

Augmented Reality Bordspellen

Wouter Franken

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen,
hoofdspecialisatie Mens-machine
communicatie

Promotor:
Prof. dr. ir. Philip Dutre

Assessor:
TODO

Begeleider:
Ir. J. Baert

© Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail info@cs.kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Voorwoord

DANKWOORD

Wouter Franken

Inhoudsopgave

Voorwoord	i
Samenvatting	iv
Lijst van figuren en tabellen	v
Lijst van afkortingen en symbolen	vii
1 Inleiding	1
1.1 Lorem ipsum 4–5	1
1.2 Lorem ipsum 6–7	1
2 Het eerste hoofdstuk	3
2.1 Eerste onderwerp in dit hoofdstuk	3
2.2 Tweede onderwerp in dit hoofdstuk	3
2.3 Besluit van dit hoofdstuk	4
3 Naïeve legoblokdetectie	5
3.1 Begrippen	5
3.2 Blokdetectie op basis van simpele geometrische informatie	6
3.3 Implementatie, resultaten en evaluatie	9
3.4 Vereenvoudiging: blokdetectie met behulp van een 2D grid	15
3.5 Uitbreidung: Tree Adaptive A*	20
3.6 Besluit	22
4 Legoblokdetectie op basis van CAD modellen	23
4.1 Begrippen en algoritme	23
4.2 Features	26
4.3 Evaluatie features	28
4.4 Classificatie methodes	32
4.5 Evaluatie classifier methodes	33
4.6 Discussie / Besluit	37
5 Legoblokdetectie op meerdere niveau's	41
5.1 Begrippen	41
5.2 Algoritme	41
5.3 Tweede onderwerp in dit hoofdstuk	42
5.4 Besluit	42
5 Besluit	11

INHOUDSOPGAVE

A De eerste bijlage	15
A.1 Meer lorem	15
A.2 Lorem 51	16
B De laatste bijlage	17
B.1 Lorem 20-24	17
B.2 Lorem 25-27	18
Bibliografie	19

Samenvatting

ABSTRACT

Lijst van figuren en tabellen

Lijst van figuren

3.1	Twee voorbeelden die aangeven hoe de pose vanuit een contour gevonden kan worden.	8
3.2	Resultaat van het naïeve algoritme.	10
3.3	Resultaat van het naïeve algoritme bij veel schaduw.	11
3.4	Resultaat van het naïeve algoritme bij rode tint in het beeld.	11
3.5	Simpele (links) en complexe scene (rechts) voor performantie evaluatie van naïeve blokdetectie.	11
3.6	Gemiddelde performantie van het naïeve algoritme.	12
3.7	Gemiddelde performantie van het "Lemming updatenonderdeel van het naïeve algoritme.	13
3.8	Ruis in threshold vóór morfologische operaties werden toegepast.	16
3.9	Fout van het vereenvoudigde naïeve algoritme in de vorm van schaduw.	16
3.10	Resultaat van het tweede naïeve algoritme.	18
3.11	Gemiddelde performantie van de vereenvoudiging van het naïeve algoritme.	18
3.12	Voorbeeld van het Tree Adaptive A* algoritme (uit [HSKM11]).	20
3.13	Gemiddelde performantie van de uitbreiding op de vereenvoudiging van het naïeve algoritme.	21
4.1	Pose van het CAD model van een 2x2 legoblokje.	24
4.2	Uitgebreide Haar-like featureset [LM02].	26
4.3	Een 9x9 MB-LBP operator: elk blok bestaat uit 9 pixels [LZL ⁺ 07].	27
4.4	De 20 afbeeldingen die als testset werden gebruikt.	30
4.5	Feature performantie tijdens detectie.	31
4.6	Feature robuustheid.	32
4.7	Robuustheid classifier methodes.	36
4.8	Classifier methodes performantie tijdens detectie.	36
4.9	Impact schaalfactor op performantie en robuustheid van HOG feature.	37
4.10	Een voorbeeld van een legoblokconstructie waarin het groen omlijnde vlak het enige is wat we van die legoblok zien.	38
4.11	De vergroting van een rand in een muur van legoblokken.	38
5.1	Deze figuur toont hoe de calibratiefase met twee kleuren verloopt.	42

Lijst van tabellen

4.1	Parameters gebruikt tijdens training en detectie van features met cascade classificatie als classificatie methode.	29
4.2	Feature training tijdsduur.	31
4.3	Parameters gebruikt tijdens training en detectie van features met SVM als classificatie methode.	34
4.4	Classifier methodes training tijdsduur.	35

Lijst van afkortingen en symbolen

Afkortingen

Symbolen

Hoofdstuk 3

Naïeve legoblokdetectie

In dit hoofdstuk bespreken we een naïef algoritme om legoblokken te detecteren. Dit algoritme zal op basis van kleurthresholding de legoblokken in de afbeelding lokaliseren en vervolgens op basis van de geometrie van een legoblok de pose bepalen. Hierna vereenvoudigen we dit algoritme door in plaats van expliciet de pose van een legoblok te bepalen gebruik te maken van het grid van de virtuele wereld in ons AR spel. Ten slotte breiden we deze vereenvoudiging uit met een efficiënter algoritme voor het zoeken van een pad in de virtuele wereld, wat de performantie ten goede komt.

In sectie 3.1 worden eerst enkele begrippen uitgelegd. Vervolgens wordt het naïeve blokdetectie algoritme uitgelegd in sectie 3.2. Sectie 3.3 bespreekt dan de implementatie, resultaten en evaluatie van dit algoritme. Verder wordt in sectie 3.4 de vereenvoudiging van het algoritme behandeld. De uitbreiding op de vereenvoudiging wordt uitgelegd in sectie 3.5 en we sluiten af met een besluit in sectie 3.6.

3.1 Begrippen

Thresholding is een operatie die vaak in beeldverwerking wordt gebruikt om twee delen in een afbeelding te scheiden van elkaar (vaak voorgrond en achtergrond). Dit kan door bijvoorbeeld alle pixels binnen een bepaald interval te scheiden van de pixels die buiten dit interval vallen. Het resultaat is een binaire afbeelding waarin de achtergrond meestal wordt aangeduid met zwart en de voorgrond met wit.

YUV, **RGB** en **HSV** zijn drie verschillende kleurenruimtes die elk op een andere manier kleuren definiëren. Bij YUV gebeurt dit door de helderheid (Y) van de kleur te scheiden van twee kleurcomponenten (U en V). In RGB zijn er enkel drie kleurcomponenten waarbij de helderheid dus in deze componenten verwerkt zit. HSV scheidt tint (H), verzadiging (S) en helderheid (V) van elkaar. Omdat HSV drie componenten scheidt die een duidelijk verschillende invloed hebben op de uiteindelijke kleur wordt deze kleurenruimte vaak gebruikt voor kleurthresholding. In deze kleurenruimte is het immers eenvoudiger om kleuren van elkaar te scheiden zonder de verzadiging of de helderheid te beïnvloeden. In dit hoofdstuk en verdere

3. NAÏEVE LEGOLOKDETECTIE

hoofdstukken gaan we er steeds van uit dat de waarden van H, S en V liggen tussen 0 en 255.

Pinhole model is het model dat OpenCV [Its15] gebruikt om 3D punten van een scène te transformeren naar de afbeelding via een perspectieftransformatie. In dit hoofdstuk wordt het gebruikt om 2D punten om te zetten naar 3D (en omgekeerd). Dit is de formule van het pinhole model:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Hierbij is (u, v) het 2D punt, (X, Y, Z) het 3D punt, de eerste matrix in het rechterlid de camera intrinsics matrix (bekomen door camera calibratie), de tweede matrix in het rechterlid is de camera extrinsics matrix (afgeleid uit de ligging van de markers) en s , tenslotte, is een schaalfactor. Wanneer de twee cameramatrices bekend zijn kunnen we een 3D punt omzetten in 2D en vice versa. Een belangrijke opmerking hierbij is dat, bij omzetting van 2D naar 3D, wat informatie te kort is (het aantal dimensies verhoogt immers). Deze extra informatie kan worden gegeven in twee vormen: ofwel kennen we de uiteindelijke Z coördinaat, ofwel kennen we de uiteindelijke X en Y coördinaten. Daarna is het slechts een kwestie van een stelsel van vergelijkingen uit te werken.

Morfologische operaties zijn operaties die de geometrie van vormen in een binaire afbeelding kunnen wijzigen. De twee basis operaties zijn *dilate* en *erode*. In deze operaties komt het erop neer dat we de convolutie nemen van de afbeelding met een kernel, die eender welke vorm of grootte kan hebben. Hierbij wordt de kernel over de afbeelding geschoven en dan wordt de pixel in het ankerpunt van de kernel vervangen door een maximum of minimum van alle pixels die binnen de kernel vallen. Bij een *dilate* operatie is dit het maximum en bij een *erode* operatie is dit het minimum. Bij een *dilate* en *erode* operatie wordt een zwart vlak omgeven door wit dus respectievelijk kleiner en groter.

Naast de basis morfologische operaties bestaan ook nog de veel gebruikte *open* en *close* operatie. *Open* is in feite een *dilate* operatie toepassen op een afbeelding die eerder een *erode* operatie onderging. De *close* operatie is het omgekeerde. De *open* operatie wordt vaak gebruik om kleine witte vlekjes te verwijderen, terwijl de *close* operatie wordt gebruikt om kleine zwarte vlekjes te verwijderen.

3.2 Blokdetectie op basis van simpele geometrische informatie

Dit naïeve blokdetectie algoritme steunt op de geometrie van een legoblok om te bepalen waar legoblokken zich bevinden en wat hun pose is. Het algoritme verloopt in drie fases die in deze sectie worden beschreven. Eerst wordt de locatie van de legoblok bepaald in de afbeelding door gebruik te maken van kleurthresholding. Vervolgens wordt de pose van de blok bepaald door te steunen op de geometrische

3.2. Blokdetectie op basis van simpele geometrische informatie

informatie van een legoblok. Ten slotte wordt de blok toegevoegd aan de virtuele wereld.

3.2.1 Locatie van legoblok bepalen

Eerst wordt de nieuwe frame van het YUV formaat omgezet naar het HSV formaat (via RGB, vanwege de beperkingen van OpenCV). Dit is een erg belangrijke stap omdat het HSV formaat (zoals reeds aangehaald in sectie 3.1) veruit het eenvoudigste formaat om te gebruiken bij thresholding operaties.

Op de HSV frame wordt vervolgens kleurthresholding toegepast om de legoblokken te scheiden van de wit-zwarte achtergrond. In dit eerste algoritme werd enkel gewerkt met de kleur rood. Experimenteel werd ondervonden dat om rood te scheiden van een wit-zwarte achtergrond de HSV waarden in het volgende interval moeten liggen:

$$160 < H < 180; 153 < S < 255; 30 < V < 255$$

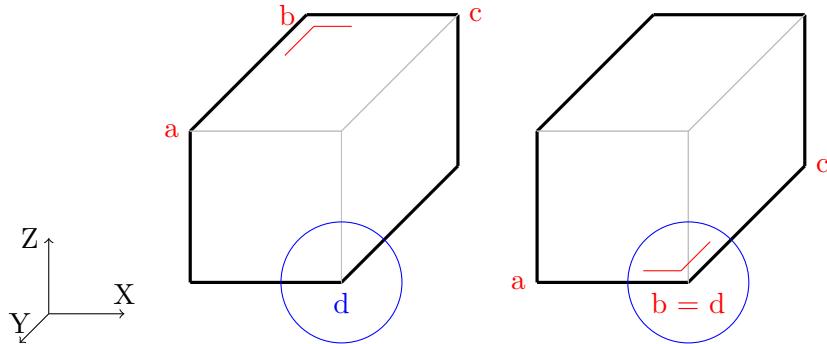
Contouren worden dan bepaald met behulp van het algoritme dat beschreven wordt in [S+85]. Dit algoritme is gebaseerd op een simpel algoritme dat de afbeelding scant naar contouren. Wanneer men zulke contour tegenkomt wordt deze aangeduid, bewaard en op zoek gegaan naar de volgende. Bovendien wordt een hiërarchische boom opgebouwd die aangeeft hoe de verschillende contours met elkaar gerelateerd zijn maar dit is niet van belang voor dit algoritme.

In de volgende stap wordt deze contour benaderd met een polygon. Hiervoor wordt het Douglas-Peucker algoritme gebruikt [DP73]. Dit is een eenvoudig algoritme dat iteratief een lijnsegment gaat zoeken dat minder punten bevat dan het oorspronkelijke lijnsegment (in ons geval een contour). Deze stap is nodig omdat een contour uit een enorme hoeveelheid punten bestaat waarvan we enkel de belangrijkste, de hoekpunten van de legoblok, nodig hebben. Het Douglas-Peucker algoritme bevat een parameter ϵ die aangeeft wat de maximale afwijking mag zijn van de benadering ten opzichte van de oorspronkelijke contour. Om ervoor te zorgen dat we een zo goed mogelijke benadering verkrijgen in alle gevallen, maken we deze ϵ steeds groter tot we net een maximum van zes hoekpunten hebben bereikt. Het adaptief aanpassen van deze ϵ parameter zorgt ervoor dat we de kwaliteit van de contour behouden maar in ruil daarvoor moeten we de veronderstelling maken dat de contour van een legoblok bestaat uit maximaal zes hoeken (wat het geval is voor alle balkvormige legoblokken indien ze vanuit een perspectief standpunt worden bekeken). Dit resulteert in enkele nadelen die later aan bod zullen komen.

Nu kan het zijn dat we contours overhouden met een te kleine oppervlakte door ruis in de kleurthreshold. Om deze te verwijderen wordt de oppervlakte van alle contours bekeken en contours verwijderd met een oppervlakte die 100x kleiner is dan de oppervlakte van de grootste contour.

Het bepalen en verfijnen van de contour is afgelopen, nu moet bepaald worden wat de pose is van de legoblok. Dat gebeurt in het tweede deel van het algoritme.

3. NAÏVE LEGOBLOKDETECTIE



FIGUUR 3.1: Twee voorbeelden die aangeven hoe de pose vanuit een contour gevonden kan worden.

3.2.2 Bepalen van de pose

Na het vinden van de contour moet hieruit worden achterhaald wat de pose van de legoblok is. Hiervoor bepalen we eerst in welk vlak van de legoblok elk hoekpunt van de contour ligt: onder- of bovenvlak. Dit geeft ons de Z coördinaat van al deze hoekpunten waardoor ze kunnen worden omgezet van 2D naar 3D via het pinhole model. Bij uitbreiding zijn dan alle hoekpunten van de legoblok in 3D gekend en dus kennen we ook de pose van de legoblok.

Eerst wordt uitgelegd hoe we kunnen achterhalen in welk vlak (boven of onder) elk hoekpunt van de contour ligt. Deze redenering wordt verduidelijkt met twee voorbeelden in figuur 3.1, waarin twee legoblokken worden getoond met hun contourlijnen aangeduid met de vette zwarte lijnen.

Uit de geometrische informatie van een legoblok weten we dat minstens drie opeenvolgende hoekpunten in hetzelfde Z -vlak liggen. Om te bepalen welke drie hoekpunten dit zijn zetten we alle 2D contourpunten om naar 3D (geprojecteerd op het grondvlak $Z = 0$) en bepalen we welke drie opeenvolgende 3D hoekpunten een hoek vormen die het dichtst bij 90 graden ligt (deze punten noemen we (a, b en c). Van deze hoeken zijn we dan zeker dat ze in hetzelfde vlak liggen. In het eerste voorbeeld van figuur 3.1 ligt de hoek tussen drie hoekpunten in het bovenvlak dichtst bij 0, in het tweede voorbeeld is dit de hoek op het grondvlak. Een ander geval is onmogelijk omdat, na projectie op het grondvlak, hoeken tussen punten in een verschillend Z -vlak sterker zullen afwijken van 90 graden.

Vervolgens bepalen we het hoekpunt (d) dat in 3D (opnieuw geprojecteerd op grondvlak $Z = 0$) zich het dichtst bij de camera bevindt, hiervan kan met zekerheid gezegd worden dat het werkelijk in het vlak $Z = 0$ ligt. Punten van het bovenvlak zijn namelijk na projectie verder van de camera verwijderd dan hun overeenkomstige punten op het grondvlak. Dit dichtstbijzijnde hoekpunt is in de voorbeelden aangeduid met een blauwe cirkel. Nu kan met zekerheid gezegd worden of a, b en c in het onder- of bovenvlak van de legoblok liggen:

- Als $d == a$ OF $d == b$ OF $d == c$, dan liggen a, b en c in het ondervlak;

- Anders liggen a , b en c in het bovenvlak.

In het eerste voorbeeld van figuur 3.1 liggen de hoekpunten a , b en c dus in het bovenvlak aangezien hoekpunt d niet gelijk is aan één van deze hoeken. In het tweede voorbeeld is dit wel het geval en dus liggen de drie punten in het ondervlak. Dit geeft ons (via het pinhole model) de 3D posities van alle hoekpunten van de contour en bij uitbreiding van de volledige legoblok.

3.2.3 Legoblok toevoegen aan de virtuele wereld

Nu we de pose van de legoblok bepaald hebben wordt de legoblok toegevoegd aan de virtuele wereld. Om ervoor te zorgen dat blokken niet meermaals worden toegevoegd worden blokken die voldoende dicht bij elkaar liggen samengenomen met behulp van een voting mechanisme. Dit verloopt als volgt:

Wanneer elk hoekpunt van het ene legoblok dichter dan 0.8 cm ligt bij een hoekpunt van een reeds gedetecteerde blok worden deze blokken gezien als dezelfde blok. In dat geval kan gestemd worden op de grootte en positie van de blok door het gemiddelde te nemen van de groottes en posities van alle blokken die zo dicht bij elkaar liggen. De waarde 0.8 cm is niet toevallig gekozen: het is exact de helft van de breedte van een 2x2 legoblok, zo dicht kunnen legoblokken dus nooit bij elkaar liggen (dit is voor ons namelijk het kleinste legoblokje dat gedetecteerd kan worden).

Bovendien heeft een legoblok in de virtuele wereld nog een status waarin deze verkeert. De twee mogelijk statussen zijn om te bepalen wanneer de blok een deel van het spel wordt (*actief*) en wanneer de blok er geen deel meer van uitmaakt (*inactief*). Wanneer een blok minstens in drie frames werd gedetecteerd wordt hij actief, maar indien hij minstens drie frames achter elkaar niet werd gedetecteerd wordt hij inactief en wanneer hij na vijf frames achter elkaar niet is gedetecteerd wordt hij zelfs verwijderd. Dit mechanisme maakt het algoritme flexibeler om blokken te kunnen toevoegen of verwijderen uit het spel wanneer de speler dat wil.

3.3 Implementatie, resultaten en evaluatie

3.3.1 Implementatie

De volledige implementatie en experimenten zijn uitgevoerd op een Sony Xperia Z C6602 met een Qualcomm Quad-core 1.5 GHz Krait processor, Adreno 320 GPU, 2GB RAM en Android 4.4.4 KitKat. Alles werd geprogrammeerd in Java 7 met gebruik van de Cardboard VR SDK van Google voor stereo rendering voor de Google Cardboard en de OpenCV 2.4.9 bibliotheek [Its15] voor implementaties van computer vision algoritmes.

De omzetting van YUV naar HSV werd geïmplementeerd met de `cvtColor` methode van OpenCV. Voor de detectie van de contouren gebruikten we de `findContours` methode. Ten slotte werd ook nog de `approxPolyDP` methode gebruikt om de contouren met minder hoekpunten te beschrijven. Verder werden voornamelijk mathematische operaties van OpenCV gebruikt om de rest van het algoritme te implementeren.

3. NAÏEVE LEGOLOKDETECTIE

3.3.2 Resultaten

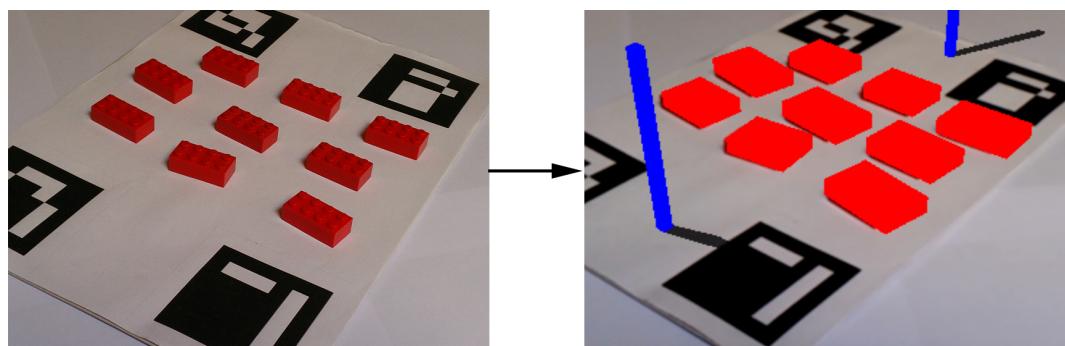
Figuur 3.2 toont het resultaat van het algoritme. De linkerafbeelding toont de legoblokken en de rechteraafbeelding wat de detectie van legoblokken oplevert.

Merk op dat de rendering van de gedetecteerde legoblokken een kleine afwijking bevat van de werkelijke legoblokken. Dit wordt veroorzaakt door een trage performantie van het algoritme, waardoor de rendering in feite enkele frames oud is.

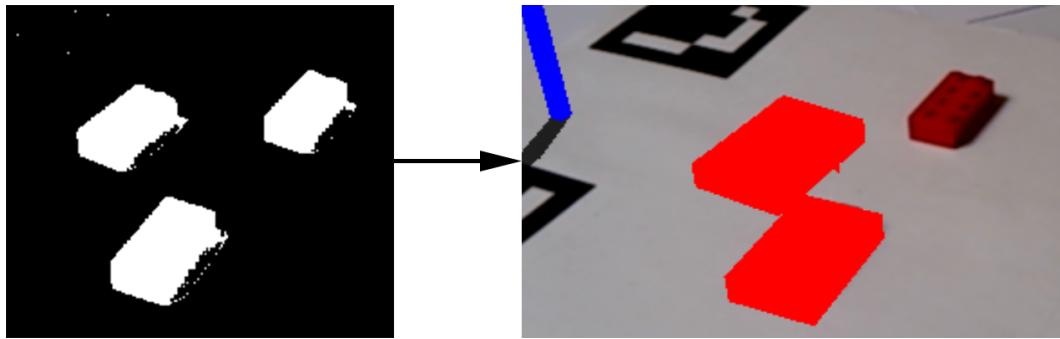
Verder valt ook op te merken dat de virtuele legoblokken iets groter zijn dan de werkelijke legoblokken. Dit is expliciet toegevoegd aan het algoritme omdat de virtuele wereld bestaat uit een discreet grid. Wanneer lemmings in dit discreet grid dan wandelen van node naar node kan het zijn dat ze deels in een legoblok wandelen omdat de blok niet exact stopt op een node. Om er dus zeker van te zijn dat de lemmings niet gedeeltelijk in een legoblok wandelen worden de virtuele legoblokken iets groter gemaakt dan de grootte van echte legoblokken.

In figuur 3.3 wordt aangetoond dat schaduw een negatief effect heeft op het resultaat van het algoritme. We zien in de threshold dat er wat artefacten op de plaats van de schaduw aanwezig zijn, dit resulteert uiteindelijk in legoblokken die moeilijk worden gevonden.

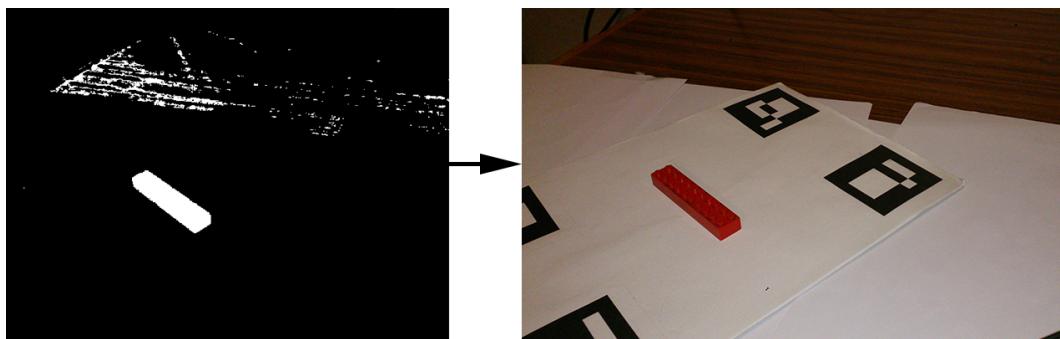
Omdat we kleurthresholding gebruiken is een rode tint in het beeld is absoluut uit den boze, elke kleur in het beeld dat een beetje in de grenzen van de threshold valt vormt een probleem. Dit tonen we aan in figuur 3.4, waarin een bruine achtergrond in beeld komt. We zien duidelijk dat veel ruis in de threshold voorkomt. Het gevolg hiervan is vooral dat het algoritme een heel stuk trager is dan normaal: soms duren alle berekeningen samen wel tot 5 seconden. Dit komt omdat ruis in de threshold het algoritme doet denken dat op die plaatsen ook legoblokken staan en daardoor zal het al die contouren ook willen verwerken wat veel tijd vraagt. Bovendien gebeurde het zelden wel eens dat de camera zelf zijn belichting aanpaste (bijvoorbeeld wanneer plots grote hoeveelheden donkere kleuren in beeld komen), hierdoor werd de threshold voor de legoblok zelf ook slechter wat ertoe leidde dat de blok helemaal niet meer gedetecteerd werd.



FIGUUR 3.2: Resultaat van het naïeve algoritme.



FIGUUR 3.3: Resultaat van het naïeve algoritme bij veel schaduw.

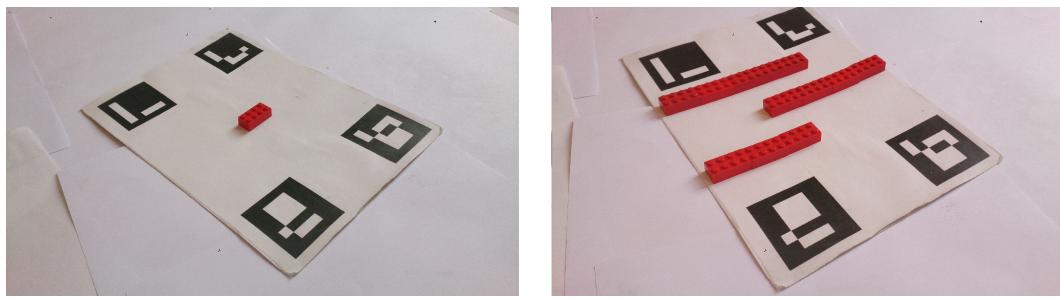


FIGUUR 3.4: Resultaat van het naïeve algoritme bij rode tint in het beeld.

3.3.3 Performantie

Evaluatiemethode

De performantie van werd geëvalueerd door de tijd op te meten van verschillende onderdelen van het algoritme terwijl een lemming loopt van het start- naar het eindpunt. Deze meting werd gedaan voor een eenvoudige en voor een complexe scène, zie figuur 3.5. Merk op dat dit scenes zijn die simpel en complex zijn voor



FIGUUR 3.5: Simpele (links) en complexe scène (rechts) voor performantie evaluatie van naïeve blokdetectie.

3. NAÏEVE LEGOBLOKDETECTIE

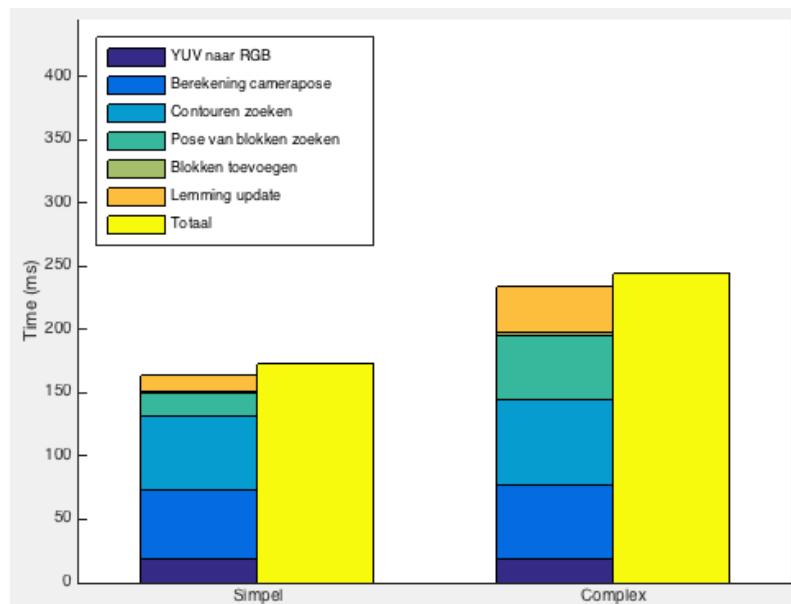
deze algoritmes, maw. het zijn scenes die beide algoritmes kunnen detecteren en respectievelijk weinig / veel legoblokken bevatten. Vervolgens werd per onderdeel het gemiddelde genomen van de gemeten tijd om geen grote uitschieters te hebben.

Evaluatie

Figuur 3.6 toont de gemiddelde performantie van het naïeve algoritme voor de simpele scene (links) en complexe scene (rechts). De gele blok stelt telkens de totale tijd die het algoritme nodig heeft, de gekleurde blokken geven de onderverdelingen aan van deze tijd:

- YUV omzetten naar RGB;
- Camerapose berekenen (via markers);
- Threshold berekenen en contouren van legoblokken zoeken (eerste deel van het algoritme);
- Pose van de blokken zoeken (tweede deel van het algoritme);
- Blokken toevoegen aan de virtuele wereld, mergen van blokken en voten voor grootte en positie van blokken (derde deel van het algoritme);
- Lemmings updaten (oa. pad van de lemming updaten).

Merk op dat de som van de gekleurde blokken niet exact gelijk is aan de grootte van de gele blok. Dit komt omdat enkel de belangrijkste onderdelen uit het algoritme zijn



FIGUUR 3.6: Gemiddelde performantie van het naïeve algoritme.

3.3. Implementatie, resultaten en evaluatie

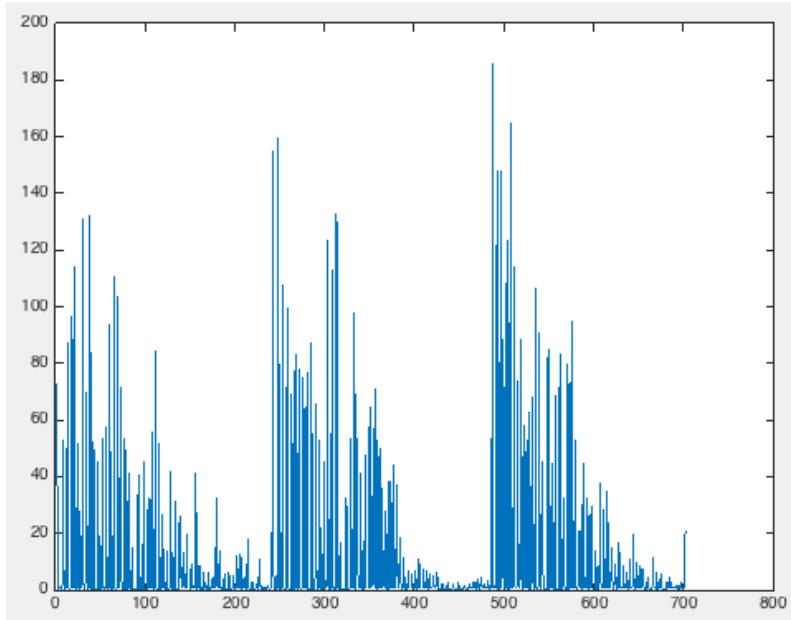
opgenomen in deze grafiek, enkele implementatie specifieke details zijn weggelaten omdat ze erg weinig tijd innemen en onbelangrijk zijn.

In de grafiek zien we dat het zoeken van de pose erg veel verschilt tussen de complexe en simpele scene. Dit is normaal aangezien het aantal legoblokken hoger ligt en dus moet dit naïeve algoritme meer iteraties doen in een complexe scène. Het zoeken van contours daarentegen duurt niet significant langer in een complexe scène. De verklaring hiervoor is dat het bepalen van de kleurthreshold, wat in beide gevallen slechts eenmaal moet gebeuren, het langst duurt en dat de rest van de berekeningen, die even vaak moeten gebeuren als het aantal gedetecteerde legoblokken, slechts een zeer kleine bijdrage zijn.

We merken ook op dat het updaten van de lemmingen een stuk langer duurt, dit is ook aannemelijk omdat bij de complexe scène het pad van een lemming van begin tot eindpunt een stukje langer is dan bij de simpele scène. De lemming moet immers om elke muur heen lopen en niet slechts om één legoblokje. Aangezien dit pad ook steeds opnieuw wordt berekend (om *on-the-fly* toevoegen van blokken toe te kunnen staan) is dit verschil significant.

Als we kijken naar de totale tijd de volledige berekening inneemt, dan wordt duidelijk dat dit algoritme erg traag is. Het komt neer op 8.9 fps voor de simpele scène en 3.2 fps voor de complexe scène, wat natuurlijk erg traag is voor een AR spel.

Figuur 3.7 geeft weer hoe de performantie van het updaten van de lemming veranderd gedurende het spel. Hiervoor hebben we de lemming drie maal laten lopen van start- tot eindpunt. We zien duidelijk een performantie van orde $O(1/x)$ die



FIGUUR 3.7: Gemiddelde performantie van het "Lemming updatenonderdeel van het naïeve algoritme.

3. NAÏVE LEGOLOKDETECTIE

zichzelf telkens herhaalt. Wanneer de lemming namelijk dichter bij het eindpunt komt wordt het pad tot het eindpunt steeds korts en dus duurt het zoeken van het pad steeds minder lang tot het moment dat hij het eindpunt heeft bereikt. Hierna zal een nieuwe lemming starten vanaf het beginpunt waardoor het terug lang duurt om het pad te zoeken. We zien hier drie iteraties omdat drie lemmingen hebben gewandeld van begin- tot eindpunt.

Merk ook op dat de tijdsduur regelmatig voor een korte tijd bijna nul is. Dit komt omdat we de lemming updaten aan een constante snelheid, dus ongeacht hoe lang het volledige algoritme erover doet de lemming zal steeds aan constante snelheid lopen. Dit kan ervoor zorgen dat de lemming tijdens de volgende lemming update de volgende node nog niet heeft bereikt waardoor voorlopig nog geen nieuw pad wordt berekent. Omdat dan geen pad wordt berekent valt de zwaarste berekening van het onderdeel "Lemming updaten" weg en dus is de tijdsduur van dit onderdeel dan zo goed als nul.

3.3.4 Besluit

Deze naïeve implementatie heeft één belangrijk voordeel:

- Ze laat toe dat legoblokken *on-the-fly* worden toegevoegd aan de virtuele wereld.

Aan deze methode zijn echter meer nadelen verbonden:

- Legoblokken mogen niet in een hoek naast elkaar staan omdat dan de veronderstelling dat een contour maximaal zes hoeken bevat niet meer geldig is.
- Enkel rode legoblokken kunnen worden gedetecteerd. Dit nadeel is echter eenvoudig weg te werken zoals wordt besproken in sectie 3.4.
- Legoblokken mogen niet in meerdere niveau's gebouwd worden. Muren (meerdere legoblokken op elkaar maar de breedte en de lengte van de constructie is op elk niveau gelijk) zouden in principe wel kunnen (ze hebben immers maximaal zes hoekpunten) maar dan moet op één of andere manier de grootte van deze muur bepaald worden, dit wordt uitgebreider behandeld in hoofdstuk 5. Andere constructies in de hoogte kunnen echter niet omdat we dan meer dan zes hoekpunten deel kunnen uitmaken van de contour.
- Het algoritme kan niet goed om met veranderingen qua belichting. Dit komt omdat we puur thresholden op basis van kleur maar deze waarden zijn sterk afhankelijk van welke belichting we gebruiken. In hoofdstuk 5 komt een calibratiemethode aan bod die dit probleem verhelpt.
- Zoals aangegeven in de resultaten geeft een rode tint in het beeld een erg slecht resultaat, vooral qua performantie.
- Performantie is erg laag.

Om een deel van deze nadelen weg te werken en bovendien de berekening van de pose eenvoudiger te maken voeren we in de volgende sectie een vereenvoudiging van dit algoritme in.

3.4 Vereenvoudiging: blokdetectie met behulp van een 2D grid

In deze vereenvoudiging zullen we in plaats van werkelijk de pose en positie van de legoblok te bepalen enkel de threshold gebruiken. Deze wordt dan over het grid van de virtuele wereld gelegd om te bepalen welke nodes niet toegankelijk zijn. Eerst behandelen we de aanpassingen ten opzichte van het vorige algoritme, vervolgens bespreken we de gevallen die deze aanpassingen hebben en ten slotte de resultaten en performantie van deze vereenvoudigde versie.

3.4.1 Algoritme

Het eerste deel van het naïeve algoritme wordt sterk ingekort aangezien we enkel een threshold nodig hebben. Maar er zijn ook een tweetal toevoegingen om tegemoet te komen aan de nadelen van het algoritme:

Ten eerste berekenen we opnieuw een kleurthreshold maar deze keer niet enkel voor de kleur rood, ook voor geel en blauw. Op die manier willen we aantonen dat het algoritme zonder problemen met meerdere kleuren overweg kan. Dit zijn kleurengrenzen die werden gebruikt om in een HSV afbeelding te thresholden zodat het resultaat een afbeelding is waarin de legoblokken wit zijn en de achtergrond zwart:

$$\text{Rood} : 160 < H < 180; 153 < S < 255; 30 < V < 255$$

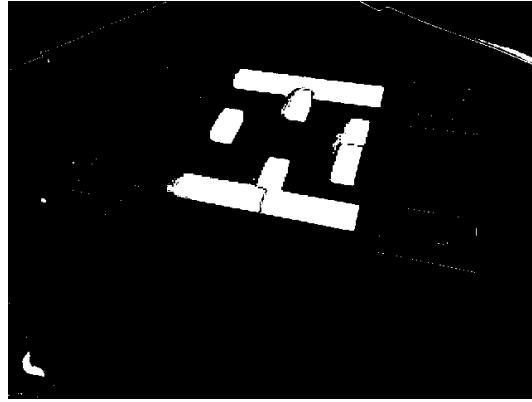
$$\text{Geel} : 135 < H < 160; 147 < S < 255; 30 < V < 255$$

$$\text{Blauw} : 0 < H < 112; 42 < S < 255; 13 < V < 255$$

Ten tweede, omdat we nu thresholden op meerdere kleuren kan het gebeuren dat er wat ruis optreedt in onze threshold. Deze ruis vertoont zich in hoopjes witte of zwarte lijntjes op plaatsen waar het niet hoort (zie figuur 3.8). Om dit te vermijden worden de morfologische operaties *open* en *close* toegepast op de threshold.

Het tweede en derde deel van het vorige algoritme (zie secties 3.2.2 en 3.2.3) valt volledig weg, logisch want we hebben geen contour bepaald. In plaats daarvan worden alle nodes van het grid van onze virtuele wereld naar 2D omgezet (via het pinhole model). Dit 2 dimensionaal grid kunnen we vervolgens overlappen met de eerder berekende threshold en wanneer een node van de virtuele wereld zich in een witte regio van de threshold bevindt beschouwen we deze als ontoegankelijk. Omdat we enkel de nodes die tijdens het A* algoritme worden geëxplooreerd moeten omzetten naar 2D, doen we dit tijdens de berekening van het pad en enkel voor de nodes die worden geëxploreerd.

3. NAÏVE LEGOLOKDETECTIE



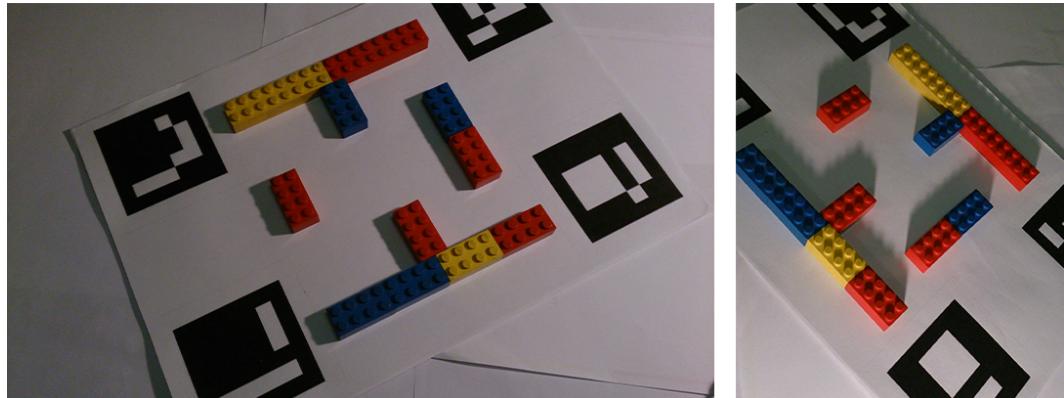
FIGUUR 3.8: Ruis in threshold vóór morfologische operaties werden toegepast.

3.4.2 Gevolgen

Deze vereenvoudiging is een grove versimpeling van het eerste algoritme aangezien de positie en grootte van een legoblok niet meer expliciet worden bepaald. Nu vragen we ons af welke consequenties dat heeft.

Ten eerste wordt een fout gemaakt in het aantal nodes die bezet zijn door een legoblokje. Deze fout wordt gemaakt omdat een onbezette deel schuilt achter de legoblok waardoor dit niet zichtbaar is in de threshold en deze delen dus onterecht als ontoegankelijk worden beschouwd. Sterker nog: we kunnen precies definiëren dat die fout even groot is als de schaduw op het grondvlak wanneer een lichtbron vanuit de camera schijnt in dezelfde richting als de camera kijkt. Afbeelding 3.9 toont de fout in de vorm van schaduw als de camera op de plaats van de lichtbron zou staan. We zien dat de fout relatief klein is wanneer de camera hoog boven het grondvlak zou staan (links) en groter wordt wanneer de camera dichter naar het grondvlak toe zou bewegen (rechts).

Ten tweede heeft deze vereenvoudiging ook een aantal voordelen in die zin dat



FIGUUR 3.9: Fout van het vereenvoudigde naïeve algoritme in de vorm van schaduw.

3.4. Vereenvoudiging: blokdetectie met behulp van een 2D grid

het enkele nadelen uit het naïeve algoritme wegwerkt:

- Legoblokken in meerdere kleuren kunnen gebruikt worden, het gevolg is wel dat morfologische operaties de threshold moeten oppoetsen om er ruis uit te halen zie figuur 3.8. Dit werd hoogstwaarschijnlijk veroorzaakt door de belichting die niet altijd constant is, zodat de thresholdwaarden niet altijd even nauwkeurig zijn. Merk op dat dit even goed in het eerste algoritme had kunnen worden geïmplementeerd.
- Legoblokken kunnen nu ook in een hoek tegen elkaar staan, wat bij het naïeve algoritme niet was toegestaan. Een nadeel is echter wel dat meerdere niveau's in dit algoritme helemaal uitgesloten zijn, terwijl dit in theorie in het vorige algoritme wel was toegestaan (met bepaalde restricties). Dit komt omdat wanneer we de constructie hoger maken, de fout van deze vereenvoudiging steeds groter wordt. Er is immers een nog groter gedeelte van het grondvlak, waar geen legoblokken op staan, verborgen achter de legoblokconstructie.

3.4.3 Implementatie

De morfologische operaties werden geïmplementeerd met behulp van de `morphologyEx` functie van OpenCV. Deze operatie werd in dit algoritme gebruikt om eerst een *close* operatie met een ellipsvormige kernel van 9x9 en vervolgens een *open* operatie met een ellipsvormige kernel van 17x17 uit te voeren op de kleurthreshold. Experimenteel werd ondervonden dat deze parameters het beste werken.

Verder werd er ook voor gekozen om niet elke keer het volledige grid van de virtuele wereld om te zetten van 3D naar 2D. In plaats hiervan worden enkel de nodes van het grid omgezet die geëxplooreerd worden door het A* algoritme. Deze keuze is een kwestie van performantie: ten eerste omdat het grid elke keer opnieuw van 3D naar 2D moet worden omgezet (de pose van de camera is immers elke keer verschillend) en ten tweede omdat in het A* algoritme bijna nooit alle nodes worden geëxplooreerd.

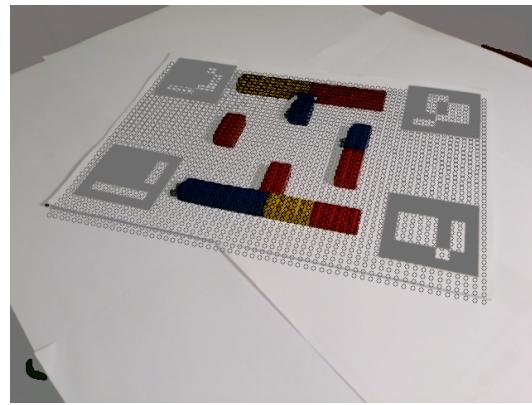
3.4.4 Resultaten en performantie

Resultaten

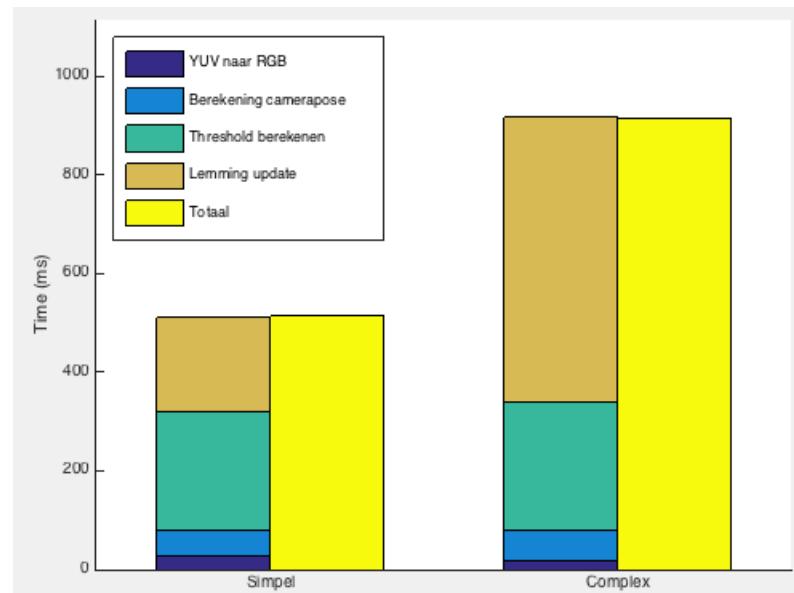
Voor deze vereenvoudiging is het niet mogelijk om een rendering te maken van de virtuele legoblok (zoals bij het eerste algoritme in sectie 3.2). Dit komt omdat de legoblokken in principe nooit volledig worden berekend: er wordt enkel een threshold gebruikt waar vervolgens een grid over wordt gelegd.

Figuur 3.10 toont het resultaat door het zwarte deel van de threshold deels transparant te maken. We kunnen duidelijk zien dat de legoblokken goed werden gedetecteerd via kleurthresholding. Ook werd over het grondvlak het grid gelegd, hierdoor is het duidelijk welke nodes niet toegankelijk zijn voor lemmings.

3. NAÏEVE LEGOLOKDETECTIE



FIGUUR 3.10: Resultaat van het tweede naïeve algoritme.



FIGUUR 3.11: Gemiddelde performantie van de vereenvoudiging van het naïeve algoritme.

3.4. Vereenvoudiging: blokdetectie met behulp van een 2D grid

Performantie

Figuur 3.11 toont de gemiddelde performantie van de vereenvoudiging van het naïeve algoritme voor de simpele scene (links) en complexe scene (rechts). Opnieuw stelt de gele blok de totale tijd voor en de gekleurde blokken zijn de onderverdelingen van de totale tijd. Omdat in deze vereenvoudiging een deel van het oorspronkelijke algoritme niet meer wordt gebruikt is de onderverdeling ook licht anders:

- YUV omzetten naar RGB;
- Camerapose berekenen (via markers);
- Threshold berekenen;
- Lemmings updaten (oa. pad van de lemming updaten en grid vergelijken met threshold).

Het belangrijkste dat we kunnen opmaken uit deze figuur is dat de totale tijd van deze vereenvoudiging meer dan dubbel zoveel is als de totale tijd van het oorspronkelijke algoritme. Deze vereenvoudiging kunnen we dus niet echt een verbetering noemen op vlak van performantie. Dit komt enerzijds omdat we elke keer opnieuw een volledig pad moeten berekenen en dus ook veel nodes moeten omzetten van 3D naar 2D. Anderzijds komt dit ook omdat de berekening van de threshold veel langer duurt dan de berekening van de contour bij het oorspronkelijke algoritme. De oorzaak hiervan vinden we bij het gebruik van meerdere kleuren waardoor ook enkele morfologische operaties nodig zijn wat erg dure operaties zijn.

We zien ook dat de berekening van de threshold in beide gevallen ongeveer even lang duurt: dit komt omdat een threshold berekenen onafhankelijk is van het aantal legoblokken dat wordt gedetecteerd. Het updaten van de lemmings duurt echter significant langer bij de complexe scène. Dit is normaal omdat het pad van de lemming dan langer is waardoor er dus meerdere nodes worden geëxplooreerd en dus ook meer die van 3D naar 2D moeten worden omgezet.

3.4.5 Besluit

Deze vereenvoudiging maakte het algoritme veel simpeler en loste ook enkele nadelen op. Helaas is dit algoritme wel erg traag en laat het ons helemaal niet toe om legoconstructies met meerdere niveau's te gebruiken.

Om het performantieprobleem te verhelpen breiden we in de volgende sectie dit algoritme nog eenmaal uit. In deze uitbreiding vervangen we het A* algoritme met Tree Adaptive A* om paden te zoeken in de virtuele wereld. In deze uitbreiding zullen grote delen van reeds berekende paden herbruikbaar zijn waardoor per frame minder nodes van 3D naar 2D moeten worden omgezet.

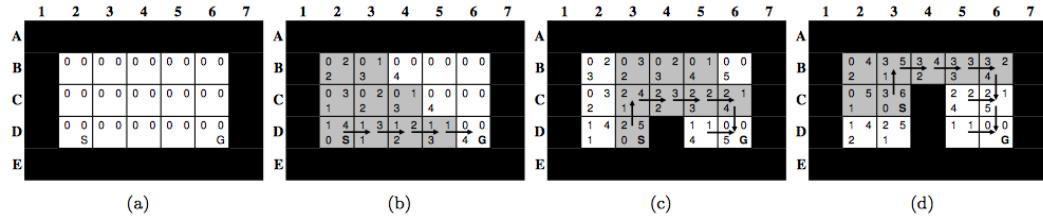
3.5 Uitbreiding: Tree Adaptive A*

In deze sectie leggen we uit hoe Tree Adaptive A* (Tree-AA*) werkt en in welke mate het de performantie heeft verhoogd. Dit algoritme is in dieper detail uitgelegd in [HSKM11].

3.5.1 Algoritme

Tree Adaptive A* is een algoritme dat een uitbreiding is op Path Adaptive A*, wat op zijn beurt een uitbreiding is op Adaptive A*. Het algoritme gaat uit van de *freespace* assumptie, waarin een agent zich een pad door een onbekend terrein moet banen naar een doelcel. Met een onbekend terrein bedoelen we dat hij niet weet waar obstakels zijn, hij kan deze enkel om zich heen voelen. Hierdoor zal de agent elke keer hij een onbekend obstakel voelt het pad met optimale kost moeten aanpassen. Bij elke aanpassing mag de agent wel gebruik maken van de suffixen van vroeger berekende minimale kost paden (behalve natuurlijk de stukjes paden die door het obstakel gingen). De vorige paden die herbruikbaar zijn worden steeds opgeslagen in een datastructuur die de *herbruikbare boom* wordt genoemd (het is namelijk een boom met zijn wortel in de doelcel). Zo moet de agent telkens slechts een deel van het pad aanpassen wat het zoeken van een pad een heel stuk efficiënter maakt. De grootste kost zit dan in de initiële zoektocht van een pad maar aangezien de meeste obstakels dan nog niet door de agent kunnen worden gezien zal dit best meevalen.

We zetten onze uitleg nog kracht bij met het voorbeeld dat ook werd aangehaald in de paper ([HSKM11]), zie figuur 3.12. Alle grijze tegels zijn geëxpandeerde nodes en de zwarte tegels zijn de obstakels waar de agent weet van heeft. Figuur 3.12a toont het beginsituatie, figuur 3.12b toont de situatie waarin gezocht wordt naar een pad met minimale kost tot de doelcel. Nog geen obstakels zijn gevonden omdat deze zich niet bevinden rondom de huidige cel van de agent (cel D1). Het zoeken stopt wanneer de agent net cel D6 wil gaan expanderen, het resulterende pad is dan [D1, D2, D3, D4, D5, D6]. De agent gaat dan naar cel D3 en voelt van daaruit een obstakel in cel D4. Het vorige minimale kost pad deels verwijderd omdat dit door het obstakel ging. Het stukje tussen D5 en D6 blijft echter over in de *herbruikbare boom* omdat dit nog steeds deel kan uitmaken van een minimaal kost pad (we weten immers niet of op plaats D5 ook een obstakel zou liggen). De agent vindt vervolgens een nieuw pad dat ook aan de boom wordt toegevoegd. In de laatste stap wordt een



FIGUUR 3.12: Voorbeeld van het Tree Adaptive A* algoritme (uit [HSKM11]).

obstakel ontdekt in cel C4 waardoor het stuk pad [D3, C3, C4, C5] verdwijnt uit de boom terwijl het stukje [C5, C6, D6] in de boom blijft. Bij het vinden van het nieuwe pad wordt het stukje [C6, D6] herbruikt.

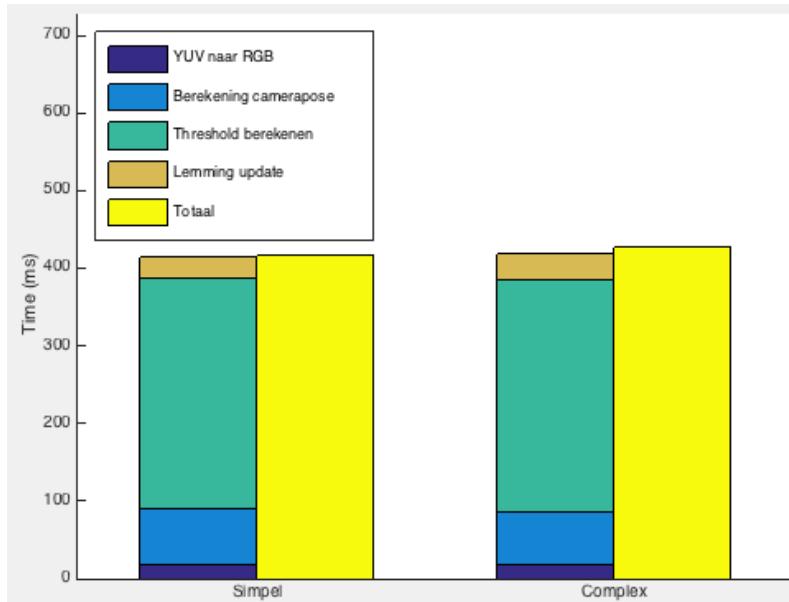
Het grote voordeel aan dit algoritme voor een AR spel is dat deze *herbruikbare boom* kan gedeeld worden tussen lemmingen. Dus als de wereld niet veranderd moeten weinig stukjes pad herberekend worden.

3.5.2 Performantie

Figuur 3.13 toont de gemiddelde performantie van de uitbreiding op de vereenvoudiging van het naïeve algoritme voor de simpele scène (links) en complexe scène (rechts). Opnieuw stelt de gele blok de totale tijd voor en de gekleurde blokken zijn de onderverdelingen van de totale tijd.

We zien dat de totale tijd is afgenomen ten opzichte van de niet-uitgebreide versie van het algoritme. De performantie is echter wel nog steeds lager dan bij de oorspronkelijke versie. Dit wordt veroorzaakt door de tijd die de morfologische operaties nodig hebben (zoals eerder aangegeven in sectie ??). Het is wel zo dat de tijd om een pad te zoeken van de lemming zoals verwacht sterk is afgenomen. Het is zelfs zo sterk afgenomen dat het berekenen van de threshold veel langer duurt dan alle andere operaties.

Het is bovendien opmerkelijk dat er geen significante verhoging is van de performantie tussen de simpele en complexe scène. Erg onlogisch is dit natuurlijk niet voor de berekening van de camera pose, omzetten van YUV naar RGB of berekenen van



FIGUUR 3.13: Gemiddelde performantie van de uitbreiding op de vereenvoudiging van het naïeve algoritme.

3. NAÏEVE LEGOLOKDETECTIE

de threshold aangezien dit onafhankelijk is van het aantal gedetecteerde legoblokken. Bij het updaten van de lemming is dit echter wel opmerkelijker aangezien het pad langer is maar door het gebruik van Tree Adaptive A* duurt het gemiddeld gezien telkens even lang. Dit toont aan dat dit algoritme robuust is tegen een verhoging van het aantal legoblokken in de scène.

3.6 Besluit

In dit hoofdstuk werd een naïef algoritme voorgesteld dat op basis van een kleur-threshold en geometrische eigenschappen legoblokken detecteert. Dit algoritme had echter een heleboel nadelen: oa. konden legoblokken niet in een hoek tegen elkaar staan, konden enkel rode legoblokken worden gedetecteerd en was de performantie erg laag. Daarom werd een vereenvoudiging van dit algoritme voorgesteld waarin de pose van een legoblok niet expliciet werd berekent. In plaats daarvan werd het 3D grid van de virtuele wereld over de threshold gelegd om te bepalen welke nodes ontoegankelijk waren. Het probleem was echter dat de performantie nog lager was dan het oorspronkelijke algoritme. Daarom werd het Tree Adaptive A* algoritme ingevoerd maar hoewel de performantie wel een heel stuk beter was, was het nog steeds niet te vergelijken met de performantie van het oorspronkelijke algoritme (voornamelijk door de morfologische operaties). Een ander nadeel van het vereenvoudigde algoritme was dat nu helemaal niet meer in de hoogte kon worden gebouwd omdat dit de fout van dit algoritme enkel groter maakt.

In het volgende hoofdstuk gaan we op zoek naar een methode waarmee ook legoconstructies met meerdere niveau's gedetecteerd kunnen worden. Om dit te verwezenlijken proberen we alle geometrische informatie te gebruiken die een legoblok bevat (niet slechts enkele eigenschappen zoals dit algoritme deed).

Hoofdstuk 4

Legoblokdetectie op basis van CAD modellen

In het voorgaande hoofdstuk werd een naïeve methode aangebracht om legoblokken te detecteren. Deze methode worstelde echter met verschillende nadelen waardoor het (bijna) niet mogelijk was om constructies van legoblokken te detecteren die bestaan uit meerdere niveau's. Om dit soort constructies wel te kunnen detecteren moeten we op zoek naar een meer generiek algoritme dat op basis van alle geometrische informatie een legoblok kan detecteren (zodanig dat de blok kan gevonden worden in eender welke legoconstructie). Daarom wordt in dit hoofdstuk een algoritme behandeld dat op basis van features geëxtraheerd uit CAD modellen legoblokken kan detecteren in een videoframe.

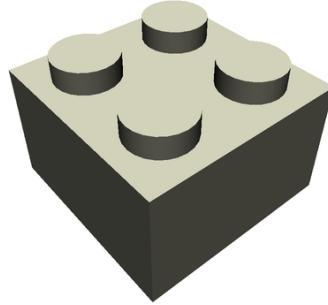
Eerst leggen we enkele begrippen uit en schetsen we kort het algoritme in sectie 4.1. Vervolgens bespreken we verschillende soorten features voor dit algoritme in sectie 4.2 en vergelijken we ze met elkaar in sectie 4.3.1. Ook enkele classificatie methodes voor dit algoritme komen aan bod en worden vergeleken respectievelijk in secties 4.4 en 4.5.1. Tenslotte geven we in sectie 4.6 aan waarom deze methode niet kan worden gebruikt in een AR spel om legoblokken te detecteren.

4.1 Begrippen en algoritme

4.1.1 Begrippen

Computer Aided Design (CAD) modellen zijn collecties van punten die de geometrie van een 3 dimensionaal object beschrijven, meestal worden ze gebruikt om objecten in een virtuele 3D wereld voor te stellen. Afbeelding 4.1 toont een afbeelding van het CAD model van een legoblok dat later in dit hoofdstuk nog wordt gebruikt.

Window sliding is een methode waarbij een rechthoek (het window) geschoven wordt over een afbeelding. Elke keer de rechthoek zich op een nieuwe plaats op de afbeelding bevindt wordt meestal een andere afbeelding vergeleken met het gedeelte van de afbeelding in het window. Deze methode wordt onder andere gebruikt bij object detectie om een object te zoeken in een afbeelding. In dit geval wordt dan,



FIGUUR 4.1: Pose van het CAD model van een 2x2 legoblokje.

steeds wanneer de rechthoek verschuift, bekeken of het te detecteren object zich in dit window bevindt. Omdat er tijdens window sliding echter geen rekening wordt gehouden met de schaal van het object doen we dit meermaals en wordt de schaal van het window telkens aangepast.

Een **classifier** is een functie die bij object detectie gebruikt wordt om tijdens window sliding te beslissen of het window het object bevat. In ons geval is de taak van de classifier dus om tijdens het window sliding te beslissen of het window een legoblok bevat.

AdaBoost is een algoritme dat vaak wordt gebruikt in object detectie algoritmes om classifiers sterker te maken tijdens de training. Het idee is dat enkel die zwakke classifiers (die op basis van 1 sample classificeren) worden gekozen die de kleinste trainingsfout maken en deze te combineren tot één sterke classifier [FS95]. Hier wordt het bij de Haar-like features (zie sectie 4.2.1) ook gebruikt om features te selecteren.

Een **integral image** is een afbeelding die gegenereerd wordt van een oorspronkelijke afbeelding door elke pixel vervangen met de som van alle pixels die in de linkerboven rechthoek liggen ten opzichte van de pixel [VJ01]. Dus indien de pixel in de linkerbovenhoek pixel $(0, 0)$ is, dan geldt voor elke pixel in de integral image:

$$p(x, y) = \sum_{x' \leq x, y' \leq y} p(x', y')$$

4.1.2 Algoritme

Een CAD model bevat alle geometrische informatie over een object (het moet dat object immers in 3D voorstellen) en dus kan dit model gebruikt worden om een object in een afbeelding te detecteren. Het grote voordeel is dan dat aangezien dit model alle geometrische informatie bevat het object in om het even welke pose kan worden gedetecteerd. Dit is de methode die werd aangebracht in [AME⁺14].

Om CAD modellen te kunnen gebruiken voor object detectie moet eerst nuttig informatie uit deze modellen worden gehaald. In een afbeelding is echter slechts een deel van het object te zien (de afbeelding is namelijk 2D), daarom is het beter eerst afbeeldingen te genereren van de verschillende poses van het CAD model (figuur 4.1 toont een voorbeeld).

Deze poses bevatten niet het exacte object dat we zoeken in de testafbeelding (het heeft misschien een andere kleur, andere belichting, andere achtergrond, etc.). Daarom moet nog informatie (in vorm van een classifier) geëxtraheerd worden uit deze afbeeldingen dat we vervolgens kunnen gebruiken voor de detectie. Dit gebeurt tijdens een trainingsfase.

Training

De training bestaat uit het berekenen van een classifier uit een aantal parameters en een hoop positieve en negatieve samples. We hebben ervoor gezorgd dat een deel van de negatieve samples bestaan uit het grondvlak zonder legoblokjes. Hiermee zijn de negatieve samples representatief voor de gebieden waarvan we willen dat ze niet als legoblok worden aanzien (de achtergrond).

De positieve samples zijn opgebouwd uit een set positieve afbeeldingen. Deze set bestaat enerzijds uit de virtueel gegenereerde poses van een 2x2 legoblok en anderzijds uit foto's van zulke legoblok, beide op een witte achtergrond (we gebruiken tijdens detectie immers ook een witte achtergrond). De poses werden gegenereerd van slechts een kwart van de legoblok omdat deze symmetrisch is. De foto's zijn nuttig omdat ze helpen de invloed van belichting op de classifier te minimaliseren aangezien ze werden getrokken in een realistische belichting.

Tijdens de training wordt een classifier berekend die zo goed mogelijk de positieve samples scheidt van de negatieve samples. Om de verschillende samples met elkaar te kunnen vergelijken worden eerst features berekend van alle samples die we vervolgens kunnen vergelijken met elkaar. Aangezien dit proces afhangt van welke classifier methode wordt gekozen, bespreken we dit uitgebreider in sectie 4.4.

Detectie

Na de training kan met behulp van de gegenereerde classifier en window sliding de legoblok gedetecteerd worden in een testafbeelding. Hiervoor wordt voor elke window de feature vector berekend die vervolgens aan de classifier wordt gegeven. Deze beslist dan of dit window het object bevat of niet.

Omdat de keuze van classifier methode en feature type erg belangrijk is voor het welslagen van dit algoritme worden de verschillende feature types en classifier methodes met elkaar vergeleken in de rest van dit hoofdstuk. De experimenten gebeuren allemaal op een desktop en niet op de smartphone waarop de uiteindelijke legobrick detectie moet worden geïmplementeerd. Dit is vooral omdat een desktop eenvoudiger is om op te experimenteren en omdat het voldoende is om het beste feature type en de beste classificatie methode te bepalen.

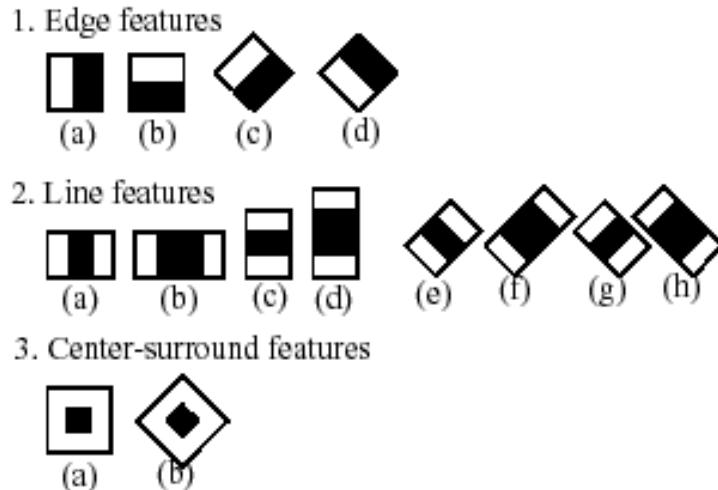
4.2 Features

In deze sectie worden drie verschillende soorten features besproken en met elkaar vergeleken: Haar-like [VJ01], Multiscale Block Local Binary Patterns [LZL⁺07] (MB-LBP) en Histogram of Oriented Gradients [DT05] (HOG). Tenslotte wordt ook een parts-based feature besproken (gebaseerd op HOG), dat gebruikt werd in [AME⁺14], en waarom deze niet is opgenomen in de evaluatie [FGMR10].

4.2.1 Haar-like

Bij de berekening van een Haar-like feature vector wordt het verschil genomen van de som van pixels tussen verschillende rechthoekige regio's. Oorspronkelijk werden vier soorten features gebruikt: twee die elk bestonden uit twee rechthoekige gebieden, één soort met drie rechthoekige gebieden en één soort met vier rechthoekige gebieden [VJ01]. Later werden extra features toegevoegd tot de uiteindelijke featureset 14 soorten features bedraagt [LM02]. De uitgebreide featureset is te zien in figuur 4.2: het verschil wordt bepaald door de som te nemen van pixels in de witte regio's en af te trekken van de som van pixels in de zwarte regio's. Alle soorten features zijn eenvoudig uit te rekenen aan de hand van een *integral image* (of een afgeleide daarvan [LM02]).

Om goede features te vinden wordt eerst een enorme hoeveelheid aan features aangemaakt. Aangezien dit aantal features een heel stuk hoger is dan het aantal pixels in het window moet een efficiënte methode gebruikt worden om de juiste features te selecteren. Hiervoor wordt AdaBoost gebruikt, een methode die origineel wordt gebruikt om een classifier sterker te maken. Aangezien AdaBoost het feature selecteert dat de kleinste fout maakt op de classificatie van positieve en negatieve samples, impliceert dit ook dat elke feature een threshold bevat die aangeeft wanneer



FIGUUR 4.2: Uitgebreide Haar-like featureset [LM02].

deze feature een detectiewindow als juist of onjuist classificeert. De uiteindelijke feature vector van het volledige detectiewindow is de set van de geselecteerde features.

4.2.2 MB-LBP

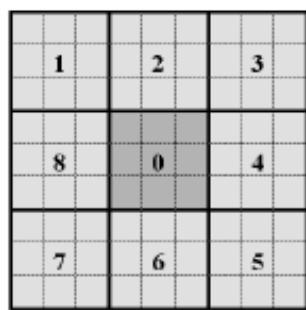
Een MB-LBP feature is een uitbreiding op de originele LBP feature waarin elke pixel werd vergeleken met zijn acht buren. Deze acht buren werden steeds in dezelfde volgorde overlopen en telkens een buur een grotere intensiteit had dan de pixel zelf werd een '1' genoteerd, in de andere gevallen een '0'. Zo wordt voor elke pixel een getal verkregen van acht bits dat de gradiënt van de pixel voorstelt in de verschillende richtingen [OPH96].

Hier wordt de MB-LBP feature gebruikt in plaats van de oorspronkelijke LBP omdat deze meer robuust is. In de MB-LBP variant worden, in plaats van een pixel met zijn buren te vergelijken, blokken van pixels met hun buren vergeleken (zie figuur 4.3). Hierdoor is de feature meer robuust en encodeert het bovendien, naast microstructuren, ook macrostructuren in een afbeelding [LZL⁺07].

Om de feature vector van een volledig window te berekenen wordt het window in cellen opgesplitst en vervolgens histogrammen berekend van de MB-LBP features binnen een cel. De histogrammen van alle cellen worden ten slotte aaneengeschakeld om de feature vector te bekomen.

4.2.3 HOG

Bij een HOG feature worden eerst gradiënten van alle pixels in een window berekend. Vervolgens wordt het window onderverdeeld in cellen en wordt voor elke cel een histogram van de gradiënten gemaakt. Om problemen met belichting en contrast te voorkomen worden cellen gecombineerd tot blokken waarover deze histogrammen worden genormaliseerd. Ten slotte worden alle histogrammen aaneengeschakeld (net als bij MB-LBP) om een feature vector van het window te bekomen [DT05].



FIGUUR 4.3: Een 9x9 MB-LBP operator: elk blok bestaat uit 9 pixels [LZL⁺07].

4.2.4 Deformed parts-based

Het deformed parts-based feature bestaat uit twee onderdelen: een HOG feature van de volledige window (root filter) en aantal kleinere HOG features van een subwindow (part filters). Deze verschillende part filters worden gedefinieerd door een anker punt ten opzichte van de plaats van de root filter en een functie die alle mogelijke plaatsen van deze part filter beschrijft relatief ten opzichte van het anker punt. De score van de hypothese dat een window het te detecteren object bevat wordt berekend door de sommatie te nemen van de verschillen (voor elke filter) tussen de score van een filter (HOG feature) en een kost voor de vervorming van de filter. Dit model maakt het HOG feature een stuk meer flexibel voor vervormingen of andere poses van een object [FGMR10].

Een groot nadeel van deze methode is dat ze nogal wat rekenwerk vraagt om uit te voeren en dus ongeschikt is voor gebruik in real time. Dit probleem is eerder al aangehaald in andere state-of-the-art papers die de performantie hebben kunnen verhogen. Hoewel bijvoorbeeld in paper [YLWL14] wordt aangegeven dat 40 FPS haalbaar is, is dit slechts na parallelisatie (over zes threads) en bovendien op een desktop computer. De desktop die in deze recente paper werd gebruikt heeft een 2.66GHz Hexacore Intel X5650 CPU. Dit is een heel stuk krachtiger dan de smartphone (met een 1GHz Quadcore Qualcomm Snapdragon S4 Pro APQ8064) die wij uiteindelijk willen gebruiken om het volledige algoritme op te implementeren. Het zou dus erg moeilijk zijn en bovendien veel tijd vragen om dit optimaal te implementeren op de smartphone. Dit verklaart waarom dit feature type voor ons geen interessante piste was en waarom deze niet is opgenomen in onze feature evaluatie.

4.3 Evaluatie features

Deze sectie behandelt de evaluatie van de feature types. Eerst wordt kort besproken hoe de evaluatie verloopt en vervolgens worden alle soorten features vergeleken op vlak van robuustheid en performantie.

4.3.1 Evaluatiemethode

De evaluatie van de verschillende features gebeurt door eerst een classifier te trainen voor alle features en vervolgens de geleerde classifiers te gebruiken voor detectie.

Training

Bij de training werden de features steeds getraind tot een cascade classifier (zie sectie 4.4.1), dit om geen invloed op de resultaten te krijgen door een andere keuze in classificatie methode. Om de training uit te voeren werd gebruik gemaakt van het programma `opencv_traincascade`, dat deel is van de OpenCv bibliotheek. Er werden ongeveer even veel positieve als negatieve samples gebruikt: de positieve samples begon met een set van 128 afbeeldingen die via transformaties op een

Parameter	Waarde
Training	
# classif. in cascade	20
Wind. size	64x64
Min. hit rate	0.999
Max. false alarm rate	0.5
Mode (Haar-like)	ALL
Detectie	
Schaalfactor	1.1
Min. # buren	15
Min. grootte	10x10 (px)
Max. grootte	100x100 (px)

TABEL 4.1: Parameters gebruikt tijdens training en detectie van features met cascade classificatie als classificatie methode.

willekeurig gekozen negatieve achtergrond (uit de negatieve samples) werd vergroot tot 3000 samples. Deze transformaties helpen dus om een groter aantal positieve samples te bekomen maar ook om het effect van belichting en rotaties van legoblokken kleiner te maken.

Omdat de training erg lang kan duren werd deze maar eenmaal uitgevoerd per soort feature om toch een grote indicatie te geven van hoe lang zulke training duurt. De parameters die tijdens de training werden gebruikt zijn aangegeven in tabel 4.1.

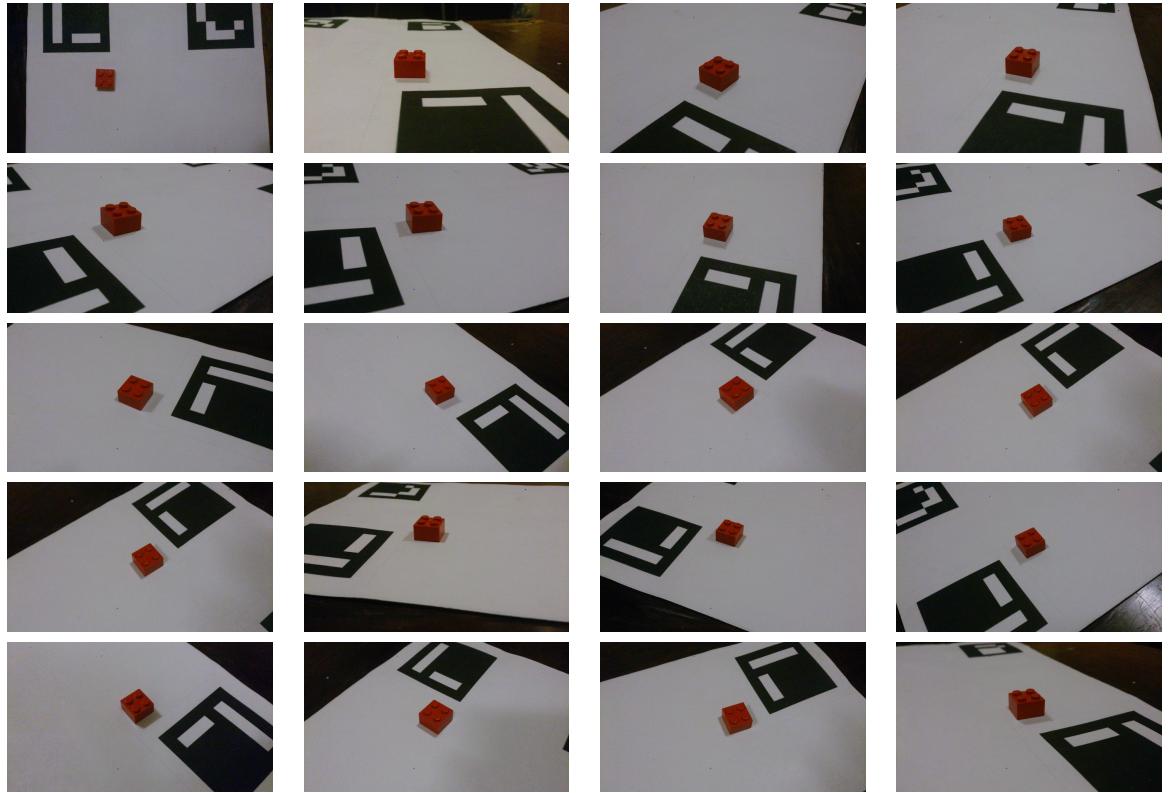
De training werd uitgevoerd op een machine met een 2.4 GHz CPU met 12 cores en 47GB RAM. Dit is niet dezelfde machine als gebruikt wordt tijdens de detectie omdat die machine veel te traag is om de training op uit te voeren. Bovendien is het niet nuttig om performantie van de training en detectie met elkaar te vergelijken omdat deze waarden ver uit elkaar liggen.

Detectie

De detectie gebeurt op 20 afbeeldingen van een legoblok in verschillende poses. Hierbij werden 3 standaardposes gekozen: bovenaanzicht, zijaanzicht en een aanzicht waarin de legoblok vanuit een hoekpunt wordt gezien. De rest van de poses zijn allemaal willekeurig gekozen. Deze afbeeldingen zijn te zien in figuur 4.4: de drie afbeeldingen in links in de eerste rij zijn de drie specifiek gekozen aanzichten, de rest zijn de willekeurige. Deze keuze van poses zal een idee geven over hoe robuust / performant de features zijn tegen wijzigingen in pose en welke poses robuuster / performanter zijn dan andere. De parameters die gebruikt werden tijdens de detectie zijn ook aangegeven in tabel 4.1.

Bij het vergelijken van features zal naar performantie en robuustheid worden gekeken. Voor de experimenten die focussen op performantie werd de detectie 10x uitgevoerd op alle 20 testafbeeldingen en per afbeelding het gemiddelde genomen.

4. LEGOLOKDETECTIE OP BASIS VAN CAD MODELLEN



FIGUUR 4.4: De 20 afbeeldingen die als testset werden gebruikt.

De experimenten werden uitgevoerd op een **machine** met een Intel Core 2 Duo 2,4 GHz processor, een NVIDIA GeForce 320M 256 MB grafische kaart, 8 Gb RAM en Mac OS X 10.10.2. Alle implementaties zijn geschreven in C++ en maken gebruik van OpenCV 2.4.9.

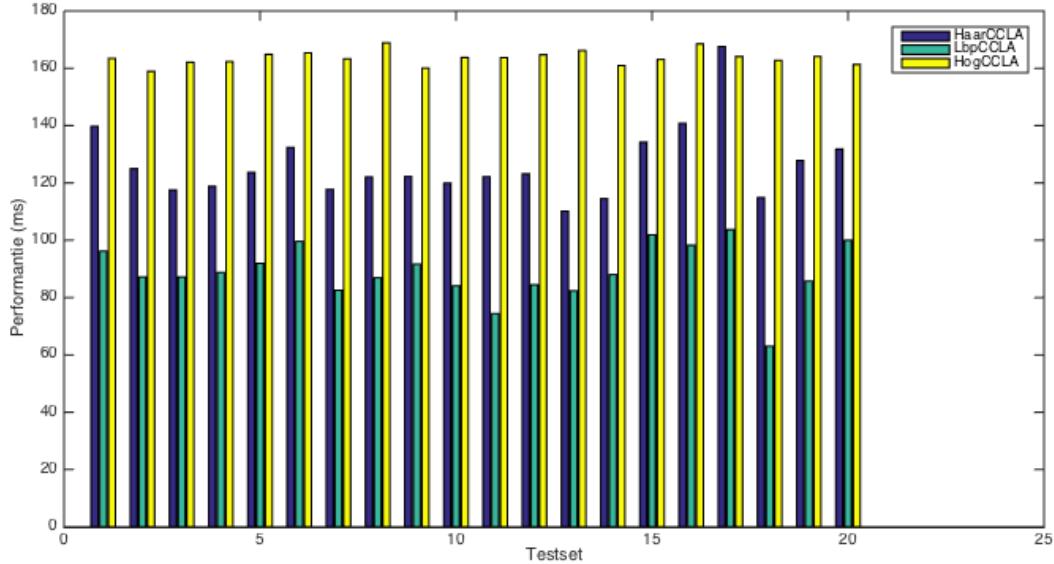
4.3.2 Vergelijking

In deze sectie worden de eerste drie besproken feature types met elkaar vergeleken. Ze worden vergeleken qua performantie en robuustheid.

Performantie

De vergelijking van performantie tussen verschillende soorten features bestaat uit twee onderdelen: enerzijds de snelheid bij training van de classifier en anderzijds de snelheid bij detectie.

De duur van de training wordt aangegeven in tabel 4.2. Het is belangrijk op te merken dat tijdens de HOG training slechts 1 core wordt gebruikt terwijl de andere soorten alle CPU cores gebruikten, dit is inherent aan de implementatie van OpenCV. We kunnen er wel uit afleiden dat HOG waarschijnlijk snelle kan worden getraind dan MB-LBP indien alle cores werden gebruikt. Het opmerkelijkste is het groot



FIGUUR 4.5: Feature performantie tijdens detectie.

verschil is tussen de tijd nodig om een Haar-like classifier te trainen en tijd nodig om een LBP of HOG classifier te trainen. Dit valt als volgt te verklaren: bij het trainen van een Haar-like classifier wordt eerst voor elke window een enorme hoeveelheid aan features gegenereerd die vervolgens worden uitgeselecteerd met AdaBoost [FS95]. Deze hoeveelheid aan features is vele malen groter dan het aantal pixels in het window, in tegenstelling tot MB-LBP en HOG waarbij voor elk window enkel een aantal histogrammen worden opgesteld.

De performantie tijdens detectie van de 20 testafbeeldingen wordt weergegeven in figuur 4.5. Het is duidelijk dat MB-LBP de snelste is, dit komt omdat bij MB-LBP integerwaarden worden vergeleken terwijl bij de rest floating point getallen worden vergeleken. Haar-like is bovendien sneller dan HOG, wat valt te verklaren door het feit dat aan Haar-like een intensieve training vooraf gaat waarna specifieke features zijn geselecteerd die eenvoudig te berekenen zijn via een *integral image*. Bij HOG moet tijdens de detectie voor elk window een volledig histogram worden berekend wat rekenintensiever is dan het uitrekenen een beperkte set van eenvoudig te berekenen features.

Feature	Training	# cores
Haar-like	> 4 dagen	12
MB-LBP	$\pm 2\text{h}$	12
HOG	$\pm 3\text{h}$	1

TABEL 4.2: Feature training tijdsduur.

Robuustheid

Uit de resultaten van figuur 4.6 kan worden afgeleid dat detectie met behulp van HOG de beste resultaten geeft: geen enkele afbeelding uit de testset was vals positief en op 4 afbeeldingen uit de testset na werd de legoblok overal correct gevonden. Hoewel met MB-LBP alle legoblokken werden gevonden, vertoont het heel wat meer valse positieven. Haar-like ligt tussen beide in: het detecteert één legoblok meer als HOG maar heeft wel twee valse positieven, terwijl HOG er geen enkele heeft.

4.3.3 Besluit

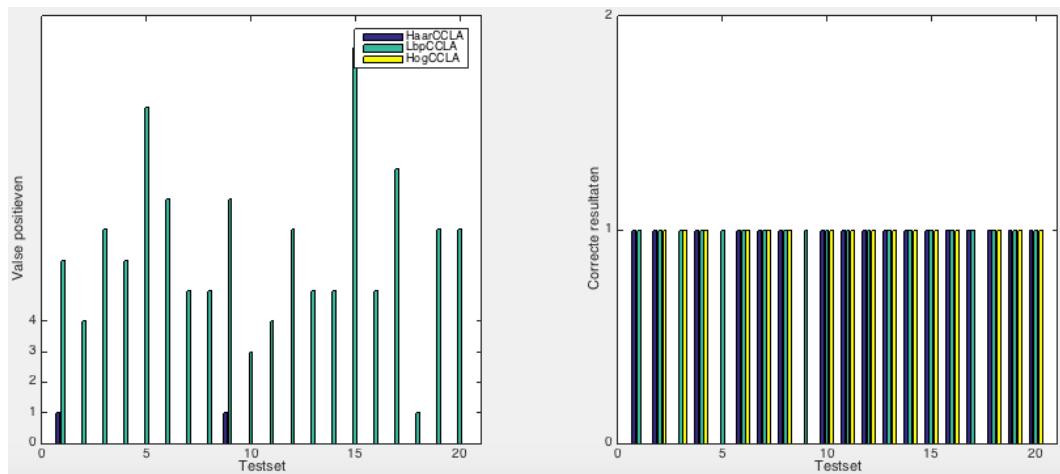
We kunnen besluiten dat HOG het meest robuuste feature type is maar dat MB-LBP een stuk sneller. Ook is belangrijk dat hoewel Haar-like features redelijk robuust en performant zijn, er erg veel tijd in kruipt om ze trainen.

4.4 Classificatie methodes

Deze sectie behandelt de classificatie methodes. De taak van de classificatie methodes is om op basis van berekende features een legoblok te detecteren. De volgende classificatie methodes besproken: Cascade Classifiers [VJ01] (CCLA) en Support Vector Machines [Joa99] (SVM).

4.4.1 CCLA

Bij een **CCLA** wordt een classifier getraind door een aantal feature classifiers te combineren. Elk van deze classifiers bestaat uit een aantal features op basis waarvan elke subwindow van een frame wordt geclasseerd. Zulke cascade classifier werkt door eerst elke subwindow van de frame te laten evalueren door de eerste classifier, indien de eerste classifier de subwindow aanvaardt triggert dit een evaluatie door de



FIGUUR 4.6: Feature robuustheid.

volgende classifier in de cascade. Enkel wanneer een window door alle classifiers is aanvaard, kan er vanuit worden gegaan dat dit window het te detecteren object bevat. Door een cascade van classifiers te gebruiken moet elke classifier van de cascade niet al te sterk zijn waardoor negatieve windows al sneller kunnen worden afgevoerd, wat de detectie des te sneller maakt.

4.4.2 SVM

Een **SVM** classificeert op een andere manier: elk subwindow wordt voorgesteld als een p-dimensionale vector in een p-dimensionale ruimte. Met behulp van (p-1)-dimensionale hyperplane worden de subwindows gesplitst in twee groepen: de ene groep wordt aanvaard, de andere niet. De *hit threshold* parameter bepaalt hoe ver een vector van de hyperplane moet liggen om te worden aanvaard of afgekeurd. Deze lineaire classifier was het oorspronkelijke algoritme [Vap63], later werd ook een methode ontwikkelt voor non-lineaire classificatie [BGV92].

4.5 Evaluatie classifier methodes

Deze sectie behandelt de evaluatie van de classifier methodes. Eerst wordt kort besproken hoe de evaluatie verloopt, vervolgens worden beide methodes (CCLA en SVM) vergeleken op vlak van robuustheid en performantie. Ten slotte wordt de impact bekeken van de *schaalfactor* parameter op robuustheid en performantie voor zowel CCLA als SVM.

4.5.1 Evaluatiemethode

De evaluatie van de verschillende classifier methodes gebeurt door elke soort classifier te trainen voor één bepaalde feature en vervolgens de geleerde classifiers te gebruiken voor detectie.

Training

Bij de vergelijking tussen de classificatie methodes werd steeds voor de HOG features (zie sectie 4.2.3) gekozen om geen invloed op de resultaten te krijgen door een andere keuze in features. Om de training uit te voeren werd gebruik gemaakt van het programma `opencv_traincascade` (van de OpenCv bibliotheek) om een CCLAClassifier te trainen en van de SVMLight bibliotheek [Joa99] om een SVM classifier te trainen. Als positieve samples werd de oorspronkelijke set van 128 afbeeldingen gebruikt en als negatieve samples werden dezelfde afbeeldingen gebruikt als bij de features (maar ze werden wel opgesplitst in tiles van 64x64 omdat de implementatie van SVMLight dat vereist).

Opnieuw werd training maar eenmaal uitgevoerd per classifier methode om een grofke indicatie te geven van hoe lang zulke training duurt. De parameters die tijdens de training werden gebruikt zijn aangegeven in tabellen 4.1 en 4.3 voor CCLA en SVM respectievelijk.

4. LEGOLOKDETECTIE OP BASIS VAN CAD MODELLEN

Parameter	Waarde
Training	
Padding (HOG)	0x0 (px)
Wind. stride (HOG)	8x8 (px)
Wind. size (HOG)	64x64 (px)
Detectie	
Schaalfactor	1.1
Hit threshold	0.4
Padding (HOG)	0x0 (px)
Wind. stride (HOG)	8x8 (px)
Wind. size (HOG)	64x64 (px)

TABEL 4.3: Parameters gebruikt tijdens training en detectie van features met SVM als classificatie methode.

De training werd opnieuw uitgevoerd op een machine met een 2.4 GHz CPU met 12 cores en 47GB RAM, vanwege dezelfde reden als bij de feature evaluatie.

Detectie

De detectie gebeurt, net als bij features, op 20 afbeeldingen van een legoblok in verschillende poses. Deze afbeeldingen zijn te zien in figuur 4.4: de drie afbeeldingen in rechts in de eerste rij zijn de drie specifiek gekozen aanzichten, de rest zijn de willekeurige. De parameters die gebruikt werden tijdens de detectie zijn ook aangegeven in tabellen 4.1 en 4.3.

Bij het vergelijken van classifier methodes zal naar performantie, robuustheid en de impact van de schaalfactor op de performantie en robuustheid worden gekeken. Voor de experimenten die focussen op performantie werd de detectie 10x uitgevoerd op alle 20 testafbeeldingen en per afbeelding het gemiddelde genomen. In het geval van experimenten met de schaalfactor wordt bij de performantie het gemiddelde genomen over alle testafbeeldingen.

De experimenten werden, net als bij de features, uitgevoerd op een machine met een Intel Core 2 Duo 2,4 GHz processor, een NVIDIA GeForce 320M 256 MB grafische kaart, 8 Gb RAM en Mac OS X 10.10.2. Alle implementaties zijn geschreven in C++ en maken gebruik van OpenCV 2.4.9 (zowel training als detectie).

4.5.2 Evaluatie

Robuustheid

Figuur 4.7 toont de robuustheid van de twee classifier methodes. Merk op dat de grafiek met valse positieven is weggelaten omdat deze in beide gevallen overal nul was. We merken op dat de robuustheid van de SVM hoger ligt dan die van de CCLA. De oorzaak hiervan dat SVM alle features gebruikt om die lineair te classificeren,

terwijl CCLA slechts een subset van alle features gebruikt om een zo klein mogelijke fout te maken op de classificatie.

Performantie

De vergelijking van performantie tussen verschillende soorten classificatie methodes bestaat uit twee onderdelen: enerzijds de snelheid bij training van de classifier en anderzijds de snelheid bij detectie.

De duur van de training wordt aangegeven in tabel 4.2. Dit geeft aan dat een SVM classifier een heel stuk sneller kan getraind worden dan een CCLA classifier. Dit komt omdat de SVM classifier features moet uitrekenen en een hyperplane bepalen tussen de features. CCLA, daarentegen, moet een hele cascade van classifiers opbouwen waarbij telkens opnieuw goede features moeten worden gekozen, dit vraagt heel wat meer rekenwerk.

De performantie tijdens detectie van de 20 testafbeeldingen wordt weergegeven in figuur 4.5. Dat de detectie langer duurt bij een SVM is logisch aangezien het een lineaire classifier is war vergeleken kan worden met één enkele classifier uit een cascade. Om een voldoende goede detectie te behalen moet een SVM dus erg correct kunnen classificeren, dit vergt meer tijd aangezien een cascade juist is ontwikkeld om efficiënter te zijn dan een enkele classifier.

Schaalfactor

De schaalfactor is een parameter die bepaalt hoe sterk de schaal van het window wordt gewijzigd tussen een minimum en maximum grootte. Dus wanneer deze kleiner wordt zullen normaal gezien meer iteraties nodig zijn, terwijl een nauwkeurigere detectie plaatsvindt.

Om de impact van de schaalfactor te bepalen, wordt de performantie en robuustheid van de HOG feature opgemeten wanneer de schaalfactor 1.01, 1.05, 1.10, 1.15, ... , 4 bedraagt. Uit figuur 4.9 is het duidelijk dat aan de verwachtingen voldaan is: de performantie (rechts) is hoger (het aantal ms daalt) bij een hogere schaalfactor, terwijl het aantal correcte detecties (links) daalt. Het aantal valse positieven werd ook gemeten maar deze was voor beide classifiers bijna altijd 0.

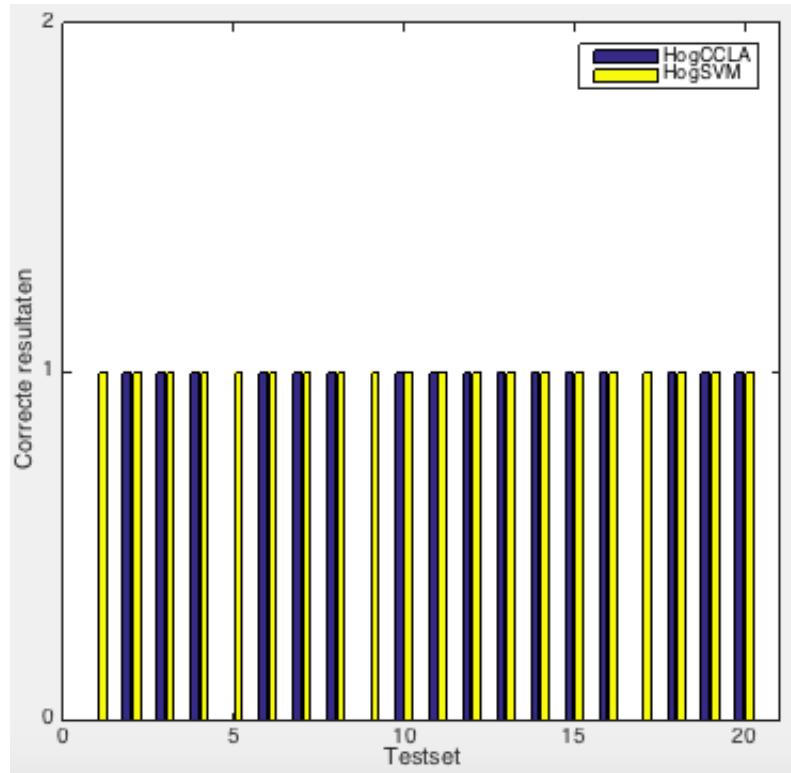
Bij de grafiek die het **aantal correcte detecties** toont zien we dat in het begin SVM robuuster is dan CCLA, daarna wisselt het wat af.

Dat het aantal correcte poses in het begin hoger ligt bij SVM dan bij CCLA terwijl zijn performantie over het algemeen hoger ligt kan op dezelfde manier verklaard worden als bij de performantie is geredeneerd. Vanwege het lineaire karakter van een

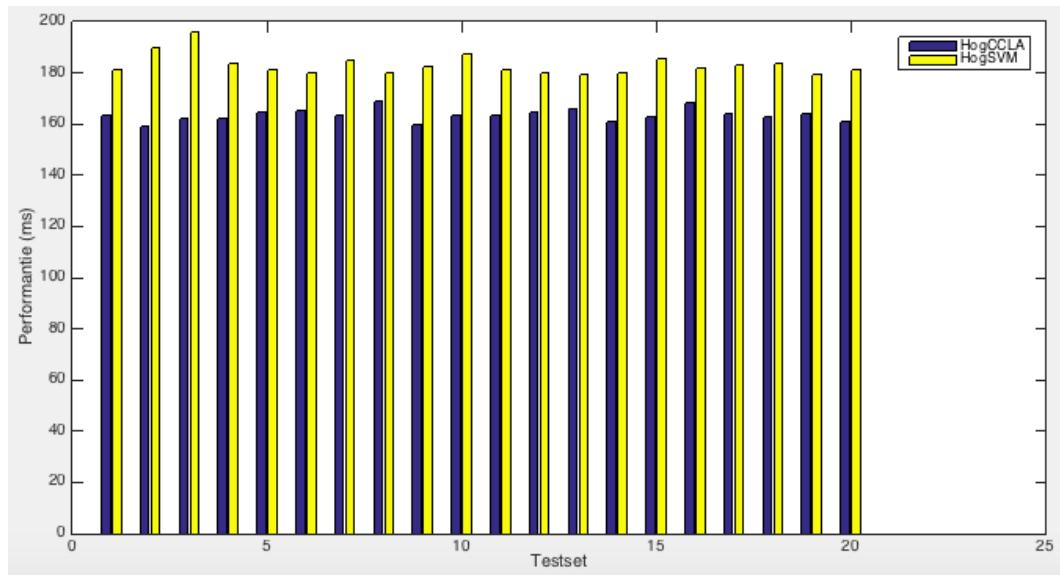
Feature	Training	# cores
CCLA	± 3h	1
SVM	± 40s	1

TABEL 4.4: Classifier methodes training tijdsduur.

4. LEGOBLOKDETECTIE OP BASIS VAN CAD MODELLEN



FIGUUR 4.7: Robuustheid classifier methodes.



FIGUUR 4.8: Classifier methodes performantie tijdens detectie.

SVM is het dan ook logisch dat wanneer een hogere robuustheid wordt gehaald (in het begin) de performantie lager ligt.

Het wisselend karakter van de eerste grafiek ligt aan de aard van de parameter: indien de schaalfactor toevallig een waarde heeft waardoor het window even groot wordt als een bepaalde legoblok zal deze worden gedetecteerd. Merk op dat deze eerste grafiek slechts tot 1.65 wordt getoond, dit komt omdat het aantal correcte detecties hierna toch steeds 0 is. Dat het reeds bij 1.65 altijd 0 is, komt omdat wanneer de schaalfactor te hoog wordt slechts enkele verschillende groottes als window worden gebruikt. Indien de groottes van deze windows niet toevallig dicht genoeg liggen bij de grootte van de legoblok zal deze niet gedetecteerd worden.

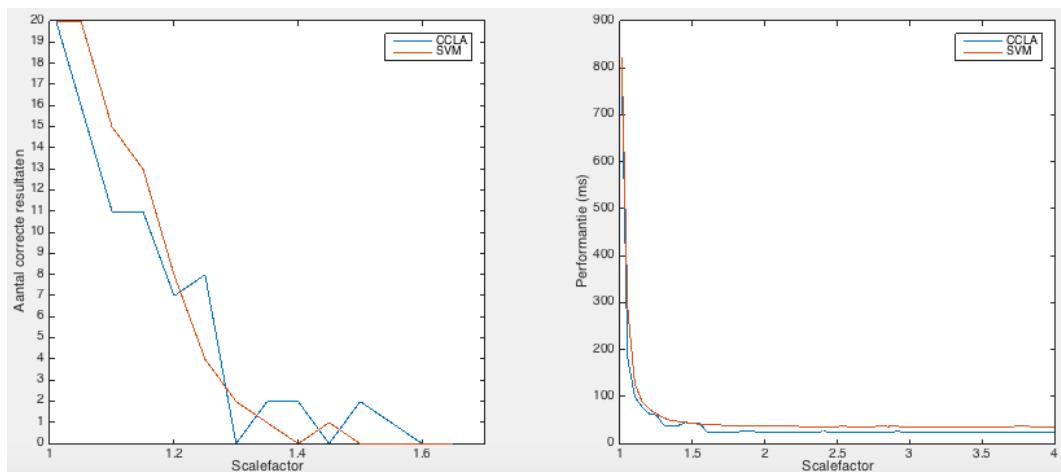
Opmerkelijk is de vorm van de grafiek die de **performantie** toont, het geeft aan dat beide classifier methodes een performantie hebben van de orde $O(1/(s - 1))$ met s de schaalfactor. Merk op dat de orde $1/(s - 1)$ is en niet $1/s$ omdat de verticale asymptoot moet liggen op $s = 1$ want als de schaalfactor 1 is duurt de detectie oneindig lang, het window wordt dan immers niet groter.

4.5.3 Besluit

In deze evaluatie werd aangetoond dat SVM een meer robuuste methode is dan CCLA, maar dat CCLA wel de meer performante is. Vervolgens werd aangetoond dat een hogere schaalfactor de performantie verhoogt maar de robuustheid verlaagt. Tenslotte werd duidelijk dat de performantie van beide methodes van orde $O(1/(s-1))$ is met s de schaalfactor.

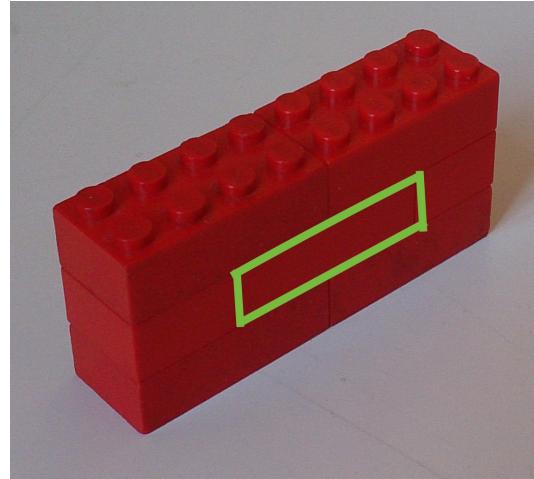
4.6 Discussie / Besluit

In dit hoofdstuk werd onderzocht welke feature types en classifier methodes beter werken voor detectie van legoblokken. Zo is aangetoond dat HOG de meer robuuste



FIGUUR 4.9: Impact schaalfactor op performantie en robuustheid van HOG feature.

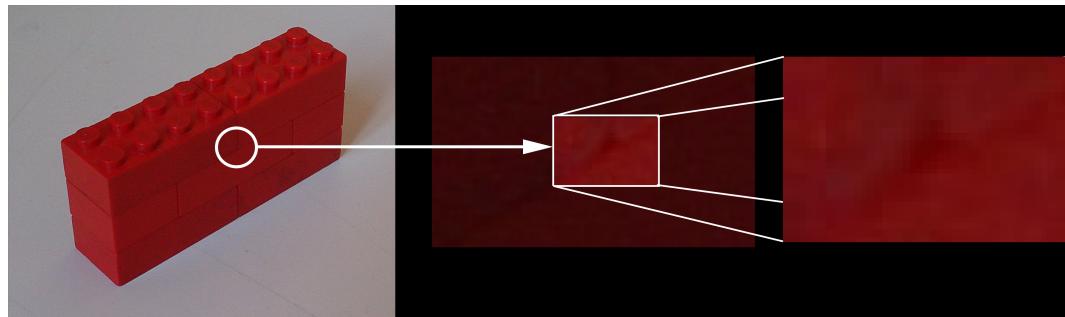
4. LEGOLOKDETECTIE OP BASIS VAN CAD MODELLEN



FIGUUR 4.10: Een voorbeeld van een legoblokconstructie waarin het groen omlijnde vlak het enige is wat we van die legoblok zien.

feature type is en dat MB-LBP de meest performante van de drie is. Verder werd aangegeven dat een vierde type feature (op basis van deformed parts-based modellen) te traag is om nuttig te gebruiken in een AR applicatie. Ten slotte toonde de vergelijking tussen classifier methodes aan dat detectie met een SVM classifier vaak trager is maar tegelijk ook meer robuust, dat een hogere schaalfactor leidt tot hogere performantie maar lagere robuustheid en dat de performantie van de classifier methodes van orde $1/(s-1)$ is met s de schaalfactor. De methode die in dit hoofdstuk werd onderzocht is echter niet nuttig om te gebruiken in een AR spel om legoblokken te detecteren en hier is waarom:

Hoewel het wel mogelijk is om een enkele legoblok te detecteren is het erg moeilijk om in een constructie van legoblokken de blokken afzonderlijk te detecteren. Dit resulteerde telkens in erg slechte resultaten, zelfs wanneer het beste type feature en de beste classificatie methode werd gekozen. Dit wordt hoofdzakelijk veroorzaakt door de enorme hoeveelheid aan occlusie die een legoblok in een constructie grotendeels



FIGUUR 4.11: De vergroting van een rand in een muur van legoblokken.

verbergt. Hierdoor is slechts een erg beperkt deel van de legoblok (in sommige gevallen slechts één vlak, zie figuur 4.10) zichtbaar wat het erg moeilijk maakt om met features een legoblok te detecteren. Zelfs indien op voorhand een optimale pose zou worden gekozen kan voor sommige legoblokken enkel één vlak zichtbaar zijn in de afbeelding.

Men zou dan kunnen zeggen dat op basis van de randen de verschillende legoblokken in de constructie kunnen worden gedetecteerd. Dit is echter niet waar omdat deze randen erg zwak zijn wanneer legoblokken met eenzelfde kleur naast elkaar staan. Indien we dit probleem van naderbij zouden bekijken bekomen we een resultaat zoals in figuur 4.11. In deze figuur is een deel van de afbeelding dat legoblokranden bevat tweemaal uitvergroot. Deze rand neemt slechts twee à drie pixels in beslag en verandert bovendien erg weinig van kleur. Dit maakt het enorm moeilijk, zo niet onmogelijk om de verschillende legoblokken op basis van hun randen te detecteren.

Deze redenering leidt tot een volgend resultaat: het is erg moeilijk, zo niet onmogelijk, om in een legoconstructie alle legoblokken apart te detecteren. Daarom moeten we proberen om, in plaats van de blokken individueel te proberen detecteren, ze als groter geheel te detecteren. Zulke legoconstructie kunnen we zien als een flexibele legoblok die eender welke hoogte, diepte en breedte kan hebben. Dat is exact waarvoor het deformed parts-based feature type gebruikt wordt maar zoals aangehaald in sectie 4.2.4 is deze methode te traag om te gebruiken in een mobiele AR applicatie. Dat is de reden waarom in de volgende sectie een sneller alternatief wordt voorgesteld dat gebruik maakt van het idee dat lego constructies moeten gedetecteerd worden in plaats van individuele legoblokken. Hierbij wordt echter wel afgestapt van het idee om CAD modellen te gebruiken.

Bijlagen

Bibliografie

- [AME⁺14] Mathieu Aubry, Daniel Maturana, Alexei A Efros, Bryan C Russell, and Josef Sivic. Seeing 3d chairs: exemplar part-based 2d-3d alignment using a large dataset of cad models. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 3762–3769. IEEE, 2014.
- [BGV92] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.
- [DP73] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.
- [DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.
- [FGMR10] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(9):1627–1645, 2010.
- [FS95] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.
- [HSKM11] Carlos Hernández, Xiaoxun Sun, Sven Koenig, and Pedro Meseguer. Tree adaptive a*. In *The 10th International Conference on Autonomous Agents and Multiagent Systems- Volume 1*, pages 123–130. International Foundation for Autonomous Agents and Multiagent Systems, 2011.
- [Its15] Itseez. OpenCV library, 2015. [Online; accessed 24-April-2015].

- [Joa99] Thorsten Joachims. Svmlight: Support vector machine. *SVM-Light Support Vector Machine* <http://svmlight.joachims.org/>, University of Dortmund, 19(4), 1999.
- [LM02] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, volume 1, pages I–900. IEEE, 2002.
- [LZL⁺07] Shengcai Liao, Xiangxin Zhu, Zhen Lei, Lun Zhang, and Stan Z Li. Learning multi-scale block local binary patterns for face recognition. In *Advances in Biometrics*, pages 828–837. Springer, 2007.
- [OPH96] Timo Ojala, Matti Pietikäinen, and David Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern recognition*, 29(1):51–59, 1996.
- [S⁺85] Satoshi Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985.
- [Vap63] Vladimir Vapnik. Pattern recognition using generalized portrait method. *Automation and remote control*, 24:774–780, 1963.
- [VJ01] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.
- [YLWL14] Junjie Yan, Zhen Lei, Longyin Wen, and Stan Z Li. The fastest deformable part model for object detection. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 2497–2504. IEEE, 2014.

Fiche masterproef

Student: Wouter Franken

Titel: Augmented Reality Bordspellen

Engelse titel: Augmented Reality Boardgames

UDC: 681.3

Korte inhoud:

Hier komt een heel bondig abstract van hooguit 500 woorden. LATEX commando's mogen hier gebruikt worden. Blanco lijnen (of het commando) zijn wel niet toegelaten!

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Thesis voorgelegd tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Mens-machine communicatie

Promotor: Prof. dr. ir. Philip Dutre

Assessor: TODO

Begeleider: Ir. J. Baert