

# Interaktiv kunstinstitution ved brug af computervision teknikker

ROSKILDE UNIVERSITET

4. SEMESTER, FORÅR 2018



VEJLEDER: JOHN PATRICK GALLAGHER

GRUPPENS MEDLEMMER:

Emilie Beske Unna-Lindhard	60741	ebul@ruc.dk
Jacob Nørbæk Krag	61007	jnkrag@ruc.dk
Jesper Arne Erne Jensen	55321	jaej@ruc.dk
Jesper Dreier Sørensen	60636	dreier@ruc.dk
Mia Malene Flarup Hartmann	60624	mimaha@ruc.dk
Nils Müllenborn	61462	nimu@ruc.dk

---

## Abstract English

The purpose of this computer science project is to investigate how computer vision techniques can be used to create interactive art. The project group have had an experimentative approach to the matter of creating interactive art, by using various computer vision strategies. By researching topics such as frame differencing, blob tracking, particles and particle systems, the group has found various usages of pixel manipulation and drawing techniques to make users interact with the various outputs. The project has used tools like webcams, Kinects and computers with different OS. Despite variations in tools, all the programs have been developed with the opensource IDE called Processing. With this project report the group wants to explain their design considerations, describe how their effects work, give an in-depth description of the different programme structures, test the programs and give a conclusion on the overall products there has been produced.

## Abstract Dansk

Formålet med dette datalogi projekt er at undersøge, hvordan computer vision teknikker kan bruges til at lave interaktiv kunst. Projektgruppen har haft en eksperimenterende tilgang til formålet om at lave interaktiv kunst ved at benytte forskellige computer vision strategier. Ved at undersøge emner, såsom frame differencing, blob tracking, partikel og partikelsystemer, har gruppen fundet flere nyttige metoder til at manipulere pixels og teknikker til at tegne, for at få brugerne til at interagere med de forskellige outputs. Der er blevet gjort brug af redskaber såsom webcams, Kinects og computere med forskellige OS. Trods forskelle i redskaber er alle programmerne udviklet i open-source IDE'et Processing. Med dette projekt ønsker gruppen at forklare deres design overvejelser, hvordan deres effekter virker, give en dybdegående beskrivelse af de forskellige programstrukturer, teste metoderne og give en konklusion af det overordnede produkt, som er blevet udviklet.

# Indhold

<b>1</b>	<b>Indledning</b>	<b>4</b>
<b>2</b>	<b>Problemanalyse</b>	<b>6</b>
2.1	Problemstilling . . . . .	6
2.2	Interaktiv kunst . . . . .	6
2.3	Interaktions design . . . . .	6
2.4	Computer Vision . . . . .	7
2.4.1	Frame Differencing . . . . .	9
2.5	Partikelsystem . . . . .	10
2.6	Tracking . . . . .	11
2.6.1	Blob tracking . . . . .	11
2.7	Redskabsbeskrivelse . . . . .	11
2.7.1	Kinect v1 . . . . .	11
2.7.2	Processing . . . . .	13
<b>3</b>	<b>Arbejdsproces</b>	<b>15</b>
<b>4</b>	<b>Beskrivelse af programmet</b>	<b>16</b>
4.1	Program struktur . . . . .	16
4.2	Komponenter . . . . .	16
4.2.1	Frame Differencing . . . . .	16
4.2.2	Blob Tracking . . . . .	18
4.2.3	Partikelsystemer . . . . .	22
4.2.4	Kinect dybde . . . . .	24
4.3	Prototype 1 - Visualisering af bevægelse . . . . .	25
4.4	Prototype 2 - Skrabelod effekt . . . . .	27
4.5	Prototype 3 - Vægmaling . . . . .	29
4.6	Prototype 4 - Partikel interaktion . . . . .	30
4.7	Prototype 5 - Farvesporing . . . . .	32
4.8	Interface . . . . .	35
<b>5</b>	<b>Brugervejledning</b>	<b>40</b>
5.1	Når programmet skal installeres . . . . .	40
5.2	Når programmet køres . . . . .	41
5.3	Eventuelle fejl . . . . .	42
<b>6</b>	<b>Afprøvning</b>	<b>43</b>
6.1	Unittest . . . . .	43
6.1.1	Kinect dybde kamera test . . . . .	43
6.1.2	Frame differencing test . . . . .	46
6.1.3	Unitest og stresstest af blob-programmet . . . . .	47
6.1.4	Fejlmeddelelser . . . . .	49
<b>7</b>	<b>Til og fravalg</b>	<b>50</b>
7.1	OpenFrameworks . . . . .	50
7.2	Processing . . . . .	50
7.3	Kinect v2 . . . . .	50
7.4	Webcam . . . . .	51
<b>8</b>	<b>Diskussion og konklusion</b>	<b>52</b>

**9 Litteraturliste**

**54**

# 1.0 Indledning

Interaktiv kunst er et vidt begreb, som først og fremmest skal indskrænkes og defineres. Interaktivitet kan være mange forskellige ting og hvor stor en del af en installation, der er påvirket af interaktion, kan variere. Dette projekt er en interaktiv kunstinstitution, da det der bliver fremvist af programmet påvirkes af objekter og bevægelse. Nogle af prototyperne har ikke noget visuelt output, medmindre der er bevægelse, mens andre af prototyperne har en visuel effekt uden bevægelse.

Der er forskellige begreber, som dækker over kunstværkers interaktion. Et kunstværk kan være enten passivt eller interaktivt (Seevinck, 2017). Dvs. at kunstværket enten bliver påvirket af tilskuernes deltagelse eller ikke. Interaktiv kunst kan yderligere inddeles i underkategorier. Herunder er begrebet dynamisk-interaktiv kunst og dynamisk-interaktiv(varierende) kunst. Dynamisk-interaktiv kunst reagerer på tilskuernes interaktion og ændrer i en eller anden udstrækning form herefter. I dynamisk-interaktiv(varierende) kunst, ændrer interaktionen i udstillingens udfald. Med dette menes, at interaktionen ændrer, hvad der senere vil blive fremvist ved interaktion (Seevinck, 2017). Altså den interaktion der sker i dag, ændrer hvordan interaktionen med det samme program kommer til at se ud i morgen. Prototyperne i dette projekt går indunder den første kategori dynamisk-interaktiv kunst, da interaktionen ikke ændrer prototypernes struktur.

De forskellige metoder, der er brugt til at udvikle prototyperne bliver beskrevet på tre niveauer. Først beskrives metoderne overordnet set. Her menes, at de beskrives overfladisk, hvad er hovedsagen af den pågældende metode og hvad kan denne bruges til. Metoderne forklares også ud fra pseudo kode, her beskrives de altså på en smule mere konkret niveau. Til sidst beskrives metoderne i kodeafsnittene. Denne forklaring er helt kodenær og er kun et eksempel på, hvordan vi har valgt at benytte metoden.

Prototyperne visualiserer tilskuernes bevægelser ved brug af forskellige metoder, herunder frame differencing, blob tracking og partikler. Inputtet kommer henholdsvis fra en Kinect og den pågældende computers indbyggede webcam. Outputtet bliver i alle prototyperne visualiseret i et display vindue i programmet Processing. Nogle af prototyperne bliver aktiveret af tilskuernes bevægelse såsom prototypen visualisering af bevægelse, hvorimod prototypen partikel interaktion har grafisk indhold, selv uden bevægelse. Altså i partikel interaktion prototypen ændres bevægelses adfærden af programmet, men programmet er også aktivt uden bevægelse. Nogle af prototyperne har ikke det visuelle resultat, som var intentionen. Projektet har mødt udfordringer i form af udstyr og nogle af prototyperne har vist sig sværere at programmere end hidtil antaget. Hver prototype har et separat afsnit, som udover at gennemgå koden for den pågældende prototype, også evaluerer arbejdet med den pågældende prototype. Her menes, at hvert prototype afsnit har en mindre diskussion af, hvorvidt intentionen med prototypen er nået i mål, hvad der mangler for at prototypen ville have været som ønsket og eventuelle forslag til, hvad der kunne arbejdes med for at forbedre prototypen, hvis man skulle arbejde videre med koden. Rapporten har et generelt redegørende afsnit omkring de programmer og det udstyr, der er blevet brugt for at lave projektet. Det er en forudsætning, hvis læseren ønsker at følge kodens logik, at læseren har en forforståelse for Java og Processing. Dvs. selvom rapporten indeholder redegørende afsnit for de benyttede elementer, er det ikke udførligt nok beskrevet, til at en læser uden forståelse for kodning kan følge projektets kode del. Rapporten har, udover de mindre diskussions afsnit omhandlende prototyperne, et overordnet diskussions af-

snit, som diskuterer hele projektet som helhed.

Projektet har formået at skabe flere mindre prototyper, som kan defineres som interaktiv kunst og dermed kan man konkluderende sige, at projektet kan bekræfte sin hypotese. Projektets ambitioner har ændret form, mange gange undervejs, som udfordringer har vist sig, men har stadig produceret et produkt, som opfylder projektets først antaget mål.

## 2.0 Problemanalyse

### 2.1 Problemstilling

*Hvordan kan man ved hjælp af computervision teknikker skabe interaktiv kunst?*

**Arbejds spørgsmål:**

1. Hvordan kan vi bruge motiontracking og farve til at visualisere kroppens bevægelser?
2. Er det muligt at skabe et interface, hvor det er muligt at tilgå flere prototyper i Processing?
3. Hvilke redskaber kan bruges til at lave interaktiv kunst og hvad er fordele/ulemper ved disse?

### 2.2 Interaktiv kunst

Interaktiv kunst er defineret ved kunst, som kræver eller giver brugeren mulighed for at interagere med "kunst" (Seevinck, 2017). Kunst er her sat i anførselstegn, da det ofte er meget individuelt, hvad der opfattes som værende kunst alt efter kilde eller holdninger. Eksempler på interaktiv kunst kan f.eks. være teaterforestillinger, shows af anden karakter, kunstinstallationer f.eks. med en computer, som er tilfældet her. Det er forskelligt, hvad der defineres som interaktiv kunst deltagelse fra tilskuerne kan være påkrævet eller blot tilføje nogle effekter til kunsten. Hvis der er tale om en installation, som først aktiveres ved tilskuers deltagelse, er installationen afhængig af deltagelsen. Hvorimod f.eks. et teaterstykke, som inddrager tilskuere, hvis de ønsker at være en aktiv del, ikke er direkte afhængig af tilskuernes deltagelse.

Ernest Edmond argumenterer for, at kunst først kaldes for interaktiv, hvis deltagelse fra tilskuere er en essentiel del af kunstværket (Seevinck, 2017). Altså kan kunst, i hans øjne, ikke defineres som interaktiv kunst, hvis ikke det er afhængigt af tilskuernes deltagelse. Wolf Lieser beskriver interaktiv kunst som en udstilling, der inddrager verden helt generelt og at visualisere det på en let opfattelig måde, f.eks. som et billede (Seewinck, 2017). Altså kan en installation, ifølge dette synspunkt, også visualisere et træes bevægelse og stadig være interaktiv kunst, så længe noget eksternt for installationen påvirker resultatet, som bliver visualiseret.

### 2.3 Interaktions design

Hvad er interaktion egentlig? Interaktion kan defineres som udveksling af information mellem to eller flere aktive aktører. Når vi taler om interaktion i programmerings verdenen, er det som udgangspunkt fordi, at den ene af aktørerne er en eller anden form for computer system. Personen, som computeren eller det tekniske system bliver designet til, kaldes for 'brugeren' og det som brugeren benytter kaldes 'systemet' (Noble, 2012). 'Interfacet' er en anden vigtig del, når det kommer til kommunikation i interaktion. Interfacet er fladen mellem brugeren og systemet og det indeholder alt det delte materiale som brugeren og systemet benytter til at sende og modtage meddelelser (Noble, 2012). Det at have et funktionelt, udtryksfuldt og attraktivt interface er vigtigt, når man gerne vil skabe interaktion. Disse elementer er alle vigtige, når der skal skabes en god brugeroplevelse. Alt fra farver, former, lyde, grafik og tekst spiller en stor rolle for, hvad brugeren synes om systemet.

Selvom brugeren foretrækker et pænt interface, så har de brug for et funktionelt interface. Funktionaliteten af et interface er det som gør, at brugeren kan finde ud af at navigere rundt i systemet (Noble, 2012).

## 2.4 Computer Vision

Computer vision (CV), som har været forskningsobjekt i mere end 40 år, beskæftiger sig med at få en computer til at se og forstå den virkelige verden. Dette indebærer at få computeren til at "forstå" et tredimensionelt rum med autentiske bevægelser, mennesker og objekter, og med logik udføre opgaver, som en automation eller imitation af, hvad det menneskelige syn og fornuft udfører (Ikeuchi 2014). Disse opgaver opnås ved at analysere billeder taget fra et kamera repræsenteret af pixels i et array, som også er definitionen af CV. Gøremålet for opgaverne er ofte med henblik på at løse en specifik opgave. Det kunne være at genkende og måle bevægelse, eller at systematisere pixels med samme egenskaber som farve og dybden fra kinecten (Noble, 2012). Kendte eksempler på CV kunne være ansigtsgenkendelse fra smartphones og 'goal line technology' fra fodbold.

Et billede er et array af pixels med en bestemt farve. En pixel er en lille firkant. I Processing er farvemodellen Alpha, Red, Green, Blue (ARGB) med en 8 bit værdi. Alpha er gennemsigtigheden af en pixel. Er A lig med 255, er pixelen ikke gennemsigtig og hvis A er lig med 0 er pixelen helt gennemsigtig. I et program med en bredde på 1280 pixels og en højde på 720 pixels, vil der være bredde \* højde antal pixels, hvilket i dette tilfælde svarer til  $1280 \cdot 720 = 921600$  pixels. Hver af disse 921600 pixels gemmes i et endimensionelt array med en 8-bit farveværdi. I en video med 30 frames per second, vises et nyt array af pixels 30 gange i sekundet. Det vil sige at en video er en sekvens af billeder. Skal vi finde ud af hvor i arrayet en tilfældig pixels plads på et billede er, gælder det at:

```
int [pixel_lokation_i_array]=x+y*bredde_på_billedet
```

Her følger en forklaring af ligningen. Det egentlige array på computeren med 6 pixels i.

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
----------	----------	----------	----------	----------	----------

Figur 2.1: En visualisering af et array med pixel

Et teoretisk billede på 6 pixels, kan for eksempel have 3 pixels i bredden og 2 pixels i højden. hen af bredden er x værdierne og ud af højden er y værdierne. I punkt 1,1 på billedet findes pixel 4 i arrayet. Hvis vi bruger formlen:  $x+y \cdot \text{bredde} = \text{position}$  i arrayet, får vi  $1+1 \cdot 3=4$

0 (0,0)	1 (1,0)	2 (0,2)
3 (0,1)	4 (1,1)	5 (2,1)

Figur 2.2: En visualiseringen af et billede, hvor pixelplaceringen i arrayet er vist med tilhørende koordinater

Hvis billedet er højere, gælder samme formel. For punkt 2,2 er pladsen i arrayet  $2+2 \cdot 3=8$ , for 2,3 gælder  $2+3 \cdot 3=11$  og for 2,4 gælder  $2+4 \cdot 3=14$



0	1	2
3	4	5
6	7	<b>8</b>
9	10	<b>11</b>
12	13	<b>14</b>

Figur 2.3: En visualiseringen af et billede, hvor pixelplaceringen i arrayet er vist

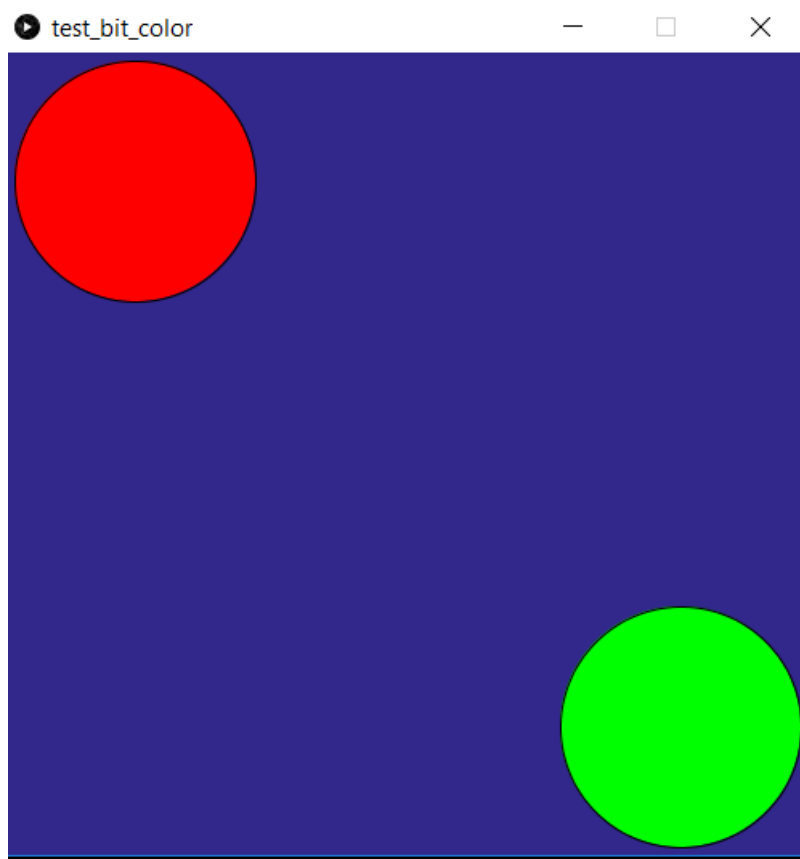
Hvis farven grøn erklæres til at være  $int\ g = 255$ , er det muligt at oversætte denne værdi til den egentlige farveværdi som findes i det endimensionelle array. For at gøre dette, skal man forstå binære tal. Et 1 tal betyder sandt og 0 betyder falsk. Den største værdi for 8-bit er 255 da  $11111111=255$ . Regnestykket ser således ud:  $1*128 + 1*64 + 1*32 + 1*16 + 1*8, 1*4, 1*2 + 1*1 = 255$ . Et 8-bit tal kan gå fra 0 til FF. FF betyder den største værdi for det binære tal.

Den grønne værdi i det endimensionelle array hvor billedet er gemt er: 00000000 00000000 11111111 00000000. Ved hjælp af bit shifting kan vi rykke  $g=255$  8 pladser til venstre. Der er adskillige bitshifting operationer. En af dem er en left shift operator ( $\ll$ ), som skubber et binært tal til venstre x antal gange efter operationen.  $g = 00000000\ 00000000\ 00000000\ 11111111$ .  $g\ll 8 = 00000000\ 00000000\ 11111111\ 00000000$  (Noble, 2012). Ønsker man at finde den grønne farve med en alpha værdi på 255, skal kan man i processing skrive:

```
g<<8 & 0xFF
```

0x betyder at der nu følger et binært tal. (Noble, 2012)

Det er muligt at køre fill()-eksempel i processing skrevet med binære tal. I følgende skærmbilleder ses et program, hvor den røde cirkel er tegnet med fill(00000000, 11111111, 00000000) og den grønne cirkel er tegnet med fill(00000000, 11111111, 00000000).



Figur 2.4: Test program, som viser en rød og grøn cirkel på en blå baggrund



Figur 2.5: Et billede af en rød cirkel der er zoomet så meget ind, at man kan se de enkelte pixels.

Når vi zoomer ind på billedet ses tydeligt, hvordan de små pixel på skærmen, danner den røde cirkel.

I Processing gælder det at der først skal bruges en funktion som hedder 'loadPixels()', før man kan skrive eller læse et 'pixels[]' array. Det som 'loadpixels()', funktionen gør, er at gemme et øjebliksbillede inde i et 'pixels[]' array.

### 2.4.1 Frame Differencing

Frame differencing kan groft sagt bruges til at genkende bevægelse. Det er simpelt forklaret en metode, som sammenligner to billeder (i denne forbindelse kaldet frames) og finder forskellene på disse (Singla, 2014). Metoden benytter sig af at have et referencebillede, som den kan sammenligne det nyeste billede med. Denne billedesammenligning er en af de mest almindelige metoder til

bevægelsesdetektion (Singla, 2014). Frame differencing finder forskellen i to billeder ved at sammenligne farven i hvert enkelt pixel i de to pågældende billeder. I processing har hvert enkelt pixel en RGB-farvekode, som tilsammen danner et billede. Det er denne værdi som sammenlignes for hvert enkelt pixel for at finde farveforskellen. Dermed skaber selv en lille bevægelse en forskel i et pixel array's farvekode og kan dermed visualiseres (Singla, 2014). Ulempen ved denne metode er, at den kun skelner billederne fra hinanden gennem pixel-forskelle i RGB værdierne. Dette er et problem i forhold til udefrakommende faktorer som lys. En ændring i lyset vil lave forskelle i RGB-værdierne og dermed kan metoden visualisere bevægelser, hvor der ikke er en egentlig bevægelse. Metoden bliver ofte brugt sammen med andre metoder, alt efter hvad målet med programmet er.

Frame differencing bliver brugt med meget varierende formål. Udover at kunne bruges til at lave interaktive kunstinstitutioner, bruges det også som en del af overvågningssystemer eller til studier af menneskets kommunikative evner.

## 2.5 Partikelsystem

Partikelsystem er en metode, som går ud på at have en klasse til at styre underklasser (subclasses). Dvs. partikelsystem-metoden benytter sig af nedarvning. Med partikler skrives der en klasse, som beskriver hver enkelt partikels egenskaber (Shiffman, 2012). Det vil sige at partikelsystemer er en samling af individuelt programmerede objekter og dermed vil et velstruktureret partikelsystem være objektorienteret programmering. Partikelsystemer kan have en hvilken som helst form og forskellige egenskaber (Shiffman, 2012). Et partikelsystem består typisk af tre hovedelementer: samlingen af partikler, et element der genererer partikler og et der definerer partiklernes egenskaber (Noble, 2012). Man vil som oftest lave partiklernes egenskaber som en separat klasse, hvor man definerer partiklernes egenskaber og udformning. Herefter vil man generere disse partiklerne i hovedklassen. Dvs. at der vil blive generet en ny partikel for hver frame programmet køres, dermed vil der også komme for mange partikler til, at Processing kan køre programmet uden at blive langsommere, hvilket til sidst vil resultere i at programmet går ned (Shiffman, 2012). Derfor skal partikelklassen også indeholde en regel som "fjerner" partikler igen. Her kan man f.eks. give partikler en bestemt levetid, så de "dør" efter et bestemt tidsrum eller hvis de bevæger sig ind på et bestemt område af skærmen osv. Partiklerne får tilføjet både en lokation, en hastighed, en acceleration samt denne regel, der afgør hvor længe den lever. Denne opdeling af kode, gør hver funktion simpel og dermed lettere at følge. Efter en partikel er død, tjekker man hvorvidt en partikel er død eller levende og fjerner alle de døde partikler fra programmet, så programmet ikke fylder op og kan blive ved med at køre. For at holde styr på alle disse partikler benyttes en `ArrayList` (Shiffman, 2012). Herefter kan man have endnu en klasse som genererer partikelsystemer. Dvs. en klasse som genererer partikelsystem og en underklasse som genererer partikler, altså et system af systemer. Denne struktur er en form for nedarvning, da partikelsystem klassen i denne prototype arver egenskaberne af underklassen (Shiffman, 2012).

## 2.6 Tracking

Tracking er en proces hvor man identificere et eller flere objekter, som er i bevægelse indenfor kameraets "synsfelt". En hånd eller et farvet objekt er typisk brugte eksempler, på objekter som kan trackes (Noble, 2012). Tracking er traditionelt en undergren af computer vision, da det er en teknik der automatiserer adskillige opgaver, som mennesker almindeligvis udfører. Vi har dog valgt at adskille de to emner fra hinanden, da tracking er blevet et stort emne for sig selv med eksempelvis gps-tracking.

### 2.6.1 Blob tracking

Blob tracking er en metode, som kan benyttes til at spore et objekt. Når en computer skal spore en genstand, der bevæger sig, kan det være en god idé at lave et midlertidig objekt. Dette objekt har til formål at følge de pixels, som opfylder bestemte givne kriterier. Dette midlertidige objekt kan kaldes en blob. En blob kan altså tage form af en hånd, et ansigt eller noget helt tredje. Et kriterie for at finde en blob er, om der kan findes en forskel i pixels. Det kunne eksempelvis være en ensfarvet baggrund og et menneske. I dette tilfælde er det muligt at tracke alle pixels, der ikke er i baggrunden, eller alle pixels inden for en bestemt grænseværdi i forhold til farven for hver pixel.(Noble, 2012)

En blob findes ved, at finde grupperinger af ensartede pixels. Blob tracking er at følge den bevægende genstands position i billedet med nøjagtighed. Blob tracking er en overskuelig opgave i kontrollerede omgivelser, men i et varierende og ukontrolleret miljø, kan det vise sig en sværere. For eksempel hvis der er to objekter som er magen til hinanden, såsom en person der går forbi i baggrunden. Det er muligt at følge flere objekter, men matematisk og computationelt er det en mere kompliceret opgave. (Noble, 2012)

## 2.7 Redskabsbeskrivelse

Projektet har siden begyndelsen haft en klar vision om, at skabe et interaktivt kunstværk, som fanger brugernes opmærksomhed. Projektet begyndte med at researche, hvilke sensorer og programmer som kunne bruges til, at opnå dette mål. Research af programmer tog en del tid af projektets opstart. Det efterfølgende afsnit redegør for og uddyber valget af Xbox 360 Kinect version 1414 og Processing i et designmæssigt perspektiv. Webcammet i computeren er et kamera, som fungerer efter de principper, som er forklaret i Computer Vision afsnittet, derfor er webcammet ikke yderligere redegjort for i dette afsnit.

### 2.7.1 Kinect v1

I dette projekt har vi anvendt et Kinect kamera, som er udviklet af Microsoft til Xbox spillekonsoller. Der findes to versioner af Kinecten, Kinect v1 og Kinect v2, som er udviklet til henholdsvis Xbox 360 og Xbox ONE. Begge versioner har en tilnærmelsesvis ens funktion, som genererer en strøm af data i et endimensionelt array. Kinecten er oprindeligt kun produceret til Xbox spillekonsoller, men Microsoft har senere udgivet en adapter der kan levere data fra en Kinect via en USB-indgang. Microsofts Kinect er et særligt kamera, der ikke kun opfanger et almindeligt RGB billede, men også opfanger en dybdeværdi for hver RGB pixel. Dybdeværdien er angivet ved et tal inden for et

bestemt interval der indikerer et objekts afstand fra kameraet.



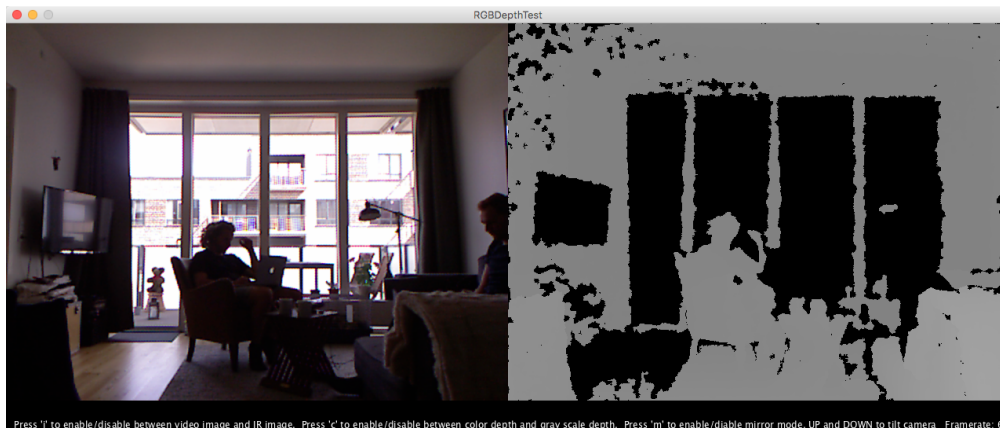
Figur 2.6: Billede af Kinect v1

Kinect v1 anvender et RGB-kamera, en infrarød (IR) projektor og en IR sensor til at opfange billeder og data (J. Lange et al, 2009). Kinecten udregner en afstandsværdi for et objekt (en pixel af gangen) ved at måle afstanden fra IR projektoren til IR sensoren. IR sensoren er en monokrom CMOS sensor der opfanger styrken af det infrarøde lys og kan på denne måde vurdere afstanden af objekter (Canon). Alle beregninger og bearbejdning af data fra sensoren og kameraet foregår på en samling af flere computerchips udviklet af PrimeSense. Chippene kan blandt andet synkronisere billederne fra både IR sensoren og RGB kameraet, genererer et endimensionelt array med al optaget data - en pixel af gangen - og beregne afstandsværdien for hver pixel. PrimeSenses chips har yderligere funktioner som for eksempel at genkende formen af en menneskekrop, men da dette ikke er nemt tilgængeligt via USB-adapteren til en computer, benyttes funktionerne ikke i dette projekt og er derfor ikke relevante. (Microsoft)(D. Takahashi, 2013)

	Kinect 1
RGB kamera	640 x 480 @30 fps
IR sensor	320 x 240
Maks. dybde afstand	~4.5 m
Min. dybde afstand	40 cm
Horizontale "Field of view"	57 grader
Vertikale "Field of View"	43 grader
USB standard	2.0

Figur 2.7: En tabel over tekniske specifikationer af Kinect v1 (M. Szymczyk, 2014)

På figur 2.8 og figur 2.9 kan man se det output Kinect v1 giver. På venstre hånd af figur 2.8 vises et skærbillede af det billede RGB kameraet optager, på højre hånd af figuren vises den rå dybde data repræsenteret ved en bestemt tone af grå – jo længere væk et objekt er jo mørkere bliver det.



Figur 2.8: Skærbillede af output fra Kinect v1

På venstre hånd af *figur 2.9* vises det billede IR sensoren optager som anvendes til afstandsberegning af objekter. På højre hånd af figuren vises igen den rå dybde data, denne gang repræsenteret ved en farve frem en tone af grå. Dette projekt anvender en blanding af de forskellige billeder Kinecten leverer.



Figur 2.9: Skærbillede af output fra Kinect v1

## 2.7.2 Processing

Processing er et af de første open source projekter, som er specifikt designet til at gøre processen ved at lave interaktive grafiske applikationer mere simple, så folk der ikke nødvendigvis er uddannede programmører lettere kan lave kunsthinstallationer (Noble, 2012). Processing er baseret på programmeringssproget Java, dog i en mere simplificeret udgave, som gør det lettere for nye brugere at lære sproget. Eksempelvis kan en bruger benytte sig af 'ellipse' eller 'rect'-funktionerne, hvis de ønsker at tegne en cirkel eller en firkant. Disse er blot en af mange forsøg på at simplificere java-sproget, som findes i Processing. Der er titusindvis af brugere, herunder studerende, artister og designere, som hovedsageligt bruger Processing til at lære Java og lave prototyper. (<https://processing.org/>) I Processing findes udvidelser i form af 'libraries' og 'tools' og det er i særdeleshed her, IDE'en er aktuel for os. (<https://processing.org/reference/environment/>). Daniel Shiffman, som er kendt for sine hundredevis af programmerings-videoer på Youtube med næsten en halv million følgere, (<https://www.youtube.com/user/shiffman>) har udviklet et bibliotek til Kinect v1 og v2 til Mac. Senere har Thomas Sanchez udviklet videre på disse biblioteker, så de virker på Windows.

## Setup() metoden

Processing har to fundamentale metoder: `setup()` og `draw()`. `Setup()` metoden bliver kaldt én gang når applikationen startes. `Setup()` metoden sørger for, at alt den information, som du skal bruge i resten af applikationen, er klar til brug, og det er i denne der defineres værdier såsom skærmstørrelse og loades medier såsom billeder og fonte. Der kan kun være én `setup()` funktion per program, og den skal ikke kaldes, efter programmet er startet. Hvis størrelsen på programvinduet skal være en anden, end den størrelse den har som udgangspunkt, skal det defineres i den første linje af `setup()` metoden. Variabler som er deklareret inde i `setup()` kan ikke tilgås i andre funktioner, inklusiv `Draw()` (Noble, 2012).

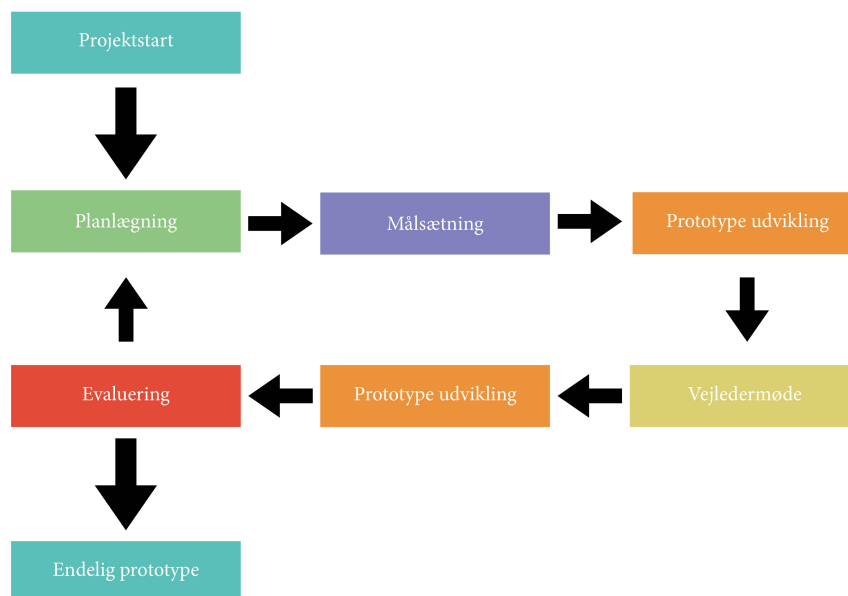
## draw() metoden

`Draw()` metoden er hvor 'tegningen' af programmet finder sted. Alt du skriver i `draw`, vil blive kaldt 60 gange i sekundet som programstandard, så længe programmet kører. `Draw()` bliver kaldt automatisk og bør ikke blive kaldt eksplicit. Alle Processing programmer opdaterer skærmen i slutningen af `draw()` funktionen og aldrig før. Antallet af gange `draw` bliver kaldt i sekundet kan ændres ved hjælp af `frameRate()` funktionen og det er muligt at ændre frameraten helt ned til 25, uden at det har betydning for det visuelle udtryk. Der kan kun være én `draw()` funktion i hvert program og `draw()` skal eksistere, hvis du ønsker, at koden skal kaldes gentagende gange, eller hvis du ønsker at benytte events såsom `mousePressed()` (Noble, 2012).

## 3.0 Arbejdsprocess

Da vi er seks mennesker i gruppen, valgte vi fra start af at opdele os i mindre programmeringsgrupper af to, så vi på den måde kunne få produceret mere og samtidigt bruge hinanden til feedback. Til at starte med afprøvede vi, at vi hver uge fik en ny programmeringspartner. Idéen var, at alle skulle prøve at programmere med hinanden, fordi vi tænkte, at det muligvis kunne gavne processen, da alle i gruppen har forskellige kompetencer og områder hvor, at vi hver især er bedst. Dette forsøg varede dog ikke mere end to uger, da vi hurtigt fandt ud af, at det næsten blev en større stressfaktor at skulle skifte hver partner hver uge end, at det bidragede til at bryde folk ud af deres vandte arbejdsrutiner. Det viste sig, at grupperne hurtigt blev opslugt af at arbejde med hver deres effekter, og det endte derfor ud i, at grupperne arbejdede videre med deres egne effekter, men stadig brugte de andre medlemmer af gruppen som sparringspartnere, hvis vi stødte ind i problemer. Vi har altså derfor hver især arbejdet på visuelle effekter, som vi til sidst i processen har arbejdet på at sætte sammen til ét samlet program.

Ud fra de erfaringer som vi hurtigt skabte os igennem de første par uger, mener vi ikke, at en model som vandfaldsmodellen vil gavne et projekt som dette, da det er svært at forudsige på forhånd, hvilke behov og udfordringer man vil støde på undervejs. Dette vil gøre det svært at ligge en målrettet plan fra start til slut. Vi har derimod, benyttet en iterativ tilgang til processen (se figur: 3,1), hvor udvikling og afprøvning af prototyper har været kernen. I og med, at vi fra start har delt os op i mindre programmeringsgrupper, har det givet os mulighed for at udvikle små prototyper, som vi har kunnet arbejde på at forbedre og ikke mindst koble komponenter fra hinandens prototyper sammen.



Figur 3.1: *En model over gruppens iterative arbejdsprocess*

Et faktum er dog, at vi arbejder med interaktiv kunst, som er et subjektivt fænomen. Vi har i modsætning til normale forhold, ikke nogle krav fra fremtidige brugere, som vi skal arbejde på at opfylde. Det er derfor svært for os at måle på, om der er nogle behov som er blevet opfyldt, da denne vurdering er subjektiv og dermed op til det enkelte individ at vurdere.



## 4.0 Beskrivelse af programmet

### 4.1 Program struktur

Det endelige program består af en række prototyper, som hver visualiserer bevægelse efter forskellige metoder. Disse prototyper er samlet i et program, som her kaldes for grænseflade. Hver af disse metoder beskrives separat uafhængigt af vores prototyper og forklares i afsnittet komponenter. Komponenterne er beskrevet ud fra simplificeret pseudo kode og en tilhørende gennemgang af et konkret kodeeksempel. Efterfølgende gennemgås prototyperne en for en. Prototype-afsnittene indeholder først en introduktion til prototypen, dvs. hvad var målet med netop denne prototype og hvordan har vi valgt at visualiserer den ønskede effekt. Herefter gennemgås hvordan denne prototypen bruger komponenterne og til sidst en delkonklusion på den pågældende prototype. Delkonklusionen er en mindre diskussion af, hvorvidt intentionen med programmet er opnået, hvor der er eventuelle fejl og mangler i koden, samt hvad man kunne gøre, hvis man skulle arbejde videre med prototypen.

### 4.2 Komponenter

For at opnå det mest tilfredsstillende resultat ved brug af det samelede program, forventes det at følgende gør sig gældende:

- Et stabilt eller fastgjort kamera.
- Et stabilt lys.
- Ensfarvet baggrund, med farvekontrast til forgrunden.

#### 4.2.1 Frame Differencing

Her er beskrevet en pseudokode for en frame differencing algoritme:

```
1 Indlæs billede 1
2 Indlæs billede 2
3 FOR hver pixel i billede 1
4     Udtag RGB værdier
5 FOR hver pixel i billede 2
6     Udtag RGB værdier
7 Udregn billede forskel = nuværende billede - tidligere billede
8 IF der er forskel mellem billederne
9     Vis forskel i billede 3
```

Nedenstående ses tre illustrationer af, hvordan frame differencing sammenligner to billeder og skaber et tredje ud af forskellene



Figur 4.1: Til venstre ses billede 1 og til højre ses billede 2. Herunder ses resultatet af frame differencing. I dette eksempel har vi valgt at vise forskellen i rød farve. Hvis der ikke er forskel i de to billeder, bliver farven sort.



Figur 4.2: Her er et muligt resultat af frame differencing mellem to billeder. Dette er resultatet af figur 4.1

Nedenstående ses et konkret eksempel på, hvordan en frame differencing kode kan skrives i Processing.

```

1 PImage frameDiff(PImage billede 1, color [] billede 2) {
2     billede 1.loadPixels();
3     PImage displayBillede = createImage(billede 1.width, billede 2.height, RGB);
4     displayBillede.loadPixels();
5     for (int i =0; i<billede 1.pixels.length; i++) {
6         color currColor = billede 1.pixels[i];
7         color prevColor = billede 2[i];
8
9         int currR = (currColor >> 16) & 0xFF;
10        int currG = (currColor >> 8) & 0xFF;
11        int currB = currColor & 0xFF;
12
13        int prevR = (prevColor >> 16) & 0xFF;
14        int prevG = (prevColor >> 8) & 0xFF;
15        int prevB = prevColor & 0xFF;
16
17        int diffR = abs(currR - prevR);
18        int diffG = abs(currG - prevG);
19        int diffB = abs(currB - prevB);
20
21        displayBillede = diffR+diffG+diffB;

```

```

22     displayBillede.pixels[i] = color(255,0,0);
23     }}
24     return displayBillede;
25 }

```

Denne kode er et eksempel på, hvor frame differencing metoden er benyttet i vores program. Koden gør brug af to billeder (billede 1 og billede 2) til sammenligningen og skaber ud af denne sammenligning det tredje billede, som her kaldes 'display billede'. Vi laver et PImage display billedet ved hjælp af createimage. Createimage får den ønskede størrelse på billedet, som er samme størrelse som billede 1. Vi loader alle pixels fra vores nye displayBillede og laver en forløkke, som løber igennem hele arrayet med pixels fra billede 1. Vi sætter currColor til at være pixels i billede 1 arrayet og sætter prevColor til at være billede 2 arrayet. Herefter definerer vi de forskellige farver med bit manipulation. Det gøres både for de farver, som er i det nutidige billede og for de farver som er i det tidligere billede. Vi trækker de tidligere farver fra de nutidige farver for at få forskellen. Der bliver i denne del af koden brugt bitmanipulation i stedet for de definerede farver i Processing. Det handler om ren og skær effektivitet, der gør programmet hurtigere og dermed visualiseringen bedre. Farve forskellen regnes sammen på linje 21. Funktionen i linje 24 returnerer et PImage kaldet for displayBillede.

Denne funktion kan også bruges ud fra et todimensionelt array. Her skal blot tilføjes et par linjer kode, som omregner en pixels plads til dens plads i et todimensionelt grid. Denne udregning er forklaret i afsnittet "Computer Vision".

## 4.2.2 Blob Tracking

Følgende program går kort sagt ud på at finde pixels, der tilnærmelsesvis har samme RGB farveværdi som en given farve, der skal spores. Herefter skal der laves et midlertidigt objekt (blob'en), som har til formål at spore dette midlertidige objekt, hvis det skulle bevæge sig.

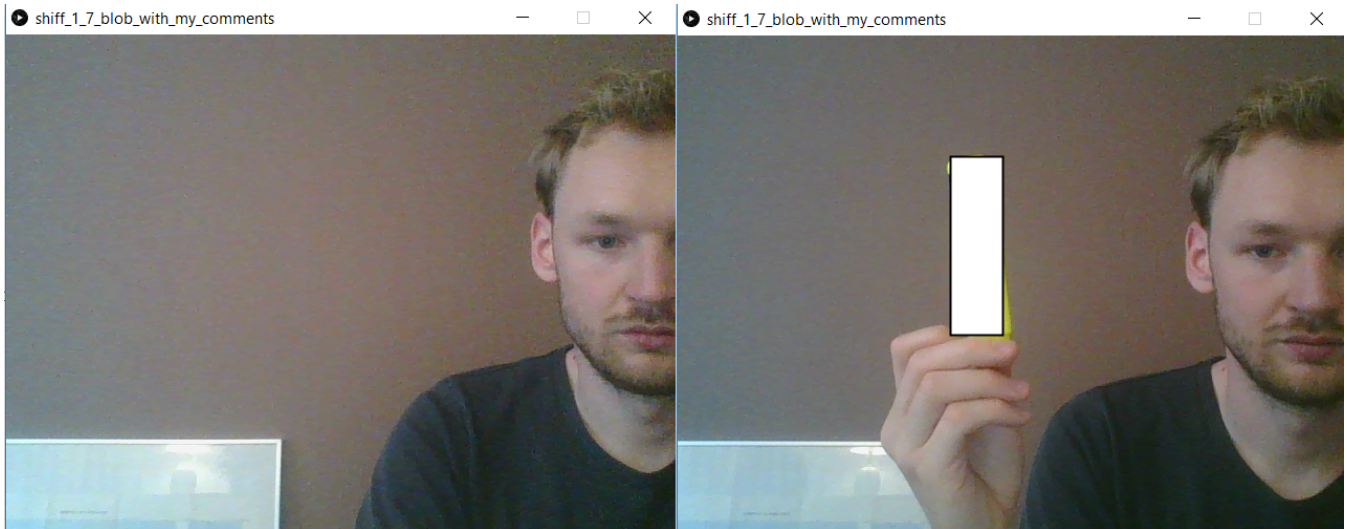
Pseudo-kode til blob-tracking:

```

1  Indlæs et billede der skal analyseres
2  Erklær hvilken farve der skal spores
3  Loop gennem billedets x og y koordinater (dobbelforløkke)
4  FOR hver pixel i billedet
5     Uddrag RGB værdier
6  Find den absolutte forskel for hver pixel mellem [farven der ønskes at spore] og
   ↪ den [fundne farve for hver pixel i billedet].
7  IF den samlede farveforskelle for en pixel er mindre end en given GRÆNSEVÆRDI for
   ↪ farven
8     FOR hver af de pixel der er inde for farve-GRÆNSEVÆRDI
9         og IF antal af fundne pixels > var størrelse
10            tilføj et midlertidigt objekt
11 Hop ud af den dobbelte forløkke og vis det midlertidige objekt(er) på billedet

```

Nedenfor ses to billeder, hvor der spores efter gul farve RGB(204, 110,228). I første billedet er der ikke en gul farve. I næste billede holdes en gul tusch, som spores med en firkant af programmet.



Figur 4.3: Til venstre ses et billede af en baggrund med en person og et billede uden en gul tusch og til højre ses samme billede hvor en gul tusch bliver holdt

Følgende blob-tracking program har taget udgangspunkt i Daniel Shiffmans gennemgang af computer vision på youtube. Især følgende video: 11.7: Computer Vision: Blob Detection - Processing Tutorial ( <https://www.youtube.com/watch?v=ce-212wRq08>)

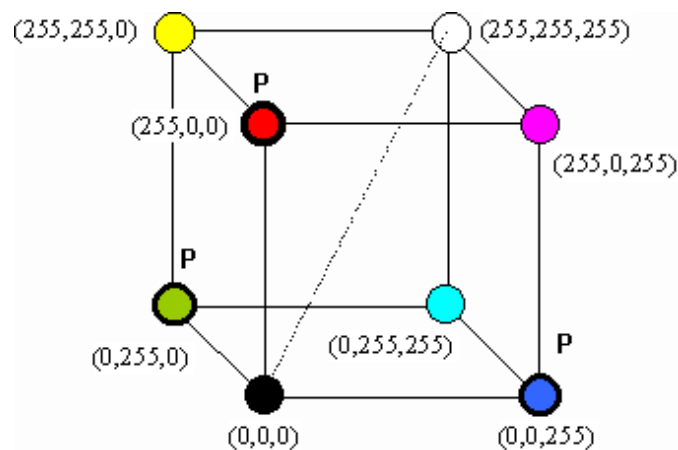
I void draw() starter vi ud med at load alle pixels fra webcammet og oprette et billede, der indeholder billederne fra webcammet.

```

1 void draw() {
2   video.loadPixels();
3   image(video, 0, 0);

```

Næste skridt er, at finde farveværdierne for hver pixel, som er forklaret tidligere i projektrapporten. I computer vision er en blob en samling af pixels, der opfylder nogle bestemte kriterier. I vores tilfælde er kriteriet, at RGB-niveauerne skal være tæt på en bestemt RGB-grænseværdi. I model 4.4 ses en illustration af, hvordan RGB-farver er distanceret fra hinanden i et tredimensionelt rum. Forskellen for en farve findes ved at finde den absolutte forskel mellem RGB-værdierne. Modellen er lidt misvisende, da en diagonal linje er tegnet mellem den sorte og hvide værdi. RGB-forskellen findes ved ikke at gå diagonalt på modellen. så fra tyrkis til gul er der  $255+255=510$  forskel.



Figur 4.4: RGB model taget fra (<http://courses.cs.vt.edu/~cs4624/s98/sspace/imgproc/index.html>)

$$(R2 - R1) + (G2 - G1) + (B2 - B1) = \text{RGB-forskellen}$$

For at finde pixels, der opfylder kravene for at danne en blob, er det nødvendigt at finde RGB-værdien for hvert pixel, i det array der skal undersøges, og sammenligne disse værdier med den farve, der ønskes at spore.

```

1  for (int x = 0; x < video.width; x++ ) {
2      for (int y = 0; y < video.height; y++ ) {
3          int loc = x + y * video.width;
4          color currentColor = video.pixels[loc];
5          int r1 = (currentColor >> 16) & 0xFF;
6          int g1 = (currentColor >> 8) & 0xFF;
7          int b1 = currentColor & 0xFF;
8          int r2 = (trackColor >> 16) & 0xFF;
9          int g2 = (trackColor >> 8) & 0xFF;
10         int b2 = trackColor & 0xFF;
11
12         float d = distSq(r1, g1, b1, r2, g2, b2);

```

Vores to float distSq metoder er meget tæt på Processings egen dist-funktion, men der burde være en fordel i hastigheden ved at skrive dem selv. Vi gør brug af method overloading, som er muligt i Java hvis to metoder hedder det samme, men har forskellige argumenter. I Java kan du have to forskellige funktioner med samme navn, men programmet ved stadig, hvilken den skal kalde.

Som ses i modellen kan farver have en stor eller lille distance til en anden farve. Farveforskellen er tredimensional (RGB) og for distancen mellem blob's er forskellen todimensional (x,y). Vi ganger hver forskel med sig selv, da vi ønsker det absolutte tal, som forskellen. Dette kommer vi rundt om ved at gange grænseværdien med sig selv.

Dette er for blob distance.

```

1  float distSq(float x1, float y1, float x2, float y2) {
2      float d = (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1);
3      return d;
4  }

```

Dette er for farve distance.

```

1  float distSq(float x1, float y1, float z1, float x2, float y2, float z2) {
2      float d = (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) + (z2-z1)*(z2-z1);
3      return d;
4  }

```

Hvis d er mindre end farve-grænseværdien ganget med sig selv er en pixel kvalificeret til at være en blob.

Hvis distancen mellem blobs er indenfor et distThreshold former de en blob tilsammen via isNear-funktionen.

Hvis der ikke er fundet en blob i forvejen, bliver der tilføjet en blob til arrayListen. Derefter gennemgår programmet kriterierne for at lede efter en ny blob. Hvis kriteriene for en ny blob er opfyldt, leder programmet efter den eksisterende blob for at tjekke, om de tilsammen skal danne en blob via isNear-metoden i Blob-klassen. Hvis dette ikke er tilfældet tilføjes en ny blob til arrayListen.

Til sidst skal alle blobs vises via blob.show() funktionen, hvis der er mere end 500 fundne pixels.

```

1  if (d < threshold*threshold) {
2      boolean found = false;
3      for (Blob b : blobs) {
4

```

```

5         if (b.isNear(x, y)) {
6             b.add(x, y);
7             found = true;
8             break;
9         }
10    }
11    if (!found) {
12        Blob b = new Blob(x, y);
13        blobs.add(b);
14    }
15 }
16 }
17 }
18
19 for (Blob b : blobs) {
20     if (b.size() > 500) {
21         b.show();
22     }
23 }

```

I Blob klassen erklærer vi, at en blob har fire floats, som er minimum x og y samt maksimum x og y. Disse variabler bruger vi til at definere en blobs position, som vil tage form som en usynlig firkant, der sporer en angiven farves position ved at overlapse og følge farven.

```

1 class Blob {
2
3     float minx;
4     float miny;
5     float maxx;
6     float maxy;
7
8     Blob(float x, float y) {
9         minx = x;
10        miny = y;
11        maxx = x;
12        maxy = y;
13    }

```

Nedestående void show viser en firkant ved de fundene koordinater.

```

1 void show() {
2     stroke(0);
3     fill(255);
4     strokeWeight(2);
5     rectMode(CORNERS);
6     rect(minx, miny, maxx, maxy);
7 }

```

For at kunne tillægge blob-objekter koordinater, finder vi den mindste x værdi for en pixel, der er med i en blob, hvilket vil sige den pixel, der er indefor thresholden længst til venstre. Dette gør vi ved at bruge Processings indbyggede funktioner min og max, som finder den største og mindste værdi. I 'float size()' definerer vi størrelsen på vores objekt, så denne kan tilgås i draw klassen, hvor vi kan bestemme en minimal størrelse for en blob, før den bliver vist.

```

1 public void add(float x, float y) {
2     minx = min(minx, x);
3     miny = min(miny, y);
4     maxx = max(maxx, x);
5     maxy = max(maxy, y);
6 }

```

```

7  float size() {
8      return (maxx-minx)*(maxy-miny);
9  }

```

Med 'isNear' funktionen opstiller vi parametre for, hvor tæt blobs må være på hindanden fra deres centrum, som er defineret med 'cy' og 'cx' koordinater ("center x", "center y"). Hvis objekterne står tættere sammen, end en given grænseværdi, bliver objekterne sammensluttet til et nyt objekt.

```

1  float cx = (minx + maxx) / 2;
2  float cy = (miny + maxy) / 2;
3
4  float d = distSq(cx, cy, x, y);
5  if (d < distThreshold*distThreshold) {
6      return true;
7  } else {
8      return false;
9  }

```

### 4.2.3 Partikelsystemer

I dette afsnit beskrives et overordnet partikelsystem der anvendes i nogle af vores prototyper. I dette partikelsystem, bliver partiklerne genereret ud fra et oprindelsespunkt og bliver tildelt en tilfældig hastighed i en given retning. Alle partikler påvirkes af en given kraft og tildeles en "levetidsværdi". En partikel kan dø og blive fjernet ud fra bestemte parametre. I partikelsystemets hovedklasse, foregår følgende:

- 1 Partikelsystemets hovedklasse kaldes og et nyt partikelsystem genereres
- 2 Nye partikler tilføjes ved hver iteration af `draw()`
- 3 Partiklernes opførsel og bevægelse udføres ved hver iteration af `draw()`

Partikelsystemet styres på følgende måde:

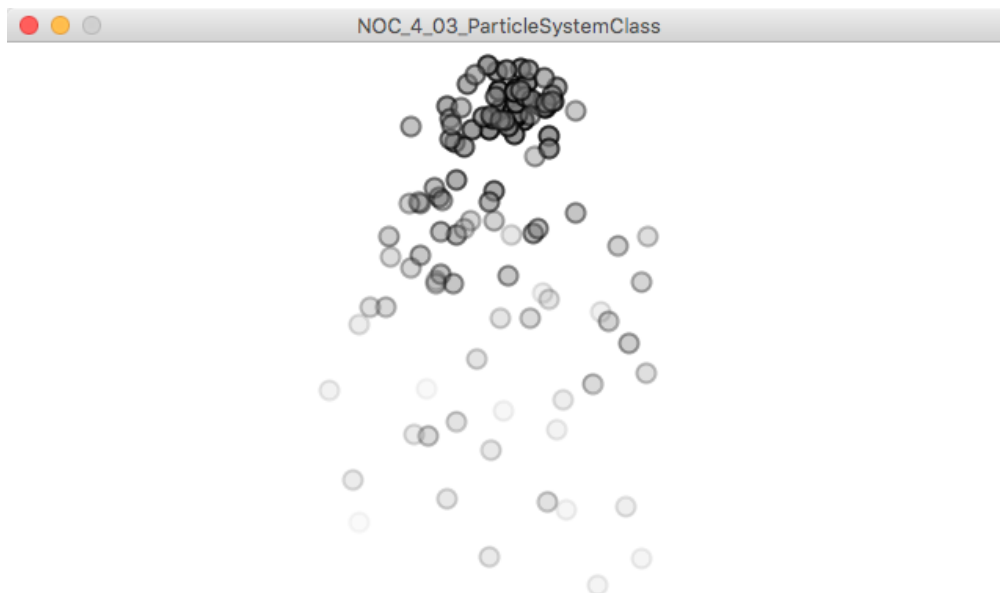
- 1 Et array med alle partiklerne i defineres og genereres
- 2 Et oprindelsespunkt `for` partiklerne genereres
- 3 En metode der kaldes i hovedklassen genererer partikler ved oprindelsespunktet
- 4 En metode der kaldes i hovedklassen udfører partiklernes bevægelse/opførsel, her
  - besluttet også om en partikel skal bevæge sig eller ej: For alle partikler i
  - partikelarrayet udføres deres opførsel, hvis partikelen er erklæret død,
  - fjernes den fra partikelsystemet.

En partikels egenskaber er defineret på følgende måde:

- 1 Accelerationen der påvirker partiklens bevægelse defineres
- 2 Den retning og hastighed en partikel bevæger sig i defineres
- 3 Den location en partikel har bliver givet en variabel
- 4 Den "livstidsværdi" en partikel har, når den genereres, defineres
- 5 Metoden der kaldes i partikelsystem klassen som opdaterer partiklens opførsel
  - defineres:
- 6 Først udregner metoden partiklens bevægelse ud fra acceleration og retning og
  - partiklen får en ny location. Partiklens "livstidsværdi" reduceres med 2.

- 7 Næst tegnes partiklen ved den nye location.
- 8 Hvis en partikel har en "livstidsværdi" mindre end nul, erklæres partiklen **for**  
 → død og slettes

På *figur 4.5* ses et skærmbillede af et eksempel på partikelsystem hvor partiklerne er genereret ud fra et oprindelsespunkt og bliver tildelt en tilfældig hastighed i en tilfældig retning, med en kraft der trækker partiklerne nedad.



Figur 4.5: På billedet ses et skærmbillede af et partikelsystem hvor partiklerne er genereret ud fra et oprindelsespunkt og bliver tildelt en tilfældig hastighed i en tilfældig retning, med en kraft der trækker partiklerne nedad

Koden for dette slags partikelsystem i Processing kan se ud som følgende. I hovedklassen generes partikelsystemet:

```
1 ps = new ParticleSystem();
```

Yderligere kaldes to funktioner der tilføjer, bevæger og tegner partiklerne:

```
1 ps.addParticle();
2 ps.run();
```

I partikelsystem klassen genereres en arraylist der indeholder, partikler og en PVector med en tilfældig start location ved position  $y = 50$ :

```
1 void addParticle() {
2   origin = new PVector(random(width),50);
3   particles.add(new Particle(origin));
4 }
```

Endnu en metode i partikelsystemklassen sørger for at alle partikler i partikelsystem arrayListen udfører deres bevægelse, og fjerner en partikel hvis den erklæres død:



```
1 void run() {
2     for (int i = particles.size()-1; i >= 0; i--) {
3         Particle p = particles.get(i);
4         p.run();
5         if (p.isDead()) {
6             particles.remove(i);
7         }
8     }
9 }
```

I den sidste klasse, som er partikel klassen er der metoden der opdaterer partiklens location:

```
1 void update() {
2     velocity.add(acceleration);
3     location.add(velocity);
4     lifespan -= 2.0;
5 }
```

Og partiklen tegnes:

```
1 void display() {
2     stroke(0,lifespan);
3     strokeWeight(2);
4     fill(127,lifespan);
5     ellipse(location.x,location.y,12,12);
6 }
```

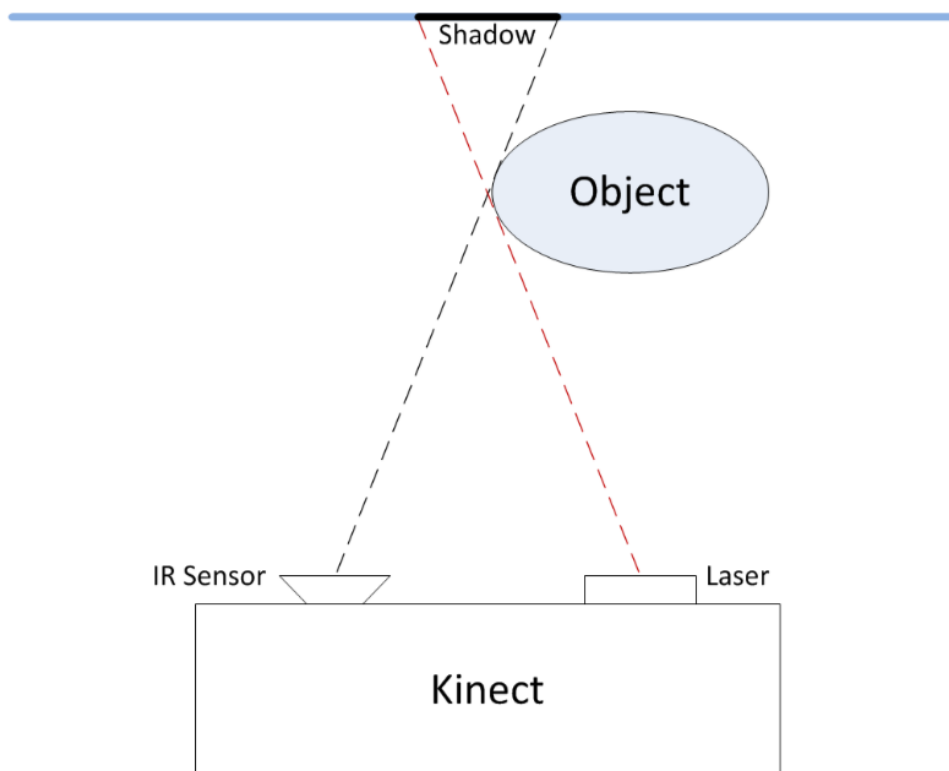
Til sidst i denne klasse checkes om et partikel er død ved at kigge på ”livstidsværdien”:

```
1 boolean isDead() {
2     if (lifespan < 0.0) {
3         return true;
4     } else {
5         return false;
6     }
7 }
```

#### 4.2.4 Kinect dybde

Som det også er beskrevet i Kinect afsnittet, fungerer Kinectens dybde ved hjælp af en infrarød laser og et kamera, som læser afstanden til objekter, ved hjælp af den infrarøde laser (Andersen, 2012). Kinecten kan måle afstanden i dybden fra 0,8 meters afstand til 3,5 meters afstand (Andersen, 2012). Afstanden, som Kinectens dybdekamera kan måle, er upræcis da den kan måle afstande længere væk, end hvad giver gode resultater. Dvs. at nogen kilder sætter dybdekameraets maks. højere end andre, da denne vurdering handler om, hvor præcis en afbildning man ønsker. Kinect v1 har et spektrum på 2048, denne information fortæller os ikke, hvor langt væk man kan stå fra

Kinecten, men det fortæller os noget om, hvor meget Kinectens dybdekamera kan videregive til computeren. Jo længere væk fra Kinecten man bevæger sig, jo dårligere vil visualiseringen blive også selvom computeren stadig modtager information fra Kinecten. Kinectens dybdekamera sender information i form af 11 binære tal. Dvs. at dybdekameraet sender information til computeren i form af 11 binære tal. Samtidig har dybdekameraet en udfordring i forhold til skygger, som illustreret på *figur 4.6*. Da det kamera, som måler afstanden, og den infrarøde laser ikke er bygget samme til en, kommer der skygger bagved objekter, som gør effekterne upræcise.



Figur 4.6: Billedet illustrerer den skygge, som bliver skabt, da den laser som udsender den infrarøde laser sidder forskudt fra det kamera, som måler afstanden til objekter i dybden (Andersen, 2012).

### 4.3 Prototype 1 - Visualisering af bevægelse

Formålet med denne effekt var, at visualisere bevægelse ved hjælp af farver. Prototypen gør brug af komponenten frame differencing. Denne effekt er lavet ved, at gemme billederne fra Kinecten i et array og sammenligne disse med det nyeste billede fra Kinecten. Vi har valgt at visualisere bevægelse ved, at farve de pixels, hvori der er en ændring, med rød og lade dem, hvor der ikke er sket en ændring, være sort. Trail effekten gør bevægelsen mere synlig ved, at gemme et slør af frame differencing billeder, som fader ud jo tættere på trail længden de kommer. Denne sløring er skabt ved brug af en Processing funktionen kaldet "tint".

Først sætter vi `diffImg` til at være lig med `frameDiff`, som består af vores nuværende billede og det tidligere billede. Derefter laver vi et trail og tilføjer `diffImg` til denne. Efter dette har vi en if-statement som siger, at hvis vores længde på vores trail overskrider dens maksimale længde, som vi har sat til at være 10, så skal det billede som er på plads 0 i `arrayList` fjernes. Grunden til at vi har valgt, at der skal huskes 10 billeder i `arrayList` er for at tydeliggøre bevægelses effekten.

```

1 void draw() {
2     diffImg = frameDiff(img, prevPixels);
3     trail.add(diffImg);
4     if(trail.size() > trailLength){
5         trail.remove(0);
6     }

```

I den nedenstående forløkke gøres der brug af en tint funktion, som er tilgængelig i Processing. Vi sætter tintValue til at være 10. Denne tintValue bliver brugt til, at bestemme gennemsnitligheden af billeder i tint() funktionen. Tint() funktionen får i dette tilfælde to informationer. Den første information er en RGB værdi og den anden er en alpha værdi, som bestemmer hvor gennemsnitligt et billede er. For hver gang forløkken køres igennem, trækker vi 1 fra tintValue. Dette betyder, at for hver gang forløkken køres igennem, bliver billedet en smule mere gennemsnitligt. For at få vist billederne i den rigtige rækkefølge, kører man denne forløkke igennem bagfra. Hvis vi havde skrevet denne forløkke med +, ville det første billede med mindst gennemsnitlighed blive vist først og dermed ville det billede med mindst gennemsnitlighed ligge foran de andre og tint funktionen ville miste sin effekt. Til sidst vises det "billede" som den er nået til i indexet. Dermed bliver alle billeder vist, som den løber igennem indexet, og vi kommer derfor til at se frame differencing fra de sidste 10 billeder, for at få en bedre og mere tydelig effekt - altså en form for trail.

```

1     for(int i = trail.size() - 1; i >= 0; i--){
2         tintvalue = 10;
3         tint(255, tintValue);
4         tintValue = tintValue-1;
5         image(trail.get(i), 0, 0);
6     }
7 }

```

Der afsluttes med en if statement, som siger, at hvis forskellen af de tre forskellige farver er højere end vores threshold (som er defineret i starten af programmet som 70), så skal forskellen visualiseres med rød farve. Som ses illustreret på *figur 4.7* i Demonstrations afsnittet.

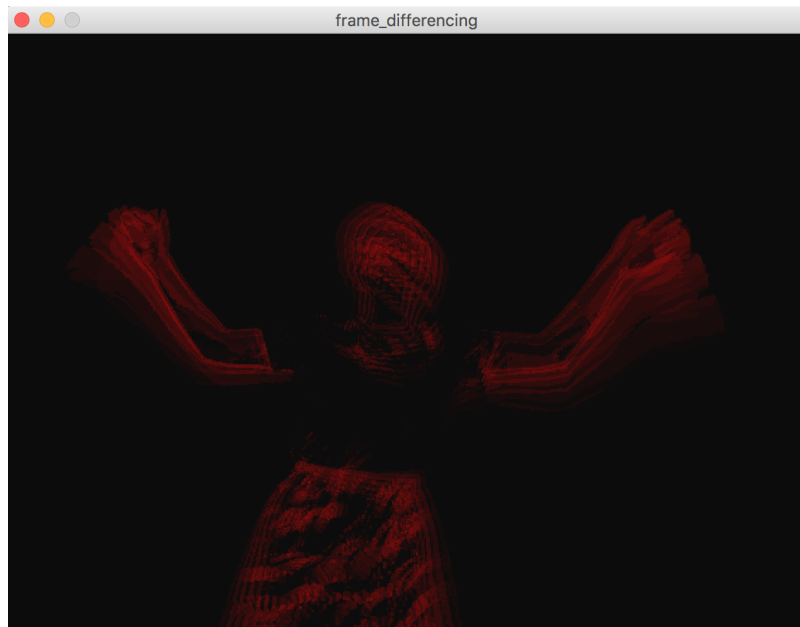
```

if (diffR+diffG+diffB > threshold) {
    display billede.pixels[i] = color(255, 0, 0);
}

```

## Demonstration

I demonstrationen ses et hovedsageligt sort billede med nogle røde pixels, som visualiserer omridset en person. Da prototypen er programmeret til, at skulle vise bevægelse med rød, kan det altså konkluderes, at det kun er skikkelsen som er i bevægelse under denne demonstration, da resten af billedet er sort.



Figur 4.7: Billedet er et skærmebillede af Processing display vindue, når prototype bliver kørt. Billedet viser en sort skærm med røde pixels, som viser omridset af et menneske. Når mennesket er visualiseret med rød, betyder det, at mennesket er i bevægelse

## Delkonklusion

Målet med "visualisering af bevægelse"effekten var, at visualisere de pixels, hvori der er en forskel, med farve. Hvis ikke der er nogle bevægelse, skulle der ikke være nogen visualisering. Det kan altså derfor konkluderes, at effekten har opnået sit overordnede mål. Hvis man skulle arbejde videre med denne effekt kunne man f.eks implementere dybdeeffekter. Da Kinecten har et dybdekamera, kunne man også registrere bevægelse i dybden og ikke kun i x og y retningen.

## 4.4 Prototype 2 - Skrabelod effekt

Skrabelod effekten dannes af Kinectens video, en sort forgrund og bruger samtidig de værdier, som computeren får fra Kinectens dybde kamera. Intentionen med programmet var, at gøre det muligt for brugeren at skrabte Kinectens video billede frem fra den sorte forgrund, ved hjælp af bevægelse. Skrabelod effekten gør brug af en grænseværdi, som sørger for, at der ikke bliver taget højde for uønsket bevægelse i omgivelserne. Dette gøres således, at hvis der opfanges bevægelse indenfor en hvis grænseværdig i dybde kameraet, så vises Kinectens video i stedet for den sorte forgrund - hvis bevægelsen finder sted udenfor denne grænseværdi, så forbliver den sorte forgrund uændret.

Skrabelod effekten benytter sig af Kinectens dybdekamera til, at lave et threshold, som fjerner unødige bevægelse. Dette gøres ved udregningen, som her kaldes for offset. Teorien bag denne udregning er forklaret i Computer Vision afsnittet, sammen med teorien bag loadPixels(). I koden har vi har en dobbelt forløkke, som løber igennem alle pixels i Kinectens bredde og højde. Vi laver en int som vi kalder for  $d$  og sætter her de to værdier depth og offset sammen. Det vil sige, når begge forløkker er kørt igennem og placeringen i arrayets placering i det todimensionelle array, så findes denne specifikke pixels dybdeafstand og denne værdi tildeles til variabelen  $d$ . Herefter laver vi en if statement. Denne if statement skaber en grænseværdi for dybden, som betyder, at det kun er pixels indenfor denne grænse, som bliver visualiseret. Til sidst opdaterer vi alle pixels i vores nye

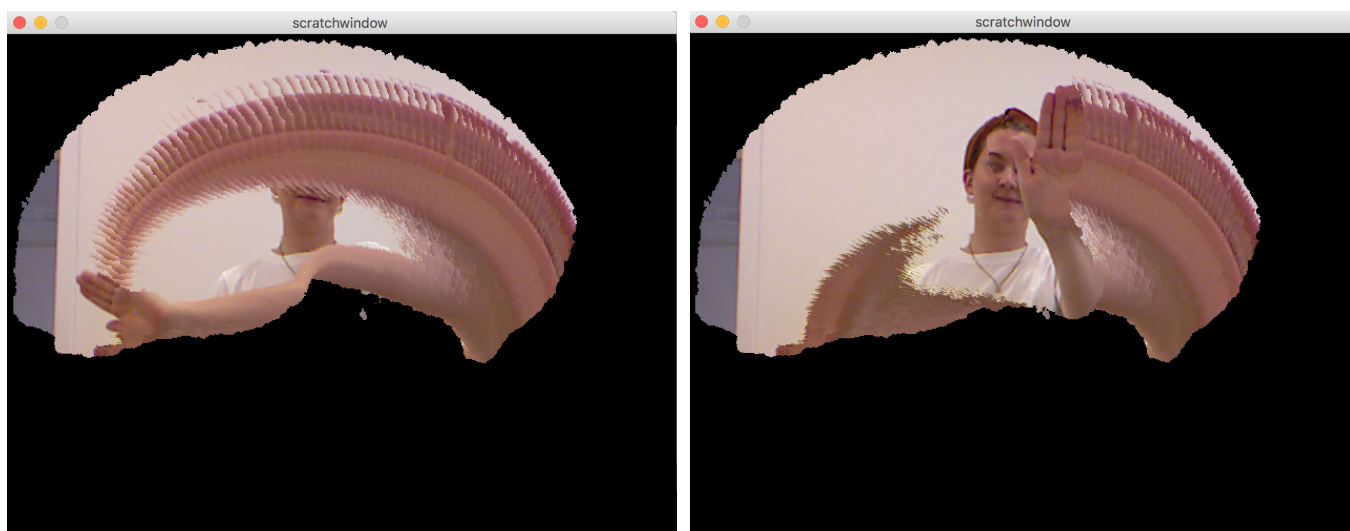
PIImage og viser dette billede, ved brug af funktionen 'image'.

```

1 void draw() {
2   display.loadPixels();
3   depth = kinect.getRawDepth();
4   for (int x = 0; x < kinect.width; x++) {
5     for (int y = 0; y < kinect.height; y++) {
6       int offset = x + y * kinect.width;
7       int d = depth[offset];
8       if (d < 550 && d > 100) {
9         display.pixels[offset] = img.pixels[offset];
10      }
11    }
12  }
13  display.updatePixels();
14  image(display, 0, 0);
15 }

```

## Demonstration



Figur 4.8: På højre billede ses Processing display vinduet fra prototypen scratchcard. På display billedet bliver der "scratchet" fra venstre mod højre. På billedet til venstre fjernes det billede, der er tegnet ovenpå og viser livefeeden bagved. Her ses tydeligt, at baggrunden ikke er et livefeed, som var intentionen med prototypen.

## Delkonklusion

Målet med scratch card effekten er delvist lykkedes. Brugeren kan scratche den sorte forgrund væk, men visualiseringen kunne forbedres. Hvis man forsøger at scratche ved f.eks. at bevæge sin hånd, så fjernes det sorte ikke kun der hvor brugeren bevæger sin hånd, men også et godt stykke til ene side af det bevægende objekt. Vi er kommet frem til, at dette formentlig handler om placeringen af kameraerne i Kinecten. Da denne prototype gør brug af flere forskellige kameraer opfatter den bevægelse fra forskellige perspektiver og dermed fjerner den, den sorte forgrund skævt i forhold til det bevægende objekt. Samtidig er der en fejl i visualiseringen af det der er "scratched", altså der hvor det sorte er fjernet ved bevægelse. Intentionen var, at der hvor det sorte er blevet fjernet skulle man se, hvad Kinectens kamera opfatter i momentet. Som man kan se på figur 4.8 er det i prototypen ikke opnået denne effekt kunne forbedres ved videre arbejde.

## 4.5 Prototype 3 - Vægmaling

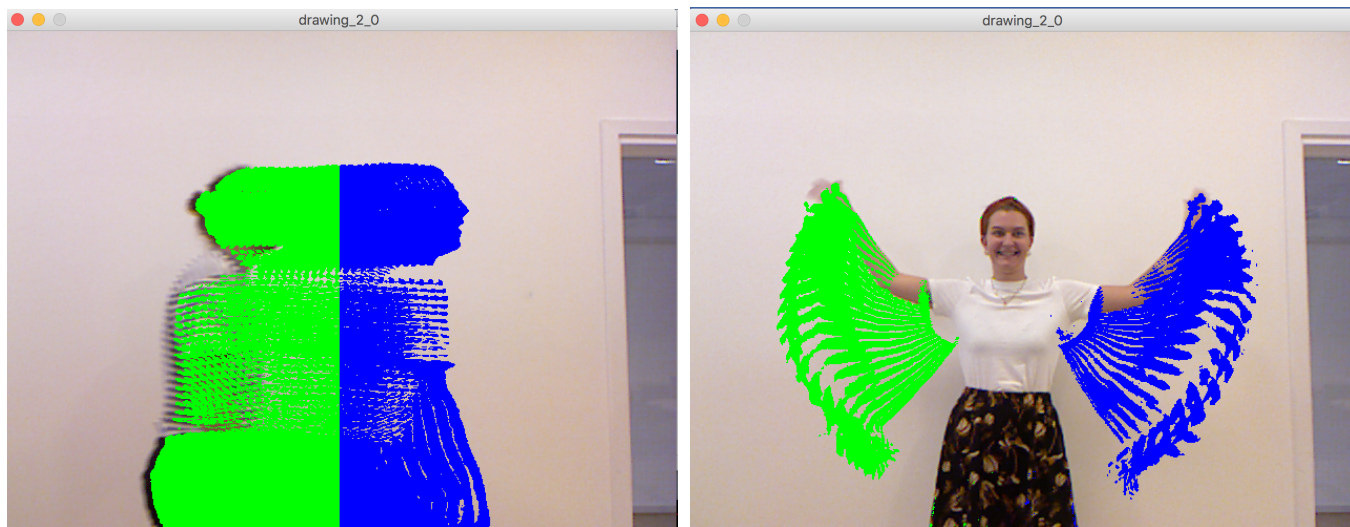
Formålet med denne effekt var at visualisere, at brugeren tegner med sine hænder. Dette har vi forsøgt at gøre ved hjælp af frame differencing. I denne prototype har vi lavet et længere trail end i den forgående effekt (skabe lods effekten), så flere billeder bliver gemt og fremvist. Derudover skifter visualiseringen farve, alt efter om bevægelsen finder sted på højre eller venstre side af skærmen.

Denne prototype omregner et todimensionelt array til, at finde positionen i det endimensionelle array, hvor dataen er gemt (Dette er yderligere forklaret i computer vision afsnittet) for, at kunne bestemme farven afhængigt af kun x-værdierne. Den første forløkke løber igennem bredden af skærmen, altså x-retningen. For hver enkelt x løber den anden forløkke igennem alle pladser i y-retningen i arrayet. Uddybende forklaring af vores dybde grænseværdig kan findes i kodebeskrivelsen af skrabelods effekten. Denne prototype har modsat Visualisering af bevægelse effekten to Pimages til sammenligningen af farver, i stedet for et PImage og et array af farver. Vi bruger stadig vores threshold, men laver en forskel i den viste farve, som er afhængig af x-koordinatet. Dermed hvis der er bevægelse i et område hvor x-værdien er højere end 320, bliver bevægelsen illustreret med blå og ellers bliver bevægelsen illustreret med grøn. Dette er blot for at tilføje en visuel effekt, samtidig med at man får gjort brug af den dobbelte forløkkes egenskaber, i form af at bruge x-koordinaterne en visuel effekt.

```

1      int loc = x + y * i1.width;
2      int d = depth[loc];
3
4      if (diffR + diffG + diffB > threshold && d < 500 && d > 10) {
5          if (x >=320) {
6              outImage.pixels[loc] = color(0, 0, 255);
7          } else {
8              outImage.pixels[loc] = color(0, 255, 0);
9          }
10     }
11     i2.pixels[loc] = currColor;
12 }
13 }
14
15 updatePixels();
16 return outImage;
17 }
```

## Demonstration



Figur 4.9: 1. Billedet viser et Processing display vindue med prototypen drawing. Drawing effekten vises, da den farve som bevægelsen har skabt, stadig er der selvom personen har rykket videre i billedet. Bevægelsen er visualiseret med grøn og med blå alt efter hvorhenne personen befinder sig foran Kinecten 2. Billedet viser et Processing display vindue med prototypen drawing. Billedet er et billede ligesom det foregående billede. Dog kan man på dette billede se, hvordan det kun er det som bevæger sig på billedet, der bliver farvet og ikke det, som er forholdsvis stilstående.

## Delkonklusion

Intentionen med Vægmalning prototypen var, at brugeren skulle kunne tegne med sine hænder. Denne prototype kommer dog aldrig helt til at kunne "tegne", da vi gemmer en begrænset mængde billeder i vores trail. Her kommer programmets effektivitet til kort, som også nævnt i kodebeskrivelsen. Vi kan ikke blive ved med at gemme billeder i vores arrayList, da det bliver for meget for programmet at behandle og det vil dermed crashe. Vi har derfor forsøgt at løse problemet ved at fjerne det ældste billede i arraylisten. Dette ødelægger dog lidt den ønskede tegne effekt, da brugerens "tegning" forsvinder forholdsvis hurtigt. Problemet kunne løses ved, at i stedet for at fylde et array med billeder, så gemme det seneste display billede med den nyeste tegnede farve. Selv hvis denne løsning blev implementeret, ville vi dog stadig have en udfordring med, at effekten ikke er "fintfølende" nok. Brugeren ville altså stadig ikke opnå den ønskede "tegne" effekt, da frame differencing metoden reagerer på alt bevægelse. Retrospektivt ville en form for tracking formentlig have været bedre, hvis man skulle opnå den tegne effekt i stedet for 'maling'. Dette lykkes dog ikke og man kan dermed sige, at intentionen med denne prototype ikke er helt nået.

## 4.6 Prototype 4 - Partikel interaktion

Målet med denne effekt er at brugeren kan påvirke bevægelsen af partikler fra et partikelsystem. Partiklerne skal falde fra toppen af et canvas og hoppe op hvis der er et objekt eller en person foran Kinect kameraet inden for et bestemt område. Dette foregår ved at bruge et partikelsystem og dybdekameraet fra et Kinect Kamera.

I main klassen, som i vores program er navngivet ParticleBounce tages data fra kinecten og

checkes for objekter inden for en bestemt grænseværdi.

```

1 for (int i=0; i < rawDepth.length; i++) {
2     if (rawDepth[i] >= minDepth && rawDepth[i] <= maxDepth) {
3         depthImg.pixels[i] = color(0, 255, 0);
4     } else {
5         depthImg.pixels[i] = color(0);
6     }
7 }

```

Metoden ovenfor checker arrayListen med information om hver pixel for at se om pixelen ligger indenfor et bestemt interval. Hvis pixelen gør dette, farves den grøn ellers farves den sort. Det endelige billede der opstår vises så på skærmen. Dernæst kaldes partikelsystem klassen, denne klasse er ens til det partikelsystem som er forklaret i afsnit 4.2.3.

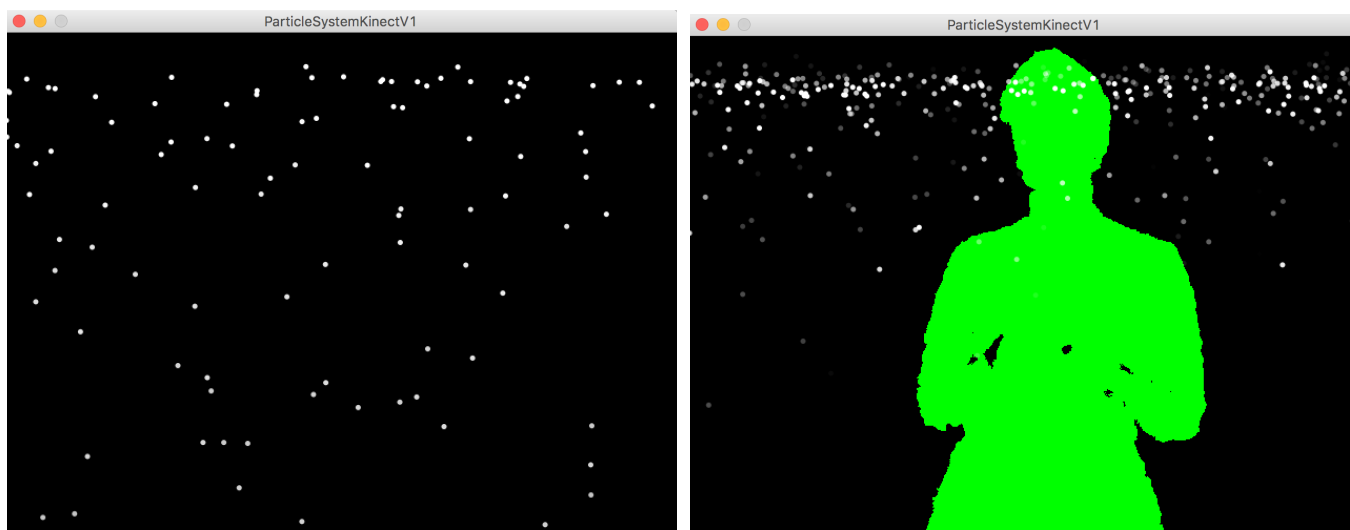
Den sidste klasse knyttet til denne prototype er partikel klassen. Denne klasse er også ens til det partikelsystem der er forklaret . Der er dog en lille forskel, i den metode der bestemmer hvordan en partikel skal bevæge sig, er der indsat en if-lykke der tjekker for grønne pixels på skærmen. Hvis en pixel er grøn skal partikelen bevæge sig i den modsatte retning af dens tidligere bevægelse, hvis ikke skal partikelen blive ved med at bevæge sig fremad:

```

1 if (depthImg.pixels[index] == color(0, 255, 0)) {
2     velocity.add(acceleration);
3     velocity.mult(-1);
4     location.add(velocity);
5 } else {
6     velocity.add(acceleration);
7     location.add(velocity);
8     lifespan -= 1.0;
9 }

```

## Demonstration



Figur 4.10: Billedet til venstre viser partikler der falder fra oven uden et objekt eller et menneske. Billedet til højre viser partikler der falder fra oven, men hopper op igen da der er en person foran Kinecten.

## Delkonklusion

Formålet med effekten er delvist opfyldt. Partiklerne falder fra toppen i en konstant strøm og uafbrudt når der ikke er noget foran Kinecten. Det billede der vises når programmet kører, viser



også som ønsket, kun de pixels hvor et objekt eller en person er foran kameraet. Når kameraet opfanger et objekt inden for grænseværdien, repræsenteres de som grønne pixel og det er synligt at partiklerne begynder at hoppe væk fra de grønne pixels. Problemet er at de partikler der falder ude i det sorte område, også hopper når der er grønne pixels andetsteds på skærmen. Grunden til fejlen er ukendt, men de kan være at det er forårsaget af støj fra kameraet der ikke bliver frafiltreret. Dermed kan det konkluderes at programmet ikke virker som det skal.

## 4.7 Prototype 5 - Farvesporing

Denne prototype har til formål at illustrere en simpel sporingsmetode illustreret med partikler, der ligner en digital flamme/røgsky. Partikelklassen til denne prototype, fik vi tilsendt af vores vejleder John Gallagher (se bilag 3.1). Denne partikelklasse er efterfølgende blevet modificeret i hovedklassen og implementeret som den effekt, der vises hvor en blob er. Programmet er lavet ved hjælp af det indbyggede webcam, som findes i de fleste bærbare computere. Der findes flere biblioteker, som kan være behjælpelige, når et objekt skal spores, så som openCV til Processing og Kinect v2 for Processing (til Windows). Her findes eksempler på avanceret sporing, som sporer et eller flere mennesker med så stor nøjagtighed, at programmet ved, hvor de forskellige kropsdele befinder sig. Da vi bestræber os på at blive klogere på Java-sproget i sig selv, har vi besluttet os for ikke at benytte os af disse eksempler, men derimod at kode os frem til en simpel blob tracking. Dette styrker vores forståelse af, hvordan sporingsmetoder fungerer, da vi forstår hovedprincipperne i denne slags systemer.

Farvesporing-programmet indeholder en kombination af partikelsystem og blob tracking. "Wind" fra partikelsystemet er ikke en del af det interaktive udtryk endnu, da vi ikke har nået at implementere denne idé i programmet uden at bruge tastaturknapper. Dog har vi taget det med, da det er en ting, der kunne arbejdes videre på. `Wind()` putter en retning på røg-partiklerne, så strålen af partikler hælder den ene eller den anden vej.

Med et klik på musen kan farven, der spores ændres så den bliver lig den farveværdi der, er på den pågældende pixel. Dette gøres ved at finde placeringen af pixels i `arrayList` den er gemt i, og derefter sætte `trackColor`-variablen til at være den fundne RGB-værdi. `constrain()` gør at en variabel holder sig inde for en min og max værdi. Syntaksen er først at give værdien der skal begrænses og så give parametrene for begrænsningen.

```

1 void mousePressed() {
2   int loc = constrain(mouseX + mouseY*video.width, 0, video.pixels.length-1);
3   trackColor = video.pixels[loc];
4 }

```

`DistThreshold` og `threshold` kan ændres ved en tast på tastaturet. Dette er vigtigt i test-fasen for at finde den optimale variabel for grænseværdierne. Ved et tryk på 'a' forhøjes `distThreshold` med 1 da `distThreshold++`; er det samme som `distThreshold = distThreshold+1`;. Det er også muligt at alterere vinden, hvis dette ønskes.

Partikelsystemet kaldes for hver tilgængelig blob i det første forloop, og i det næste forloop er det muligt at tilføje flere partikelsystemer for en mere kraftig effekt. Partikelsystemet tilføjes ved blobbens midte, da  $(\text{minx} + \text{maxx})/2, (\text{miny} + \text{maxy})/2$  er midten af blob objektet. efterfølgende looper vi igennem, alle partiklerne som skal vises fra de to tidligere for-loops. Først tildeler vi

dataen fra partikelsystemet til partiklen p. Herefter kaldes metoderne fra partikel-klassen, som kreerer og viser effekten. Hvis fade-variablen for en partikel er falmet til under 0, sletter vi den fra partikelsystem-arrayet. Til sidst kalder vi blobs.clear(), da vi kun ønsker midlertidige blobs.

```

1 void draw() {
2   for (int i=0; i<blobs.size(); i++) {
3     for (int j=0; j<2; j++) {
4       ps.add(new Particle(img, (blobs.get(i).minx+blobs.get(i).maxx)/2,
5         (blobs.get(i).maxy+blobs.get(i).miny)/2));
6     }
7   }
8   for (int i=0; i< ps.size(); i++) {
9     p = ps.get(i);
10    p.move();
11    p.update();
12    p.applyWind(wind);
13    p.showImg();
14    if (p.fade < 0) ps.remove(p);
15  }
16
17  blobs.clear();

```

Vi ønsker imidlertid kun at tracke to blobs, da grænseværdierne kan være svære at justere til de omgivelser, hvor webcammet er i og fordi røg-effekten kan få programmet til at køre langsomt.

```

1 if (!found && blobs.size() <=1) {
2   Blob b = new Blob(x, y);
3   blobs.add(b);
4 }

```

I starten af denne prototypes partikelklasse, som er i en separat tab, erklærer vi fire forhold. Et partikelsystem skal have et start koordinatsæt (x,y), partiklerne skal kunne bevæge sig (vx, vy), som står for velocity x og velocity y. Partikeleffekten skal kunne falme (fade), og til slut gør vi plads til et PImage, som hedder smokeImg, hvilket er hvor selve røg-effekten kommer fra.

```

1 class Particle {
2   float x, y;
3   float vx, vy;
4   int fade;
5   PImage smokeImg;

```

Nu til Particle()-constructor'en for denne prototype. Den indeholder et PImage og x- og y-koordinater. Den indeholder x- og y-koordinator, så vi kan sætte effekten til at være i midten af en funden blob. Før PImage kaldes giver vi partiklerne en tilfældig vertikal hastighed (vy), som er mellem 1,4 pixels per frame, hvilket svarer til 1\*30 til 4\*30, da vi kører prototypeprogrammet med 30 frames per second. Værdierne er negative da koordinatsystemets 0,0-punkt findes oppe i højre hjørne i Processing. Vi giver også partiklerne en tilfældig horisontal hastighed mellem -1 til 1, som er en bevægelse fra højre til venstre side (vx). Da vx og vy variablerne er floats kan de indeholde decimaltal. Som det sidste før vi kalder billedet, tildeler vi det en fade, som senere skal få effekten til at falme. vx og vy variablerne får ikke billedet til at bevæge sig endnu. Dette håndteres i void move(), som betyder at vx og vy skal stige med sig selv for hver frame. void update() får effekten til at falme med -2 per frame og void applyWind() tildeler vx-variablen en ekstra hældning.

```

1 Particle(PImage img, float x1, float y1) {
2   x = x1;
3   y = y1;
4
5   vx = random(-1, 1);
6   vy = random(-4, -1);
7   fade = 255;
8   smokeImg = img;
9 }
10
11 void move() {
12   x += vx;
13   y += vy;
14 }
15
16 void update() {
17   fade -= 2;
18 }
19
20 void applyWind(float w) {
21   vx += w;
22 }

```

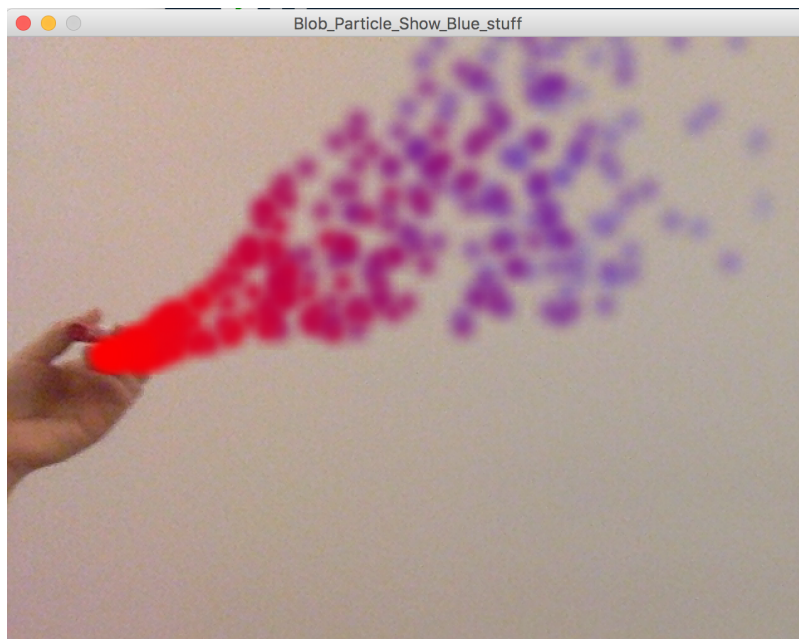
I void show()-funktionen opretter vi en cirkel, som falmer ved hjælp af den indbyggede transparensfunktion i Processing. Cirklen har en radius på 5 pixels og starter på x- og y-koordinatet. I showImg()-funktionen opretter vi selve effekten. Ved hjælp af tint()-funktionen i Processing er det muligt for os at få partiklerne til langsomt at falme. tint()-funktionerne forventer følgende variabler (v1,v2,v3,alpha), hvor v1, v2, v3 er R,G,B eller nuance, mætning, lysstyrke alt efter hvilket color-mode du er i. I prototypeprogrammet er den den farven i showing() rød i starten og mere blålig når effekten falmer. Dette er gjort ved at sætte rød lig med 255 og blå 255-fade. Når draw setuppet har kørt i gennem 0 gange er den blå farve  $255-255=0$ , og når programmet er kørt i gennem 60 gange er den blå farve  $255-(255-60*2)=135$ .

```

1   void show() {
2     noStroke();
3     fill(200, fade);
4     ellipse(x, y, 5, 5);
5   }
6
7   void showImg() {
8     tint(fade, 0, 255-fade, fade);
9     image(smokeImg, x, y);
10    tint(255, 255);
11  }
12 }

```

## Demonstration



Figur 4.11: I billede ses et eksempel, hvor vi sporer en tusch. Der er tilføjet vindeffekt, for at illustrere hvordan der kunne arbejdes videre på programmet. Eksempelvis kunne vindeffekten aktiveres når der er bevægelse på  $x$  akse.

## Delkonklusion

Koden illustrerer, hvordan en simpel sporingsmetode kan illustreres med en visuel effekt. I starten var idéen, at programmet skulle være i stand til at finde et menneske og afkode forskellige kropsdele, så som et hoved og nogle hænder. Det ville have givet større frihed i forhold til den visuelle effekt, der virker bedst, hvis man har en meget skarp farve, såsom en rød kop eller lignende. Ydermere skal man trykke på musen for at spore en ny farve. For en bedre interaktiv installation, ville det være mere optimalt, hvis denne proces kunne automatiseres. Som koden er lige nu, er det altafgørende hvilke omstændigheder, man afprøver koden i. Farven på en pixel er nemlig meget afhængig af, hvad dit udstyr kan optage af data. Hvis der kommer meget lys ind af et vindue, kan pixelene ændre farve på grund af modlys og dannelse af skygger med mere.

## 4.8 Interface

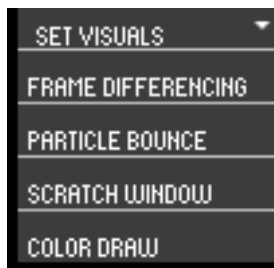
Da vores projekt består af forskellige prototyper der udforsker mulighederne for billedmanipulation og tracking, i et processing miljø, har vi fundet det nødvendigt at integrere de separate prototyper i et samlet program. Dette skyldes til dels, at det vil gøre det lettere for brugeren, at manøvrere rundt mellem programmerne, uden at skulle åbne hvert program et efter et, og dels for at drage fordel ved Objekt-orienteret Programmering (OOP) og gøre det lettere at udvide programmet med flere prototyper/effekter.

### ControlP5

Vores brugergrænseflade består af ét simpelt GUI-element, for ikke forstyrre det visuelle fra prototyperne. En dropdown-menu i toppens højre hjørne af processing-vinduet, gør det muligt at skifte

mellem effekterne. Dropdown-menuen er en del af ControlP5 biblioteket, som indeholder en lang række GUI-elementer så som sliders, buttons, knobs, textfields m.m. (<http://www.sojamo.de/libraries/controlP5/>).

For at inkorporere dropdown-menuen i processing har vi kaldt et ControlP5 og DropDownList kaldet *droplist* objekt. Derefter tilføjer vi en DropDownList til droplist objektet, og sætter dens position i processing vinduet.



Figur 4.12: Dropdown Menuens udseende i processingvinduet

```

1  import controlP5.*;
2
3  ControlP5 cp5;
4  DropDownList droplist;
5
6  void setup() {
7      size(640, 480);
8      cp5 = new ControlP5(this);
9      droplist = cp5.addDropDownList("Set Visuals").setPosition(542, 0);
10
11 }

```

Dropdown menuen består nu kun af et titel-element, hvor der står "Set Visuals". For at tilføje flere muligheder til menuen laver vi først et *array* af *strings* som vi kalder "programs". Størrelsen af det *array* afhænger af antal programmer vi har valgt at tilføje. Hver *index* får dertil en *string* som er navnet på den prototype der skal være på den tilsvarende plads i dropdown-menuen. Til sidst går vi igennem et *loop* for at tilføje "programs" *arrayet* til dropdown menuen.

```

1  String[] programs = new String[5];
2  programs[0] = "Frame Differencing";
3  programs[1] = "Particle Bounce";
4  programs[2] = "Scratch Window";
5  programs[3] = "Color Draw";
6  programs[4] = "Blob Tracking";
7
8  for (int i=0; i<programs.length; i++) {
9      droplist.addItem(programs[i], i);
10 }
11 }

```

## PGraphics

For at drage fordel af OOP har vi brugt PGraphics biblioteket, til at skabe et "offscreen" canvas til hver enkelte prototype. For at tegne på dette offscreen canvas skal man indkapsle den del af koden der skal tegnes i *beginDraw()*; og *endDraw()*; . Dette giver også muligheden for, at ligge canvas som lag oven på et eksisterende vindue, når processingprogrammet kører.

Her er et eksempel fra frame differencing prototypen hvor dets *trail* skal tegnes på frame differencings "offscreen" canvas. Derfor er det indkapslet i *beginDraw()*; og *enddraw()*; metoderne.

```

1 pg.beginDraw();
2   for (int i = frameTrail.size() - 1; i >= 0; i--) {
3     tintValue = 10;
4     pg.tint(255, tintValue);
5     tintValue = tintValue-1;
6     pg.image(frameTrail.get(i), 0, 0);
7   }
8   pg.endDraw();

```

For at vise de forskellige canvas fra prototyperne i processingvinduet har vi skulle lave et nyt *PGraphics* canvas. Det canvas får tildelt den grafik der kommer fra vores forskellige prototyper.

```

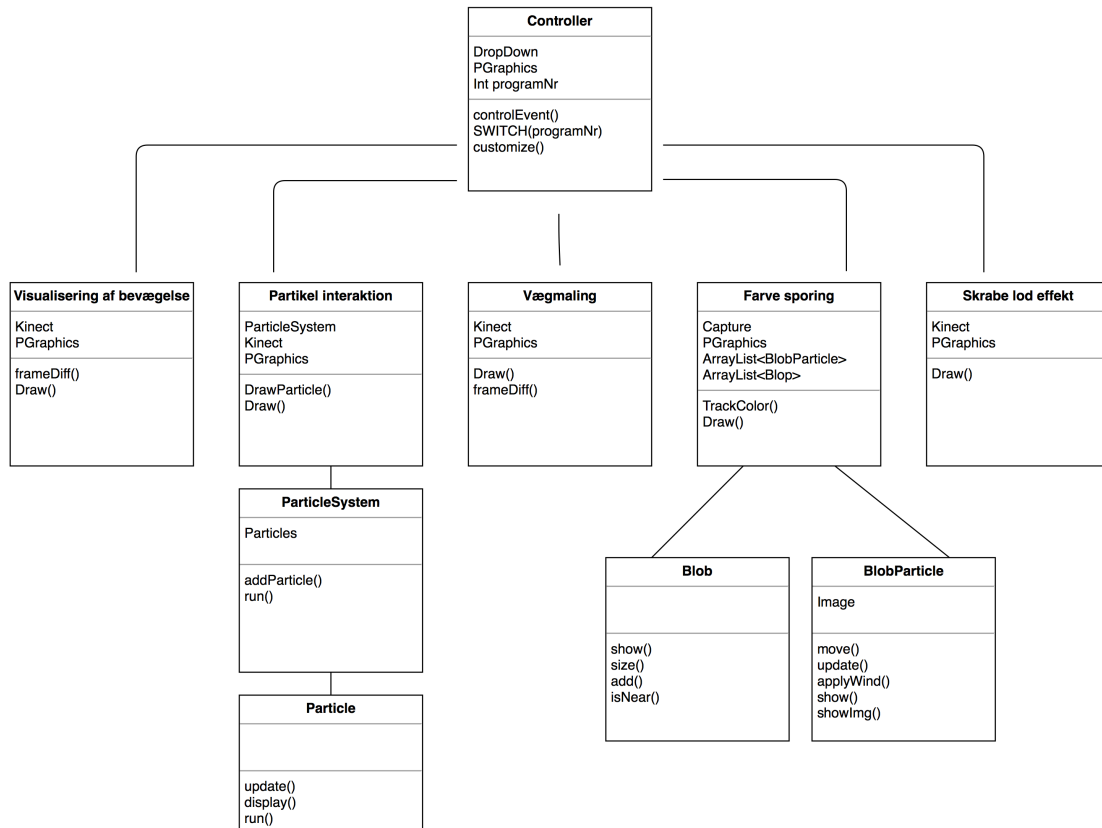
1 FrameDiff fd;
2 PGraphics g;
3
4 void setup() {
5   g = createGraphics(width, height);
6 }
7 void draw() {
8   fd.draw(g);
9 }

```

Her ses det at der bliver kaldt en draw metode fra frame differencing prototype objektet. Det betyder hver prototype har sin egen klasse med sin egen draw() metode, der bliver tegnet på til tilhørende *PGraphics* canvas.

### Programstruktur

Ved at bruge *PGraphics* har vi gjort det muligt, at isolere prototypernes variabler og udelukkende have de variabler der er globale i controlleren.



Figur 4.13: En illustration af programstrukturen, vist med klasser og deres indbyrdes forhold til hinanden.

For at skifte mellem de forskellige prototyper har vi gjort brug af et *SWITCH*-statement, med *cases* til hvert af prototyperne, som sø starter tilsvarende prototypes *draw()*; Vi har været nødsaget til at have en *noTint()*; i hvert *case*, grundet at *tint* er en global variable, som bliver overført til alle dele af programmet hvis ikke at man fjerner der, eller giver det en ny værdi.

```

1 void draw() {
2     switch(programNr) {
3         case 0:
4             g.noTint();
5             fd.draw(g);
6             break;
7         case 1:
8             g.noTint();
9             pb.draw(g);
10            break;
11        case 2:
12            g.noTint();
13            sw.draw(g);
14            break;
15        case 3:
16            g.noTint();
17            cd.draw(g);
18            break;
19        case 4:
20            g.noTint();
21            bt.draw(g);
22            break;
23        default:
24            break;
25    }
26    image(g, 0, 0);
27 }

```

For at skifte mellem programmerne bruger vi et *controlEvent* som ændre værdien af *programNr*, og køre igennem den tilsvarende case i *SWITCH*-statement.

```

1 void controlEvent(ControlEvent theEvent) {
2     if (theEvent.isGroup()) {
3         if (theEvent.getGroup().getName() == "Set Visuals") {
4             }
5     } else if (theEvent.isController()) {
6         if (theEvent.getController().getValue() == 0.0) {
7             background(0);
8             programNr = 0;
9         }
10        if (theEvent.getController().getValue() == 1.0) {
11            background(0);
12            programNr = 1;
13        }
14        if (theEvent.getController().getValue() == 2.0) {
15            background(0);
16            programNr = 2;
17        }
18        if (theEvent.getController().getValue() == 3.0) {
19            background(0);
20            programNr = 3;
21        }
22        if (theEvent.getController().getValue() == 4.0) {
23            background(0);
24            programNr = 4;
25        }
26    }
27 }

```

```
26     }  
27 }
```

## Program udvidelse

Ved at bruge denne struktur skal der foretages meget få ændringer form at udvide programmet yderligere med flere prototyper. Nedenstående ses en simpel guideline for, hvad der er nødvendigt at gøre for, at tilføje flere prototyper til programmet:

- Opret  $X$  ny/nye klasse/klasser.
- Opret tilhørende objekter i kontrolleren.
- Udvid *programs-array* med  $X$ .
- Tilføj programnavn til *programs[X]*
- Udvid *SWITCH*-statement med ny case  $X$ .
- Udvid *controlEvent* metoden med endnu et *if*-statement og ændre *getValue()==X*, og *programNr = X*

## Delkonklusion

Bruget af PGraphics i sammenspil med en ControlP5 DropDownList har gjort programmet lettere at forstå, udvide og integrere med. Selvom programmet allerede er simpelt, er det dog stadig muligt, at simplificere koden endnu mere.



## 5.0 Brugervejledning

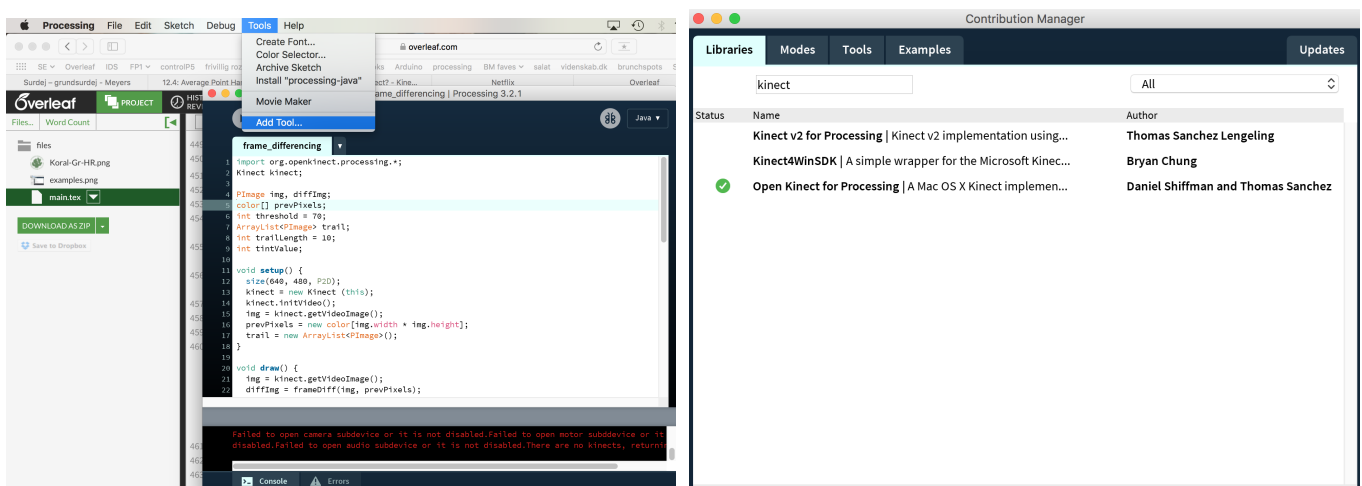
Dette interaktive program består af en række mindre prototyper. Prototyperne, som hver især indeholder en interaktiv effekt, er samlet i et interface, hvor brugeren ved hjælp af musen kan vælge imellem de forskellige prototyper, ved hjælp af en drop down menu. Prototyperne forsøger på forskellig vis, at få brugeren til at interagere med installationen. Målet med programmet er, at installationen skal fange brugerens opmærksomhed og give dem lyst til, at eksperimentere med de forskellige effekter.

### 5.1 Når programmet skal installeres

Nedenstående ses de forudsætninger som skal være opfyldt for, at programmet kan åbnes og køres:

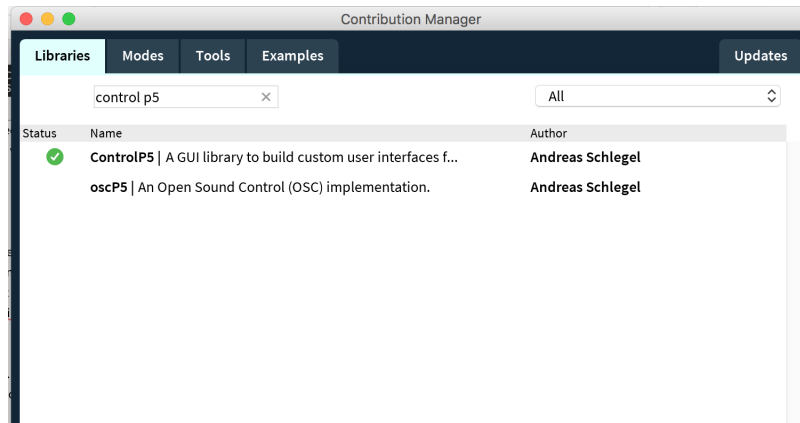
- Det er en forudsætning, at programmet køres på en Mac computer og ikke Windows.
- Det er en forudsætning, at brugeren har importeret de korrekte biblioteker i Processing
- Det er en forudsætning, at den benyttede computer har en forbundet Kinect v1 1414.
- Prototyperne er lavet med Processing version 3.2.1 og Processing version 3.3.7. Hvorvidt prototyperne vil fungere med andre versioner, garanteres ikke.

For at programmet skal kunne fungere er det nødvendigt, at ovenstående forudsætninger er opfyldt. Processing skal downloades og installeres fra internettet og brugeren skal have en Kinect v1 1414 til rådighed. Når Processing er installeret og programmet er åbnet, skal der installeres et redskab fra det bibliotek, som er til rådighed i programmet. Dette gøres ved at vælge "tools" oppe i menulinjen, klikke på "add tools" og "libraries" (se venstre billede af figur 5.1). I søgefeltet indtastes "Kinect" og ud fra de muligheder som kommer op, vælges "Open Kinect for Processing". Til sidst installeres bibliotekerne (se højre billede af figur 5.1).



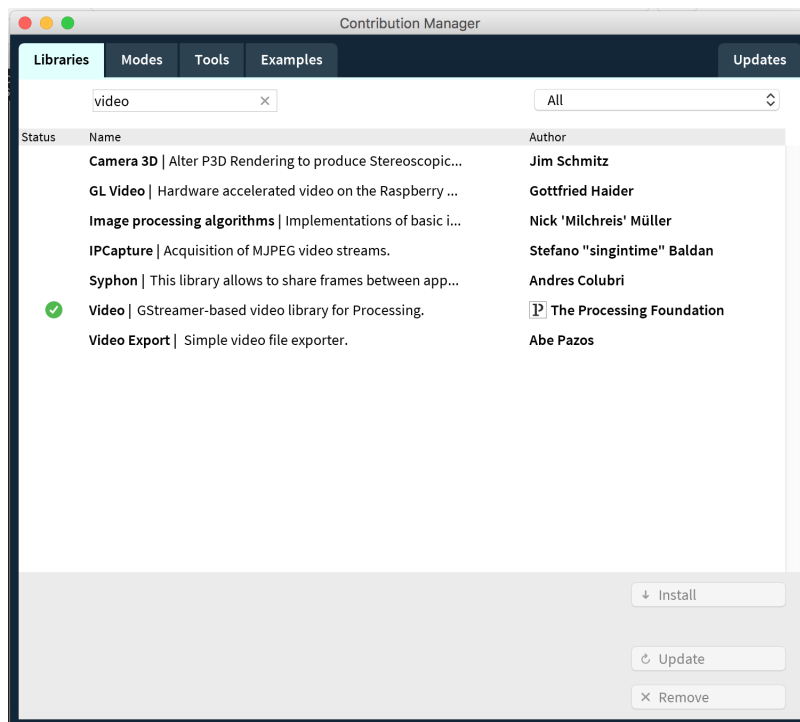
Figur 5.1: To skærbilleder som illustrerer, hvordan man henter og installerer et bibliotek i Processing. På det venstre billede ses det, hvordan man får biblioteks vinduet frem, oppe i menu linjen. På det højre billede ses det, hvordan man finder og installere de nødvendige biblioteker.

Derudover er det nødvendigt at downloade biblioteket "Control P5" for, at interfacet virker. Dette bibliotek installeres på samme måde som det bibliotek, som gør Kinecten brugbar - brugeren søger på "ControlP5" i søgefeltet og vælger det bibliotek med samme navn (se figur 5.2).



Figur 5.2: Skærbilledet viser, hvordan man installerer biblioteket "ControlP5"

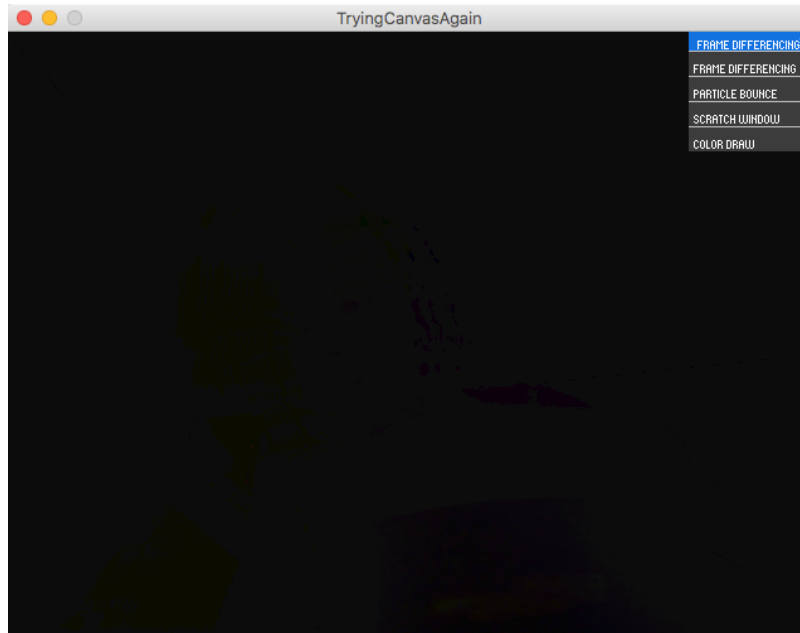
Ligeså skal der installeres et bibliotek før at prototypen, som benytter blob tracking, virker. Dette bibliotek hedder "Video" og hentes på samme måde som de forgående biblioteker (se figur 5.3).



Figur 5.3: Skærbilledet viser, hvordan man installerer biblioteket "Video"

## 5.2 Når programmet køres

Når Processing er hentet og installeret, Kinecten er sat til og de benyttede biblioteker er hentet, kan programmet køres. Når brugeren åbner programmet, kan drop down menuen fremvises ved, at vedkommende holder musen oppe i højre hjørne. Når denne er fremme kan brugeren frit skifte imellem de forskellige effekter, hvorefter disse vil vises på skærmen (se figur 5.4).

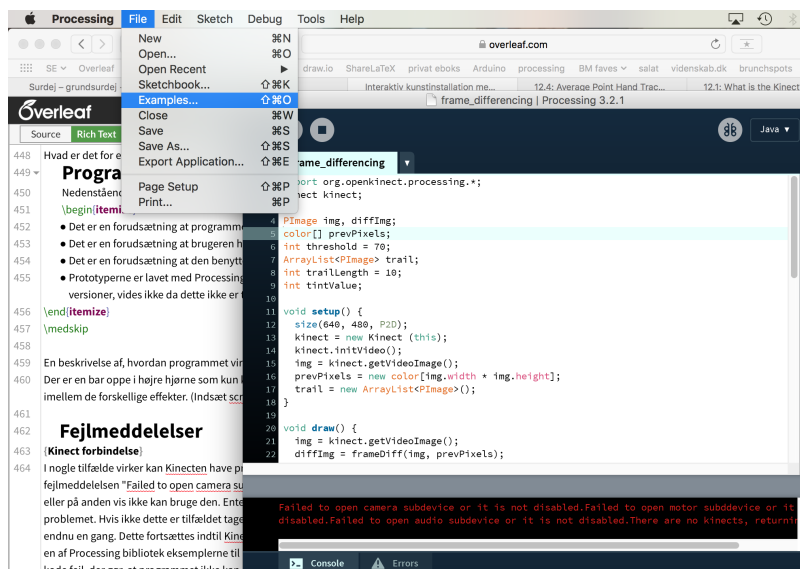


Figur 5.4: Skærbilledet viser, hvordan interfacet ser ud, med en drop down menu oppe i højre hjørne.

## 5.3 Eventuelle fejl

### Kinect forbindelse

I nogle tilfælde kan der opstå problemer med, at få kinecten til at forbinde korrekt til computeren. Hvis brugeren oplever at få fejlmeddelelsen "Failed to open camera subdevice" er det fordi Processing ikke kan finde den tilsluttede Kinect. Enten kan brugeren forsøge at benytte en anden USB port og se, om dette løser problemet. Hvis ikke dette er tilfældet, tages Kinecten ud og sættes i USB porten igen, hvorefter programmet startes på ny. Dette skal muligvis gentages et par gange, før forbindelsen oprettes. Her kan det være en god ide, at benytte et af Processings bibliotek eksempler til at teste, da man er sikker på disse koder virker og det dermed ikke er en koden som er skyld i fejlen. For at få adgang til et bibliotek vælges "files" og "examples", som vist herunder på figur 5.5.



Figur 5.5: Skærbilledet viser, hvordan man åbner et kode eksempel i Processing.

# 6.0 Afprøvning

## 6.1 Unittest

Unittesting er en metode som bliver brugt til at teste software. Formålet med unittesting er at isolere hver metode i et program og vise, at de individuelle metoder virker efter hensigten. Unittesten er en kontrakt som en kode enhed skal opfylde. En test er fyldestgørende, når alle tilstande i en funktion eller en metode er opfyldt. Dette kan f.eks. være ved brug af en boolean, hvor der både skal testes for, hvis udfaldet er true eller false, eller i en if/else statement, hvor det skal undersøges, at begge udfald virker efter hensigten. I vores tilfælde er det visuelle effekter, vi tester og det gøres derfor ikke på helt samme måde. Vi ved, hvilket udfald vi forventer af f.eks frame differencing og vi kan derfor teste denne éne metode, separat for resten af effekten den indgår i. En fordel ved at benytte unittesting er, at fejl kan opdages tidligt i udviklingsprocessen. Vi har valgt at udføre en unittest på vores frame differencing metode. Dette har vi netop gjort for at undersøge, om den pågældende kodedel virker. Da Processing ikke har nogle testfunktioner eller hjælpemidler inkorporeret i programmet som eksempelvis IntelliJ har, har vi selv måttet skrive et test program, som tester den pågældende funktion. Hvis vi havde skrevet vores program i eksempelvis IntelliJ, havde vi kunne benytte os af den indbyggede unittest klasse og dermed også se, hvor stor en procentdel af koden, som reelt set bliver brugt, når programmet kører. Da Processing ikke tilbyder denne slags hjælpemidler og det er en omfattende opgave at teste hele sit program (da dette i princippet fylder lige så meget som den originale kode), har vi valgt kun at teste frame differencing- og blob tracking-komponenterne. Dette giver selvfølgelig ikke et retmæssigt billede af programmets overordnede funktionsdygtighed, men det viser, at vi ved, hvad unittesting er, og hvorfor dette er vigtigt og ikke mindst nyttigt i udviklingsprocessen.

### 6.1.1 Kinect dybde kamera test

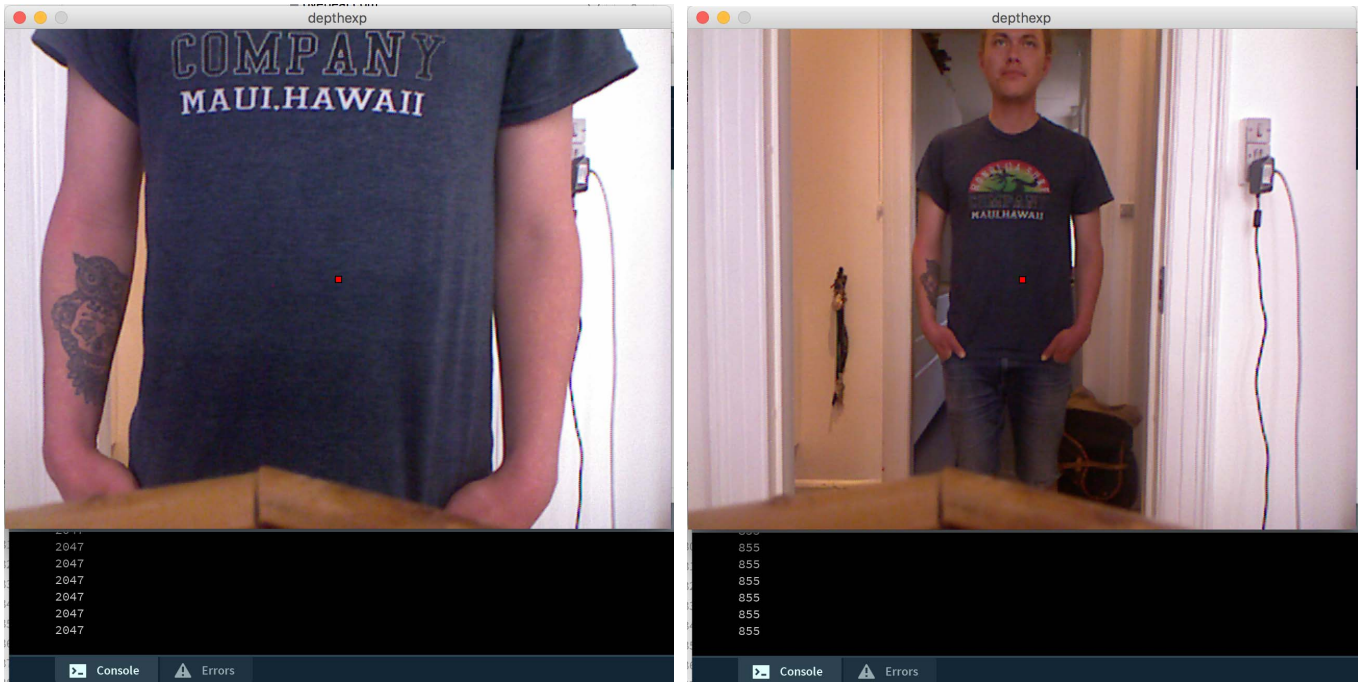
Test udført d. 26.05.2018, kl 23.14.

Vi har valgt at teste Kinectens dybdekamera, for bedre at få en forståelse for, hvilke information computeren modtager fra Kinectens dybdekamera og hvor langt man kan bevæge sig fra kameraet og stadig få "gode" resultater. Til testen har vi skrevet et kort stykke kode, som printer den dybdeværdi, som computeren modtager fra Kinecten i et bestemt punkt. Vi bruger i koden `kinect.getRawDepth()` for at få dybdeværdierne fra Kinecten. Vi har en dobbelt forløkke, hvori vi har en if statement. Her udvælges et bestemt punkt (i midten af skærmen), ved hjælp af et fikseret x og y koordinat. Dette punkt er udgangspunktet for vores testmåling. Dette punkt markeres med en rød firkant. Den røde firkant er en visuel effekt, så vi har mulighed for nøjagtigt at teste målingerne. Der printes dybdeværdien for dette punkt. For at få dybdeværdien benyttes udregning forklaret i afsnittet Computer Vision, og herefter sættes dybden fra Kinecten til den udregnede værdi. Dermed får vi værdien for præcis det pixel i det todimensionelle array.

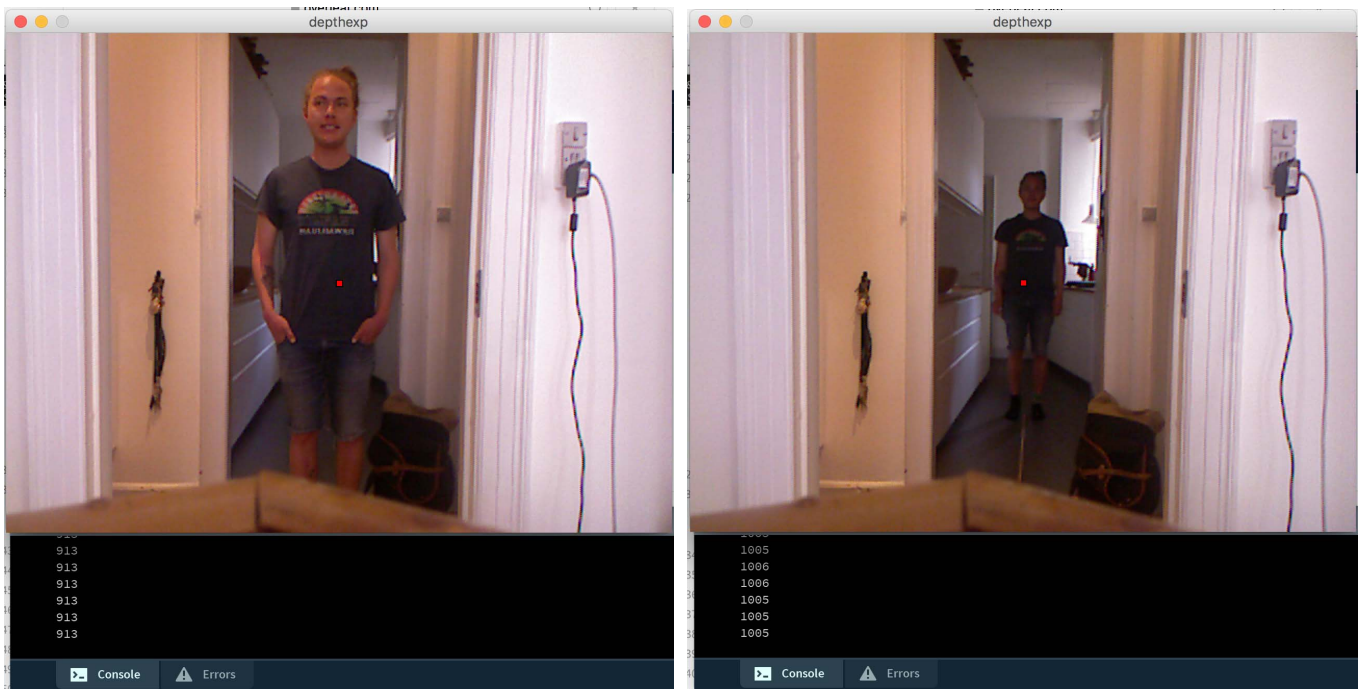
```
void draw() {
    image(img, 0, 0);
    for (int x = 0; x < kinect.width; x++) {
        for (int y = 0; y < kinect.height; y++) {
```

```
int offset = x + y * kinect.width;
int d = depth[offset];
if (x == 320 && y == 240){
    fill(255,0,0);
    rect(x-3,y-3,6,6);
    println(d);
}
}
}
}
```

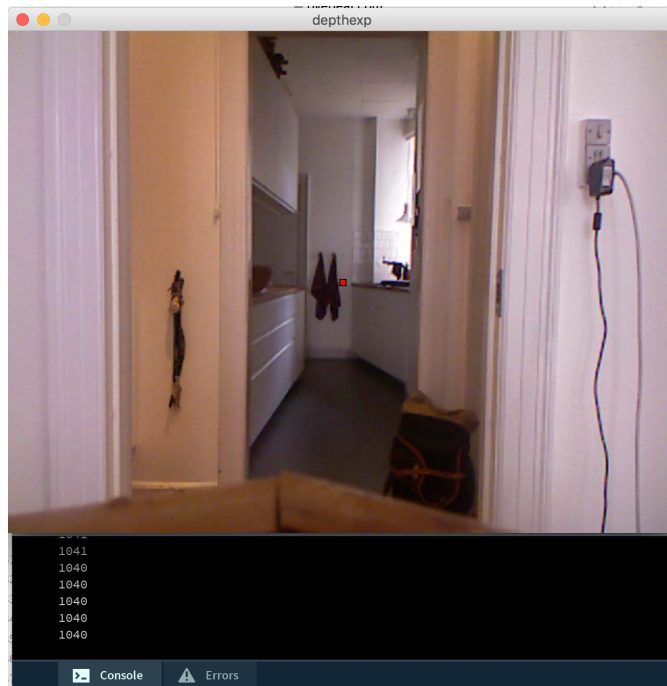
På billederne herunder ses en person, som står henholdsvis 50cm, 1,5 meter, 2 meter, 4 meter og 6,5 meter væk fra Kinectens kamera. Venstre billede på *figur 6.1* på dette billede er værdien 2047, som er maksimumsværdien for Kinectens dybdekamera. Da personen står så tæt på, kan vi her konkludere, at personen står for tæt på, til at Kinecten kan måle dybden til objektet. De printede værdier ændrer sig, som forventet ved de andre billeder, da disse bliver højere jo længere væk personen bevæger sig fra Kinecten. Dermed kan vi konkludere, at der er en sammenhæng mellem, hvor langt væk en person står fra Kinecten og den dybdeinformation computeren får. Forskellen i dybdeværdierne fra det højre billede på *figur 6.1* til det venstre billede på *figur 6.2* er ret stor i forhold til, hvor lidt personen har flyttet sig. Her ser vi dybdeværdien skifte fra 855 til 913 på kun 50 centimeters forskel. På det venstre billede på *figur 6.2* og det højre billede på *figur 6.2*, ser vi kun en dybdeværdis forskel på 913 til 1005, selvom der er en afstandsforskel på to meter. Dermed kan vi konkludere, at Kinecten sender færre informationer til computeren jo længere væk personen bevæger sig og dermed bliver informationen mere upræcis. Selv ved at bevæge sig 6,5 meter væk fra Kinecten, som ses på *figur 6.3* printes ikke en højere værdi end 1037. I forhold til vores viden om Kinecten burde dens maks. print være 2047, men præcis, hvornår Kinectens maks. dybdeværdi opnåes, vides ikke med sikkerhed ud fra denne test.



Figur 6.1: På billeder til venstre ses en person, som står med en 50 centimeter distance til Kinect kameraet. Dybdeværdien der bliver printet i console er 2047. På billedet til højre ses personen med en distance til Kinect kameraet på halvdanden meter. Dybdeværdien printet i console er 855.



Figur 6.2: På billedet til venstre ses personen med en to meter distance til Kinect kameraet, dybdeværdien printet i console er 913. På billedet til højre ses personen med en distance til Kinect kameraet på fire meter, dybdeværdien printet i console er ca. 1005.



Figur 6.3: På billedet en væg, som er målt til at være seks en halv meter væk fra Kinectens kamera. Dybdeværdien printet i console er 1040.

## 6.1.2 Frame differencing test

Test udført d. 22.05.2018, kl. 15:00.

Vi har valgt at teste frame differencing funktionen ved, at bruge to billeder. Vi har et billede som er helt hvidt og et hvor der er tilføjet en sort cirkel i midten, som repræsenterer forskel i pixels. Vi sætter disse to ind i programmet og tester vores frame differencing funktion ud fra disse to billeder, i stedet for Kinectens video input. Dette gør vi fordi, at vi ved, hvilket output vi forventer at få, når vi har disse to billeder som input. Når vi først viser det hvide billede og derefter det sorte, så ved vi, at der er sket en forskel i pixelsne der hvor, at den sorte cirkel er. Da vi viser de steder hvor, at der er sket en forskel med rød, vil det sige, at vi forventer at få et billede af en rød cirkel på en hvid baggrund. Nedenstående ses den kodedel hvor, at vi tester frame differencing funktionen:

```

    PImage img1, img2, diffImg;
    int threshold = 70;

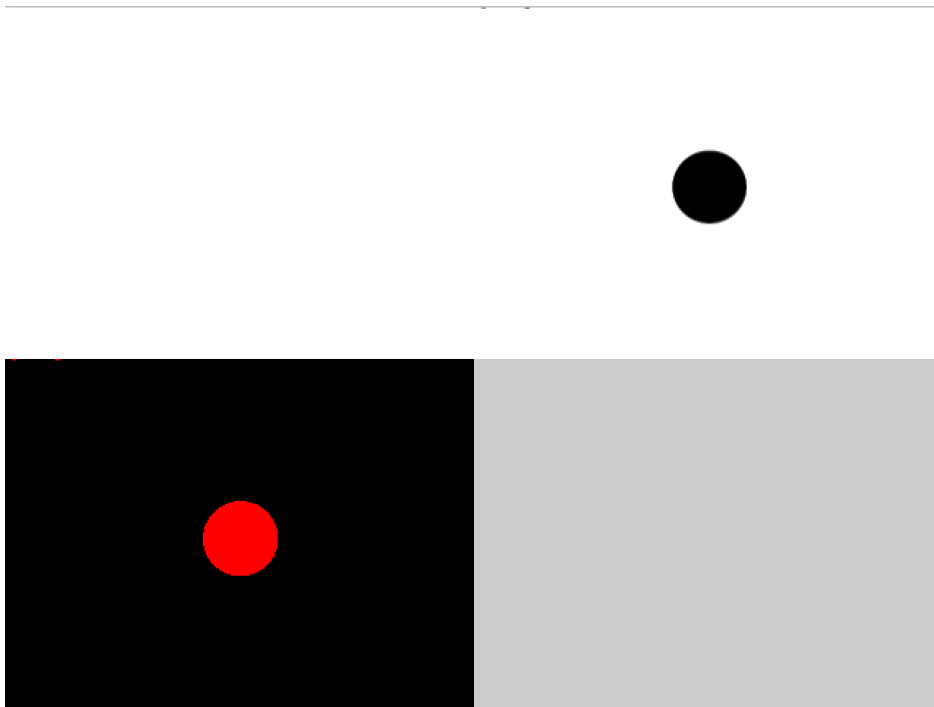
    void setup() {
        size(640, 480, P2D);
        img1 = loadImage("dot.png");
        img2= loadImage("white.png");
        img1.resize(320, 240);
        img2.resize(320, 240);
        img2.loadPixels();
        diffImg = frameDiff(img1, img2.pixels);
        diffImg.resize(320, 240);
    }

```

```

void draw() {
  image(img1, 320, 0);
  image(img2, 0, 0);
  image(diffImg, 0,240);
}

```



Figur 6.4: Illustrationen viser vores frame differencieng test. De øverste to billeder (det helt vide billede og billedet hvor der er tilføjet en sort cirkel) er vores input. Billedet med den sorte baggrund og den røde cirkel er vores output.

På den ovenstående figur ses vores testresultater. Som det kan ses har vores testresultat det forventede output. Funktionen danner en rød cirkel der hvor, at den sorte cirkel opstår - altså der hvor der sker en forskel i pixels.

### 6.1.3 Unitest og stresstest af blob-programmet

Test udført d. 22.05.2018, kl. 15:45.

På den nedenstående figur ses to skærbilleder taget fra **Processing-programmet**. Vi har tegnet blå cirkler på en tavle, med varierende former, længder og med varierende distance til hinanden. Testen er taget i et stort lokale med mange vinduer, hvilket vil sige, at der kommer meget lys ind i lokalet. Vi har valgt at lave en stresstest, fordi vi vil teste om blob tracking metoden også virker under ikke optimale forhold. De ikke optimale forhold er for eksempel lyset i lokalet, der påvirker programmet og den ikke særlig kraftige blå farve, som er brugt til at tegne cirkler på tavlen. Testen er udført således, at den blå farve er fundet ved hjælp af `mousePressed()` funktionen, fra farvesporings prototypen, og så er programmet kørt både med og uden at vise fundne blobs.

Udstyr:

System SKU: Aspire VN7-592G\_1039\_1.12

Processor: Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz, 2301 Mhz, 4 Core(s), 4 Logical Proc



GPU: Nvidia GeForce GTX 960M

WebCam: Resolution: 1280x720, FPS: 30



Figur 6.5: Skærbilledet viser vores blob test. På det venstre billede ses nogle blå cirkler tegnet på en tavle. På det højre billede ses vores blobs, vist med hvide firkanter.

På de to skærbilleder ses nogle blå cirkler i forskellige størrelser tegnet på en tavle. Som det kan ses, gør lyset en stor forskel på, hvilken nuance tavlen har. Man kan derfor forestille sig, at den blå nuance også er forskellig mellem de to billeder. Testen illustrerer dog alligevel, at man kan tracke de ønskede blobs med de rette grænseværdier. Derudover viser testen også, at `distThreshold` er en vigtig faktor, hvis man ønsker at vise alle små cirkler med blå farve. Det ses tydeligt i de to billeder, hvordan de blå cirkler bliver fanget af den samme blob. Desuden kan det ses på billederne, at nogle blå streger ikke kommer med i en blob. Det kan skyldes at farven på tavlen ikke er blå nok og i den største blob, skyldes det, at `distThreshold` er sat, så den blob ikke kan blive større. I sidste ende afhænger dette program meget af, hvilket udstyr der er til rådighed og hvilke omstændigheder programmet afprøves i. Den overordnede konklusion af testen er, at den klarede opgaven bedre end forventet, da den kunne identificere alle de blå farver.

Programmet brugt til testen er et blob-program meget tæt på komponentsbeskrivelsen af blob tracking. I forhold til prototypen farvesporing er testprogrammet uden partikelsystemet og uden et maksimum på, hvor mange blods der kan være. I stedet for røg-effekten vises den egentlige firkant blobben består af. Og ved hvert tryk på musen i programmet, vises `r`, `g` og `b` værdien for denne pixel. Ydermere kan `distThreshold` og `threshold` (for farven) ændres ved at trykke på en knap på tastaturet.

```

1         if (!found) {
2             Blob b = new Blob(x, y);
3             blobs.add(b);
4         }
5
6     for (Blob b : blobs) {
7         if (b.size() > 500) {
8             b.show();
9         }
10    }
11 }

```

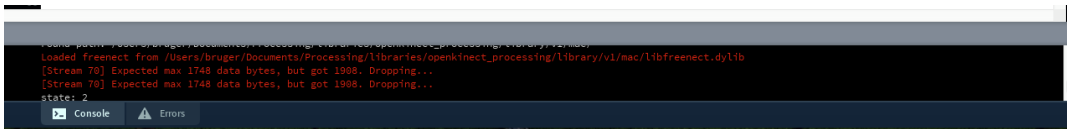
```
12
13 class Blob {
14     void show() {
15         stroke(0);
16         fill(255);
17         strokeWeight(2);
18         rectMode(CORNERS);
19         rect(minx, miny, maxx, maxy);
20     }
```

### 6.1.4 Fejlmeddelelser

Der er flere fejlmeddelelser, som viser sig under brug af prototyperne. Begge fejlmeddelelser bliver ofte vist, men programmet kan godt køre og fungerer selvom disse fejlmeddelelser bliver printet. Det er muligt, at disse fejl ville kunne løses.

A screenshot of a Processing console window. The text is red on a black background. It reads: "2018-05-21 11:02:08.525 java[20875:2316137] pid(20875)/eid(501) is calling TIS/TSM in non environment, ERROR : This is NOT allowed. Please call TIS/TSM in main thread!!!".

Figur 6.6: Skærbillede af en error printet i Processing console. Prototyperne kan godt køres i Processing selvom denne fejlmeddelelse printes i console.



Figur 6.7: Skærbillede af en fejl printet i Processings console. Prototyperne kan godt køres i Processing selvom denne fejlmeddelelse printes i console.

## 7.0 Til og fravalg

### 7.1 OpenFrameworks

Da vi påbegyndte projektet havde vi flere overvejelser og eksperimenterede med et redskab kaldet openFrameworks. Dette redskab er et open source værktøj, som gør brug af sproget C++ og har implementeret biblioteker som blandt andet Open Source Computer Vision Library (OpenCV) på en intuitiv måde(<https://openframeworks.cc/about/>). OpenCV var hovedårsagen til, at vi havde disse overvejelser, da dette anerkendte bibliotek har 2500 optimerede algoritmer, som bliver benyttet i både forsknings- og kommercielle regi (<https://opencv.org/about.html>). Da vores research omkring biblioteket i starten af projektet viste disse imponerende fordele, forsøgte vi at dykke ned i det tekniske og forsøgte at eksperimenterede med det. Vi mødte for mange udfordringer med SDK(System development kit) og IDE implantationer, som ikke førte os videre i processen omkring udforskningen af computer vision og derfor endte vi med at vælge Processing. Timerne vi brugte på at få et simpelt video input/output til at fungere i OpenFrameworks var for overvældende i forhold til den nominerede tid.

### 7.2 Processing

Processing er gearret til at udvikle programkoder, som kan visualiseres på kort tid (<https://processing.org/overview/> - 24/05/18). Dette har gavnet processen, da det ikke kræver meget viden eller erfaring, før man kan starte med at programmere og eksperimenterede. Derudover har Processing nogle biblioteker, som er lavet til Kinect samt webcam, og som dermed gør det nemmere at lave effekter, som benytter Kinect og webcams som input. Nogle af ulemperne ved at have brugt Processing er, at det for det første ikke er godt til at håndtere større mængder data. Dette har blandt andet været et problem, når vi i Vægmalning effekten ønskede, at brugeren skulle kunne tegne én lang streg, så længe programmet kører. Dette var ikke muligt, da programmet hurtigt fik for meget data, og derfor crashede efter 10-15 sekunder. For at undgå dette har vi måtte begrænse os og i stedet benytte os af et trail, som stadig ikke måtte være for langt. Noget andet er, at det har givet os nogle udfordringer i forhold til, når vi har skulle samle alle vores effekter i ét program. Da Processing er bygget op omkring en Setup() og en Draw() metode, har det skabt et problem at strukturere de mange enkelte effekter i et samlet program. Her havde det måske været smartere at benytte et program som IntelliJ, da det er nemmere at strukturere separate klasser i et større program.

### 7.3 Kinect v2

I Kinect v2 kameraet er der, ligesom i Kinect v1, et RGB kamera, en IR projektor og en IR sensor. I Kinect v2 anvendes et Time-of-flight system, der udregner afstanden baseret på den tid det tager lyset fra IR projektoren at ramme IR sensoren.(Wikipedia) Kinect v2 har en større præcision, som gør flere effekter mulige. Ved projektets start fik gruppen stillet tre Kinect v1 og en Kinect v2 til rådighed. Da vi fandt ud af, at Windows computere ikke kunne bruges med Kinect v1, "fik"Windows

brugerne i gruppen Kinect v2. Ideen var, at de andre prototyper ville kunne visualiseres med Kinect v2, når prototyperne var færdigt. Her gik det hurtigt op for gruppen, at Kinect v2 har en anden terminologi end Kinect v1 og at lave prototyperne designet til Kinect v1 om til at kunne visualiseres med Kinect v2 ikke ville være så nemt som først antaget. Samtidig gav Kinect v2 adgang til nogen andre biblioteker på Windows ned på Mac. Derudover sad Windows brugerne med en prototype, som ikke krævede de effekter, som benytter de egenskaber Kinect v2 har. Dermed kom vi frem til at brugen af Kinect v2 ikke var hensigtsmæssigt, især da det så krævede endnu et redskab kunne bruge det samlede interface.

## 7.4 Webcam

For at også Windows brugerne skulle eksperimentere med at programmere en prototype, måtte vi finde et andet alternativ, som skulle kunne bruges af både Windows og Mac af hensyn til Interfacet. Derfor har gruppen valgt at bruge webcammet til Farvesporing prototypen. Dette var, som før nævnt, både fordi at denne prototype ikke skulle bruge egenskaberne fra en Kinect alligevel og at en prototype lavet med brug af et webcam også ville kunne åbnes af en Mac bruger. Da vi brugt Kinect v1 til at programmere de andre prototyper, er det ikke alle prototyperne, der vil kunne bruges til en Windows computer. Dermed vil det samlede interface kun kunne benyttes af en Mac computer.

## 8.0 Diskussion og konklusion

Vi har i dette projekt forsøgt at besvare problemformuleringen: *Hvordan kan man ved hjælp af computervision teknikker skabe interaktiv kunst?*

Til udviklingen af vores prototyper har vi valgt at inddele gruppen i mindre arbejdsgrupper. Arbejdsgrupperne bestod af to gruppemedlemmer, som hver fik tildelt en effekt, som de skulle forsøge at udvikle. Dette viste sig at være en effektiv arbejdsfordeling, da det gav en sparringspartner. På den måde undgik at sidde fast med kodefunktioner og fejlmeddelelser, som vi måske ikke ville opdage eller løse selv. Vi valgte at arbejde i programmet Processing, da det er en simplere version af Java og dermed ikke kræver lige så meget viden og erfaring før, at man kan begynde at programmere. En af ulemperne ved Processing er programmets evne til, at håndtere større mængder data. Dette problem resulterer i, at der ikke skal meget til, før at programmet når sin maksimale "datamængde" og dermed crasher. Noget andet er, at Processing ikke inddeles i klasser på samme måde som andre Java baserede programmer gør. Dette skabte en udfordring sent i projektprocessen, da interfacet skulle kodes og samle alle prototyperne i et samlet program. Her havde det været en klar fordel at benytte et program, som tilbyder en bedre programstruktur. Vi valgte, ved projektets begyndelse, at eksperimentere både med en Kinect version 1 og 2. Vi ramte hurtigt ind i nogle problemer med Kinect v1 og Windows. Det viste sig, at disse ikke kunne bruges sammen og derfor havde vi to gruppemedlemmer, som udelukkende kunne arbejde med Kinect v2. Kinect v2 havde dog også nogle udfordringer, da den f.eks. ikke kunne benytte samme biblioteker som Kinect v1 til mac. Ved nærmere diskussion kom gruppen frem til, at de ting som Kinect v2 kunne, ikke ville blive nødvendigt til udviklingen af vores prototyper alligevel. Men da gruppen bestod af to Windows brugere og fire Mac brugere, måtte vi finde et andet redskab, så også Windows brugerene kunne programmere. Derfor endte vi med at bruge webcam til en af prototyperne.

Projektet er endt ud med fem forskellige prototyper og et samlet interface, som giver brugeren mulighed for at skifte imellem de forskellige effekter. Fire af prototyperne gør brug af Kinect v1, mens blob tracking gør brug af webcammet i den pågældende computer. Interfacet kan dermed kun køres på en Mac computer, hvis det skal være muligt at gøre brug af alle prototyperne, da Kinect v1 ikke ville kunne bruges på en Windows. Når man kigger tilbage på processen kan man overveje, om det havde været smartere at benytte nogle redskaber, som både kunne benyttes af Mac og Windows, når gruppen havde begge type computere. Vi har haft målsætninger for alle prototyperne, men det er ikke alle som det er lykkedes at opfylde inden afleveringsfristen. Alle prototyperne viser en effekt, men mangler forbedringer for at kunne opfylde vores målsætninger. En af de elementer som vi gerne ville have haft, at vores arbejdsprocess skulle indeholde, var en brugerevaluering. En brugevaluering foretages normalt for at undersøge, om det pågældende produkt eller program lever op til brugerens forventninger. Man kan se om produktet eller programmet skulle have nogle fejl, som man ikke selv havde opdaget, om programmet er intuitivt for brugeren at benytte eller hvorvidt brugeren opnår den ønskede oplevelse. Da interaktiv kunst er mere udefinérbart, er det her mere op til kunstneren selv at bestemme, hvilken oplevelse de ønsker brugeren skal have. Vores formål med programmet var, at det skulle fange brugerens opmærksomhed, det skulle være intuitivt og ikke mindst skulle det give brugeren lyst til at eksperimentere rundt med de forskellige effekter. Vi kunne have testet om vores program opfylder disse krav, ved at placere det et sted hvor, at der er mange forbigående

(f.eks. ved RUC's kantine eller ved Studenterhuset) og dermed observere, om effekterne fanger de forbigåendes opmærksomhed, om det giver dem lyst til at stoppe op og prøve det af, om der er nogle af effekterne som virker mere fængende end andre osv. Her kunne man også gøre brug af et spørgeskema og dermed få en direkte respons fra testpersonerne. Dette kunne man både have gjort undervejs i udviklings processen, så man kunne arbejde videre ud fra de observationer man gjorde sig, og man kunne gøre det når programmet er færdigt for at se, om man havde opnået den ønskede effekt hos de forbigående/brugere. Da rammerne ikke er faste indenfor interaktiv kunst og oplevelsen med installationerne er subjektive, ville brugerevalueringen også blive meget subjektiv og ikke direkte målbar. Med en brugerevaluering mener vi, at kunne konkludere, om programmet opfylder de mål som vi selv havde sat. Nemlig om det fanger deres opmærksomhed, giver dem lyst til at eksperimentere og ikke mindst om, det er intuitivt at benytte.

Der er en masse ting som vi er blevet klogere på i løbet af arbejdsprocessen, og som vi kan tage med os videre til fremtidige projekter. Vi er blevet klogere på, hvordan en udviklingsproces ser ud og hvordan sådan én langt fra følger en lineær arbejdsproces. Vi har lært, at udvikling af prototyper gavner udviklingsprocessen og ikke mindst, at man ikke kan starte på disse for tidligt. Vi har lært hvordan en iterativ arbejdsproces ser ud og sidst men ikke mindst er vi blevet klogere på, hvordan man ved hjælp af motiontracking og computervision teknikker, kan skabe interaktiv kunst.

## 9.0 Litteraturliste

- Andersen, M.R., Jensen, T., Lisouski, P., Mortensen, M.K., Gregersen, T., og Ahrendt, P. Kinect Depth Sensor Evaluation for Computer Vision Applications. Technical Report Electronics and Computer Engineering, [S.l.], v. 1, n. 6, sep. 2012. ISSN 2245-2087. Available at: <https://tidsskrift.dk/ece/article/view/21221>; Senest besøgt 27-maj-2018;
- About | openFrameworks; <https://openframeworks.cc/about/>; Senest besøgt 27-maj-2018;
- About - OpenCV library; <https://opencv.org/about.html>; Senest besøgt 27-maj-2018;
- Color and Image Processing; <http://courses.cs.vt.edu/~cs4624/s98/sspace/imgproc/index.html>; Senest besøgt 27-maj-2018;
- Canon; [http://cpn.canon-europe.com/content/education/infobank/capturing\\_the\\_image/ccd\\_and\\_cmos\\_sensors.do](http://cpn.canon-europe.com/content/education/infobank/capturing_the_image/ccd_and_cmos_sensors.do); Senest besøgt 27-maj-2018;
- Edwards, Natalie; Lange, Jessica; "Project Natal Fact Sheet May 09", 2009, [online] <https://web.archive.org/web/20120121223600/http://download.microsoft.com/download/A/4/A/A4A457B3-DF5D-4BF2-AD4E-963454BA0BCC/ProjectNatalFactSheetMay09.zip>; Senest besøgt 27-maj-2018;
- elsotanoperdido.com; <https://www.elsotanoperdido.com/noticias/microsoft-lanza-kinect-para-windows/2012020230268>; Senest besøgt 27-maj-2018;
- Ikeuchi, Katsushi (2014); Computer Vision - A Reference Guide, Springer Science+Business Media New York 2014; Springer; Boston; MA
- Microsoft; <https://msdn.microsoft.com/en-us/library/jj131033.aspx>; Senest besøgt 27-maj-2018;
- Noble, Joshua; 2012; Programming Interactivity: 2. edition;
- Processing; <https://processing.org>; Senest besøgt 27-maj-2018;
- Processing; Environment (IDE); <https://processing.org/reference/environment/>; Senest besøgt 27-maj-2018;
- Processing; captureEvent() Language (API) Processing 3+; [https://processing.org/reference/libraries/video/captureEvent\\_.html](https://processing.org/reference/libraries/video/captureEvent_.html); Senest besøgt 27-maj-2018;
- Schlegel, Andreas; <http://www.sojamo.de/libraries/controlP5/>; Senest besøgt 27-maj-2018;
- Seevinck, Jennifer; 2017; Interaction in Art and Computing. In: Emergence in Interactive Art. Springer Series on Cultural Computing; Springer, Cham;
- Shiffman, Daniel; 2012; The Nature of Code: Simulating Natural Systems with Processing.

- Singla, Nishu; 2014; Motion Detection Based on Frame Differencing Method. International Journal of Information and Computer Technology, volume 4;
- Szymczyk, Matthew; 2014; <https://zugara.com/how-does-the-kinect-2-compare-to-the-kinect-1>; Senest besøgt 27-maj-2018;
- Takahashi, Dean; 2013; <https://venturebeat.com/2013/01/20/beyond-kinect-primesense-wants-to-drive-3d-sensing-into-more-everyday-consumer-gear/#FoKgcL716z2qct25.99>; Senest besøgt 27-maj-2018;
- Wikipedia; 2018; [online] <https://en.wikipedia.org/wiki/Kinect>; Senest besøgt 27-maj-2018;
- Youtube; The Coding Train; <https://www.youtube.com/user/shiffman>; Senest besøgt 27-maj-2018;
- Youtube; 11.7: Computer Vision: Blob Detection - Processing Tutorial; <https://www.youtube.com/watch?v=ce-212wRq08>; Senest besøgt 27-maj-2018;