KU Leuven
Departement Computerwetenschappen

P&O: Computerwetenschappen

Eindverslag

Team:
Geel

Coomans Arno
(Coördinator)
Saniz Bernardo
(Secretaris)
Geens Tomas
Peeters Florian
Theuns Flor
Willemot Toon

Academiejaar 2017-2018

Samenvatting

Drones zijn een steeds belangrijker aspect van de moderne wereld. Ze moeten echter veel tests ondergaan op vlak van prestaties en controle. Dit rapport beschrijft het ontwerp en implementatie van een softwarepakket die deze tests in een virtuele omgeving kan simuleren. Het beschreven systeem is niet volledig, maar kan later als basis dienen voor complexere systemen.

Dit verslag geeft een overzicht van de verschillende componenten van het software pakket, alsook van hoe deze componenten met elkaar communiceren. Het biedt ook een analyse van gebruikte algoritmes en van gemaakte benaderingen. Heel het project is aan de hand van een expliciete opgave gemaakt. Daarom wordt ook een reflectie gemaakt van in hoeverre het systeem voldoet aan de vereisten uit deze opgave.

Inhoudsopgave

1 Inleiding

Bernardo, Arno

Drones zijn tegenwoordig overal te vinden. Een autonoom, vliegend systeem biedt een robuuste en veilige oplossing voor verschillende situaties. Momenteel zijn drones echter nog geen goedkoop gegeven. Het is daarom interessant om de prestatie van zo een drone virtueel te kunnen testen. Hierbij is er namelijk geen risico op dure ongevallen en is het verzamelen van data bijgevolg veel minder kostelijk. Dit verslag beschrijft het ontwerp van een softwarepakket dat het uitvoeren van dergelijke tests toelaat.

De opgave is eenvoudig: simuleer een drone die door middel van beeldherkenning en flight control in een virtuele wereld kan navigeren en een reeks doelen kan bereiken. De drone zal beginnen met positie $(0, 0, 0)$, met zijn neus gericht naar de negatieve z-as. In deze virtuele wereld worden afstanden in meters uitgedrukt. In de ruimte voor de drone zullen er kubussen gegenereerd worden binnen een cilinder met een straal van 10 meter rond de negatieve z-as. Om de 40 meter zal een kubus voorkomen. De drone is uitgerust met een statische camera die beelden van zijn omgeving neemt en verwerkt tot nuttige informatie. Aan de hand van deze verworven informatie moet de drone zijn traject veranderen om een doel te bereiken. Dit moet gebeuren door de controleoppervlakken en stuwkracht van de drone aan te sturen. Een fysisch model moet het effect dat deze aansturing uiteindelijk zal hebben bepalen. Eens dit gebeurd is begint het hierboven beschreven proces opnieuw.

Het ontwerpen van zo een drone bestaat uit twee delen. Ten eerste moet de software een fysische wereld simuleren waarin objecten krachten ondervinden en van toestand veranderen in functie van de tijd. Doorheen het verslag verwijzen we hiernaar met het woord *testbed*. Dit testbed omvat alles dat de wereld definieert: fysische wetten, objecten, en visualisatie van alle objecten doorheen de tijd. Ten tweede moet de software een drone kunnen controleren die in deze gesimuleerde wereld bestaat. Dit deel, de *autopilot*, staat in voor de beeldherkenning en controle van de drone. De implementatie en prestaties van elk van deze onderdelen worden in wat volgt in groter detail toegelicht.

2 Ontwerp

Florian

Het testbed bestaat uit vier delen. Het eerste deel is de physics engine, dit deel is verantwoordelijk voor het behandelen van de positie en orientatie van de drone. Zoals de naam ook vertelt, wordt hierbij gebruik gemaakt van de wetten van de fysica die van toepassing zijn op de drone. De uitwerking van de physics engine wordt verder toegelicht in hoofdstuk 3.

Het tweede van de vier delen is de rendering. Dit deel zorgt voor de visuele voorstelling van de wereld op het scherm. Het maakt ook een camerabeeld dat de autopilot nodig zal hebben om zijn weg te kunnen vinden. De implementatie van de rendering wordt besproken in hoofdstuk 4. Het derde deel is de manier waarop de gebruiker het testbed kan bedienen: de graphical user interface (GUI). Dit deel wordt besproken in hoofdstuk 7. Het laatste deel van het testbed is de communicatie met de autopilot tijdens de simulatie. Dit is een zeer beperkt deel van de software, en wordt daarom ook niet in een apart hoofdstuk besproken, maar in sectie 2.1.

De autopilot die we ontworpen hebben, bestaat uit twee grote delen: de motion planning en de beeldherkenning. De beeldherkenning herkent het doel in het beeld van de camera van de drone, en bepaalt hiervan de locatie. Deze locatie is het doel van de motion planning, die de drone probeert te sturen naar het doel. Motion planning en beeldherkenning worden respectievelijk besproken in hoofdstuk 6 en 5. Onze autopilot toont ook een kleine GUI, deze wordt nog kort besproken in sectie 7.3.

2.1 Communicatie

Florian

De communicatie tussen autopilot en testbed kan op twee manieren verlopen. Een eerste methode is met de *Java* API, dan communiceren ze op basis van de *Interfaces* uit de opgave [1]. Dit is de snelste methode, en ook de methode die aangeraden wordt om te gebruiken indien mogelijk. De andere communicatiemethode gebruikt *Java sockets* [2], waarbij testbed en autopilot in aparte processen worden uitgevoerd, en de communicatie verloopt over een bitstream volgens het wireframe beschreven in de opgave [12]. Hierbij gedraagt het testbed zich als een server waarmee de autopilot zich kan verbinden om de simulatie uit te voeren.

De sockets methode is momenteel geïmplementeerd, maar is niet beschikbaar om mee te simuleren. Aangezien dit een minder belangrijke vereiste was in het project, hebben we deze met laagste prioriteit behandeld. En omdat andere delen van het project meer problemen ondervonden, is de verbinding tussen testbed en de sockets implementatie nog niet afgewerkt.

3 Physics engine

Florian

De taak van de physics engine bestaat uit het berekenen van de positie, oriëntatie en snelheid van de drone. In elke update wordt de physics engine gevraagd deze parameters van de drone te berekenen. Hieronder staan de vier vereisten die we hebben opgelegd aan onze implementatie, met telkens de bijhorende redenering;

1. Houdt alle waarden nodig voor berekeningen intern bij na initialisatie.
   Zo kunnen we de physics engine als een black box beschouwen, die enkel een tijdsstap nodig heeft om een update uit te voeren. Indien we de implementatie veranderen van de physics engine, zou dit geen problemen veroorzaken in de rest van het testbed.

2. Berekent de oplossingen van de bewegingsvergelijkingen door middel van een numerieke benadering.
   Deze vereiste helpt met het beperken van de van de complexiteit van de code, en de tijdsduur van complexe beweginsvergelijkingen.

3. Respecteer alle vereisten gegeven in de opgave van het project.
   Dit is vanzelfsprekend, de physics engine moet alle formules en beperkingen uit *Autopilot.datatypes* [12] implementeren.

4. Ondersteun een variabele tijdsstap.
   De optie om de tijdsstap gedurende de simulatie aan te passen laat ons toe de simulatie dynamisch te versnellen of vertragen.

Om deze vereisten te realiseren hebben we gebruik gemaakt van een library die ons toelaat met vectoren en matrices te werken, JOML[3]. Dit leek een verstandige keuze aangezien JOML het werken met *floats* toelaat, en we dezelfde library ook gebruiken bij de rendering.

Nu volgt er uitleg over onze implementatie van deze vereisten. De eerste sectie geeft een algemene uitleg over welke fysische wetten optreden en welke differentiaalvergelijkingen gebruikt worden. De tweede sectie beschrijft hoe we deze hebben vertaald naar de vereisten in het project, en hoe de eigenlijke implementatie eruit ziet. Vervolgens hebben we het over de gebruikte assenstelsels en de transformaties ertussen. Tot slot komt een kort verslag over het verloop van de ontwikkeling van de physics engine.

[1] *Autopilot, AutopilotConfig, AutopilotInputs en AutopilotOutputs* uit de datatypes folder van de opgave [12]
[2] https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
[3] Java math library for OpenGL

3.1 Wetten en vergelijkingen

Florian

Iedere beweging van een object in de ruimte is volledig gedefinieerd door combinatie van een rotatie- en een translatiebeweging. Beide bewegingen hebben hun eigen fysische wetten waaraan ze moeten voldoen. In de berekeningen die het testbed maakt om de drone te simuleren wordt het tweede postulaat van Newton gebruikt voor de translaties, en de definitie van het impulsmoment voor de rotaties.

$$\sum \vec{F} = m\vec{a} \tag{1}$$

$$I\vec{\alpha} = \frac{d\vec{L}}{dt}$$
$$\frac{d\vec{L}}{dt} = \sum \vec{M} + \vec{\omega} * \vec{L}$$
$$I\vec{\alpha} = \sum \vec{M} + \vec{\omega} * \vec{L} \tag{2}$$

Hier staat $\vec{F}$ voor de krachten, m voor de totale massa en $\vec{a}$ voor de versnelling. Verder staat I voor de inertiematrix, $\vec{\alpha}$ voor de hoekversnellingsvector, $\vec{L}$ voor het angulair moment, $\vec{M}$ voor momenten en $\vec{\omega}$ voor de rotatiesnelheid. Deze vergelijkingen leiden tot een set differentiaalvergelijkingen waaruit de snelheid en rotatiesnelheid worden bepaald. Vergelijkingen 3 en 4 zijn bekomen door het respectievelijk omvormen van vergelijkingen 1 en 2. In de tweede differentiaalvergelijking wordt ook nog een definitie van het impulsmoment toegepast: $\vec{L} = I\vec{\omega}$

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{\sum \vec{F}}{m} \tag{3}$$

$$\vec{\alpha} = \frac{d\vec{\omega}}{dt} = I^{-1}\left(\sum \vec{M} + \vec{\omega} * I\vec{\omega}\right) \tag{4}$$

Aangezien zowel de krachten als de momenten te schrijven zijn als een functie van de snelheid en de rotatie, hebben we een stelsel van twee vectoriële differentiaalvergelijkingen. Dit stelsel beschrijft de snelheid en rotatiesnelheid van het object volledig. Om de positie en oriëntatie van het object te bekomen, zijn nog twee extra vectoriële differentiaalvergelijkingen nodig. Hier staat $\vec{x}$ voor de positie en $\vec{\theta}$ voor de oriëntatie.

$$\frac{d\vec{x}}{dt} = \vec{v} \tag{5}$$

$$\frac{d\vec{\theta}}{dt} = \vec{\omega} \tag{6}$$

Voor een gedetailleerdere uitleg over de volledige uitwerking van de dynamica, zie een uitgebreidere studie van de dynamica, zoals bijvoorbeeld het handboek van mechanica 2 [9], waar al deze formules uit komen.

3.2 Toepassing in het project

Florian

De taak van de physics engine in het testbed bestaat erin vergelijkingen 3 t.e.m. 6 per tijdstap op te lossen, en zo de positie van de drone te bepalen. Het exact oplossen van dit stelsel van 12 differentiaalvergelijkingen is zeer complex, daarom is het aangewezen om hier een numerieke methode voor te gebruiken. Wij hebben gekozen voor de voorwaartse methode van Euler (zie vergelijking 7), met als stapgrootte (h) de tijdstap. Dit is één van de eenvoudigere numerieke methodes, en behaalt, tot nu toe, de nodige precisie.

$$x_{n+1} = x_n + h \cdot f(t, x_n) \qquad [14] \tag{7}$$

Een andere benadering die we maken is de onderlinge onafhankelijkheid van de differentiaalvergelijkingen. We beschouwen in elke differentiaalvergelijking de anderen constant op hun vorige waarde. Dit toegepast op het stelsel van vergelijkingen, met $x[0]$ t.e.m. $x[k]$ als de te bepalen variabelen, levert dan de volgende te implementeren differentievergelijkingen:

$$\begin{cases} x[0]_{n+1} = x[0]_n + dt \cdot f(t, x[0]_n, \cdots x[k]_n) \\ \vdots \\ x[k]_{n+1} = x[k]_n + dt \cdot f(t, x[0]_n, \cdots x[k]_n) \end{cases} \tag{8}$$

Dit geeft de 12 formules die de physics engine moet uitrekenen in elke tijdsstap. Drie van de 12 bekomen waarden, namelijk de drie parameters van oriëntatie, worden dan niet opgeslagen. Ze worden gebruikt om de transformatiematrix tussen wereld- en drone-assenstelsel te roteren zodat deze de correcte oriëntatie bevat. Het roteren van een matrix gebeurt door vermenigvuldigen van de matrix met een rotatiematrix. De heading, pitch en roll die volgens de specificaties berekend moeten worden voor de autopilot, kunnen uit de transformatiematrix berekend worden aan de hand van de in de opgave vermelde formules.

3.3 Assenstelsels

Florian

In de voorstelling van de wereld, wordt gebruik gemaakt van een assenstelsel dat daar vast in zit, en dus niet transleert of roteert. Dit noemen we het wereldassenstelsel, en dit is het assenstelsel waar het testbed hoofdzakelijk mee werkt. Het andere assenstelsel dat door het testbed gebruikt wordt, is het drone-assenstelsel. Dit zit vast aan de drone volgens de vereisten uit de opgave. Alle berekeningen in de physics engine vinden plaats in dit assenstelsel, en de bekomen vectoren worden op het einde terug naar het wereldassenstelsel getransformeerd. Beide assenstelsels zijn rechtsdraaiend, en bij het opstarten van het testbed zullen beide assenstelsels dezelfde oriëntatie hebben.

Om coördinaten tussen deze assenstelsels uit te wisselen, maakt de physics engine gebruik van een transformatiematrix. Deze wordt geïnitialiseerd op de eenheidsmatrix, aangezien beide assenstelsels initieel dezelfde oriëntatie hebben. Hierna wordt de transformatiematrix met de drone mee geroteerd, zoals beschreven in de vorige sectie.

3.4 Ontwikkeling

Florian

De physics engine bleek al heel snel een zeer belangrijke component in het hele project. Daarom zijn we dan ook zo snel mogelijk begonnen met het implementeren ervan. Eerst kwam een studie van de mechanica en fysica [9], met dan een eerste implementatie. Het probleem met de eerste implementatie was dat we de impulsmoment-wet (2) niet gebruikten, maar $\sum \vec{M} = I\vec{\alpha}$. Dit vereenvoudigde de situatie veel te veel en leidde in de derde week al tot een nieuwe versie van de berekeningen van de rotaties.

Deze nieuwe versie bleek op het eerste zicht een correctere implementatie te zijn, maar achteraf gezien zat deze nog vol grote fouten en problemen. Een van de grote hindernissen in dit project bleek dat we week na week nog meer problemen bleven vinden in de physics engine. Dit zorgde voor problemen bij de autopilot die we aan het ontwerpen waren, aangezien de fysica in de software bleef veranderen. Uiteindelijk hebben we besloten om terug van nul te beginnen en met de verworven kennis een volledig nieuwe implementatie te ontwerpen. Deze nieuwe implementatie bleek snel veel beter te zijn dan de vorige versie, en bleek in vergelijking met het provided testbed[4] een zeer gelijkaardig gedrag te vertonen.

[4] voorziene testbed [5]

Door de vele problemen bij het ontwikkelen van de physics engine, heeft het totale project veel vertraging opgelopen. Als we meer tijd hadden geïnvesteerd in het testen van de implementatie, hadden we waarschijnlijk de fouten in de implementatie sneller gevonden, en hadden we minder tijdsdruk gehad bij de fine-tuning van de autopilot, aangezien deze sterk afhankelijk is van de physics engine.

4 Rendering

Arno

Voor dit project willen we een duidelijke voorstelling uitwerken van de drone en de kubussen die hij kan zien in de wereld. Om de positie van de drone duidelijk af te beelden is het handig om vanuit meerdere posities te kijken. De manier waarop we dat hebben aangepakt wordt in deze sectie besproken.

Een volledig overzicht van rendering kan je vinden op figuur 1 op pagina 8, hierin staan ook de klassen die in sectie 4.3 worden besproken.

4.1 Overwegingen

Arno

De grafische voorstelling van onze wereld wordt door *LWJGL* voorzien. We hebben hiervoor gekozen omdat dit volgens ons één van de beste Java opties is. Andere keuzes zoals *JMonkey* en *JavaFX 3D* waren minder geschikt. *JMonkey* bevat veel meer functionaliteiten dan we nodig hebben, we geven dus een voorkeur aan het meer lightweight alternatief, *LWJGL*. *JavaFX 3D* daarentegen heeft een kleinere groep actieve gebruikers en bevat essentiële onderdelen niet.

In het begin hebben we een tutorial [6] gevolgd om de basis van onze software op te bouwen. Hierin staat uitgelegd hoe de basis van een 3D rendering project opgezet moet worden. De code die we schrijven is in essentie pure *OpenGL* code met hier en daar stukken Java. De vormen die gerenderd worden, worden doorgegeven aan de GPU in vertex buffer objects en vertex array objects [11][8]. Hiermee kunnen we de berekeningen doen op de GPU i.p.v. op de CPU, wat alles sneller maakt. Het eerste beeld genereren van een kubus heeft heel wat tijd gekost. Door de robuste opbouw waren alle toekomstige ontwikkelingen beduidend sneller.

Voor de projectie op het scherm hebben we gebruik gemaakt van projectie-, model- en world-matrices [2]. In *LWJGL* is er geen Camera class dus hebben we die zelf geïmplementeerd, ook hier aan de hand van hiervoor benoemde tutorial, door de world objects met een worldViewMatrix te vermenigvuldigen. Hierdoor beweegt alles eigenlijk rond de camera,met de camera statisch in de oorsprong, richting de negatieve Z-as van het wereldassenstelsel.

De eerste weken van het project werkten sommige leden van onze groep met een Macbook. We hebben support voor deze toestellen na meerdere issues stopgezet. Het grootste probleem dat we tegenkwamen was *LWJGL* tezamen runnen met *JPanel*'s [13]. Om dit netjes op de lossen zou de helft van ons project moeten sneuvelen, waardoor we dus gezamenlijk besloten hebben om de Mac gebruiker verder te laten werken op Ubuntu.

Een nadeel aan *LWJGL* is dat het niet makkelijk is om tekst te renderen in een overzichtelijke GUI. We vonden het onnodig om nog een extra library toe te voegen aan het project dus hebben we opties zoals *Nanovg* [5] niet gebruikt. Al de tekst field based GUI's zijn dus verhuisd naar *Swing*, meer daar over in sectie 7 op pagina 17. Momenteel werken we aan een uitbreiding waarbij we de *Swing* GUI's renderen in de *LWJGL* context, dit gaat onze renderomgeving veel ordelijker maken.

[5] een vector image rendering library voor LWJGL

4.2 View Ports

Arno

Om de verschillende camera's te tonen, hebben we enkele opties geprobeerd. Meerdere schermen tonen zorgde voor problemen met context switching in *OpenGL* en werken met een FBO[6] bleek veel te complex voor wat we eigenlijk nodig hebben, dus hebben we voor viewports gekozen. Viewports staan toe om in een bepaalde regio van je *LWJGL* context te renderen. Deze optie vormde een veel kleinere workload om te implementeren dan FBO's of multiple window rendering en is nog steeds zeer efficiënt.

Het *LWJGL* scherm toont op elk moment meerdere beelden, elk komend van een andere camera in de wereld. Hieronder komt een opsomming van alle camera's die te zien zijn, en hoe ze eventueel bestuurd kunnen worden.

4.2.1 Chase Camera

Een *chase camera* is een camera die de drone volgt op een bepaalde afstand en enkel rond de verticale as draait. Je kan dus de bewegingen van de drone zien met de staart gericht naar de camera. Deze camera geeft een duidelijk beeld over de pitch en roll van de drone.

De camera volgt de drone door de rotatie van de camera over de Z-as gelijk te stellen aan die van de drone. De positie van de camera bepalen we als volgt:

$$\{x + (d * \sin\theta), y, z + (d * \cos\theta)\} \tag{9}$$

Met x, y en z de posities van de drone, d de afstand tot de drone en θ de heading vector van de drone. De heading vector is de richting waarin de drone vliegt, geprojecteerd op het XZ-vlak.

4.2.2 Drone Camera

Deze staat gepositioneerd op het massacentrum van de drone en heeft dezelfde oriëntatie. De camera beschouwt de drone als een transparant voorwerp en geeft deze dus niet weer. De achtergrond van deze camera is wit zoals gespecificeerd in de opgave. Voor motion planning sturen we het gerenderde beeld van deze camera met algoritme 1 door naar de autopilot. Dit bespaart veel werk aangezien er geen raycast of rasterization moet worden voorzien.

4.2.3 Free Camera

Met een free camera kan je als gebruiker zelf rondbewegen en zo de drone observeren. De controls kan je vinden in tabel 1. Ook kan je met een right click van de muis of trackpad de richting van de free camera aanpassen.

Tabel 1: Free camera keyboard layout

Richting	Boven	Onder	Links	Rechts	Versnellen	Screenshot	Toggle Ortho
Toets	W	S	A	D	SHIFT	C	R

[6]Frame Buffer Object

4.2.4 Orthografische Camera

De top en right orthographic camera hebben we gemaakt aan de hand van een orthografische projectie omvat in JOML [7]. Zij baseren zich op een projectie matrix beschreven door Song Ho Ahn [1]. Het grootste probleem dat we hiermee ondervonden was een *scope* kiezen voor de orthografische camera. De huidige setup voor de dimensies van de camera's laat het gebied voor de drone zien en, voor de rechtse orthografische camera, een groot deel eronder zien. Als we ooit volledig moeten kunnen ronddraaien met de drone dan gaan we deze dimensies veranderen. Maar dit is een slechts een kwestie van variabelen aanpassen. Wanneer je de simulatie opstart is de orthografische camera niet direct zichtbaar, hun rendering kan getoggled worden met de R toets.

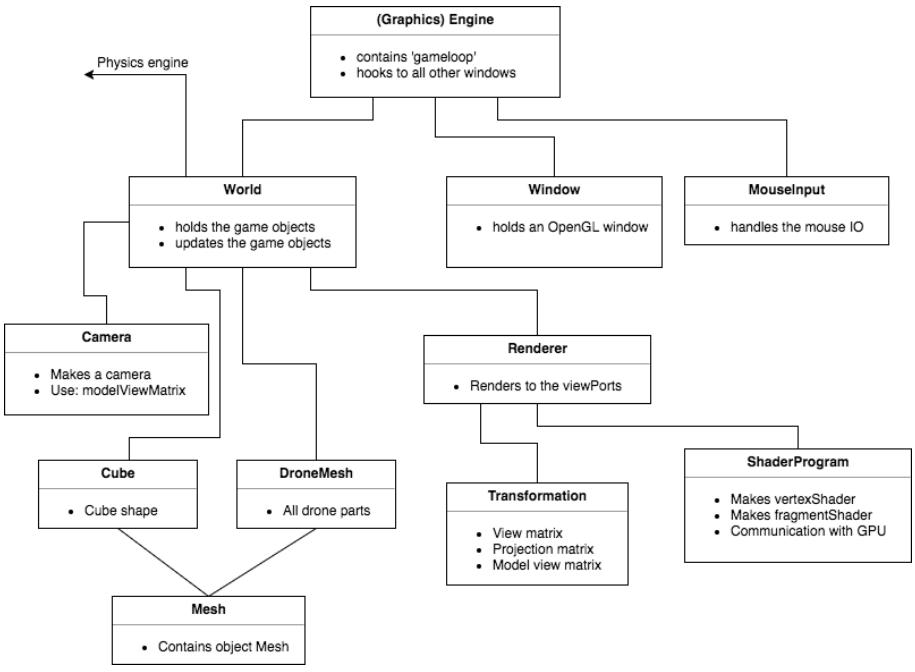
4.3 Wereld

Arno

De beschrijving van de wereld is relatief eenvoudig. Elk item in de wereld is een eigen object met een *Mesh*. Dit laatste is een verzameling van punten waardoor driehoeken worden getrokken om zo 3D-figuren te creëren. We hebben ervoor gekozen om niet met *.obj* files [8] te werken omdat het niet moeilijk is om snel een voorstelling te maken van het vliegtuig en van de kubussen.

De wereld bevat een update methode die opgeroepen wordt elke keer er een update in de gameloop gebeurt. Op deze manier worden de interne voorstellingen van alle objecten up-to-date gehouden. Deze updates houden rekeningen met de physics engine. Dit gedrag is te observeren op figuur 1 op pagina 8. Een wereld inladen vanuit een file is zeer makkelijk. Je geeft voor alle cubes de x, y en z coördinaten op en eindigt met een newline.

In de GUI die het testbed opstart moet men dan gewoon dit wereldbestand selecteren. In die interface kunnen er indien gewenst ook nog aanpassingen gebeuren aan de instellingen van de drone en de world config. Meer daarover in hoofdstuk 7.



Figuur 1: Diagram overzicht world en rendering.

[7] Java math library for OpenGL
[8] met zulke .obj bestanden kan je voorgedefinieerde modellen in laden en dan als drone gebruiken

4.4 Screenshot

Arno, Florian

De besturing van de drone, gebeurt op basis van een camerabeeld. Aangezien dit camerabeeld al op het *LWJGL* scherm getoond wordt, leek het ons het eenvoudigst om de afbeelding van het scherm af te lezen. Dit doen we aan de hand van he volgende algoritme. Het is gebaseerd op een Stackoverflow post [7] maar met aanpassingen voor compatibiliteit met Mac laptops met een Retina display. Deze hebben namelijk een hogere pixeldichtheid en dit zorgde voor problemen in het originele algoritme.

```
1 Function screenShot()
      input  : none
      output: A byte[] containing the image
2     ByteBuffer fb ← ByteBuffer met dimensions (width*heigth*3)   // 3 bytes per RGB pixel
3     Laad pixels van drone viewport in fb                          // glReadPixels(...)
4     Byte[] pixels ← cast buffer to a Byte[]
5     return pixels
```

Algorithm 1: Screenshot algoritme

5 Beeldherkenning

Tomas, Toon

Het doel van de beeldherkenning is het identificeren van de verschillende kubussen in beeld en het zo nauwkeurig mogelijk schatten van hun positie binnen de wereld. Het algoritme begint met de identificatie van de verschillende kubussen. Er wordt geïtereerd over al de pixels van het ontvangen beeld en iedere pixel wordt geclassificeerd als deel van een kubus of de achtergrond. Deze classificatie gebeurt a.d.h.v. het gegeven dat geen kubussen dezelfde combinatie van H- en S-waarde kunnen hebben binnen de HSV-kleurruimte. Voor iedere gevonden kleurencombinatie wordt een cube-object aangemaakt. Binnen deze cube wordt een lijst met de verschillende pixels waaruit hij wordt opgemaakt bijgehouden.

5.1 Afstandsschatting

Tomas, Toon

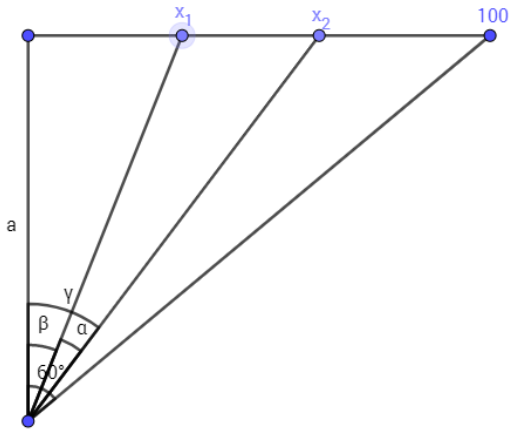
Eens alle kubussen gevonden zijn moet de afstand tussen drone en kubus geschat worden om later de positie van de kubus te kunnen bepalen. De schatting gebeurt door eerst op zoek te gaan naar de twee punten van de kubus die het verst verwijderd zijn van elkaar. Eenmaal die punten gevonden zijn berekent het programma de hoek tussen deze punten vanuit de drone. Gegeven dat het over een eenheidskubus gaat kan de afstand geschat worden met driehoeksmeetkude.

De eerste stap is op zoek gaan naar de twee pixels van de kubus die het verst van elkaar verwijderd zijn op het beeld. Deze twee pixels zijn of van hetzelfde vlak (ze hebben dezelfde kleur) en vormen de diagonaal van dit vlak, of ze zijn een verbinding van één van de diagonalen van de kubus. De lengte van deze diagonalen is gekend voor eenheidskubussen en kunnen dus helpen met de afstandsschatting. Om niet alle punten met elkaar te vergelijken gebruiken we eerst het Jarvis March-algoritme[10] om de convex omhullende van de verzameling pixels van de kubus te bepalen. Omdat het over een convexe figuur gaat zijn de twee punten die het verst verwijderd liggen van elkaar deel van de convex omhullende. Door enkel met de punten van de convex omhullende te werken besparen we het programma veel rekenwerk. We verkozen het Jarvis March algoritme ($O(nh)$) met n het aantal punten en h het aantal punten van de convex omhullende) boven andere algoritmes (meestal $O(n \log n)$) omdat een kubus maar een relatief klein aantal punten op zijn convex omhullende heeft. Dichtbijzijnde kubussen zullen veel pixels hebben en daar zal $\log n > h$. Het zijn net die kubussen waarvoor het algoritme het meest moet rekenen, maar het Jarvis

March-algoritme minimaliseert het rekenwerk door de convex omhullende te bepalen.

Het berekenen van de hoek tussen de rechten die van de drone naar de twee pixels gaan, gebeurt in twee stappen. Eerst berekent het algoritme de hoek tussen de x- en de y-coördinaten. Vervolgens combineren we de twee: $\alpha = \sqrt{(angleX)^2 + (angleY)^2}$ waarin α de gezochte hoek is en $angleX$ en $angleY$ respectievelijk de hoek tussen de x-en y-coördinaten.

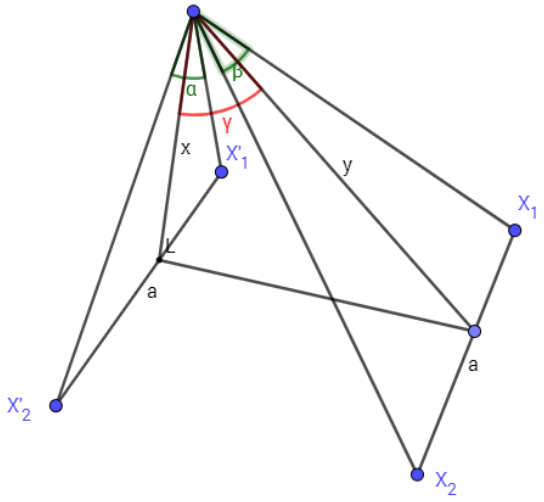
De hoek tussen de x- en y-coördinaten van twee pixels gebeurt op analoge manier. We werken hieronder enkel de afleiding uit voor de hoek tussen de x-coördinaten. In figuur 2 is het onderste punt de drone en x_1 en x_2 zijn de x-coördinaten als ze op de horizontale as zouden staan. We zijn op zoek naar α. Geweten is: $a = \frac{100}{tan(60°)}$, $tan(\beta) = \frac{x_1}{a}$ en $tan(\gamma) = \frac{x_2}{a}$. Hieruit volgt: $\beta = artan(\frac{x_1}{100} * tan(60°))$ en $\gamma = artan(\frac{x_2}{100} * tan(60°))$. Dus $\alpha = \gamma - \beta = artan(\frac{x_2}{100} * tan(60°)) - artan(\frac{x_1}{100} * tan(60°))$.



Figuur 2: bepalen van hoek tussen x-coördinaten op de horizontale as.

Deze formule berekent de hoek als de twee pixels op de horizontale as van het beeld zouden liggen maar dit is vaak niet het geval. Dus moet, met de gemiddelde y-coördinaat van de pixels, de gevonden hoek nog eens aangepast worden. In figuur 3 zijn X_1 en X_2 de uiterste punten en X_1' en X_2' hun projecties op de horizontale as. Neem α de eerder gevonden hoek, β de te vinden hoek en γ de hoek tussen de horizontale as en de gemiddelde y-coördinaat. X is de afstand van de drone tot de kubus geprojecteerd op de horizontale as en y de werkelijke afstand van drone tot kubus. Dan is $tan(\frac{\alpha}{2}) = \frac{\frac{a}{2}}{x}$, $tan(\frac{\beta}{2}) = \frac{\frac{a}{2}}{y}$ en $cos(\gamma) = \frac{x}{y}$. Dus is $tan(\frac{\beta}{2}) = \frac{x * tan(\frac{\alpha}{2})}{y}$. De gezochte hoek $\beta = 2 * artan(cos(\gamma) * tan(\frac{\alpha}{2}))$ is de hoek tussen de x-coördinaten van de twee uiterste punten.

Eenmaal de hoek gekend is gaan we ervan uit dat de twee uiterste punten van de kubus en de positie van de drone een gelijkbenige driehoek vormen. Zo benaderen we een schatting voor de afstand. Als de twee uiterste punten een verschillende kleur hebben, zijn ze van verschillende vlakken van de kubus. Dit betekent dat de twee punten een diagonaal van de kubus vormen ende afstand ertussen $\sqrt{3}$ is. Omdat het om een gelijkbenige driehoek gaat bestaat er volgend verband tussen de afstand x en de eerder gevonden hoek α: $tan(\frac{\alpha}{2}) = \frac{\frac{\sqrt{3}}{2}}{x}$. Herschrijven geeft voor de afstand $x = \frac{\frac{\sqrt{3}}{2}}{tan(\frac{\alpha}{2})}$. Als de twee uiterste punten dezelfde kleurwaarde hebben, ziet de drone maar één vlak en is de afstand tussen die twee punten $\sqrt{2}$. Dan is de afstand tussen de drone en de kubus $x = \frac{\frac{\sqrt{2}}{2}}{tan(\frac{\alpha}{2})} + 0.5$. We tellen op met 0.5 omdat we de afstand tot het midden van de kubus willen weten en niet tot het voorste vlak.



Figuur 3: bepalen van hoek

Er zijn dus twee benaderingen. Enerzijds is er het feit dat de geziene pixels niet altijd nauwkeurig zijn. Als een pixel voor de helft gekleurd zou moeten zijn, zal hij helemaal gekleurd zijn. Als hij daarentegen 45% gekleurd zou moeten zijn, zal hij helemaal niet gekleurd zijn. Vooral voor grote afstanden zal dit onnauwkeurigheden in het resultaat geven. Anderzijds is er de stap waarbij wij ervan uitgaan dat de rechte van de drone naar de kubus loodrecht op de diagonaal van de kubus staat en zo een gelijkbenige driehoek vormt. De maximale fout komt hierbij voor wanneer één zijvlak nog net zichtbaar is. De fout hierop zal nooit meer zijn dan de werkelijke afstand vermenigvuldigd met $\frac{\sqrt{3}}{\sqrt{2}}$.

5.2 Locatiebepaling

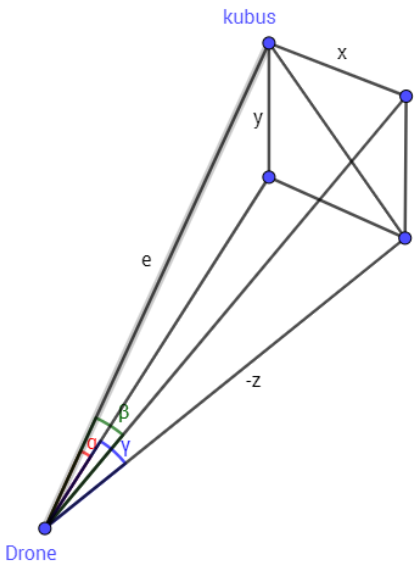
Tomas, Toon

Om te beginnen moet voor iedere kubus nagekeken worden of hij aan de rand van de afbeelding ligt of achter een andere kubus. Wanneer dit het geval is wordt de kubus genegeerd en wordt er nog geen locatie voor bepaalt. Nakijken of een kubus aan de rand ligt is eenvoudig. Een algoritme itereert over alle punten van de eerder gevonden convex omhullende en kijkt voor iedere van deze pixels of hij aan de rand ligt. Kijken of een kubus achter een andere ligt is complexer. We kunnen alweer voor iedere kubus de convex omhullende bekijken en voor ieder punt van die omhullende controleren of hij een kleur van een andere kubus raakt. Als dit het geval is, zijn er nog twee opties: ofwel ligt de kubus die we aan het inspecteren waren achter de kubus met de nieuwe kleur, of de nieuwe kubus ligt achter deze kubus. Gegeven dat de kubussen ver genoeg uit elkaar liggen, zal de kubus die het dichtst bij ligt ook het meeste pixels kleuren. Zo vinden we welke kubus achteraan ligt en we voegen die kubus toe aan de lijst met kubussen die we moeten negeren. Een versie in pseudocode van het beschreven algoritme staat hieronder in algoritme 2.

```
   input  : cubes: a list of all cubes found
   output: A list of cubes behind other cubes
 1 ArrayList iList ← new ArrayList;
 2 for cube : cubes do
 3 |   if cube is on border then
 4 |   |   iList.add(cube)
 5 |   else
 6 |   |   ArrayList tCubes ← arraylist of all cubes that touch cube;
 7 |   |   for tCube : tCubes do
 8 |   |   |   if cube.size < tCube.size then
 9 |   |   |   |   iList.add(cube)
10 |   |   |   else
11 |   |   |   |   iList.add(tCube)
12 |   |   |   end
13 |   |   end
14 |   end
15 end
16 return iList
```

Algorithm 2: finding cubes on border or behind other cubes

De tweede stap is het bepalen van de locatie van iedere kubus in het assenstelsel van de drone. Dit is het assenstelsel waarbij de kijkrichting volgens de negatieve z-as ligt, de x-as naar rechts en de y-as naar boven. Iedere coöordinaat wordt apart berekend. De berekening van de x- en y-coördinaten gebeurt op analoge manier. De hoeken α en β in figuur 4 zijn op dezelfde manier als hierboven beschreven bepaald. Neem e de schatting van de afstand, dan zijn x en y: $x = e * sin(\beta)$, $y = e * sin(\alpha)$. γ is de hoek tussen twee pixels op de horizontale as. Dan is $z = e * cos(\alpha) * cos(\gamma)$.



Figuur 4: Locatiebepaling.

Ten derde moet de gevonden locatie voor iedere kubus getransformeerd worden naar de locatie in het wereldassenstelsel. Hiervoor gebruiken we een transformatiematrix.

6 Motion Planning

Flor, Bernardo

Eens de positie van een kubus geschat is, moet de drone naar die positie kunnen manoeuvreren. Hiervoor is een systeem nodig dat op basis van de huidige toestand van de drone ervoor zorgt dat de volgende toestand dichter bij het doel ligt. Als input beschikt het motion planning systeem enkel over gegevens uit de beeldherkenning en de andere gegevens omschreven in AutopilotInputs. De output van de motion planner moet het gewenste resultaat leveren door enkel de vleugels, stabilisators, en thrust van de drone aan te sturen. Zo een controlesysteem is ook lichter dan de physics engine, dit komt doordat er minder gegevens beschikbaar zijn en dus minder berekeningen worden gedaan. Een relatief ruwe benadering zal dus voldoende zijn om de werking van de physics engine te simuleren. De gekozen aanpak wordt hieronder in meer detail beschreven.

6.1 Theorie

Flor, Bernardo

Alle doelen liggen binnen een cilinder met een straal van 10 meter, met een onderlinge z-afstand van 40 meter. Dit reduceert het probleem tot 2 vrijheidsgraden: de x- en y-posities van de drone op het moment dat die een kubus voorbijvliegt in de z-richting. Om deze reden bestaat de motion planning uit twee delen. adjustHeight zorgt dat de drone op ongeveer dezelfde y-positie komt als de kubus. adjustHeading laat de drone zich oriënteren in de richting van de kubus. Deze methodes werken aan de hand van *PID controllers*. Een PID controller is een wiskundig werktuig dat als input een doel (*setpoint*) en een fout (*error*, afstand tot het doel) meekrijgt, en geeft dan een output die van de fout afhankelijk is. Dit soort controllers wordt vaak gebruikt in systemen die moeilijk deterministisch te beschrijven zijn, typisch systemen met inertie, en daarom zijn deze zeer geschikt voor dit project.

PID staat voor *proportional-integral-derivative*. *Proportional* verwijst naar het basisconcept van een controller: hoe verder deze van het doel verwijderd is, hoe sterker de PID zal corrigeren. Enkel de proportionele component gebruiken kan echter een constante fout niet altijd wegwerken. Het effect van de zwaartekracht is hier een voorbeeld van. Daarom is de integraal-term ingevoerd. Deze zal de compensatie versterken als gedurende lange tijd de fout blijft aanhouden. De integraal-term kan enerzijds *overshoots* of pieken veroorzaken, en anderzijds te traag reageren. Bovendien kan zo een controller nog leiden tot oscillerend gedrag, wat in veel gevallen ongewenst is. De afgeleide-term (*derivative*) houdt rekening met de verandering in de fout op elk tijdstip. Dit heeft twee effecten: de grote pieken in de output worden afgevlakt, en de reactiesnelheid wordt verbeterd. 75% van alle controllers zijn echter slechts PI-controllers. Dit komt omdat de afgeleideterm in werkelijke systemen onvoorspelbare invloed kan hebben op stabiliteit.

Het eigenlijk gedrag van een PID controller is zeer gevalsafhankelijk. De uitgang $u(t)$ van de controller wordt door de volgende formule beschreven[4]:

$$u(t) = K(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(\tau)}{dt}) \quad , \tag{10}$$
$$\text{met} \quad \frac{K}{T_i} = K_i \quad \text{en} \quad K \cdot T_d = K_d \quad ,$$

waarbij e(t) de error op tijdstip t voorstelt. De gewichten of *gain* voor elk van de drie termen K_p, K_i en K_d moeten experimenteel bepaald worden. Dit proces heet *tunen*.

De drone heeft 4 vrijheidsgraden die gecontroleerd kunnen worden: pitch, heading, roll, en snelheid. Deze worden gecontroleerd door respectievelijk de pitch-, heading-, roll- en thrust-PID.

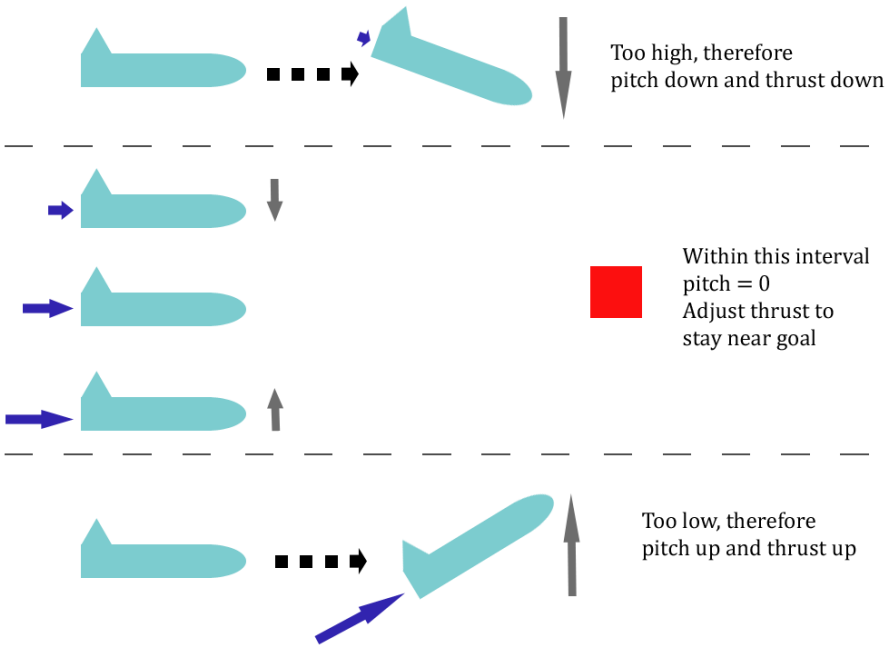
Afhankelijk van de huidige situatie waarin de drone zich bevindt krijgt elk van de PID's een setpoint. De opeenvolging van deze verschillende toestanden en setpoints moeten er uiteindelijk voor zorgen dat het vliegtuig zich als gewenst gedraagt. In elke toestand wordt een verschillende combinatie van roll, pitch, heading, en de snelheid nagestreefd. Deze worden onrechtstreeks door de PID controllers gecontroleerd: pitch door de horizontale stabilisator, heading door de verticale stabilisator, roll door de twee vleugels, en de snelheid door de thrust. Al deze factoren hebben een invloed op elkaar, waardoor er op elk restricties ingevoerd moeten worden zodat de drone stabiel blijft vliegen. Elke verandering in werkwijze vraagt mogelijk het opnieuw tunen van de controllers.

6.2 Implementatie

Flor, Bernardo

Er zijn dus twee delen aan de motion planning, enerzijds voor de x-coördinaat en anderzijds voor de y- coördinaat. De methode *adjustHeight* regelt de y-component en de methode *adjustHeading* regelt de x-component.

De methode *adjustHeight* herkent 5 mogelijke situaties. Is de positie van de drone veel lager dan het doel, dan pitcht deze omhoog en drijft deze de thrust op om in de y-richting te versnellen. Hierbij spelen twee PID controllers een rol, namelijk die voor de horizontale stabilisator en die voor de thrust. Ligt de drone dichter bij het doel, dan gaat die zich horizontaal oriënteren en trager stijgen enkel op basis van lift van de vleugels die wordt opgevoerd door de thrust te verhogen. In het geval dat het vliegtuig hoger zou liggen dan het doel pitcht het naar beneden. Ook de thrust wordt door de PID omlaag geregeld zodat deze met een voorafbepaalde snelheid daalt. Dicht bij het doel wordt de val verder afgeremd door de drone terug horizontaal te pitchen. Hierdoor wordt voorkomen dat de PID in de volgende stap te sterk zou compenseren. Ligt het vliegtuig binnen een klein interval rond de doelhoogte, dan zal de thrust zo geregeld worden dat de verticale snelheid van de drone 0 wordt. Er is zowel voor thrust als voor pitch gekozen voor twee aparte PID's, 1 voor het naar boven bewegen en 1 voor het naar onder bewegen. Dit is zo gekozen zodat er twee verschillende tunings gehanteerd kunnen worden.



Figuur 5: Vijf verschillende situaties voor *adjustHeight*

De methode *adjustHeading* berekent op elke tijdstap de richting in het xz-vlak tussen de drone en het doel. Een PID controller stuurt de verticale stabilisator zodanig aan dat de heading van de drone samenvalt met deze richting. De richting verandert als functie van de positie van de drone en is dus tijdsafhankelijk, maar de PID controller moet dit kunnen compenseren. Deze draaibeweging leidt zelf tot een nieuw probleem. Door het verschil in absolute snelheid tussen de twee vleugels zal de ene meer lift genereren dan de andere, hierdoor gaat het vliegtuig rollen. Dit probleem is in principe makkelijk opgelost door de methode *adjustRoll*. *adjustRoll* zet de vleugels op een voorafbepaalde inclinatie, maar geeft dan beide ook een kleine afwijking die door een PID controller gegeven wordt. Deze afwijking is voor beide vleugels gelijk, maar tegengesteld in teken. Het gevolg hiervan is een moment in de tegengestelde zin, waardoor het vliegtuig stabiliseert. De combinatie van *adjustHeading* en *adjustRoll* moet voor stabiele en nauwkeurige draaibewegingen zorgen.

6.3 Drone

Flor, Bernardo

De drone heeft verschillende parameters, deze parameters hebben een grote invloed op het gedrag van de drone t.o.v. de motion planning. Hieronder is een overzicht gegeven van alle parameters van de drone. Deze parameters kunnen nog veranderen tegen de demo.

Tabel 2: Parameters drone

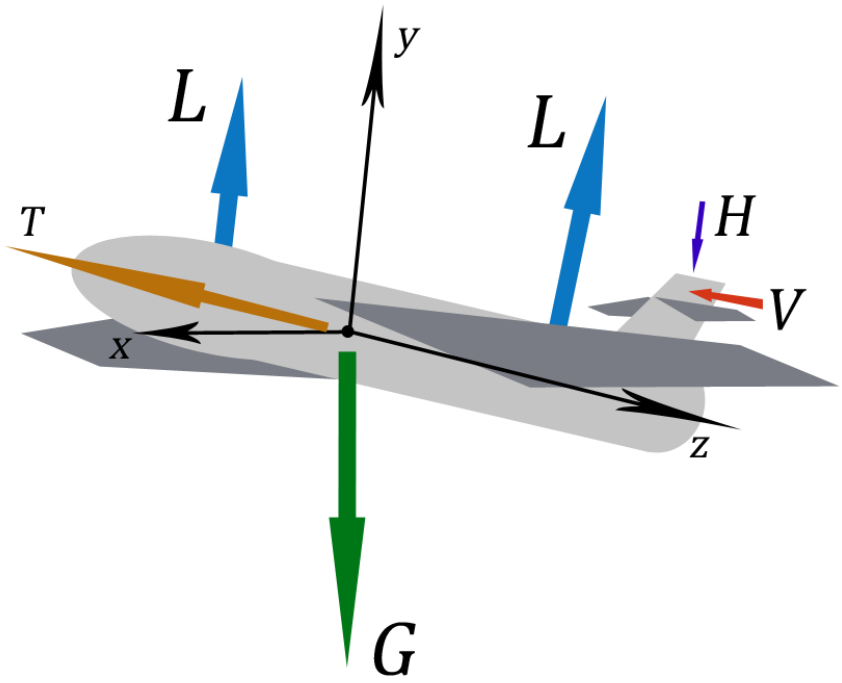
Parameter	**Waarde**
Gravity	9.81
Wing size	0.5
Tail size	0.5
Engine mass	0.25
Wing mass	0.25
Tail mass	0.125
Max Thrust	5
Max AOA	Π/4
Liftslope wings	0.1
Liftslope stabilisers	0.05
Angle of view	120°

6.4 Resultaten

Flor, Bernardo

Bij het schrijven van dit verslag kan de drone stijgen en dalen tot een hoogte van ongeveer 16 meter over een afstand van 50 meter. Dit is net niet snel genoeg om alle mogelijke doelen te bereiken. Dit ligt aan verschillende factoren. Zo is de snelheid van de drone $\pm 15 \frac{m}{s}$. Met deze snelheid zou de drone dus op iets meer dan 1 seconde tijd het hoogteverschil moeten overbruggen. Dit proberen we in de toekomst op te lossen door het gewicht te verkleinen.

De methode *adjustHeading* werkt nog niet naar behoren. De juiste heading wordt niet bereikt. Dit komt doordat het vliegtuig draait maar het een snelheid heeft niet overeenkomstig met de z-as van de drone. De drone gaat dus niet in een rechte lijn naar de kubus en de heading zou heel de tijd moeten worden aangepast. Dit kan opgelost worden door zoals bij *adjustHeight* het probleem in verschillende stappen onder te verdelen. Eerst wordt door te yawen de heading van het vliegtuig aangepast. Vanaf dat de drone de x-coördinaat goed genoeg benadertbegint de drone terug in de andere kant te yawen totdat de heading 0 is. Dan vliegt het vliegtuig rechtdoor naar de kubus. Deze implementatie is nog niet gebeurd maar proberen we voor de demo te realiseren. Als deze uitvoering onmogelijk zou blijken, bestaat de mogelijkheid om bochten te nemen op basis van het



Figuur 6: Voorstelling van inwerkende krachten op de drone.
L = lift, T = thrust, G = zwaartekracht, H = hor. stabilisator lift,
V = ver. stabilisator lift

bank-manoeuvre. Door het vliegtuig te laten rollen krijgt de lift een component in de x-richting. Die component dient dan als centripetale kracht, waardoor het vliegtuig een cirkelboog volgt. Deze aanpak is niet zonder problemen. De controle van een bank is moeilijker dan draaien met een yaw beweging. Dit wordt bemoeilijkt door het fenomeen van *adverse yaw*. *Adverse yaw* is een nefaste rotatie die optreedt in een bank-manoeuvre, als gevolg van het verschil in oriëntatie van de liftvectoren. Deze zou dan weer door de PID controller van de horizontale stabilisator gecompenseerd moeten worden.

De PID controllers voor pitch en thrust zijn heel goed op elkaar afgestemd. *adjustHeight* geeft consistent snelle en stabiele resultaten. De PID controllers voor roll werken afzonderlijk heel goed. Pas bij de combinatie met de PID voor yaw treedt er vreemd gedrag op. Dit doet vermoeden dat er een redeneringsfout zit in het verwerken van de output van deze laatste. De bovengenoemde verbeteringen kunnen vrij eenvoudig geïmplementeerd en getest worden. We verwachten hier wel nog problemen mee dus we weten niet in hoeverre we alles kunnen afwerken voor de deadline. De bewegingen naar boven en beneden kunnen wel getoond worden op de demo.

7 GUI

Arno

Al de vensters die we naast het main render gedeelte laten zien, zijn voornamelijk ontstaan doordat er nood aan was tijdens het productieproces. Vooral het setup venster is zeer nuttig geweest, dit heeft de hoeveelheid initialisatie overhead sterk naar beneden gehaald.

7.1 Testbed

Arno

De lay-out van de verschillende camera's en de bijkomende design keuzes kan je vinden in de subsecties van sectie 4. Alle bijkomende interfaces worden voorzien door *Swing*. Momenteel hebben we losstaande vensters naast het *LWJGL* frame en dat is behoorlijk rommelig. Er is een oplossing waarmee we deze kunnen renderen in een *OpenGL* context en dit gaan we voor de finale demo proberen te voorzien.

De GUI voor het testbed is best simpel; deze toont enkel de snelheid, de positie en de draaiing van de drone. Meer hebben we ook niet nodig omdat het merendeel van de gewenste functionaliteit al omvat wordt door het setup venster.

7.2 Setup

Arno

Dit venster bestaat uit drie verschillende onderdelen. Vanuit elk van de opties kan je de simulatie starten door op *Start Simulation* te drukken.

WorldGen Dit is het handigste onderdeel van de GUI. Dit deel laat de gebruiker toe een eigen wereld te maken en/of te selecteren. Als je één van de vooraf gemaakte werelden kiest worden alle andere settings die de user heeft gemaakt overschreven. Naast premade worlds kan je uit een dropdown menu kiezen om willekeurige cube posities of één cube te genereren alsook cubes inlezen vanuit een bestand. De resterende optie is opdracht *M2.3* uit de opgave [12] waar we de cubes in een cilinder generen met voorwaarden:

$$(x_i, y_i, z_i)_{i=1}^{5}$$
$$z_i = i * -40m$$
$$\sqrt{x_i^2 + y_i^2} \leq 10m$$

Config Staat de gebruiker toe om manueel opties in te vullen voor de drone. Men kan hierin alles aanpassen dat aan bod komt in de AutopilotConfig klassen. Als hier geen gebruik van wordt gemaakt zorgen we ervoor dat er default values gebruikt worden.

Drone Deze lijkt sterk op de bovenstaande. Hier wordt minder gekeken naar de fysische eigenschappen van de drone, maar eerder naar de locatie, rotatie en beweging van de drone. Zo kan je bij elke run de drone ook in een bepaalde situatie plaatsen

7.3 Autopilot

Arno

Hierin wordt een overzicht gegeven van de informatie die de Autopilot zelf weet alsook een beeld van de drone camera. De gebruiker kan zo kijken naar de informatie die de drone momenteel denkt binnen te krijgen en hoe hij reageert op situaties. De drone zijn acties kan je makkelijk volgen aan de hand van de sliders die zichzelf aanpassen aan de stand van de vleugels van de drone.

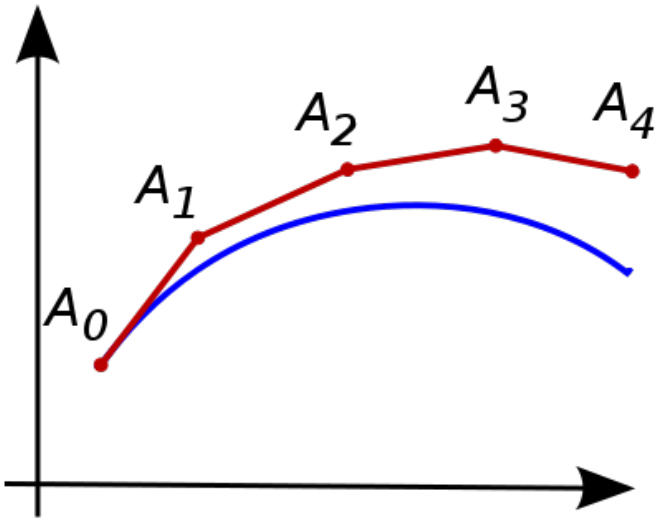
8 Testen

8.1 Physics Engine

Florian

Om de correctheid van de physics engine te testen, hebben we deze vergeleken met de resultaten van het provided testbed. Dit is een testbed voorzien door de begeleiders van het project [5], en implementeert een hogere orde numerieke formule bij de differentiaalvergelijkingen. Om onze implementatie hiermee te vergelijken, hebben we een autopilot geschreven die de positie en oriëntatie van de drone naar een bestand schrijft, elke keer dat de *timePassed(time)* methode opgeroepen wordt. Voor de rest voert deze log-pilot geen controle uit, en voorziet deze enkel de mogelijkheid om de vleugels op een bepaalde stand te houden. Men kan dan beide testbeds uitvoeren met deze log-pilot, en de bekomen waarden dan vergelijken en op grafieken zetten. De grafieken die bijgevoegd zijn in het verslag, staan wegens hun grootte in appendix A.

Een eerste test die we gedaan hebben, is alle vleugels op een neutrale stand te zetten, en dan de drone met een beginsnelheid van $10m/s$ te laten vallen. Dit leverde zeer vergelijkbare resultaten op, getoond in afbeelding 8 op pagina 23. Hier blijkt duidelijk dat onze physics engine het verloop van de physics engine uit het provided testbed goed lijkt te benaderen. De afwijking die we toch zien bij de pitch, is mogelijk te verklaren door de karakeristieke afwijking van de voorwaartse Euler methode. Aangezien de volgende waarde benaderd wordt met de huidige afgeleide, kan de volgende waarde een over- of undershoot zijn van de eigenlijke waarde, zoals geïllustreerd op afbeelding 7.



Figuur 7: De karakteristieke afwijking van voorwaartse Euler methode [3]

Dit experiment kan herhaald worden met een steeds kleiner wordende tijdsstap, wat zou moeten leiden tot convergentie. In appendix A staat een tweede figuur, namelijk figuur 9, die dezelfde situatie als hierboven toont, maar deze keer met een 10 keer kleinere tijdsstap voor ons testbed. Het verschil met figuur 8 is amper zichtbaar op de absolute plot, dus kan men op figuur 10 het verschil tussen provided en ons testbed zien, en dit van beide gevallen.

Uit deze figuur wordt het duidelijk dat de fout kleiner wordt bij een kleinere tijdsstap, en dit zowel bij de y- als de z-component. De fout bij de x-, heading- en roll-component blijft binnen de *java Float*-precisie, aangezien deze drie componenten op nul blijven gedurende de hele simulatie. De afwijking die te zien is op de grafiek van de pitch, als deze veroorzaakt zou worden door de reden besproken hierboven, zou moeten dalen met kleinere tijdsstap. Op de grafiek is echter te zien dat dit niet gebeurt, en de fout dus van ergens anders moet komen. Dit is hoogstwaarschijnlijk een

verschil in implementatie tussen beide testbeds, maar aangezien de code van het provided testbed niet beschikbaar is, kunnen we dit niet controleren.

8.2 Beeldherkenning

Toon, Tomas

Om het beeldherkenning algoritme, en meer bepaald de afstandsschatting, te testen hebben we een testwereld gebouwd waarin een kubus steeds een nieuwe, gekende positie aanneemt in functie van de tijd. Meestal begint hij in de buurt van de oorsprong en gaat hij volgens een bepaalde richting naar achter. Voor iedere nieuwe positie wordt de afstand geschat en vergeleken met de werkelijke afstand. De resultaten van deze schattingen worden geplot in een grafiek die ook de absolute fout weergeeft. Wat volgt zijn een aantal grafieken die uit dergelijke tests voortkomen.

Bij de volgende reeks testen kijkt de drone vanuit de oorsprong volgens de negatieve Z-richting.

Een eerste test laat de Z-coördinaat van de kubus geleidelijk aan afnemen terwijl de andere coördinaten onveranderd blijven. Het resultaat is te zien in figuur 11. Opmerkelijk is hier het trapsgewijze verloop van de geschatte afstand, verklaarbaar door de weergave van de kubus op het scherm. Het aantal pixels dat de kubus weergeeft verkleint naarmate de afstand toeneemt. Zo wordt de kubus op een afstand van 20m weergegeven door slechts 4 pixels, en dit blijft zo tot op een afstand van meer dan 50m, daarna is de kubus niet meer zichtbaar.

In een tweede test neemt de Z-coördinaat nog steeds af maar is er nu ook een toename van de X-coördinaat (zie figuur 12), en in een derde test neemt ook de Y-coördinaat toe (zie figuur 13). In deze grafieken valt op dat het trapsgewijze verloop sterk afgevlakt is in vergelijking met de eerste test. Dit is logisch aangezien deze situaties meer informatie over de kubus hebben. Zo zijn er bijkomende zijvlakken te zien. Ook het aantal pixels dat de kubus weergeeft, verandert vlotter naarmate de afstand toeneemt.

Eén van de slechtere gevallen vanuit theoretisch standpunt van het algoritme is te zien in de vierde test (figuur 14). Hier neemt de Z-coördinaat vijf keer zo snel af als de X-coördinaat toeneemt en is er dus steeds een klein deeltje van een zijvlak te zien. Dit betekent dat het algoritme rekent met de diagonaal van een kubus i.p.v. deze van een vlak. Voor korte afstanden overschat het algoritme dus altijd de afstand, omdat de schatting ervan uitgaat dat de rechte van de drone naar de kubus loodrecht staat op de gebruikte diagonaal, wat in dit geval niet waar is. Bij de grotere afstanden is het zijvlak niet altijd zichtbaar en zijn schattingen niet altijd te groot.

In een vijfde en zesde test houden we de Z-coördinaat onveranderd en laten we eerst enkel de X-coördinaat toenemen (figuur 15) en vervolgens zowel de X- en Y-coördinaat (figuur 16). In deze grafieken zien we een soort zaagblad verloop. Dit verloop is het gevolg van de weergave van de kubus op het scherm. Deze weergave kan ingebeeld worden als een soort harmonica-beweging. Eerst zullen er pixels bijkomen (de afstandsschatting is kleiner), dan verdwijnen er pixels aan de andere kant (de schatting is groter). In het begin van beide grafieken is de overgang van formule die de diagonaal van een vlak gebruikt naar die die de diagonaal van de kubus gebruikt zichtbaar.

In een tweede reeks testen kijkt de drone vanuit de oorsprong volgens een pitch van 45°.

We gaan verder met de zevende test waarin enkel de Z-richting aangepast wordt (figuur 17). We krijgen een grafiek die sterk lijkt op deze van de eerste test. Het trapsgewijze verloop is echter weer afgezwakt doordat het aantal pixels dat de kubus afbeeldt groter is, zodat de hoeken tussen de twee uiterste pixels nauwkeuriger geschat kan worden.

Bij de achtste test passen we de positie in de drie richtingen aan. Het resultaat is afgebeeld in figuur 18. We krijgen weer voldoende nauwkeurige benaderingen. Zelfs in de negende test, één van

de theoretisch slechtere gevallen, (figuur 19) treden geen problemen op.

Uiteindelijk doen we ook nog de test waarbij de kubus een vaste Z-coördinaat heeft en de X- en Y-coördinaten toenemen. De grafiek (figuur 20) gaat alweer op en neer door de manier waarop de beweging van de kubus wordt weergegeven. Ook hier is er een duidelijk overgang van het gebruik van de diagonaal van het vlak naar het gebruik van de diagonaal van de kubus voor het berekenen van de afstand.

Ten slotte hebben we nog drie testen met een pitch en een heading van 30°: figuur 21 geeft de schattingen als de kubus zijn Z-coördinaat afneemt, figuur 22 als ook de X- en Y-coördinaten toenemen en figuur 23 wanneer de X- en Y-coördinaten toenemen bij een vaste Z-coördinaat. Deze grafieken lijken sterk op de grafieken van dezelfde verplaatsingen bij het vorige geval. Het is niet onverwacht dat bij de drie gevallen de grafieken een gelijkaardig gedrag vertonen voor dezelfde situaties. De afstandsschatting houdt geen rekening met de stand van het vliegtuig. Hij zoekt de twee uiterste pixels van de kubus en bepaalt met die pixels de afstand. Het is pas wanneer de autopilot de locatie van de kubus gaat bepalen dat de locatie en de oriëntatie van de drone er toe doen.

8.3 Motion planning

Flor

Er zijn veel testen en aanpassingen vooraf gegaan aan de huidige versie van de motion planning. Dit op zowel het niveau van de drone (de waarden voor de parameters), de PID controllers (deze moesten getuned worden) en de implementatie zelf.

De eerste versie van de motion planning bevatte geen PID controllers maar werd volledig deterministisch opgesteld. Dit bleek al snel te moeilijk te worden naargelang er meer variabelen bij kwamen zoals de pitch, heading, roll... We hebben dan de beslissing genomen om met PID's te beginnen werken, de eerste versie werkte redelijk goed om de hoogte van het vliegtuig te besturen maar deze was ingesteld op de oude physics engine die niet correct was. Toen de nieuwe physics engine af was moesten alle PID controllers terug worden getuned totdat we terug een stabiele werking hadden. Deze implementatie was ook nog niet goed genoeg omdat het vliegtuig veel te snel vloog en te traag reageerde, hierop hebben we de grootte en massa van het vliegtuig sterk verminderd. Na het opnieuw tunen van de PID's zijn we dan tot de uiteindelijke implementatie gekomen die er nu is. Hierin merkten we wel dat de implementatie om de x-coördinaat aan te passen niet werkte, dat probleem hopen we tegen de deadline wel weggewerkt te krijgen.

Het tunen van de PID werd voornamelijk gedaan door het runnen van het programma en verschillende waarden uit te printen. Vooral de hoogte, y-snelheid, pitch en de inclinatie van de horizontale stabilisator waren heel waardevol voor het tunen van de pitch- en thrust-PID bij de methode *adjustHeight*.

Na het opnieuw aanpassen van het gewicht zal er hoogst waarschijnlijk opnieuw moeten getuned worden en dit gaan we voor de demo proberen te doen.

9 Besluit

Tomas, Flor

Het doel was om een virtueel testbed en een autopilot te maken. De autopilot moet de drone zo kunnen aansturen dat die naar verschillende eenheidskubussen kan vliegen. Ons virtueel testbed heeft een werkende physics engine die uitgebreid getest is. Deze hebben we naast het provided testbed laten runnen en presteerde gelijkaardig. Het testbed laat ook de wereld met de drone en de kubussen zien, dit door verschillende standpunten die als camera's zijn geïmplementeerd. Aan de hand van deze camera's is het gedrag van de drone in het testbed goed zichtbaar. Het beeld bekeken uit het standpunt van de drone en de status van de drone worden doorgegeven aan de autopilot via de *Java* API. De beeldherkenning kan op basis van de ontvangen afbeelding een schatting van de afstand maken. Deze schatting is voor meerdere situaties getest en geeft een nauwkeurige locatie. In de code die de locatie bepaalt a.d.h.v. de afstandsschatting zit nog een fout die we voor de demonstratie zullen herstellen. Met een schatting van de locatie van de kubus kunnen de hoeken van de vleugels en de thrust aangepast worden.

Voorlopig kan de drone enkel stijgen en dalen met behulp van PID controllers, die voldoende getest zijn. Ook zijn we bezig aan een methode die de drone kan laten draaien maar hiervan is nog niet duidelijk of deze af zal geraken voor de demo. Ons testbed werkt zeer goed en voldoet aan de mijlpalen, ook onze autopilot is bijna af maar hier is nog wel wat werk aan voor de demo.

Referenties

[1] S. H. Ahn, *Opengl projection matrix*, 2008.

[2] M. Alamia, *World, view and projection transformation matrices.* codinglabs.net, 2013.

[3] O. Alexandrov, *File:euler method.* Wikimedia Commons, 2007.

[4] K. J. Åström, *Control system design.* Caltech cursustekst, 2002.

[5] W. Baert, *p-en-o-cw-2017: Provided testbed.* Github, 2017.

[6] A. H. Bejarano, *3d game development with lwjgl 3.* Gitbook: Lwjglgamedev, 2015.

[7] R. Blax, *How to make a simple screenshot method using lwjgl.* Stackoverflow, dec 2011.

[8] K. Contributors, *Vaos, vbos, vertex and fragment shaders.* Khronos Wiki, 2009-2014.

[9] W. Desmet and J. V. Sloten, *Toegepaste Mechanica 2, deel dynamica - Cursustekst*, CuDi VTK, 2014, ch. II Dynamica.

[10] R. Jarvis, *Gift wrapping algorithm*, 1973.

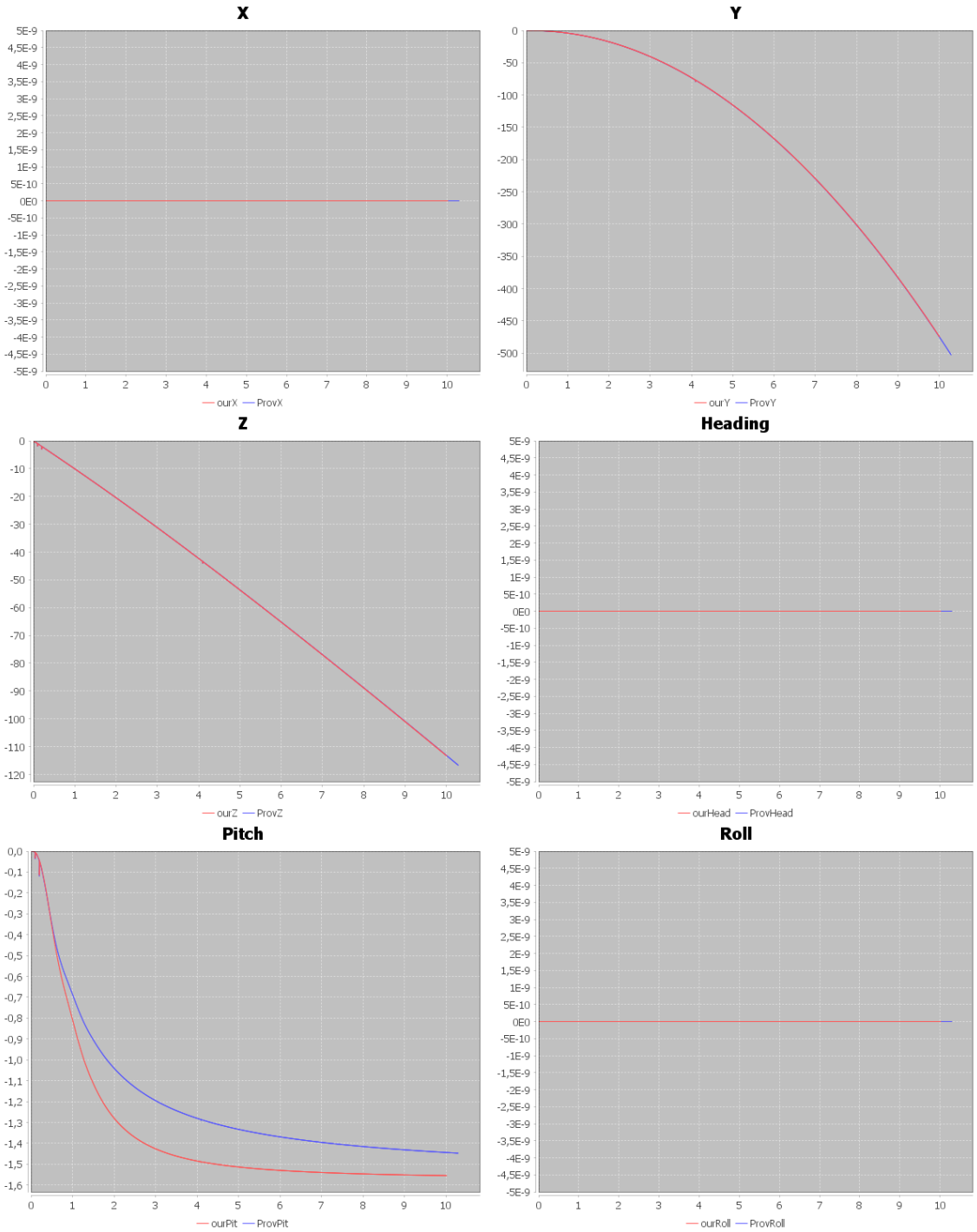
[11] opengl tutorial, *Vbo indexing.* opengl-tutorial.org, 2011.

[12] PenO Team 2017, *p-en-o-cw-2017: Opgave en ter beschikking gestelde bestanden.* Github, 2017.

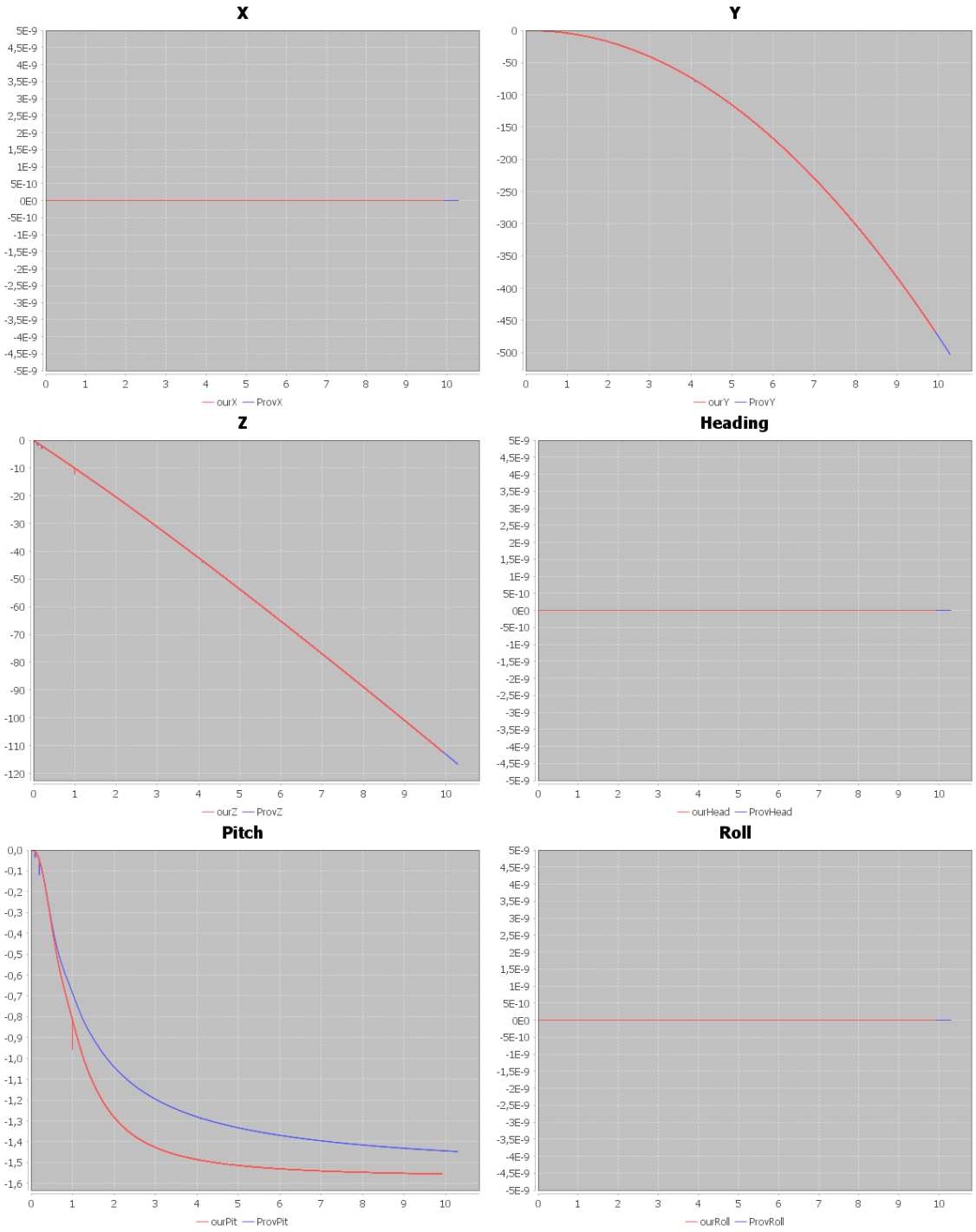
[13] sorifiend, *Lwjgl java.awt.headlessexception thrown when making a jframe.* Stackoverflow, oct 2017.

[14] M. Zeltkevic, *Forward and backward euler methods.* MIT web Course notes, 1998.

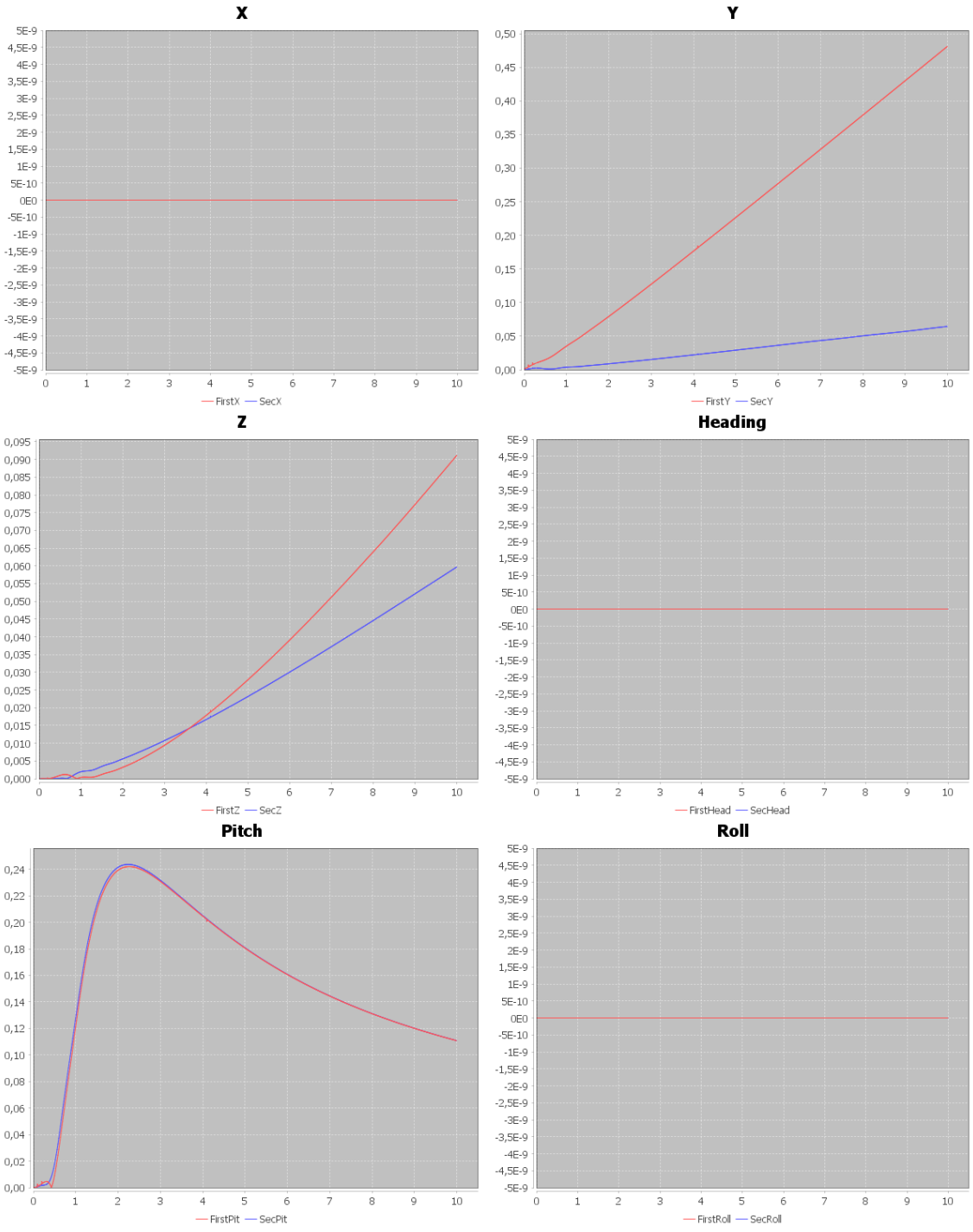
A Grafieken



Figuur 8: Resultaten van vergelijking tussen het provided testbed (blauw) en ons testbed (rood), voor de eerste 10 simulatieseconden, beide weergegeven in absolute waarden. Beide testbeds werken met een tijdsstap van $0.01s$

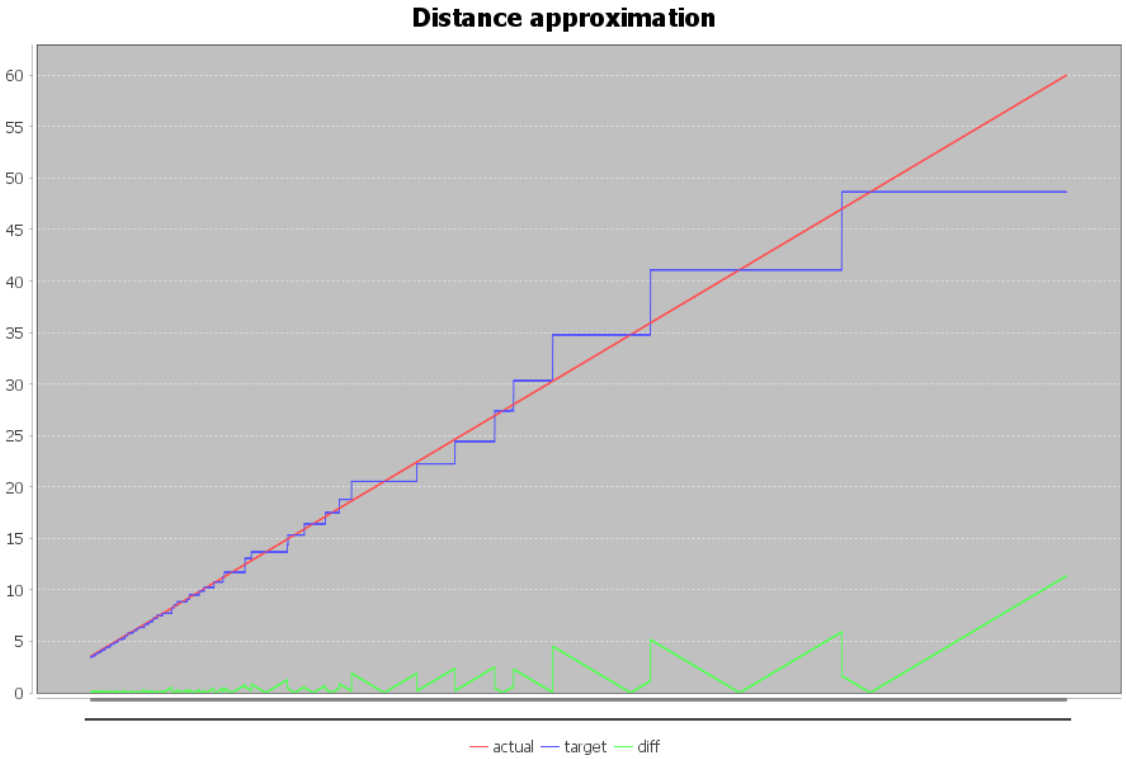


Figuur 9: Resultaten van vergelijking tussen het provided testbed (blauw) en ons testbed (rood), voor de eerste 10 simulatieseconden, beide weergegeven in absolute waarden. Het provided testbed werkt met een tijdsstap van $0.01s$, en ons testbed werkt met $0.001s$

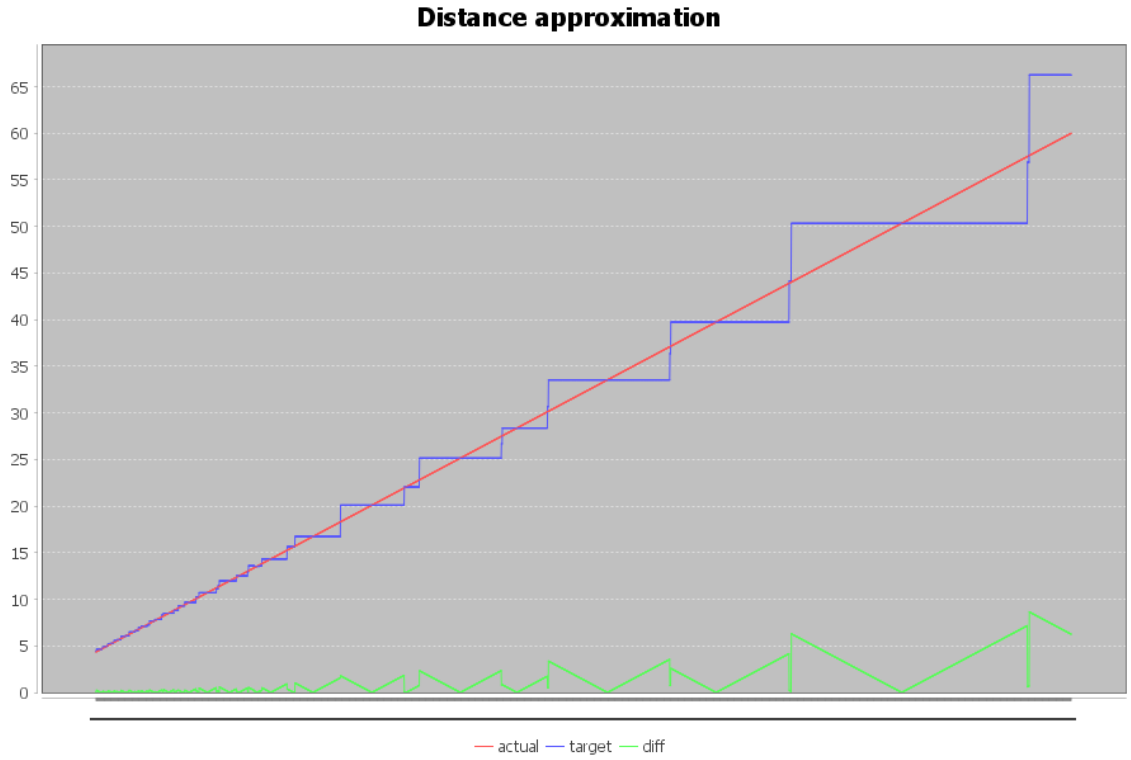


Figuur 10: Deze grafieken tonen het absolute verschil tussen de waarden uitgekomen door ons testbed en het provided testbed. De rode lijnen komen overeen met de waarden uit figuur 8 met tijdsstap $0.01s$, en de blauwe komen van de waarden uit figuur 9 met een tijdsstap van $0.001s$ voor ons testbed.

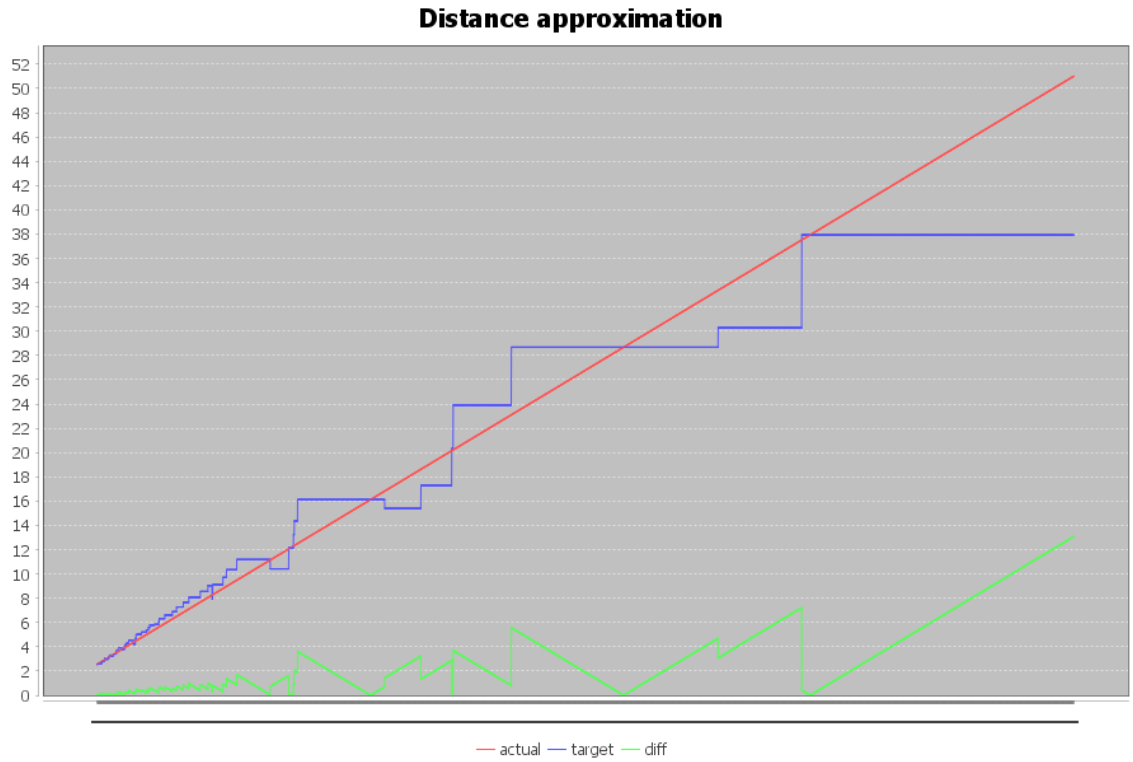
Figuur 11: Test 1:geen pitch, geen heading, veranderende z-coördinaat.



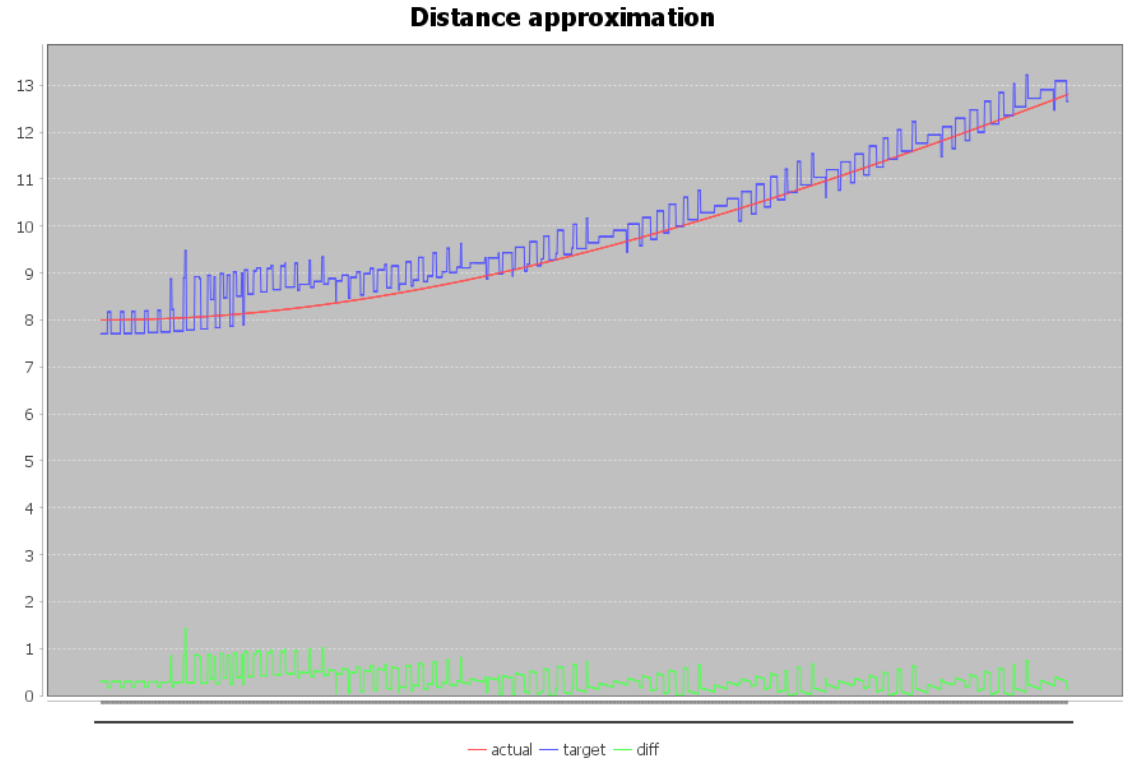
Figuur 12: Test 2: geen pitch, geen heading, veranderende x- en z-coördinaten.



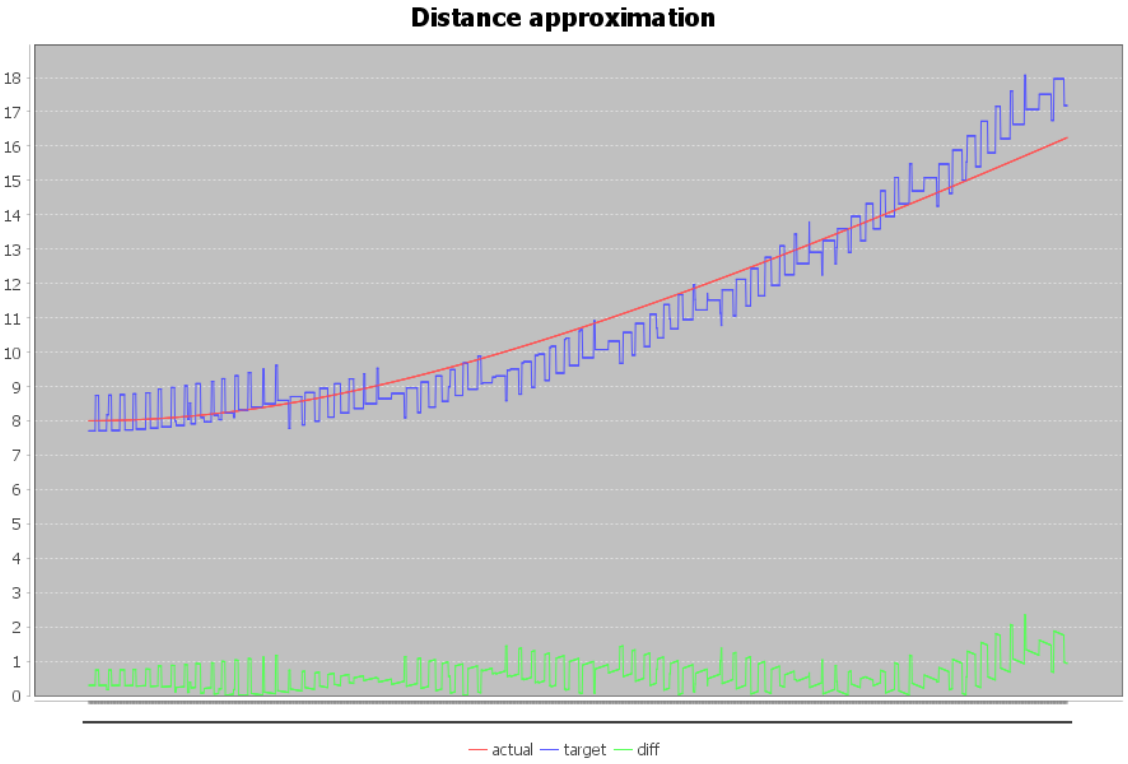
Figuur 13: Test 3: geen pitch, geen heading, veranderende x- y- en z-coördinaten.



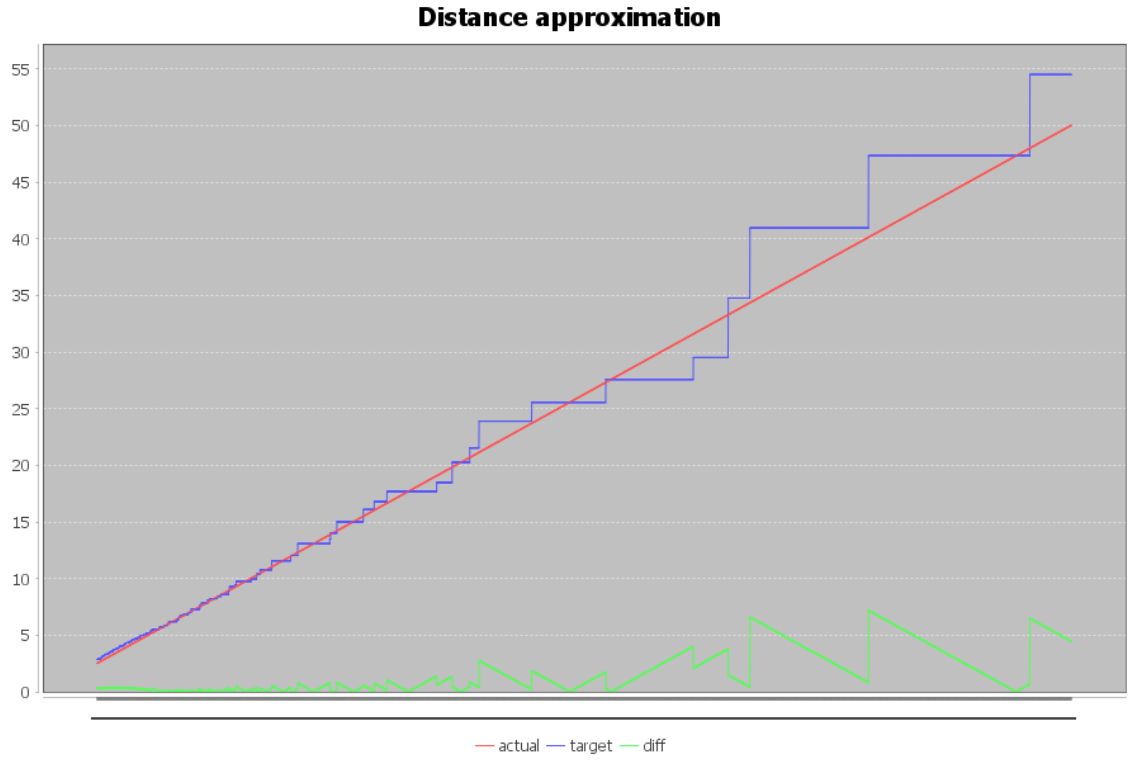
Figuur 14: Test 4: geen pitch, geen heading, veranderende x- en z-coördinaten, verandering van z is groter dan die van x.



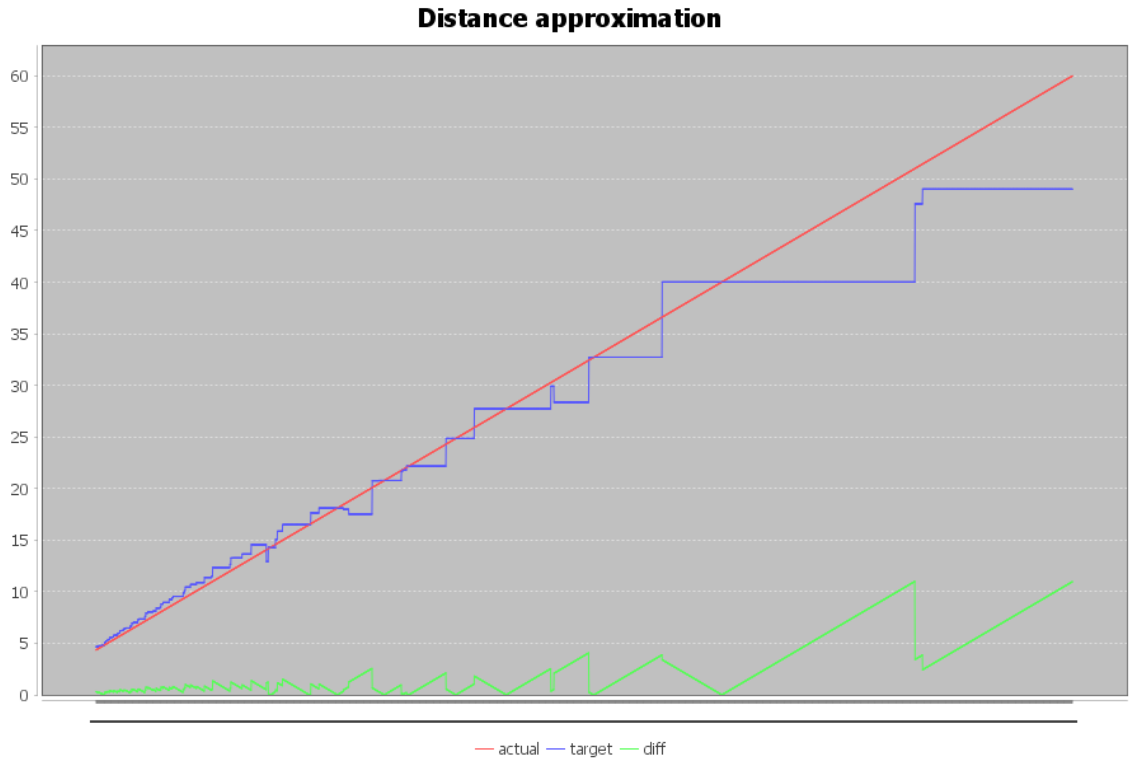
Figuur 15: Test 5: geen pitch, geen heading, vaste z-, veranderende x-coördinaat.



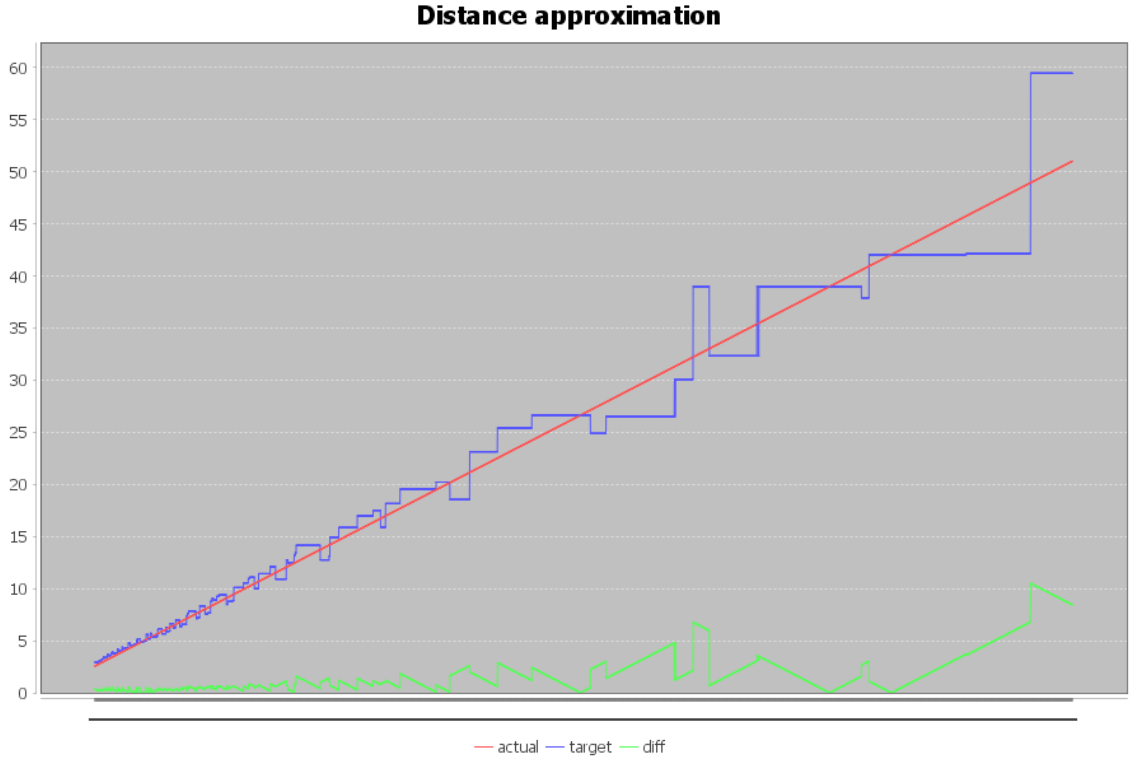
Figuur 16: Test 6: geen pitch, geen heading, vaste z-, veranderende x- en y-coördinaten.



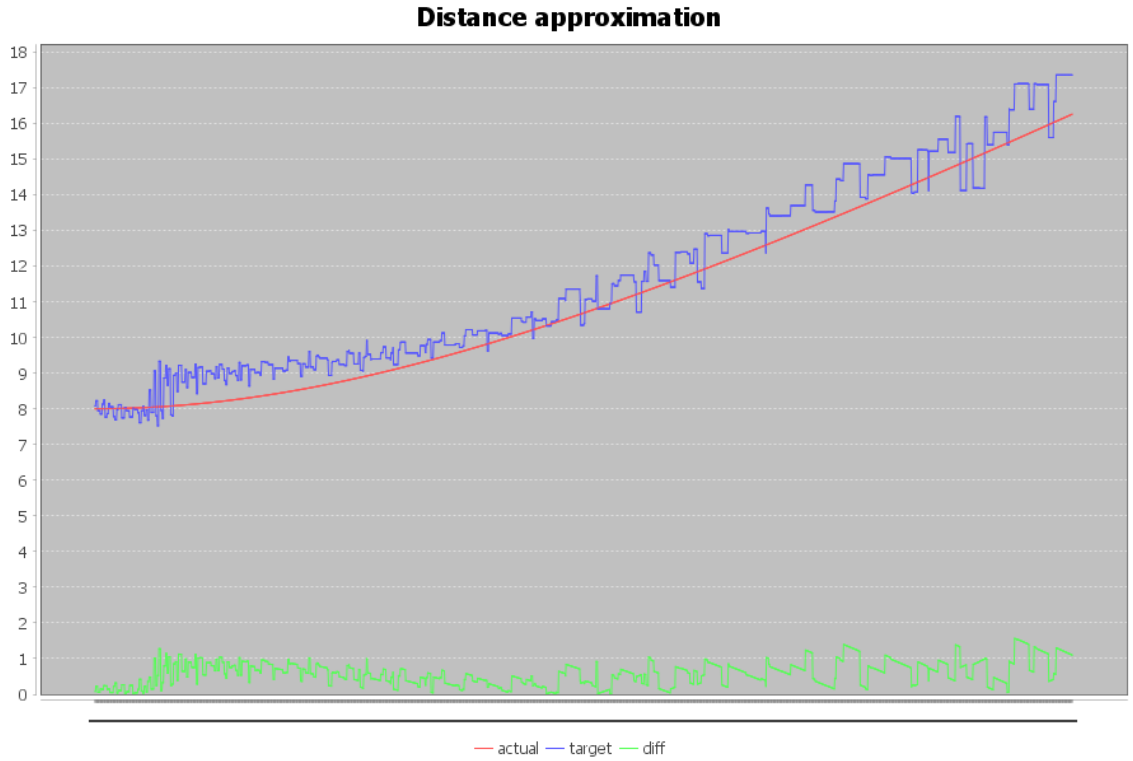
Figuur 17: Test 7: pitch van 45°, geen heading, veranderende z-coördinaat.



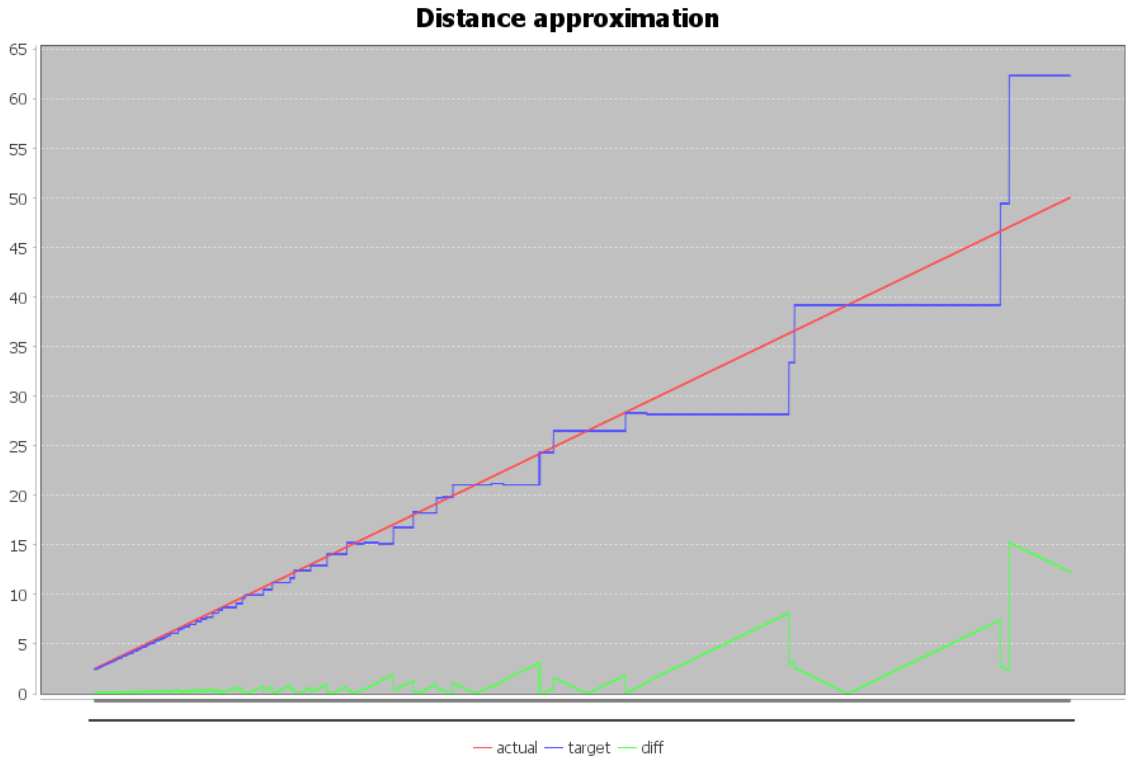
Figuur 18: Test 8: pitch van 45°, geen heading, veranderende x- y- en z-coördinaten.



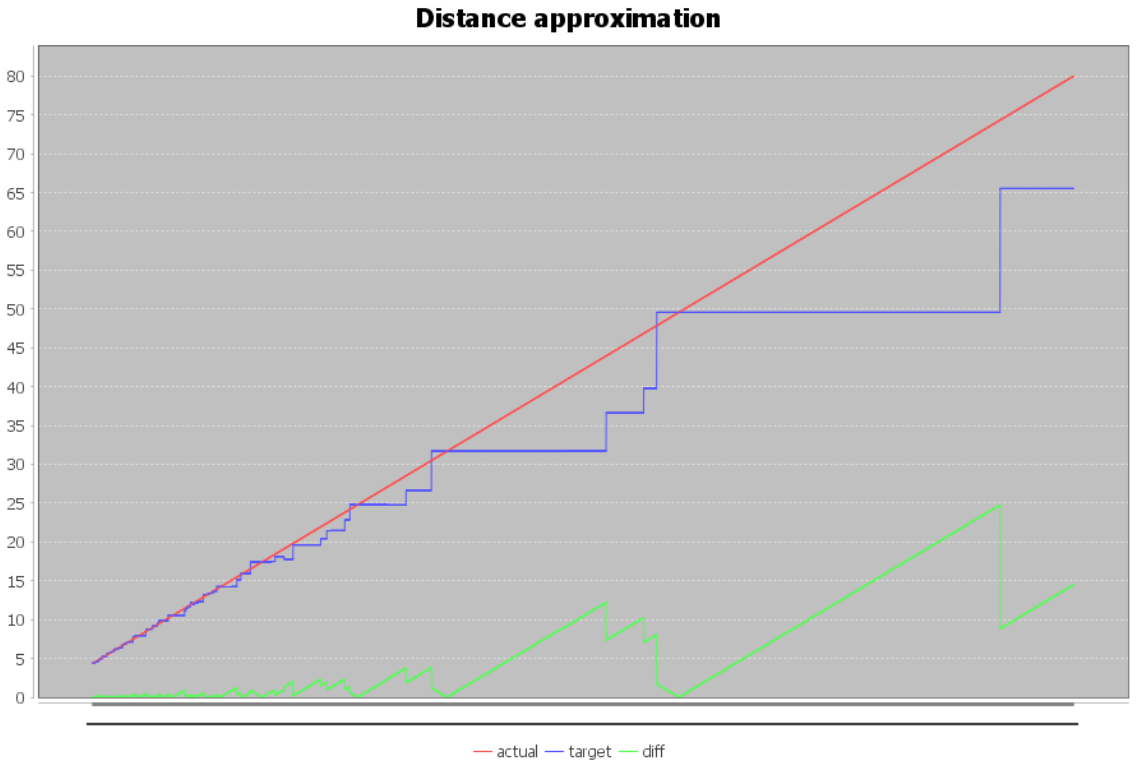
Figuur 19: Test 9: pitch van 45°, geen heading, veranderende x- en z-coördinaten, verandering van z is groter dan die van x. *Toon, Tomas*



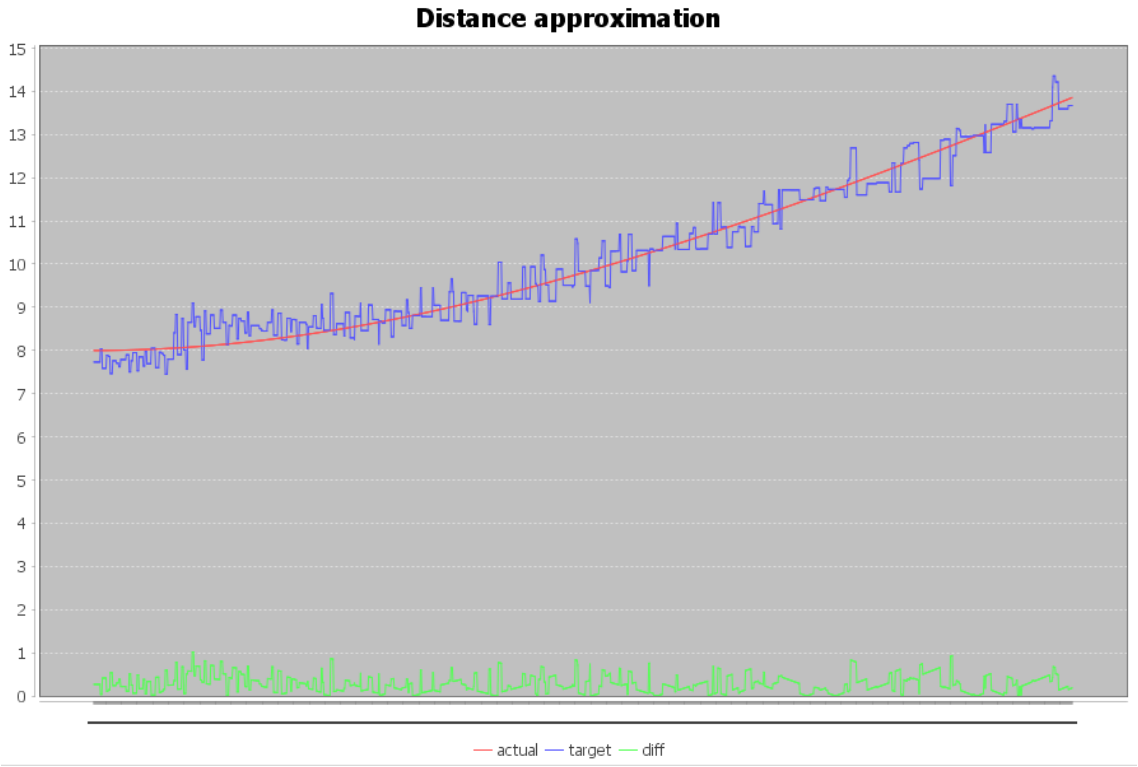
Figuur 20: Test 10: pitch van 45°, geen heading, vaste z-, veranderende x- en y-coördinaten.



Figuur 21: Test 11: pitch van 30°, heading van 30°, veranderende z-coördinaat.



Figuur 22: Test 12: pitch van 30°, heading van 30°, veranderende x- y- en z-coördinaten.



Figuur 23: Test 13: pitch van 30°, heading van 30°, vaste z-, veranderende x- en y-coördinaten.