

Face Swap

TEK5030 – Maskinsyn - Universitetet i Oslo

Nikolai René Berg

Pål Primstad

Scott A. F. Sørensen

1 Intro

I dette prosjektet skal vi utføre face swap, altså to ansikter får ansiktene sine byttet med hverandre. Målet er å kunne ta inn en videofeed og bytte om ansiktene live i video-feeden. Det er flere utfordringer med å lage en robust face swap algoritme. Det kan være store forskjeller mellom to ansikter, fasongen, rotasjonen til ansiktene og forskjeller i hudtone gjør det vanskelig å gjøre en overbevisende face swap.

2 Metode

For å utføre en face swap begynner vi først med å detektere ansiktene i bildet, og deretter hentes ut punkter i ansiktene. Disse punktene er koordinater til ulike ansiktstrekk som øyenbryn og kjeve. Så lages et omriss av ansiktene basert på ytterpunktene av disse punktene. Neste steg er å gjøre Delaunay triangulering på ansiktene, som gjør at vi kan dele opp ansiktene i trekanner. Disse trekantene blir brukt til å affin warp trekantene i det ene ansiktet til trekantene i mottakeransiktet. Med Delaunay triangulering og affine warp blir forskjeller i rotasjon og størrelse håndtert. For å fjerne skille mellom mottakeransiktet og det nye på-limte ansiktet brukes Laplace blending.

2.1 Ansiktsdeteksjon og punkter i ansiktene

Først steg er å finne ansiktene i bildet og hente ut punktene vi trenger i ansiktene. For å gjøre dette bruker vi dlib funksjonene `get_frontal_face_detector()` og `shape_predictor()`, som er dlib sin implementasjon av algoritmen beskrevet i forskningsartikkelen One Millisecond Face Alignment with an Ensemble of Regression Trees av Vahid Kazemi and Josephine Sullivan[1]. Vi bruker også datafilen '`shape_predictor_68-face_landmarks.dat`' som sendes inn til `shape_predictor()`, som er en ferdig trenet modell for ansiktsgjenkjenning. Med `predictor()` kan vi hente ut alle landemerkepunktene som vises i Figure 1, men vi henter bare ut punktene 0 til 26 siden vi senere skal lage et convex hull av ansiktet og trenger da ikke punktene som er midt i ansiktet.



Figure 1: Punktene som ansiktsdeteksjonen kan finne

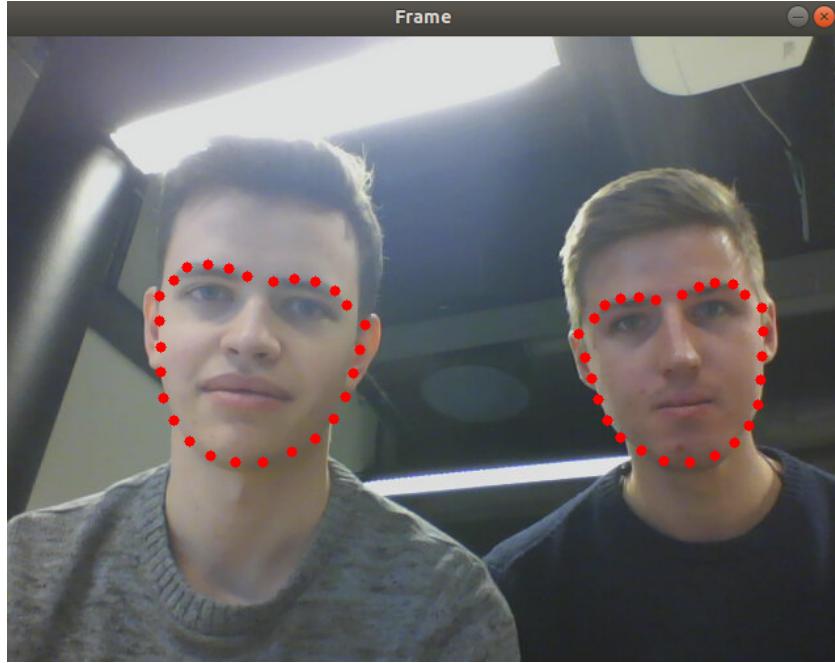


Figure 2: Våre ansikt med landemerkene 0-26

2.2 Convex hull algoritme

For å bytte om ansiktene og lage en maske av det aktuelle området trenger vi et omriss av ansiktene, generert fra punktene vist i Figure 1. Så vi trenger et convex hull av det aktuelle området. Convex hull algoritmen vår er basert på en algoritme introdusert i 1973 kalt Jarvis March[2], også refert til som gift wrapping. I algoritmen begynner man med å finne det punktet lengst til venstre (minste x-punkt) av alle oppgitte punkter. Dette punktet er garantert til å ligge på omrisset og vi starter letingen fra dette punktet. Deretter sjekker man alle punktene, og finner det punktet som er nærmest det nåværende punktet i klokkeretning. Du legger så til dette punktet i convex hull arrayen og fortsetter som ovenfor med å finne neste punkt som ligger nærmest i klokkeretning. Får å redusere søkerområdet sletter vi punkter vi har vært innom underveis, utenom startpunktet, siden når vi finner startpunktet som neste punkt i klokkeretning så betyr dette at vi har gått en runde og har funnet vårt convex hull. Returnerer den nye arrayen med punkter som bare inneholder punkter på omrisset og i rekkefølge.

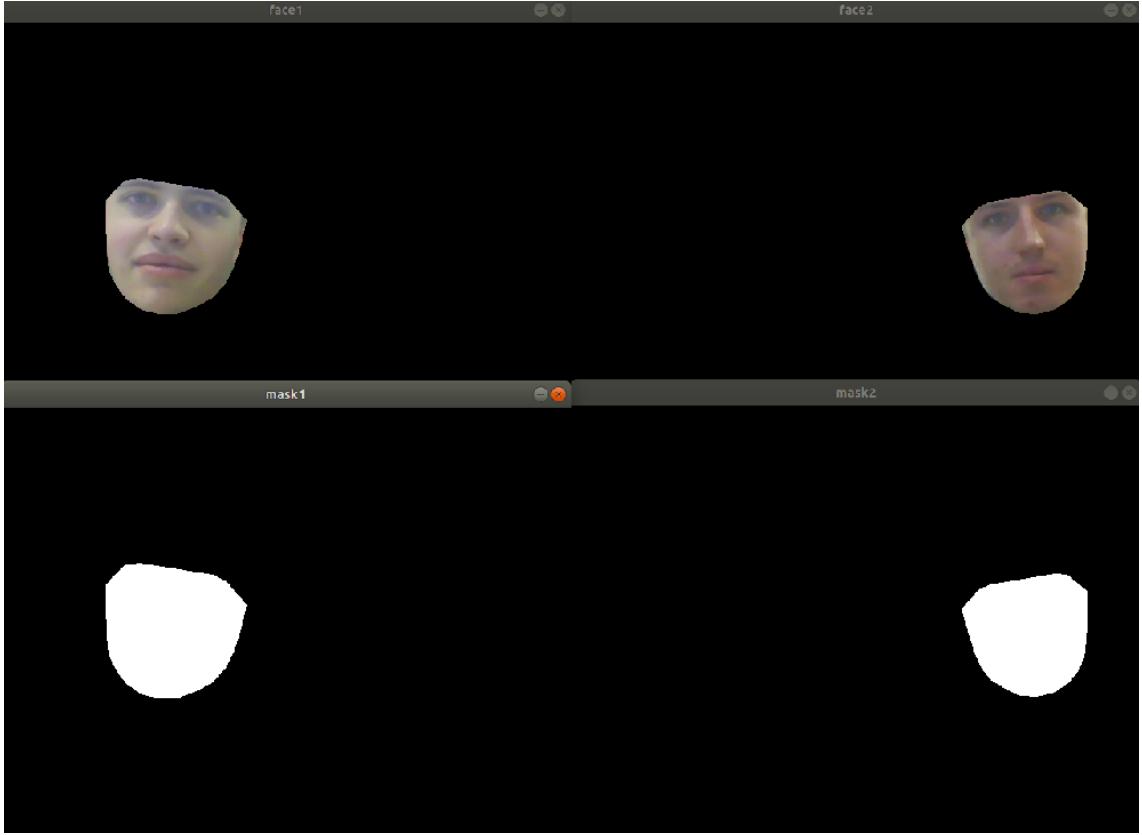


Figure 3: Convex hull

2.3 Delaunay triangulering

Vi har nå det området av ansiktet vi vil bytte ut, og må nå finne ut hvordan vi skal få til selve byttet. Her satt vi fast en stund og måtte lese en del på nett for inspirasjon, og vi kom da frem til å bruke delaunay triangulering[3]. Det er en metode som mest blir brukt til modellering av terreng og morphing av bilder, men den fungerer også bra i generelle tilfeller der et sett med punkter må deles opp i trekantene. Det denne metoden gjør er å dele ansiktet inn i ikke overlappende trekanter, slik at vi senere kan transformere trekantene fra det ene ansiktet til det andre. Grunnen til at vi ønsker å dele ansiktet inn i trekantene er fordi det er lettere å transformere trekantene enn å transformere hele ansiktet. Vi bruker cv2 sin funksjon Subdiv2D() punktene i Figure 2 for å generere disse trekantene.

Når vi nå har delt ansiktet inn i trekantene er utfordringen å finne ut hvilke trekantene som tilsvarer hverandre i de to ansiktene. Måten vi løste det på er å kun gjøre delaunay triangulering på ett ansikt. Siden trekantene ble laget basert på punktene i Figure 1 og 2, så kan vi bruke indeksene til disse punktene for å finne ut av korresponderende trekantene. Hver av trekantene er tre piksel-koordinater, og disse koordinatene vil da tilsvare et av punktene vist i Figure. 2. Når vi da har alle punkt-indeksene for det ene ansiktet kan vi tegne tilsvarende trekantene i mottakeransiktet, og vi har da kontroll over hvilke trekantene som hører sammen. Siden vi ikke bare skal bytte ut ett ansikt, men to, så gjør vi dette to ganger der vi bytter om på source/destination ansikt.

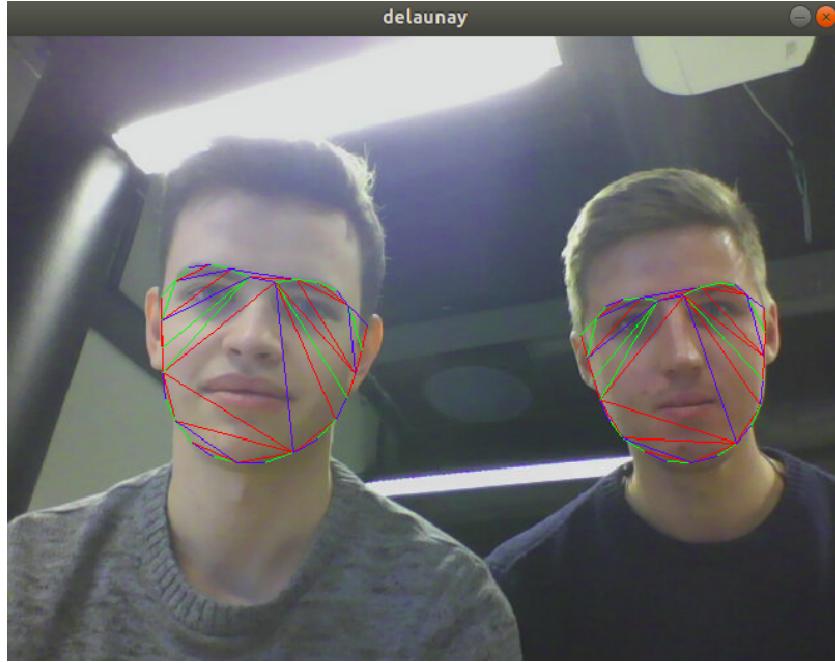


Figure 4: Delaunay triangulering

Hvordan fungerer Delaunay triangulering?

Kriteriet til Delaunay triangulering er at et punkt skal ligge utenfor eller på, men ikke innenfor den omskrevne sirkelen til en trekant, se Figure 5. Dette kriteriet er også en viktig egenskap som gjør at man kan flippe linjer for å oppnå kriteriet. Har man fire punkter ABCD som skal deles opp i to trekanner så trekker man en linje mellom to av punktene, f.eks en linje BD. Hvis denne linjen skaper to trekanner som ikke oppfyller kravet så kan man flippe linjen, slik at linjen da blir mellom AC, og den oppfyller kravet.

Typisk har man mye mer enn fire punkter, og måten det gjøres på at er at man legger til ett og ett punkt. Hvis et nytt punkt p blir lagt til havner innenfor en allerede eksisterende trekant, så deles trekanten som inneholder p inn i tre, så bruker vi flipp-algoritmen. Man søker gjennom alle trekanner for å finne den som inneholder p , og kjører flipp-algoritmen på alle. Det gjør at man får en kjøretid på $O(n^2)$ så det kan potensielt gå ganske tregt.

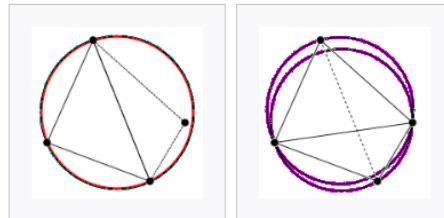


Figure 5: Ikke gyldig Delaunay triangulering til venstre,
gyldig Delaunay triangulering til høyre

2.4 Affin transformasjon

For å transformere trekantene og å håndtere forskjeller i ansiktene, størrelse og rotasjon, bruker vi affin transformasjon. Affin transformasjon i 2D trenger minimum tre punkter for å kalkulere affin matrisen, som vi har, og hvert triangel blir affine warped til å passe med tilsvarende triangel i mottakeransiktet. Affin transformasjonen mapper de tre hjørnene i et triangel til de tre hjørnene i det tilsvarende triangellet i mottakeransiktet. Her bruker vi OpenCVs `getAffineTransform()` og `warpAffine()` på hvert triangel. Vi bruker affine transformasjon siden vi vil beholde linjene i trekantene våre sånn at det er enkelt å plassere de inn i det andre ansiktet. Affine endrer på rotasjon, translasjon og skalering, og siden dette er endringene vi trenger for å tilpasse trekantene til hverandre så brukte vi denne. Vi bruker ikke en homografi som kan gjøre det samme og mer siden det er mer regnekrevende og unødvendig når affine får til alt vi vil. Fremgangen for å bruke affine metodene til opencv var å lage et `boundingRect` rundt trekantene, for så å tilpasse til trekanten med et offset på rektangelet, deretter sendte vi dette inn til opencv sine affine funksjoner. Så slicet vi originalbildet og nullet ut med en maske den delen av mottaker ansiktet som skulle byttes ut, for så å slice tilbake den delen som hadde blitt affine tilpasset

til det nye området. Dette ble gjort i en loop så alle trekantene blir tatt i tur og orden til hele ansiktet er byttet ut som Figure: 6 viser.

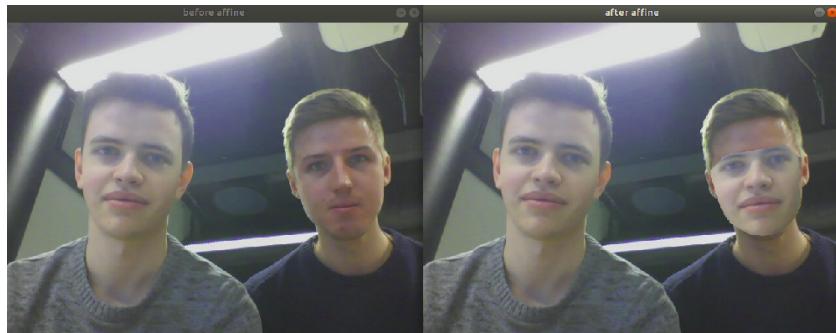


Figure 6: Etter at vi har brukt affine på alle trekantene i den første ansiktet og tilpasset dem til det andre

2.5 Laplace blending

Slik at vi får en finere fargeovergang mellom hodet og det bytta ansiktet bruker vi Laplace blending som forklart i Lecture 2.3. For å lage masken til Laplace blending slår vi sammen svart hvitt maskene i Figure 3 og bruker et gauss-filter på dette, se Figure 7. Størrelsen på filteret avhenger av størrelsen på ansiktet i piksler, slik at små ansikter ikke blir blurret for mye.

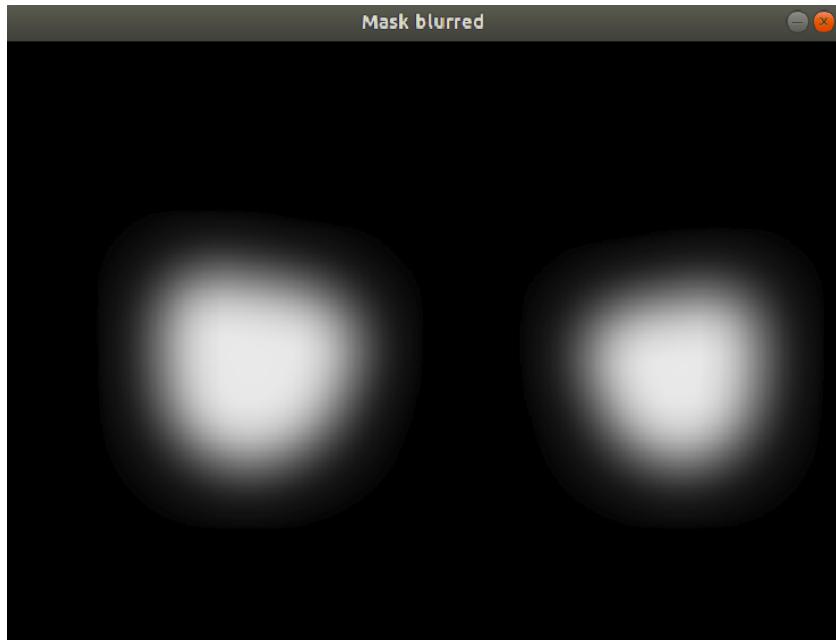


Figure 7: Masken til Laplace blending

Vi prøvde først med en enkel lineær blending, men som forventet ga det oss ikke så veldig bra resultat. Selv etter Laplace blending (Figure 12) så er det fortsatt en merkbar overgang mellom ansikt og panne, men det er det beste resultatet vi klarte å oppnå med kun Laplace blending.

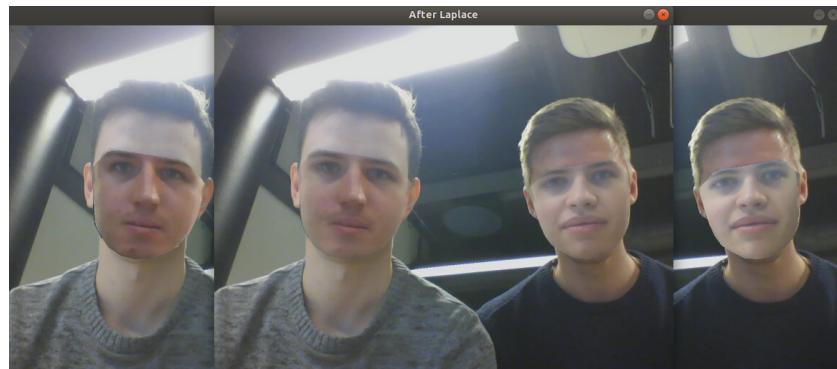


Figure 8: Resultat fra Laplace blending i midten og input-bilde på siden til referanse

3 Resultat

Under kan man se resultatene, hentet fra live video.

Vi syntes resultatet blir ganske bra når vi ser samme vei, men som forventet fungerer det ikke særlig bra hvis vi ser hver vår vei, siden vi ikke tar hensyn til hvilken vei personen ser. Resultatet er også avhengig av god belysning, som også er forventet siden vi ikke tar hensyn til kontrast- og fargeforskjeller.



Figure 9: Resultat: Ser rett frem



Figure 10: Resultat: Avstandsforskjell

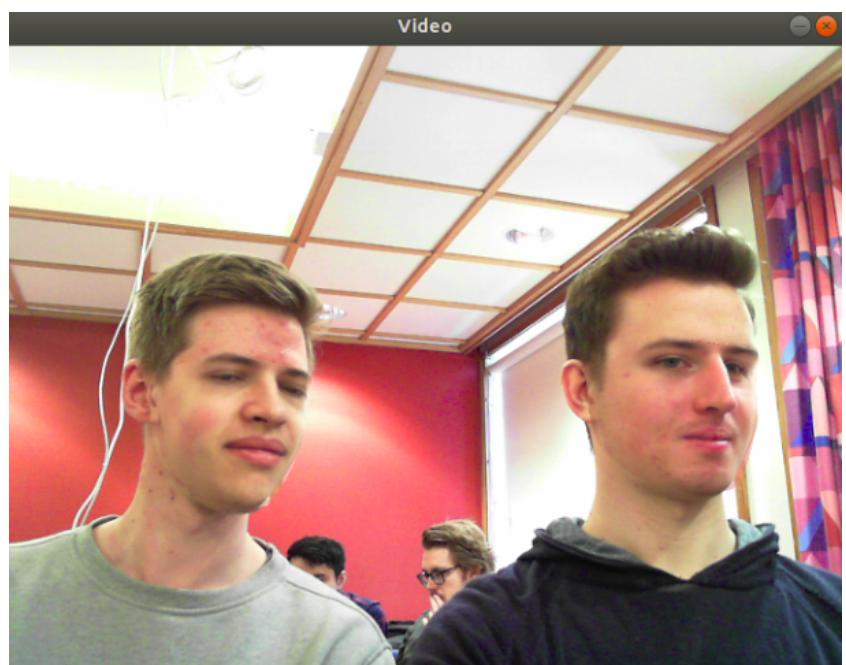


Figure 11: Resultat: Begge ser samme vei



Figure 12: Resultat: Vi ser hver vår vei

Helt mot slutten prøvde vi oss på å håndtere at vi kan se hver vår vei, vi prøvde å løse det ved å ta med et punkt som er midt på nesa i delaunay triangulering (Figure 13). Det løste ikke problemet helt, det blir bedre når vi ser hver vår vei, men ellers syntes vi det så bedre ut når vi ikke tok med dette punktet. Ansiktet blir morphed litt fordi nesen vår ikke er plassert likt i ansiktene våres, og da ligner resultatet mer på ansikts-morphing enn ansiktsbytte, og vi velger da å ikke ta med dette punktet.



Figure 13: Litt eksperimentering

4 Problemer underveis og forbedringer

Vi prøvde å ta hensyn til fargeforskjeller i ansiktene ved å bruke en metode vi fant som heter ”Poisson image editing”, men det var altfor omfattende med tanke på tiden vi hadde, og det er en veldig regnekrevende metode som nok ville gjort programmet veldig tregt, og ville nok ikke da fungert med live-kamera.

Det siste vi gjorde var å få det til å fungere med live video, tidligere jobbet vi bare med ett enkelt bilde. Det å gjøre det med video ga oss noen problemer, som at diverse verdier ble satt til 0 og funksjoner som da feilet, samt 100% CPU-bruk etter at programmet hadde kjørt en stund. Vi fikk løst disse problemene, men alle ble kanskje ikke løst på den beste måten. I noen tilfeller der programmet tidligere krasjet så returner vi bare input-bildet, så det hender da at det er én frame innimellom der ansiktene ikke er byttet skikkelig.

Vi hadde problemer med nesten hver eneste funksjon vi skrev, og mye av koden ble skrevet om på nytt, og ting gikk ikke så lett som vi hadde håpet på, men vi ble fornøyd med resultatet til slutt.

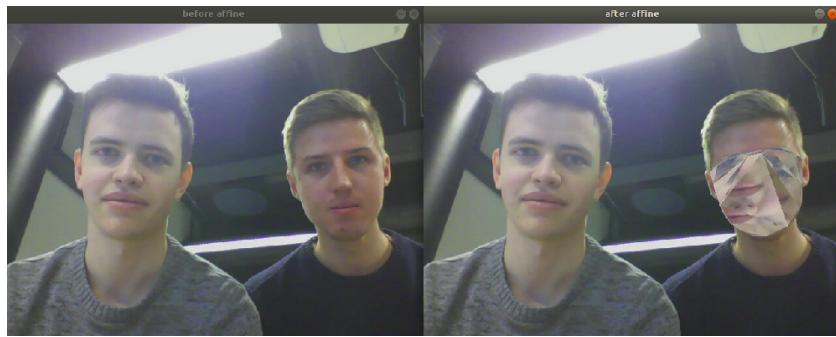


Figure 14: Litt problemer med affine transformen

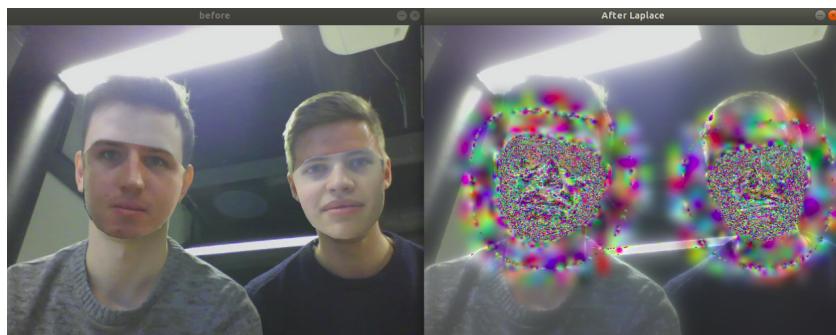


Figure 15: Laplace blending var ikke problemfritt selv om vi har gjort det i en tidligere lab

References

- [1] Vahid Kazemi and Josephine Sullivan *One Millisecond Face Alignment with an Ensemble of Regression Trees*. 2014.
- [2] Jarvis, R. A. *On the identification of the convex hull of a finite set of points in the plane*. 1973. <https://www.sciencedirect.com/science/article/pii/0020019073900203?via%3Dihub>
- [3] de Berg M., van Kreveld M., Overmars M., Schwarzkopf O.C. *Delaunay Triangulations*. In: *Computational Geometry*. Springer, Berlin, Heidelberg 2000