

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	14
3.5 Uitbreiding: Tree Adaptive A*	17
3.6 Besluit	19
4 Legoblokdetectie op basis van CAD modellen	19
4.1 Begrippen en algoritme	19
4.2 Features	22
4.3 Evaluatie features	24
4.4 Classificatie methodes	28
4.5 Evaluatie classifier methodes	29
4.6 Discussie / Besluit	33
5 Het laatste hoofdstuk	11
5.1 Eerste onderwerp in dit hoofdstuk	11
5.2 Tweede onderwerp in dit hoofdstuk	11
5.3 Besluit van dit hoofdstuk	11
5 Besluit	11
A De eerste bijlage	15

INHOUDSOPGAVE

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	Simple (links) en complexe scene (rechts) voor performantie evaluatie van naïeve blokdetectie.	11
3.6	Gemiddelde performantie van het eerste naïeve algoritme.	12
3.7	Ruis in threshold vóór morfologische operaties werden toegepast.	15
3.8	Fout van het vereenvoudigde naïeve algoritme in de vorm van schaduw.	15
3.9	Resultaat van het tweede naïeve algoritme.	16
3.10	Gemiddelde performantie van de vereenvoudiging van het naïeve algoritme.	17
3.11	Voorbeeld van het Tree Adaptive A* algoritme (uit [HSKM11]).	18
3.12	Gemiddelde performantie van de uitbreiding op de vereenvoudiging van het naïeve algoritme.	19
4.1	Pose van het CAD model van een 2x2 legoblokje.	20
4.2	Uitgebreide Haar-like featureset [LM02].	22
4.3	Een 9x9 MB-LBP operator: elk blok bestaat uit 9 pixels [LZL ⁺ 07].	23
4.4	De 20 afbeeldingen die als testset werden gebruikt.	26
4.5	Feature performantie tijdens detectie.	27
4.6	Feature robuustheid.	28
4.7	Robuustheid classifier methodes.	32
4.8	Classifier methodes performantie tijdens detectie.	32
4.9	Impact schaalfactor op performantie en robuustheid van HOG feature.	33
4.10	Een voorbeeld van een legoblokconstructie waarin het groen omlijnde vlak het enige is wat we van die legoblok zien.	34
4.11	De vergroting van een rand in een muur van legoblokken.	34

Lijst van tabellen

4.1	Parameters gebruikt tijdens training en detectie van features met cascade classificatie als classificatie methode.	25
4.2	Feature training tijdsduur.	27
4.3	Parameters gebruikt tijdens training en detectie van features met SVM als classificatie methode.	30
4.4	Classifier methodes training tijdsduur.	31

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 in sectie 3.2. Sectie ?? 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 ?? en we sluiten af met de redenen waarom in de volgende hoofdstukken op zoek wordt gegaan naar andere algoritmes in sectie ??

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). 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 LEGOBLOKDETECTIE

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 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 gebruikt 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.

3.2. Blokdetectie op basis van simpele geometrische informatie

Vervolgens wordt de pose van de blok bepaald door te steunen op de geometrische informatie. 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. Dit is een erg belangrijke stap vanwege twee redenen: ten eerste kan OpenCV niet werken met het YUV formaat en ten tweede is 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:

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

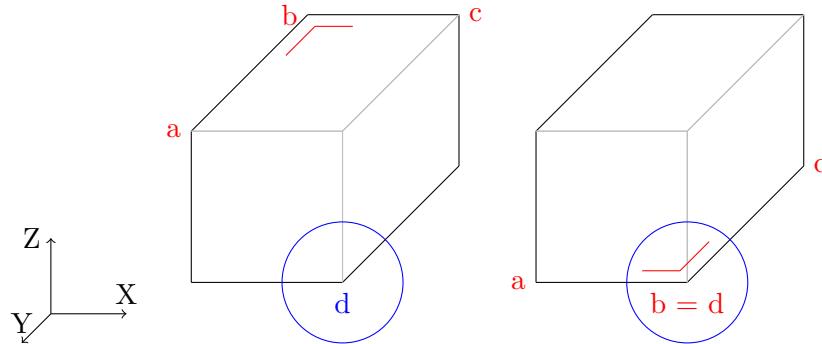
Contouren worden dan bepaald met behulp van het algoritme dat beschreven wordt in [S+85]. Dit algoritme is gebaseerd op een simpel algoritme dat de afbeelding scant naar contouren. Wanneer men zulke contour tegenkomt wordt deze aangeduid, bewaard en op zoek gegaan naar de volgende. Bovendien wordt een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

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, na deze benadering blijven de belangrijkste punten over: de hoekpunten van de legoblok. Dit 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 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 maar in ruil daarvoor moeten we de veronderstelling maken dat de contour van een legoblok bestaat uit maximaal zes hoeken. Dit resulteert in enkele nadelen die later aan bod zullen komen.

Nu kan het zijn dat we contours hebben 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 legoblokjes worden getoond met hun contourlijnen aangeduid in zwart.

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 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

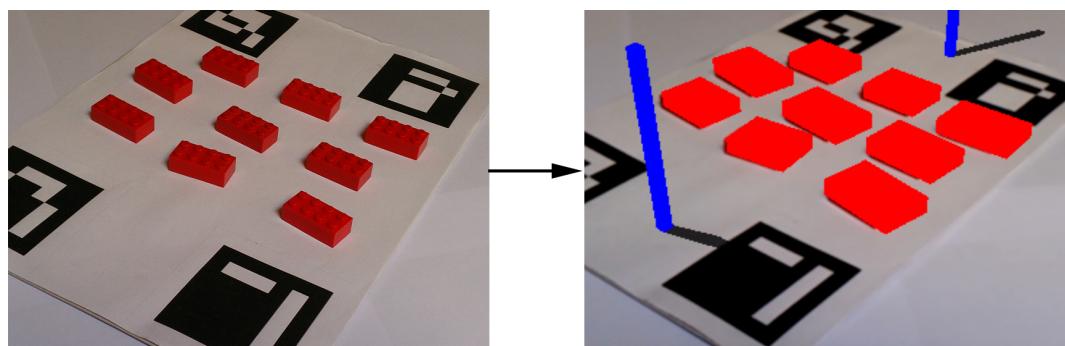
Figuur 3.2 toont het resultaat van het algoritme. De linkerafbeelding toont de legoblokken en de rechteraafbeelding wat de detectie van legoblokken oplevert.

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.

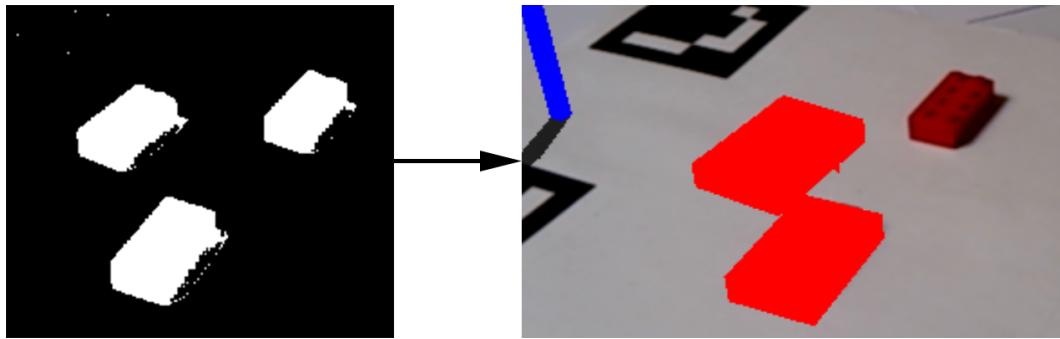
Verder valt ook op te merken dat de virtuele legoblokken iets groter zijn dan de werkelijke legoblokken. Dit is expliciet toegevoegd aan het algoritme omdat de virtuele wereld bestaat uit een discreet grid. Wanneer lemmings in dit discreet grid dan wandelen van node naar node kan het zijn dat ze deels 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.

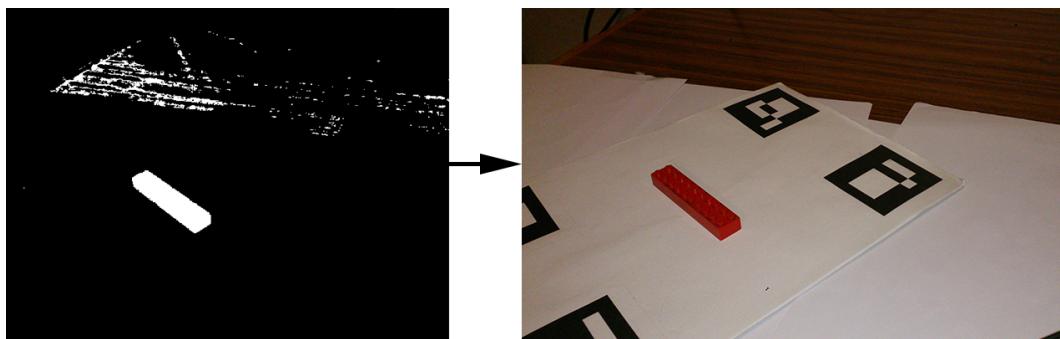
Omdat we kleurthresholding gebruiken is een rode tint in het beeld is absoluut uit den boze, elke kleur in het beeld dat een beetje in de grenzen van de threshold valt vormt een probleem. Dit tonen we aan in figuur 3.4, waarin een bruine achtergrond in beeld komt. We zien duidelijk dat veel ruis in de 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.

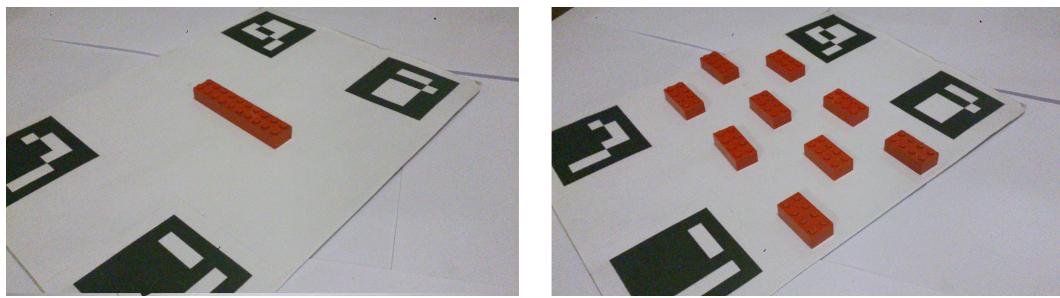


FIGUUR 3.4: Resultaat van het naïeve algoritme bij rode tint in het beeld.

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

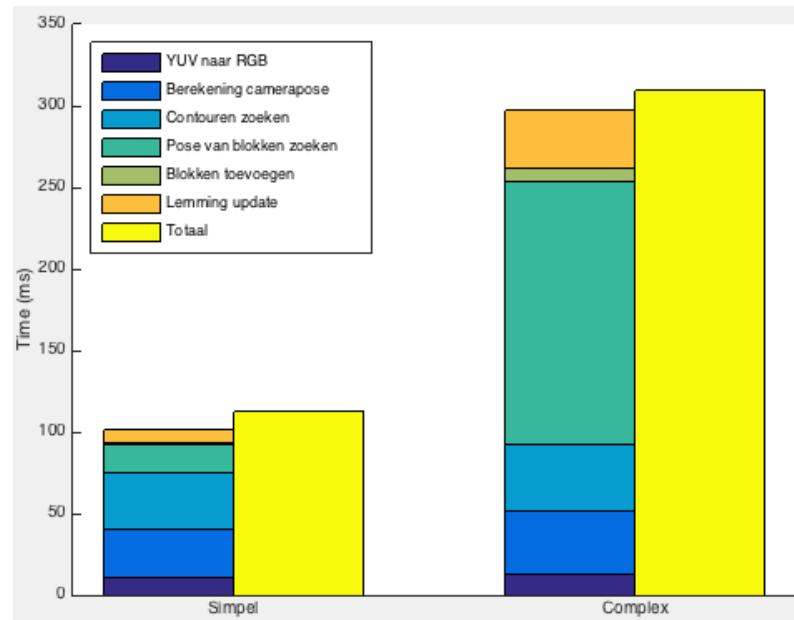
deze algoritmes, maw. het zijn scenes die beide algoritmes kunnen detecteren en respectievelijk weinig / veel legoblokken bevatten. Vervolgens werd per onderdeel het gemiddelde genomen van de gemeten tijd om geen grote uitschieters te hebben.

Evaluatie

Figuur 3.12 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 eerste 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.

Uit de figuur kunnen we vooral opmaken dat het gedeelte van het zoeken van de pose erg veel verschilt tussen de complexe en simpele scène. 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. 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.

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 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.

3. NAÏVE LEGOLOKDETECTIE

- 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 gevolgen 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:

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

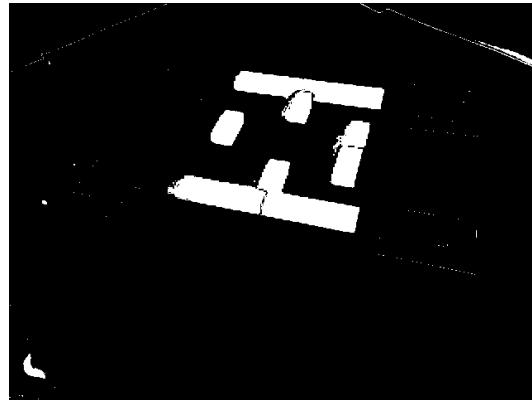
$$\text{Geel} : 135 < H < 160; 147 < S < 255; 30 < V < 255$$

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

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 3.7). 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.

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



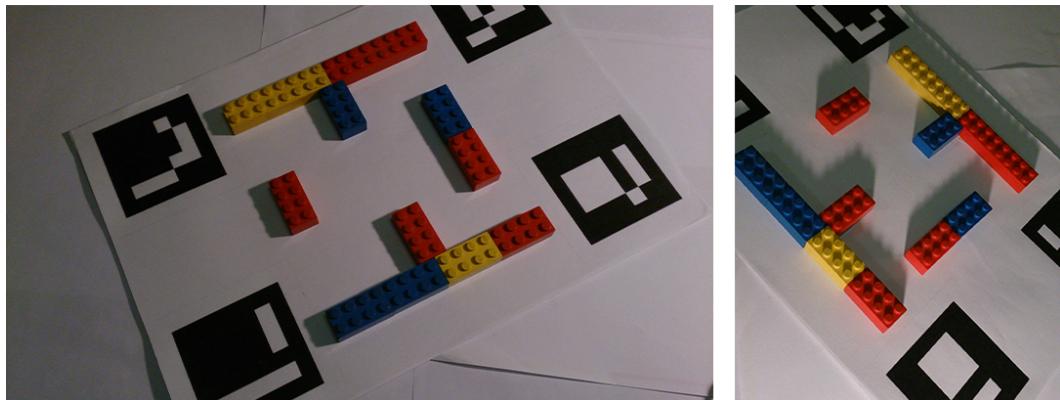
FIGUUR 3.7: Ruis in threshold vóór morfologische operaties werden toegepast.

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 3.8 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 3.8: Fout van het vereenvoudigde naïeve algoritme in de vorm van schaduw.

3. NAÏEVE LEGOLOKDETECTIE

het enkele nadelen uit het naïeve algoritme wegwerkt:

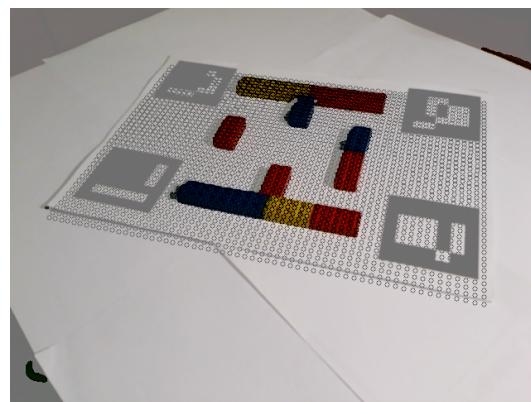
- Legoblokken in meerdere kleuren kunnen gebruikt worden, het gevolg is wel dat morfologische operaties de threshold moeten oppoetsen om er ruis uit te halen zie figuur 3.7. Dit werd hoogstwaarschijnlijk veroorzaakt door de belichting die niet altijd constant is, zodat de thresholdwaarden niet altijd 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 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 3.9 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.



FIGUUR 3.9: Resultaat van het tweede naïeve algoritme.

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

Performantie

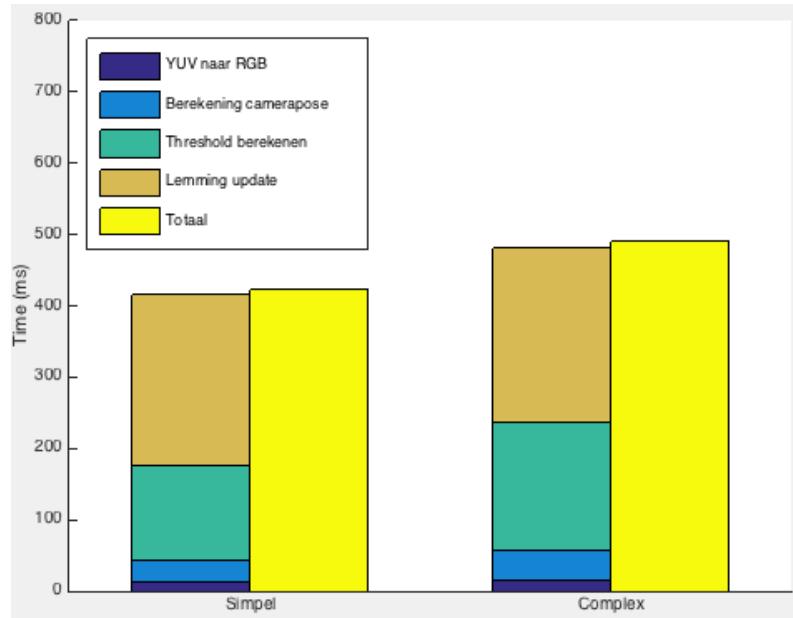
TODO: BESPREKING PERFORMANTIE

Figuur ?? toont de gemiddelde performantie van 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. 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 is opmerkelijk hoe weinig het verschil is tussen de eenvoudige en complexe scène.6

Uit de figuur kunnen we vooral opmaken dat het gedeelte van het zoeken van de pose erg veel verschilt tussen de complexe en simpele scène. 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



FIGUUR 3.10: Gemiddelde performantie van de vereenvoudiging van het naïeve algoritme.

3. NAÏVE LEGOLOKDETECTIE

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 scene het pad van een lemming van begin tot eindpunt een stukje langer is dan bij de simpele scene. 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 scene en 3.2 fps voor de complexe scene, wat natuurlijk erg traag is voor een AR spel.

3.4.4 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.

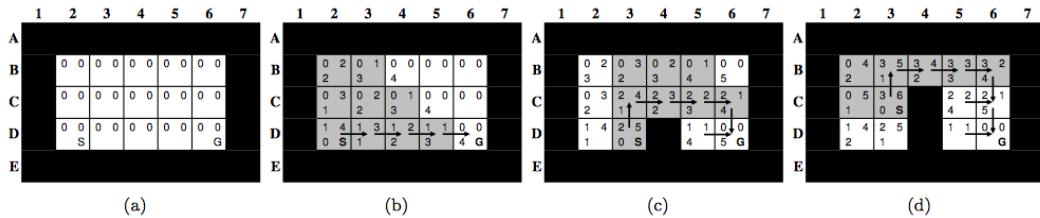
Om het performantieprobleem te verhelpen breiden we in de volgende sectie dit algoritme nog eenmaal uit. In deze uitbreiding vervangen we het A* algoritme met Tree Adaptive A* om paden te zoeken in de virtuele wereld. In deze uitbreiding zullen grote delen van reeds berekende paden herbruikbaar zijn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

3.5 Uitbreidings: 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



FIGUUR 3.11: Voorbeeld van het Tree Adaptive A* algoritme (uit [HSKM11]).

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 3.11. Alle grijze tegels zijn geëxpandeerde nodes en de zwarte tegels zijn de obstakels waar de agent weet van heeft. Figuur 3.11a toont het beginsituatie, figuur 3.11b 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 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.

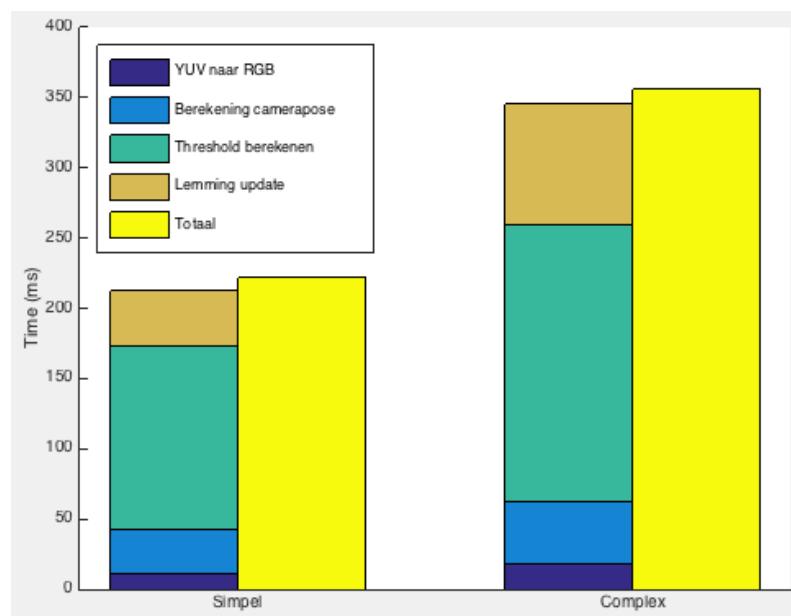
Het grote voordeel aan dit algoritme voor een AR spel is dat deze *herbruikbare boom* kan gedeeld worden tussen lemmingen. Dus als de wereld niet veranderd moeten weinig stukjes pad herberekend worden.

3.5.2 Performantie

TODO

3.6 Besluit

3. NAÏVE LEGOLOKDETECTIE



FIGUUR 3.12: Gemiddelde performantie van de uitbreiding op de vereenvoudiging van het naïeve algoritme.

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.
- [Joa99] Thorsten Joachims. Svmlight: Support vector machine. *SVM-Light Support Vector Machine <http://svmlight.joachims.org/>, University of Dortmund*, 19(4), 1999.

BIBLIOGRAFIE

- [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