— Eindverslag P&O CW 2019–2020 —

Department Computerwetenschappen – KU Leuven 2 december 2019

Team 12

Frédéric Blondeel
Martijn Debeuf
Toon Sauvillers
Dirk Vanbeveren
Bert Van den Bosch
Seppe Van Steenbergen

Inhoudsopgave

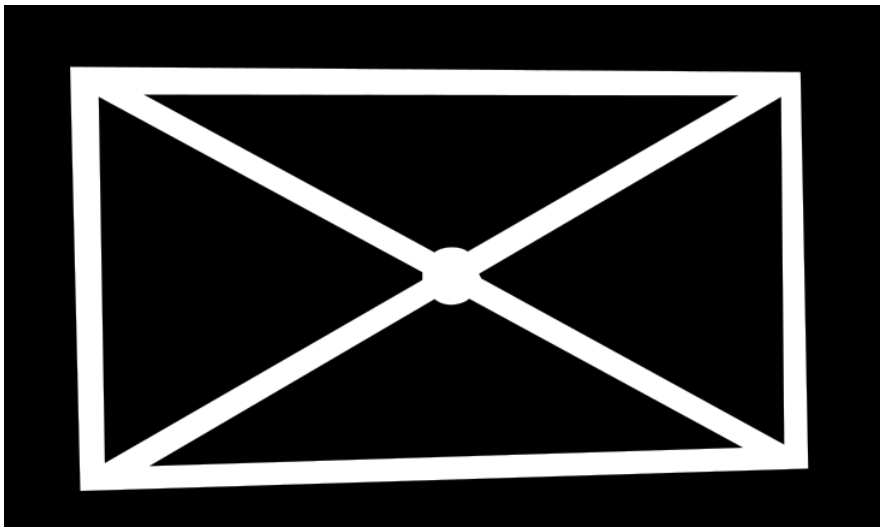
1 Detectie

1.1 Flood fill

Elk individueel slave scherm laat een vooraf bepaalde afbeelding zien met gekende kleurwaarden (*afbeelding van html). Om deze schermen te detecteren wordt de foto gefilterd op basis van gekende HSL ranges (TODO: verwijzing naar kleur van Seppe) en een standaard four-way flood fill algoritme [4 way afbeelding of te veel?] (*verwijzing wikipedia flood fill) om de associatie van de verbonden pixels te behouden. Na de executie van de flood fill is er voor elk gedetecteerd eiland een pixel masker met een unieke ID waar de verdere bewerkingen op uitgevoerd zullen worden. Het geimplementeerde flood fill algoritme groeit volgens de vier pixel-buren en een stack-based iteratie process om recursie-overflow tegen te gaan bij grote afbeeldingen en eilanden. In een worst-case scenario zal dit algoritme een eiland detecteren over de volledige afbeelding. Aangezien elke pixel maximaal vier keer in de stack terecht kan komen, door zijn vier buren, loopt deze flood fill volgens een tijdscomplexiteit van

$$O(4mn) = O(mn)$$

met m en n de dimensies van de afbeelding. De grootte van elk eiland zal in de praktijk over het algemeen een stuk kleiner zijn dan de volledige afbeelding (*verwijzing tijdscompl.).



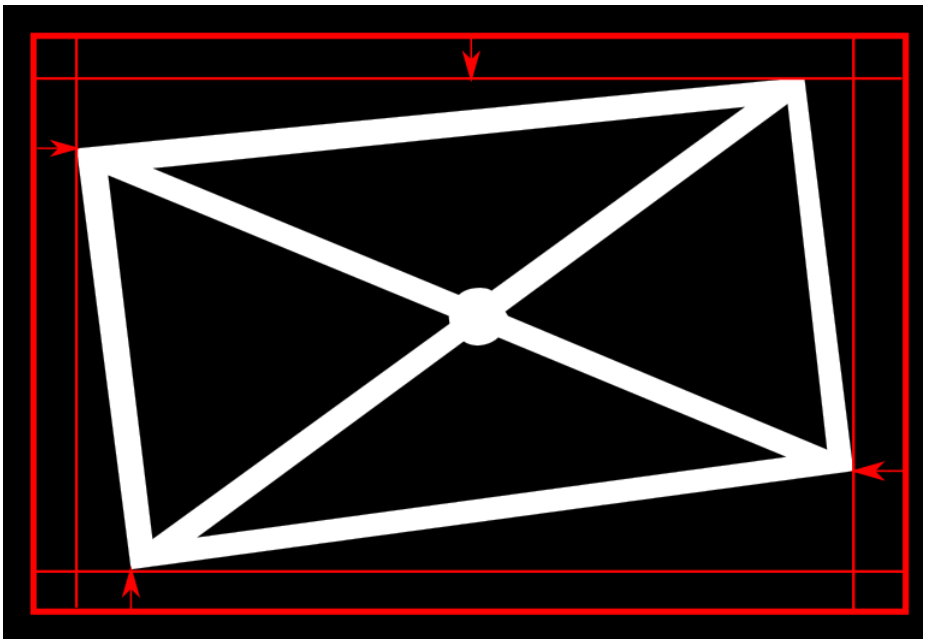
Figuur 1: Kleurenmasker van een scherm

Niet enkel de associatie van de gemaskeerde pixels wordt op deze manier behouden, maar deze methode maakt ook dat de komende bewerkingen maar over een minimale bounding box uitgevoerd worden ten opzichten van de volledige pixel matrix. De afbeelding dat op elk slave-screen afgebeeld wordt voor detectie, is een combinatie van vorige iteraties van het project. Het combineren van een border met een kruis geeft het meeste informatie over de associatie van de kleur-gefilterde pixels bij overlap of afgedekte delen van het scherm en bovendien meer informatie over de oriëntatie van het scherm op de foto.

1.2 Hoekpunten

In de eerste stap wordt er bepaald of het scherm voornamelijk recht of gekanteld is ten opzichte van de foto. Hiervoor wordt langs de linker kant van het kader nagegaan hoe hoog de standaarddeviatie van witte pixels is. Bij een standaardafwijking onder de 15%

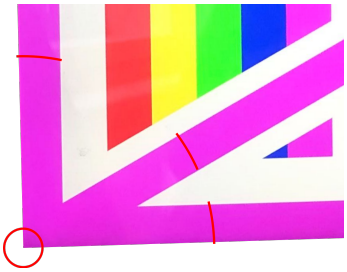
wordt een scherm als liggend of verticaal gezien op de foto.
Er werden vooraf algemene hoek-detectie algoritmes zoals shi-tomashi geïmplementeerd en getest, maar gaven een te complex resultaat op de binaire maskers om de juiste hoeken te filteren. Dit algoritme draagt ook een relatief grote overhead door de x- en y-sobel operaties die in een eerste stap toegepast moeten worden. Onze algoritmes zijn veel simplistischer, maar geven een perfect bruikbaar resultaat voor onze noden.
Als in eerste instantie het scherm als gedraaid beschouwd wordt, zal er vanuit elke rand van de bounding box van het eiland het de eerste mask-pixel als corner beschouwd worden. In het geval dat het scherm relatief horizontaal of verticaal recht staat, zal er loodrecht op de randen gezocht worden, maar volgens een diagonaal tot een mask-pixel [zie afbeeldingen].



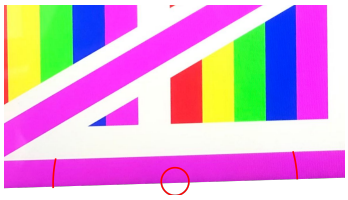
Figuur 2: Loodrechte search

Beide variaties van hoek-detectie zal altijd vier hoeken als resultaat opleveren. Dit zullen door overlap en foutjes in het maskeren niet altijd correcte hoeken zijn. Na het bepalen, worden de hoeken nagekeken of deze resultaten wel degelijk kwalificeren als hoek. Deze kwalificatie is gebaseerd op bepaalde eigenschappen die in de buurt van elke hoek moeten gevonden worden, namelijk twee lijnen die tot de boord behoren en een diagonaallijn die naar het middelpunt van het scherm loopt **??**. Deze lijnen zijn bepaald door te filteren door de border- en diagonaal-kleur die gescheiden zijn door een witte rand (*beter verwijzen naar waar calccircle wordt uitgelegd in reconstructie?). Als in een eerder gevonden hoek deze voorwaarden niet aanwezig zijn, wordt deze hoek verworpen **??**.

(linken/binden naar reconstruction) Als er geen vier correcte hoeken gevonden worden, zullen missende hoeken gereconstrueerd worden uit de wel gevonden hoeken en het middelpunt.



(a) Een correcte hoek



(b) Hoek zonder volledige features

2 Reconstructie

Na het filteren van de hoeken kan het dus voorkomen dat er geen vier meer overblijven. Volgende kunnen hiervoor de oorzaak zijn. Enerzijds is het mogelijk dat bepaalde delen van schermenelkaar overlappen in de opstelling, anderzijds kunnen één of meerdere hoeken niet of slecht detecteerbaar zijn door een obstakel. Er wordt opgelegd dat minstens twee aanliggende hoekpunten en het middelpunt zichtbaar zijn. Indien men zich aan deze vooropgestelde eis houdt kan met volgend algoritme het scherm steeds volledig gereconstrueerd worden. Na een opsomming van de stappen volgt een meer gedetailleerde uitleg.
De input die wordt meegegeven is een dictionary van de reeds gevonden hoekpunten. De sleutels zijn LU,RU,RD en LD m.a.w. posities van hoeken en als bijhorende waarden coördinaten voor deze die reeds gevonden zijn en een *null* als plaatshouder voor de nog te reconstrueren hoeken.
Eerst worden de vier punten bepaald die zich rond het middelpunt op de diagonalen bevinden. Deze worden ook in een dictionary opgeslaan met dezelfde structuur als de input. Daarna wordt hoek per hoek gekeken welke nog ontbreken. Indien reconstructie nodig is, wordt het overeenkomende punt van de diagonalen genomen. Samen met het middelpunt wordt hieruit een eerste reconstructielijn opgesteld. Daarna wordt vanuit een aanliggend hoekpunt het laatste hulppunt bepaald waardoor de tweede rechte wordt getrokken. De gezochte hoek is dan het snijpunt van de twee constructielijnen. Deze stappen herhalen voor andere ontbrekende hoeken resulteert in een dictionary met alle hoekenpunten van het scherm. Hieronder volgt een uitgebreidere uitleg van gebruikte methodes met veronderstellingen, voordelen en nadelen.

2.1 Hulppunten op diagonalen

Telkens wanneer reconstructie nodig is zulllen de punten op diagonalen rond het middelpunt bepaald en geordend opgeslaan worden in een dictionary. Hiervoor worden alle pixels overlopen die op de cirkel met een bepaalde radius rond het middelpunt liggen overlopen. De straal van deze cirkel wordt gedefiniëerd als een vierde van de grootste afstand tussen de reeds gevonden hoeken en het middelpunt. Op deze manier wordt rekening gehouden met de grootte van het scherm. 25 procent van deze afstand nemen zorgt ervoor dat de pixels niet tot het middelpunt behoren en er zich tegelijkertijd niet te ver van bevinden. Het startpunt van waaruit de cirkel doorlopen wordt, is een pixel die niet tot een diagonaal behoord. Waarom dit belangrijk is zal later duidelijk worden. Bepalen of een punt deel uitmaakt van een diagonaal gebeurt aan de hand van een functie die controleerd of er zich tussen de desbetreffende pixel en een tweede meegegeven punt (in dit geval het middelpunt) een wit deel pixels bevindt. Indien er tussen de twee meegegeven punten een stuk wit voorkomt, betekend dit dat de pixel afkomstig is van een deel barcode **??** (*verwijzing foto

van screendetectie html). Doordat de kleuren van de boord en diagonalen ook in de barcode voorkomen, sluit deze functie dus eigenlijk uit dat stukjes barcode toegevoegd worden aan de lijst. Het interval waarmee de hoek van nul tot twee keer pi loopt is één procent, hiermee worden net genoeg pixels overlopen. Naar later toe zou deze waarde ook relatief kunnen gezet worden naar de grootte van het scherm. Voor elke aaneenschakeling van pixels die geen wit kruisen op hun pad naar het middelpunt wordt een lijst aangemaakt tijdens het doorlopen van de cirkel. Uiteindelijk heeft men dan vier sublijsten van pixels in een lijst die elk een diagonaal voorstellen. Uit elke lijst wordt dan het middelste element genomen die het midden van dat cirkelsegment is. Hier komt al een eerste voordeel van het bepaalde startpunt naar voor. Indien dit niet gedaan zou worden kon het voorkomen dat een cirkel binnen een diagonaal startte. Dit zou resulteren in meer dan vier lijsten die aangemaakt worden wat het bepalen van de middens aanzienlijk complexer zou maken. Zoals eerder vermeld worden deze vier punten dan vanuit een lijst geordend in een dictionary geplaatst. Als referentie wordt gestart vanuit de HTML voor de schermdetectie. Daarin zijn de twee bovenste hoeken geel en de onderste roze. Door de manier waarop de punten bepaald werden (het doorlopen van een cirkel) zitten de gele en roze pixels alreeds bij elkaar. Dus moet enkel nog gecontroleerd worden of de twee gele punten de eerste twee elementen in de rij zijn. Is dit niet het geval wordt de lijst geroteerd totdat aan de voorwaarde is voldaan. Elk element wordt dan in die volgorde toegevoegd aan de dictionary waardoor de punten telkens dezelfde ordening zullen hebben.

2.2 Reconstrueren van een hoekpunt

Indien een hoekpunt moet gereconstrueerd worden m.a.w. de waarde van de desbetreffende sleutel in de dictionary van hoekpunten is *null*, wordt in de daarnet aangemaakte dictionary het bijhorende punt op de diagonaal geselecteerd. Als bijvoorbeeld de bijhorende waarde van LU *null* is, wordt in de dictionary van punten op de diagonalen ook de waarde van LU geselecteerd. Vanuit dit punt en het middelpunt kan de vergelijking voor een rechte opgesteld worden die de eerste constructielijn zal vormen. Een aanliggend hoekpunt met gekende coördinaat, hulphoek genaamd, zal door de voorwaarde van minstens twee aanliggende detecteerbare hoeken steeds gevonden worden. Vanuit deze worden weer de cirkelsegmenten van boorden en diagonaal bepaald, dit met de radius die berekend werd in 2.1. Met als grote verschil dat nu niet de middens moeten bepaald worden maar de twee uiterste punten. Dit zullen dan de pixels zijn die op de rand van het scherm liggen. Hiervoor wordt van elk cirkelsegment het eerste en laatste element opgeslaan. Dit is meteen ook het tweede pluspunt van het startpunt, het zorgt ervoor dat de eerste en laatste pixel van de lijsten altijd de buitenste punten van een boord of diagonaal zijn. Dit stelt het mogelijk de lijst waarover geïtereerd moet worden te reduceren tot 6 pixels (2 punten per segment, 2 boorden en 1 diagonaal). De twee pixels die het verst van elkaar gelegen zijn zoeken in deze lijst is veel efficïenter dan alle punten van de segmenten te moeten overlopen. Door de offset en andere opgestapelde afrondingen kan het mogelijk zijn dat kan het zijn dat op de rechte vanuit elke van deze punten naar de hulphoek pixels gekruist worden die niet dezelfde id hebben als dat punt van waaruit de rechte werd opgesteld. Indien dat het geval is wordt de dichtsbijzijnde pixel genomen op dat cirkelsegment waarvoor dit wel mogelijk is. Afhankelijk van de onderlinge ligging tussen het te reconstrueren hoekpunt en de hulphoek moet beslist worden welke van de twee berekende punten verder nodig zal zijn voor de reconstructie. Dit kan als volgt achterhaald worden. Ter referentie wordt eerst de rechte tussen hulp- en overstaande hoek van de te reconstrueren hoek bepaald. Dan wordt voor de twee te reduceren punten de rechte opgesteld met de hulphoek. Degene die de grootste hoek vormt samen met de referentie rechte, bevat het te behouden punt en

deze vormt dan ook de tweede constructielijn. De nieuwe hoek wordt dan bekomen door het snijpunt van de twee constructielijnen te bepalen **??** (*ref naar wikipedia intersectie class line). Om de id van de nieuw bepaalde hoek te bepalen, kan gekeken worden naar de tegenovergestelde hoek die reeds gekend zal zijn. Is die roze dan is de nieuwe hoek heel en vice versa.

3 Triangulatie

Wanneer de schermen herkent en geïdentificeerd zijn, worden ze aan elkaar gelinkt door middel van een triangulatie. Het project gebruikt een Delaunay triangulatie. Dit is een speciale vorm waarbij de kleinste hoek gemaximaliseert wordt en waarbij de driehoeken niet overlappen. [1] Er wordt gebruik gemaakt van het Bowyer-Watson algoritme. [2] Het heeft een tijdscomplexiteit van $O(n^2)$, dit is zeker niet de beste complexiteit om een Delaunay triangulatie te berekenen. Aangezien in de toepassing maximaal 120 schermen gebruikt kunnen worden, voldoet $O(n^2)$. Met de eenvoudige implementatie is dit dan ook een voordehandliggende keuze.

3.1 Bowyer-Watson

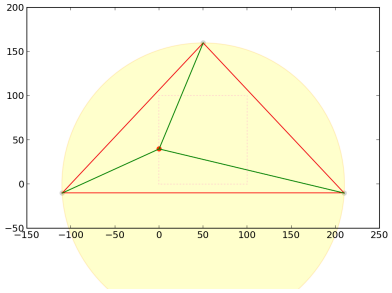
Bowyer-Watson gaat er van uit dat punten enkel worden toegevoegd in een al bestaande Delaunay triangulatie. Als eerste worden er twee superdriehoeken gezocht. Deze driehoeken zullen alle te trianguleren punten bevatten. De implementaties waarop het algoritme is gebaseerd [2] [3] stelden beiden een 'superdriehoek' voor, zie figuur 4a Echter is het eenvoudiger om een omkaderende vierhoek te vormen en deze op te delen in twee driehoeken. Vervolgens worden alle punten één voor één toegevoegd.

Voor elk punt worden alle driehoeken gezocht waarvan het punt in de omschreven cirkel zit. Wanneer twee driehoeken eenzelfde zijde delen, wordt deze verwijderd. Alle punten van de omschreven veelhoek van de twee driehoeken worden nu verbonden met het toegevoegde punt, zie figuur 4b. Met deze werkwijze zal er op elk moment een Delaunay triangulatie zijn en moeten de driehoeken achteraf niet meer overlopen worden. Met het gevolg dat het data management voor deze methode minder complex is.

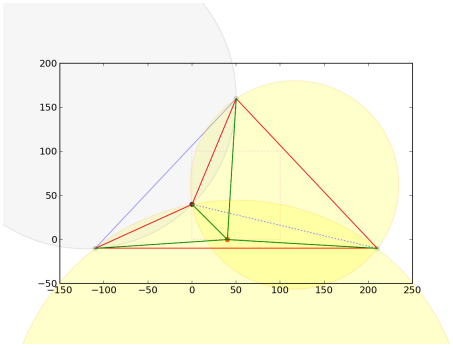
Eens alle punten toegevoegd zijn, worden de driehoeken die één of meer hoeken van de omkaderende vierhoek bevatten verwijderd, zie figuur 4c. Aangezien deze driehoeken aan de buitenkant liggen is dit toegestaan. Er zal een Delaunay triangulatie overblijven van alle onderzochte punten.

3.2 Valkuilen

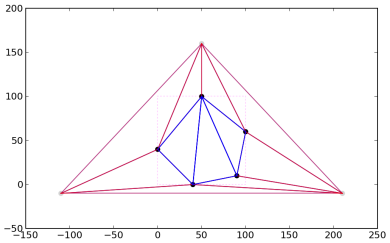
In theorie kan er van elke opstelling waarbij de punten niet allemaal colineair zijn een triangulatie worden gevonden, zie figuur 5. Echter door afronding bij de berekeningen zal er bij bijna colineaire punten geen juiste configuratie gevonden worden, zie figuur 6.



(a) De rode superdriehoek waarin alle punten zullen worden toegevoegd.



(b) Een nieuw punt wordt toegevoegd. In geel de omschreven cirkels. De gestippelde zijde wordt verwijderd, de groene toegevoegd.



(c) In rood alle driehoeken verbonden met de superdriehoek, deze worden uiteindelijk verwijderd.

Figuur 4: Het Bowyer-Watson algoritme [2]

Figuur 5: Van colineaire punten kan geen triangulatie gevonden worden.

Figuur 6: Van bijna colineaire punten kan geen triangulatie gevonden worden door afrondingsfouten bij berekeningen.

4 Animation

4.1 Sprites

De animatie van de kat wordt bekomen door het snel achter elkaar tekenen van delen van een sprite sheet. Deze sheet bevat in dit geval 8 frames, zie figuur 7. De breedte van de sheet zal door het aantal frames worden gedeeld om zo met een "draw"methode elke deel appart en achter elkaar te displayen. Op deze manier creëert men de illusie van beweging op een zeer simpele manier. In dit geval loopt de kat naar rechts maar met een spiegel methode kan de andere richting bekomen worden.

Figuur 7: Kat sprite sheet [**?**]

4.2 Delaunay

Voor de animatie wordt gebruik gemaakt van de reeds geschreven delaunay triangulatie. Deze zal het pad vormen waarover de kat zal lopen. Wanneer de kat zich dicht genoeg bij het endPoint bevindt zal deze de firstPoint worden en zullen de buren ervan opgevraagd worden. Er zal willeukeurig een punt worden gekozen als nieuw endPoint. Dit punt kan niet het firstPoint van de vorige beweging zijn. Dit wil zeggen dat de animatie nooit terug op zijn stappen komt. Enkel als de triangulatie een rechte lijn vormt zal de sprite op en neer lopen.

Referenties

[1] Delaunay triangulation. https://en.wikipedia.org/wiki/Delaunay_triangulation, October 2019.

[2] Bowyer-watson algorithm. https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm, October 2019.

[3] S.W. Sloan and G.T. Houlsby. An implementation of watson's algorithm for computing 2-dimensional delaunay triangulations. Technical report, Department of Civil Engineering, University of Newcastle and Department of Engineering Science, University of Oxford, 1984.

[4] Kat sprite sheet. https://docs.coronalabs.com/guide/media/spriteAnimation/index.html, December 2019.