NIIP Assignment : Visible Light Communication

In this assignment, we will establish communication between two endpoints using visible light. Such types of communication systems are currently in use for various purposes. They are often applied when more traditional forms of communications (WiFi, wired,...) fail or are restricted in some fashion. The most well known in popular literature is LiFi.

Because of limitations in the current situation, we will make some abstractions :

- Communication is between a smartphone and a laptop.
- The communication is 1-way; there is no need for a return channel
- Communication will be slow and prone to errors

In essence, we are recreating the principles used between your infrared remote control and television set. However, in this case we do not use any specialized hardware such as phototransistors/diodes and infrared LEDs. Rather, we will be using commodity hardware you have available at home.

The setup for this assignment is :

- Your smartphone is the sending device. The user types a text string into an input field in the smartphone application and this is subsequently converted into a series of flashes of the screen (so we do not use the flashlight at the rear of the phone, just the screen).
- This flashing pattern is received by your laptop's webcam, converted back into the text string and displayed on the screen.

To facilitate things, we recommend the following. You do not HAVE to apply these, but we recommend you do, at least for the initial version.

- Use a web page on your mobile phone. Given that you only need to query for simple input and display something on screen, this can all be done using HTML/Javascript.
- To develop the software on your laptop, we recommend the use of OpenCV. This has Python bindings as well, so this facilitates development. We only need the very basic features of OpenCV; no advanced programming using this library is needed. We primarily recommend it because it abstracts access to the webcam and enables you to do some very basic operations (cutting out regions of the image, image filtering, access to individual color channels, etc).
- Restrict yourself to (alpha)numeric input only.
- Webcams typically (automatically) adjust themselves to surrounding light conditions. This can have an impact on frame rates or exposure levels. To make sure this does not happen :
    - Experiment in normal ambient light conditions. This means that you do not run the experiment in a completely dark room. Neither point the webcam in the direction of large windows with the sun shining through. Use a neutral setting.
    - To facilitate detection of the screen and to avoid the environment from impacting the settings of the webcam, we recommend taking a large sheet of cardboard (find an large box or

something) and cutting a square hole into it for the smartphone screen to show through. Place this setup in the middle of the view of the camera and make sure it is stationary. We do not (initially) expect you to support moving phones etc, simplify things ! Place it at a normal distance to the webcam (tens of centimeters) so the setup covers the entire view. Alternatively, if you struggle with getting a correct frame capture from your smartphone, look into OpenCV ROI to crop out a selection of your captured frames.
    - Make sure your screen brightness is set to a constant level and not too high (avoid overexposure)
    - Most webcams are restricted to 25-30fps, so take this into account.

You will probably find out that communication is error-prone and very slow. After development of your initial version, try to come up with optimizations to either decrease the error rate or increase the throughput (or both, if possible). Before implementing them, make sure you have the basics up and running and talk to the teaching team before starting on the optimizations. That way, we avoid pitfalls and dead ends.

Make sure you test and measure the performance of your setup. This is especially important if you are making improvements. We want to quantify each improvement, so an exact number is needed (e.g. a *34%* improvement in throughput). Without thorough testing/measuring, this is not possible and we cannot validate your statements.

We will allow for two weeks to implement this assignment (until April 24th). The evaluation will be done through a hangouts session (after April 26th) in which you will have to demonstrate and discuss your findings.

Basis implementatie

Om VLC mogelijk te maken met een goedkope webcam als sensor en een smartphonescherm als lichtbron bedachten we 2 mogelijke strategieën:

1. De framerate van de webcam locken en het volledige scherm veranderen van zwart naar wit. Indien we bijvoorbeeld versturen met 4 bits per seconde (4bps), locken we de framerate op 8 fps. Zo verkrijgen we een window van 2 frames die (hopelijk) beiden dezelfde waarde zagen, die waarde kunnen we dan toevoegen aan een bitarray.
2. Een tweede mogelijkheid bestaat uit het splitsen van het scherm in 2 delen: een clock en een data deel. Telkens bij een verandering van clock, lezen we het datadeel uit en voegen we de waarde toe aan de bitstream. Bij een lichtsensor zou dit overeenkomen met een actie op opgaande en neergaande flank. Omdat we nu enkel data moeten lezen na het veranderen van de clock, kan de fps van de webcam uncapped blijven.

Merk op dat in beide gevallen de fps van de webcam (= acquisition rate) telkens minstens het dubbele moet zijn van de bps. Indien dit niet het geval is, zouden we data kunnen missen. Dit wordt gegeven door het theorema van Nyquist-Shannon, waar het eerder bij het samplen van analoge signalen geldt, maar het is ook toepasbaar voor deze situatie. Indien de webcam bijvoorbeeld slechts 12 fps haalt (ondanks dat die vb 30 fps zou aankunnen, blijkt dit in de praktijk minder te zijn, natuurlijk ook door de OpenCV overhead), zou de Transmitter dus maximaal met 6 bps mogen zenden om alles nog goed te laten verlopen.

Inlezen van webcam

Hier maken we gebruik van de functie `cv2.VideoCapture(0)` om de capture-stream op te vangen. We voorzagen een klasse Webcam die deze function calls wrapt:

```
from webcam import Webcam
cam = Webcam()
cam.capture_start()
cam.capture_loop(Webcam.show_frame)
```

Tracken van de GSM

We gebruiken de MOSSE tracker (met behulp van `cv2.TrackerMOSSE_create()`). Door op de ‘s’-toets te drukken kan je een regio selecteren die bijgevolg getraceerd zal worden. De selectie kan bevestigd worden door op de RETURN-toets te duwen. Bijgevolg wordt er een groen kader toegevoegd aan het scherm die aantoont waar de getraceerde regio zich bevindt.

```
tracker = cv2.TrackerMOSSE_create()

while True:
	frame = cam.capture_new()
	success, box = tracker.update(frame)
	if success:
		# Use bounding box to crop frame
		# ...

		# Process cropped frame
		handle_frame(cropped)
```

```
def handle_frame(frame):
	# Make grayscale
	# Apply Gaussian blur
	# Apply Binary thresholding with OTSU threshold
	# Send result to callback function in detector
```

In de ScreenTracker-klasse die we maakten, bestaat ook de mogelijkheid om een andere OpenCV tracker toe te voegen zoals: `csrt`, `kcf`, `boosting`, `mil`, `tld`, `medianflow`, `mosse`. We voegden ook een `static` tracker toe die geen objecten volgt, maar gewoon op dezelfde plaats blijft kijken.

De `handle_frame` callback kan vervangen worden door een andere functie, zoals één uit de eigenlijke Detector.

Detecteren van bits

Hierbij gebruiken we een callback functie die bij elke frame gaat bepalen of er een zwart (0) of wit (1) scherm ingelezen wordt. De resulterende data wordt weergegeven op het scherm en in de console.
We testen 2 modi: één waarbij het volledige scherm nagekeken wordt en de timings van de Transmitter en de Detector dus in sync moeten zijn en één waarbij de bovenste helft van het scherm als klok dient en de onderste helft als data. Hieronder de werking van het eerste geval:

```
def callback(frame):
	# Check white to black ratio in pixels
	# If more whites, append True to buffer else False
	# When certain start bits are received (configurable),
	# slice the buffer to that index and append each following byte
	# to an output stream.

	# When an endbyte(s) is received, return the resulting string.
```

Voor de tweede modus, syncen we dus op de klokinput. Als de klok van waarde verandert (dus bij neergaand of opgaande flank in elektronica), wordt de volgende databit gesampled. Afhankelijk van die waarde voegen we '1' of '0' toe aan de buffer. Indien het startsymbool nog niet ontvangen werd, zoeken we eerst hier naar, daarna voegen we de bits per byte aan de resulterende string. Na het ontvangen van het stopsymbool (een new-line character), resetten we de staat en wachten we terug op het startsymbool.

Omdat het frame enkel gesampled wordt op de klokinput, zou de data altijd goed moeten doorkomen, op voorwaarde dat we sneller frames kunnen behandelen dan de Transmitter bits verzend. Minstens twee keer zo snel volgens het theorema van Nyquist-Shannon, i.e. indien de webcam slechts 12 fps haalt, zou de Transmitter dus maximaal 6 bits per seconden mogen zenden om alles nog goed te laten verlopen.

Transmitter

De transmitter bestaat uit een simpele webpagina waar de gebruiker een input string kan ingeven en de bitrate kan aanpassen van de transmissie. De transmissie kan bijgevolg gestart worden door op de witte zone te tappen. Verder voorzien we enkele andere instellingen zoals de keuze van de transmit modus en de toevoeging van Hamming bits en pariteit (zie optimalisaties verder).

De transmit functie verloopt als volgt:

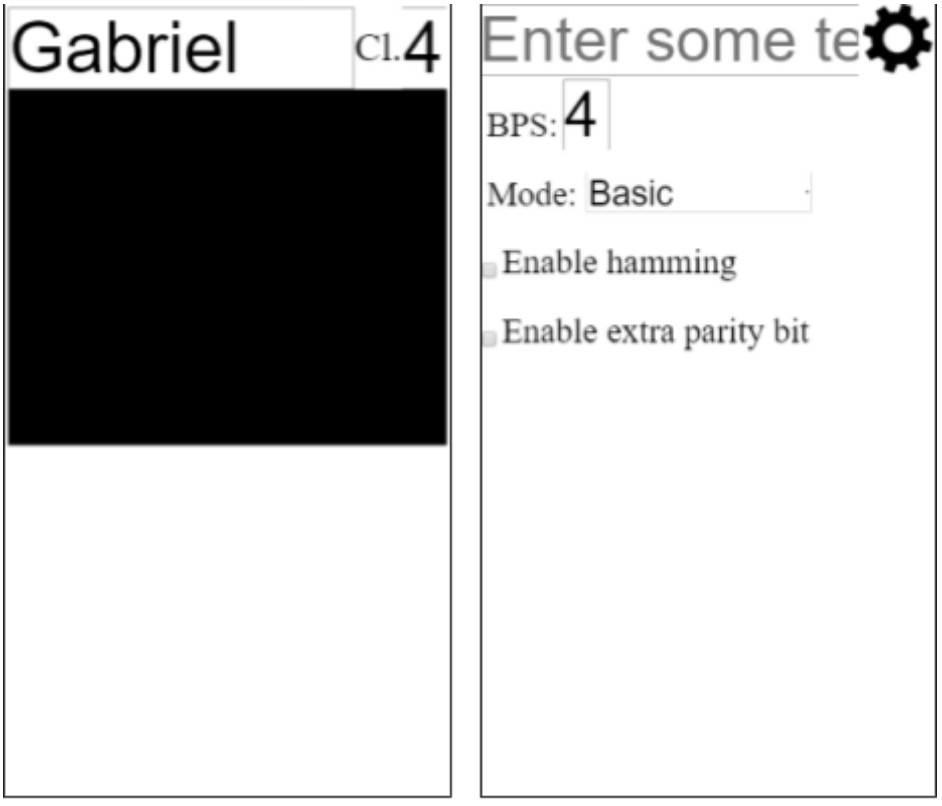
```
function transmit() {
	// Make list of bytes that has to be sent
	// Add the startbyte: 11110011
	// Add the bytes of the input text
	// Add the stopbyte: '\n'

	// Convert the bytes to a bit array
	// Transmit the bits by setting timeout (according to the bitrate set above)
```

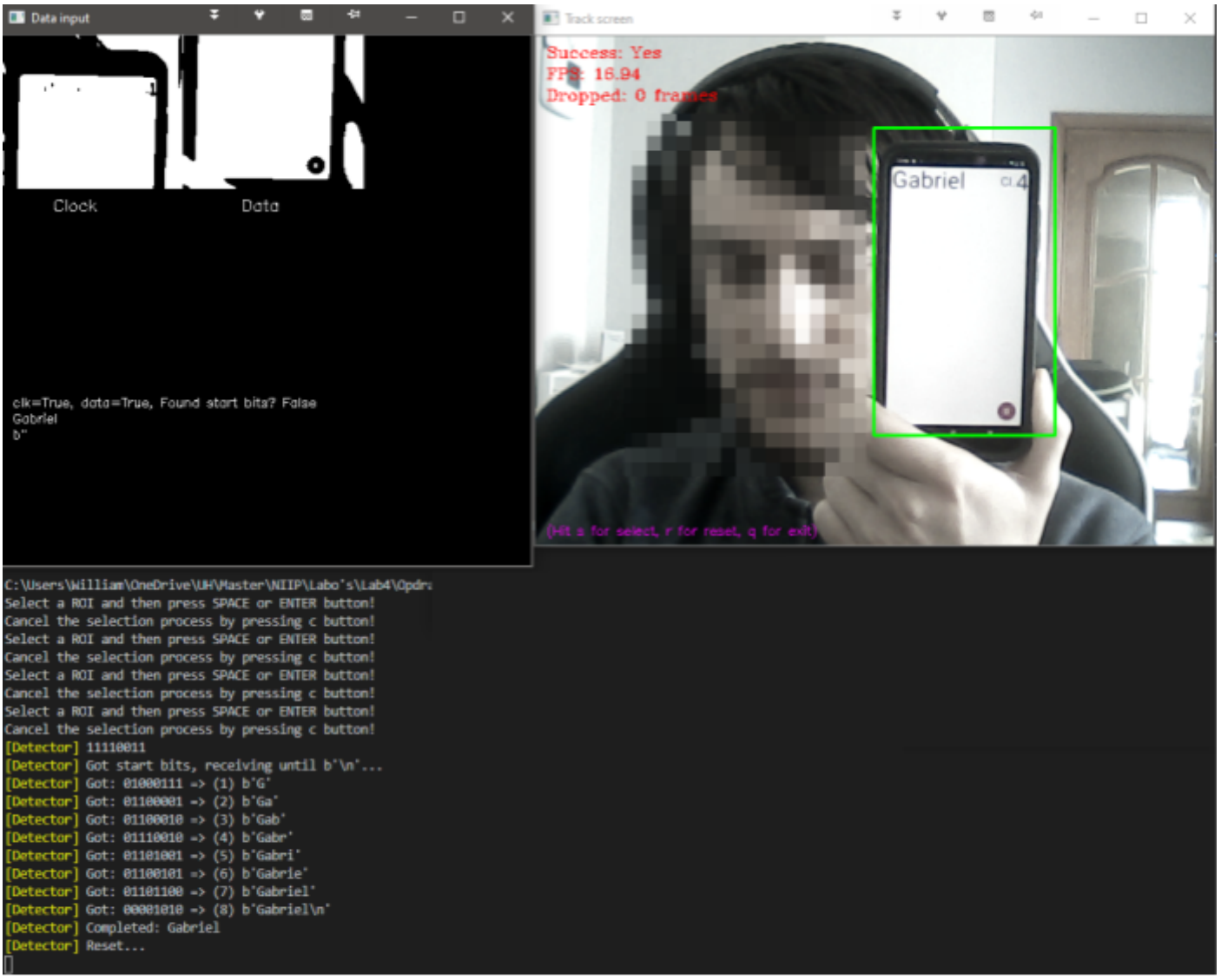
In de 2de modus, laten we toe een clock te activeren. Dit zorgt ervoor dat er een betere synchronisatie ontstaat tussen de ontvanger en de receiver. Er zal een kader getoond worden die geflipt wordt, telkens er een bit gestuurd wordt.

Test demo

Rechts de pagina op de smartphone. Met bovenaan het tekstinputvak en de bits per second input. In dit geval is de klokmodus geactiveerd. Op de afbeelding daarnaast is het instellingenscherm opengeklapt.



Transmitter in JavaScript met klokvlak (links) en settings scherm (rechts)



Detector op PC

Rechts bovenaan is het oorspronkelijke beeld van de webcam te zien. Bij het drukken van de 's'-toets, gaat er een nieuw scherm open waar men de Region Of Interest (ROI) kan aanduiden. Dit kan enkel een rechthoekvorm zijn die rechtop staat (of ligt), maar niet gedraaid. OpenCV zal vervolgens de regio tracken met de ingebouwde Tracker (of statisch houden als die geselecteerd werd in de code). Een tweede scherm opent waar de input gegeven is, teruggegeven door de Detector handle_frame callback. Hier zien we dat de getrackte regio in twee wordt gesplitst: de klokinput bovenaan en de datainput onderaan van de regio. Indien de telefoon liggend getrackt wordt, zal hij rechtop gedraaid worden. Check wel of de klokinput inderdaad de bovenste helft is van het scherm. Met de 'r'-toets kan de staat van de Detector gereset worden.

Na het drukken op het scherm, begint de Transmitter met het zenden van het startsymbool (`11110011`). Tegelijk is te zien dat de preview van de klok en data verandert en dat er wordt weergegeven wat de huidige waarden zijn. Na het ontvangen van het startsymbool, blijft de Detector bytes ontvangen tot de laatste byte een new-line character is. De staat zal daarna resetten en wachten tot het startsymbool opnieuw voorbij komt.

De snelheid van het systeem is vooral gelimiteerd aan de kant van de Detector, aangezien de webcam geen hoge framerate kan halen en deze verder omlaag wordt gebracht door de processing met OpenCV. Een goede combinatie lijkt de fps niet te limiteren, maar wel nakijken of deze hoger is dan 8 en de Transmitter instellen op 4 bps. Een snelheid van 1 of 2 bps werkt in bijna alle gevallen, waar een snelheid van 3 bps soms faalt om juist bits te detecteren. 4 bps lijkt een optimum en hogere waarden worden ondanks het gebruik van de klok, snel onnauwkeurig doordat de framerate van de webcam zo variabel is.

Om de framerate van de webcam te effectief locken, droppen we frames die te veel zijn, zodat er altijd een vaste framerate behouden wordt. Hiermee lopen we natuurlijk wel het risico dat er net een belangrijk frame wordt weggegooid.

Mogelijke optimalisaties

Andere technieken

1. ### Aanpassen van brightness (modulatie op brightness; OK - Niels)
   Een mogelijke optimalisatie zou het implementeren van verschillende brightness levels kunnen zijn. We zouden 4 verschillende grijswaarden kunnen onderscheiden (wit - 33% - 66% - zwart, respectievelijk `00`, `01`, `10` en `11`) en op deze manier een hogere throughput voorzien. De bitrate zou hierdoor 2x zo groot kunnen worden.

2. ### Andere set van symbolen (Niet doen - Niels)
   Een andere mogelijkheid is het beperken van de symbolen set die we ondersteunen. Bijvoorbeeld een beperking die enkel cijfers, (lower- en higher case) letters en een spatie kunnen doorsturen. Hiermee beperken we onze set tot 63 symbolen, waardoor we per karakter 2 bits kunnen uitsparen.

3. ### Modulatie op frequentie
   Zoals bij radiosignalen kunnen wij ook modulatie toepassen op de frequentie. Hierbij denken we aan kleurfrequenties. Wanneer we ons spectrum van kleuren uitbreiden, kunnen we een grotere throughput voorzien. Deze methode zou dan bijvoorbeeld omgezet kunnen worden naar een RGB led, aangezien deze ook meerdere kleuren kan doorsturen.

4. ### Encoden van symbolen (Niet doen - Niels)
   Zoals in 2, zouden we elke letter en cijfer ook kunnen voorstellen met een nieuw, kleiner symbool. Voorbeeld zoals het opstellen van een Hufman tree en deze gebruiken om characters te verzenden. Hierdoor zouden characters uit een variabel aantal bits (tot aan een leaf in de tree) bestaan.
   Aangezien dit overeenkomt met compressie, zullen we dit niet verder beschouwen.
   Anderzijds zouden we ook het aantal symbolen sterk kunnen beperken door bijvoorbeeld volledige woorden of zinnen voor te stellen met een klein aantal bits. Hiermee zouden we dan ook enkel die woorden of zinnen kunnen sturen en ontvangen.

Error detectie/correctie

1. ### Toevoegen van CRC bit(s)
   Omdat we ASCII bytes doorsturen, wordt technisch gezien de eerste bit niet gebruikt, deze is altijd 0. We kunnen eventueel deze bit gebruiken als checkbit om te weten of de huidige byte effectief goed ontvangen werd. Grote bit errors zijn dan wel niet te detecteren.

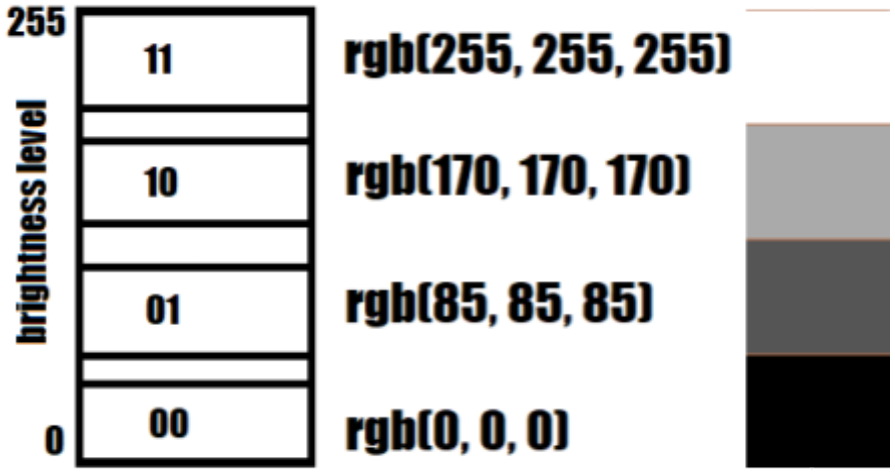
2. ### Toevoegen van bitcorrectie
   a. #### Hamming code met pariteitbits
      We kunnen een deel van een symbool vervangen door pariteitbits. Als we bijvoorbeeld een (7,4)-Hammingcode gebruiken, bekomen we een symbool van 7 bits, waarvan 4 data- en 3 pariteitsbits. Hierbij zou men alle 1 bit errors kunnen corrigeren, maar 2 bit errors enkel kunnen detecteren. Aangezien er in ons VLC-systeem soms meer dan 2 bit errors voorkomen, is het misschien beter om minder data bits per symbool te versturen, zodat ook grotere errors gecorrigeerd zouden kunnen worden.

   b. #### Reed-Solomon (of andere)

Toegepaste optimalisaties

Toepassen van brightness voor hogere transmission rate

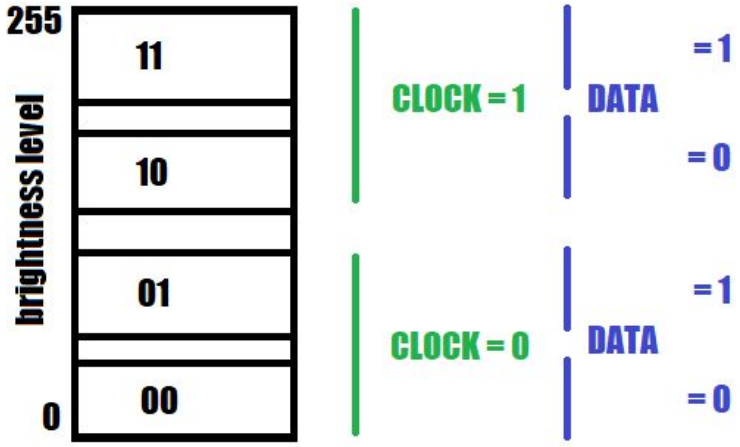
Een eerste optimalisatie die we hebben ingebouwd is die van het toepassen van de brightness. Hierbij nemen we telkens twee bits tegelijk in acht. De grijswaarde van de transmitter wordt vervolgens bepaald aan de hand van volgend schema:



Op deze manier hopen we een transmissie rate te bekomen die dubbel zo snel is als de basis implementatie.

Brightness modulation

We implementeren een 2de versie gebruik makend van brightness modulation. Hiervoor gebruiken we het volgende schema:



We detecteren 4 intervallen van helderheden. Afhankelijk van welke het is, zien we dat de clock of data bit verandert. Enkel als de klok verandert, samplen we de data zoals voorheen. Indien enkel de data bit verandert, maar de klok bit niet, dus brightness blijft boven of onder de helft, wordt de data bit dus niet beschouwd. Op deze manier kunnen we een clock implementeren zonder het scherm te hoeven opsplitsen in verschillende waarneembare regio's

In de implementatie berekenen we de intervallen dynamisch aan de hand van de gevraagde levels (4 in dit geval). We kunnen de grootte van elk interval wel instellen, zodat er een deadzone tussen de intervallen ontstaat (zoals in de afbeelding hierboven) of zodat ze net aansluiten aan elkaar. In de praktijk zagen we echter dat afhankelijk van het omgevingslicht, de weerkaatsing op het telefoonscherm en de helderheid van

het scherm, de kleuren, met name de 2 grijswaarden, soms niet correct werden gedetecteerd. Soms zag hij het licht grijs ook als wit, terwijl het donkergrijs dan in een deadzone viel, of lichtgrijs leek te worden.

Om deze problemen te mitigeren, voegden we tijdelijk een hardcoded interval toe met volgende waarden (met helderheidswaarden tussen 0 (zwart) en 255 (wit)):

```
ranges = [
      0,  30,  # Zwart
     30, 128,  # Donkergrijs
    128, 230,  # Lichtgrijs
    230, 255   # Wit
]
```

De grootte van het zwarte en witte interval is hierbij veel kleiner, omdat deze 2 uitersten altijd goed worden gezien. De grijswaarden spannen een veel groter bereik.

Hamming encoding

Om de error-rate van onze datatransmissie te verlagen hebben we een optie toegevoegd om Hamming encoding toe te passen op de data. Hiervoor gebruiken we de standaardimplementatie die de Hammingpariteitbits berekend en in het volgende (7,4)-Hammings encoding schema stopt:

```
(P) H2 H1 D3 H0 D2 D1 D0
```

De Hamming bits worden berekend met een XOR operatie:

- H2: D1 ^ D2 ^ D4
- H1: D1 ^ D3 ^ D4
- H0: D2 ^ D3 ^ D4
- P: H2 ^ H1 ^ D0 ^ H0 ^ D1 ^ D2 ^ D3

Met als eerste bit een optionele extra pariteit, gevolgde door de Hammingpariteiten op posities 1, 2 en 4 (vanaf MSB) en de databits van een nibble op de overige posities. Deze optie kan toegepast worden op alle modi die hierboven beschreven staan. In de omgekeerde (decode) richting, wordt de *syndrome* berekend afhankelijk van de Hammingpariteitsbits en kan gekeken worden of er een bit verkeerd ontvangen werd, en kan men deze ook corrigeren. Grotere biterrors (2 bits) kunnen niet gecorigeerd worden, maar wel gezien met de extra pariteitsbit.

Testresultaten

We sturen de string "Hello" (startbyte + 5 ascii values + '\n') met verschillende bitrates in elke modus. We testten elke combinatie minstens 10 keer (>= 5 keer met de hardware bij Bram en >= 5 bij William). Verder werd hier de statische tracker telkens gebruikt om errors door verandering in belichting bij het bewegen van de smartphone te voorkomen.

Snelheid \/ Modus:	**Single screen** (2*bps fps lock)	**Clock/Data** (var fps >= 2*bps)	**Brightness mod**	**Brightness clk**
1 bps	10/10 F	10/10 F (6 fps lock)	/	/
2 bps	9/10 F, 1/10 P[1]	10/10 F (6 fps lock)	/	/
3 bps	7/10 F, 3/10 NS	2 F, 3/5 P[1] (var fps) 10/10 F (9 fps lock)	/	/
4 bps	7/10 F, 3/10 NS	4 F, 7 P[1-6], 4/15 NS (var fps) 6 F, 3 P[1,5,6], 1/10 NS (12 fps lock)	5 P[6] (telkens zelfde verkeerde bytes), 5/10 NS	10/10 NS
5 bps	5/10 F, 5/10 NS	3 F, 7 P[2-5], 5/15 NS	10/10 NS	10/10 NS
6 bps	1 F, 2 P[1], 3/10 NS	1 F, 10 P[1-5], 4/15 NS	10/10 NS	10/10 NS
4 bps (Hamming)	7/10 F, 3/10 NS	7 F, 7 P[1-3], 1/15 NS	5 P[6], 5/10 NS	6P[6], 4/10 NS
6 bps (Hamming)	5 F, 1 P[2], 7/13 NS	6 F, 6 P[1-2,4-5], 3/15 NS	/	/
8 bps (Hamming)	1 F, 1 P[6], 8/10 NS	4 F, 6 P[1, 5-6], 5/15 NS	/	/

Met:

- F: full, startsymbool en volledige data ontvangen zonder errors;
- NS: No startsymbol, geen startsymbool gesynct, rest van data was mogelijk goed, maar out of sync;
- P[x]: partial, startsymbool ontvangen, en ten hoogste x bytes **slecht** ontvangen (van de 6);
- N: none, startsymbool ontvangen, maar geen enkele byte was correct.
- /: geen meting uitgevoerd

Opmerkingen

- Voor de Single screen modus, zetten we de capture fps vast op 2 keer de datarate. Dit zorgde voor de beste resultaten. Bij het zenden van meer dan 4 bps, zien we een sterke drop in accuraatheid. Door het toevoegen van Hamming, bleek het wel terug iets beter te gaan.
- In de tweede modus, met Clock/Data-scherm, lijkt de synchronisatie effectief beter te verlopen bij het vastzetten van de FPS. Bij variabele FPS gaat het iets slechter. Zoals in modus 1, zien we hier ook een sterke verbetering bij het toevoegen van Hamming-encoding.
- De implementatie is sterk afhankelijk van de helderheid van het scherm. Vooral de brightness modulation is hier zeer gevoelig voor. Ondanks dat bijvoorbeeld de smartphone van Bram een OLED-scherm heeft met een e-reader modus, wat betere overgangen voor helderheden biedt, blijkt het detecteren van Brightness modulation niet nauwkeurig genoeg.
  We konden hier amper meetbare resultaten uit afleiden. Een toevoeging van een kalibratiestap zou dit mogelijk kunnen verbeteren. Dan zouden we de verschillende kleurintervallen beter kunnen bepalen op het huidige gebruikte device met de gebruikte webcam en de lichtomstandigheden.