

---

## Team 12

---

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

---

## Abstract

Het laten samenwerken van verschillende schermen van computers, tablets, smartphones en dergelijke met toegang tot een moderne web-browser, om samen één volledig beeld te vormen. Dit alles gecontroleerd door één master toestel. De toestellen zijn geconnecteerd via een webapplicatie en worden geconfigureerd door één foto te nemen met het delegerende toestel. Wel of geen overlap van toestellen, sommige schermen dicht en sommige verder van de camera, recht of gekanteld, het weergegeven resultaat moet vlak tonen vanuit het standpunt van de gebruiker.

Hiervoor moeten de verschillende toestellen eerst gedetecteerd en geïdentificeerd worden. Het volledige resultaat bereiken door het maken van één enkele foto via een standaard smartphone camera was vanaf de start prioritair. Dit zou het hinderlijke probleem van een inconsistentie camerapositie door het nemen van verschillende foto's meteen elimineren. Al snel werd gewerkt met voorbepaalde kleuren die de toestellen afbeelden. Hierop kan dan gefilterd worden met geassocieerde kleurenbreedtes. Op basis hiervan is het dan mogelijk de verschillende maskers af te leiden. Verdere algoritmes zoals het detecteren van de hoeken en het reconstrueren van afgedekte delen gebeuren aan de hand van deze maskers. De link tussen gevonden schermen en verbonden toestellen met de server, worden gelinkt door het lezen van een unieke kleuren-barcode die overeen komt met een gebruikers-id. Aan de hand van de bekomen toestelspecificaties bij het gedetecteerde scherm wordt er een perspectief matrix opgesteld om een correct resultaat op elk scherm af te beelden. De getransformeerde afbeelding zal zowel continuïteit als visuele correctheid over de verschillende schermen heen garanderen.

Tijdens het steeds verder uitbreiden van de grenzen van het algoritme is het duidelijk geworden dat kleurendetectie het grootste struikelblok was van de volledige methode. Deze aanpak viel al snel uit elkaar bij relatief kleine afwijkingen in de omgevingsbelichting en reflecties op de schermen. Enkel bij schermen en camera's die gebruikt werden voor het afstellen van de kleurdetectie werd een consistent resultaat verkregen. Een eenvoudige oplossing is om de barcode voor identificatie te beperken tot grijswaarden. Zo kan een grotere rijkweidte om de randen en diagonalen van het scherm te detecteren toegelaten worden. De ontworpen algoritmes voor hoekdetectie en reconstructie werken consistent bij een correct gevormd masker.

# Inhoudsopgave

<b>1 Inleiding</b>	<b>4</b>
<b>2 Framework</b>	<b>4</b>
2.1 Masters en clients . . . . .	4
2.2 Communicatie protocollen . . . . .	5
2.2.1 Commando's . . . . .	5
2.2.2 Video verzenden . . . . .	5
2.3 Beveiliging . . . . .	5
2.4 Uitbreidingsmogelijkheden . . . . .	6
<b>3 Synchronisatie</b>	<b>6</b>
3.1 Vertraging berekenen . . . . .	6
3.2 Kloksynchronisatie . . . . .	7
3.3 Aftelklok . . . . .	7
<b>4 Algemeen verloop algoritme</b>	<b>8</b>
<b>5 Detectie</b>	<b>9</b>
5.1 Kleuren . . . . .	9
5.1.1 Kleurmodellen en ruimtes . . . . .	9
5.1.2 Ranges in de HSL kleurruimte . . . . .	10
5.1.3 Verdere verbeteringen . . . . .	10
5.2 Flood fill . . . . .	11
5.3 Hoekpunten . . . . .	12
<b>6 Reconstructie</b>	<b>13</b>
6.1 Hulppunten op diagonalen . . . . .	14
6.2 Reconstrueren van een hoekpunt . . . . .	14
<b>7 Identificatie</b>	<b>15</b>
7.1 Barcode . . . . .	15
7.2 Verdere verbeteringen . . . . .	16
<b>8 Transformatie voor image-show</b>	<b>16</b>
8.1 transformatiematrix . . . . .	17
<b>9 Triangulatie</b>	<b>17</b>
9.1 Bowyer-Watson . . . . .	17
9.2 Valkuilen . . . . .	18
<b>10 Animation</b>	<b>19</b>
10.1 Sprites . . . . .	19
10.2 Delaunay . . . . .	19
10.3 Server/Client side . . . . .	20
<b>11 Besluit</b>	<b>20</b>
<b>Referenties</b>	<b>22</b>

<b>A</b>	<b>Overzicht algoritme</b>	<b>23</b>
A.1	Algemeen . . . . .	23
A.2	Hoekdetectie en filtering . . . . .	24
A.3	Reconstructie . . . . .	25
<b>B</b>	<b>Kleuren plots</b>	<b>26</b>
B.1	RGB plots . . . . .	26
B.2	Histogrammen . . . . .	27
B.3	HSL plots . . . . .	28
B.3.1	3D . . . . .	28
B.3.2	2D . . . . .	28

# 1 Inleiding

De wereld wordt alsmaar digitaler. Mensen ontmoeten elkaar digitaal, spreken elkaar digitaal, zien elkaar digitaal. Ook het buitenspelen lijkt verleden tijd, samen een spelletje spelen online is ook geen rariteit meer. De digitale wereld past zich in sneltempo aan en zo moeten de toepassingen meegroeien. Wanneer mensen elkaar online willen zien of online een spel tegen elkaar willen spelen, willen ze dit het liefst zo echt mogelijk meemaken. Dit door een zo groot mogelijk scherm. Niet iedereen beschikt echter over de capaciteiten om een 100" scherm aan te kopen. Een mogelijkheid die in dit verslag wordt besproken is het doen samenwerken van schermen. Eender welk scherm met een internetverbinding zou zo aan elkaar gekoppeld kunnen worden. Op de geconnecteerde schermen is het dan mogelijk de achtergronden in te stellen op een gekozen kleur, pijlen te tekenen, synchroon af te tellen vanaf een gekozen getal, een afbeelding of video te vertonen en zelf een "kat en muis" animatie weer te geven.

Het verslag gaat eerst dieper in op het framework van de online applicatie. Hoe communiceren master en clients? Welke protocollen worden gebruikt? Beveiliging en dergelijke worden hier beantwoordt [2](#). Vervolgens komt synchronisatie van de schermen aan bod in sectie [3](#), dit is van groot belang bij het synchroon laten aftellen. Hierna wordt de detectie van de schermen besproken in sectie [5](#), de schermen zullen gedetecteerd worden door één enkele foto. Wanneer schermen overlappen zullen er hoeken niet zichtbaar zijn. Daarom zal het scherm gereconstrueerd moeten worden. Hiervoor wordt verondersteld dat per scherm het middelpunt en twee aanliggende hoekpunten zichtbaar zijn. Het verloop hiervan wordt besproken in sectie [6](#). Hierna volgt nog de identificatie in sectie [7](#). Waar en hoe is het scherm gesitueerd en georiënteerd op de ingezonden afbeelding? Alsook de transformatie van de af te beelden objecten komen aan bod. Er wordt geëindigd met triangulatie en vervolgens een kleine animatie die al op de schermen vertoond kan worden. Dit om te tonen dat de schermen juist met elkaar verbonden zijn en er een vloeiende overgang mogelijk is.

## 2 Framework

In dit project wordt gebruik gemaakt van het *VueJS framework* voor de frontend. Hierdoor is het eenvoudiger om de code in verschillende bestanden te verdelen. Ook is het handig om hiermee de user interface te ontwerpen en implementeren.

### 2.1 Masters en clients

Bij het openen van de webpagina is er een knop om zich aan te melden bij het platform. Dit zorgt ervoor dat later een login systeem met wachtwoord gemakkelijk kan worden toegevoegd indien dit gewenst is. Hierna heeft men de keuze om master of client te worden, deze informatie wordt in het geheugen van de server opgeslaan.

**Kamers** Elke master heeft zijn eigen kamer waarbij clients zich kunnen aansluiten. Door het gebruik van kamers kunnen tegelijkertijd meerdere masters op verschillende plaatsen hun groep clients aansturen. Telkens wanneer een commando wordt verstuurd krijgen enkel de clients in die specifieke kamer dat signaal. Omdat deze functionaliteit later toevoegen voor moeilijkheden kan zorgen, is dit nu alreeds geïmplementeerd.

## 2.2 Communicatie protocollen

Om de communicatie tussen master en clients gestructureerd te laten verlopen, worden verschillende manieren gebruikt. Hieronder worden de verschillende gebruikte protocollen om data te verzenden uitgelegd.

### 2.2.1 Commando's

Elke client kan op de canvas van zijn scherm pijlen tekenen, de achtergrond van kleur laten veranderen of een afbeelding weergeven. De commando's om de schermen te controleren via de master worden allemaal doorgestuurd via *SocketIO*. Deze techniek is gekozen over het *http-protocol* omdat de hoeveelheid te verzenden data niet zo groot is en SocketIO ook op constante basis de client, master en server aangesloten houdt. Hierdoor worden alle berichten onmiddelijk ontvangen. Een voorbeeld van wat er wordt verstuurd word weergegeven in figuur 1. Enkel het 'payload' gedeelte wordt verstuurd naar de clients vanaf de server.

```
1  {
2    payload: {
3      type: "flood-screen",
4      data: {
5        command: [
6          {
7            type: "color"/"interval",
8            value: "[255,0,0] / "200" ] (integer or rgb list)
9          }
10        }
11      to: [user_id or "all"] (to single user or all users in
12        room)
13    }
14 }
```

Figuur 1: Voorbeeld van een JSON commando om schermen van kleur te laten veranderen, verzonden naar de server

### 2.2.2 Video verzenden

De video's worden niet direct verstuurd naar elke client om af te spelen. Dit zou een lange wachttijd vereisen vooraleer het video bestand helemaal verzonden is en ontvangen door elke client. Het uploaden van video is zo geïmplementeerd dat deze eerst naar de server wordt verstuurd via een HTTPS POST request (door middel van een HTML Form) en opgeslagen als een bestand op de server. Vervolgens wordt naar elke client de link doorsgestuurd vanwaar ze de video kunnen streamen via een HTML Video element. De video wordt dan gebufferd en kan onmiddelijk worden afgespeeld zonder dat het hele bestand al gedownload is.

## 2.3 Beveiliging

**HTTPS** Staat voor Hypertext Transfer Protocol Secure. Door dit protocol zijn alle berichten geëncrypteerd waardoor het voor buitenstaanders niet zomaar mogelijk is verzonden berichten te lezen .

**Login security** Naast HTTPS is het ook belangrijk dat een aangesloten gebruiker zich niet zomaar kan voordoen als een andere gebruiker. Of eender wie zomaar aan alle informatie kan. Om deze redenen hebben alle ingelogde gebruikers een geëncrypteerde cookie] die hun `gebruikers_id` bevat. Dit maakt het mogelijk om elke verschillende gebruiker te identificeren in de backend-server door deze te decrypteren. Hierdoor is het mogelijk om bepaalde server endpoints, met bijvoorbeeld video of foto bestanden, enkel aan ingelogde gebruikers bloot te stellen en niet aan andere gebruikers die geen inlog cookie hebben. Vervolgens is veiligheid ook een probleem bij sockets. Als sockets hun aansluiting verliezen worden ze automatisch opnieuw aangesloten. Hierbij veranderd enkel hun id. Om te kunnen zien tot welke client een socket toebehoort, moet elke gebruiker zijn `gebruikers_id` verzenden via de socket om deze aan zijn `gebruikers_id` te kunnen linken. Deze informatie wordt allemaal in de server opgeslaan.

## 2.4 Uitbreidingsmogelijkheden

**Login systeem** Momenteel is er een anoniem login systeem ingebouwd waar er geen wachtwoord voor vereist is. Doordat dit nu al ingewerkt zit, kan dit later makkelijk omgevormd worden naar een login met inloggegevens.

**Foto upload met https** Nu worden de foto's doorgestuurd via sockets. Later is het de bedoeling om net zoals bij video de afbeeldingen ook door te sturen via https. Zodat de master slechts één keer de foto moet uploaden en de clients dan zelf de foto aanvragen.

**SocketIO beveiliging** Om nu de socket te identificeren moet elke gebruiker zijn `gebruikers_id` versturen. Dit kan voor problemen zorgen als een mogelijke hacker een andere `gebruikers_id` verzend via de socket om zich als iemand anders voor te doen. Om dit te voorkomen kan een tijdelijk inlog wachtwoord aangemaakt worden dat de gebruiker kan verzenden om zich te identificeren via de socket.

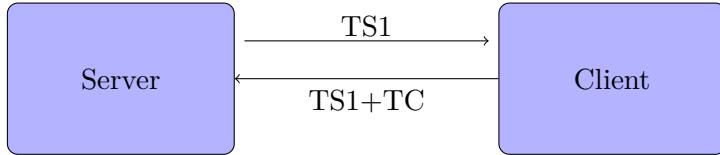
## 3 Synchronisatie

Bij een live applicatie is het belangrijk dat de aangesloten apparaten de mogelijkheid hebben synchroon een beeld weer te geven. Dit is niet altijd vanzelfsprekend dankzij verschillende factoren, zoals vertraging op en snelheid van een apparaat. Hiervoor zijn er verschillende technieken om toch synchronisatie te bekomen.

### 3.1 Vertraging berekenen

Er is een vertraging tussen een aangesloten client en de server, de *ping*. Dit is gemeten in miliseconden. Deze wordt gemeten door een bericht met de actuele tijd te verzenden van de server naar de client, en terug. De verzonden tijd wordt afgetrokken van de actuele tijd waarmee we de ping verkrijgen. In figuur 2 is de informatieoverdracht zichtbaar. In de server wordt de server tijd (TS1) berekend, verzonden naar de client en terug gekregen. Nu wordt de actuele tijd berekend in de server TS2. Dus de uiteindelijke ping is:

$$ping = TS2 - TS1$$



Figuur 2: Diagram van informatieoverdracht. (TS: Time Server, TC: Time Client)

---

```

1   {
2     type: 'count-down',
3     data: {
4       start: [integer]
5       interval: [integer, in ms],
6       startTime: [date in ms]
7     }
8   }

```

---

Figuur 3: countdown JSON commando verzonden naar clients

### 3.2 Kloksynchronisatie

Het is niet gegarandeerd dat de klok van de clients allemaal gesynchroniseerd zijn met de server. Daarom is het ook nodig om te weten wat het verschil is tussen de tijd aan de kant van de client en de server. Bij het terug verzenden van de client naar de server wordt de client tijd (TC) er bij het bericht gezet. Met deze TC en de berekende *ping*, is het mogelijk het tijdsverschil tussen de client en de server te bepalen (*DeltaTime*). Door dit verschil toe te voegen aan de servertijd is het mogelijk de correcte clienttijd te vinden. Dit wordt gebruikt om een starttijd te bepalen voor elke client dat op exact hetzelfde moment zal beginnen:

$$\text{DeltaTime} = (\text{TC} + \text{ping}/2) - \text{TS2}$$

Zo is de tijd van de client ten opzichte van de server altijd:

$$\text{TimeClient} = \text{TimeServer} + \text{DeltaTime}$$

### 3.3 Aftelklok

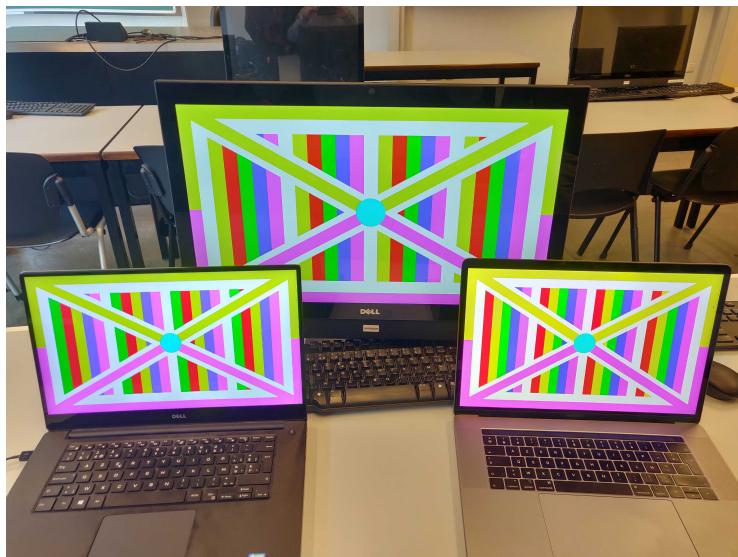
Bij de naïve implementatie van de aftelklok is er gebruikt gemaakt van de *setTimer()* functie die recursief een getal aftelt en tekend op een canvas. Een probleem hierbij is dat apparaten niet even snel het getal kunnen tekenen op het scherm waardoor er apparaten kunnen zijn die sneller zijn dan anderen. Daarom is er gebruik gemaakt van *setInterval()* dat periodisch het getal berekend, relatief ten opzichte van de meegegeven starttijd. Als er een client trager is en niet optijd op het scherm kan tekenen, dan zal het getal worden overgeslagen omdat het apparaat het sowieso niet zou aankunnen. Hierdoor blijven de getallen op het scherm synchroon en zal het aftellen ook op hetzelfde moment stoppen op elk scherm. Het getal is als volgt berekend:

$$\text{number} = \text{startNum} - \text{Math.floor}((\text{actualTime} - \text{startTime})/\text{interval})$$

## 4 Algemeen verloop algoritme

Er gaat heel wat vooraf aan het weergeven van een afbeelding, video of animatie op alle geconnecteerde toestellen. In bijlage A.1 is een algemeen overzicht te vinden.

Eens de gewenste opstelling bekomen is, kan de master het commando geven om detectieschermen weer te geven op de toestellen. Dit wordt weergegeven op figuur 4. Deze configuratie wordt nu vastgelegd door de master en geüpload om te analyseren. Indien nodig wordt de afbeelding eerst herschaald volgens 1920x1080 pixels om performantie te bevorderen. Ook wordt van *RGBA* naar *HSLA* spectrum gegaan. Vervolgens wordt er gefilterd op de twee randkleuren van het detectiescherm. Hieruit volgen de zogenaamde *islands* die gefilterd worden tot er enkel mogelijke schermen overblijven.



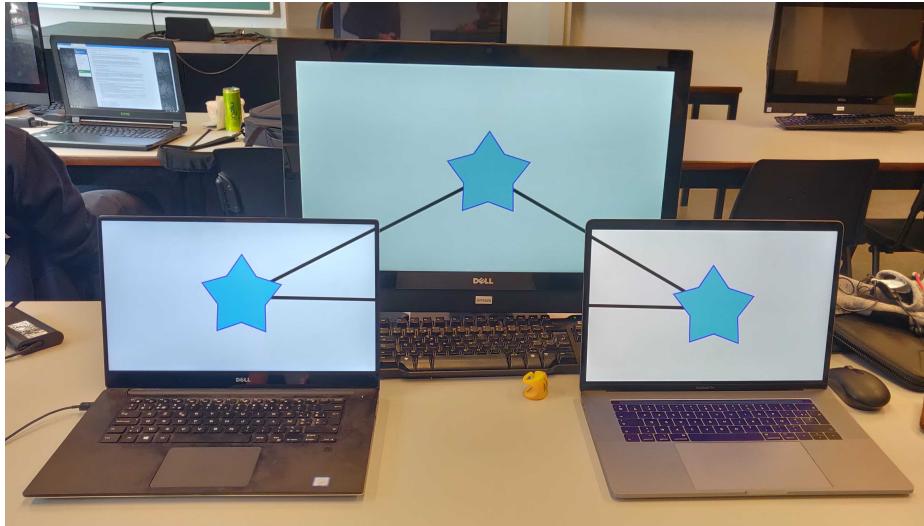
Figuur 4: Opstelling waarvan een foto wordt genomen.

**Hoekdetectie** Vervolgens zoekt het algoritme naar hoeken, zie bijlage A.2. Er wordt gekeken of het scherm gekanteld of recht staat en met deze info het bijpassende hoekdetectie algoritme toegepast. Hierna worden de hoeken gefilterd en zijn de hoeken gedetecteerd. Als er niet genoeg hoeken gevonden zijn, is het geen geldig *island*.

**Hoekreconstructie** In bijlage A.3 en deel 6, staat het hoekreconstructie gedeelte gedetailleerder beschreven. Dit wordt uitgevoerd wanneer er niet genoeg hoeken zijn herkend voor een scherm, maar wel genoeg om te reconstrueren. Het gaat de missende hoeken reconstrueren door de lijnen vanuit de reeds gevonden hoeken te volgen.

Met alle hoeken en middelpunt op zak is er een scherm gevonden en wordt het geïdentificeerd met behulp van een barcode. Het scherm wordt toegewezen aan een cliënt. Het zoeken gaat verder tot alle cliënts gevonden zijn of alle pixels uit de afbeelding zijn overlopen.

**Triangulatie** Na de analyse wordt het resultaat getoond aan de hand van de Delaunay triangulatie, weergegeven in figuur 5. Schermen die niet gedetecteerd werden, worden hierin niet betrokken.



Figuur 5: opstelling na detectie van de schermen.

## 5 Detectie

### 5.1 Kleuren

Het onderscheiden en detecteren van verschillende kleuren speelt een belangrijke rol in detectie van de schermen. De waarneming van een kleur aan de hand van een foto stemt echter niet altijd overeen met de afgebeelde kleur op een scherm. De voornaamste oorzaken hiervan zijn lichtinval en reflectie. Tijdens detectie moet er dus rekening gehouden worden met deze factoren. De keuze van de kleurruimte zal hierbij essentieel zijn om een goede range op te stellen voor detectie van de verschillende kleuren.

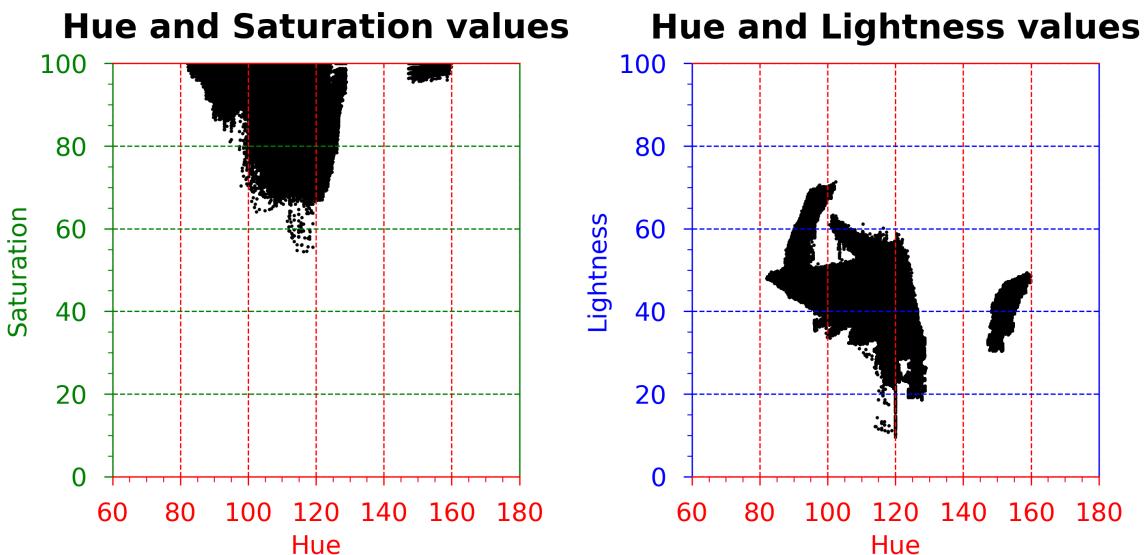
#### 5.1.1 Kleurmodellen en ruimtes

Door de jaren heen zijn veel verschillende modellen ontwikkeld om het visuele lichtspectrum weer te geven. Al deze modellen hebben hun voordelen en nadelen en worden dan ook voor verschillende doeleinden gebruikt. De meest voorkomende modellen zijn RGB, CIE en HSL/HSV. Het RGB model is een additief kleurmodel waarbij een kleur wordt beschreven aan de hand van de drie primaire kleuren: rood, groen en blauw. Elke kleur wordt gevormd aan de hand van een combinatie van deze drie kleuren. RGB wordt heel veel gebruikt in grafische toepassingen. Wanneer een foto door een computer wordt uitgelezen zal dit ook in RGB waarden gebeuren. Alleen zal dit model jammer genoeg niet gebruikt kunnen worden voor detectie aangezien het model een niet-lineaire en discontinue ruimte vormt. Deze discontinuiteit maakt het beschrijven van een verandering in tint moeilijk. Daarnaast is het RGB model gevoelig aan verandering van licht, wat resulteert in een verandering van tint. Het CIE model is gemaakt door de Commission Internationale de l'Éclairage (CIE). Het was het eerste model dat kleuren wiskundig kan voorstellen. Hierbij wordt gebruik gemaakt van de link tussen de verschillende golflengtes van het visueel spectrum en de manier waarop mensen kleuren waarnemen. Het CIE model lag ook aan de basis van bijna alle andere kleurmodellen die achteraf ontwikkeld zijn. Het grootste nadeel van het gebruik van dit model is de complexe omzetting van RGB naar CIE. Om deze reden wordt dit model niet gebruikt voor de detectie. HSL en HSV maken allebei deel uit van het cylindrisch model. Deze worden beschreven in drie dimensies: een hoek, die de tint

voorstelt gaande van  $0^\circ$ (rood), naar  $120^\circ$ (groen), richting  $240^\circ$ (blauw), om uiteindelijk bij  $360^\circ$ (rood) rond te zijn. Een horizontale dimensie, die de saturatie beschrijft en een verticale dimensie, die de lichtheid (HSL) of waarde (HSV) bepaalt. Het voornaamste voordeel en de reden voor het gebruik van deze modellen over RGB en CIE is het feit dat deze modellen immuun zijn aan veranderingen in licht. Want deze zitten in een aparte dimensie. Een ander voordeel is de continue tint in de HSV en HSL modellen. HSL wordt uiteindelijk boven HSV verkozen door zijn symmetrie voor licht en donker. [1] [2]

### 5.1.2 Ranges in de HSL kleurruimte

Naast de keuze van het kleurmodel, HSL, moeten ook de kleuren voor detectie bepaald worden. Uiteindelijk heeft onze gebruikte methode zes verschillende kleuren nodig om te detecteren. De keuze is gegaan naar de drie basiskleuren (rood, groen en blauw) aangevuld met de kleuren, die theoretisch het verst van elkaar verwijderd liggen op de cirkel (cyaan, magenta en geel). Bijgevolg liggen de gekozen kleuren allemaal op  $60^\circ$  van elkaar verwijderd. Vervolgens werd voor elke kleur een range bepaald onder verschillende omstandigheden. Hiervoor zijn foto's genomen van verschillende schermen in verschillende lichtomstandigheden en reflecties met verschillende camera's. Deze foto's dienen als dataset om de range van elke kleur te bepalen. Elke pixel van een foto werd uitgelezen en nadien omgezet van een RGB waarde naar een HSL waarde. Deze HSL waarden zijn vervolgens geplot in functie van tint en saturatie, alsook in functie van tint en lichtheid, zie figuur B.24. De bekomen scatterplots tonnen telkens een geconcentreerd gebied van datapunten aan dat aan de hand van lineaire functies begrensd werd. Deze lineaire functies worden dan gebruikt als range om de kleur te detecteren. Dit werd voor alle zes kleuren uitgevoerd. [3]



Figuur 6: Scatter plots in functie van tint en saturatie, alsook in functie van tint en lichtheid voor de kleur groen.

### 5.1.3 Verdere verbeteringen

Uit de plots is echter gebleken dat aparte onderscheiding van zes verschillende kleuren geen goede optie is door overlap van de ranges van de verschillende kleuren. Dit is het gevolg van zeer verschillende waarnemingen van een kleur naargelang de omstandigheden waarin

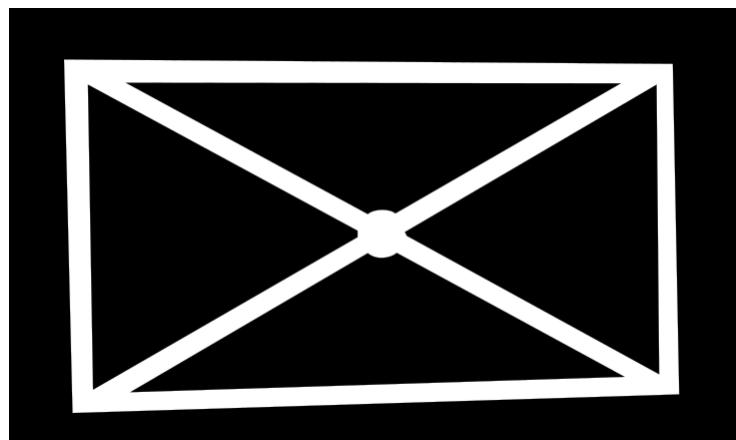
een foto getrokken wordt. Het is dus niet mogelijk om zes verschillende kleuren correct te onderscheiden in alle omstandigheden. Nieuwe methodes moeten gevonden worden waar niet zoveel verschillende kleuren van elkaar onderscheiden moeten worden. Een eerste mogelijkheid is om een detectie uitgaande van slechts drie kleuren te verkiezen boven een detectie met zes kleuren. Bij het gebruik van een drie-kleuren-detectie zijn twee opties, waarbij de kleuren opnieuw zo ver mogelijk van elkaar verwijderd liggen ( $120^\circ$ ), mogelijk. Eenderzijds het gebruik van de kleuren rood, groen en blauw of anderzijds de kleuren cyaan, magenta en geel. De uiteindelijke keuze valt op rood, groen en blauw aangezien deze minder ver fluctueren van de theoretische waarde dan cyaan, magenta en geel. Aan deze optie wordt reeds gewerkt om de detectie te verbeteren. Daarnaast is het ook een optie om eerder relatief naar het contrast tussen kleuren te kijken in plaats van detectie op basis van absolute ranges. Ook deze optie wordt reeds bekeken om de identificatie te verbeteren.

## 5.2 Flood fill

Elk individueel slave scherm laat een vooraf bepaalde afbeelding zien met gekende kleurwaarden (\*TODO: afbeelding van html). Deze afbeelding 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 relatieve oriëntatie van het scherm op de foto. Om deze schermen te detecteren wordt de foto gefilterd op basis van gekende HSL ranges (zie 5.1.2) en een standaard four-way flood fill algoritme [4 way afbeelding of te veel?] [4] 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 per eiland opgeslagen in elke pixel, waar de verdere bewerkingen op uitgevoerd zullen worden. Het geïmplementeerde floodfill 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.



Figuur 7: Kleurenmasker van een scherm na floodfill

Niet enkel de associatie van de gemaskeerde pixels wordt op deze manier behouden, maar deze methode maakt ook dat de komende pixel bewerkingen maar over een minimale bounding box uitgevoerd worden ten opzichten van de volledige pixel matrix. Elk resulterend resultaat van floodfill geeft een verbonden pixel verzameling, een island, als resulterend. De resulterende islands worden achteraf gefilterd opdat elk eiland de drie kleuren van de border en het middelpunt bevatten. Als er aan deze voorwaarde voldaan is wordt er een poging gedaan om lokaal een middelpunt en geldige barcode te lezen [7.1](#). Bij het falen van een van deze verificatiestappen wordt het eiland verworpen, de overblijvende eilanden zijn geldige eilanden voor verdere hoekdetectie.

### 5.3 Hoekpunten

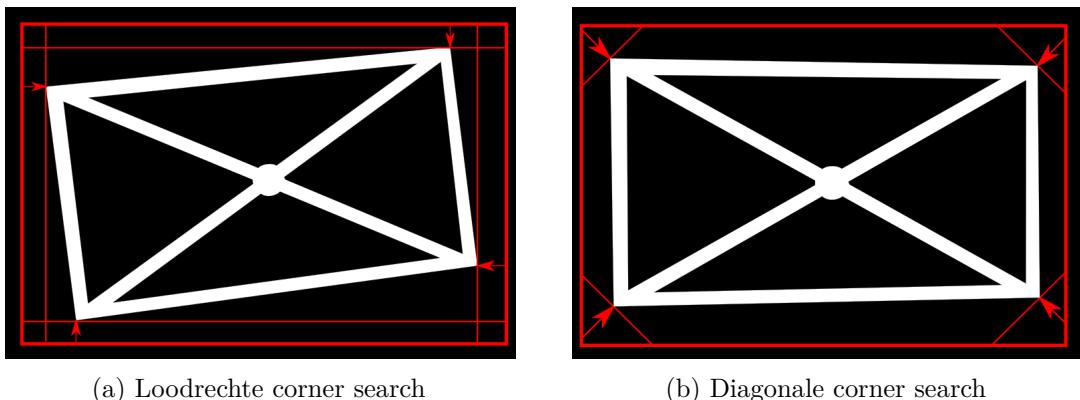
Er werd vooraf een algemeen hoek-detectie algoritme Shi-Tomashi [\[5\]](#) geïmplementeerd en getest, maar gaf een te complex resultaat op de binaire maskers om de juiste hoeken te filteren. Dit algoritme draagt ook een relatief grote overhead door de x- en y-sobel operaties die elks een volledige pixel convolution over alle pixels als pre-processing toepassen. Onze algoritmes zijn veel simplistischer en zullen in de praktijk nooit verder dan de boundary van een eiland uitgevoerd worden in tegenstelling tot de volledige island matrix, maar geven een perfect bruikbaar resultaat voor onze noden.

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 de standaarddeviatie van pixels in het masker berekend. Bij een standaardafwijking onder de 15% wordt een scherm als liggend of verticaal gezien op de foto en niet gekanteld.

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 (zie figuur: [8a](#)), maar volgens een diagonaal tot een mask-pixel (zie figuur: [8b](#)). Beide variante hoekdetectie algoritmes lopen in worst-case scenario volgens

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

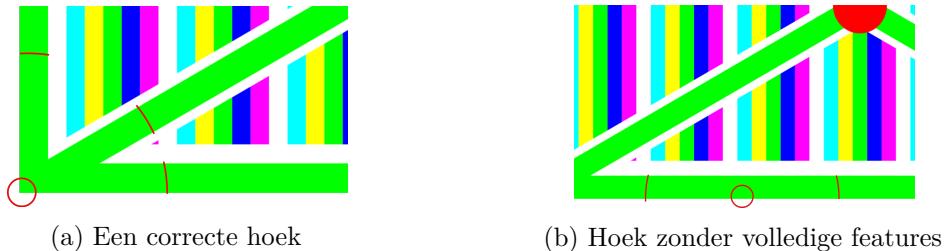
maar zoals eerder vermeld zullen deze in de praktijk maar tot de boundary van het masker lopen.



Figuur 8: Hoekdetectie

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 (zie figuur: 9a). Deze lijnen zijn bepaald door te filteren door de border- en diagonaal-kleur die gescheiden zijn door een witte rand. Deze aanwijzingen moeten gevonden worden binnen een relatieve straal die gelijk is aan 25% van de maximale afstand van de hoekpunten tot het middelpunt. Als in een eerder gevonden hoek deze voorwaarden niet aanwezig zijn (zie figuur: 9b), wordt deze hoek verworpen als resultaat van de hoek detectie.



Figuur 9: Hoekfiltering

## 6 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 schermen elkaar 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.

**Algoritme** 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.

## 6.1 Hulppunten op diagonalen

Telkens wanneer reconstructie nodig is zullen 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 gedefinieerd 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.

**Startpunt** Het startpunt van waaruit de cirkel doorlopen wordt, is een pixel die niet tot een diagonaal behoort. 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 controleert 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, betekent 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.

**Ordenen** 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.

## 6.2 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 6.1. 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 verstuif van elkaar gelegen zijn zoeken in deze lijst is veel efficiënter 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 geel en vice versa.

## 7 Identificatie

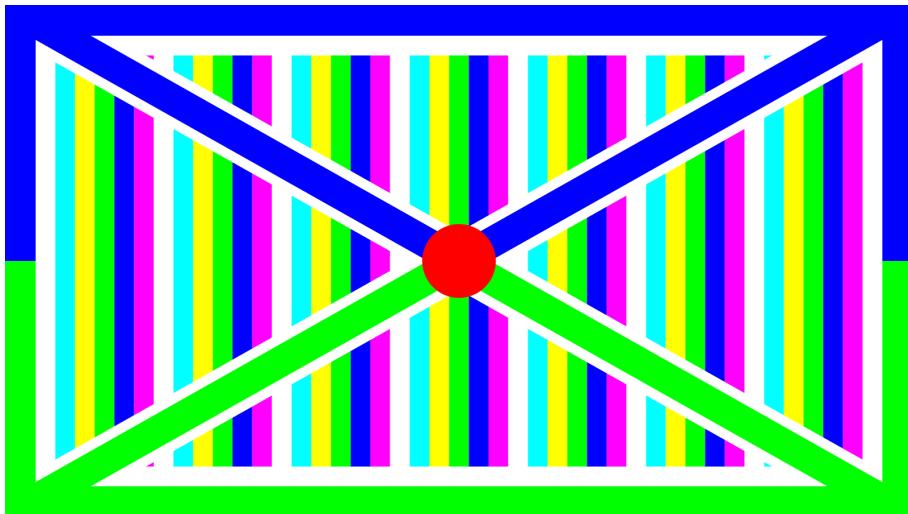
### 7.1 Barcode

Om de verschillende schermen te identificeren wordt gebruik gemaakt van een kleuren barcode. De barcode bestaat uit een herhalend patroon van 5 unieke kleuren telkens gevolgd door een witte lijn. Door het gebruik van deze witte lijn weet het algoritme waar het patroon eindigt en de volgende sequentie terug opnieuw begint. Het detecteren van deze 5 kleuren gebeurt aan de hand van opgestelde HSL ranges, zie 5.1.2. Voor de identificatie van de slaves wordt dus een unieke combinatie van deze 5 kleuren weergegeven, zie figuur 10. Deze vormt dan een unieke vijfcijferige code, die gelinkt kan worden aan de bijhorende slave. Dit zorgt ervoor dat we in theorie een totaal van  $5! (= 120)$  verschillende schermen op één moment kunnen detecteren. Herhaling van het patroon heeft als resultaat dat bij overlap het scherm nog steeds geïdentificeerd kan worden. Het algoritme zal twee keer over alle pixels itereren. Hierdoor heeft het algoritme een tijdscomplexiteit van

$$O(2mn) = O(mn)$$

met m en n de dimensies van het eiland waarin de barcode gelezen wordt. Het algoritme gaat een keer horizontaal en een keer verticaal over de pixels. Op deze manier kan de barcode in alle mogelijke orientaties van het scherm gelezen worden. Vervolgens worden de HSL waarden van deze pixels bekeken om de overeenkomstige kleur van elke pixel te achterhalen. Wanneer een HSL waarde binnen de gewenste range valt, wordt het overeenkomstig cijfer opgeslaan in een lijst. Bij het bereiken van een witte lijn weet het algoritme dat het aan het einde van het patroon is. Wanneer dit het geval is, wordt het inlezen van de volledige vijfcijferige code gecontroleerd op volledigheid. Zo niet, wordt de lijst leeg gehaald en zoekt

het algoritme verder. Het herhalend patroon is dus essentieel aan het correct inlezen van de barcode. Een groter aantal herhalingen stemt overeen met een hogere kans op mogelijke detectie, maar stemt ook overeen met een kleinere oppervlakte per herhaling. Deze kleinere oppervlakte is dan weer nadelig voor detectie. Aangezien hiermee de kans stijgt dat een kleur niet gedetecteerd wordt. Nadat het algoritme over alle pixels geweest is, wordt de ratio berekend tussen de code die het meeste keer gelezen is en de totale aantal codes die zijn gelezen. Deze ratio bepaalt dan of de code, die door de horizontale iteratie of door de verticale iteratie het meest gelezen is, gebruikt wordt.



Figuur 10: Scherm met randen en kruis voor detectie met barcode erachter voor identificatie.

## 7.2 Verdere verbeteringen

Zoals reeds vermeld in 5.1.3 is het individueel detecteren van zoveel verschillende kleuren geen goed idee. Dit is dan ook de reden dat andere opties reeds bekeken worden. Zoals ook reeds vermeld zou een eerste optie zijn om te kijken naar het contrast tussen de opeenvolgende kleuren in plaats van de kleuren apart te detecteren. Een andere optie die bekeken wordt is het overschakelen naar een zwart-wit barcode waarbij gebruik gemaakt wordt van het contrast tussen zwart en wit. Een andere mogelijke verbetering is het aantal herhalingen afhankelijk maken van de grootte van het scherm. Dit heeft als voordeel dat het aantal herhalingen geoptimaliseerd is voor de grootte van het scherm.

voor image-show

## 8 Transformatie voor image-show

Alle schermen moeten een deel van een foto weergeven. Echter doordat de schermen drie-dimensionaal gedraaid zijn, moet de weer te geven foto getransformeerd worden. De afbeelding moet namelijk vanuit perspectief twee-dimensionaal zijn afgebeeld. Het gebruikte algoritme gaat vier bron- en bestemmingshoeken gebruiken als invoer om tot een transformatie-matrix te komen. Deze matrix wordt gebruikt om later met *image3d transform* een snelle transformatie op een canvaselement mogelijk te maken.

## 8.1 transformatiematrix

De vier bron- en bestemmingshoeken zijn respectievelijk de start- en eindhoeken. Er zal per lijst van hoeken een matrix worden berekent nadat de coefficiënten berekend zijn geweest.  $(x_i, y_i)$  zijn de hoekpunten,  $i = 1..4$  voor de start-hoeken en  $i = 5..8$  voor de eind-hoeken. [6]

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda \\ \mu \\ \tau \end{bmatrix} = \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} \lambda \\ \mu \\ \tau \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix}$$

Daarna wordt de 3x3 matrix geschaald met de gevonden coefficiënten.

$$\begin{bmatrix} \lambda x_1 & \mu x_2 & \tau x_3 \\ \lambda y_1 & \mu y_2 & \tau y_3 \\ \lambda & \mu & \tau \end{bmatrix}$$

De twee matrices A en B, respecitivelijk met de bron- en bestemmingshoeken, worden gebruikt om de transformatiematrix te berekenen.

$$C = AB^{-1}$$

$C$  is de transformatiematrix, deze wordt later gebruikt om de *matrix3d* op te stellen. [7]

$$\left\{ \begin{array}{cccc} C_{0,0} & C_{0,1} & 0 & C_{0,2} \\ C_{1,0} & C_{1,1} & 0 & C_{1,2} \\ 0 & 0 & 1 & 0 \\ C_{2,0} & C_{2,1} & 0 & C_{2,2} \end{array} \right\}$$

voor image-show

## 9 Triangulatie

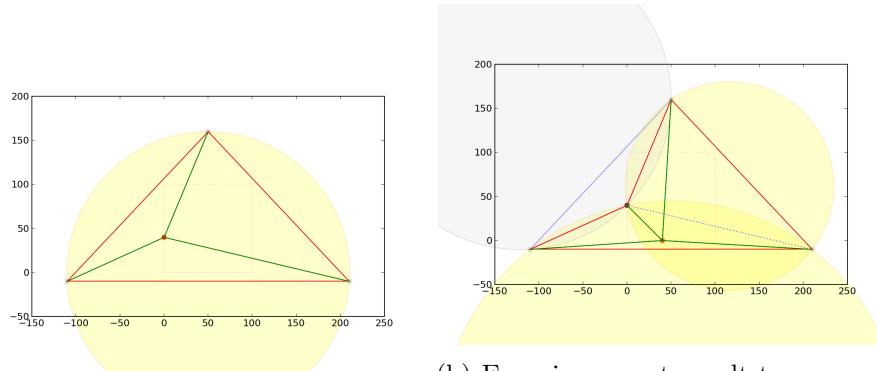
Wanneer de schermen herkent en geïdentificeerd zijn, worden ze aan elkaar gelinkt door middel van een triangulatie. Het project gebruikt een Delaunay triangulatie. Dit is een speciale vorm waarbij de kleinste hoek gemaximaliseert wordt en waarbij de driehoeken niet overlappen. [8] Er wordt gebruik gemaakt van het Bowyer-Watson algoritme. [9] 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.

### 9.1 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 [9] [10] stelden beiden een ‘superdriehoek’ voor, zie figuur 11a. 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 11b. 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 11c. Aangezien deze driehoeken aan de buitenkant liggen is dit toegestaan. Er zal een Delaunay triangulatie overblijven van alle onderzochte punten.

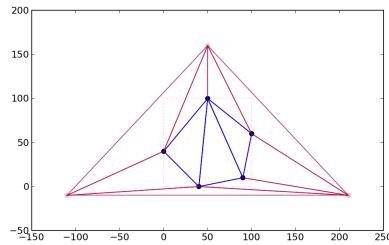


(a) De rode superdriehoek waarin alle punten zullen worden toegevoegd.

In geel de omschreven cirkels. De gestippelde zijde wordt verwijderd, de groene 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 11: Het Bowyer-Watson algoritme [9]

## 9.2 Valkuilen

In theorie kan er van elke opstelling waarbij de punten niet allemaal colineair zijn een triangulatie worden gevonden, zie figuur 12. Echter door afronding bij de berekeningen zal er bij bijna colineaire punten geen juiste configuratie gevonden worden, zie figuur 13.

.....

Figuur 12: Van colineaire punten kan geen triangulatie gevonden worden.



Figuur 13: Van bijna colineaire punten kan geen triangulatie gevonden worden door afrondingsfouten bij berekeningen.

## 10 Animation

### 10.1 Sprites

De animatie van de kat of muis wordt bekomen door het snel achter elkaar tekenen van delen van een sprite sheet. Deze sheet bevat in dit geval 7 frames, zie figuur 14 en 15. De breedte van de sheet zal door het aantal frames worden gedeeld om zo met een "draw"methode elke deel apart en achter elkaar te displaynen. Op deze manier creërt men de illusie van beweging op een zeer simpele manier. In dit geval loopt de kat of muis naar rechts maar met een spiegel methode kan de andere richting bekomen worden. De sprite zal ook worden geroteerd om in de correcte richting te bewegen. Meerdere muizen zullen achter elkaar lopen zodat op verschillende schermen sprites worden afgebeeld, zie figuur 16. Dit werkt aan de hand van een stack waarin zich de vorige posities van de eerste kat bevinden. Na de laatste "draw"methode zal het eerste element van de lijst worden verwijderd. Alle dieren zullen dus hetzelfde pad nemen.



Figuur 14: Kat sprite sheet [11]



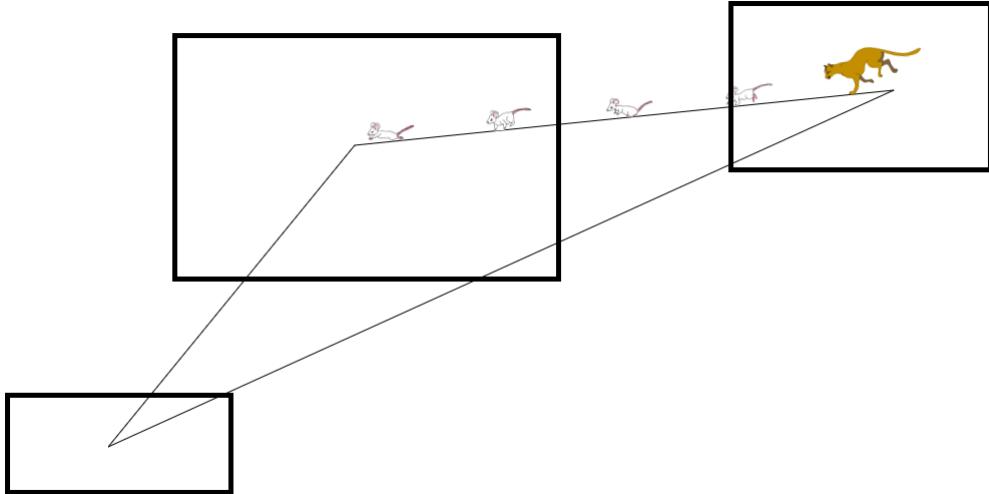
Figuur 15: Muis sprite sheet [12]

### 10.2 Delaunay

Voor de animatie wordt gebruik gemaakt van de reeds geschreven delaunay triangulatie. Deze zal het pad vormen waarover de kat of muis zal lopen. Wanneer het dier zich dicht genoeg bij het endPoint bevindt zal deze de firstPoint worden en zullen de buren ervan opgevraagd worden. Er zal willekeurig 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.

### 10.3 Server/Client side

De server zal simpelweg de basis info doorgeven aan de client, zelf de berekeningen maken voor de volgende posities en de triangulatie bijhouden. De client krijgt enkel een dictionary met de x,y positie, de hoek, de frame en een boolean voor spiegeling mee en heeft geen weet van de triangulatie.



Figuur 16: Theoretische animatie met verschillende schermen

## 11 Besluit

Er is een applicatie ontwikkeld waarin een master verschillende cliënt schermen kan detecteren en manipuleren. Het framework houdt rekening met uitbereiding naar verschillende *kamers*. Voor video wordt er gestreamd in plaats van volledig gedownload. Met behulp van synchronisatie zal een video of een aftelklok gelijk lopen op elk scherm. Enkel bij uitzonderlijke gevallen waarbij de *ping* toevallig fout wordt gemeten (door bv. actieve achtergrondprocessen) kan er een synchronisatiefout optreden.

Het master device kan vanuit één enkele foto alle schermen herkennen. Hierbij wordt gebruik gemaakt van een bepaald herkenningspatroon. Bij sterke reflectie, grote 3d-draaiing en verschillende kleurweergave van de schermen kan het mislopen. Schermen worden dan niet herkent en zullen dus niet kunnen samenwerken met de herkende schermen. Een oplossing wordt hiervoor gezocht in de vorm van kleurverschil i.p.v. kleurdetectie. Hiervoor zijn er al plots opgesteld om de reikwijdte van de verschillende kleuren te determineren en zo de kleurschema's beter en efficiënter te gebruiken.

Het reconstrueren van schermen, wanneer er een deel van het scherm wordt bedekt, vertrouwd er op dat er 2 aanliggende hoeken en het middelpunt wordt gevonden. Met behulp van een transformatiematrix wordt de ligging en draaiing van het scherm bepaald. Hierdoor kan een afbeelding met juiste transformate geprojecteerd worden op elk scherm. Deze heeft echter vier hoeken nodig en bouwt verder op de reconstructie van het scherm.

Als laatste is er nog de animatie die zeer nauw samenhangt met de triangulatie. Hiervoor is een Delaunay triangulatie gekozen omdat deze uniek is, niet overlappend en voor een maximale kleinste hoek streeft. Wanneer alle punten (bijna) colineair zijn, zal er door afronding geen triangulatie gevonden worden, de animatie kan dan ook niet worden afgespeeld.

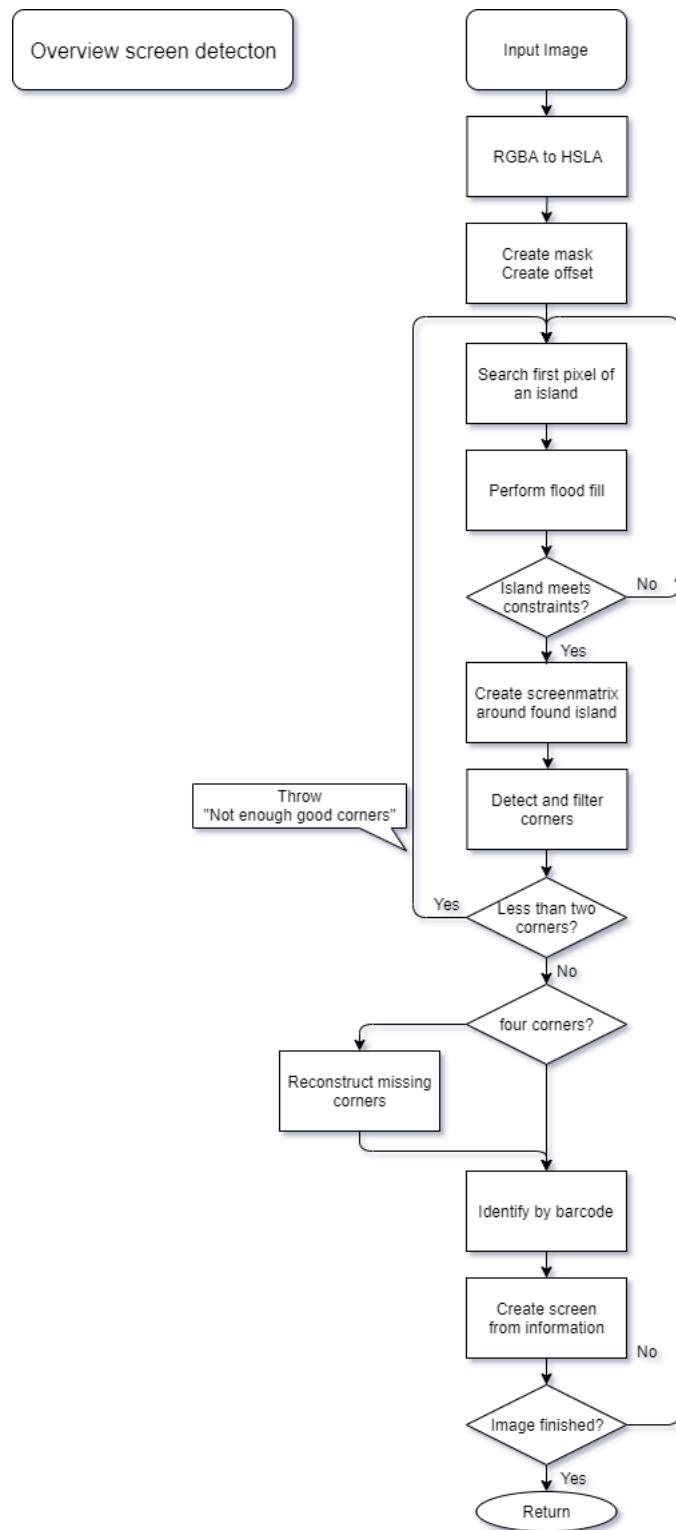
De applicatie heeft een brede waaier aan functionaliteiten. Doordat er maar één foto nodig is voor de schermen te herkennen, is het plaatsen van de schermen nauwkeurig maar kan de identificatie sneller mislopen. Alle opdrachten, exclusief de cat caster, zijn geïmplementeerd. Met enkel nog een paar kleine *bugs*, waarnaar gestreefd wordt deze op te lossen, is er spraken van een geslaagde opdracht.

## Referenties

- [1] Edgar Chavolla, Daniel Zaldivar, Erik Cuevas, and Marco Cisneros. *Color Spaces Advantages and Disadvantages in Image Color Clustering Segmentation*, pages 3–22. 01 2018.
- [2] Amir Rasouli and John K Tsotsos. The effect of color space selection on detectability and discriminability of colored objects. *arXiv preprint arXiv:1702.05421*, 2017.
- [3] Shun-Hung Tsai and Yu-Hsiang Tseng. A novel color detection method based on hsl color space for robotic soccer competition. *Computers & Mathematics with Applications*, 64(5):1291 – 1300, 2012. Advanced Technologies in Computer, Consumer and Control.
- [4] Flood fill. [https://en.wikipedia.org/wiki/Flood\\_fill](https://en.wikipedia.org/wiki/Flood_fill). geraadpleegd oktober 2019.
- [5] Harris corner detection and shi-tomasi corner detection. <https://medium.com/pixel-wise/detect-those-corners-aba0f034078b>. Geraadpleegd november 2019.
- [6] stack overflow. Redraw image from 3d perspective to 2d. <https://stackoverflow.com/questions/14244032/redraw-image-from-3d-perspective-to-2d>. Geraadpleegd november 2019.
- [7] StackExchange. Finding the transform matrix from 4 projected points (with javascript). <https://math.stackexchange.com/questions/296794/finding-the-transform-matrix-from-4-projected-points-with-javascript>. Geraadpleegd november 2019.
- [8] Delaunay triangulation. [https://en.wikipedia.org/wiki/Delaunay\\_triangulation](https://en.wikipedia.org/wiki/Delaunay_triangulation). Geraadpleegd novermber 2019.
- [9] Bowyer-watson algorithm. [https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson\\_algorithm](https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm). Geraadpleegd okotber 2019.
- [10] S.W. Sloan and G.T. Housby. 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. Geraadpleegd november 2019.
- [11] Kat sprite sheet. <https://docs.coronalabs.com/guide/media/spriteAnimation/index.html>. Geraadplaagd december 2019.
- [12] Kat sprite sheet. <https://i.imgur.com/AUVnIPz.gif>. Geraadplaagd december 2019.

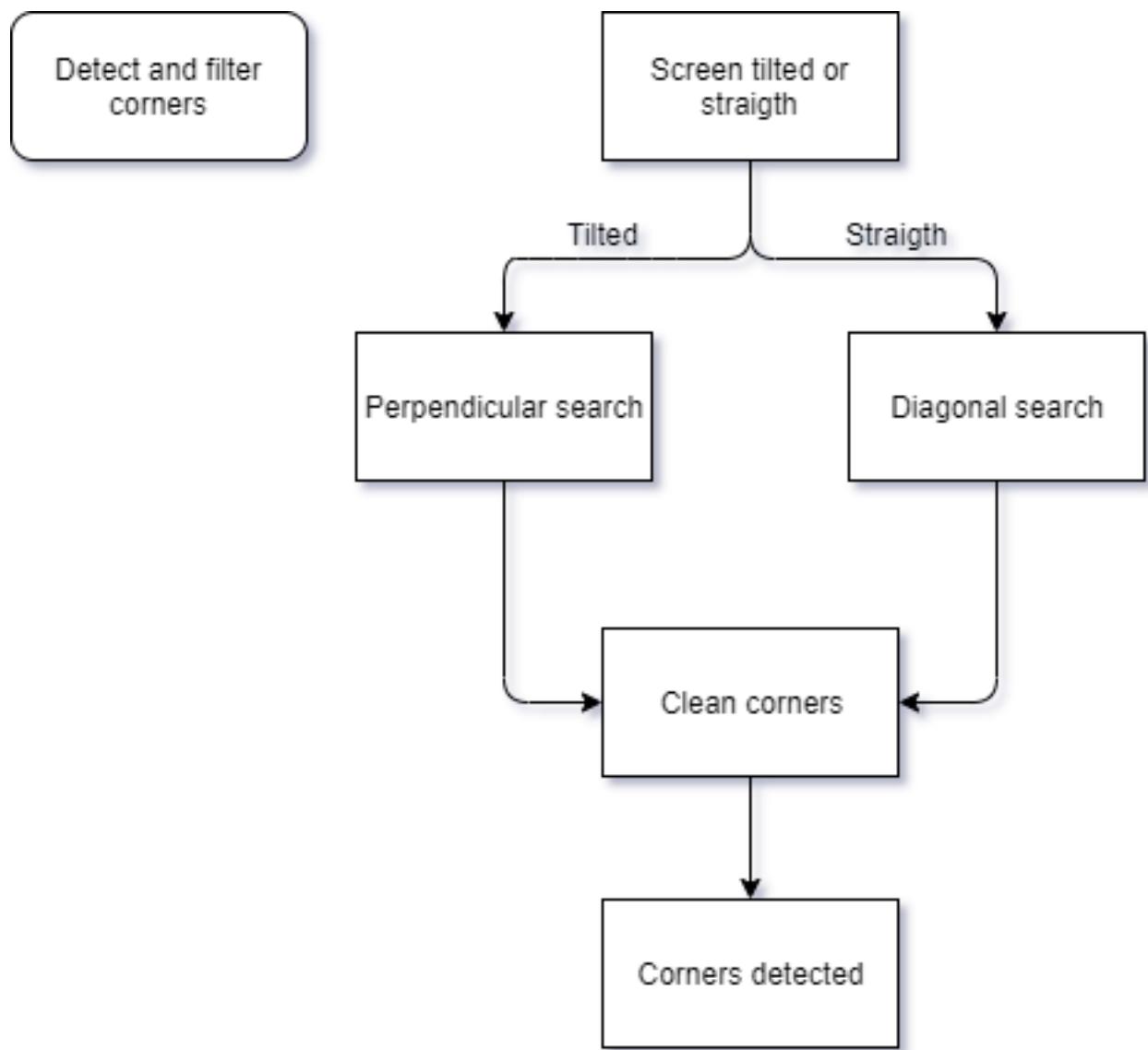
## A Overzicht algoritme

### A.1 Algemeen



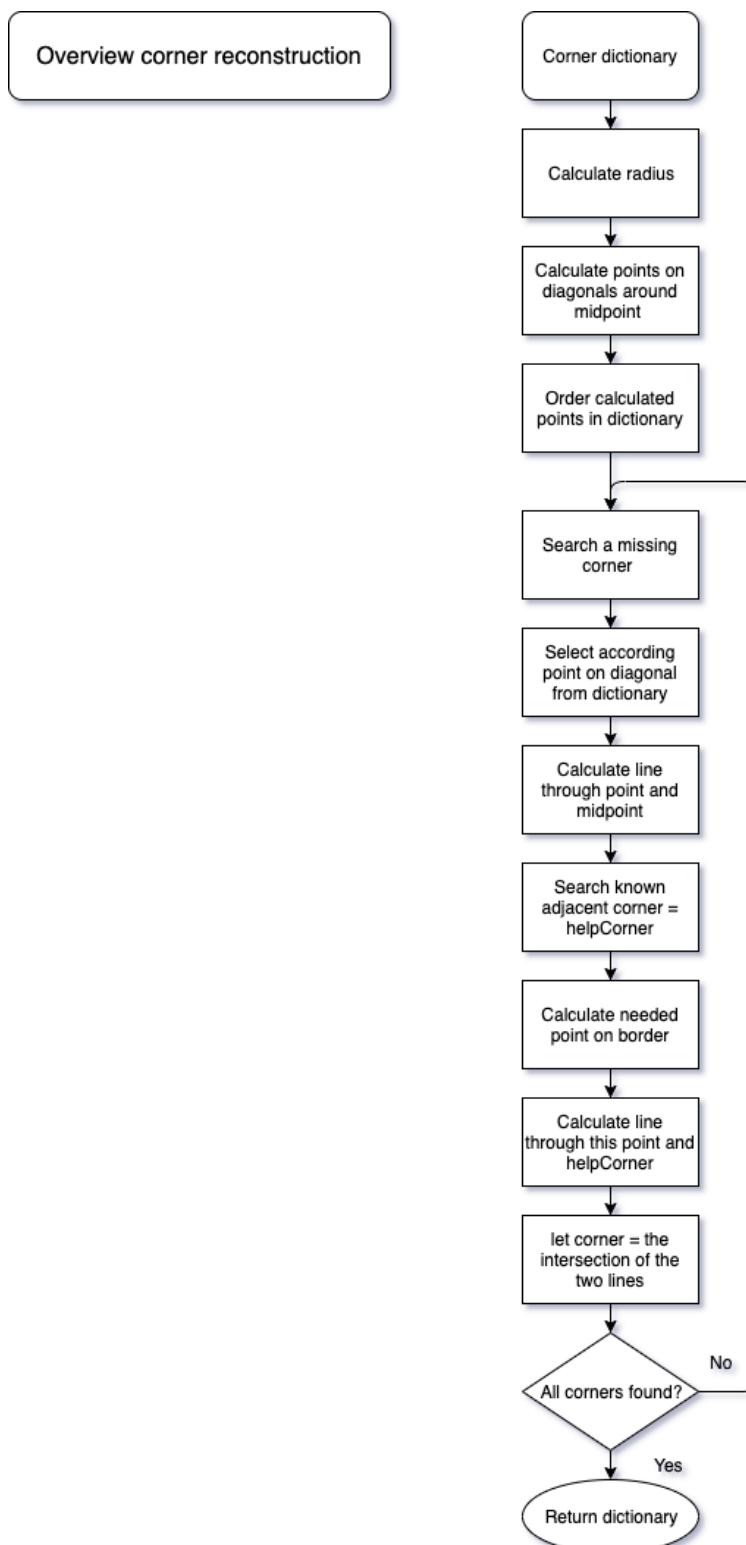
Figuur A.1

## A.2 Hoekdetectie en filtering



Figuur A.2

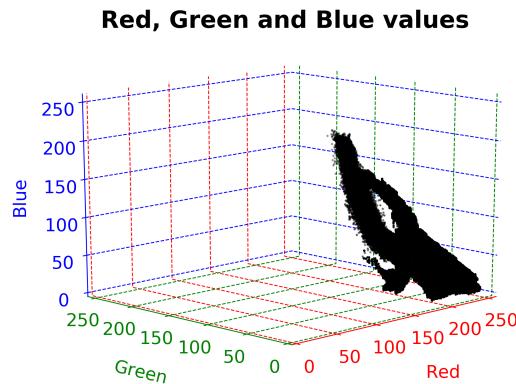
### A.3 Reconstructie



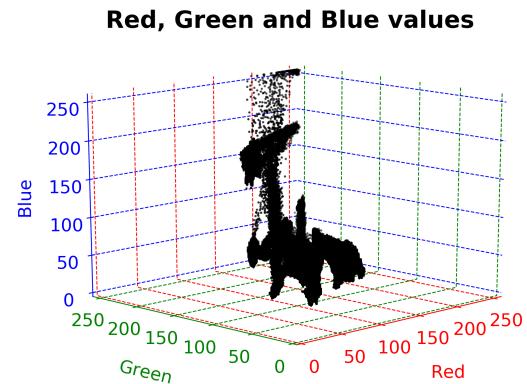
Figuur A.3

## B Kleuren plots

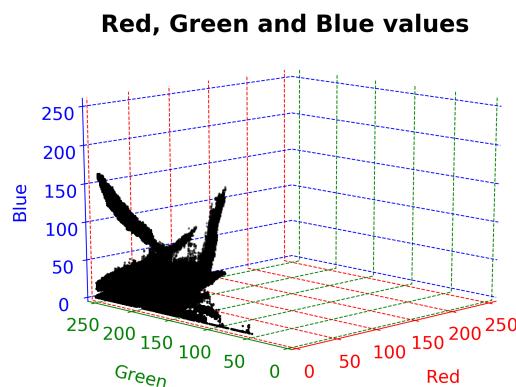
### B.1 RGB plots



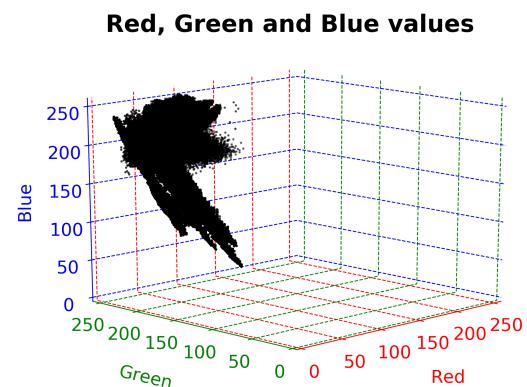
Figuur B.4: RGB plot voor de kleur rood.



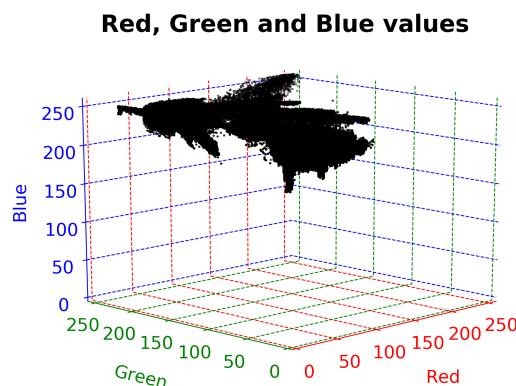
Figuur B.5: RGB plot voor de kleur geel.



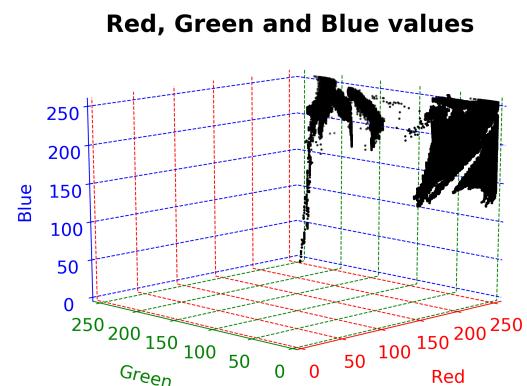
Figuur B.6: RGB plot voor de kleur groen.



Figuur B.7: RGB plot voor de kleur cy-an.

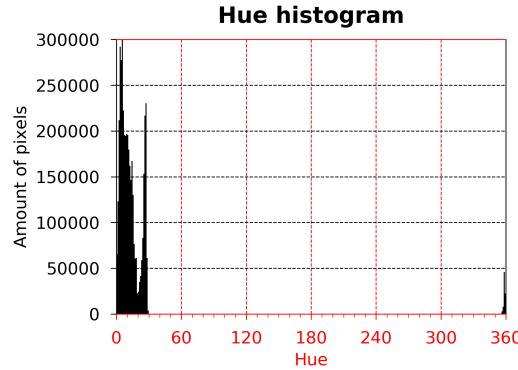


Figuur B.8: RGB plot voor de kleur blauw.

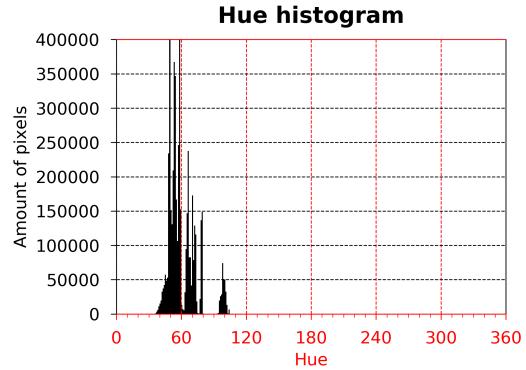


Figuur B.9: RGB plot voor de kleur ma-genta.

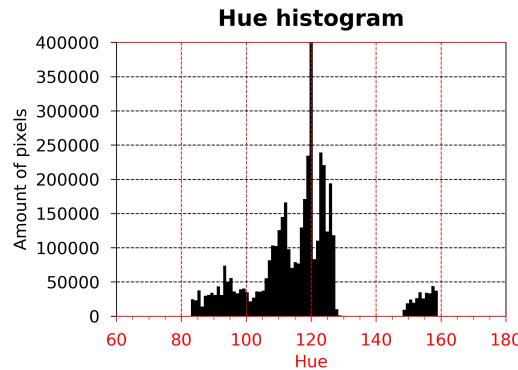
## B.2 Histogrammen



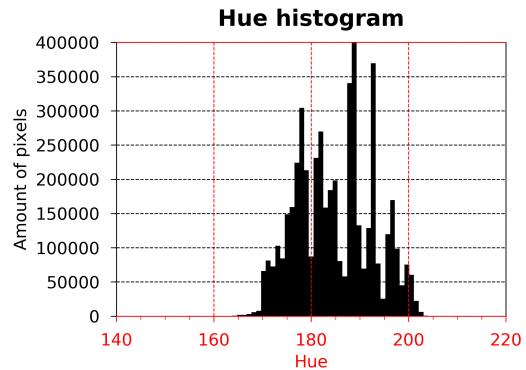
Figuur B.10: Histogram van de tint in functie van het aantal waargenomen pixels voor de kleur rood.



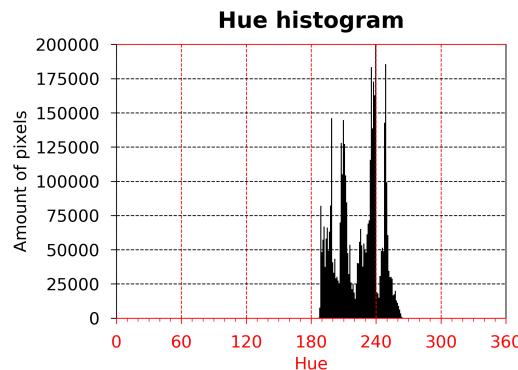
Figuur B.11: Histogram van de tint in functie van het aantal waargenomen pixels voor de kleur geel.



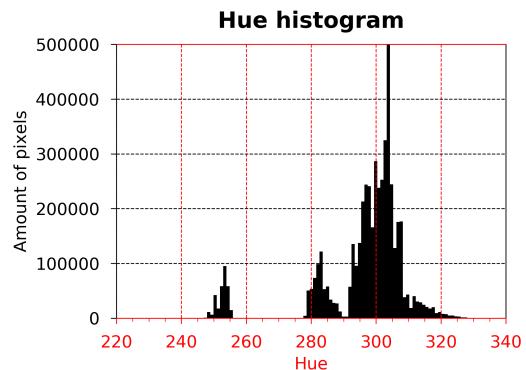
Figuur B.12: Histogram van de tint in functie van het aantal waargenomen pixels voor de kleur groen.



Figuur B.13: Histogram van de tint in functie van het aantal waargenomen pixels voor de kleur cyaan.



Figuur B.14: Histogram van de tint in functie van het aantal waargenomen pixels voor de kleur blauw.

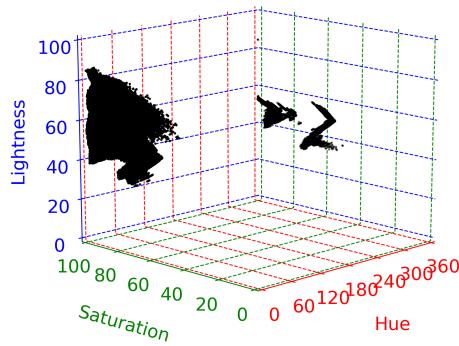


Figuur B.15: Histogram van de tint in functie van het aantal waargenomen pixels voor de kleur magenta.

### B.3 HSL plots

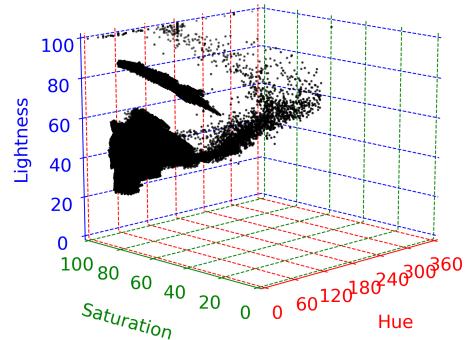
#### B.3.1 3D

**Hue, Saturation and Lightness values**



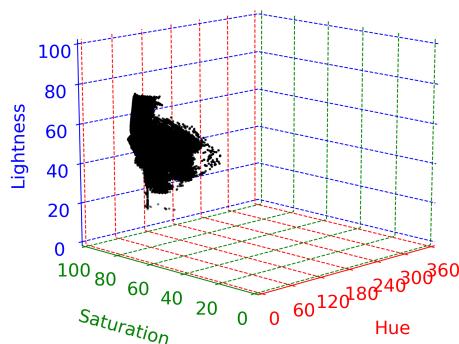
Figuur B.16: HSL plot voor de kleur rood.

**Hue, Saturation and Lightness values**



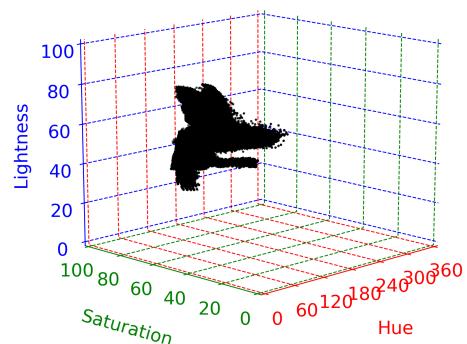
Figuur B.17: HSL plot voor de kleur geel.

**Hue, Saturation and Lightness values**



Figuur B.18: HSL plot voor de kleur groen.

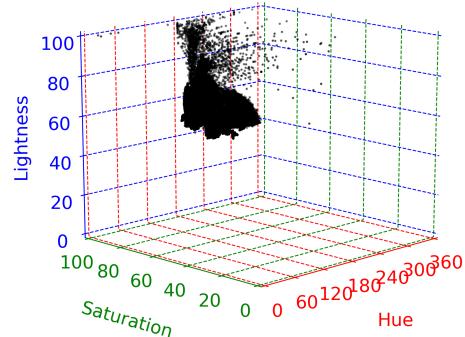
**Hue, Saturation and Lightness values**



Figuur B.19: HSL plot voor de kleur cyaan.

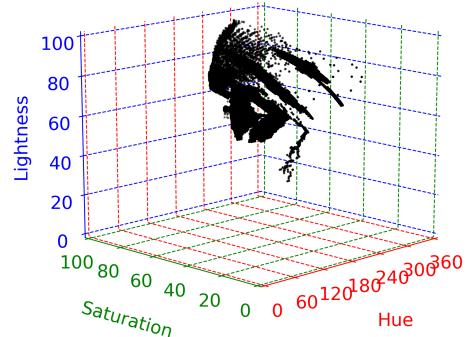
#### B.3.2 2D

**Hue, Saturation and Lightness values**



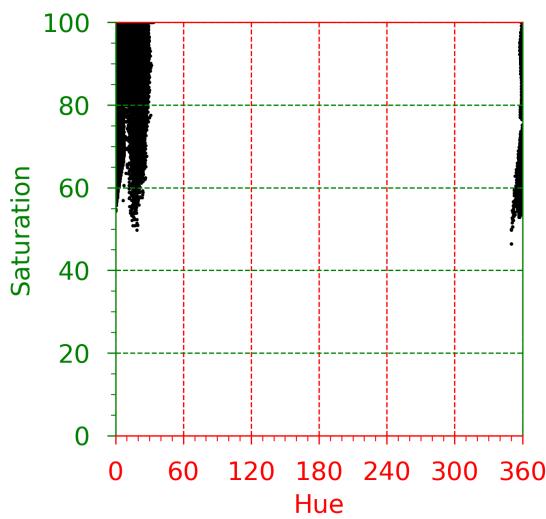
Figuur B.20: HSL plot voor de kleur blauw.

**Hue, Saturation and Lightness values**



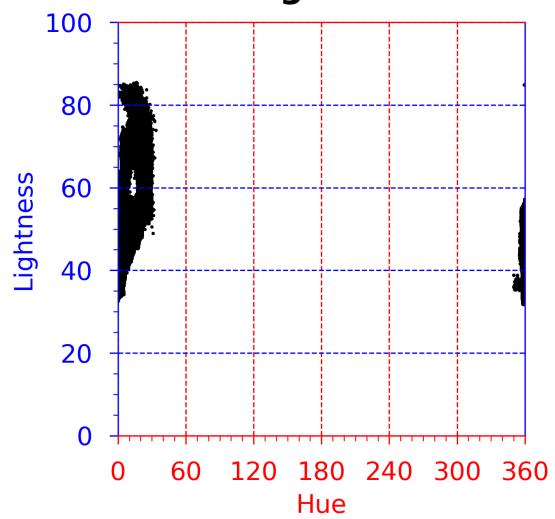
Figuur B.21: HSL plot voor de kleur magenta.

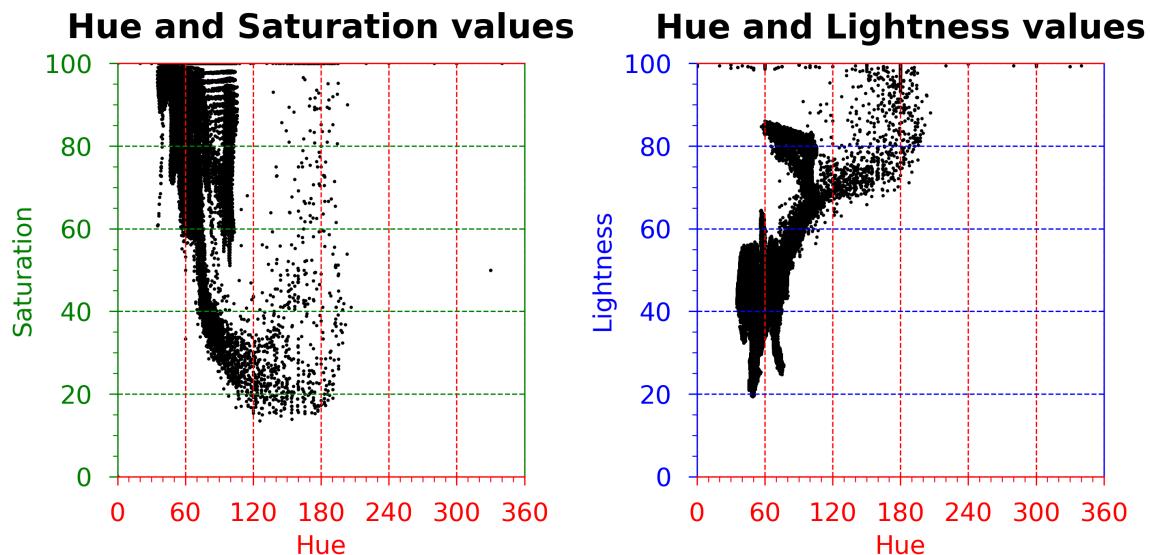
**Hue and Saturation values**



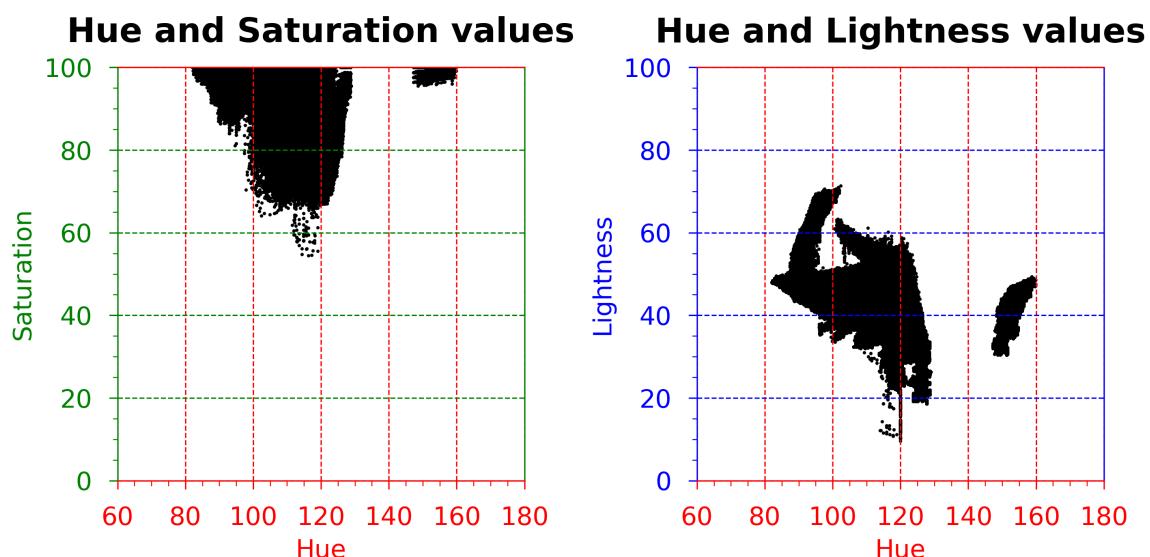
Figuur B.22: Scatter plots in functie van tint en saturatie, alsook in functie van tint en lichtheid voor de kleur rood.

**Hue and Lightness values**

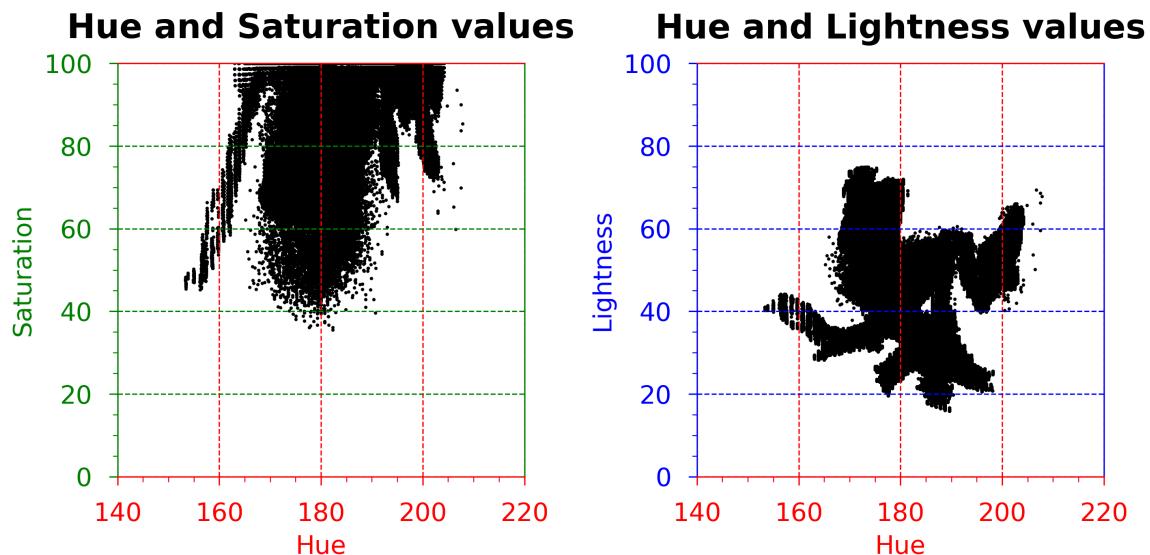




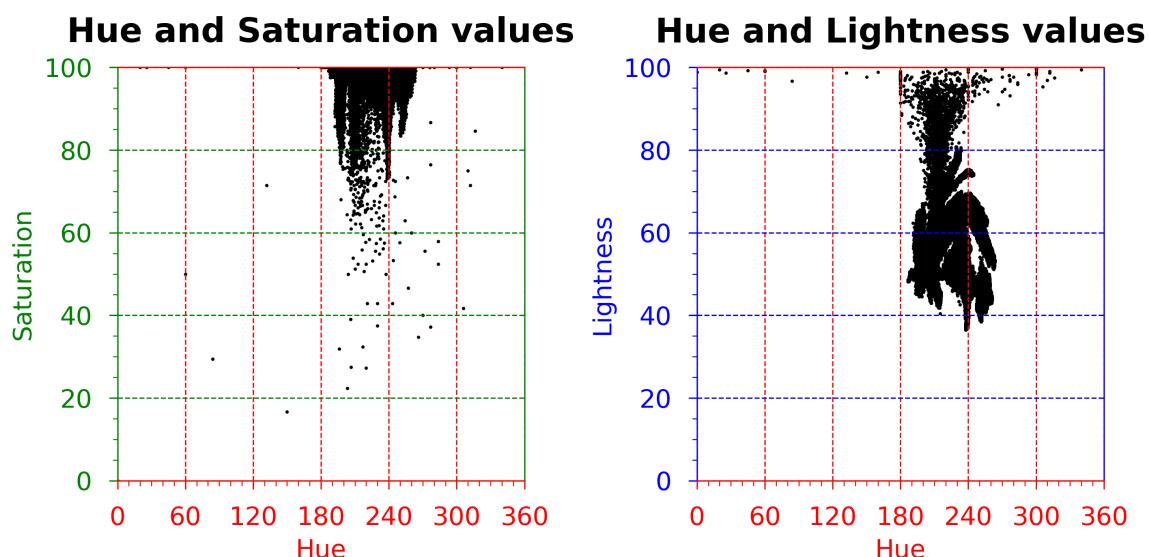
Figuur B.23: Scatter plots in functie van tint en saturatie, alsook in functie van tint en lichtheid voor de kleur geel.



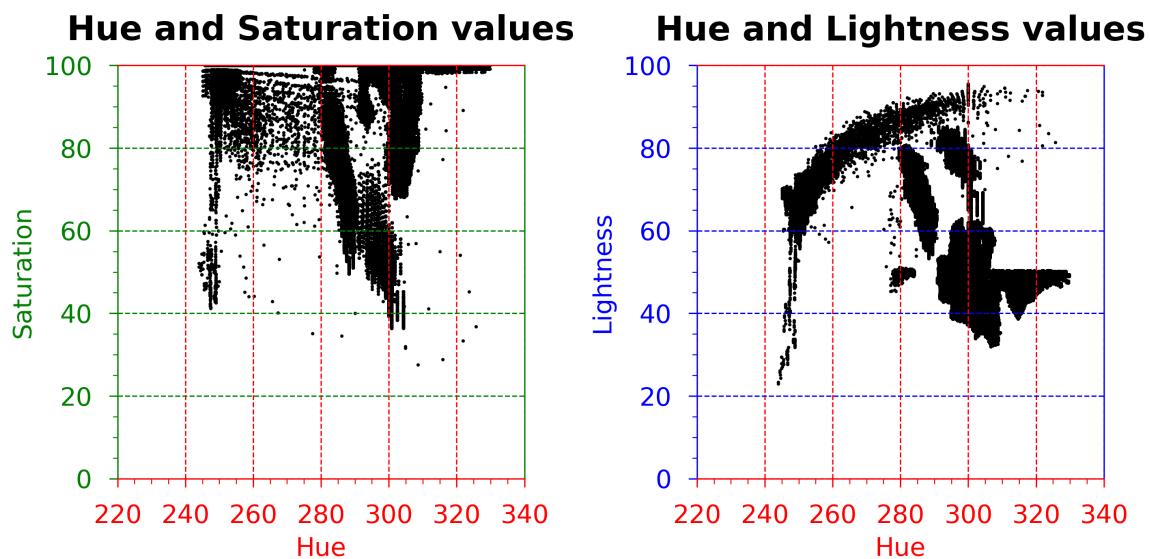
Figuur B.24: Scatter plots in functie van tint en saturatie, alsook in functie van tint en lichtheid voor de kleur groen.



Figuur B.25: Scatter plots in functie van tint en saturatie, alsook in functie van tint en lichtheid voor de kleur cyaan.



Figuur B.26: Scatter plots in functie van tint en saturatie, alsook in functie van tint en lichtheid voor de kleur blauw.



Figuur B.27: Scatter plots in functie van tint en saturatie, alsook in functie van tint en lichtheid voor de kleur magenta.