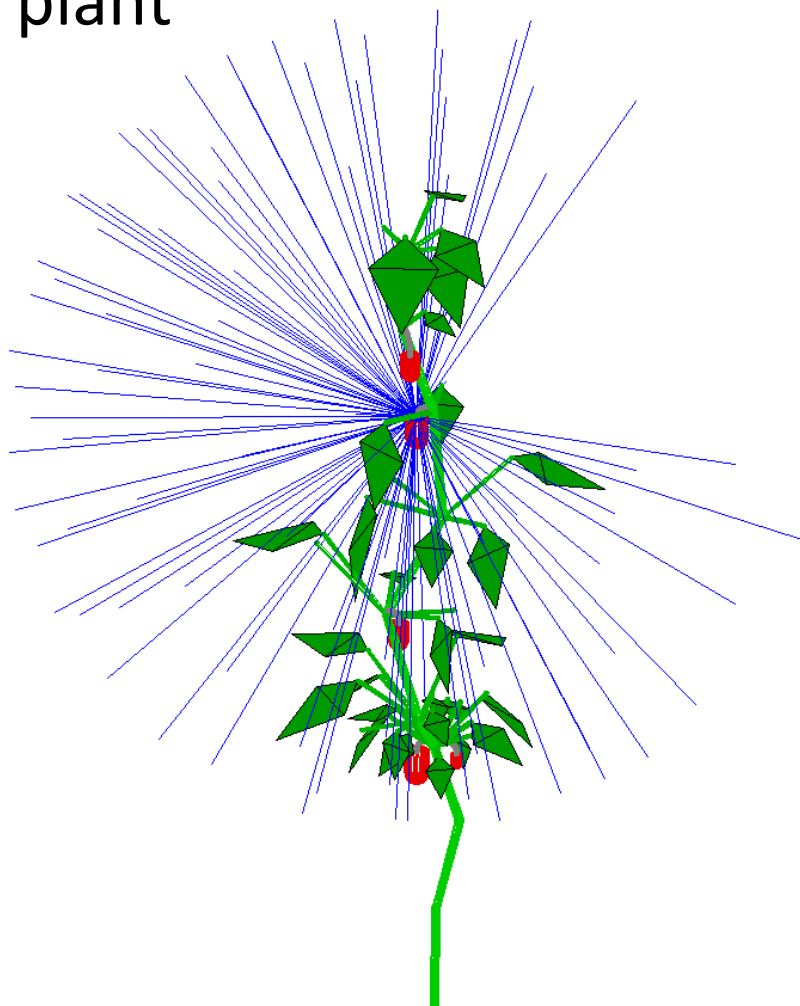


Bereikbaarheidsanalyse van paprika's aan een virtuele plant

Thijs de Swart
870411-819-030

April 2012



WAGENINGEN UNIVERSITY
WAGENINGENUR

CRO**P**S 
"Clever Robots for Crops"

Bereikbaarheidsanalyse van paprika's aan een virtuele plant

Vaknaam : Bachelorafsluiting Agrotechnologie
Vakcode : FTE - 80812
Studielast : 12 ECTS
Datum : 20-04-2012

Student : Thijs de Swart
Registratienummer : 870411-819-030
Studie : BSc Agrotechnologie (BAT)

Begeleider : Prof. dr. Ir. E.J. van Henten
Examinator : Prof. dr. Ir. P.W.G. Groot Koerkamp
Leerstoelgroep : Agrarische bedrijfstechnologie
Droevendaalsesteeg 1
6708 PB Wageningen
Tel. 0317- 48 29 80

Inhoud

Inleiding.....	1
Probleemstelling en afbakening.....	2
Probleemstelling.....	2
Afbakening.....	2
Onderzoeksvragen.....	3
Literatuurstudie vrije ruimte bepalen.....	4
Toepassingsgebieden.....	4
State of the art technieken	5
<i>Cameralocatie bepaling voor vormreconstructie.....</i>	<i>5</i>
<i>Sferische zichtbaarheidskaarten</i>	<i>6</i>
<i>Werktuigoriëntatie correctie</i>	<i>10</i>
Visibility cones	11
Conclusie.....	11
Vergelijkbaar onderzoek.....	12
Materialen en methode.....	13
Ontwerp plant.....	13
<i>3D tekening van een paprikaplant.....</i>	<i>13</i>
<i>Resultaat.....</i>	<i>15</i>
Programmeren plant model	16
<i>Het programma</i>	<i>16</i>
<i>Plant class.....</i>	<i>17</i>
<i>Graphical user interface.....</i>	<i>19</i>
Bepalen vrije zichtlijnen.....	21
<i>Gebruikte techniek.....</i>	<i>21</i>
<i>Implementatie</i>	<i>21</i>
Analyse.....	25
Resultaten.....	26
Een enkele plant	26
<i>De plant</i>	<i>26</i>
<i>De zichtlijnen</i>	<i>26</i>
Meerdere planten.....	30
Discussie	33
Conclusie	34
Aanbevelingen	34

Literatuur.....	35
Bijlages	
Bijlage 1, code 'Plant.m'	

Inleiding

De *Capsicum annuum* komt oorspronkelijk uit Zuid-Amerika. In de tijd van het Groot Ottomaanse Rijk hebben de Turken deze plant geïntroduceerd in Hongarije. In Hongarije is men aan de slag gegaan met het veredelen van deze plant waar uiteindelijk een zoete ronde variant uit voort kwam. Deze vrucht noemden de Hongaren de paprika. Vanuit Hongarije is de vrucht ook in Nederland terecht gekomen (MidFresh, z. d.). Inmiddels is Nederland de op een na grootste paprikaproducteur van Europa (AGF, 2010).

Er worden in de kassen zo veel paprika's verbouwd dat het plukken van deze vruchten een dagtaak is voor vele mensen in het oogstseizoen (Van Dijk, z. d.). Het inhuren van mensen wordt echter steeds duurder en daarom is men op zoek naar een alternatief (MNP, 2007). Een machinale plukker. Een plukker die niet moe wordt, geen pauze nodig heeft en geen salaris vraagt. Maar dit is geen simpele opdracht. Het plukken van een paprika is door een mens simpel uit te voeren, maar om een machine te maken die dit net zo goed kan is een hele klus.

Eerst moet deze machine naar de plant toe, vervolgens moet de paprika gevonden worden en vervolgens ook nog geplukt. En tijdens dit alles mag de plant niet beschadigen.

Probleemstelling en afbakening

Probleemstelling

Huidige situatie

De huidige situatie is dat er in een kas paprika's aan paprikaplanten hangen. Deze paprika's moeten geplukt worden om ze te kunnen verkopen. Dit plukken gebeurt handmatig omdat dit op het moment de enige optie is. Arbeid wordt echter steeds duurder, daarom is het tijd om te kijken of er ook andere mogelijkheden gevonden kunnen worden zoals automatisering (MNP, 2007). Daarbij komt nog dat in de economische crisis van 2009 de glasgroentebedrijven het zwaarst geraakt zijn van alle glastuinbouwbedrijven. De paprika heeft vervolgens de minste prijs herstel meegemaakt van de verschillende glasgroenten (LEI, 2011).

Een van de problemen van het automatisch plukken, is het vinden van de paprika. Waar begin je te zoeken en hoe kun je deze het meest efficiënt vinden? Wanneer een paprika gevonden is, moet deze geplukt worden zonder dat de plant wordt beschadigd door de plukker. Hoe wordt dit gerealiseerd?

Gewenste situatie

Vanwege de hoge kosten van arbeid zou de ideale situatie zijn dat alle paprika's geplukt kunnen worden zonder dat daar mensen aan te pas komen. Er moet hiervoor een systeem komen dat zichzelf autonoom door de kas kan verplaatsen, de paprika's kan vinden en vervolgens de paprika kan plukken en elders deponeren.

In de ideale situatie kan deze efficiënt de paprika's vinden. Wanneer een paprika gevonden is, wordt bepaald waar de vrije werkruimte rondom deze paprika zich bevindt. Binnen deze ruimte bevindt zich niets dat het plukken in de weg kan zitten en daardoor beschadigd kan raken, en hierbinnen kan de plukker ongestoord zijn werk doen.

Problemen

Voordat er een werkende autonome paprikaplukker aan de slag kan gaan, moeten er heel wat obstakels genomen worden. Ten eerste moet het geheel op een mobiel platform worden geplaatst waarmee het systeem zich voort kan bewegen. Dit platform moet tot op zekere hoogte ook kunnen navigeren om alleen naar de juiste werklocatie te verplaatsen. Als het systeem is aangekomen op de werklocatie moet het door middel van sensoren de vrucht(en) aan de plant kunnen vinden. Wanneer de vrucht is gelokaliseerd moet het systeem bepalen hoe de manipulator van het systeem zonder schade aan te richten naar de vrucht gebracht kan worden. Bij de vrucht aangekomen moet de manipulator de vrucht losmaken van de plant. Tenslotte moet deze vrucht onbeschadigd op een gespecificeerde plek worden gedeponerd.

Voor al deze taken zijn er technieken beschikbaar om deze uit te voeren. Al deze technieken moeten echter aangepast worden aan deze specifieke toepassing.

Afbakening

Het deel van het probleem waarop dit rapport gericht is, is het vinden van de paprika aan de plant, en vervolgens het bepalen van de werkruimte rondom de paprika. Deze informatie is nodig om de manipulator die de vrucht gaat plukken naar de vrucht te kunnen sturen zonder schade aan te

richten aan de overige delen van de plant en andere planten in de buurt. Deze analyse wordt gedaan in een modelomgeving.

Allereerst is er een model nodig van de omgeving van de paprika. In dit geval dus een model van één, en eventueel naburige, paprikaplant(en) met daaraan de paprika's. Deze planten in de modelomgeving zullen willekeurig gegenereerd worden door een computer. Het modellerprogramma krijgt echter ook de optie om meetgegevens in te voeren om op die manier ook meer waarheidsgetrouwe situaties te kunnen testen. Deze meetgegevens zijn op dit moment niet beschikbaar, en het verzamelen van deze gegevens ligt buiten mijn opdracht, daarom richt ik mij op willekeurig gegenereerde planten voor de verdere berekeningen.

Zodra de plant gegenereerd is kan de vrije ruimte bepaald worden rondom de vruchten. Uiteindelijk is het doel de mogelijke routes vanuit de omgeving van de plant naar de vrucht te bepalen die geen interferentie met delen van de plant heeft.

Onderzoeksvragen

- Welke methode is te gebruiken om vrije ruimte rondom een vrucht te bepalen?
- Vanuit welke richtingen kan de vrucht het best benaderd worden zonder schade aan te richten aan het gewas?

Literatuurstudie vrije ruimte bepalen

Om te beginnen wordt er voor het bepalen van de vrije ruimte niet verder gekeken dan rechte lijnen, dit vanwege de beschikbare tijd. De vraag of een bepaald punt op een oppervlak langs een rechte lijn te bereiken is, is een bekend vraagstuk. Ook is bij het gebruik van rechte lijnen een link te leggen met zichtbaarheid. Als er namelijk een rechte lijn bestaat tussen een gegeven punt en de omgeving die nergens door wordt gehinderd, betekent dat dat het genoemde punt zichtbaar is vanuit de omgeving langs de genoemde lijn. In de literatuur ben ik dus op zoek geweest naar technieken om de zichtbaarheid van een bepaald punt te bepalen.

Toepassingsgebieden

In de literatuur zijn de volgende 5 belangrijke toepassingsgebieden naar voren gekomen (Keeler et al., 2007; Wang et al., 2007; Woo, 1994; Zetu et al., 2005).

NC / CNC

De toepassing van dergelijke techniek is van belang in verschillende gebieden. Zo zijn er verschillende fabricagetechnieken waarbij bereikbaarheid via een rechte lijn van belang is. Denk bijvoorbeeld aan computergestuurde freesmachines (NC /CNC). Een dergelijke machine werkt met een rechte frees, die bij een kleine diameter kan worden gezien als een rechte lijn. Het is hierbij natuurlijk belangrijk om te berekenen of de gewenste positie op het oppervlak van het werkstuk kan worden bereikt zonder daarmee andere delen van het werkstuk of de machine geraakt worden.

CMM

Een andere machine die sterk afhankelijk is van het langs een rechte lijn kunnen bereiken van een oppervlak is een zogenaamde coördinatenmeetmachine (CMM). Deze machines werken met een voeler, waarvan de positie bekend is. Zodra deze voeler een oppervlak raakt wordt de positie doorgegeven. Op deze manier kan de exacte positie van verschillende punten worden gemeten om te controleren of een werkstuk aan de specificaties voldoet, of een object te digitaliseren. Deze voeler is meestal gevormd als een dunne naald met een specifiek gevormd uiteinde. Deze naald is dus in dit geval de rechte lijn die het oppervlak moet bereiken, zonder onderweg iets anders te raken.

Malproductie

Een complete industrie die sterk afhankelijk is van het berekenen van vergelijkbare rechte lijnen is het vormen van mallen. Een object dat wordt gemaakt in een - niet flexibele - mal die vaker gebruikt dient te worden zal er in een rechte lijn uit moeten komen. Om te berekenen of het object zonder problemen uit de mal zal komen is het dus nodig dan vanuit een bepaalde hoek het volledige oppervlak van de mal langs een rechte lijn bereikt kan worden. Wordt deze rechte lijn op een of meer plekken geblokkeerd, dan zal het object niet uit de mal verwijderd kunnen worden.

CAD software

Dan is er vervolgens het drie dimensionaal tekenen op de computer, beter bekend als CAD (Computer Aided Design). Hierbij moet tijdens het ontwerpen een drie dimensionaal model worden weergegeven op het computerscherm. Bij het weergeven van dit model moet worden berekend

welke delen van het model er zichtbaar zijn vanuit het camerastandpunt op dat moment. Hier komt dus het bepalen van de zichtbaarheid van een punt aan te pas.

Cameravisie

Tenslotte zijn er nog de cameratoepassingen. Als er een bepaald punt bekeken moet worden met een camera is het natuurlijk van belang dat dat punt ook zichtbaar is. De plaatsen waar vandaan het gewenste punt bekeken kan worden moet dus worden bepaald om de plaatsing van de camera te bepalen.

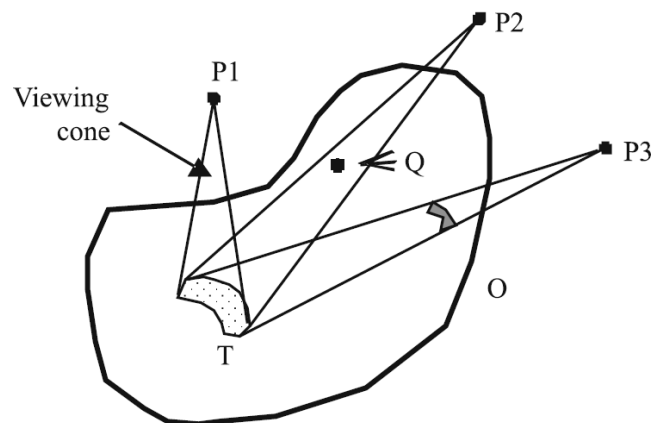
State of the art technieken

Cameralocatie bepaling voor vormreconstructie

Deze techniek wordt beschreven door Zetu et al. (Zetu et al., 2006).

Bij een vormreconstructie wordt een object met behulp van een camera vanuit verschillende hoeken vastgelegd. Door al deze verschillende beelden met elkaar te vergelijken kan de vorm van een object worden gereconstrueerd. Bij sommige vormen is het echter moeilijk om deze te reconstrueren, bijvoorbeeld de vorm van een holte. De camera's kijken namelijk van buitenaf naar het object, en de vorm van een holte is dan soms niet, en anders alleen vanuit bepaalde hoeken te zien. Om zo efficiënt mogelijk te kunnen werken zijn er slimme technieken om te bepalen vanuit welke hoeken de vormen waar te nemen zijn. Het ultieme doel hierbij is om met zo min mogelijk afbeeldingen, een onbekende vorm te reconstrueren.

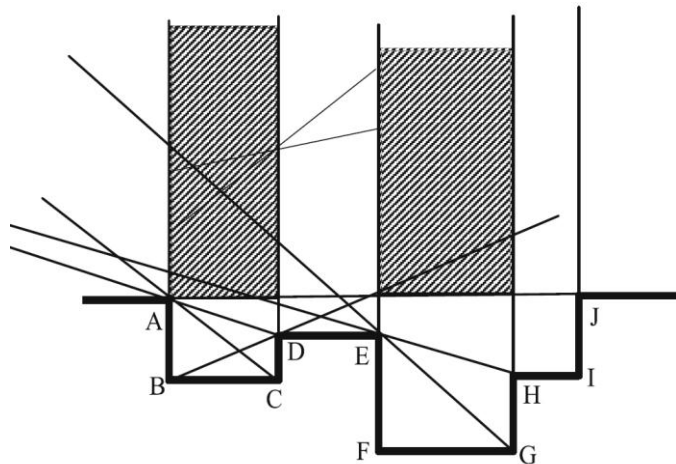
Om dit te bereiken moet er een zichtbaarheidsgebied worden gedefinieerd. De gehanteerde definitie is als volgt: een camerapositie ligt binnen het zichtbaarheidsgebied van een object wanneer en alleen wanneer het in de vrije ruimte buiten het object. Van alle punten is de viewing cone naar kenmerk T van object O getekend. De viewing cone van P1 valt verder met niets samen, daarom valt P1 binnen de definitie van het zichtbaarheidsgebied. De viewing cones van P2 en P3 vallen beiden samen met andere objecten en/of kenmerken en daardoor wordt het zicht op T gedeeltelijk of geheel geblokkeerd. Om die reden vallen P2 en P3 buiten het zichtbaarheidsgebied.



Figuur 1 Zichtbaarheid definities voor kenmerk T van het object O voor drie punten P1, P2 en P3. (Zetu et al, 2006)

De reconstructie van een vorm op de computer wordt opgebouwd uit driehoeken, beter bekend als polygonen. Om deze polygonen te kunnen vormen is het zaak om de randen te definiëren. Wanneer de randen zijn gedefinieerd kunnen de polygonen opgevuld worden. Deze randen zijn op te delen in twee categorieën: concave randen en convexe randen. Concaaf is in dit geval een rand waarbij de hoek tussen de vlakken aan weerszijde groter is dan 180 graden, gemeten aan de binnenkant van het object. In Figuur 2 zijn dit punten B, C, F, G en I. Bij convexe randen is deze hoek dus minder dan 180 graden, zoals bij de punten A, D, E, H en J.

Er worden vervolgens drie theorieën aangedragen. Ten eerste is het zo dat wanneer er een convexe reeks van concave randen begrensd is door twee convexe randen, dan is er slechts één camerastandpunt nodig om alle randen te detecteren. Zie reeks A-B-C-D in Figuur 2. Ten tweede is er, wanneer er na een concave rand twee convexe randen bevinden, een nieuw camerastandpunt nodig. Dit is te zien in reeks C-D-E-F in Figuur 2. Deze twee theorieën samen hebben het volgende gevolg: wanneer er geen twee opeenvolgende convexe randen voorkomen tussen een reeks concave randen, is er geen nieuw camerastandpunt nodig. Zie reeks E-F-G-H-I-J in Figuur 2.



Figuur 2 Zichtbaarheidsgebieden voor een polygooncontour. Voor een volledige zichtbaarheid moet het observatiepunt binnen het gearceerde gebied vallen. Twee gebieden, zoals hier het geval is, betekent minimaal twee observatiepunten. (Zetu et al, 2006)

Voor de derde theorie moet er eerst nog een definitie gegeven worden, de draaiingshoek. Deze hoek is de hoek tussen twee convex verbonden vlakken, maar in tegenstelling tot eerder is de hoek niet binnenin gemeten. De hoek wordt gedefinieerd als de hoek tussen het verlengde van het eerste vlak met het tweede vlak, oftewel de draaiing van het tweede vlak ten opzichte van het eerste. De derde theorie geldt voor een reeks convex verbonden randen, en luidt: een reeks convex verbonden randen vereisen slechts één camerastandpunt wanneer de draaiingshoek van het huidige punt kleiner is dan de som van de draaiingshoeken van de voorgaande punten. Deze theorie kan worden geïnterpreteerd als: wanneer, voor een serie van convex verbonden randen, er twee parallel zijn, dan horen ze bij twee verschillende camerastandpunten.

Aan de hand van deze theorieën kan men snel het aantal benodigde camerastandpunten bepalen voorafgaand aan het reconstructieproces. Het object dat moet worden gereconstrueerd kan snel met het oog worden bekeken. Als de hierboven besproken theorieën als richtlijnen gebruikt worden kan men snel bepalen of een extra camerastandpunt al dan niet nodig is.

Sferische zichtbaarheidskaarten

Deze techniek wordt beschreven in verschillende variaties door Keeler et al., Liu et al. en Woo (Keeler et al., 2007; Liu et al., 2009; Woo, 1994).

Bij het weergeven van een drie dimensionaal model op de computer zijn er altijd delen van het model niet zichtbaar. Het is niet zo zeer nodig om te weten welke delen er wel en niet zichtbaar zijn, de computer kan simpelweg het hele model berekenen en weergeven. In dat geval worden de onzichtbare delen vanzelf bedekt door de zichtbare delen. In dit geval duurt het berekenen van een model echter langer dan nodig is. Dat kan vervelend zijn, maar met een snelle computer en/of een klein model is dit geen probleem.

Lastiger wordt het als er een twee dimensionaal plaatje gemaakt moet worden van het drie dimensionale model. Hiervoor zijn er twee mogelijkheden. De eerste is simpel en dit is dan ook de optie die vrijwel altijd wordt gebruikt: een rasterafbeelding. Hierbij wordt een afbeelding geproduceerd van een vast aantal pixels. Dit vaste aantal pixels geeft een vast aantal datapunten om

te berekenen. Deze berekening hoeft slechts één maal uitgevoerd te worden. Voor ieder punt wordt er berekend wat er zichtbaar is en dit wordt in de afbeelding opgeslagen.

De tweede manier is moeilijker: een vectorafbeelding. Bij een dergelijke afbeelding wordt er geen vast aantal pixels opgeslagen. Er wordt aan de hand van vectoren een exacte vorm berekend van de weer te geven onderdelen. Dit houdt in dat er in principe oneindig ingezoomd kan worden op de afbeelding zonder kwaliteitsverlies. Daarbij kosten dergelijke afbeeldingen minder opslagruimte. Een dergelijke vorm van afbeeldingen produceren is dus te verkiezen boven een rasterafbeelding. De moeilijkheid hier is dat er geen aantal datapunten vast ligt. Zolang het model nog drie dimensies heeft kan voor ieder zoomniveau de weergave opnieuw worden berekend. Zodra de afbeelding naar twee dimensies is omgezet is de data voor de derde dimensie verloren gegaan en kan er niets meer herrekend worden.

Tot op heden is er geen geschikte techniek om een exacte omzetting van drie dimensies naar vectoren in twee dimensies mogelijk te maken. Veel grote 3D modelleerprogramma's gericht op de entertainmentindustrie hebben deze optie dan ook niet. Enkele programma's gericht op de CAD industrie hebben deze optie wel, maar kunnen met de verkeerde instellingen een incorrect resultaat produceren. De techniek die hier vaak wordt gebruikt is het opbouwen van de afbeelding van achter naar voren. Zo worden de onderdelen die achter een ander onderdeel liggen eerst getekend en vervolgens overschreven. Als de tekening dus in de verkeerde volgorde wordt opgebouwd is het resultaat ook verkeerd. Daarnaast kost deze techniek ook te veel tijd, omdat ook ieder onzichtbaar vlak volledig doorgerekend wordt. Een algoritme om deze omzetting exact uit te voeren is gewenst.

Een techniek die regelmatig terugkeert in de literatuur is het maken van een sferische zichtbaarheidskaart (Keeler et al., 2007; Liu et al., 2009; Woo, 1994). Hierbij wordt het gebied waar binnen een bepaald punt zichtbaar is geprojecteerd op een bol. Dit kan op twee manieren. Bij de eerste staat het observatiepunt vanuit waar de afbeelding gemaakt dient te worden vast, en dat punt is het midden van de bol waarop de projectie plaatsvindt. De andere manier is om de bol rondom een vastgesteld oppervlak te vormen en daarop wordt een gebied geprojecteerd waarvandaan het vlak gezien kan worden, om zo de mogelijke observatiepunten te berekenen.

Vast observatiepunt

Wanneer het observatiepunt vast staat, kan men de technieken toepassen beschreven door Keeler et al. (Keeler et al., 2007). Alle vormen van een drie dimensionaal computermodel bestaan uit vlakken begrensd door rechte lijnstukken. Om een dergelijke lijn te projecteren op een bol neemt men drie punten: het middelpunt van de bol en de twee punten die het lijnstuk definiëren. Deze drie punten samen definiëren een vlak. De lijn waar dit vlak en de bol snijden beschrijft een cirkel die zich op het oppervlak van de bol bevindt. Deze cirkel deelt de bol op in twee halve bollen. Aan de hand van de vorm die getekend dient te worden kan de juiste helft geselecteerd worden, om zo onnodig werk op de andere helft te voorkomen. Op de juiste helft wordt nu het volgende lijnstuk op dezelfde manier geprojecteerd. Dit definieert een kleinere sectie van de bol waarop vervolgens weer het volgende lijnstuk wordt geprojecteerd. Zo wordt er een sectie van het oppervlak van de bol gedefinieerd, wat een projectie is van het vlak. In Figuur 3 staat een voorbeeld van de projectie van een driehoek.

Iedere zijde van de projectie krijgt een Booleaanse waarde mee bij het aanmaken, te weten ‘waar’. Dit geeft aan dat deze zijde van een vlak getekend dient te worden in het eindresultaat. Zodra een zijde wordt doorsneden door een cirkel die wordt getekend als projectie van een ander lijnstuk van het model, wordt deze Booleaanse waarde veranderd. Dit geeft aan dat dit lijnstuk niet geheel zichtbaar zal zijn in het eindresultaat.

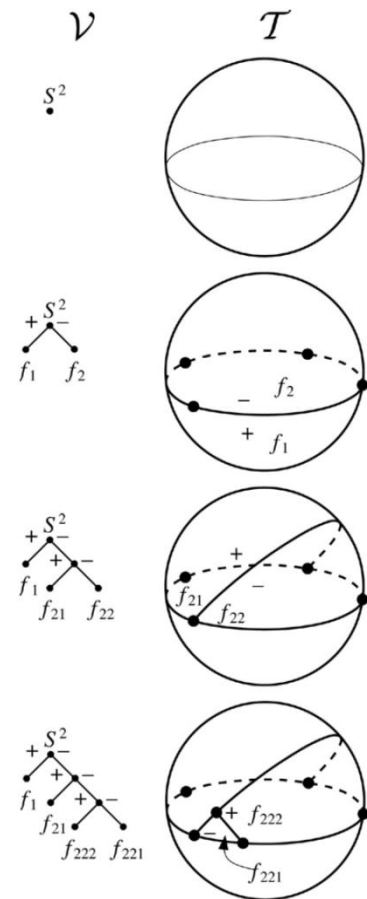
De toegepaste techniek om hiermee om te gaan is dat de polygoon die slechts gedeeltelijk zichtbaar is wordt opgedeeld in twee polygonen: een zichtbare en een onzichtbare. Op deze manier wordt de berekening van de twee dimensionale afbeelding versneld, maar niet bemoeilijkt. Er hoeven namelijk alleen nog maar de zichtbare polygonen berekend te worden waardoor de berekening sneller zal gaan. Er hoeven echter nog steeds alleen maar gehele polygonen berekend te worden wat er voor zorgt dat de berekeningen die uitgevoerd worden niet veranderd hoeven te worden.

Mogelijke observatiepunten berekenen

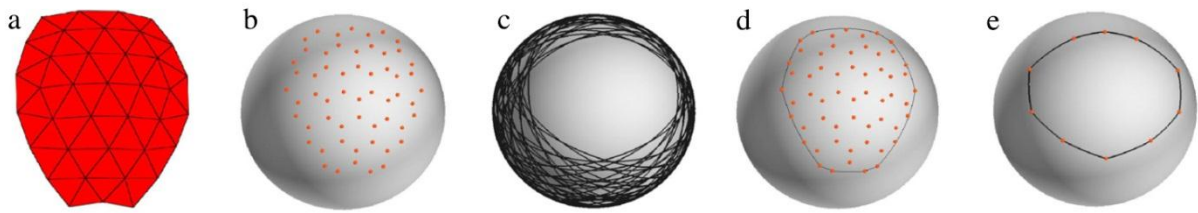
Bij deze aanpak, beschreven door Lui et al. en Woo (Liu et al., 2009; Woo, 1994), wordt de bol niet om het observatiepunt gemaakt, maar om het vlak wat men wil bekijken. Bij deze aanpak is de vraag dus “waarvandaan kan ik dit vlak zien?” in plaats van “wat kan ik zien vanaf hier?”. Om te beginnen wordt er van het vlak dat men wil analyseren een Gaussian map gemaakt. Dit is een speciale manier van projecteren van een vlak op een bol. De Gaussian map van een vlak is een enkel punt op een bol. Ook hier hebben we met een computermodel te maken, dus een model opgebouwd uit platte vlakjes. Ieder van deze platte vlakjes resulteert in een punt op de bol.

Dit punt op de bol is het snijpunt van de normaalvector van het vlak met het oppervlak van de bol. Een vlak kan vanaf de ‘voorkant’ – de kant die men wil zien, de positieve richting van de normaalvector - gezien worden vanuit ieder punt aan die kant van het vlak. Dit houdt in dat de richtingen vanuit waar het vlak gezien kan worden, worden beschreven door een halve bol. De afscheiding van een halve bol van de rest van de bol is een cirkel met als middelpunt het middelpunt van de bol. Een verzameling punten op de bol kan worden omgezet naar een verzameling cirkels die de afscheidingen vormen van een verzameling halve bollen.

Nu is het dus zo dat een punt dat het oppervlak van een model kan worden omgezet naar een verzameling cirkelvormige afscheidingen. Een punt op de bol dat zich binnen al deze afscheidingen bevindt, is een richting vanuit waar alle vlakjes die samen het oppervlak van het model vormen kunnen worden waargenomen. Deze punten kunnen worden beschreven als een vlak op de bol. Ieder punt op dit vlak is dus een mogelijke richting. Dit vlak wordt de lokale zichtbaarheidskaart genoemd. Deze wordt lokaal genoemd omdat er geen rekening wordt gehouden met de omgeving, dit zijn alle richtingen vanuit waar je het gehele vlak kan zien in een open ruimte. In Figuur 4 is een grafische representatie van dit proces te zien.



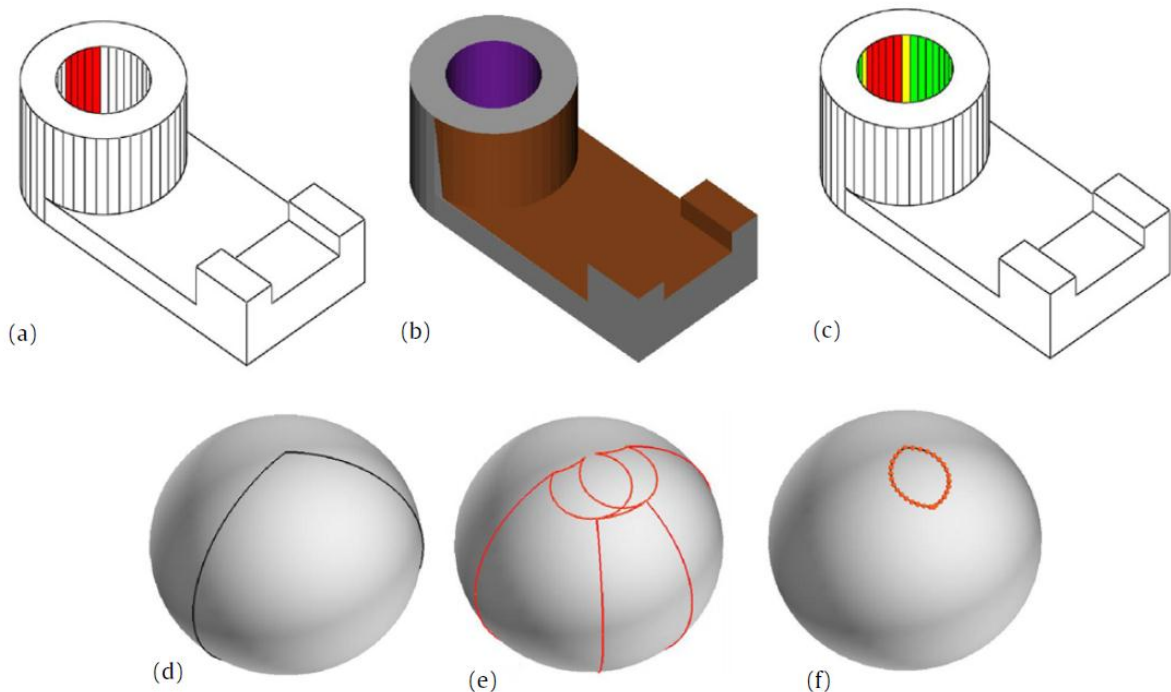
Figuur 3 Projectie van een driehoek op een bol bij een vast observatiepunt (Keeler et al, 2007)



Figuur 4 Vorming van de lokale zichtbaarheidskaart (a) het oppervlak, (b) de discrete Gaussian map, (c) de verzameling cirkels die de halve bollen afbakenen, (d) de sferische afbakening van (b), (e) de puntenverzameling die binnen alle afbakeningen in (c) liggen, de lokale zichtbaarheidskaart (Liu et al., 2009)

Deze methode wordt nog uitgebreid door Lui et al (Liu et al., 2009). Er kunnen in een model ook oppervlakken zitten die het oppervlak wat we willen bekijken gedeeltelijk of geheel verbergen. Als twee vlakken convex staan ten opzichte van elkaar kunnen die elkaar nooit afschermen. Er hoeft dus alleen maar rekening gehouden te worden met concave oppervlakken. Dezelfde techniek die wordt gebruikt om een lokale zichtbaarheidskaart te genereren kan worden gebruikt om een globale afschermingskaart te maken. Op dezelfde manier dat een lokale zichtbaarheidskaart alle richtingen bevat vanuit waar een vlak waar te nemen is, bevat de globale afschermingskaart alle richtingen die worden geblokkeerd door andere vlakken.

De globale zichtbaarheidskaart bevat alle richtingen vanuit waar een vlak zichtbaar is. Dit zijn 4 alle richtingen vanuit waar een vlak lokaal waar te nemen is, maar niet de richtingen die afgeschermd worden door andere vlakken. Dit komt erop neer dat de globale zichtbaarheidskaart gelijk is aan de lokale zichtbaarheidskaart min de globale afschermingskaart. Het resultaat hiervan is te zien in Figuur 5.



Figuur 5 vorming van een globale zichtbaarheidskaart (a) het model met het vlak wat men wil bekijken worden in rood, (b) de mogelijk afdekkende vlakken, (c) de werkelijk afdekkende vlakken in geel en groen, (d) de lokale zichtbaarheidskaart, (e) de globale afdekkingskaart, (f) de globale zichtbaarheidskaart (Liu et al., 2009)

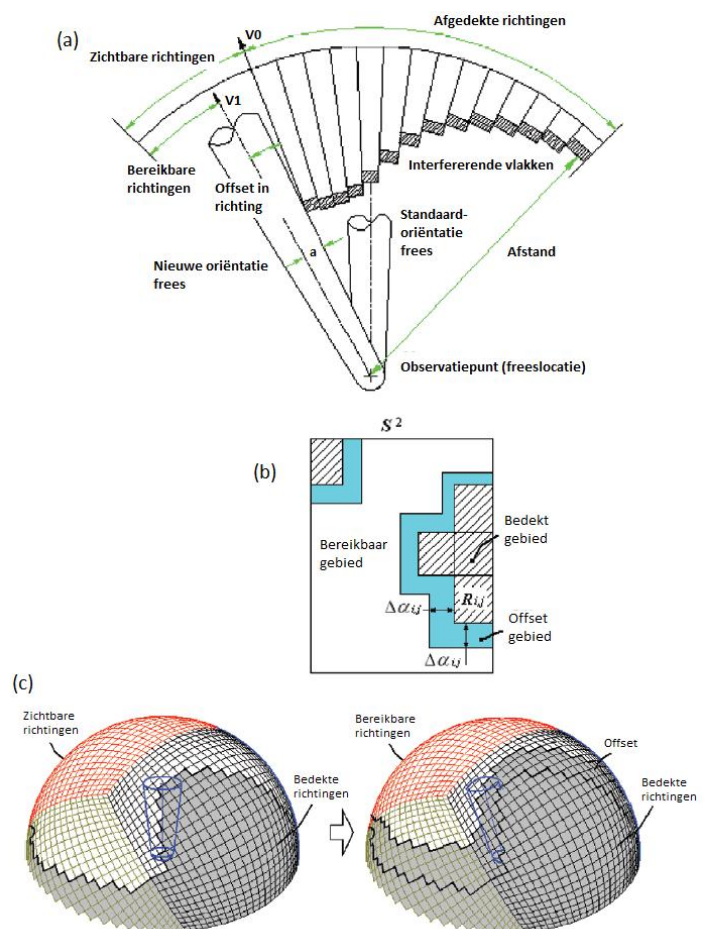
Werktuigoriëntatie correctie

Deze techniek wordt beschreven door Wang et al. (Wang et al., 2007).

Bij het omzetten van een drie dimensionaal computermodel naar een aansturing voor een freesmachine moet worden gecontroleerd of de frees geen onbedoeld contact maakt met het werkstuk. Hierbij kan zowel het werkstuk dat wordt bewerkt vernield worden door stukken weg te frezen die niet weg moesten, maar ook de freesmachine zelf kan beschadigd raken. In sommige gevallen kan er weinig aan gedaan worden als er een ongewild contact wordt gemaakt. Een freesmachine met vijf vrijheidsgraden heeft echter altijd verschillende manieren om een positie te benaderen.

Huidige systemen hebben een simpele techniek om botsingen op te lossen. Zodra er een botsing wordt gedetecteerd wordt de frees kop simpelweg weggedraaid van de botsing. Vervolgens wordt er gecontroleerd of de botsing is verholpen. Als dat het geval is kan er verder gegaan worden met het pad plannen. Is dit echter niet het geval wordt de frees kop opnieuw weggedraaid en de nieuwe positie gecontroleerd. Dit principe werkt meestal goed als de botsing een enkel punt betreft. Als er echter meerdere punten geraakt worden kan het wegdraaien bij het ene punt een grotere botsing in het andere punt veroorzaken. Zo kan het programma in een oneindige lus terecht komen waarbij de kop steeds heen en weer gedraaid wordt. Bij een botsing waarbij de frees kop volledig samenvalt met het werkstuk is er geen vrije richting te bepalen en werkt dit principe helemaal niet.

Mocht het zo zijn dat deze techniek niet werkt moet er een ander principe worden toegepast. Er moet worden bepaald welke richtingen een vrije toegang bieden tot het punt wat men wil bereiken. Dit wordt in dit geval gedaan door een kubus te plaatsen rondom het probleempunt. De normaalvector van het punt snijdt het middelpunt van het bovenvlak van de kubus. Ieder vlak van de kubus wordt opgedeeld als een raster. Vervolgens wordt er voor ieder punt van dit raster bekeken of er vanuit het middelpunt, het punt waar de frees kop naar toe moet, vrij zicht is op dat punt op de kubus. Omdat er binnen het programma al een drie dimensionaal model bestaat wordt dit simpelweg gedaan door vanuit het



Figuur 6 Vormen van een kaart met bereikbare richtingen. (a) benodigde offset van het interfererende vlak bepalen, (b) offset gebied vormen, (c) complete bereikbaarheid geprojecteerd op een bol (Wang et al., 2007)

middelpunt een rasterafbeelding te maken in de richting van de vlakken van de kubus. In deze afbeelding wordt vervolgens voor iedere pixel gecontroleerd of er op deze pixel een vlak van het model is te zien of dat deze vrij zicht heeft. Het programma weet ook de locatie van alle vlakken, dus als er geen vrij zicht is kan de afstand tot het interfererende vlak uitgelezen worden uit het model. Zo wordt een drie dimensionale kaart gemaakt die alle mogelijke richtingen aangeeft waar langs het gewenste punt in een rechte lijn kan worden benaderd.

Een zichtlijn heeft echter geen dikte, een frees wel. Daarom moet er ook nog een afstand bewaard worden tussen de vlakken die in de weg zitten en de middenlijn van de frees. Aan de hand van de dimensies van de frees en de vrije afstand tussen het te bereiken punt en het interfererende vlak wordt de benodigde offset bepaald. Deze wordt vervolgens meegenomen in de kaart. Deze kaart wordt op een bol geprojecteerd om de verhoudingen één op één weer te geven. Dit principe wordt geïllustreerd in Figuur 6.

Nu is er hele verzameling mogelijke richtingen bekend. Het volgende probleem is kiezen welke richting te gebruiken. Dat is simpel opgelost: de richting die het dichtst bij de richting van het vorige punt komt. Op deze manier hoeft de frees kop het minst te draaien en verkrijgt men het snelste pad.

Visibility cones

De besproken technieken gaan veelal uit van rechte lijnen om de zichtbaarheid van een punt te bepalen. Echter, als men in een bepaalde richting kijkt is er meer zichtbaar dan alleen een punt. De zichtbare ruimte is niet een lijn maar heeft de vorm van een kegel. Een dergelijke kegel kan worden beschreven met enkel een beginpunt, een middenlijn en een hoek. Dit is slechts één parameter meer dan een zichtlijn en zou daarom een logische volgende stap zijn. Met één extra parameter kan een gebied worden beschreven waar binnen het gewenste punt kan worden bereikt (Kim et al., 1995). Dit vergt echter een veel ingewikkelder programma om te berekenen, en om die reden zal het niet binnen de afbakening van het praktijkgedeelte van dit project voorkomen.

Toch is dit onderwerp wel interessant om te bekijken. Een zichtlijn heeft namelijk maar één dimensie, een lengte. Een robot die een paprika moet plukken heeft echter ook nog een breedte en een hoogte. Er is daarom meer nodig dan een vrije lijn om de paprika te bereiken. Met een visibility cone kan een ruimte worden gedefinieerd waar binnen de robot zijn werk kan doen.

Een visibility cone kan ook nog een stap verder gaan. Een visibility cone hoeft namelijk ook niet rond te zijn. Een vrij gevormde visibility cone kan ook gebieden beschrijven die een niet ronde doorsnede hebben. Hiermee kan een gebied dus nog nauwkeuriger worden beschreven. Het nadeel hiervan is dat de definitie dan een heel stuk moeilijker wordt dan de eerder genoemde middenlijn en hoek.

Van een visibility map, die eerder in dit hoofdstuk is besproken, kan een visibility cone worden gemaakt en vice versa (Yin et al., 2000). Dit geeft ook weer mogelijkheden om de beschreven technieken van visibility maps toe te passen voor de vrije ruimte bepaling en daaruit een ruimte te definiëren in de vorm van een visibility cone.

Conclusie

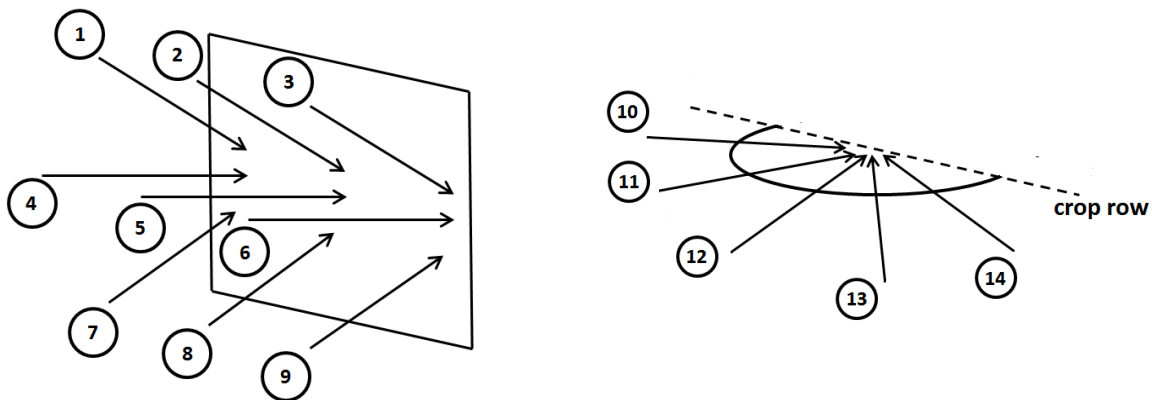
De techniek die gebruikt zal worden voor de analyse van de paprikaplant zal afgeleid worden van de techniek beschreven door Wang et al. (Wang et al., 2007). De reden om deze techniek te gebruiken is dat deze vrijwel één op één te gebruiken is, met enkele kleine aanpassingen. Deze techniek gaat

namelijk net als deze casus uit van een bestaand drie dimensionaal computermodel. Vervolgens wordt er vanaf een bekend punt berekend welke richtingen een vrije toegankelijkheid tot dit punt verschaffen. Ook is deze techniek redelijk simpel te programmeren, wat wenselijk is gezien de beperkte tijd beschikbaar voor dit project. De techniek gaat namelijk enkel uit van rechte lijnen.

De paprika wordt het bekende middelpunt vanuit waar rechte lijnen naar buiten getrokken worden. Van deze lijnen wordt bepaald of deze al dan niet interfereert met een onderdeel van de plant.

Vergelijkbaar onderzoek

Ruizendaal (Ruizendaal, 2011) heeft vrijwel hetzelfde vraagstuk onderzocht. In zijn onderzoek is met een camera vanuit verschillende posities en verschillende hoeken naar een rij paprikaplanten gekeken, zie Figuur 7. Vervolgens is in deze beelden bepaald hoeveel paprika's er zichtbaar zijn.



Figuur 7 De verschillende hoeken waaronder er naar de paprikaplanten is gekeken (Ruizendaal, 2011)

Dit is een alternatieve aanpak van hetzelfde vraagstuk. Beide methoden zoeken naar een richting voor optimaal zicht op de paprika. Het verschil zit in het feit dat Ruizendaal van buitenaf de plant bekijkt en in die beelden de paprika's detecteert. In dit rapport wordt vanuit de paprika naar buiten gekeken. De resultaten van het onderzoek van Jos Ruizendaal worden in de discussie vergeleken met de resultaten van dit rapport.

Materialen en methode

Voor een ruimtebepaling in drie dimensies rondom delen van een plant zijn er een aantal benodigdheden. Om te beginnen een model van een plant. Eerst is er een ontwerp nodig dat een paprikaplant zo goed mogelijk benadert, maar niet teveel variabelen bevat. Dit is zowel van belang voor het programmeren als voor het eventueel opmeten van een bestaande plant.

Als dit ontwerp er is zal dit omgezet moeten worden naar een computermodel om analyse van het model mogelijk te maken. Dit zal moeten gebeuren met een programma dat zowel het model in drie dimensies weer kan geven, en een analyse uit kan voeren. Voor een bredere toepassing zal er ook een gebruikersinterface moeten komen.

Zodra er een computermodel van de plant beschikbaar is dient de computer een analyse uit te voeren. Hiervoor is een algoritme nodig dat de benodigde informatie uit het model kan genereren. Dit algoritme zal ook geprogrammeerd moeten worden, zodat de analyse door een computer uitgevoerd kan worden. In de komende hoofdstukken zal elk van deze benodigdheden in detail beschreven worden.

Ontwerp plant

Het ontwerp van het drie dimensionale model van de paprikaplant is gemaakt door Bac (Bac, 2011). Onderstaande tekst en Figuur 8 zijn een vertaling van de beschrijving die Bac zelf heeft geschreven over zijn ontwerp.

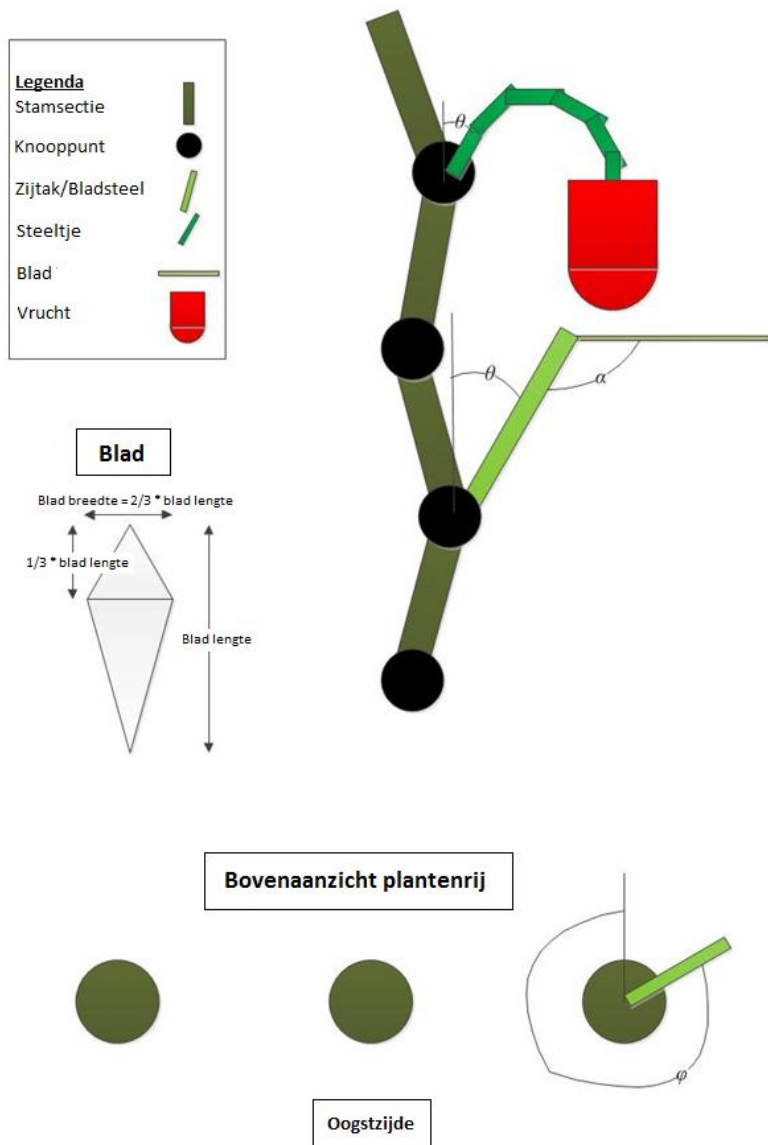
3D tekening van een paprikaplant

De stam

De stam is opgebouwd uit knooppunten. Het aantal knooppunten wordt gedefinieerd bij het aanmaken van een nieuwe plant. Ieder knooppunt wordt op zijn beurt gedefinieerd door middel van x-, y- en z-coördinaten. De dikte van de stam staat vast op 1 cm en blijft dat over de volledige lengte. De overige onderdelen van de plant worden allemaal verbonden aan een van de knooppunten.

Zijtak

Zijtakken worden gemaakt langs een rechte lijn tussen een knooppunt en een gedefinieerd punt. Dit eindpunt wordt gedefinieerd door twee hoeken, φ en θ , en een lengte. Deze parameters kunnen ofwel opgegeven worden ofwel willekeurig gegenereerd. De willekeurig gegenereerde parameters dienen binnen bepaalde grenzen te worden gedefinieerd om een realistisch model te verkrijgen. De dikte van een zijtak staat vast op een 0,5 cm.



Figuur 8 Grafische weergave van de verschillende variabelen die het model van de paprikaplant beschrijven (Bac, 2011)

Blad

Een blad bestaat uit twee onderdelen, de steel en het bladoppervlak. De steel van het blad wordt op dezelfde manier gemaakt als de zijstelen aan de stam. Het bladoppervlak wordt gemaakt aan de hand van twee driehoeken. De twee driehoeken komen samen op het breedste punt van het blad. Parameters voor het blad zijn dezelfde als die voor een zijtak: φ , θ en de lengte. Verder is er nog de hoek α tussen de steel en het blad. Deze hoek wordt aangenomen 60° te zijn. De dikte van het blad wordt aangenomen als minder dan 0,1 cm en wordt daarom genegeerd.

Voor de maximale breedte, de positie van de maximale breedte en de lengte van de steel worden allemaal aangenomen om gerelateerd te zijn aan de lengte van het blad. Handmatige metingen aan planten resulteerden in de volgende aannames:

- De maximale breedte van het blad is twee derde van de bladlengte
- De maximale breedte bevindt zich op een derde van de bladlengte
- De lengte van de steel is twee derde van de bladlengte

Vruchten

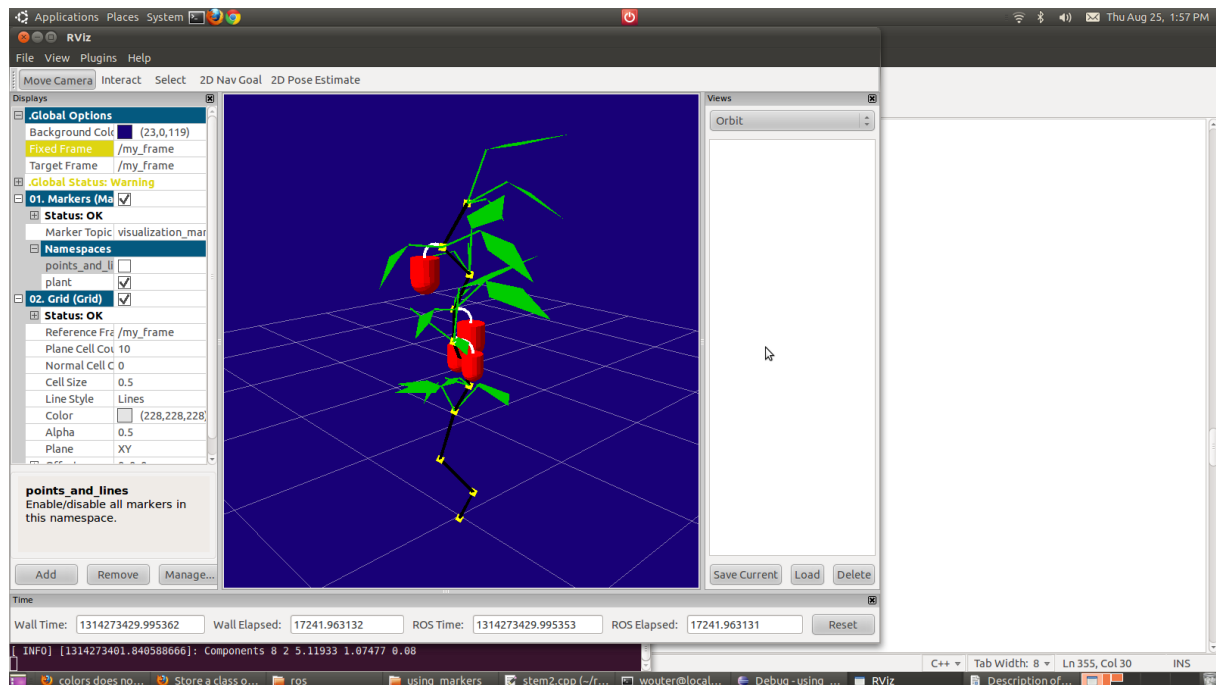
Een vrucht is samengesteld uit een cilinder en een halve bol. De cilinder representeert de bovenste helft, de halve bol de onderste helft. De diameter van de halve bol en de hoogte én breedte van de cilinder zijn daarom gelijk genomen. Er wordt aangenomen dat de vruchten recht naar beneden hangen en dat het steeltje op het middelpunt van het bovenzvlak bevestigd is.

Steeltjes

Een steeltje is samengesteld uit een aantal verbonden lijnstukken. De parameters voor een steeltje zijn wederom φ , θ en de lengte. Het aantal stukken waaruit een steeltje wordt opgebouwd verschilt afhankelijk van de hoek θ .

Resultaat

Dit ontwerp werd gemodelleerd in RViz (<http://www.ros.org/wiki/rviz>), een visualisatieprogramma dat gebruik maakt van C++ code. Het resultaat is te zien in Figuur 9.



Figuur 9 Een instantie van een plant gemodelleerd in RViz volgens het model van Wouter Bac (Bac, 2011)

Programmeren plant model

Het programma

Het model is al geprogrammeerd door de ontwerper, Wouter Bac (Bac, 2011). Dit model is geprogrammeerd in C++ code en wordt gevisualiseerd in een visualisatieprogramma genaamd RViz. Dit programma kan echter geen berekeningen uitvoeren op het resulterende model. Om die reden moet het model eerst omgezet worden naar MatLab code. De functies die het programma al dan niet moet gaan bevatten zijn opgedeeld in eisen en wensen.

Eisen

Plantmodel genereren

Om te beginnen moet het programma een model van de plant kunnen genereren. Dit model bestaat enkel uit variabelen en bevat alle gegevens die nodig zijn om de plant te beschrijven. Aan de hand van deze gegevens kunnen alle berekeningen worden uitgevoerd en kan de plant worden getekend. Dit is de beginstap voor alle eisen en wensen die nog volgen, zonder dit model werkt er niets.

Plant tekenen in 3D

Mensen zijn visueel ingesteld. Lijsten met variabelen zijn voor een persoon niet bruikbaar, daaruit kan men zich niet voorstellen wat er gebeurt en hoe alles eruit ziet. Om het bovenstaande model weer te kunnen geven op een manier dat de gebruiker ook werkelijk begrijpt wat er gebeurt, zal een afbeelding geproduceerd moeten worden. Dit zal een drie dimensionaal plaatje worden om goed te kunnen zien hoe de plant eruit ziet en een beeld te vormen van wat ermee gebeurt.

Willekeurige plant genereren

Er zijn op het moment nog geen gegevens beschikbaar uit metingen van een echte plant. De plant zal zonder meetgegevens gegenereerd moeten worden.

De computer kan snel veel willekeurige getallen produceren. Er zal daarom een functie komen die willekeurige planten kan genereren. Zo kan ondanks het ontbreken van meetgegevens toch op meerdere planten worden geëxperimenteerd.

Vrije ruimte en richtingen bepalen

Het programma moet vanuit een plantmodel gaan berekenen waar de vrije werkruimte rondom de paprika zich bevindt. Uit deze vrije ruimte moet worden bepaald vanuit welke richtingen de plukrobot de paprika kan benaderen.

Wensen

Naast deze functies die absoluut nodig zijn, zijn er nog een aantal functies die de bruikbaarheid van het programma vergroten. Deze zorgen voor een bredere toepassingsmogelijkheid in latere stadia en bruikbaarheid door een grotere doelgroep.

Plant kunnen opslaan

Omdat het programma meestal met willekeurig gegenereerde planten werkt, zal er ook regelmatig een nutteloze plant tussen zitten. Bijvoorbeeld een plant met twee paprika's in elkaar, of een paprika die door een blad heen zit. Dit is op zich niet erg, omdat er zo weer een nieuwe is gemaakt. Als er dan echter een geschikte plant voorbij komt, of bijvoorbeeld een plant waarbij de paprika niet bereikt kan worden is het nuttig als deze bewaard kan worden voor nadere analyse. Daarom zou een functie om de huidige plant op te slaan zeker een toegevoegde waarde hebben.

Meetgegevens kunnen invoeren

Willekeurige planten zijn snel gemaakt en nuttig om veel verschillende situaties te testen, maar het blijven willekeurig gegenereerde planten. Om de werkelijkheid dichterbij te benaderen zou het nuttig zijn om een echte plant na te meten en deze gegevens te kunnen gebruiken om een plantmodel van af te leiden. Op deze manier wordt de simulatie realistischer en zijn de uitkomsten te toetsen in de werkelijkheid.

Plant kunnen laden

Als een plant opgeslagen kan worden, is het natuurlijk van belang dat deze ook weer teruggehaald kan worden in het programma. Ook als men meetgegevens van een plant heeft ingevoerd moet het mogelijk zijn dat deze in het programma geladen kunnen worden. Deze functie volgt vrijwel automatisch uit de twee voorgaande gewenste functies.

Een GUI

Om het programma bruikbaar te maken voor mensen die geen verstand hebben van MatLab en/of programmeren, maar gewoon een model van een plant willen hebben is het zeer nuttig om een simpele GUI (Graphical User Interface), een grafische gebruikersinterface te hebben. Deze bestaat uit knoppen en schuiven die iedereen kan gebruiken.

Plant class

Bij het programmeren is gebruik gemaakt van het zogeheten object georiënteerd programmeren (McGarrrity, 2009). Alle functies zijn verwerkt in een zogenaamde 'class'. Een class is de klasse van een object. Er zijn verschillende standaardklassen binnen MatLab zoals variabelen en strings. Voor de paprikaplant is er een nieuwe klasse aangemaakt. Dit houdt in dat wanneer er een plant gemaakt wordt, en een object wordt aangemaakt van de klasse 'Plant'. Dit object bevat alle eigenschappen die de paprikaplant beschrijven, de zogeheten 'properties'. Deze klasse beschrijving bevat ook de verschillende functies die nodig zijn om de plant als drie dimensionaal model weer te geven, de zogeheten 'methods'. Hieronder volgt een beschrijving van de properties en de methods die in de class zijn verwerkt.

Properties

De properties zijn opgedeeld in drie verschillende categorieën. De 'public properties' zijn de openbare eigenschappen. Dit zijn de basiseigenschappen van de plant, deze eigenschappen zijn nodig om een plantmodel aan te maken. Deze eigenschappen zijn zo uit te lezen en worden ook

weergegeven als eigenschappen bij de uitleg van de klasse. De volgende twee categorieën zijn ‘private’, oftewel privé. Die kunnen niet worden uitgelezen door de gebruiker, maar kunnen wel door de verschillende functies worden gebruikt. Hieronder vallen de ‘constant properties’, de constanten en de ‘dependent properties’, de afhankelijke eigenschappen. De constanten zijn eigenschappen zoals kleuren en diktes van stam en takken. De dependent properties zijn eigenschappen die door het programma worden gebruikt voor het tekenen van de plant. Deze kunnen worden berekend uit de openbare eigenschappen, vandaar de naam afhankelijk. Deze eigenschappen hoeven niet ingevoerd te worden.

Public properties

Hieronder vallen de eigenschappen die nodig zijn om te plant te creëren. Om te beginnen zijn er het aantal van de verschillende onderdelen van de plant. Wanneer er een willekeurige plant wordt gemaakt is die het enige wat ingevoerd dient te worden. Wil men echter een volledig gedefinieerde plant maken, dienen ook alle eigenschappen van deze onderdelen opgegeven te worden. De makkelijkste manier is de ‘Load’ functie, die later besproken wordt. Men kan echter ook alle gegevens met de hand invoeren.

Constant properties

Hier staan de constanten van het programma, de eigenschappen die voor iedere plant hetzelfde zijn. Dit zijn de dikte van de stam, de dikte van de stelen en alle benodigde kleuren. Deze kunnen niet worden aangepast of uitgelezen door de gebruiker, maar wel gebruikt door de functies die later besproken worden.

Dependent properties

Deze eigenschappen staan niet op zichzelf, maar zijn volledig afhankelijk van de openbare eigenschappen. Deze eigenschappen bestaan ook niet als constanten binnen de plant, maar worden berekend op het moment dat ze nodig zijn. Onder deze categorie vallen onder andere de coördinaten van de begin- en eindpunten van de stengels. Deze zijn voor de gebruiker niet nuttig om te kennen, aangezien de hoeken en de lengte al bekend zijn in de openbare eigenschappen. Deze zijn echter wel nodig om de plant te kunnen tekenen en voor de berekeningen die binnen de plant worden uitgevoerd.

Methods

De methods zijn ook opgedeeld in categorieën. Er zijn de ‘public methods’, de functies die door de gebruiker worden aangeroepen. Deze zijn openbaar zodat de gebruiker kan zien welke functies hij of zij beschikbaar heeft. Naast deze openbare functies zijn er nog de ‘private methods’. Deze zijn niet zichtbaar voor de gebruiker en daardoor ook niet door de gebruiker aan te roepen. Deze functies worden gebruikt om taken uit te voeren voor de openbare functies zoals onderdelen van de plant tekenen.

Public methods

Onder de openbare functies vallen functies zoals het aanmaken van een plant, het tekenen van het drie dimensionale model en het opslaan en laden van de plant. Deze functies worden aangestuurd door de gebruiker van het programma. Dit kan handmatig via het invoerscherm van MatLab zelf, en via de knoppen op de gebruikersinterface. Deze knoppen doen hetzelfde als de handmatig aangeroepen functies zonder dat de gebruiker de juiste commando’s hoeft te kennen.

Private methods

Deze methods zijn niet zichtbaar voor de gebruiker en daarmee ook niet aan te roepen door de gebruiker. Onder deze functies vallen bijvoorbeeld een functie die een bol op een knooppunt tekent en een functie die een blad in de aangegeven positie weergeeft. Dergelijke functies zijn nodig om het programma te laten werken, maar zijn niet nuttig om als gebruiker te kunnen aanroepen. Functies als deze worden door de openbare functies aangeroepen op het moment dat ze nodig zijn.

Graphical user interface

De 'graphical user interface' is de interface die de gebruiker te zien krijgt. De basisstructuur van de achterliggende code wordt automatisch door gegenereerd. Er wordt voor iedere knop een blok aangemaakt, alles binnen dit blok wordt uitgevoerd zodra er op de knop gedrukt is. Het aanroepen van de juiste functie bij de juiste knoppen is niet meer dan zorgen dat de juiste functieaanroep in ieder blok staat.

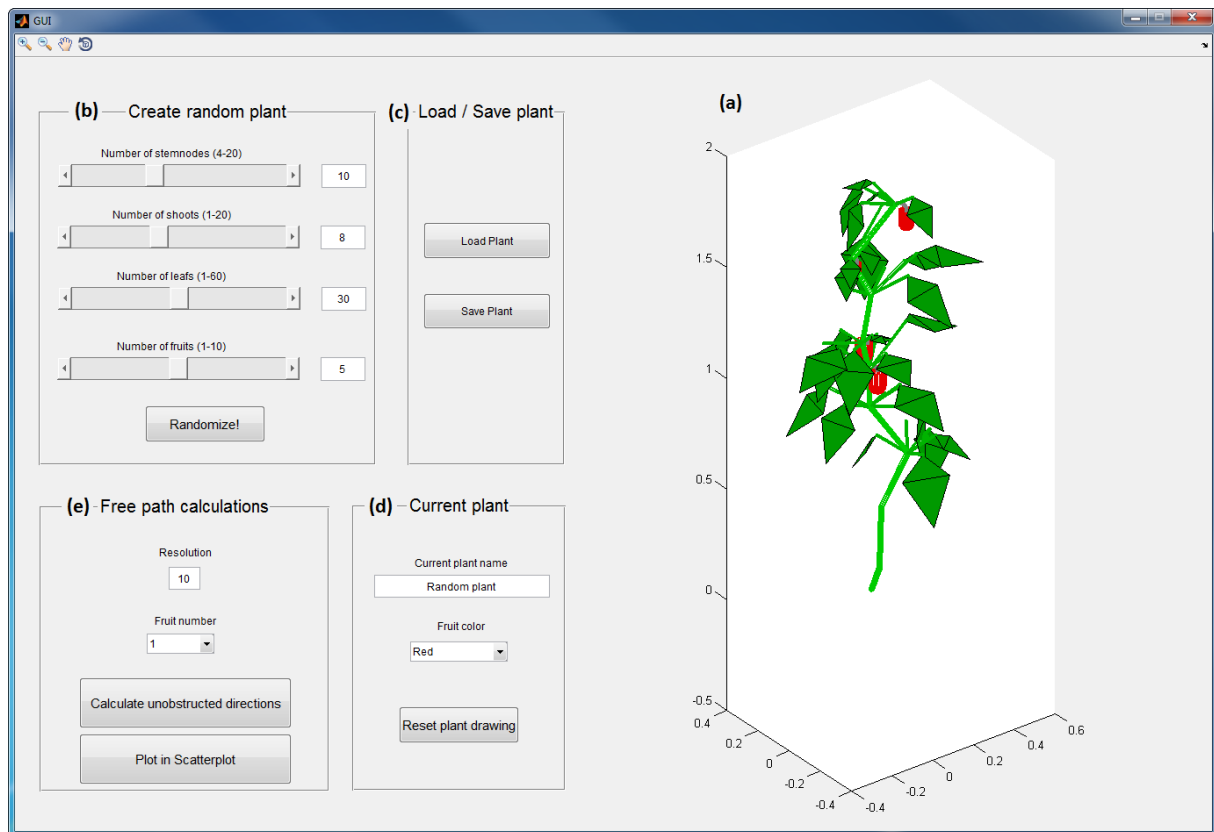
De gebruikersinterface is er uiteindelijk uit komen te zien als in Figuur 10. Rechts verschijnt het drie dimensionale model van de plant (a).

Linksboven is de besturing voor het genereren van een willekeuring plant te zien (b). Deze bestaat uit vier instellingen voor het aantal knooppunten op de stam, het aantal zijtakken, het aantal bladeren en het aantal paprika's. De tekstvakjes geven de stand van de schuiven aan in cijfers. Er kan daarin ook een waarde worden ingevoerd waarna de schuif naar de desbetreffende waarde springt, vooropgesteld dat deze binnen het bereik van de schuif ligt. Met de 'Randomize' knop wordt vervolgens een willekeurige plant gegenereerd en rechts weergegeven.

Naast de knoppen voor een willekeurige plant zitten de knoppen om een plant op te slaan en te laden (c). De 'Save' knop schrijft de openbare eigenschappen van de huidige plant naar een Excel bestand, de 'Load' knop laadt de openbare eigenschappen die in een Excel bestand staan in en genereert een plant met deze gegevens.

Daaronder staat een blok met extra functies (d). Het tekst vak geeft de naam van de weergegeven plant, zodat de gebruiker weet met welke plant hij of zij aan het werk is. Met het selectievakje daaronder kan de kleur van de paprika's worden ingesteld. Deze functie is uiteraard erg belangrijk, want het oog wil ook wat. Met de knop 'Reset plant drawing' kan de weergave van de huidige plant teruggezet worden naar de standaardweergave.

Linksonder staan dan nog de knoppen voor de analyse van de plant (e). Deze knoppen starten de berekening, waarvan de resultaten vervolgens visueel worden weergegeven in het model van de plant. Om te beginnen selecteert de gebruiker in het selectievakje de vrucht die hij of zij geanalyseerd wil hebben. In het tekstvakje kan de nauwkeurigheid worden ingevoerd voor de komende berekening. De ingevoerde nauwkeurigheid is het aantal punten dat per kwadrant wordt berekend. Hoe hoger de nauwkeurigheid, hoe nauwkeuriger de berekening. Dit duurt dan uiteraard ook langer. Met de knop 'Calculate unobstructed directions' worden de vrije zichtlijnen van de geselecteerde vrucht berekend met de ingevoerde nauwkeurigheid, meer hierover in het volgende hoofdstuk. De vrije zichtlijnen worden vervolgens in het model van de plant (a) weergegeven. De richtingen kunnen vervolgens nog worden weergegeven in een zogenaamde scatterplot door op de knop 'Plot in scatterplot' te drukken.



Figuur 10 Het resultaat, de grafische gebruikersinterface met daarin het model van de plant, (a) de drie dimensionele weergave van de huidige plant, (b) de knoppen voor het genereren van een willekeurige plant, (c) de knoppen voor het laden en opslaan, (d) knoppen voor de weergave van de huidige plant, (e) knoppen voor berekening en weergave van de vrije werkruimte

Bepalen vrije zichtlijnen

Gebruikte techniek

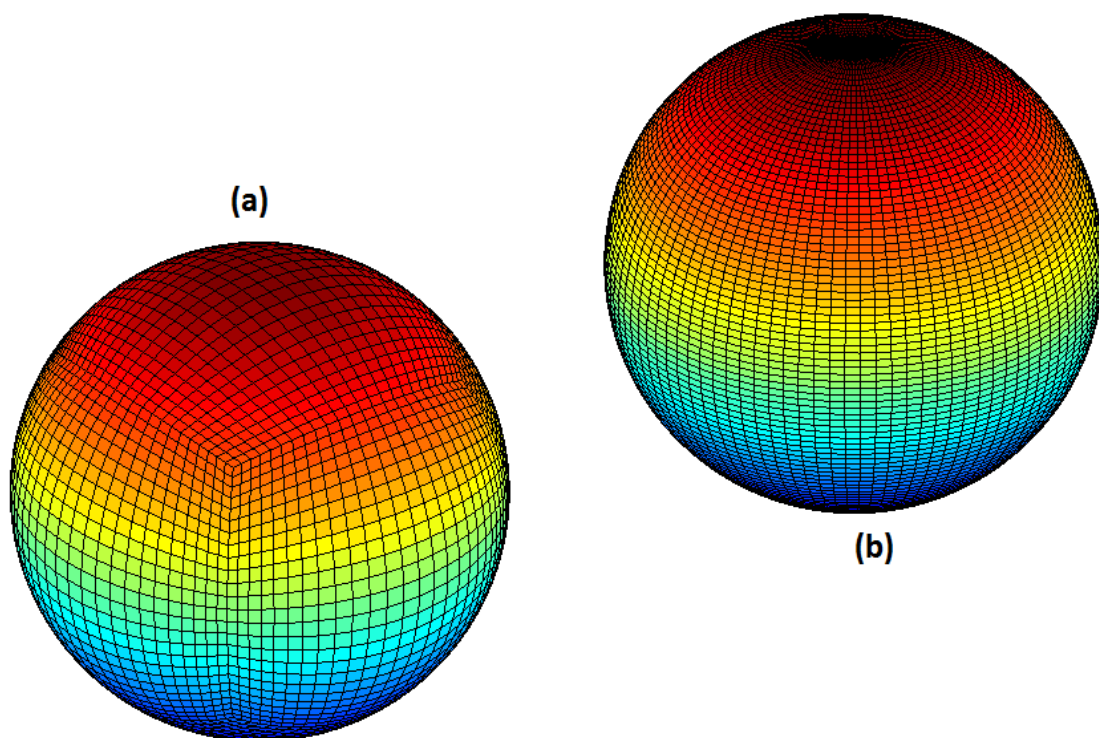
De techniek die uiteindelijk is gebruikt om de vrije richtingen te bepalen lijkt sterk op de werktuigoriëntatie correctie techniek (Wang et al., 2007) die eerder is beschreven.

Om te beginnen is de bol die in Figuur 6 te zien is nagemaakt. Vervolgens is er net als in het artikel naar ieder punt op deze bol een zichtlijn getrokken vanuit het middelpunt. Van al deze lijnen wordt vervolgens bepaald of dit een vrije zichtlijn betreft. Als het middelpunt van de bol op een paprika wordt geplaatst beschrijven deze lijnen een selectie van mogelijke zichtlijnen vanuit de paprika in alle richtingen. Als deze zichtlijnen niet worden geblokkeerd door een onderdeel van de plant, is de paprika langs die lijn zichtbaar vanaf iedere positie op die lijn buiten de plant.

Het verschil is dat de software gebruikt door Wang et al. (Wang et al., 2007) een serie rasterafbeeldingen produceert en vervolgens voor iedere pixel in deze afbeeldingen bepaalt of deze een vlak van het model raakt. MatLab heeft geen vergelijkbare functie en het model heeft, op de bladeren na, geen vergelijkbare oppervlakken. De detectie voor het al dan niet raken van een onderdeel van het model moet daarom anders worden geïmplementeerd.

Implementatie

Om te beginnen is een bol gemaakt. Deze is gemaakt door een kubus te projecteren op een bol. De reden om een dergelijke bol te gebruiken, en niet de standaard bol beschikbaar in MatLab, is de indeling van de bol. In Figuur 11 is het verschil te zien tussen de zelf geprogrammeerde bol (a), geïnspireerd op het beschreven artikel, en de standaard beschikbare bol (b).



Figuur 11 vergelijk tussen een kubus geprojecteerd op een bol (a) en een standaard bol (b). Let op het verschil in verdeling van de punten op de bol.

Het aantal vakjes rond de 'evenaar' van de bol is bij beiden gelijk, en daarmee is de nauwkeurigheid dus vrijwel gelijk. Er is duidelijk te zien dat bij de standaard bol de grootte van de vakjes afneemt naarmate men de 'polen' benadert. Dit betekent dat er rond de polen van de standaard bol heel veel meer berekeningen moeten worden uitgevoerd, met een nauwkeurigheid die veel hoger ligt dan de gewenste nauwkeurigheid. Bij de zelf geprogrammeerde bol zijn de punten veel evenrediger verdeeld. Dit geeft bij een ongeveer gelijke nauwkeurigheid een lagere rekentijd¹.

De bol die nu beschikbaar is, is opgebouwd uit punten. Vanuit het middelpunt van de bol kunnen er nu lijnen naar ieder van deze punten worden getrokken. Zoals beschreven wordt het middelpunt van deze bol op een paprika geplaatst. Vervolgens wordt voor iedere lijn bepaald of deze een onderdeel van de plant snijdt.

Obstructie door stamdelen

De plant is opgebouwd uit twee verschillende onderdelen: lijnstukken en polygonen. De stam en stelen zijn simpelweg lijnstukken beschreven door twee punten in de ruimte. Deze lijnstukken worden in het model weergegeven door middel van cilinders en de eindpunten door middel van bollen. Deze samen beschrijven een drie dimensionale ruimte rondom het lijnstuk waar ieder punt binnen valt dat een afstand tot het lijnstuk heeft die kleiner is dan de straal van dat onderdeel. Dit betekent dat wanneer de afstand tussen het lijnstuk dat de stamsectie of steel beschrijft en het lijnstuk dat een zichtlijn beschrijft minder is dan de straal van het betreffende onderdeel, dan zal de zichtlijn het onderdeel snijden. Om te bepalen of een zichtlijn een stam of een steel raakt moet de afstand tussen de twee genoemde lijnstukken worden bepaald. Is deze kleiner dan de straal, dan is de zichtlijn geblokkeerd en geen vrije richting.

Het bepalen van deze afstand gaat als volgt (Sunday, z. d.):

De eerste lijn, beschreven door de punten P_0 en P_1 kan worden geschreven als

$$P(s) = P_0 + s(P_1 - P_0) = P_0 + s\vec{u}$$

De tweede lijn, beschreven door de punten Q_0 en Q_1 kan worden geschreven als

$$Q(t) = Q_0 + t(Q_1 - Q_0) = Q_0 + t\vec{v}$$

De vector tussen twee punten op deze lijn kan worden geschreven als

$$\vec{w}(s, t) = P(s) - Q(t)$$

De kortste afstand tussen deze lijnen is de vector $\vec{w}_c = \vec{w}(s_c, t_c)$ tussen de punten $P(s_c)$ en $Q(t_c)$.

Deze vector staat per definitie loodrecht op de richtingsvector van beide lijnen, en is ook de enige vector die dit doet. Dit betekent dat de vector $\vec{w}_c$ moet voldoen aan

$$\vec{u} \cdot \vec{w}_c = 0 \text{ en } \vec{v} \cdot \vec{w}_c = 0.$$

Als we in deze vergelijkingen $\vec{w}_c$ schrijven als $\vec{w}_c = P(s_c) - Q(t_c) = \vec{w}_0 + s_c\vec{u} - t_c\vec{v}$ waarin $\vec{w}_0 = P_0 - Q_0$ krijgen we de vergelijkingen:

$$(\vec{u} \cdot \vec{u})s_c - (\vec{u} \cdot \vec{v})t_c = -\vec{u} \cdot \vec{w}_0$$

$$(\vec{v} \cdot \vec{u})s_c - (\vec{v} \cdot \vec{v})t_c = -\vec{v} \cdot \vec{w}_0$$

¹ :Bij x punten rond de evenaar bevat bol (a) $a = \left(\frac{x}{4}\right)^2 * 6 = \frac{6}{16}x^2$ vlakken en bol (b) $b = x^2$ vlakken.

Dit geeft $a = \frac{6}{16}b$, oftewel een besparing van factor $\frac{6}{16}$

Stel nu $a = \bar{u} \cdot \bar{u}$; $b = \bar{u} \cdot \bar{v}$; $c = \bar{v} \cdot \bar{v}$; $d = \bar{u} \cdot \bar{w}_0$ en $e = \bar{v} \cdot \bar{w}_0$. Dan kunnen deze vergelijkingen worden omgeschreven naar:

$$s_c = \frac{be - cd}{ac - b^2}$$

$$t_c = \frac{ae - bd}{ac - b^2}$$

Met deze twee vergelijkingen kunnen de punten op de twee lijnen gevonden worden waar deze het dichtst bij elkaar liggen. Wanneer $ac - b^2 = 0$ lopen de lijnen parallel en kan de oplossing gevonden worden door één punt vast te leggen en vervolgens de ander te berekenen.

Nu deze punten bekend zijn kan de afstand tussen deze punten berekend worden door middel van:

$$d(L_1, L_2) = |P(s_c) - Q(t_c)| = \left| (P_0 - Q_0) + \frac{(be - cd)\bar{u} - (ae - bd)\bar{v}}{ac - b^2} \right|$$

Zoals eerder gezegd, is deze afstand kleiner dan de straal van de stamsectie of steel dan zou de zichtlijn deze snijden en daardoor geblokkeerd worden. Deze berekening gaat echter uit van lijnen, geen lijnstukken. Nu moet er nog worden bepaald of de gevonden punten wel op het lijnstuk liggen dat is ingegeven, en niet in het verlengde hiervan. Dit is zo wanneer $0 \leq s_c \leq 1$ en $0 \leq t_c \leq 1$. Voldoen de gevonden punten aan deze voorwaarden, dan wordt de zichtlijn geblokkeerd.

Obstructie door bladeren

Voor de bladeren is een andere techniek nodig. Een blad is namelijk opgebouwd uit twee polygonen. In dit geval zijn deze polygonen driehoeken. Wanneer er een snijpunt bestaat tussen het lijnstuk dat de zichtlijn beschrijft en de driehoek die het blad beschrijft, dan is de zichtlijn geblokkeerd door deze driehoek. Dit houdt in dat wanneer een zichtlijn een van de twee driehoeken van het blad snijdt, dat deze zichtlijn door het blad wordt geblokkeerd.

De berekening hiervan gaat als volgt (Wikipedia, 2012):

De lijn, beschreven door de punten $l_a = (x_a, y_a, z_a)$ en $l_b = (x_b, y_b, z_b)$, kan worden geschreven als:

$$l_a + (l_b - l_a)t$$

Het vlak, beschreven door de punten $p_k = (x_k, y_k, z_k)$, met $k = 0, 1, 2$, kan worden geschreven als:

$$p_0 + (p_1 - p_0)u + (p_2 - p_0)v$$

Het snijpunt van de lijn en het vlak voldoet dan aan de vergelijking :

$$l_a + (l_b - l_a)t = p_0 + (p_1 - p_0)u + (p_2 - p_0)v$$

Welke kan worden omgeschreven naar:

$$l_a - p_0 = (l_b - l_a)t + (p_1 - p_0)u + (p_2 - p_0)v$$

In matrixvorm ziet deze vergelijking er als volgt uit:

$$\begin{bmatrix} x_a - x_0 \\ y_a - y_0 \\ z_a - z_0 \end{bmatrix} = \begin{bmatrix} x_a - x_b & x_1 - x_0 & x_2 - x_0 \\ y_a - y_b & y_1 - y_0 & y_2 - y_0 \\ z_a - z_b & z_1 - z_0 & z_2 - z_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix}$$

Deze kan worden opgelost door hem om te schrijven:

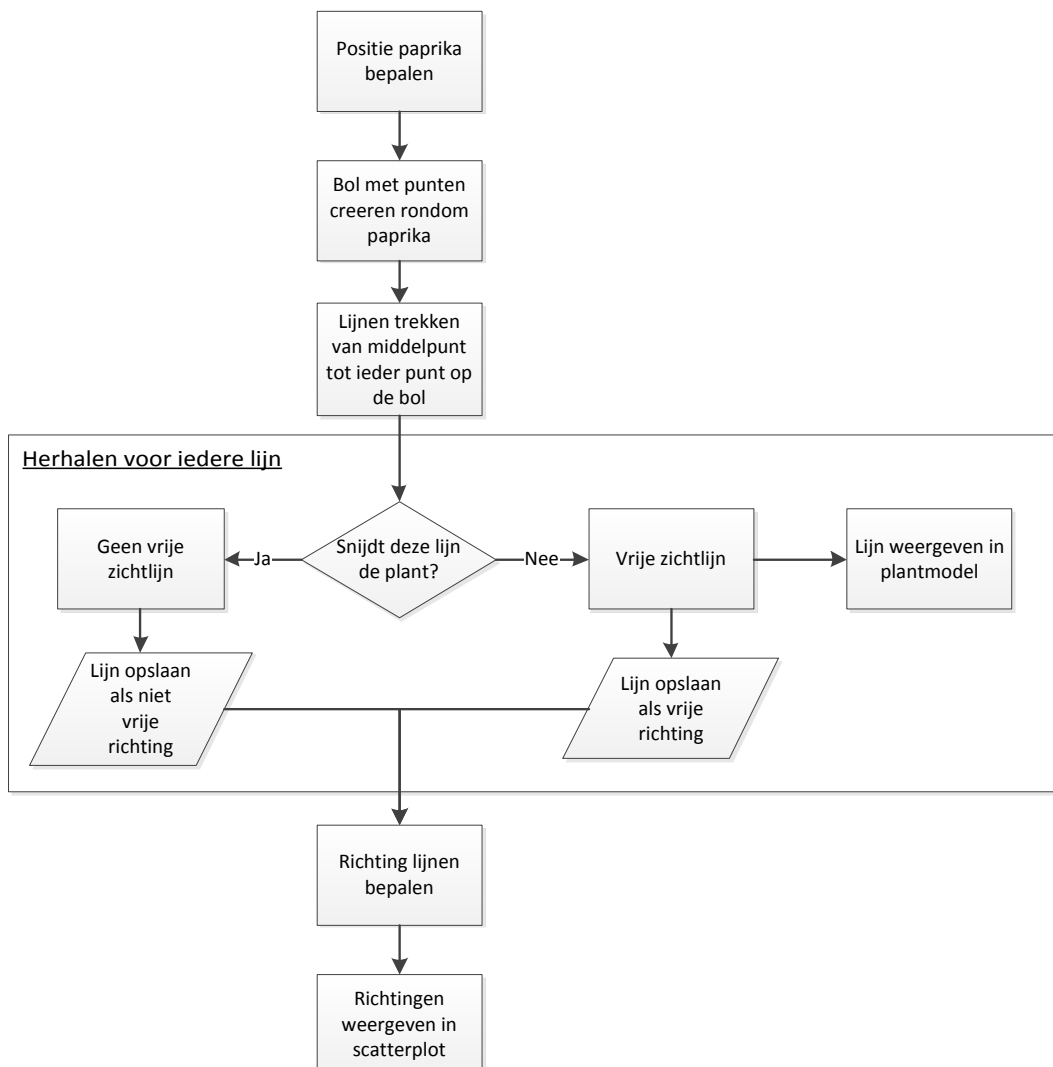
$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \begin{bmatrix} x_a - x_b & x_1 - x_0 & x_2 - x_0 \\ y_a - y_b & y_1 - y_0 & y_2 - y_0 \\ z_a - z_b & z_1 - z_0 & z_2 - z_0 \end{bmatrix}^{-1} \begin{bmatrix} x_a - x_0 \\ y_a - y_0 \\ z_a - z_0 \end{bmatrix}$$

Wanneer deze vergelijking niet is op te lossen lopen de lijn en het vlak parallel, en zullen ze nooit snijden. Is deze wel op te lossen, dan is het punt waar de lijn het vlak snijdt nu bekend. Net als bij de afstandsbepaling tussen twee lijnen gaat deze berekening uit kan een lijn, geen lijnstuk en van een vlak, geen polygoon. Om te bepalen of dit gevonden snijpunt op het lijnstuk en binnen de polygoon ligt moet de oplossing voldoen aan de voorwaarden $0 \leq t \leq 1$; $0 \leq u \leq 1$; $0 \leq v \leq 1$ en $(u + v) \leq 1$. Wanneer aan al deze voorwaarden voldaan wordt zal de zichtlijn het blad snijden.

Verwerken resultaat

Wanneer bepaald is of een zichtlijn iets raakt zijn er twee mogelijkheden. De lijn kan een vrije richting zijn en dus niets raken, dan wordt deze lijn getekend in het model en opgeslagen in een matrix met alle vrije zichtlijnen voor analyse. Als de lijn wel iets raakt en dus geen vrije richting is wordt deze niet getekend en opgeslagen in een matrix met alle andere niet vrije zichtlijnen.

In onderstaande flowchart, Figuur 12, is de werking van het algoritme nogmaals geïllustreerd.



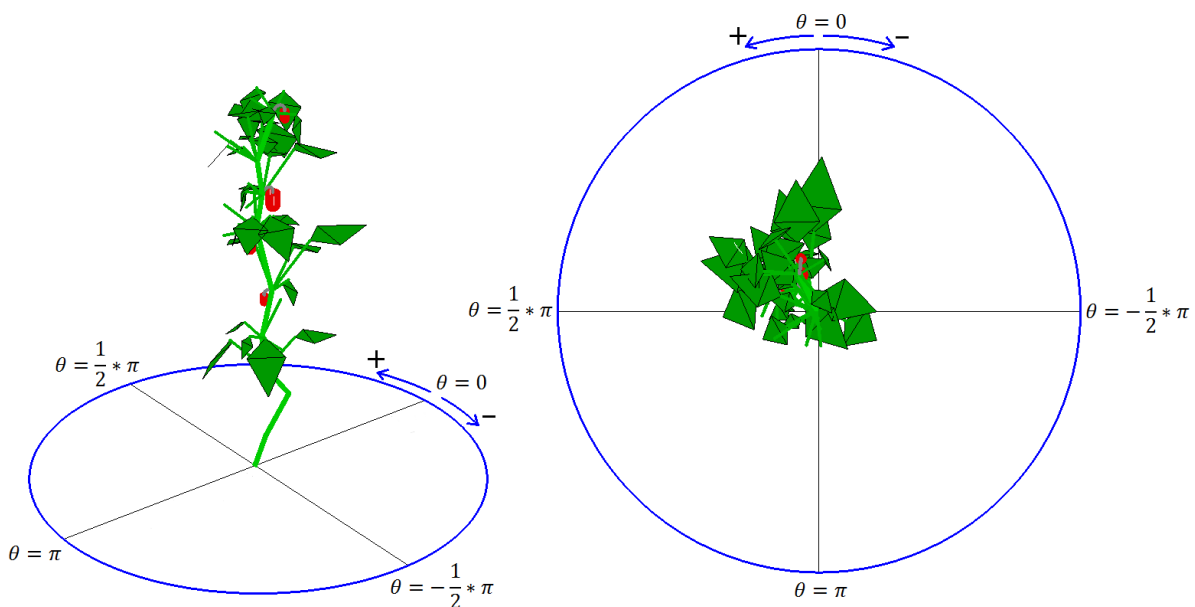
Figuur 12 Flowchart die de werking van het algoritme illustreert.

Analyse

Alle vrije zichtlijnen worden getekend in het model. Dit ziet er leuk uit en helpt een beeld te vormen hoe het resultaat eruit ziet. Het beeld is echter te onoverzichtelijk om conclusies uit te kunnen trekken. Daarom worden de resultaten verder verwerkt.

Van al deze opgeslagen lijnen kan de richting bepaald worden. Deze bestaat uit twee hoeken. Een hoek in het xy-vlak, genaamd Theta (θ). Deze heeft het bereik $-\pi < \theta < \pi$. Ook is er een hoek voor de hoogte genaamd Phi (φ). Deze heeft een bereik $-\frac{1}{2} * \pi < \varphi < \frac{1}{2} * \pi$. De hoek $\varphi = 0$ betekent dat de lijn horizontaal loopt. Een positieve waarde voor φ houdt in dat deze omhoog wijst, een negatieve waarde houdt in dat deze naar beneden wijst. Voor de hoek θ is de richting recht naar achteren vanaf de oogstzijde vastgesteld als $\theta = 0$ en de positieve richting is linksom.

De figuren van het plantmodel met de zichtlijnen geven twee aanzichten weer. Het standaard aanzicht voor een drie dimensionale weergave en daarnaast een bovenaanzicht. De richting van θ wordt geïllustreerd in Figuur 13.



Figuur 13 Grafische weergave van de indeling van de hoek θ in de gebruikte weergaven

Deze twee hoeken kunnen vervolgens tegen elkaar uitgezet worden in een scatterplot waarbij in een oogopslag de geschikte openingen te zien zijn waarlangs de vrucht benaderd kan worden. In het volgende hoofdstuk zullen deze figuren nog uitgebreid aan bod komen.

De genoemde scatterplot bevat voor ieder punt een waarde. De waarde van de lijnen is één ('waar'), of nul ('niet waar'). Wanneer een aantal van deze scatterplots bij elkaar opgeteld worden kunnen er wellicht trends worden waargenomen. Het zou kunnen zijn dat een bepaalde richting een hogere slagingskans heeft.

Resultaten

Een enkele plant

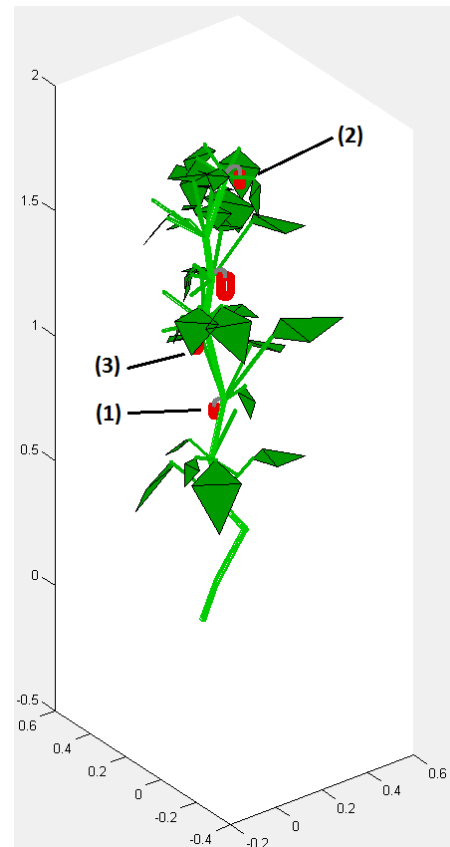
De plant

Eerst is er een analyse uitgevoerd met een enkele willekeurig gegenereerde plant. In Figuur 14 is deze plant te zien. Deze plant bevat tien knooppunten, acht zijtakken, dertig bladeren en vijf paprika's. Zoals in de figuur is te zien zit er mooie variatie in deze plant. De onderste paprika (aangeduid met (1) in Figuur 14) hangt bijvoorbeeld bijna helemaal vrij. Het zicht op deze paprika wordt bijna niet belemmerd door blad of andere onderdelen van de plant. De paprika bovenin (aangeduid met (2) in Figuur 14) hangt echter verscholen tussen de bladeren. Deze paprika is vanuit veel richtingen moeilijk waar te nemen. Deze variatie is interessant voor de analyse.

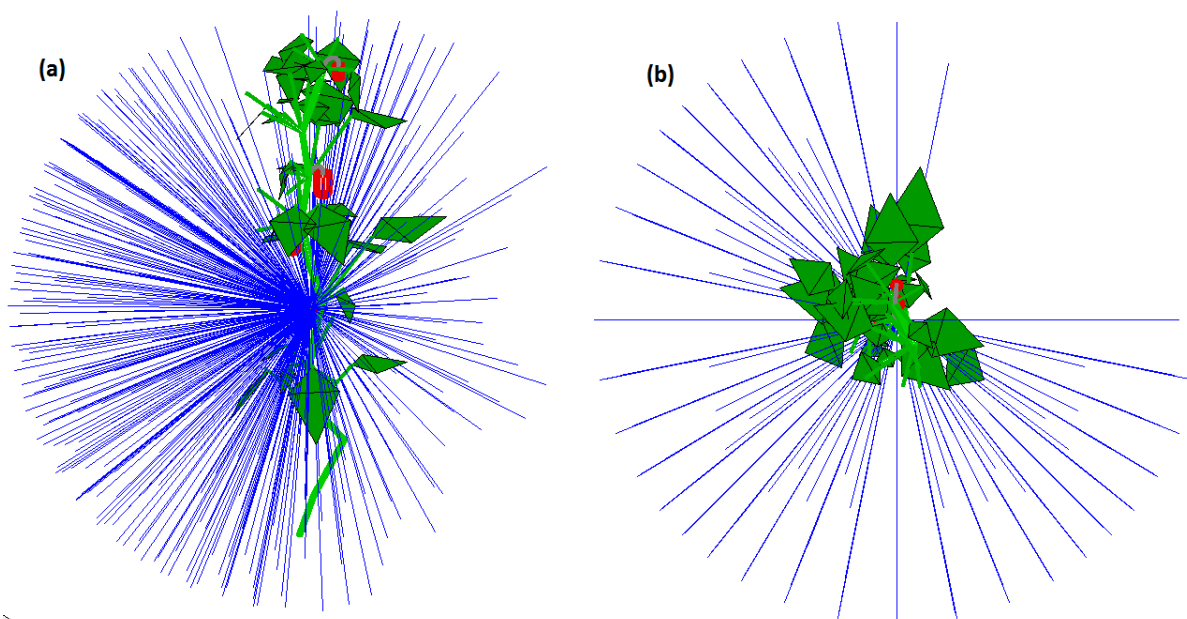
De volgende stap is om het programma uit te voeren en te zien of dat de resultaten overeenkomen met wat op het eerste gezicht verwacht wordt.

De zichtlijnen

Om te beginnen de onderste paprika (1). Zoals te zien in Figuur 15 kloppen de eerder besproken verwachtingen. De blauwe lijnen in de figuur zijn de zichtlijnen die vrij zijn. Aan de voorkant van de plant zijn vrijwel alle richtingen vrij, er lopen veel blauwe lijnen in deze richting. Zoals te zien in het bovenaanzicht zijn vrijwel alleen de zichtlijnen in de richting van de stam geblokkeerd, alleen daar ontbreken veel lijnen.

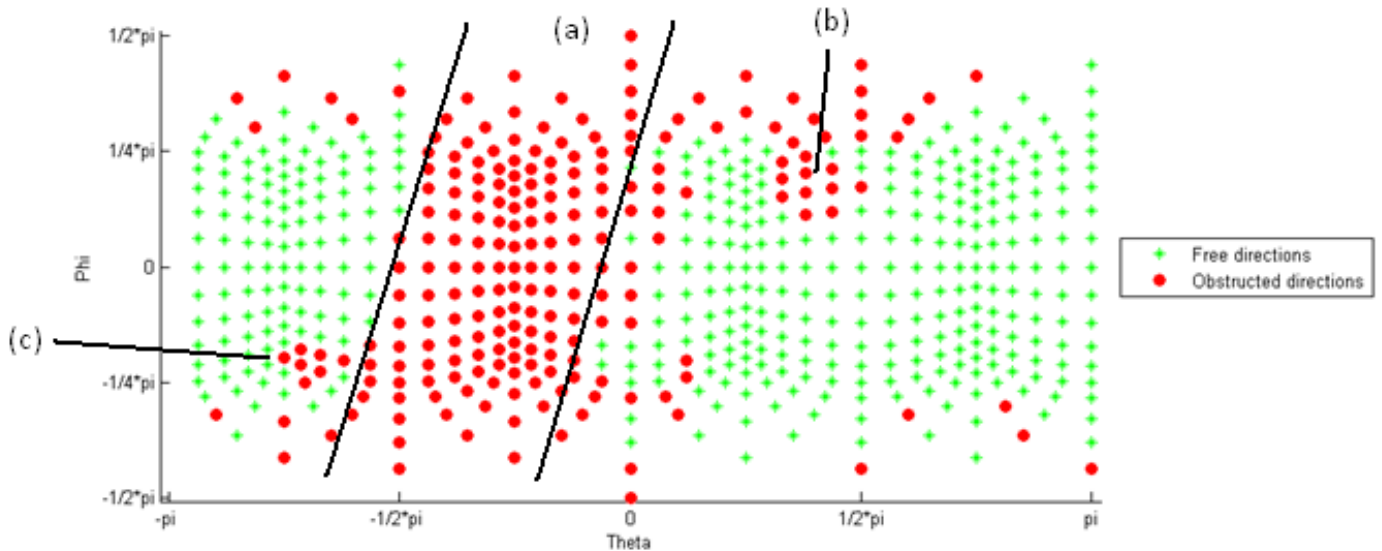


Figuur 14 De gebruikte plant, een willekeurig gegenereerde instantie van het plantmodel.



Figuur 15 De vrije zichtlijnen voor de paprika (1) in de plant. (a) zijaanzicht, (b) bovenaanzicht

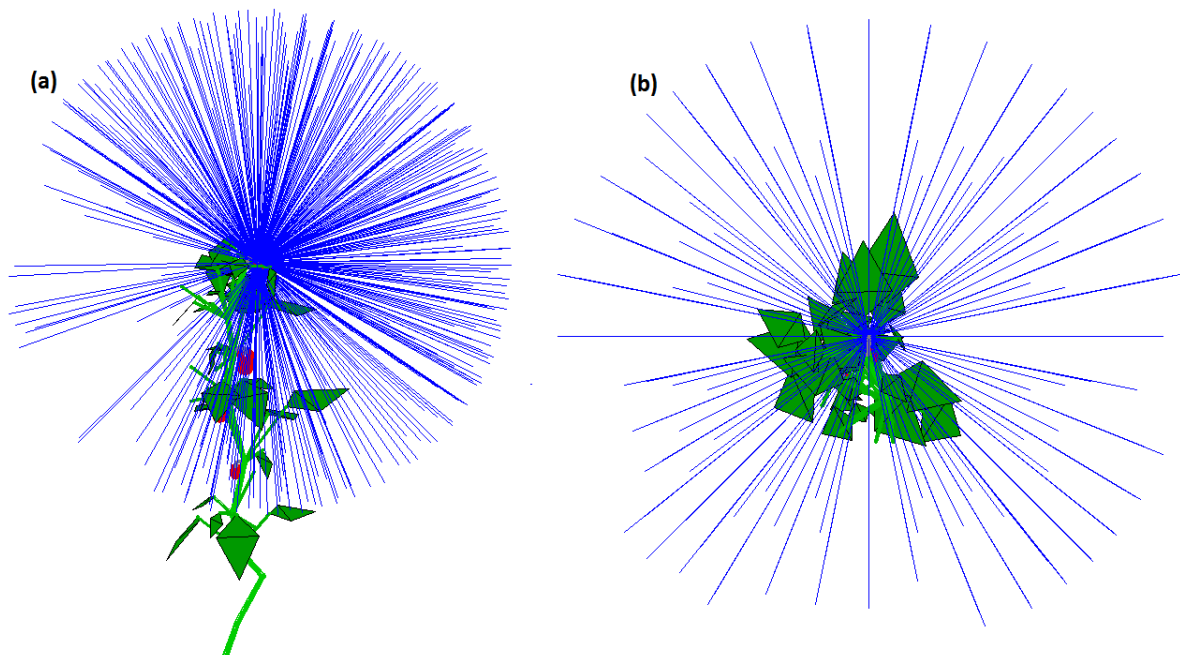
Om vervolgens een scatterplot van de richtingen te maken zoals eerder uitgelegd, is redelijk te voorspellen hoe deze eruit komt te zien. Zoals in het bovenaanzicht zichtbaar is, is er het kwadrant $-\frac{1}{2} * \pi < \theta < 0$ vrijwel geen enkele vrije zichtlijn. Dit zal ook in de scatterplot te zien zijn. Steil naar beneden en steil omhoog worden de zichtlijnen geblokkeerd door blad, daardoor zullen de uiterste waarden van φ ook weinig mogelijke richtingen bevatten. De rest van de scatterplot zal vrijwel geheel bedekt zijn met vrije zichtlijnen.



Figuur 16 Scatterplot van de zichtlijnen uit Figuur 15. Rode punten zijn geblokkeerde zichtlijnen, groene sterretjes zijn vrije zichtlijnen. De stengel van de plant (a), en twee bladeren (b) en (c) zijn herkenbaar.

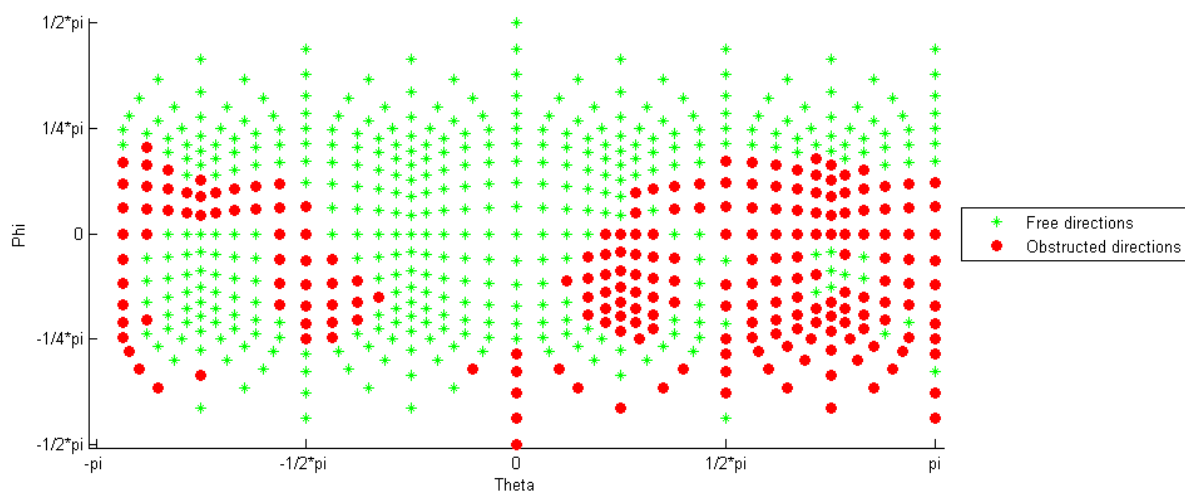
De verwachtingen blijken goed te kloppen. Tussen $-\frac{1}{2} * \pi < \theta < 0$ zijn er bijna geen vrije richtingen. Dit is de steel van de plant, aangegeven met (a) in Figuur 16. Bij de uiterste waarden van φ zijn ook weinig richtingen vrij. Wat nu nog meer te zien is, is een geblokkeerd gebied bij ongeveer $\frac{1}{4} * \pi < \theta < \frac{1}{2} * \pi$ en $\frac{1}{8} * \pi < \varphi < \frac{1}{4} * \pi$. Dit gebied, aangegeven met (b) in Figuur 16, zal waarschijnlijk een blad zijn dat in de afbeelding van de plant niet duidelijk zichtbaar is. Ook is er rond $\theta \approx -\frac{3}{4} * \pi$ en een $\varphi \approx -\frac{1}{4} * \pi$ een gebied geblokkeerd, aangegeven met (c) in Figuur 16. Dit is het blad dat midden vooraan in Figuur 14, onderaan de plant te zien is.

Een ander geval is de paprika (2) bovenin de plant. Bij deze paprika lijken er minder mogelijke zichtlijnen te zijn, omdat deze vrucht verscholen zit tussen een groot aantal bladeren. De bovenkant is echter wel grotendeels vrij, daar zullen nog wel mogelijkheden zijn.



Figuur 17 De vrije zichtlijnen voor de paprika (2) in Plant 1. (a) zijaanzicht, (b) bovenaanzicht

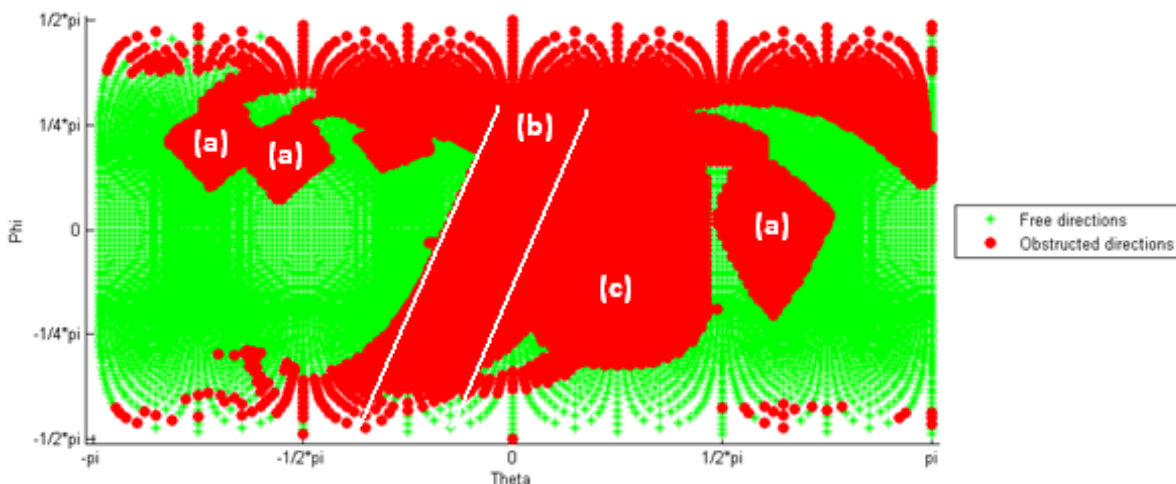
Er zijn in dit geval toch meer richtingen vrij dan verwacht. De meeste bladeren blijken lager te hangen dan de paprika. Van de zichtlijnen die omlaag wijzen zijn er een aantal geblokkeerd door de bladeren. De lijnen die omhoog wijzen zijn bijna allemaal wel vrij. We verwachten in de scatterplot in de bovenste helft veel vrije richtingen, terwijl de onderste helft het grootste deel van de geblokkeerde richtingen bevat. In het bovenaanzicht zien we nog dat het gedeelte met bij $-\frac{1}{2} * \pi < \theta < \pi$ toch een aantal richtingen mist.



Figuur 18 Scatterplot van de zichtlijnen uit Figuur 17 Rode punten zijn geblokkeerde zichtlijnen, groene sterretjes zijn vrije zichtlijnen.

In deze scatterplot is te zien waarom de verwachtingen niet helemaal juist waren. De θ van de kijkrichting in de afbeelding van de plant is $\theta = -\frac{2}{3} * \pi$. In die richting zijn er een aantal zichtlijnen geblokkeerd. Vanuit deze richting zijn ook achter de paprika veel richtingen geblokkeerd. Aan de andere zijden zijn er echter nog veel richtingen vrij, maar dat was niet zichtbaar in de afbeelding. Wel volgens de verwachtingen zijn de richtingen met een $\varphi > \frac{1}{4} * \pi$ allemaal vrij. Dit komt doordat de paprika helemaal bovenin hangt en er dus geen delen van de plant meer boven zitten om het zicht te blokkeren.

Interessant om op te merken is dat er in deze afbeelding wederom een aantal bladvormen te herkennen zijn. Deze worden nog duidelijker bij een scan met een hogere nauwkeurigheid zoals in Figuur 19, gemaakt van de paprika aangegeven met (3). De voorgaande scatterplots zijn berekend met een nauwkeurigheid van tien punten per kwadrant, voor Figuur 19 is een nauwkeurigheid van vijftig punten per kwadrant gebruikt. Dit zorgt voor een behoorlijk langere rekentijd, maar resulteert in een interessante afbeelding. In deze afbeelding zijn de plantonderdelen zo duidelijk te herkennen dat het model niet eens nodig is om te herkennen wat de zichtlijnen blokkeert.

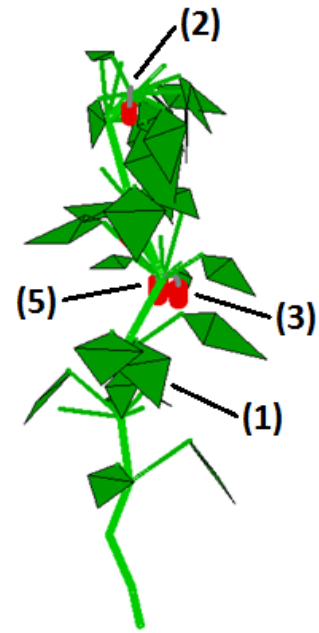


Figuur 19 Een scatterplot van zichtlijnen uitgevoerd in een hogere nauwkeurigheid. Bladeren van de plant (a), de stengel van de plant (b) en een andere paprika (c) zijn herkenbaar.

Deze afbeelding is gemaakt van de paprika die in Figuur 14 half verscholen als op een na onderste paprika te zien is, aangegeven met (3). Bovenin de scatterplot is een aantal bladeren zichtbaar, aangegeven met (a). Ook zien we duidelijk de stengel langs komen vanaf ongeveer $(-\frac{1}{2} * \pi; -\frac{1}{2} * \pi)$ schuin naar boven, aangegeven met (b). Bij $-\pi < \theta < -\frac{1}{2} * \pi$ zien we bovenin beeld een rood vlak in de vorm van bladeren, veroorzaakt door het blad dat de paprika voor de kijker verschuilt in Figuur 14. Het opvallendste in deze afbeelding is wel het grote rode vlak bij $0 < \theta < \frac{1}{2} * \pi$, aangegeven met (c). Volgens de beschrijving van de plant in het begin van dit hoofdstuk hangen er vijf paprika's aan deze plant. In Figuur 14 zijn er echter maar vier te zien. Nummer vijf is hier zichtbaar: die hangt achter de paprika die hier wordt beschreven, volledig verscholen achter blad.

Om deze bevindingen te bevestigen wordt de plant nog eens bekeken vanuit de richting $\theta = 0$, zie Figuur 20. Daarbij worden de bevindingen bevestigd:

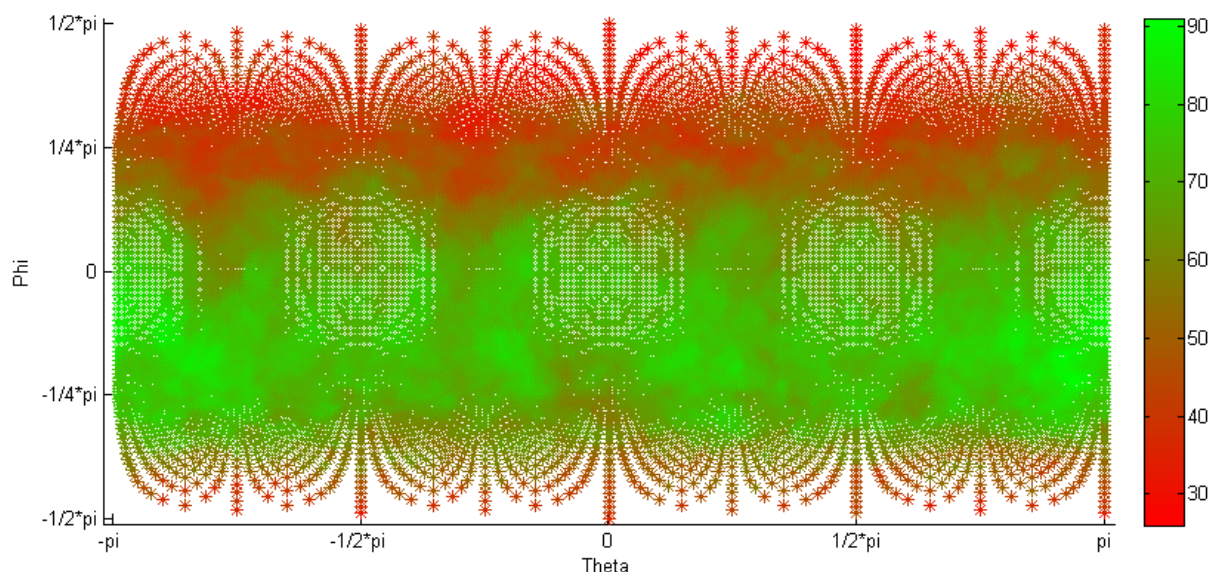
- De onderste paprika (1) is niet zichtbaar vanaf dit gezichtspunt
- De bovenste paprika (2) hangt eigenlijk grotendeels vrij
- De vijfde paprika (5) hangt naast de laatst besproken (3)



Figuur 20 Dezelfde plant bekeken vanaf $\theta = 0$. Dezelfde nummering is gebruikt voor de paprika's (1), (2) en (3). (5) was tot nog toe niet zichtbaar

Meerdere planten

Voor een enkele paprika zijn de gegevens duidelijk te interpreteren. Deze gegevens zijn echter niet toe te passen op andere paprika's dan de paprika waarvoor de berekening is gedaan. Om de gegevens toe te passen op meerdere paprika's moet er worden gezocht naar gemeenschappelijke uitkomsten. Om dit te doen is het volgende bedacht. Elke scatterplot gemaakt van een enkele paprika bestaat uit een serie punten met een waarde nul of één. Deze waarden staan voor wel of niet geblokkeerd in die bepaalde richting. Door nu de scatterplots van meerdere paprika's bij elkaar op te tellen kan er worden bekeken of er zich bepaalde trends voordoen. Figuur 21 is een dergelijke afbeelding. Dit zijn de scatterplots van 210 willekeurige paprika's bij elkaar opgeteld. Hiervoor zijn 21 willekeurige planten gegenereerd met elk tien knooppunten, acht zijtakken, veertig bladeren en tien paprika's.



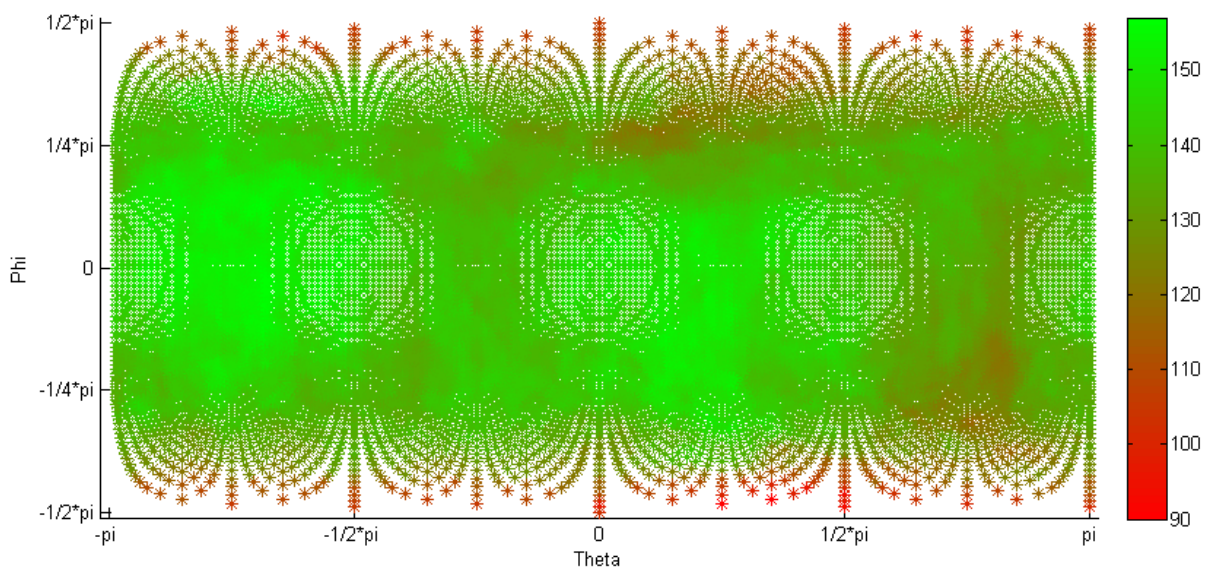
Figuur 21 210 scatterplots van willekeurige paprika's aan willekeurige planten bij elkaar opgeteld

De kleurstelling is net als bij de andere figuren maar nu met gradaties. Groen betekent een richting die vaak vrij is, rood een richting die vaak geblokkeerd is. Er is goed te zien dat de verdeling in de horizontale richting uniform verdeeld is. Er is dus niet een bepaalde richting θ die een betere kans geeft op het vinden van de paprika dan een andere. Dit is logisch te verklaren. De paprika kan namelijk niet door de stengel heen geplukt worden, en die bevindt zich in een willekeurige richting. Ook de richting θ van een blad, de andere grote invloed bij het blokkeren van zichtlijnen, is volledig willekeurig. De horizontale verdeling in Figuur 21 is daardoor volledig willekeurig.

De verdeling voor φ daarentegen geeft een duidelijke zonering weer. Richtingen $\varphi > \frac{1}{4} * \pi$ zijn vaak geblokkeerd. Richtingen $0 < \varphi < \frac{1}{4} * \pi$ en richtingen $\varphi < -\frac{1}{4} * \pi$ vertonen een verloop. Hoe verder van de nullijn, hoe meer kans dat de richting is geblokkeerd. Het gebied $-\frac{1}{4} * \pi < \varphi < 0$ heeft een opvallend hogere kans op succes dan daarbuiten. Recht naar boven en recht naar onderen, de uiterste waarden voor φ , zitten waarschijnlijk delen van de stam en de stengels van de verschillende onderdelen in de weg.

Dit verklaart echter alleen de uiterste waarden, niet het verloop in de gehele bovenste helft. De oorzaak hiervan is waarschijnlijk het blad aan de plant. De helling van het blad is namelijk niet volledig willekeurig. De hoek tussen een steel en het blad is, zoals beschreven bij het plantontwerp, vast. De hoek tussen de steel van het blad en de stam is wel willekeurig, maar blijft tussen de grenzen $0 < \varphi < \frac{1}{2} * \pi$. Dit geeft als gevolg dat een blad altijd naar beneden wijst met een hoek tussen $-\frac{2}{3} * \pi < \varphi < 0$. Een zichtlijn met een positieve φ , een zichtlijn die omhoog wijst, zal op het oppervlak van het blad terecht komen. Dan wordt er een hele bundel zichtlijnen geblokkeerd door een enkel blad. Een zichtlijn met een negatieve φ , die naar beneden wijst, zal een grotere kans hebben om ongeveer dezelfde helling te hebben als het blad. Een blad heeft geen dikte, dus een dergelijke lijn zal gewoon langs het oppervlak op gaan. Figuur 19 geeft hiervan een mooi voorbeeld. In het gebied $\varphi > 0$ zijn vlakken in de vorm van bladeren geblokkeerd. Onder deze paprika zitten ook bladeren, maar deze bedekken slechts enkele lijnen.

Om deze theorie te toetsen is dezelfde simulatie uitgevoerd met wederom 210 willekeurige paprika's, maar in dit geval hebben de planten waar deze aan hangen geen bladeren. De rest is gelijk gebleven: 21 planten met elk tien knooppunten, acht zijtakken, nul bladeren en tien paprika's. Dit geeft de volgende afbeelding.



Figuur 22 210 scatterplots zij elkaar opgeteld van willekeurige paprika's aan willekeurige planten zonder blad

De uiterste waarden voor φ zijn nog steeds vaak geblokkeerd. Dit kwam door de stam en de stelen van de verschillende onderdelen. De stam is er nog steeds, dus deze blokkeert nog steeds richtingen. Toch zijn deze richtingen beter bereikbaar dan voorheen, waarschijnlijk door het ontbreken van de bladstelen. Het gebied tussen de uiterste waarden is nu wel volledig willekeurig verdeeld. De slechtere bereikbaarheid bij positieve waarden van φ is nu weg. De verklaring dat de bladeren deze richtingen blokkeren lijkt te kloppen. In de schaalverdeling is te zien dat alle richtingen veel beter bereikbaar zijn door het ontbreken van blad.

Discussie

Een model van een plant is natuurlijk geen echte plant. Aangezien alle planten die voor dit rapport gebruikt zijn willekeurig gegenereerd zijn, zijn het niet eens modellen van een echte plant. Voor de betrouwbaarheid van de resultaten zou een echte plant beter zijn.

De robot die de paprika's uiteindelijk gaat plukken heeft een zeer flexibele manipulator. Deze kan veel meer verschillende houdingen aannemen dan alleen een rechte lijn. Het berekenen van de mogelijke richtingen om de vrucht te bereiken is in de werkelijkheid niet beperkt tot rechte lijnen. In dit rapport is niet verder gekeken dan rechte lijnen vanwege de beschikbare tijd. Een zichtlijn kan echter geen bochten bevatten, dus voor het vinden van de paprika zijn rechte lijnen wel van belang.

Een belangrijk punt dat makkelijk uit te breiden is, is het feit dat de planten in de kas opgesteld staan in rijen. Dit levert beperkingen op voor de mogelijke richtingen omdat de planten elkaar kunnen afschermen. Dit is makkelijk toe te voegen door meerdere planten naast elkaar te modelleren.

In het onderzoek van Jos Ruizendaal (Ruizendaal, 2011) is bepaald vanuit welke positie een camera de meeste paprika's in een rij paprikaplanten kan vinden. De verschillende camerapositionen zijn niet een op een te vergelijken met de resultaten van dit rapport. De camera is echter ook onder verschillende hoeken gezet. De hoek van de camera kan vergeleken worden met de hoeken φ en θ beschreven in dit rapport. De hoek van de camera vergelijkbaar met θ kan ook maar ten dele worden vergeleken omdat Ruizendaal heeft gekeken naar een rij paprikaplanten. Dit beïnvloedt het resultaat te sterk. Voor de hoek van de camera vergelijkbaar met φ is het resultaat wel degelijk interessant. Ruizendaal stelt dat de meeste efficiënte methode om paprika's te vinden is, om 3 foto's te maken vanuit $\varphi = 0$ en twee foto's vanuit $\varphi = -\frac{1}{6} * \pi$. Deze waarden liggen beiden in het bereik wat in het model van dit rapport de beste resultaten opleverde. Dit lijkt de resultaten van het model te bevestigen.

Conclusie

De methode om zichtlijnen te berekenen en te bepalen of deze geblokkeerd wordt geeft een goed beeld van de zichtbaarheid/benaderbaarheid van de paprika's. Een snelle vergelijking met de resultaten van Ruizendaal (Ruizendaal, 2011) geeft een indicatie dat de resultaten van de benadering overeenkomen met de werkelijkheid.

Op grond van de berekeningen aan virtuele modelplanten kan worden geconcludeerd dat:

1. De paprika dient niet vanuit de richting van de stengel benaderd te worden.
2. De hellingshoek van de bladeren dient gevolgd te worden.

In acht moet worden genomen dat niet aan echte planten is gerekend in een echte omgeving. Toch biedt deze methode de mogelijkheid om zichtbaarheid/benaderbaarheid te kwantificeren.

Aanbevelingen

Er zou eens in een kas een kort onderzoek gedaan kunnen worden naar de variatie in het aantal bladeren en vruchten aan een plant en de variatie in de afmetingen van deze onderdelen. Dit zou een nauwkeuriger willekeurig model opleveren dan het huidige model. Het zou ook een goede stap zijn om een echte plant na te meten en daar vervolgens een model van te genereren met het ontwikkelde programma. Het model dat ontstaat, kan dan vergeleken worden met het origineel om de nauwkeurigheid van het model te toetsen.

Als blijkt dat het model inderdaad in grote mate overeenkomt met de werkelijkheid kan er een serie planten opgemeten worden en daarvan modellen gemaakt worden. Dit zou dan een realistische simulatie opleveren en daarmee zijn de uitkomsten die dan gevonden worden ook in zekere mate bruikbaar in de werkelijkheid om de paprika's in de kas te vinden.

Een volgende stap in het vinden van mogelijke plukrichtingen is het zoeken naar paden die niet alleen door rechte lijnen beschreven worden. Dat zal meer richtingen opleveren.

Zoals genoemd bij de literatuur geven visibility cones nieuwe mogelijkheden bij het vinden van richtingen om een paprika te benaderen. Deze voegen het ruimtelijke element toe waarmee rekening gehouden dient te worden wanneer de route voor de robot uitgezet wordt. Dit zal in tegenstelling tot het voorgaande de mogelijke richtingen beperken, omdat een visibility cone meer ruimte nodig heeft dan een lijn. De beperking is echter nuttig, aangezien alle bruikbare richtingen blijven bestaan en alleen de onbruikbare richtingen zullen verdwijnen, omdat ook een robot meer ruimte nodig heeft dan een lijn.

Tenslotte zijn de resultaten beschreven in dit rapport slechts kwalitatieve resultaten. Uit de gegevens komen plaatjes die men kan bekijken en daarmee een goede schatting kan maken van de meest geschikte richtingen. Uit deze plots zou echter ook de meest geschikte richting kunnen worden berekend. Bijvoorbeeld door het grootste oppervlak van vrije richtingen te bepalen. Dit zou ook een vervolgstap zijn die nuttig is om naar te kijken. Een computer of robot kan met de huidige uitkomsten namelijk nog niets beginnen zonder menselijke analyse van de data.

Literatuur

AGF (2010). Bezocht op 19-04-2012 http://www.agf.nl/nieuwsbericht_detail.asp?id=63720

Bac, W. (2011). "Description of a plant drawing." Niet gepubliceerd

Dijk, van (z. d.). Bezocht op 19-04-2012

<http://www.vandijkbedrijven.nl/consumenten.asp?taalID=2&submenuID=12>

Keeler, T., J. Fedorkiw, et al. (2007). "The spherical visibility map." Computer-Aided Design **39**(1): 17-26.

Kim, D. S., P. Y. Papalambros, et al. (1995). "Tangent, Normal, and Visibility Cones on Bezier Surfaces." Computer Aided Geometric Design **12**(3): 305-320.

LEI (2011). "Landbouw-Economisch Bericht" <http://www.lei.dlo.nl/publicaties/PDF/2011/2011-017.pdf>

Liu, M., Y. S. Liu, et al. (2009). "Computing global visibility maps for regions on the boundaries of polyhedra using Minkowski sums." Computer-Aided Design **41**(9): 668-680.

McGarrrity, S. (2009). "Introduction to Object-Oriented Programming in MATLAB" Matlab Digest <http://www.mathworks.nl/company/newsletters/articles/introduction-to-object-oriented-programming-in-matlab.html>

MidFresh (z. d.). Bezocht op 19-04-2012 <http://www.midfresh.nl/23/De-Paprika>

Milieu- en Natuurplanbureau(MNP) (2007). "Duurzame ontwikkeling van de landbouw in cijfers en ambities" <http://www.rivm.nl/bibliotheek/rapporten/500139002.pdf>

Ruizendaal, J. (2011). "Fruit visibility analysis for different camera positions in sweet peppers" Niet gepubliceerd

Sunday, D. (z. d.). "Distance between Lines and Segments with their Closest Point of Approach" http://softsurfer.com/Archive/algorithm_0106/algorithm_0106.htm

Wang, Q. H., J. R. Li, et al. (2006). "Graphics-assisted approach to rapid collision detection for multi-axis machining." International Journal of Advanced Manufacturing Technology **30**(9-10): 853-863.

Wikipedia (2012). Bezocht op 3-6-2012 http://en.wikipedia.org/wiki/Line-plane_intersection

Woo, T. C. (1994). "Visibility Maps and Spherical Algorithms." Computer-Aided Design **26**(1): 6-16.

Yin, Z. P., H. Ding, et al. (2000). "Visibility theory and algorithms with application to manufacturing processes." International Journal of Production Research **38**(13): 2891-2909.

Zetu, D. and A. Akgunduz (2005). "Shape recovery and viewpoint planning for reverse engineering." International Journal of Advanced Manufacturing Technology **26**(11-12): 1370-1378.

Bijlages

Bijlage 1, code 'Plant.m'

```
classdef Plant
    %A bell pepper plant
    % The class definition of an object describing a bell pepper plant.
    %
    % Properties:
    %     NoStemnodes    Number of stemnodes
    %     NoShoots       Number of Shoots
    %     NoLeafs        Number of Leafs
    %     NoFruits       Number of Fruits
    %     Stemnodes      Properties of the Plants stemnodes
    %     Shoots         Properties of the Plants Shoots
    %     Leafs          Properties of the Plants Leafs
    %     Fruits         Properties of the Plants Fruits
    %
    % For detailed help on a property type help Plant.YourProperty, with
    % YourProperty being the desired property
    %
    % Available methods:
    %     Plant          Create an instance of a Plant class object
    %     Draw           Draw a 3D model of the given Plant
    %     SavePlant      Save the properties of the given Plant to an Excel
    %                   file
    %     LoadPlant      Load the properties for a new Plant from an Excel
    %                   file
    %     Directions     Calculates all unobstructed sightlines from the
    %                   indicated fruit and draws those lines in the model
    %     ScatterPlot    Creates a scatterplot of the directions
    %                   calculated in the method Directions
    %
    % For detailed help on a method type help Plant.YourMethod, with
    % YourMethod being the desired method
    %

    properties
        NoStemnodes    %Number of Stemnodes on the Plant; single integer value
        NoShoots       %Number of Shoots on the Plant; single integer value
        NoLeafs        %Number of Leafs on the Plant; single integer value
        NoFruits       %Number of Fruits on the Plant; single integer value
        Stemnodes      %Locations of the nodes of the stem
        %Format: a row [x y z] for each node
        % x, y and z are node coordinates in [m]
        Shoots         %Properties of the Plants shoots
        %Format: a row [NodeNr phi theta length] for each shoot
        % NodeNr is the number of the node where the shoot attaches to the stem
        % phi is the horizontal angle in [rad]
        % theta is the vertical angle in [rad]
        % length is the shoot length in [m]
        Leafs          %Properties of Plants Leafs
        %Format: a row [NodeNr phi theta length] for each leaf
        % NodeNr is the number of the node where the leaf attaches to the stem
        % phi is the horizontal angle in [rad]
        % theta is the vertical angle in [rad]
        % length is the leaf length in [m]
        Fruits         %Properties of Plants Fruits
        %Format: a row [NodeNr phi theta length size] for each fruit
        % NodeNr is the number of the node where the fruit attaches to the stem
        % phi is the horizontal angle in [rad]
        % theta is the vertical angle in [rad]
        % length is the peduncle length in [m]
        % size is the diameter of the fruit in [m]
    end

    properties (Constant,GetAccess=private)
        Stemdiameter=.02;    %Diameter of the stem in [m]
        Shootdiameter=.01;    %Diameter of the shoots in [m]
        Pedunclediameter=.012; %Diameter of the peduncles in [m]
        Stemcolor=[0 .8 0];   %Color of the stem
    end
end
```

```

Shootcolor=[0 .7 0];           %Color of the shoots
Leafcolor=[0 .6 0];           %Color of the leafs
Pedunclecolor=[.5 .5 .5];      %Color of the peduncles
FruitcolorRed=[.9 0 0];        %Color of the fruits when red
FruitcolorOrange=[.9 .45 0];   %Color of the fruits when orange
FruitcolorYellow=[.9 .9 0];    %Color of the fruits when yellow
FruitcolorGreen=[0 .5 0];      %Color of the fruits when green
end
properties (Dependent,GetAccess=private)
    Shootnodes %Locations of nodes of the Shoots; x,y,z
    LeafShootnodes %Locations of nodes of the LeafShoots; x,y,z
    PeduncleNodes %Locations of nodes of the Peduncles; x,y,z
    Fruitnodes %Locations of nodes of the Fruits; x,y,z
end

methods
    function obj=Plant(NoStemnodes,NoShoots,NoLeafs,NoFruits,Stemnodes,Shoots,Leafs,Fruits)
        % PLANT Creates a new 'Plant-class' object. This object will describe
        % a model of a bell pepper plant.
        %
        % Inputs:
        %   NoStemnodes
        %   NoShoots
        %   NoLeafs
        %   NoFruits
        % Optional inputs (if not defined, random values will be generated):
        %   Stemnodes
        %   Shoots
        %   Leafs
        %   Fruits
        %
        % For detailed help on a input property, type help Plant.YourProperty
        % with YourProperty being the desired property
        %
        % Example:
        %
        %   Create an object 'NewPlant' of the class Plant with 10 stemnodes, 5
        %   shoots, 26 leafs and 3 bell peppers; all of which will be
        %   randomly placed:
        %
        %   NewPlant=Plant(10,5,26,3)
        %
        obj.NoStemnodes=NoStemnodes;           %set Plant properties
        obj.NoShoots=NoShoots;
        obj.NoLeafs=NoLeafs;
        obj.NoFruits=NoFruits;
        TF=exist ('Stemnodes','var');
        if TF==1                               %check if properties are provided,
            obj.Stemnodes=Stemnodes;           %if not, create random properties.
        else
            Stemnodes=(rand(NoStemnodes,3))/5;
            Stemnodes(1,3)=0;
            for i=2:size(Stemnodes,1)
                Stemnodes(i,3)=Stemnodes(i-1,3)+rand/6+0.1;
            end
            obj.Stemnodes=Stemnodes;
        end
        TF=exist ('Shoots','var');
        if TF==1
            obj.Shoots=Shoots;
        elseif NoShoots == 0
            obj.Shoots=[];
        else
            Shoots=zeros(NoShoots,4);
            Shoots(:,1)=randi(NoStemnodes-3,size(Shoots,1),1)+3;
            Shoots(:,2)=rand(size(Shoots,1),1)*2*pi;
            Shoots(:,3)=rand(size(Shoots,1),1)*(1/2)*pi;
            Shoots(:,4)=(rand(size(Shoots,1),1)+.4)/7;
            obj.Shoots=Shoots;
        end
        TF=exist ('Leafs','var');
        if TF==1
            obj.Leafs=Leafs;
        elseif NoLeafs == 0
            obj.Leafs=[];
        else

```

```

        Leafs=zeros(NoLeafs,4);
        Leafs(:,1)=randi(NoStemnodes-3,size(Leafs,1),1)+3;
        Leafs(:,2)=rand(size(Leafs,1),1)*2*pi;
        Leafs(:,3)=rand(size(Leafs,1),1)*(1/2)*pi;
        Leafs(:,4)=(rand(size(Leafs,1),1)+.4)/7;
        obj.Leafs=Leafs;
    end
    TF=exist ('Fruits','var');
    if TF==1
        obj.Fruits=Fruits;
    elseif NoFruits == 0
        obj.Fruits=[];
    else
        Fruits=zeros(NoFruits,5);
        Fruits(:,1)=randi(NoStemnodes-3,size(Fruits,1),1)+3;
        Fruits(:,2)=rand(size(Fruits,1),1)*2*pi;
        Fruits(:,3)=rand(size(Fruits,1),1)*(1/2)*pi;
        Fruits(:,4)=(rand(size(Fruits,1),1)+1)/20;
        Fruits(:,5)=(rand(size(Fruits,1),1)+.5)/20;
        obj.Fruits=Fruits;
    end
end
function Shootnodes=get.Shootnodes(obj)
    % Creates a matrix of coordinates describing the start and end
    % points of the shoots described in the 'Shoots' property.
    %
    % Output is a 3D matrix, containing a 2-by-3 matrix for each
    % shoot; looking like [ Xbegin Ybegin Zbegin ;
    %                      Xend   Yend   Zend   ]
    %
    if obj.NoShoots == 0          %check if there are shoots,
        Shootnodes=[];          %if not: output is empty
    else                          %otherwise: calculate nodes
        Shootnodes=zeros(2,3,obj.NoShoots);
        for i=1:obj.NoShoots
            Shootnodes(1,:,i)=obj.Stemnodes(obj.Shoots(i,1),:);
            delta=[obj.Shoots(i,4)*sin(obj.Shoots(i,2)) -obj.Shoots(i,4)*cos(obj.Shoots(i,2))
                obj.Shoots(i,4)*sin(obj.Shoots(i,3))] ;
            Shootnodes(2,:,i)=Shootnodes(1,:,i)+delta;
        end
    end
end
function LeafShootnodes=get.LeafShootnodes(obj)
    % Creates a matrix of coordinates describing the start and end
    % points of the shoots to which the leafs will be connected as
    % described in the 'Leafs' property.
    %
    % Output is a 3D matrix, containing a 2-by-3 matrix for each
    % leaf shoot; looking like [ Xbegin Ybegin Zbegin ;
    %                      Xend   Yend   Zend   ]
    %
    if obj.NoLeafs == 0          %check if there are leafs
        LeafShootnodes=[];      %if not: output is empty
    else                          %otherwise, calculate nodes
        LeafShootnodes=zeros(2,3,obj.NoLeafs);
        for i=1:obj.NoLeafs
            LeafShootnodes(1,:,i)=obj.Stemnodes(obj.Leafs(i,1),:);
            delta=[obj.Leafs(i,4)*sin(obj.Leafs(i,2)) -obj.Leafs(i,4)*cos(obj.Leafs(i,2))
                obj.Leafs(i,4)*sin(obj.Leafs(i,3))] ;
            LeafShootnodes(2,:,i)=LeafShootnodes(1,:,i)+delta;
        end
    end
end
function PeduncleNodes=get.PeduncleNodes(obj)
    % Creates a matrix of coordinates describing the start and end
    % points of the segments which form the peduncles of the fruits
    % as described in the 'Fruits' property
    %
    % Output is a 3D matrix, containing a n-by-3 matrix for each
    % peduncle; looking like [ X1 Y1 Z1 ;
    %                      X2 Y2 Z2 ;
    %                      .. .. .. ;
    %                      Xn Yn Zn ]
    % n-1 being the number of segments.
    %

```

```

if obj.NoFruits == 0           %check if there are fruits
    Pedunclenodes=[];         %if not: output is empty
else                           %otherwise, calculate nodes
    Pedunclenodes=zeros(11,3,obj.NoFruits);
    for i=1:obj.NoFruits
        fruit=obj.Fruits(i,:); %Retrieve current fruit
        pedunclesegments=round((pi-fruit(3))/(0.1*pi)); %calculate number of segments
        segmentangle=(pi-fruit(3))/(pedunclesegments-1); %calculate angle between segments
        segmentlength=fruit(4)/pedunclesegments; %calculate segment length
        Pedunclenodes(1,:,i)=obj.Stemnodes(fruit(1),:); %first peduncle node is the stem node
        for j=2:pedunclesegments+1 %calculate remaining nodes
            angle=fruit(3)+(j-2)*segmentangle;
            d=segmentlength*sin(angle);
            delta=[d*sin(fruit(2)) -d*cos(fruit(2)) segmentlength*cos(angle)];
            Pedunclenodes(j,:,i)=Pedunclenodes(j-1,:,i)+delta;
        end
    end
end

end
function Fruitnodes=get.Fruitnodes(obj)
    % Creates a matrix of coordinates describing the top and bottom
    % points of the body of the bell pepper as described in the
    % 'Fruits' property,
    %
    % Output is a 3D matrix, containing a 2-by-3 matrix for each
    % pepper; looking like [ Xtop      Ytop      Ztop      ;
    %                        Xbottom   Ybottom   Zbottom ]
    %

    if obj.NoFruits == 0           %check if there are fruits
        Fruitnodes=[];           %if not: output is empty
    else                           %otherwise, calculate nodes
        Fruitnodes=zeros(2,3,obj.NoFruits);
        for i=1:obj.NoFruits
            j=11;
            while obj.Pedunclenodes(j,3,i)==0
                j=j-1;
            end
            Fruitnodes(1,:,i)=obj.Pedunclenodes(j,:,i); %top node is equal to last peduncle node
            Fruitnodes(2,:,i)=Fruitnodes(1,:,i); %bottom node aswell, except
            Fruitnodes(2,3,i)=Fruitnodes(1,3,i)-obj.Fruits(i,5); %Z-coordinate is lower by the fruitsize
        end
    end
end

end
function Draw(obj,Fruitcolor)
    % DRAW Draws a 3D model of the given Plant
    %
    % Input:
    %   Plant object
    % Optional input (default color is red):
    %   Fruitcolor
    %
    % Outputs a 3D surface plot visualising the Plant object
    %

    newplot %clear old plot
    hold on

    NodeSphere(obj.Stemnodes,obj.Stemdiameter,obj.Stemcolor);
    NodeCylinder(obj.Stemnodes,obj.Stemdiameter,obj.Stemcolor);

    if obj.NoShoots ~= 0 %if there are shoots, plot shoots
        NodeSphere(obj.Shootnodes,obj.Shootdiameter,obj.Shootcolor);
        NodeCylinder(obj.Shootnodes,obj.Shootdiameter,obj.Shootcolor);
    end

    if obj.NoLeafs ~= 0 %if there are leafs, plot leafs
        NodeSphere(obj.LeafShootnodes,obj.Shootdiameter,obj.Shootcolor);
        NodeCylinder(obj.LeafShootnodes,obj.Shootdiameter,obj.Shootcolor);
        LeafPatch(obj.Leafs,obj.LeafShootnodes,obj.Leafcolor,1);
    end

    if obj.NoFruits ~= 0 %if there are fruits plot peduncles
        NodeSphere(obj.Pedunclenodes,obj.Pedunclediameter,obj.Pedunclecolor);
    end
end

```

```

NodeCylinder(obj.PeduncleNodes,obj.PeduncleDiameter,obj.PeduncleColor);

if exist('Fruitcolor','var')==0 || Fruitcolor == 1 %if no valid color value is provided or
    Color=obj.FruitcolorRed; %color is red, set color to red
elseif Fruitcolor == 2 %otherwise set color to required color
    Color=obj.FruitcolorOrange;
elseif Fruitcolor == 3
    Color=obj.FruitcolorYellow;
elseif Fruitcolor == 4
    Color=obj.FruitcolorGreen;
end
for i=1:obj.NoFruits %and plot the fruits
    NodeSphere(obj.Fruitnodes(:, :, i), obj.Fruits(i, 5), Color);
    NodeCylinder(obj.Fruitnodes(:, :, i), obj.Fruits(i, 5), Color);
end
end

colormap([1 1 1;1 1 1;1 1 1]); %set surface color
daspect([1 1 1]) %set aspect ratio to square
view(3) %set view to default 3D-plot view
hold off

end
function SavePlant(obj,Filename)
% SAVEPLANT Saves the Plant object.
%
% Input:
%   Plant object
%   Filename
%
% Writes all the properties of the Plant object to an Excel
% file 'Filename.xlsx', making it possible to save and
% alter the Plant.
%

NumbersLegend={'Number of stemnodes';'';'Number of shoots';'';'Number of leafs';'';
    'Number of fruits';''};
xlswrite(Filename,NumbersLegend,'Numbers','A1')
xlswrite(Filename,obj.NoStemnodes,'Numbers','A2')
xlswrite(Filename,obj.NoShoots,'Numbers','A4')
xlswrite(Filename,obj.NoLeafs,'Numbers','A6')
xlswrite(Filename,obj.NoFruits,'Numbers','A8')

StemnodeLegend={'Stemnode' 'X-position' 'Y-position' 'Z-position'};
StemnodeIndex=(1:obj.NoStemnodes)';
StemnodeValues=obj.Stemnodes;
xlswrite(Filename,StemnodeLegend,'Stemnodes','A1')
xlswrite(Filename,StemnodeIndex,'Stemnodes','A2')
xlswrite(Filename,StemnodeValues,'Stemnodes','B2')

ShootLegend={'Shoot' 'Stemnode' 'Horizontal angle' 'Vertical angle' 'Length'};
ShootIndex=(1:obj.NoShoots)';
ShootValues=obj.Shoots;
xlswrite(Filename,ShootLegend,'Shoots','A1')
if obj.NoShoots ~= 0
    xlswrite(Filename,ShootIndex,'Shoots','A2')
    xlswrite(Filename,ShootValues,'Shoots','B2')
end

LeafLegend={'Leaf' 'Stemnode' 'Horizontal angle' 'Vertical angle' 'Length'};
LeafIndex=(1:obj.NoLeafs)';
LeafValues=obj.Leafs;
xlswrite(Filename,LeafLegend,'Leafs','A1')
if obj.NoLeafs ~= 0
    xlswrite(Filename,LeafIndex,'Leafs','A2')
    xlswrite(Filename,LeafValues,'Leafs','B2')
end

FruitLegend={'Fruit' 'Stemnode' 'Horizontal angle' 'Vertical angle' 'Peduncle length' 'Fruit size'};
FruitIndex=(1:obj.NoFruits)';
FruitValues=obj.Fruits;
xlswrite(Filename,FruitLegend,'Fruits','A1')
if obj.NoFruits ~= 0
    xlswrite(Filename,FruitIndex,'Fruits','A2')
    xlswrite(Filename,FruitValues,'Fruits','B2')
end
end
function [PossibleLines,ImpossibleLines,TfArray,Spherepoints] = Directions(obj,FruitNo,resolution)

```

```

% DIRECTIONS Calculates unobstructed directions from a fruit
%
% Inputs
%   Plant object
%   FruitNo      The number of the fruit to calculate directions from
%   Resolution    The accuracy for the calculation
%                 Higher is better, but also takes longer to calculate
% Outputs
%   PossibleLines A list of points describing the unobstructed directions. The
%                 first point is the centerpoint and therefore the beginning of
%                 each line, the other points are the endpoints.
%   ImpossibleLines A list of points describing the obstructed directions. The
%                  first point is the centerpoint and therefore the beginning of
%                  each line, the other points are the endpoints.
%   TFarray        An array containing ones and zeros to indicate true or false
%                  values. True is a possible direction, false an impossible direction.
%   Spherepoints   All the points on the sphere describing all directions.
%
% Creates a sphere around the specified fruit. For each point
% on this sphere a line is created from the centerpoint. All of
% these lines are then checked for interfering with any part of
% the plant model. If a direction is obstructed the value for
% this point is set to False. After the check all lines that
% still have the value True are drawn in the plant model.
% All directions are stored in either PossibleLines or
% ImpossibleLines.

Centerpoint=obj.Fruitnodes(1,:,FruitNo);
Spherepoints=CubeSphere(.75,resolution);
TFarray=ones(size(Spherepoints,1),size(Spherepoints,3));
% Check for interfering stem sections
TFarray = CheckInterference(Centerpoint,Spherepoints,obj.Stemnodes,obj.Stemdiameter,TFarray);
% Check for interfering shoots
TFarray = CheckInterference(Centerpoint,Spherepoints,obj.Shootnodes,obj.Shootdiameter,TFarray);
% Check for interfering leafshoots
TFarray = CheckInterference(Centerpoint,Spherepoints,obj.LeafShootnodes,obj.Shootdiameter,TFarray);
% Check for interfering fruit
for i=1:obj.NoFruits
    if i ~= FruitNo
        TFarray =
CheckInterference(Centerpoint,Spherepoints,obj.Fruitnodes(:, :, i),obj.Fruits(i,5),TFarray);
    end
end
% Check for interfering leafs
[AllPatchX,AllPatchY,AllPatchZ]=LeafPatch(obj.Leafs,obj.LeafShootnodes,obj.Leafcolor,1);
for i=1:size(AllPatchX,2)
    PatchX=AllPatchX(:,i);
    PatchY=AllPatchY(:,i);
    PatchZ=AllPatchZ(:,i);
    for ii=1:size(Spherepoints,1)
        for jj=1:size(Spherepoints,3)
            CurrentSpherepoint=Spherepoints(ii,:,jj)+Centerpoint;
            TF=PatchClearance(PatchX,PatchY,PatchZ,Centerpoint,CurrentSpherepoint);
            TFarray(ii,jj)=TFarray(ii,jj)*TF;
        end
    end
end
% Draw the not interfering lines, and store the endpoints of these lines
hold on
PossibleLines=Centerpoint;
ImpossibleLines=Centerpoint;
for ii=1:size(Spherepoints,1)
    for jj=1:size(Spherepoints,3)
        CurrentSpherepoint=Spherepoints(ii,:,jj)+Centerpoint;
        if TFarray(ii,jj)==1
            line([Centerpoint(1) CurrentSpherepoint(1)],[Centerpoint(2)
                CurrentSpherepoint(2)],[Centerpoint(3) CurrentSpherepoint(3)])
            PossibleLines=[PossibleLines;CurrentSpherepoint];
        else
            ImpossibleLines=[ImpossibleLines;CurrentSpherepoint];
        end
    end
end
hold off
end
end
end

```



```

methods(Static)
function [NoStemnodes,NoShoots,NoLeafs,NoFruits,Stemnodes,Shoots,Leafs,Fruits]=LoadPlant (FileName)
% LOADPLANT Loads a Plant object.
%
% Input:
%   Filename
% Outputs:
%   NoStemnodes   Number of stemnodes
%   NoShoots      Number of Shoots
%   NoLeafs       Number of Leafs
%   NoFruits      Number of Fruits
%   Stemnodes     Properties of the Plants stemnodes
%   Shoots        Properties of the Plants Shoots
%   Leafs         Properties of the Plants Leafs
%   Fruits        Properties of the Plants Fruits
%
% Reads the required properties from an Excel file
% 'Filename.xlsx'. These can be used to construct a
% new Plant object with the properties recovered from the file.
%

NoStemnodes=xlsread(FileName,'Numbers','A2');
NoShoots=xlsread(FileName,'Numbers','A4');
NoLeafs=xlsread(FileName,'Numbers','A6');
NoFruits=xlsread(FileName,'Numbers','A8');
if NoStemnodes ~= 0
    Index=['B2:D' num2str(NoStemnodes+1)];
    Stemnodes=xlsread(FileName,'Stemnodes',Index);
else
    Stemnodes=[];
end
if NoShoots ~= 0
    Index=['B2:E' num2str(NoShoots+1)];
    Shoots=xlsread(FileName,'Shoots',Index);
else
    Shoots=[];
end
if NoLeafs ~= 0
    Index=['B2:E' num2str(NoLeafs+1)];
    Leafs=xlsread(FileName,'Leafs',Index);
else
    Leafs=[];
end
if NoFruits ~= 0
    Index=['B2:F' num2str(NoFruits+1)];
    Fruits=xlsread(FileName,'Fruits',Index);
else
    Fruits=[];
end
end
function ScatterPlot(PossibleLines,ImpossibleLines)
% SCATTERPLOT Creates a scatterplot of all directions
%
% Inputs:
%   PossibleLines   A matrix with the startingpoint an all endpoints of the
%                   unobstructed directions: [Xstart Ystart Zstart;
%                   Xend 1 Yend 1 Zend 1;
%                   ..      ..      ..      ;
%                   Xend n Yend n Zend n]
%   ImpossibleLines A matrix with the startingpoint an all endpoints of the
%                   obstructed directions: [Xstart Ystart Zstart;
%                   Xend 1 Yend 1 Zend 1;
%                   ..      ..      ..      ;
%                   Xend n Yend n Zend n]
%
% Converts each line to a angle Theta and an angle Phi. For
% each line a point is plotted in a scatterplot with Theta on
% the horizontal axis and Phi on the vertical axis. The color
% and style of the point indicate Whether it is a possible or
% impossible direction.
figure('Name','Scatter plot of possible directions','NumberTitle','off')
Phi=zeros(size(PossibleLines,1)-1,1);
Theta=zeros(size(PossibleLines,1)-1,1);
for i=2:size(PossibleLines,1)
    dx=PossibleLines(i,1)-PossibleLines(1,1);
    dy=PossibleLines(i,2)-PossibleLines(1,2);
    dz=PossibleLines(i,3)-PossibleLines(1,3);

```

```

        Phi(i-1)=atan2(dy,dx);
        Theta(i-1)=atan2(dz,sqrt(dx^2+dy^2));
    end
    scatter(Phi,Theta,'*','g')
    hold on
    Phi=zeros(size(ImpossibleLines,1)-1,1);
    Theta=zeros(size(ImpossibleLines,1)-1,1);
    for i=2:size(ImpossibleLines,1)
        dx=ImpossibleLines(i,1)-ImpossibleLines(1,1);
        dy=ImpossibleLines(i,2)-ImpossibleLines(1,2);
        dz=ImpossibleLines(i,3)-ImpossibleLines(1,3);
        Phi(i-1)=atan2(dy,dx);
        Theta(i-1)=atan2(dz,sqrt(dx^2+dy^2));
    end
    scatter(Phi,Theta,'o','r','filled')
    axis([-3.2 3.2 -1.6 1.6])
    daspect([1 1 1])
    xlabel('Theta')
    ylabel('Phi')
    set(gca,'XTick',[-pi -.5*pi 0 .5*pi pi])
    set(gca,'XTickLabel',{'-pi|-1/2*pi|0|1/2*pi|pi'})
    set(gca,'YTick',[-.5*pi -.25*pi 0 .25*pi .5*pi])
    set(gca,'YTickLabel',{'-1/2*pi|-1/4*pi|0|1/4*pi|1/2*pi'})
    legend('Free directions','Obstructed directions','Location','EastOutside')
    hold off
end
end
end
function NodeSphere(NodeLocations,diameter,color)
%NODESPHERE Draws a sphere of the given diameter on the location of a given
%node
% Inputs:
%   NodeLocations The locations of the nodes where the sphere needs
%                   to go.
%   diameter      The diameter of the sphere [m]
%   color         The edgecolor of the sphere
%
% Outputs:
%   A surface plot of a sphere at the given location with the given diameter
%
% Required format of NodeLocations:
%   [x y z] locations in matrix rows. A sphere is drawn for each row.
%   Accepts two and three dimensional matrices.
%
S=size(NodeLocations); %get the number of nodes
if length(S)==2        %if the matrix is 2-dimensional,
    S(3)=1;            %size in third dimension is 1
end

for i=2:S(1)
    for j=1:S(3)
        if NodeLocations(i,3,j)~=0; %reject invalid coordinates (z=0), a zero there indicates it is
                                     %meaningless matrix filling
            [X,Y,Z]=sphere(50);      %create a unit sphere: radius is one, location is [0 0 0]

            scaledX=X*(diameter/2); %scale sphere to desired diameter
            scaledY=Y*(diameter/2);
            scaledZ=Z*(diameter/2);

            finalX=scaledX+NodeLocations(i,1,j);%move scaled sphere to desired location
            finalY=scaledY+NodeLocations(i,2,j);
            finalZ=scaledZ+NodeLocations(i,3,j);

            surf(finalX,finalY,finalZ,'EdgeColor',color)%plot the scaled an moved shpere
        end
    end
end
end
function NodeCylinder(NodeLocations,diameter,color)
%NODECYLINDER Draws a cylinder of the given diameter from the first to
%the second given location.
% Inputs:
%   NodeLocations The locations of the nodes where the cylinder needs
%                   to go.
%   diameter      The diameter of the cylinder [m]
%   color         The edgecolor of the cylinder

```

```

%
%   Outputs:
%       A surface plot of a cylinder at the given location with the given diameter
%
% Required format of NodeLocations:
% [x y z] locations in rows. A cylinder is drawn from each row to the
% next, providing there is a next row ofcourse.
% Accepts two and three dimensional matrices.
%

S=size(NodeLocations);
if length(S)==2
    S(3)=1;
end

for i=2:S(1)
    for j=1:S(3)
        if NodeLocations(i,3,j)~=0 %reject invalid coordinates (z=0), a zero there indicates it is
                                   %meaningless matrix filling
            dx=NodeLocations(i,1,j)-NodeLocations(i-1,1,j); %calculate the x, y and z difference
            dy=NodeLocations(i,2,j)-NodeLocations(i-1,2,j);
            dz=NodeLocations(i,3,j)-NodeLocations(i-1,3,j);

            a=0; %calculate angle to next point
            b=-atan(dx/dz);
            g=atan(dy/dz);

            zyx=Eulertransf_ZYX(a,b,g); %calculate required Euler transformation matrix

            [X,Y,Z] = cylinder(1,50); %create unit cylinder: radius and height is one,
                                     %location is [0 0 0]
            scaledX=X*(diameter/2); %scale cylinder to desired diameter
            scaledY=Y*(diameter/2);
            scaledZ=Z*(diameter/2);
            points=[scaledX(1,:);scaledY(1,:);scaledZ(1,:)];%put scaled cylinder in a format usable for
                                                         Euler transformation
            tpoints=transpose(points);

            [n,m]=size(tpoints);
            trpoints=ones(n,m);

            for ii=1:n
                trpoints(ii,:)=tpoints(ii,:)*zyx; %apply Euler transformation to rotate the
                                                    cylinder to calculated angles
            end

            rpoints=transpose(trpoints); %return to original format

            rotatedX=rpoints(1,:);
            rotatedY=rpoints(2,:);
            rotatedZ=rpoints(3,:);

            %move cylinder to desired location
            translatedX=[rotatedX(1,:)+NodeLocations(i-1,1,j);rotatedX(1,:)+NodeLocations(i,1,j)];
            translatedY=[rotatedY(1,:)+NodeLocations(i-1,2,j);rotatedY(1,:)+NodeLocations(i,2,j)];
            translatedZ=[rotatedZ(1,:)+NodeLocations(i-1,3,j);rotatedZ(1,:)+NodeLocations(i,3,j)];

            surf(translatedX,translatedY,translatedZ,'EdgeColor',color) %plot rotated and moved cylinder
        end
    end
end
end
function [translatedpatchX,translatedpatchY,translatedpatchZ] = LeafPatch(Leafs,LeafShootnodes,color,plot)
%LEAFPATCH Draws two triangular patches representing a leaf
%   Inputs:
%       Leafs: the matrix containing the leaf properties
%       LeafShootnodes: the matrix containing the coordinates of the
%                       shoots the leafs need to be attached to.
%       color: the patch color
%
%   Outputs:
%       Two triangular patches representing a leaf
%
% Required format of Leafs: see 'help Plant.L Leafs'
%
```

```

doubleLeafs=[zeros(size(Leafs));zeros(size(Leafs))]; %create matrices double the size of the inputs to
                                                    accommodate
doubleLeafShootnodes=zeros(size(LeafShootnodes,3)*2,3); %two patches for each leaf

for i=2:2:size(doubleLeafs,1) %fill matrices with each single row in the input
                                becoming
                                %two identical rows
    doubleLeafs(i-1,:)=Leafs(i/2,:);
    doubleLeafs(i,:)=Leafs(i/2,:);
    doubleLeafShootnodes(i-1,:)=LeafShootnodes(2,:,i/2);
    doubleLeafShootnodes(i,:)=LeafShootnodes(2,:,i/2);
end

patchX=zeros(3,size(doubleLeafs,1)); %initiate matrices to accommodate patch coordinates
patchY=zeros(3,size(doubleLeafs,1));
patchZ=zeros(3,size(doubleLeafs,1));

for i=2:2:size(doubleLeafs,1) %create leafs of the desired size starting in
                                [0 0 0]
                                %with both horizontal and vertical angles being zero
    patchY(1,i)=-(3/2)*doubleLeafs(i,4);
    patchY(2:3,i-1:i)=-(1/2)*doubleLeafs(i,4);
    patchX(2,i-1:i)=-(1/2)*doubleLeafs(i,4);
    patchX(3,i-1:i)=(1/2)*doubleLeafs(i,4);
end

points1=[patchX(1,:);patchY(1,:);patchZ(1,:)]; %transform matrices to a usable format for the Euler
                                                    transformation
points2=[patchX(2,:);patchY(2,:);patchZ(2,:)]; %from all x, all y and all z coordinates in separate
                                                    matrices
points3=[patchX(3,:);patchY(3,:);patchZ(3,:)]; %to each of the three points of the patch in
                                                    separate matrices

tpoints1=transpose(points1);
tpoints2=transpose(points2);
tpoints3=transpose(points3);

[n,m]=size(tpoints1);

trpoints1=zeros(n,m);
trpoints2=zeros(n,m);
trpoints3=zeros(n,m);

for i=1:n %rotate the leaf to the desired angle
    a=doubleLeafs(i,3)-(1/2*pi);
    b=0;
    g=-doubleLeafs(i,2);
    xyz=Eulertransf_XYZ(a,b,g);
    trpoints1(i,:)=tpoints1(i,:)*xyz;
    trpoints2(i,:)=tpoints2(i,:)*xyz;
    trpoints3(i,:)=tpoints3(i,:)*xyz;
end

rpoints1=transpose(trpoints1);
rpoints2=transpose(trpoints2);
rpoints3=transpose(trpoints3);

rotatedpatchX=[rpoints1(1,:);rpoints2(1,:);rpoints3(1,:)]; %transform matrices back to the x, y, z format
rotatedpatchY=[rpoints1(2,:);rpoints2(2,:);rpoints3(2,:)];
rotatedpatchZ=[rpoints1(3,:);rpoints2(3,:);rpoints3(3,:)];

translatedpatchX=zeros(size(rotatedpatchX));
translatedpatchY=zeros(size(rotatedpatchY));
translatedpatchZ=zeros(size(rotatedpatchZ));

for i=2:2:size(doubleLeafs,1) %move rotated leafs to desired location
    translatedpatchX(:,i-1)=rotatedpatchX(:,i-1)+doubleLeafShootnodes(i-1,1);
    translatedpatchX(:,i)=rotatedpatchX(:,i)+doubleLeafShootnodes(i,1);
    translatedpatchY(:,i-1)=rotatedpatchY(:,i-1)+doubleLeafShootnodes(i-1,2);
    translatedpatchY(:,i)=rotatedpatchY(:,i)+doubleLeafShootnodes(i,2);
    translatedpatchZ(:,i-1)=rotatedpatchZ(:,i-1)+doubleLeafShootnodes(i-1,3);
    translatedpatchZ(:,i)=rotatedpatchZ(:,i)+doubleLeafShootnodes(i,3);
end

if plot==1
    patch(translatedpatchX,translatedpatchY,translatedpatchZ,color) %plot the patches
end

```

```

end
function [zyx] = Eulertransf_ZYX(a,b,g)
    %EULERTRANSF_ZYX Creates an Euler transformation matrix from the given
    % angles. Rotation is performed in the order Z, Y, X.
    % Inputs:
    %     a: alfa, the Z rotation angle [rad]
    %     b: beta, the Y rotation angle [rad]
    %     g: gamma, the X rotation angle [rad]
    %
    % Outputs:
    %     zyx: the transformation matrix
    %
    % For more information on Euler transformations, see:
    % http://en.wikipedia.org/wiki/Euler\_angles
    %

    zyx=[cos(a)*cos(b) cos(a)*sin(b)*sin(g)-sin(a)*cos(g) sin(a)*sin(g)+cos(a)*sin(b)*cos(g);
        sin(a)*cos(b) sin(a)*sin(b)*sin(g)+cos(a)*cos(g) sin(a)*sin(b)*cos(g)-cos(a)*sin(g);
        -sin(b) cos(b)*sin(g) cos(b)*cos(g)];
end
function [xyz] = Eulertransf_XYZ(a,b,g)
    %EULERTRANSF_XYZ Creates an Euler transformation matrix from the given
    % angles. Rotation is performed in the order X, Y, Z.
    % Inputs:
    %     a: alfa, the X rotation angle [rad]
    %     b: beta, the Y rotation angle [rad]
    %     g: gamma, the Z rotation angle [rad]
    %
    % Outputs:
    %     xyz: the transformation matrix
    %
    % For more information on Euler transformations, see:
    % http://en.wikipedia.org/wiki/Euler\_angles
    %

    xyz=[cos(b)*cos(g) -cos(b)*sin(g) sin(b);
        cos(a)*sin(g)+cos(g)*sin(a)*sin(b) cos(a)*cos(g)-sin(a)*sin(b)*sin(g) -cos(b)*sin(a);
        sin(a)*sin(g)-cos(a)*cos(g)*sin(b) cos(a)*sin(b)*sin(g)+cos(g)*sin(a) cos(a)*cos(b)];
end
function [Spherepoints] = CubeSphere(size,resolution)
    % CUBESPHERE Creates a cube and projecte it onto a sphere
    %
    % Inputs
    %     size          The diameter of the sphere
    %     resolution    The accuracy of the sphere. The number of squares on each side of the
    %                   cube is resolution^2.
    %
    % Outputs
    %     Spherepoints  The points describing the sphere
    %
    % First a cube with sides of length size are is creates with
    % centerpoint [0 0 0]. Using Cube2Sphere the cube is projected onto a
    % Sphere.

    CubeTop=zeros(resolution+1,3,resolution+1);
    CubeBottom=zeros(resolution+1,3,resolution+1);
    CubeLeft=zeros(resolution+1,3,resolution+1);
    CubeRight=zeros(resolution+1,3,resolution+1);
    CubeFront=zeros(resolution+1,3,resolution+1);
    CubeBack=zeros(resolution+1,3,resolution+1);

    %Create a cube
    a=.5*size;
    step=size/resolution;
    A=-.5*size:step:.5*size;

    for i=1:resolution+1
        CubeTop(:,1,i)=transpose(A);
        CubeTop(i,2,:)=A;
        CubeTop(:,3,:)=a;
        CubeBottom(:,1,i)=transpose(A);
        CubeBottom(i,2,:)=A;
        CubeBottom(:,3,:)=-a;
        CubeBack(:,1,i)=transpose(A);
        CubeBack(:,2,:)=a;
        CubeBack(i,3,:)=A;
        CubeFront(:,1,i)=transpose(A);
        CubeFront(:,2,:)=-a;
    end

```

```

    CubeFront(i,3,:)=A;
    CubeRight(:,1,:)=a;
    CubeRight(i,2,:)=A;
    CubeRight(:,3,i)=transpose(A);
    CubeLeft(:,1,:)=a;
    CubeLeft(i,2,:)=A;
    CubeLeft(:,3,i)=transpose(A);
end

%Turn it into a sphere
SphereTop=Cube2Sphere(CubeTop);
SphereBottom=Cube2Sphere(CubeBottom);
SphereBack=Cube2Sphere(CubeBack);
SphereFront=Cube2Sphere(CubeFront);
SphereLeft=Cube2Sphere(CubeLeft);
SphereRight=Cube2Sphere(CubeRight);

Spherepoints=[SphereTop;SphereBottom;SphereBack;SphereFront;SphereLeft;SphereRight];
end
function [SphereSide] = Cube2Sphere(side)
% CUBE2SHPERE Converts a side of a cube to a side of a sphere
%
% Inputs
%   side    A matrix of points describing the side of a cube.
% Outputs
%   SphereSide  A matrix of points describing the input side as a side of a sphere.
%
% All points are described as a row [x y z]. For each point in the
% input the distance to the centerpoint is calculated. This distance is
% then set to 1 to create a sphere.
%
SphereSide=zeros(size(side));
for i=1:size(side,1)
    for j=1:size(side,3)
        sqdist=side(i,1,j)^2+side(i,2,j)^2+side(i,3,j)^2;
        dist=sqrt(sqdist);
        SphereSide(i,:,j)=side(i,:,j)*dist^-1;
    end
end
end
function [TFarray] = CheckInterference(Centerpoint,Spherepoints,NodeLocations,distance,TFarray)
% CHECKINTERFERENCE Checks the clearance of two arrays of lines
% Inputs
%   Centerpoint    The point from where the directions will be checked, the
%                   centerpoint of the sphere.
%   Spherepoints    An array of points describing a sphere. A line will be drawn to each
%                   point from the centerpoint.
%   NodeLocations    A list of points. A line will be drawn from each point to the next.
%   distance        The required clearance.
%   TFarray         An array containing True or False values for each previously
%                   checked point.
% Outputs
%   TFarray         An array containing True or False values for each
%                   checked point.
% This function receives arrays of points as an input. From these
% arrays two points are taken describing a line segment. These points
% are fed into a function calculating the clearance between the described
% line segments.

S=size(NodeLocations);
if length(S)==2
    S(3)=1;
end

for i=2:S(1)
    for j=1:S(3)
        if NodeLocations(i,3,j)~=0
            CurrentNode=NodeLocations(i-1,:,j);
            NextNode=NodeLocations(i,:,j);
            for ii=1:size(Spherepoints,1)
                for jj=1:size(Spherepoints,3)
                    CurrentSpherepoint=Spherepoints(ii,:,jj)+Centerpoint;
                    TF=LineClearance(Centerpoint,CurrentSpherepoint,CurrentNode,NextNode,distance);
                    TFarray(ii,jj)=TFarray(ii,jj)*TF;
                end
            end
        end
    end
end

```

```

        end
    end
end
function [TF] = LineClearance(p1, p2, p3, p4, distance)
% LINECLEARANCE Computes the minimum distance between two line segments.
%
% Code is adapted for Matlab by Nick Gravish from Dan Sunday's Geometry
% Algorithms originally written in C++
% Copyright 2001, softSurfer (www.softsurfer.com)
% This code may be freely used and modified for any purpose
% providing that this copyright notice is included with it.
%
% http://softsurfer.com/Archive/algorithm_0106/algorithm_0106.htm
%
% An input distance was added by Thijs de Swart to create a True or False
% output in stat of a distance output.
%
% Inputs:
%   p1          First point of the first line
%   p2          Second point of the first line
%   p3          First point of the second line
%   p4          Second point of the second line
%   distance     The required clearance
%
% Output:
%   TF  A True or False value to indicate if the required clearance is
%       present.
%
%   Calculates the minimum distance between two line segments. If this
%   distance is smaller than the input value for distance the line clearance
%   is not sufficient and the ouytpu will be False (0). If the clearance is
%   sufficient the output will be True (1).
%
u = p1 - p2;
v = p3 - p4;
w = p2 - p4;

a = dot(u,u);
b = dot(u,v);
c = dot(v,v);
d = dot(u,w);
e = dot(v,w);
D = a*c - b*b;
sD = D;
tD = D;

SMALL_NUM = 0.00000001;

% compute the line parameters of the two closest points
if (D < SMALL_NUM) % the lines are almost parallel
    sN = 0.0; % force using point P0 on segment S1
    sD = 1.0; % to prevent possible division by 0.0 later
    tN = e;
    tD = c;
else % get the closest points on the infinite lines
    sN = (b*e - c*d);
    tN = (a*e - b*d);
    if (sN < 0.0) % sc < 0 => the s=0 edge is visible
        sN = 0.0;
        tN = e;
        tD = c;
    elseif (sN > sD) % sc > 1 => the s=1 edge is visible
        sN = sD;
        tN = e + b;
        tD = c;
    end
end

if (tN < 0.0) % tc < 0 => the t=0 edge is visible
    tN = 0.0;
    % recompute sc for this edge
    if (-d < 0.0)
        sN = 0.0;
    elseif (-d > a)
        sN = sD;
    else

```

```

        sN = -d;
        sD = a;
    end
elseif (tN > tD) % tc > 1 => the t=1 edge is visible
    tN = tD;
    % recompute sc for this edge
    if ((-d + b) < 0.0)
        sN = 0;
    elseif ((-d + b) > a)
        sN = sD;
    else
        sN = (-d + b);
        sD = a;
    end
end

% finally do the division to get sc and tc
if(abs(sN) < SMALL_NUM)
    sc = 0.0;
else
    sc = sN / sD;
end

if(abs(tN) < SMALL_NUM)
    tc = 0.0;
else
    tc = tN / tD;
end

% get the difference of the two closest points
dP = w + (sc * u) - (tc * v); % = S1(sc) - S2(tc)

clearance = norm(dP);

% check whether the clearance is sufficient
if clearance > distance
    TF=1;
else
    TF=0;
end
end
function [TF] = PatchClearance(PatchX,PatchY,PatchZ,P1,P2)
% PATCHCLEARANCE Checks whether a line segment clears a triangular patch
%
% Inputs
% PatchX The three X coordinates of the patch points
% PatchY The three Y coordinates of the patch points
% PatchZ The three Z coordinates of the patch points
% P1 The first points of the line segment [x y z]
% P2 The second points of the line segment [x y z]
%
% Output
% TF A True or False value indicating whether the line segment clears the patch.
%
% Formulas taken from wikipedia page 'Line-plane intersection':
% http://en.wikipedia.org/wiki/Line-plane\_intersection

Xa=P1(1); Ya=P1(2); Za=P1(3);
Xb=P2(1); Yb=P2(2); Zb=P2(3);

X0=PatchX(1); Y0=PatchY(1); Z0=PatchZ(1);
X1=PatchX(2); Y1=PatchY(2); Z1=PatchZ(2);
X2=PatchX(3); Y2=PatchY(3); Z2=PatchZ(3);

%if line and plane are parallel, no solution can be found
%check if line and plane are parallel

ParallelCheck=[Xb-Xa X1-X0 X2-X0;Yb-Ya Y1-Y0 Y2-Y0;Zb-Za Z1-Z0 Z2-Z0];

if det(ParallelCheck) ~= 0 %if not parallel, check for line-plane intersection

    % calculate the intersection point
    A=[Xa-Xb X1-X0 X2-X0;Ya-Yb Y1-Y0 Y2-Y0;Za-Zb Z1-Z0 Z2-Z0];
    B=[Xa-X0;Ya-Y0;Za-Z0];

    C=(A^-1)*B;

```



```
t=C(1); u=C(2); v=C(3);

% check whether the intersection point lies within the patch and on
% the line segment
if t>=0 && u>=0 && v>=0 && t<=1 && u+v<=1
    TF=0;
else
    TF=1; % if no intersection between the bounds is found, it clears
end
else
    TF=1; % if parallel there is no intersection so the segment clears
end
end
```