laten verschijnen. GeoJSON 's zijn een veel gebruikt JSON-formaat die ontworpen is om geografische data voor te stellen. Het kan ook punten en veelhoeken voorstellen die gebruikt zullen worden om vogel spottingen te visualiseren. Een voorbeeld van een GeoJSON is zichtbaar op Figuur 1.

```
{
    "type": "Feature",
    "geometry": {
    "type": "Point",
        "coordinates": [125.6, 10.1]
},
    "properties": {
    "name": "Dinagat Islands"
}
```

Figuur 1: voorbeeld GeoJSON formaat

2 ALGORITMEN

Het voorkomen van de vogels zal weergegeven worden op de kaart in de vorm van veelhoeken en punten. De veelhoeken geven aan in welke zones de vogels vaker voorkomen. Om te vermijden dat er één grote zone is, zal een cluster algoritme gebruikt worden om de punten in verschillende zones op te delen. Het convex hull algoritme bepaalt de randpunten van deze zones, aangezien enkel deze nodig zijn voor het tekenen van de veelhoek. Het bepalen van de veelhoek gebeurt dus met behulp van twee algoritmes.

2.1 CLUSTER ALGORITME

2.1.1 HET JUISTE ALGORITME KIEZEN

De eerste stap in het proces van datavisualisatie is het verdelen van datapunten in meerdere clusters. Er bestaan verschillende cluster algoritmen die hiervoor gebruikt kunnen worden met elk hun eigen niche.

Het K-means algoritme is geschikt om een collectie van punten op te splitsen in een vooraf bepaald aantal groepen. Voor dit project is het echter niet wenselijk om op voorhand het aantal clusters vast te leggen. Bovendien is het algoritme ook gevoelig voor uitschieters, die het clustercentrum kunnen verplaatsen (Pandey, 2020).

Er is dus nood aan een ander algoritme dat de punten afbakent in groepen, gebaseerd op hoe afgezonderd ze liggen. Twee mogelijke opties hiervoor zijn hierarchical clustering algoritme en het DBSCAN-algoritme.

Het hierarchical clustering algoritme is handig wanneer er inzicht moet verkregen worden in de hiërarchische structuur van de gegevens (Patlolla, 2018). Dit is echter geen vereiste in de kader van dit project. Bovendien is de tijdscomplexiteit van dit algoritme $O(n^3)$, wat betekent dat de uitvoeringstijd kubisch toeneemt met de grootte van de dataset. Hiernaast is het gevoelig voor uitschieters (Xu, 2021).

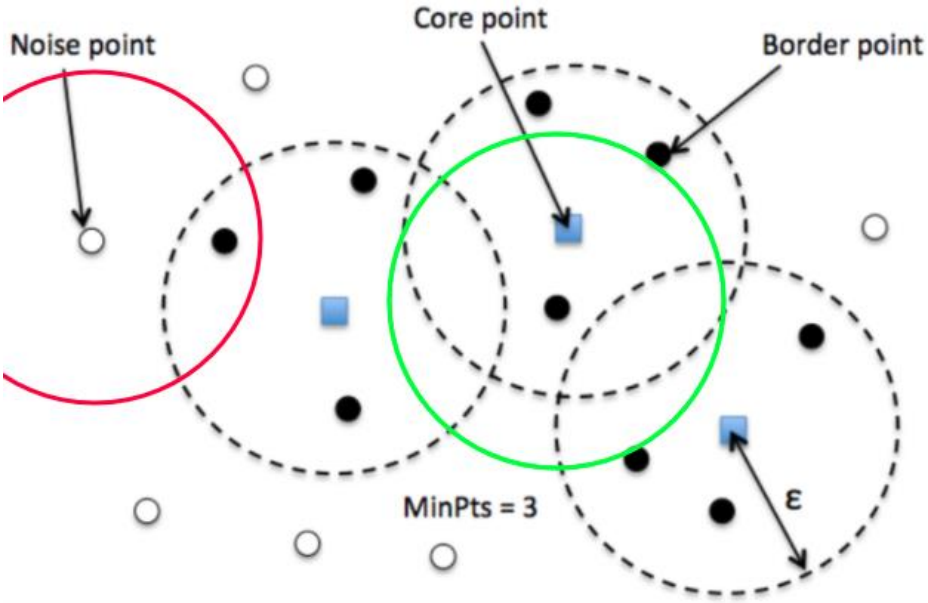
Het DBSCAN-algoritme daarentegen heeft een big O-notatie van $O(n\ log(n))$ en is daarnaast niet gevoelig voor uitschieters (Srinivasan, 2021), (Xu, 2021).

Na het onderzoeken van deze drie algoritmen is er geconcludeerd dat het DBSCAN-algoritme het meest geschikte algoritme is voor de vereisten van dit project.

2.1.2 BASISPRINCIPES VAN HET DBSCAN-ALGORITME

DBSCAN staat voor *density-based spatial clustering of applications with noise.* Het algoritme is in staat om een dataset van punten op te delen in clusters op basis van hun dichtheid in de ruimte. Het kan clusters in grote datasets identificeren door te kijken naar de lokale dichtheid van de datapunten.
Het algoritme is afhankelijk van twee parameters die als input moeten worden meegegeven, namelijk *minPts* en *epsilon*.
Epsilon is de straal van de cirkel die rond elk gegevenspunt moet worden gemaakt om de dichtheid te controleren. Een punt dat zich bevindt binnen een straal van Epsilon van een ander punt, wordt een buurpunt genoemd.
MinPts is het minimaal aantal datapunten die binnen die cirkel nodig zijn, om een datapunt te classificeren als een essentieel punt, genaamd kernpunt.

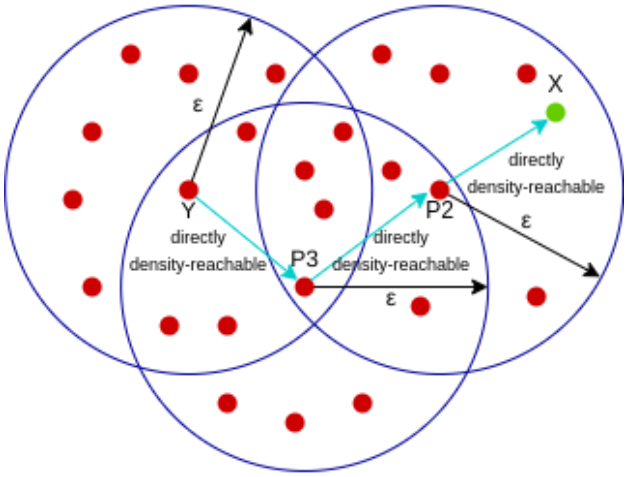
In het algemeen wordt er een onderscheid gemaakt tussen drie verschillende punten: kernpunten, randpunten en ruispunten.
Zoals eerder vermeld wordt een punt pas herkend als een kernpunt, als het een minimumaantal buurpunten bevat in een straal van epsilon. Dit minimum wordt vastgelegd door *MinPts*.
Een randpunt is een punt dat zich binnen een straal van *epsilon* bevindt van een kernpunt, maar zelf geen kernpunt is.
Een ruispunt is een punt in de dataset dat niet voldoet aan het minimumaantal vereiste buurpunten en geen enkel kernpunt als buurpunt heeft. In tegenstelling tot een randpunt, wordt een ruispunt niet beschouwd als onderdeel van een cluster.
(Chauhan, 2022), (Sharma, 2022)



Figuur 2: Verschillende soorten punten (Chauhan, 2022)

Op Figuur 15 wordt het onderscheid tussen de verschillende soorten punten duidelijk geïllustreerd. De variabele *minPts* bedraagt hier 3. De straal *epsilon*, van een willekeurig randpunt, is in het groen getekend en die van een willekeurig ruispunt in het rood.

Om dit algoritme goed te kunnen begrijpen, is het belangrijk om dieper in te gaan op twee concepten, genaamd *Density Reachability* en *Density Connectivity*.
Reachability slaat op het feit dat een punt zich binnen een bepaalde afstand (*epsilon*) van een ander punt bevindt. (Chauhan, 2022)
Connectivity daarentegen omvat een ketenbenadering tussen verschillende punten. Dit wordt geïllustreerd op Figuur 16.



Figuur 3: onnectivity relatie (Sharma, 2022)

Op Figuur 16 bevinden de punten X en Y zich niet binnen een afstand *epsilon* van elkaar, maar zijn ze wel geconnecteerd aan de hand van tussenliggende punten. Deze kettingstructuur kan beschreven worden

aan de hand van het volgende pijlendiagram: $Y \rightarrow P3 \rightarrow P2 \rightarrow X$, waarbij $Y \rightarrow P3$ een *reachability* relatie voorstelt.
Dit vormt de basis voor het vormen en uitbreiden van clusters.

2.1.3 STAPSGEWIJZE UITVOERING VAN HET DBSCAN-ALGORITME

De eerst stap is om alle punten te labelen met een “bezoekers-status”. Dit is een geheel getal en kan drie waarden aannemen.
De waarde 0 betekent dat het punt nog nooit eerder werd bezocht door het algoritme.
De waarde 1 geeft aan dat het punt werd bezocht en met zekerheid toegewijd kon worden aan één van de drie verschillende soort punten (kernpunt, randpunt en ruispunt).
Een punt krijgt de bezoekers-status 2 toegewijd wanneer het punt bezocht werd maar het soort punt nog niet bepaald kon worden

Na de uitvoering van het algoritme moet het voor elk punt duidelijk zijn tot welke cluster deze behoort. Er is dus nood aan een bepaalde identificatie. Dit wordt gerealiseerd door voor elk punt een property te voorzien, genaamd *clusterID*. Dit is een geheel getal, startende van 0, dat aan de punten wordt toegekend na het doorlopen van het algoritme. Hiermee kan elk punt zich identificeren tot een bepaalde cluster. Ruispunten kunnen niet worden toegewezen aan cluster. Deze hebben als waarde *null.*

Het algoritme begint met een willekeurig punt te kiezen uit de dataset en het aantal buurpunten op te vragen. Als het gekozen punt geen kernpunt blijkt te zijn, dan wordt de bezoekersstatus op 2 gezet. Op dit moment kan er nog niet besloten worden of het punt al dan niet behoort tot een bepaalde cluster. Alleen als het punt geen enkel buurpunt bevat, kan er met zekerheid worden gezegd dat dit een ruispunt is.

Als het gekozen punt een kernpunt blijkt te zijn, dan betekent dat er een nieuwe cluster kan gevormd worden. De functie *groeiCluster* wordt vervolgens opgeroepen.

```
groeiCluster(points, point, neighbors, clusterId, epsilon, minPts) {
    point.clusterId = clusterId;

    // Itereren over elk neighborPoint
    for (let i = 0; i < neighbors.length; i++) {
        let neighbor = neighbors[i];

        // Als het neighbor punt nog niet werd gelabeled, voeg het
toe aan de cluster en vind zijn neighbor points
        if (neighbor.loop==0) {
            neighbor.loop = 1;
            let neighborNeighbors = this.getNeighbors(points,
neighbor, epsilon);


            // Als de neighbor punt het minimum aantal neighbor
punten heeft, moet die toegevoegd worden aan de neighbors list
            if (neighborNeighbors.length >= minPts) {
                neighbors.push(...neighborNeighbors); //de neighbor
points van het huidige neighbor point toevoegen aan de lijst van
neighbors
            }
        }

        // Als de neighbor point nog niet behoort tot een cluster,
voeg die dan toe aan de huidige cluster
        if (neighbor.clusterId === null) {
            neighbor.clusterId = clusterId;
        }
    }
}
```

Figuur 4: groeiCluster functie

Figuur 4 bevat de code van de functie *groeiCluster.* Deze functie implementeert het uitbreidingsproces van de cluster. Deze functie krijgt als parameter de buurpunten mee van het kernpunt dat eerder werd gevonden.

De functie itereert over elk buurpunt en controleert of het al dan niet tot een cluster behoort. Als dat niet het geval is, dan wordt het punt in kwestie toegewezen aan de huidige cluster.
Door de buurpunten van het buurpunt in kwestie op te vragen (*neighborNeighbors*), wordt er gecontroleerd of het buurpunt een kernpunt is. Als dat het geval is, dan zullen deze punten op zijn beurt toegevoegd worden aan de lijst waarover er geïtereerd wordt.

Op deze manier worden alle buurpunten van de buurpunten van het oorspronkelijke kernpunt geëvalueerd en indien nodig toegevoegd aan de cluster. Dit proces wordt herhaald totdat er geen nieuwe buurpunten meer kunnen worden toegevoegd aan de cluster, waardoor de cluster volledig wordt gevormd.

Er kan hierbij teruggegrepen worden naar de uitleg van Connectivity van 5.1.2:
Het pijlendiagram kan nu geïnterpreteerd worden als een 2D-netwerk waarbij er vanuit elke knoop meerdere pijlen kunnen vertrekken (kernpunt bevat meerdere buurpunten die hij kan toewijzen aan de cluster), maar in elke knoop slechts één pijl kan toekomen (elk punt kan slechts aan één cluster worden toegewezen). Randpunten vormen de uiteinden van het pijlendiagram: vanuit die punten zullen er geen pijlen vertrekken.

Zoals eerder vermeld is *connectivity* een cruciaal concept in het DBSCAN-algoritme. Het bepaalt de manier waarop punten met elkaar worden verbonden. Hierdoor kunnen clusters efficiënt worden geïdentificeerd en onderscheiden worden van ruispunten. Het juiste begrip en gebruik van *connectivity* spelen daarom een essentiële rol bij het verkrijgen van betrouwbare en nauwkeurige clusteringresultaten.

2.1.4 NADELEN VAN HET DBSCAN-ALGORITME

Het DBSCAN-algoritme is een krachtig algoritme voor het ontdekken van clusters en het identificeren van ruispunten in datasets. Het is, zoals in 5.1.1 vermeld, het ideale clusteralgoritme voor dit project.
Er zijn echter ook enkele nadelen aan verbonden die in overweging moeten gebracht worden.

Een eerste nadeel is dat het resultaat van het DBSCAN-algoritme enkel afhankelijk is van de waarden van *epsilon* en *minPts.*
Kleine veranderingen in deze waarden kunnen een heel groot effect hebben op de resultaten van de clustering. Het is dus van groot belang om deze parameters correct in te stellen. Er werd daarom gekozen om de ekster toe te voegen aan de databank, de vogel is over heel Vlaanderen te vinden en is zeer herkenbaar. Op deze manier kon er op een korte periode voldoende gegevens verzameld worden om de parameters te bepalen met echte waarnemingen.

Het tweede nadeel is dat dit algoritme sterk afhangt van één statische waarde voor de densiteit. Hierdoor kan het algoritme moeite hebben met het identificeren van clusters van ongelijke dichtheden, waarbij bepaalde clusters niet correct worden herkend.
Daarnaast is het kiezen van een geschikte epsilon sterk afhankelijk van de grootte van het gebied waarop je een zicht wil krijgen.
De gekozen waarde in dit project werkt dus op een gebied zoals België.
Wanneer er uitgezoomd wordt op de kaart om een overzicht te krijgen over een groter gebied zoals Europa, zal deze waarde minder geschikt blijken.

2.2 CONVEX HULL ALGORITME

Nu alle punten toegekend zijn aan clusters kan deze informatie doorgestuurd worden naar de gebruiker. Om alles overzichtelijk te houden worden veelhoeken getekend. Voor deze veelhoeken is het enkel nodig om de hoekpunten door te sturen en is er geen nood aan de punten binnen dit gebied. Om dit te realiseren wordt gebruik gemaakt van het convexe omhulsel, deze zal minder punten bevatten dan het concave.

Er zijn meerdere varianten van het convex hull algoritme, voor de site is er gebruik gemaakt van de zogenaamde Graham scan.

2.2.1 CONVEXE VEELHOEK

Per definitie is een convexe veelhoek een figuur waar dat alle binnenhoeken minder zijn dan 180°. Dit komt erop neer dat bij het tekenen van de figuur er altijd met de klok mee-, of tegen de klok in wordt gedraaid. Een naïeve aanpak met Pythagoras en boogtangens zou wel het antwoord vinden, maar het kan een stuk efficiënter met een eigenschap van vectoren:

Alhoewel formeel gezien het kruisproduct alleen gedefinieerd is in drie dimensies, kan het toch de oplossing bieden voor deze twee dimensionale situatie. De wiskunde achter het kruisproduct moet daarvoor niet aangepast worden. Door een Z-as te introduceren loodrecht op het XY-vlak kunnen dezelfde rekenregels behouden worden.

Het kruisproduct van twee vectoren berekent een nieuwe vector die loodrecht staat op beiden. De lengte van deze nieuwe vector is per definitie gelijk aan de oppervlakte van het parallellogram opgespannen door de twee vectoren.

$$Lengte\ van\ normaal\ =\ Breedte\ *\ hoogte$$
$$Lengte\ van\ normaal\ =\ |a|\ *\ (|b|\ *\ \sin(\theta))$$

Hierbij speelt het teken van de hoek wel een rol. Een hoek in de tegenovergestelde richting levert dezelfde vector op, maar met een tegengestelde zin.

Het is deze eigenschap die het mogelijk zal maken om de draairichting elegant te bepalen. De normaal op het XY-vlak zal namelijk alleen voor zijn Z-component niet nul zijn.
Dit betekent dus dat

$$z_n\ =\ |a|\ *\ |b|\ *\ \sin(\theta)$$

De Z-component kan echter ook gevonden worden via de determinant van deze matrix:

$$\begin{bmatrix} e_x & e_y & e_z \\ x_a & y_a & z_a \\ x_b & y_b & z_b \end{bmatrix}$$

Waaruit blijkt dat:

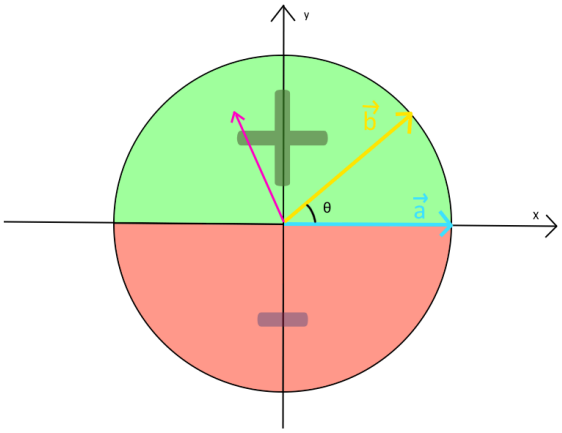
$$z_n\ =\ x_a * y_b\ -\ y_a\ *\ x_b$$

Bijgevolg is:

$$x_a \cdot y_b\ -\ y_a\ \cdot\ x_b\ =\ |a|\ \cdot |b|\ \cdot \sin(\theta)$$

Aangezien enkel het teken van de hoek van belang is, is het linker lid een veel gemakkelijkere manier om dit te vinden.
De lengtes van a en b zijn beiden altijd positief en dus wordt het teken van het rechterlid uitsluitend bepaalt door het teken van θ (De hoek ligt altijd tussen -180° en +180°). Met dat beide leden hetzelfde teken moeten hebben volstaat het om het linker lid uit te rekenen.

In een rechtshandig assenstelsel zal dit teken positief zijn wanneer de tweede vector tegen de klok in is gedraaid tegenover de eerste, en negatief wanneer hij met de klok mee is gedraaid (zie Figuur 5).
Tegen de klok in- en met de klok mee draaien stemmen dus respectievelijk overeen met “links” en “rechts te liggen van de eerste vector.



Figuur 5: visualisatie positief en negatief kruisproduct

Wanneer één van de vectoren nul is, dan zal de z-waarde ook nul zijn en levert dit een nulvector op. Het kan interessant zijn om bij het controleren van de draairichting niet alleen te kijken of de z-component groter/kleiner is dan nul, maar ook te controleren of dat de absolute waarde een bepaalde kleine waarde overschrijdt zodat de nulvector niet meegeteld wordt in bepaalde situaties.

2.2.2 PRE-ELIMINATIE MET DE ALK-TOUSAINT HEURISTIEK

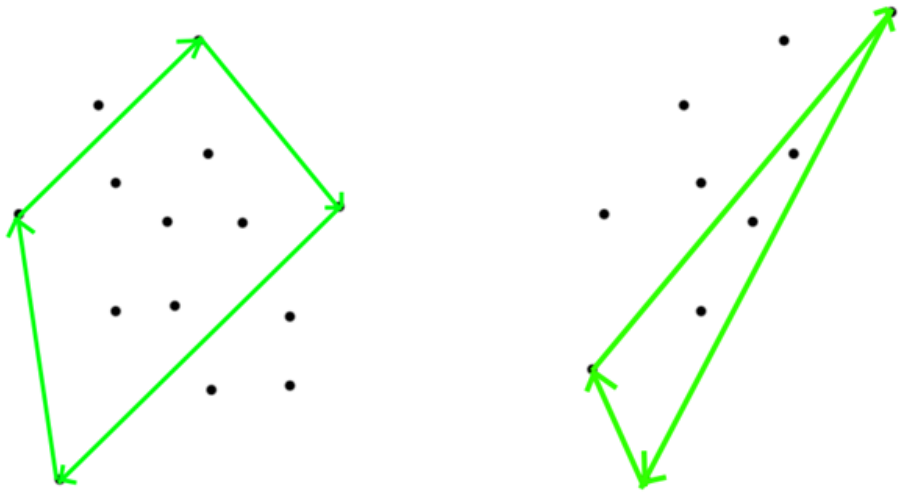
Een methode om snel het aantal te controleren punten te verminderen is de Alk-Tousaint heuristiek. Deze elimineert punten die dicht bij het midden van de verzameling liggen.

Alle punten met de uiterste x- of y-waarden liggen sowieso op het omhulsel. Dit zijn dus met andere woorden het hoogste -, laagste -, meest linkse - en meest rechtse punt.
Meestal zijn dit vier verschillende, maar in sommige gevallen zal één van de uiterst verticale punt ook het meest horizontaal uitschieten. De uitleg die volgt werkt zowel voor vier als drie verschillende punten. In het uitzonderlijke geval van slechts twee unieke punten is deze stap nutteloos sinds dat deze geen veelhoek vormen.

Wanneer deze punten verbonden worden, wordt er een convexe veelhoek verkregen, dit kan gezien worden als een minimumschatting voor het convexe omhulsel. Alle punten buiten deze veelhoek kunnen nog deel uitmaken van het omhulsel, maar de punten binnenin kunnen dat nooit, want dan wordt er een niet convex figuur gevormd.

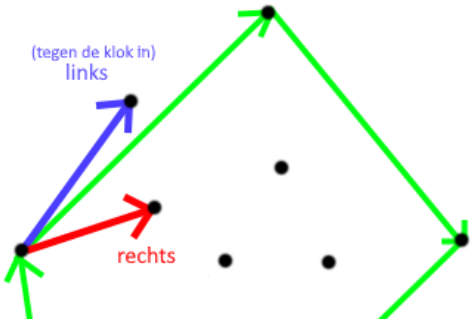
Het “verbinden” van de punten is simpelweg het opstellen van de overeenkomstige vectoren waarbij er opgelet wordt dat de richting consistent blijft, er mogen met andere woorden dus geen twee vectoren naar elkaar wijzen.

Voor de volgende uitleg is de veelhoek met de klok mee overlopen, wanneer er tegen de klok in wordt gegaan zal het resultaat van het kruisproduct tegengesteld zijn en moeten er dus aanpassingen gebeuren in onderstaande uitleg.



Figuur 6: Alk-Tousaint Heuristiek met 4 verschillende punten (links) en 3 verschillende punten (rechts)

Om te kijken of een punt in de veelhoek ligt vergelijk je de vector die de zijde voorstelt met die vanuit het aangrijpingspunt naar het testpunt. Wanneer het kruisproduct positief is dan ligt het punt links van één van de zijden en dus buiten de figuur, zo niet dan moeten de rest van de zijden nog gecontroleerd worden. Slechts als het punt rechts ligt van al de zijden, dan ligt het met zekerheid binnenin de figuur en is het dus zeker geen deel van het convexe omhulsel.



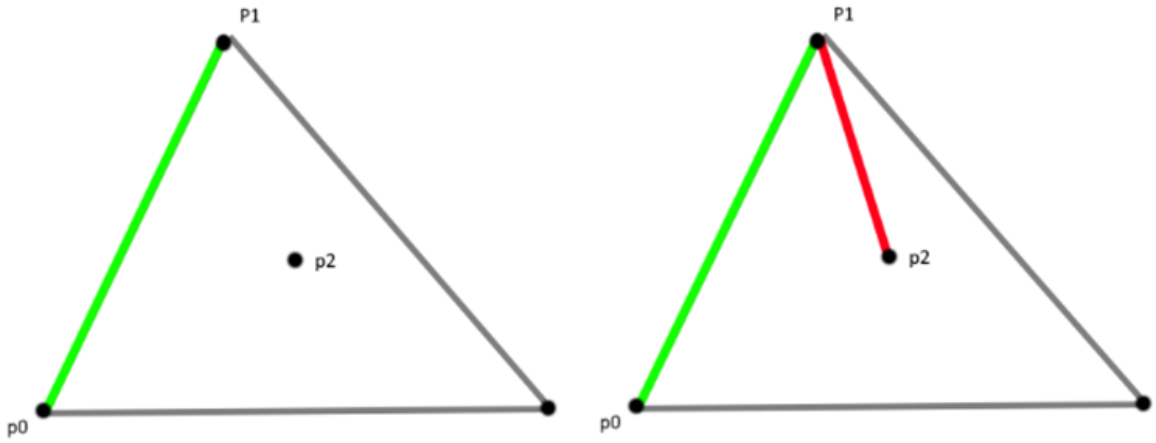
Figuur 7: Controle of punt links of rechts ligt van zijde

2.2.3 DE LUS

Om het omhulsel te vinden wordt gebruik gemaakt van een stack. Tijdens het algoritme worden alle punten één keer uitgeprobeerd en toegevoegd aan de stack. Als later blijkt dat dit punt geen deel kan uitmaken van het omhulsel omdat een nieuw punt toevoegen een concave zijde vormt, dan wordt het laatste punt van de stack gepopt totdat er geen concave zijde meer gevormd wordt.

Als de punten in een totaal willekeurige volgorde wordt overlopen dan kan er niet vanuit gegaan worden dat de punten die een concave hoek vormen toch geen deel van het convexe omhulsel zijn.

Op Figuur 8 is er te zien dat wanneer het omhulsel controleert dat al de hoeken in wijzerzin zijn, en als eerste punt het bovenste pakt, dat het toch tegen de klok in draait wanneer het naar punt twee gaat, ook al ligt punt 1 op het omhulsel.



Figuur 8: Willekeurige volgorde zorgt voor onzekerheid over het opnieuw controleren van punten

De punten kunnen wel degelijk geordend worden op een manier dat dit wel het geval is. De twee manieren zijn als volgt:

1) De punten te ordenen volgens de hoek dat ze maken met het beginpunt.
2) De punten te orden op stijgende x of y coördinaat.

Beide manieren zijn geldig:

De eerste manier moet niet berekend worden met de boogtangens, want sorteren op de boogtangens is gelijkwaardig aan het sorteren op de richtingscoëfficiënt tussen de twee punten. Ze zijn gelijkwaardig want de richtingscoëfficiënt is strikt stijgende in functie van de hoek.

De tweede manier is iets efficiënter om de punten te ordenen, maar dan moet de hele set wel twee keer overlopen worden (in het geval van het sorteren op stijgende x, dan is er een “boven” en een “onder” omhulsel.)

Voor dit project werd er gekozen voor de eerste optie.

Door de punten te overlopen in een volgorde die meegaat met de hoek kan een vorig punt dat van de stack verwijdert is nooit meer deel uitmaken van het convex omhulsel sinds dat dat een concave hoek zou vormen tussen het laatste punt op de stack, het weggesmeten punt en al de nieuwe punten.

Doordat elk punt maar één keer op de stack komt worst elk punt maar maximaal twee keer bezocht (Dus heeft het een efficiëntie van $O(n)$ als het sorteren niet wordt meegeteld.)

2.3 DATAFLOW TUSSEN DE TWEE ALGORITMEN

Om de spottingsgegevens juist te kunnen verwerken met de bovenstaande algoritmes, is er nood aan een gestructureerde dataflow tussen de twee klassen. Het is niet de bedoeling dat de klasse *DBSCAN* pas alle clusters en ruispunten doorgeeft wanneer deze allemaal gevonden zijn. De volledige uitvoertijd van het algoritme kan soms lang zijn bij heel grote datasets. Hierna zouden de clusters één voor één verwerkt moeten worden door het convex hull algoritme. Deze aanpak is niet optimaal, aangezien de gebruiker een aanzienlijk lange tijd zou moeten wachten voordat hij of zij enige data ziet op de kaart.
Een betere oplossing houdt in dat van zodra er een cluster wordt ontdekt, deze direct asynchroon op te sturen naar de *Convexhull* klasse voor verdere verwerking. Dit kan gerealiseerd worden met behulp van *promises* in Javascript. De *Convexhull* klasse zal dus telkens opgeroepen worden bij het ontdekken van een nieuwe cluster doorheen het DBSCAN-algoritme.

Er moet ook gekeken worden hoe de *Convexhull* klasse haar uitvoer moet opsturen naar de contextuele klasse van de kaart. Een mogelijke oplossingen is een fetch-cyclus te implementeren waarbij de kaart met een bepaalde frequentie herhaaldelijk vraagt aan de *Convexhull* klasse of deze nieuwe uitvoer heeft klaarstaan. Door herhaaldelijk te controleren op nieuwe output, zelfs als er geen wijzigingen zijn opgetreden, wordt er onnodig systeemresources verbruikt. Dit resulteert in een onnodige belasting en is dus geen efficiënte manier.
Een efficiëntere aanpak is het gebruik van het *Observer* pattern. Hierbij kan de *Convexhull* klasse de contextuele klasse verwittigen zodra er nieuwe output beschikbaar is. In Javascript bestaat er hiervoor een bibliotheek, namelijk RxJs. Deze biedt een set van tools en operators aan om asynchrone en event-based programma's te schrijven met behulp van observables. Er werd in dit project gebruik gemaakt van deze bibliotheek om een pipeline op te stellen tussen de twee algoritmen en de contextuele klasse.
Door de implementatie van observables, wordt naar een zwakke koppeling tussen de objecten gestreefd. De klassen bevatten geen info over elkaar, waardoor de wederzijde afhankelijkheid kleiner en het systeem flexibeler wordt (Ongenae, 2022).

In het project bevindt zich een bestand, genaamd *infoMapbox*.js. Dit bestand dient als een contextuele module. Het is namelijk verantwoordelijk voor het instantiëren, beheren en coördineren van verschillende klassen. De module zal drie observables aanmaken.

```
let obs = new Observable((subscriber) => {

    subs = subscriber;

});
```

Figuur 9: Aanmaken observable

In Figuur 9 is de code getoond voor het aanmaken van een *Observable* met behulp van de RxJs bibliotheek.
Aan het *Observable* object wordt een lambda-functie meegegeven in de constructor. Het is hierbij noodzakelijk om te begrijpen wat de subscriber precies is.
De *Subscriber* klasse fungeert als een tussenliggend object tussen de *Observable* en de *Observer*. Het biedt methoden zoals *next, error* en *complete* die worden gebruikt om waarden naar de *Observer* te sturen en te communiceren met de *Observable* (RxJs Documentation, 2022).

De *Subscriber* klasse, die toegewezen is aan de *Observabe*, moet buiten het domein van de lambda-functie beschikbaar worden gesteld. Bijgevolg wordt de initialisatie van de *Subscriber* klasse toegewezen aan een globale variabele.
Op analoge manier worden er in totaal drie *Observables* aangemaakt. Dit stelt in staat om 3 pipelines te genereren doorheen de klassen.

In het begin zullen al de datapunten opgehaald worden uit de databank met behulp van de *fetch*-functie. De uitvoer wordt doorgegeven aan de functie *makeClusters*.

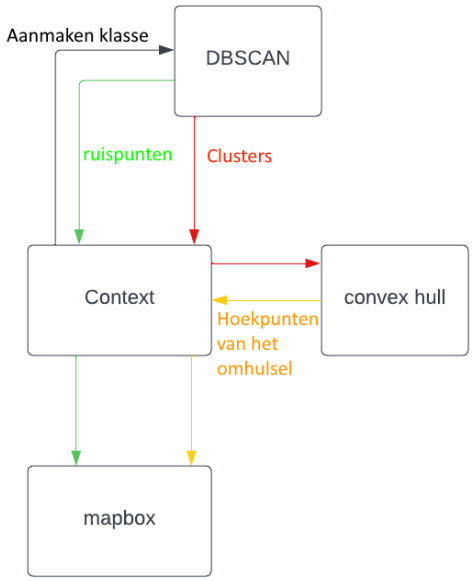
```
function makeClusters(points){


        let DBSCAN =  new Dbscan(points, epsilon,minPts, subscriber,
subscriber3);
        //uitvoeren algoritme
        DBSCAN.dbscan()
}
```

Figuur 10: MakeClusters functie

Op Figuur 10 is de code van *makeClusters* te zien. Deze functie zal een nieuwe instantie van de *DBSCAN*-klasse aanmaken en de datapunten meegeven in de constructor. Bovendien worden ook de invoerparameters van het algoritme, samen met twee *Observables* meegegeven aan de constructor. Telkens als er een nieuwe cluster wordt gemaakt door het algoritme, wordt deze op de eerste *Observable* opgestuurd. De tweede *Observable* dient om de ruispunten mee op te sturen. Het is evident dat er hiervoor twee pipelines nodig zijn, aangezien de ruispunten direct op de map worden getoond en niet moeten verwerkt worden door de *Convexhull* klasse.
Het is hier duidelijk dat er een globale scoping vereist is voor de *Subscriber* klassen van de *Oservables*, aangezien deze klassen geïnjecteerd moeten worden in de constructor van de *DBSCAN*-klasse.
De verantwoordelijkheid voor het maken van de *Observables* ligt hier in de handen van de contextuele module en niet in die van de algoritmen. Ook worden de invoerparameters (epsilon en minPts) meegegeven via de consructor. Door gebruik te maken van dependency injection, wordt er hier voldaan aan het Inversion of Control principe. De parameters en afhankelijkheden worden niet gedefinieerd in de klasse zelf, maar wel in de contextuele module. Hierdoor wordt de flexibiliteit en testbaarheid van de code vergroot, omdat verschillende implementaties van de parameters gemakkelijk kunnen worden geïnjecteerd zonder de code van het algoritme zelf te wijzigen (Ongenae, 2022).

In de contextuele module wordt er geabonneerd op deze twee *Observables*. De data die wordt ontvangen op de pipeline voor ruispunten wordt direct opgestuurd naar een functie die het tonen van punten op de kaart.

De data die wordt ontvangen op de pipeline voor clusterpunten wordt opgestuurd naar een nieuwe instantie van de *ConvexHull* klasse. Aan deze klasse wordt eveneens de derde *Observable*, die eerder werd aangemaakt, meegegeven via de constructor.



Figuur 11: Dataflow tussen de klassen

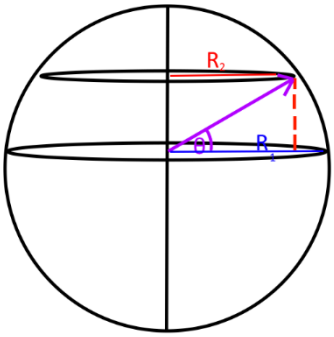
Deze *Observable* zorgt voor een derde pipeline. Deze pipeline zorgt voor de dataflow tussen de uitvoer van het convex hull algoritme en de contextuele module.
Wanneer de module data ontvangt op deze pipeline, wordt de deze opgestuurd naar een functie die verantwoordelijk is voor het tonen van veelhoeken op de kaart. Op Figuur 11 wordt dit verloop geïllustreerd aan de hand van een diagram. Voor elke pipeline is er een andere kleur voorzien.

Het is belangrijk om op te merken dat de *DBSCAN*-klasse de clusters niet zelf opstuurt naar de *Convexhull* klasse, maar dat dit gebeurt via de contextuele module. De module fungeert hier als een Mediator. Dit zorgt voor een minder sterke koppeling tussen de klassen.

2.4 AFSTAND OP EEN BOLOPPERVLAK EN BENADERINGEN

Geografische coördinaten worden uitgedrukt in hoeken wat ervoor zorgt dat Pythagoras toepassen niet het juiste resultaat zal leveren. Er wordt al een eerste benadering gemaakt door aan te nemen dat de aarde een perfecte bol is. Sowieso gaat er dus wat speling zitten op de effectieve afstand onafhankelijk van welke formule er gebruikt wordt. Voor toepassingen die nauwkeurig moeten zijn wordt er meestal beroep gedaan op de haversine formule. Deze formule is echter niet zo performant wanneer men het veel moet oproepen (zoals in het DBSCAN-algoritme). Daarom werd er gekozen voor de equirectangular benadering te gebruiken.

Deze formule houdt er rekening mee dat naarmate de breedtegraad vergroot in absolute waarde, de radius van de doorsnede verkleint. Doordat de radius verkleint, verkleint ook de “horizontale” component (evenwijdig met de evenaar) van de afgelegde weg. Deze verkleining is evenredig met de cosinus van de breedtegraad (*Veness*, 2020).



Figuur 12: Verklaring schalingsfactor lengtegraad

Voor de formule wordt het gemiddelde van de breedtegraden genomen en wordt er dan een projectie gemaakt op een vlak dat rekening houdt met Figuur 12.

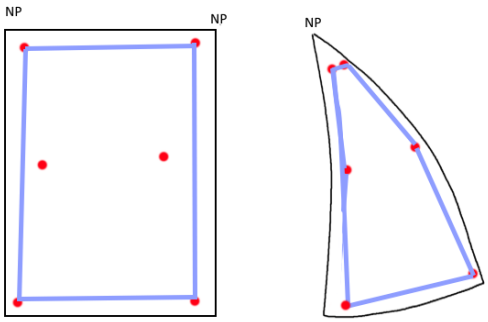
$$\theta = breedtegraad$$
$$\varphi = lengtegraad$$

$$\Delta x = R \cdot \Delta\varphi \cdot \cos\left(\frac{\theta_1 + \theta_2}{2}\right)$$
$$\Delta y = R \cdot \Delta\theta$$
$$d = \sqrt{\Delta x^2 + \Delta y^2}$$

De equirectangular benadering is minder nauwkeurig dat de haversine formule, maar de snellere berekening is belangrijker dan de extra precisie.

In de code van de *DBSCAN* klasse werd er bij het bepalen van de afstand voor efficiëntie redenen gecontroleerd op $d^2 > epsilon^2$ in plaats van $d > epsilon$. Hierdoor moet de vierkantswortel niet uitgerekend worden. Ook is het vermenigvuldigen met de straal R weggelaten uit de code. Het weglaten van de vermenigvuldig was zodat de precisie bij het vergelijken van de twee waarden niet nodeloos verloren ging door beide leden te vermenigvuldigen met een zeer groot getal.

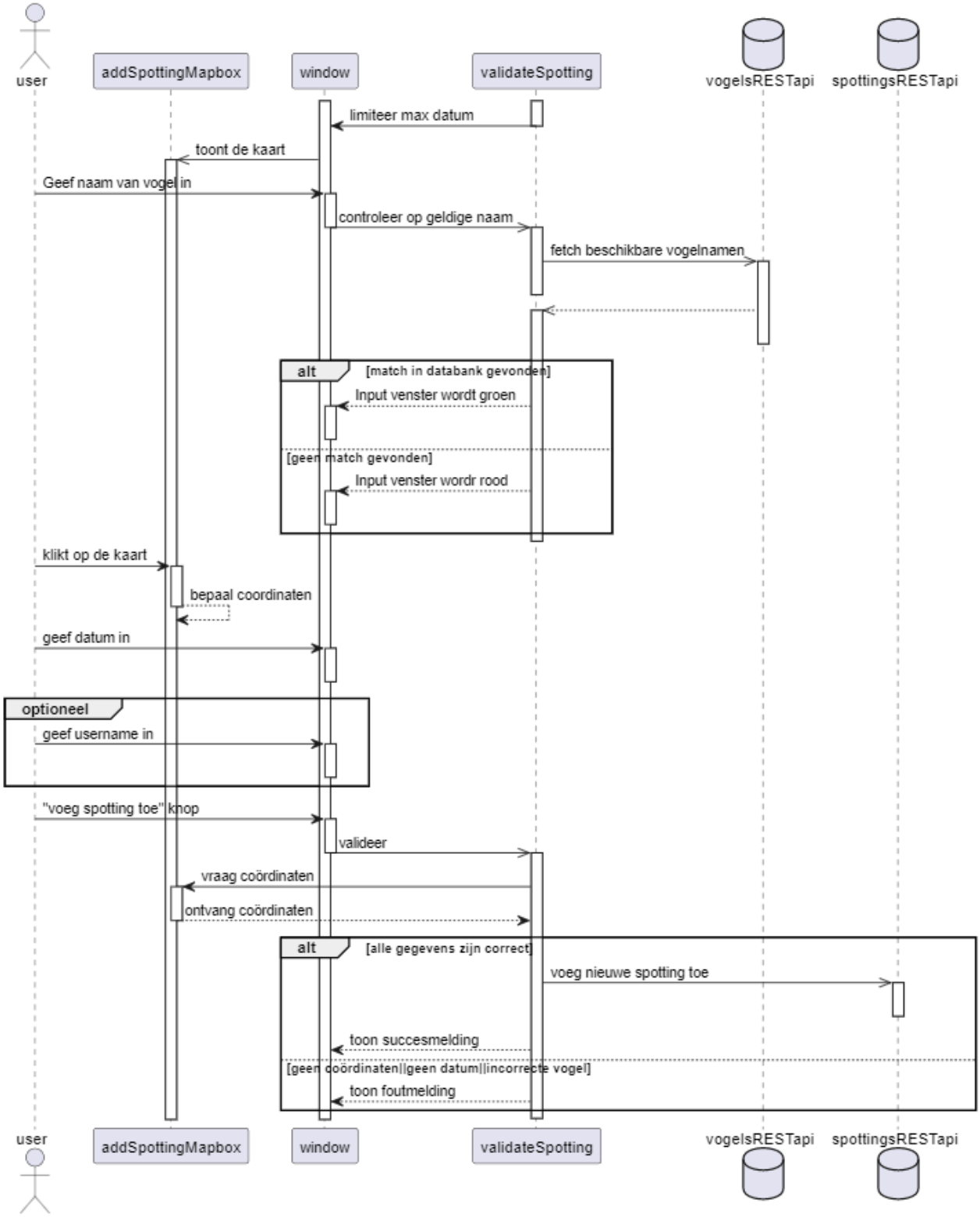
Voor het convexe omhulsel is er gerekend met de hoeken alsof ze standaard xy-coordinaten zijn op een vlak. Aangezien alleen de richting van belang is voor het convex hull algoritme is er niet overwogen om hiervoor een aanpassing te voorzien. Dit betekent niet dat er helemaal geen fout op zit. Dit is gemakkelijk weer te geven met een voorbeeld dicht bij de polen:



Figuur 13: Voorbeeld verschuiving boloppervlak in buurt van noordpool

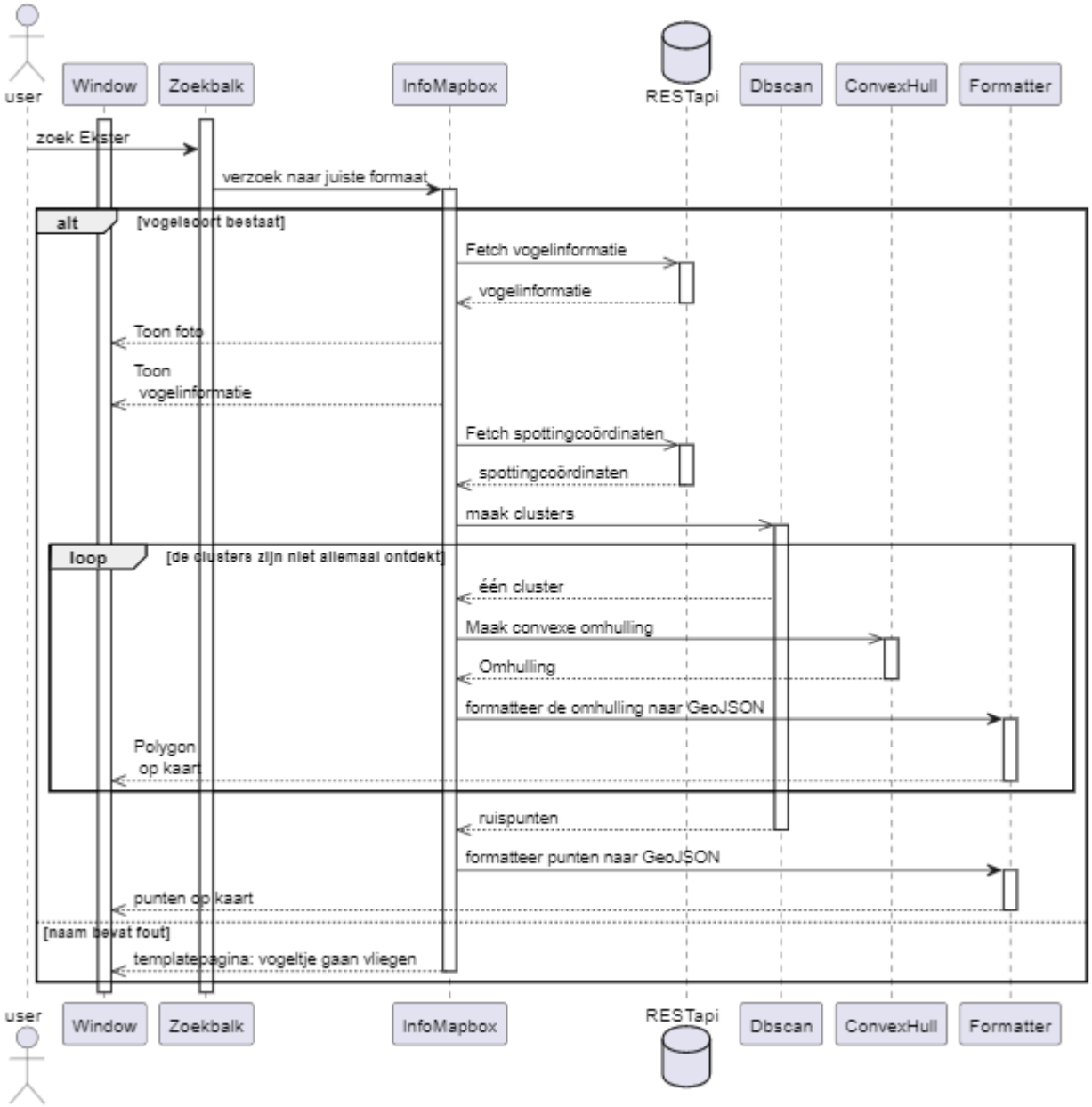
De noordpool is één punt, maar op Figuur 13 is te zien hoe dat bij het projecteren op een vlak de noordpool een volledige lijn wordt. Dit komt erop neer dat het theoretische mogelijk is voor punten die in het vlak niet op het convexe omhulsel liggen, toch samen te vallen in de werkelijkheid.

3 UML Diagrammen



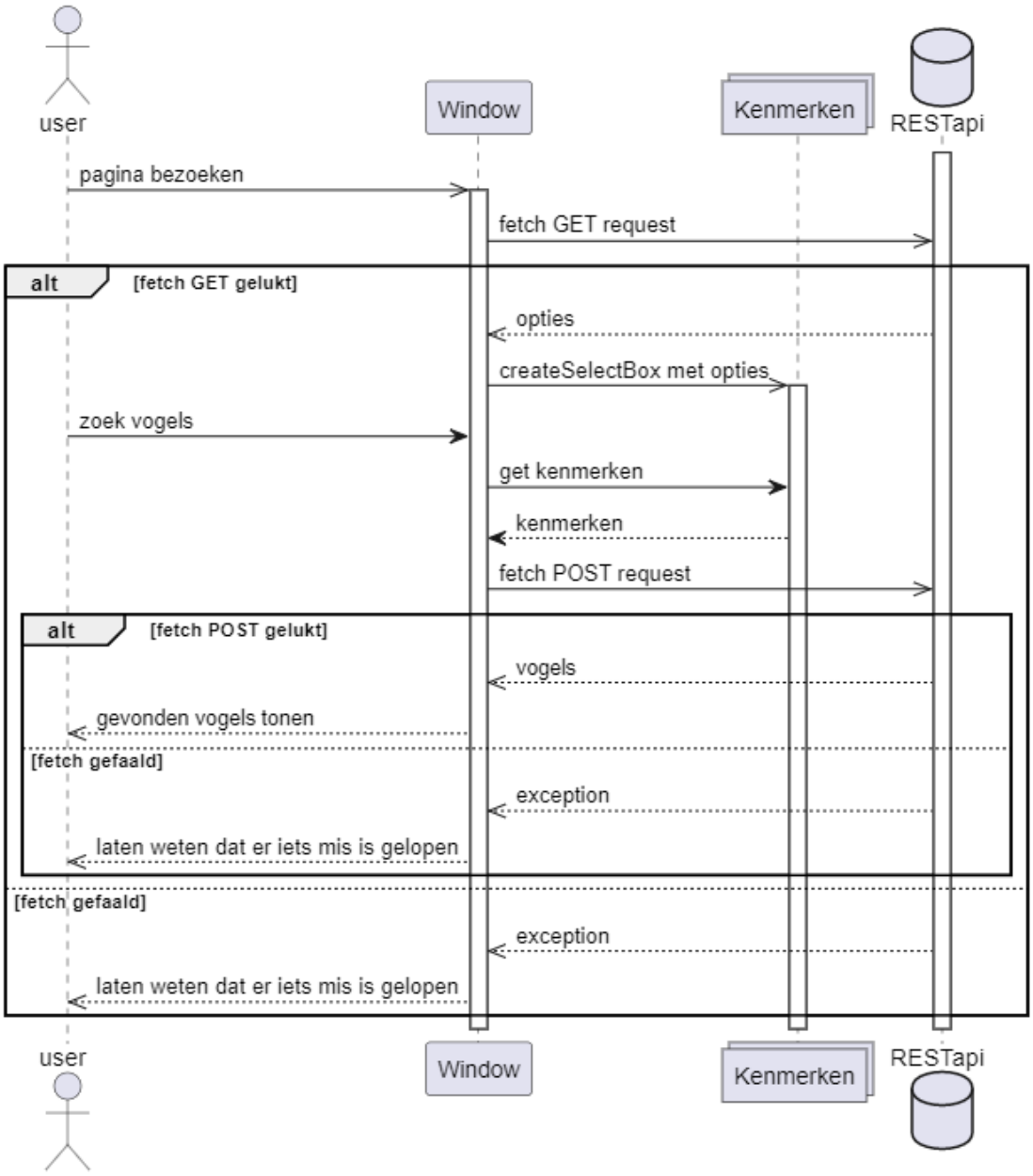
Figuur 14: Sequentiediagram "spotting toevoegen"

Op Figuur 14 is te zien hoe het toevoegen van een spotting op de “voeg Spotting”-pagina verloopt. De invoer van de gebruiker hoeft niet verplicht in deze volgorde te verlopen. Eenmaal alle invoer compleet is kan er op de knop “voeg toe” gedrukt worden en als alles in orde is zal de spot naar de databank geüpload worden.



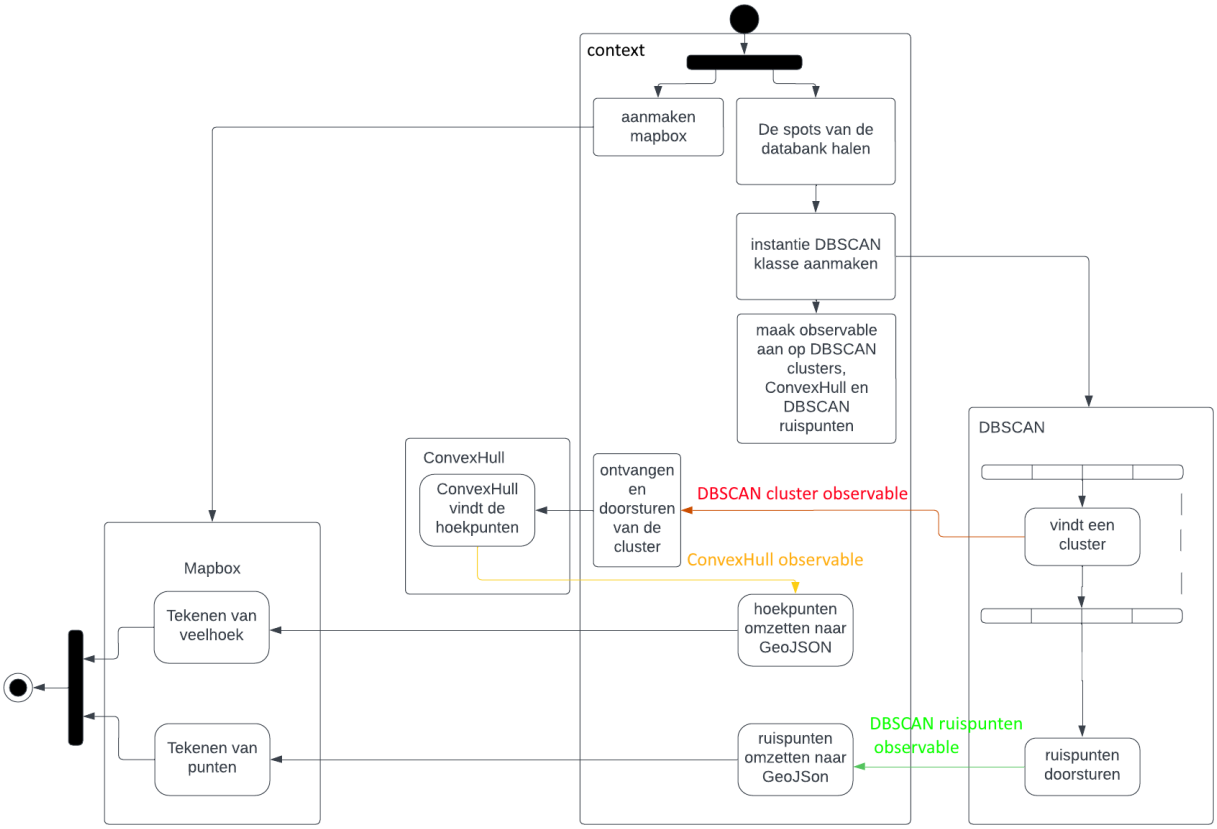
Figuur 15: SequentieDiagram “Vogel opzoeken”

Op Figuur 15 is te zien hoe het proces verloopt wanneer een gebruiker een vogelnaam ingeeft in de zoekbalk. Bij het opstarten van de pagina wordt de data van de vogel opgehaald. Als deze bestaat wordt de html verder aangevuld met informatie en de juiste data op de kaart getoond.



Figuur 16: Sequentiediagram "Vogel Herkennen"-pagina

Op Figuur 16 wordt met een sequentiediagram voorgesteld hoe een onbekende vogel geïdentificeerd wordt in de "Vogel Herkennen"-pagina.
Hier zijn alle functies in verband met de databank asynchrone functies, hierdoor moet de gebruiker niet wachten op een antwoord. Eenmaal een antwoord wordt teruggegeven zullen de juiste html elementen ook aangemaakt worden.



Figuur 17: Dataflow van het algoritme

In Figuur 17 is te zien hoe dat de infomapbox.js (context op deze diagram) de flow van data doorstuurt naar en ophaalt van andere klassen. Wanneer de context een klasse aanmaakt van een algoritme, dan zal het antwoord terug opwachten aan de hand van observables.

BRONNENLIJST

[1] Sharma, A. (2022, 20 juni). *How to Master the Popular DBSCAN Clustering Algorithm for Machine Learning. Analytics Vidhya*. Laatst geraadpleegd op 22/03/2023 via https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/

[2] Chauhan, N. S. (2022, 4 april). *DBSCAN Clustering Algorithm in Machine Learning - KDnuggets. KDnuggets* Laatst geraadpleegd op 22/03/2023 via https://www.kdnuggets.com/2020/04/dbscan-clustering-algorithm-machine-learning.html

[3] Srinivasan. (2021, 25 februari). *Top 5 Clustering Algorithms Data Scientists Should Know. Digital Vidya*. Laatst geraadpleegd op 22/03/2023 via https://www.digitalvidya.com/blog/the-top-5-clustering-algorithms-data-scientists-should-know/

[4] Pandey, A. K. (2020, 5 oktober). *A Simple Explanation of K-Means Clustering. Analytics Vidhya*. Laatst geraadpleegd op 2/03/2023 via https://www.analyticsvidhya.com/blog/2020/10/a-simple-explanation-of-k-means-clustering/

[5] Xu, Y. (z.d.). *The Computational Linguistics and Cognitive Sciences (CLCS) Lab : Homepage*. Laatst geraadpleegd op 22/03/2023 via https://clcs.sdsu.edu/

[6] Patlolla, C. R. (2022, 15 augustus). *Understanding the concept of Hierarchical clustering Technique. Medium* laatst geraadpleegd op 22/03/2023 via https://towardsdatascience.com/understanding-the-concept-of-hierarchical-clustering-technique-c6e8243758ec

[7] com.mapbox.geojson (services-geojson API). (2021, 15 maart). Laatst geraadpleegd op 22/03/2023 via https://docs.mapbox.com/android/java/api/libjava-geojson/5.8.0/com/mapbox/geojson/package-summary.html

[8] Wikipedia contributors. (2023, 4 maart*). GeoJSON. Wikipedia*. Laatst geraadpleegd op 22/03/2023 via https://en.wikipedia.org/wiki/GeoJSON

[9] Tutorials | Help. (z.d.). *Mapbox*. Laatst geraadpleegd op 15/03/2023 via https://docs.mapbox.com/help/tutorials/

[10] TypeORM - *Amazing ORM for TypeScript and JavaScript (ES7, ES6, ES5). Supports MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, WebSQL databases. Works in NodeJS, Browser, Ionic, Cordova and Electron platforms*. Laatst geraadpleegd op 13/03/2022 via https://typeorm.io/

[11] Fichier:Jarvis march convex hull algorithm diagram.svg. Wikipédia. (z.d.). https://fr.m.wikipedia.org/wiki/Fichier:Jarvis_march_convex_hull_algorithm_diagram.svg

[12*] RxJS Documantation*. (2022). Laatst geraadpleegd op 17/05/2023 via https://rxjs.dev/guide/overview

[13] Ongenae, V. (2022). *Softwareontwikkeling syllabus*. Universiteit Gent.

[14] Jonsson, L.(1993*). Vogels van Europa*. Tirion

[15] Tarick. (2022, August 2). *90 weetjes over vogels. Leuke Weetjes*. Laatst geraadpleegd op 20/05/2023 via.https://100-facts.com/nl/90-interessante-feiten-over-vogels/

[16] *Postman* Laatst geraadpleegd op 20/05/2023 via https://www.postman.com/

[*17]* Venesse, C. (2020) *Calculate distance, bearing and more between Latitude/Longitude*. *Movable Type Scripts*. Laatst geraadpleegd op 16/05/2023 via https://www.movable-type.co.uk/scripts/latlong.html

[18] Wikipedia contributors. (2022, 22 november*). Graham Scan. Wikipedia*. Laatst geraadpleegd op 10/05/2023 via https://en.wikipedia.org/wiki/Graham_scan