— Verslag P&O CW 2019–2020 Taak 3 —

Department Computerwetenschappen – KU Leuven 21 oktober 2019

Team 12	
Frédéric Blondeel	20h
Martijn Debeuf	45h
Toon Sauvillers	60h
Dirk Vanbeveren	12h
Bert Van den Bosch	52h
Seppe Van Steenbergen	54h

1 Introductie

Het herkennen van schermen, deze identificeren, lokaliseren uit een foto. Dit verslag behandeld deze uitdagingen. Het zoeken van schermen begint bij een foto. Deze foto bevat een scherm met een bepaald uitzicht, er is gekozen voor een groen-blauwe rand waarop gefilterd kan worden. Vervolgens zoekt het algoritme in de gefilterde foto naar aparte schermen en geeft aan hen een id. Met behulp van het kleurenverschil en bepaalde hoeken wordt de oriëntatie bepaald.

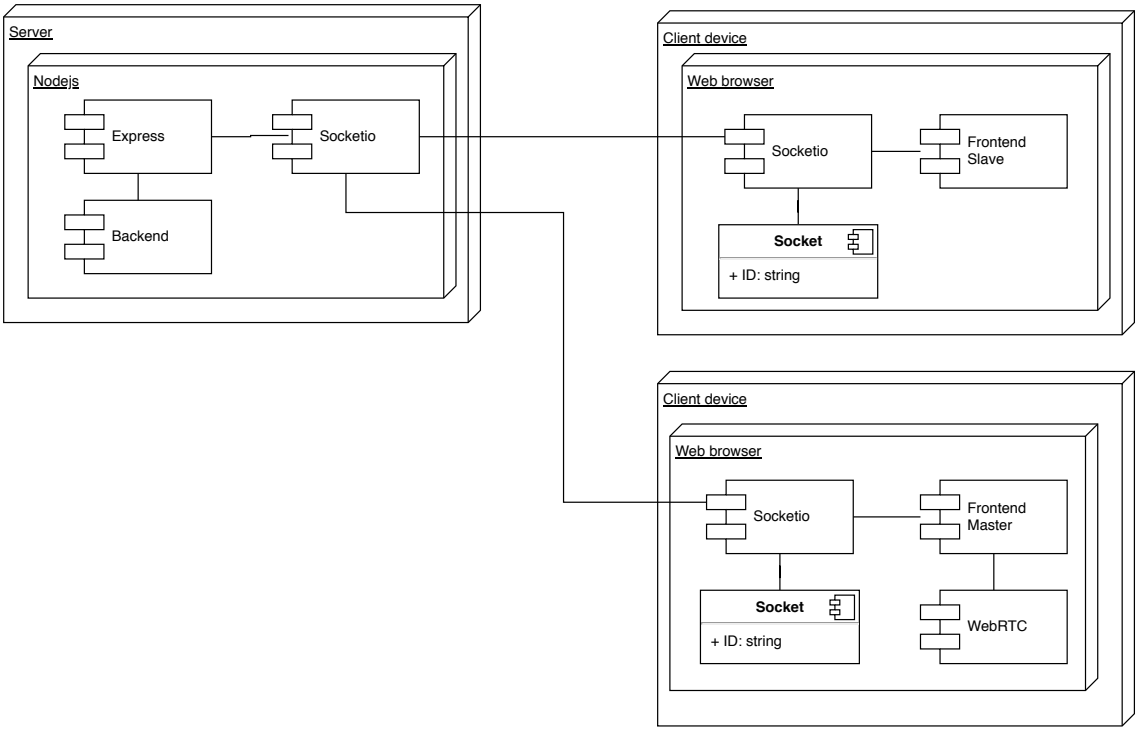
Het identificeren van het scherm gebeurd op dit moment nog apart met een kleurenbarcode. Deze barcode bevat 5 verschillende kleuren waardoor er 120 verschillende schermen geïdentificeerd kunnen worden. In de volgende weken worden deze twee, lokaliseren en identificeren, bij elkaar gezet.
Dit verslag behandeld de keuzes die gemaakt zijn alsook een uitleg bij de gebruikte algoritmen en hun tijdscomplexiteit. De mogelijke beperkingen worden met deze kennis geduid.

2 Design schematics and screenshots

(≈ 1 page. Depending on the need for a design overview.) *Design schematics, deployment diagrams, class diagrams, sequence diagrams, . . . Whatever you think we need to understand your design. You could add a little bit of text here but keep that under half a page. You reference the illustrations here from §3 when needed.*

All communication between clients runs through a nodejs server. We use the Socketio [**?**] library to set up a bidirectional communication channel between client and server. A deployment diagram can be found in Figure 1. Clients can take on a specific master or slave role by opening a specific webpage within the browser, i.e., the slaves will surf to `webaddress/slave`, while a client that wants to be the master loads the webpage found at `webaddress/master`.

We rely on webrtc [**?**] to collect a client-side video stream. This module offers the necessary tools for serialization, i.e., as a base64 encoded string. Serialization is required for transmission [**?**]. Server-side the backend code will process the image. In our demo we broadcast the image by sending it to all slaves. Slaves and masters can be reached by the server, as we use the concept of a namespace [**?**]. A namespace can be used to group clients with a similar role, and broadcast to them.



Figuur 1: Deployment diagram of the multiscreen casting framework.

A socket is assigned a unique identifier by Socketio. This identifier can be used to emit messages from the server to a specific client. A list of connected clients is available through the API of Socketio [**?**].

3 Algorithms

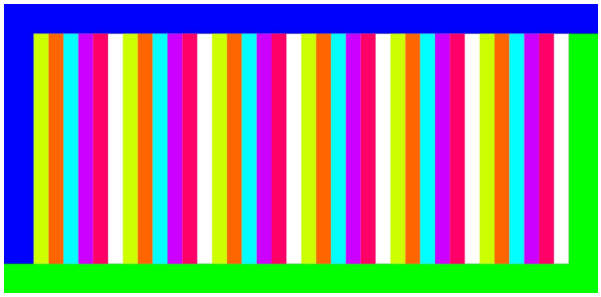
(≈ 1 page. Depending on necessity) *When you design or use an algorithm you should be able to explain how well it performs and if it satisfies the requirements. Often there will be alternatives or system parameters that need to be chosen. We want to know why you chose a specific set of parameters, or why you chose an approach over another. We supply some example text of last year.*

3.1 Scherm identificatie met barcodes

3.1.1 Algemene uitleg

Voor 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. De 5 kleuren zijn speciaal gekozen om zo ver mogelijk van elkaar verwijderd te zijn in het HSL spectrum. Op deze manier is de kans op foute detectie zo klein mogelijk gemaakt. Voor de identificatie van de slaves wordt er dus een unieke combinatie van deze 5 kleuren weergegeven. 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 van links naar rechts over de pixels, die zich in het midden van het scherm bevinden, itereren (complexiteit N). 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. Bijgevolg werd het aantal herhalingen van het patroon bekeken in een aantal testen. Uit deze testen werd een aantal van 7 herhalingen als optimaal resultaat bekomen. Een groter aantal herhalingen stemt overeen met meer kansen op een mogelijke detectie, maar stemt ook overeen met een kleinere oppervlakte per herhaling. Deze kleinere oppervlakte is dan weer nadelig voor detectie.



Figuur 2: Voorbeeld van een barcode met 6 herhalingen.

3.1.2 Data set

Tijdens de testfase van het algoritme werden een aantal foto's genomen. Deze foto's verschilden in afstand tot het scherm, het scherm zelf (kleuren kunnen afwijken van scherm tot scherm) en het aantal herhalingen van het patroon (barcode).

3.1.3 Experiments

3.2 Image

3.2.1 Algemene uitleg

3.2.2 Data set

3.2.3 Experiments

4 Testen

Voor het algemene testen van de hierboven beschreven algoritmen zijn er testen opgesteld. De testen bekijken enkele zelfgemaakte foto's. Deze zijn zelf gemaakt om de perfecte oriëntatie en grootte te weten zodat deze kunnen worden nagegaan. Er is een lijst opgesteld met alle testfoto's en daarbij hun eigenschappen. Er zijn drie verschillende testen, elk voor een belangrijke eigenschap. Zo zal het aantal schermen, de grootte van de schermen en de oriëntatie worden nagegaan. Met behulp van *console.assert* worden de verschillende testfoto's afgegaan. Bij het testen zijn er geen fouten ontdekt. Indien nodig kunnen er testen worden bijgeschreven om verdere functionaliteiten te bekijken.

5 Besluit

De *basic slave screen detection* zoals beschreven in de opgave bij taak 3 is nagenoeg volledig geïmplementeerd en werkende. Waar er in het begin *openCV* werd gebruikt, zijn er nu eigen algoritmen die al het werk leveren. Verschillende schermen kunnen worden gedetecteerd al moet er voor het zoeken naar de hoeken van deze schermen wel nog een vernufter algoritme komen. De oriëntatie van de schermen wordt snel en correct gedetecteerd. Het algoritme om de oriëntatie te vinden, gaat wel uit van correct gevonden hoeken en kan dus nog verbeterd worden. De implementatie van de algoritmen is zo efficiënt en zo simpel mogelijk gehouden, er wordt vaak niet meer gedaan dan enkel over de aparte pixels gelopen.

De identificatie van de schermen gebeurd met een kleurenbarcode. Deze kan op een zeer klein scherm nog worden gedetecteerd. Aangezien de code herhaalt kan worden op het scherm is het ook mogelijk om bij gedeeltelijke bedenking een scherm te herkennen. Deze methode berust nu nog op de assumptie dat het scherm recht staat. Er is al een oriëntatiematrix opgesteld om het scherm naar behoren te draaien maar dit is nog niet geïmplementeerd.

De basis voor *basic slave screen detection* is gelegd. De komende weken zal de code nog wat opgeruimd moeten worden. Ongebruikte functies moeten verdwijnen en geheugenmanagement moet worden herbekeken. Ook voor scaling moet er nog gewacht worden. Deze kan nog niet herkent worden. Als later de communicatie mogelijk is tussen de *slaves* en de *master* zal de grootte van het scherm worden opgevraagd. Door deze te vergelijken met de grootte van het scherm, kan de scaling worden berekend. Er is nog veel werk aan de winkel maar de basis die er nu is, mag er toch al wel zijn.