Bachelorscriptie

Automatisch parkeren: Hoe werkt het?

Kevin Vriens

Lente 2009

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Introductie

Sinds de opkomst van de auto is er veel veranderd. Vroeger werden auto's gebouwd om betrouwbaar te zijn. Dat een botsing met lage snelheid al fatale gevolgen kon hebben nam men op de koop toe. Tegenwoordig is er veel meer aandacht voor de veiligheid van de automobilist(en de overige weggebruikers). Men kan hierbij denken aan gordels, kreukelzones en airbags.

De huidige trend gaat meer richting systemen die ongelukken trachten te voorkomen, zoals ABS, parkeersensoren en sinds een aantal jaren ook systemen om automatisch in te parkeren.

Toyota was de eerste fabrikant die een dergelijk systeem uitbracht in 2003 in Japan[1]. Dit zogenaamde 'Intelligent Parking Assist system' gebruikte camera's aan de achterkant om obstakels en openingen te identificeren en rekende vervolgens een reeks manoeuvres uit om de auto in te parkeren. De bestuurder diende echter nog wel zelf het gas- en rempedaal in te drukken.

1.2 Probleemstelling

In deze paper wil ik onderzoeken wat er nodig is om een veilig (en eenvoudig) automatisch inparkeer systeem te maken d.m.v. de realiteit te vertalen naar een (realistisch) computermodel. In deze scriptie zullen de termen realistisch of realiteit gebruikt worden als men het heeft over een goede weergave van de fysieke werkelijkheid.

1.3 Gerelateerd werk

Voordat ik begonnen ben de onderzoeksvraag te beantwoorden ben ik op zoek gegaan naar anderen, die problemen van dezelfde soort hebben trachten op te lossen. Een oplossing voor parallel parkeren wordt getoond in

het door Paromtchik IE, Garnier P en Laugier C gepubliceerde *Autonomous maneuvers of a nonholonomic vehicle*[2]. Hier breiden ze het probleem van veranderen van rijbaan uit naar een parallel parkeer probleem door extra restricties te leggen op de positie, snelheid e.d. waarmee de manoeuvre uitgevoerd moet worden. Hoewel dit een erg interessante aanpak is overlapte het niet genoeg met de ideëen die ik wilde onderzoeken om op een model uit te komen. In een later tijdstip werd ik ook verwezen naar het artikel van Daro Maravall en Javier de Lope. Deze publiceerde een oplossing voor dit probleem in *Multi-objective dynamic optimization with genetic algorithms for automatic parking*[3]. Deze aanpak is geschikter en enkele ideëen gepresenteerd hierin over inparkeren zijn later ook gebruikt in het model.

De meeste papers relaterend aan dit probleem zijn wiskundige oplossingen. De intentie van deze scriptie is om naast de wiskundige kant, ook te onderzoeken hoe een programma, dat de berekeningen voor het automatisch parkeer systeem uitvoert, in elkaar steekt of zou kunnen steken.

In de artikelen worden een aantal minimale eisen besproken, enkele waar men met logisch nadenken al op kan komen, waaraan voldaan moet worden die ook in dit verslag terug te vinden zullen zijn.

1.4 Afbakening

Omdat dit probleem uit meerdere delen bestaat zal het worden opgesplitst in een aantal deelvragen.

- Wat moet er aanwezig zijn in het computermodel om de realiteit (voldoende) na te bootsen?
- Hoe zorgt men dat de auto niet kan botsen?
- Wat is er nodig om de auto automatisch in te laten parkeren?
- Hoe zijn de gegevens gebruikt in (en gehaald uit) het model terug te voeren naar de realiteit?

Tijdens het algehele proces is steeds gedacht aan de terugkoppeling met de realiteit. Verscheidene dingen uit de realiteit die niet noodzakelijk waren voor het slagen van het onderzoek/model zijn versimplificeert in het model.

1.5 Methodes

Er is voor gekozen om het model te maken in C# met behulp van XNA Game Studio 3.0, dit is een IDE voor het ontwikkelen van 2D en 3D spellen waarin gebruik wordt gemaakt van het XNA framework.

Hoofdstuk 2

Onderzoek & Resultaten

2.1 Inleiding

In dit hoofdstuk zal worden beschreven hoe tot het uiteindelijke model gekomen is. De resultaten van het onderzoek zijn beschreven in de hierna volgende subhoofdstukken. Als eerste zullen telkens de belangrijkste onderdelen in een rij onder elkaar worden uitgezet waarna één voor één wordt besproken hoe deze zijn ingevuld in dit model.

Om termen die iets weergeven in de realiteit of het model en hun interne representatie in het model uit elkaar te houden, zullen in deze scriptie voor de interne representatie Engelse termen gebruikt worden. Bijvoorbeeld: positie is een plaats en *position* is de waarde van de vector die deze plaatst representeert.

2.1.1 XNA

Omdat het model is geschreven met behulp van het XNA framework is het nuttig om wat meer over XNA te vertellen. XNA heeft een aantal standaard functies die worden aangeroepen aan het begin van of tijdens een programma. Voor een programma begint wordt de functie *Initialize* aangeroepen. Hier kan naar vereiste services worden gevraagd en kunnen niet-grafische onderdelen geladen worden. In dit model worden hier alle *WorldObject*s aangemaakt.

Vervolgens is er *LoadContent*. Deze functie wordt een keer aangeroepen aan het begin en hierin wordt alle content geladen zoals textures, models en benodigdheden om tekst op het scherm weer te kunnen geven. *UnloadContent* doet het tegenovergestelde.

Update is een functie die om een bepaald tijdsinterval wordt herhaald. Dit is de plek om je spelcode uit te voeren als het updaten van de wereld, Collision Detection, input verzamelen en dergelijke.

Draw is nog een functie die om een bepaald tijdsinterval wordt herhaald. Dit is de plek om de code te zetten die zorgt voor het tekenen van de omgeving en alles erin.

Deze structuur zorgt ervoor dat het voor bepaalde acties in het model niet gewenst is deze in while of for loops te plaatsen. Worden bijvoorbeeld meerdere verplaatsingen van een object uitgerekend en uitgevoerd voordat *Draw* weer wordt aangeroepen dan zullen deze niet zichtbaar zijn en zal het lijken alsof het object door de omgeving 'springt' in plaats van geleidelijk aan beweegt.

2.2 Beperkingen

Er wordt in dit onderzoek geabstraheerd van het manoeuvreren om obstakels heen. Er wordt vanuit gegaan dat de bestuurder de auto zo parkeert dat deze door middel van achteruit rijden en een enkele bocht maken ingeparkeerd kan worden. Is dit niet het geval dan zal een waarschuwing worden gegeven. Initiëel was het wel de bedoeling om ook obstakels te kunnen ontwijken en passeren, maar dit bleek na enig literatuur onderzoek van zodanige complexiteit te zijn dat het niet meer in dit onderzoek bij zou passen.
Ook ondersteunt dit model geen parallel inparkeren.

2.3 Wat moet er aanwezig zijn in het computermodel om de realiteit (voldoende) na te bootsen?

Er zijn een aantal duidelijk onderscheidbare onderdelen die het model op zijn minst moet hebben. Deze zijn:

- Een omgeving waarin berekeningen uitgevoerd kunnen worden
- Objecten die zich in deze omgeving bevinden
- Functies die objecten op elkaar en/of de omgeving kunnen uitoefenen
- Overige

Al deze onderdelen zijn weer verder op te splitsen in onderdelen en functies.

2.3.1 Omgeving

De omgeving bestaat uit:

- Coördinatensysteem
- Kijkrichtingen, die naar een bepaald deel van het coördinatensysteem kijken
- Manier om te wisselen tussen kijkrichtingen

2.3.1.1 Coördinatensysteem

Het coördinatensysteem ligt aan de basis van de omgeving en geeft de mogelijkheid om berekeningen uit de voeren en objecten op bepaalde posities in de omgeving te zetten. In dit specifieke geval wordt gebruik gemaakt van een 3Dimensionaal assenstelsel. Als we praten over vectoren bedoelen we dus een (x,y,z) coördinaat. Als men met een draa(̈i)ng van 0 radialen(rad) naar de (0,0,0) vector van het coördinatensysteem kijkt ziet deze er als volgt uit. De positieve x-as wijst naar links, de positieve y-as naar boven en de positieve z-as loopt in het scherm(dichtbij naar ver weg). Een positieve draai om de Y-as gaat tegen de klok in. Deze omgeving wordt *world* genoemd. Alle *Model*s die geplaatst moeten worden in het assenstelsel ondergaan een 'world transform' ofwel worden met behulp van de *world*-matrix in de omgeving gezet. Hierdoor worden vertices van een model, die oorspronkelijk relatief zijn tot een lokaal punt in het *Model* hergedefiniëeerd zodat ze relatief worden aan hetzelfde punt in een omgeving als alle andere objecten.

2.3.1.2 Kijkrichtingen

Het computermodel heeft een aantal kijkrichtingen nodig waarmee het model bekeken kan worden. Een kijkrichting bestaat in het model uit een positie van waaruit gekeken wordt, een positie waarnaar gekeken wordt en een *FIeldOFView*, die bepaalt hoe breed, tot welke diepte en vanaf welke diepte er gezien kan worden. Kijkrichtingen worden opgesplitst in twee groepen. De eerste groep zijn kijkrichtingen die nabootsen wat de bestuurder in werkelijkheid kan zien. De tweede groep zijn kijkrichtingen die een zicht op het model weergeven. Deze zijn geïmplementeerd om te testen, om duidelijk weer te geven wat er nu eigenlijk gebeurd of om simpelweg het bewegen door het model makkelijker te maken.

Werkelijkheid

Een daarvan is de 'FirstPerson'-kijkrichting, het zicht van de bestuurder van de auto wanneer deze vooruit kijkt. Om dit te bereiken heeft het model een *FirstPerson-view* waardoor men vanuit de auto recht vooruit kijkt. Elke keer als de auto van positie of richting verandert in het model wordt de kijkrichting mee veranderd.

Het berekenen welk deel van de omgeving bekeken wordt voor deze kijkrichting gaat als volgt: Als eerste wordt de rotatie bepaalde van het object waaruit men wil kijken, in dit geval de auto, en een rotatiematrix gemaakt.

Als tweede wordt bepaald vanaf welke positie gekeken moet worden. Dit wordt gedaan door de positie van het object van waaruit gekeken moet worden te berekenen, al dan niet met een transformatie,die de rotatiematrix gebruikt, zodat er niet van binnenuit tegen het object aangekeken wordt.

Als derde wordt er een vector gemaakt die dezelfde richting opwijst als de kijkrichting en hieruit wordt een positie berekend waar naartoe gekeken wordt. Uit de positie waar vanuit gekeken wordt en de positie waar naartoe gekeken wordt, samen met de up-vector(0.0,1.0,0.0) wordt een view matrix gemaakt. Deze verplaatst als het ware objecten in de world matrix om de positie en oriëntatie van waaruit gekeken wordt heen.

Als laatste wordt er een projectie matrix gemaakt met behulp van de hoek die men ziet, de aspectratio, hoe diep men ziet en vanaf hoe diep met ziet. Met kijkt dus vanuit een punt naar een breder vlak. De projection matrix zorgt ook voor het perspectief, objecten die dichtbij zijn lijken groter dan objecten die ver weg zijn, omdat het dichtstbijzijnde deel van het frustum kleiner is dan delen verder weg. Hierdoor nemen objecten dus een groter deel van het beeld in beslag waardoor ze groter lijken.

De overige kijkrichtingen gaan op een bijna identieke manier.

Een zeer noodzakelijk view is de *Rear-view*, welke het zicht van de achteruitkijkcamera's laat zien. In de realiteit zal dit het beeld zijn dat men in de auto op een scherm te zien krijgt wanneer men achteruit wil inparkeren. Ook deze wordt mee veranderd als de auto van positie verandert.

model

In de realiteit heeft de bestuurder naast recht vooruit kijken natuurlijk ook de mogelijkheid om zich heen te kijken, zonder dat de auto meebeweegt. Omdat dit echter geen noodzakelijk aspect is van het automatisch inparkeren is ervoor gekozen om in plaats van een uitgebreidere *FirstPerson-view* een *ThirdPerson-view* toe te voegen die het geheel van bovenaf de auto laat zien. Deze verandert mee als de auto in het model van positie verandert.

Vervolgens is er ook een *Free-view*. Deze kan vrijuit door de gemodelleerde omgeving bewegen. Als de auto in het model van positie verandert, verandert deze niet mee.

2.3.1.3 Wisselen tussen kijkrichtingen

Elke kijkrichting wordt weergegeven door een *state* in het model. Als men *state* verandert, verandert nadat *Update* weer wordt aangeroepen, ook de kijkrichting. Om te wisselen tussen richtingen hoeft men slechts met een knop de state te veranderen.

2.3.2 Objecten

De objecten die minstens in een omgeving moeten zijn:

- Een object dat de auto representeert
- Objecten die obstakels representeren

Alle objecten aanwezig in de omgeving zijn van het type *WorldObject*.

2.3.2.1 Auto

De auto is het belangrijkste object van het model, dit is immers het object dat aangestuurd kan worden door de gebruiker van het model. Het object dat de auto representeert wordt vanaf nu ook wel *avatar* genoemd. De auto wordt visueel gerepresenteerd door een rechthoekig *Model*, wat gemaakt is in Blender, dat blauw met gele *Textures* heeft. *Models* hebben de eigenschap dat er een rechthoekige *BoundingBox* aan toegekend kan worden, waardoor gecombineerd met een functie van XNA, snel en makkelijk geconstateerd zou moeten kunnen worden of een vector in of op deze *BoundingBox* zit. Hierover later meer in de sectie Collision Detection.

De *avatar* heeft een aantal eigenschappen ten opzichte van zijn omgeving, waaronder zijn positie (*position*) en zijn hoek ten opzichte van de Y-as(*Yaw*).

Hiernaast heeft *avatar* een lengte van 4 meter en een breedte van 2 meter. Hoewel deze breedte niet erg reel is ten opzichte van deze lengte maakt het voor het model niet veel uit, de functies werken op alle afmetingen. Er is voor een breedte van 2 meter gekozen in dit specifieke voorbeeld omdat het makkelijk was bij het ontwikkelen/testen van de broncode.

Ook heeft *avatar* variabelen die zijn voor- en achterwaartse snelheid bijhouden als ook een variabele voor de draaisnelheid. Hierover meer in Voortbewegen.

Verder heeft *avatar* een *pivotPoint* en een functie om deze te berekenen. Deze informatie is nodig om bochten te maken, hierover later meer in Bochten.

De overige variabelen en functies zullen of later aan bod komen, of zijn niet belangrijk genoeg om te bespreken. De hele code is te zien in Appendix B: Code.

De manier waarop de auto in de wereld wordt gezet is als volgt: Ten eerste wordt een world matrix gecreëerd met behulp van zijn rotatie en positie.

```
Matrix World = Matrix.CreateRotationY((float)avatar.getYaw())
   * Matrix.CreateTranslation(avatar.getPosition());
```

Vervolgens wordt het object getekend met behulp van de informatie opgeslagen in *Model*.

```
void DrawModel(Matrix world, WorldObject c1)
{
    foreach (ModelMesh mesh in c1.getModel().Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.Projection = proj;
            effect.View = view;
            effect.World = world;
            effect.Texture = c1.getTexture2D();
            effect.TextureEnabled = true;
        }
        mesh.Draw();
    }
}
```

Hierbij zijn proj, view en world de matrixen die horen bij de huidige kijkrichting.

2.3.2.2 Obstakels

Alle obstakels zijn ook van het type *WorldObject* en delen dezelfde eigenschappen(niet dezelfde waarden voor deze eigenschappen) als *avatar*. Er is voor gekozen om de obstakels in het huidige model te beperken tot andere auto's van dezelfde lengte en breedte als *avatar*. Dit is echter eenvoudig uitbreidbaar door andere modellen toe te voegen. Alle functies zijn dynamisch zodat ze werken voor rechthoeken met verschillende lengtes en breedtes. Het is nog niet mogelijk om ronde objecten toe te voegen, hoewel dit geen grote uitbreiding is, doordat het voor ronde objecten makkelijk is om uit te rekenen of ze botsen met anderen als je de straal weet. De obstakels hebben in het model blauw met rode textures.

In de huidige implementatie van het model staan alle obstakels stil.

2.3.3 Functies

De functies die de objecten minstens op elkaar en de omgeving moeten kunnen uitvoeren zijn:

- Een manier om voort te bewegen
- Een manier om bochten te simuleren
- Een manier om botsingen te detecteren(Collision Detection) en voorkomen
- Een manier om automatisch in te parkeren

2.3.3.1 Voortbewegen

Om voort te bewegen zijn een aantal dingen nodig. Ten eerste moet men de snelheid(*velocity*) weten en de richting waarin de auto wijst(*Yaw*). *velocity* is de afstand die wordt afgelegd tussen twee tijdseenheden in het model.

Om voort te bewegen wordt eerst een rotatie matrix *forwardMovement* uitgerekend voor de draaiing om de y-as met behulp van de *Yaw*. Vervolgens wordt een vector v gemaakt met de waarde (0,0,velocity). De *velocity* is de z-waarde van de vector omdat dit vooruit(verder weg) is in het coordinatensysteem. Hierna wordt vector v getransformeerd met behulp van matrix *forwardMovement*. Om vooruit te bewegen wordt *position* opgehoogd met de getransformeerde vector v.

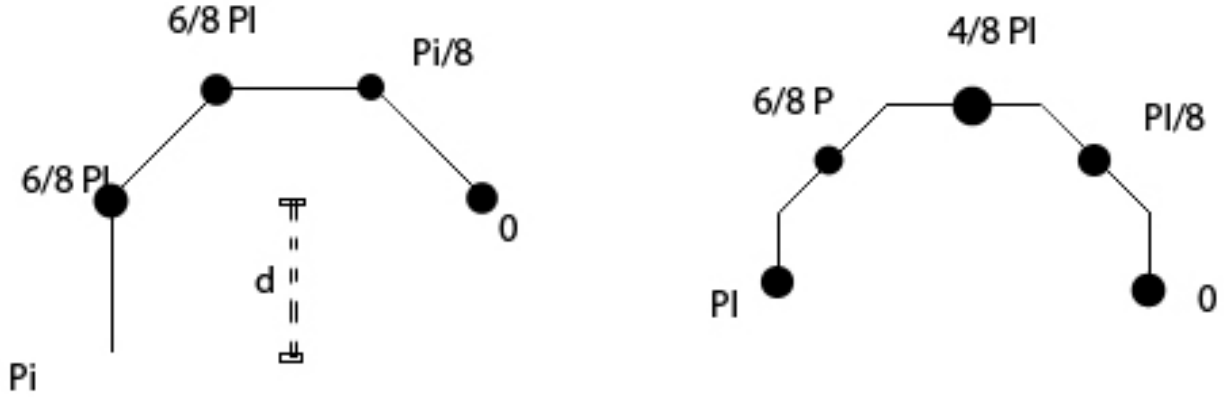
Dit kan echter niet altijd. Als het voort te bewegen object tegen een ander object aan staat of zal bewegen mag deze verplaatsing natuurlijk niet uitgevoerd worden. Om dit te voorkomen wordt gebruikt gemaakt van een, in een model niet zichtbaar, *WorldObject* genaamd *avatarFuture*. Dit is een *WO* met dezelfde *position*, *Yaw*, *velocity* en *rotationSpeed* als *avatar*.

In plaats van de *position* van *avatar* op te hogen met de getransformeerde vector v wordt de *position* van *avatarFuture* opgehoogd. Vervolgens wordt Collision Detection toegepast op alle objecten in de buurt. Als *avatarFuture* botst met een ander object wordt de getransformeerde vector v weer van de position van *avatarFuture* afgetrokken en zal er geen verplaatsing van *avatar* plaatsvinden. Vindt er geen botsing plaats dan wordt de *position* van *avatar* hetzelfde als de *position* van *avatarFuture*.

Achteruit bewegen gaat op dezelfde manier alleen wordt dan de vector v berekend door de negatieve waarde van *velocity* te nemen.

2.3.3.2 Bochten

In de realiteit wordt een bocht gemaakt door de wielen te draaien en gas te geven. Op deze manier wordt een deel van een cirkel afgelegd totdat de wielen weer terug worden gedraaid. In het model werkt het echter anders. Een cirkel wordt gemaakt door *Yaw* op te hogen of te verlagen met de draaisnelheid en voort te bewegen. Echter, als men *Yaw* maar een keer verandert en voortbeweegt zal *avatar* een rechte lijn maken in de richting waarin *Yaw* 'wijst' in plaats van een bocht. Het is dus noodzakelijk om *Yaw* bij elke stap van de bocht met dezelfde waarde(*rotationSpeed*) te veranderen wil men een bocht maken. Merk dus op dat een bocht afhangt van zowel *velocity* als *rotationSpeed*. Een bocht maken gaat dan alst volgt: Eerst verplaatst men zich de halve afstand in de huidige richting, daarna verandert men *Yaw* met *rotationSpeed* en vervolgens verplaatst men zich weer een halve afstand in de nieuwe richting. Dit wordt zo gedaan omdat het uitrekenen van de draaicirkel op deze manier logischer is. Zoals men kan zien aan de



Figuur 2.1: Bocht

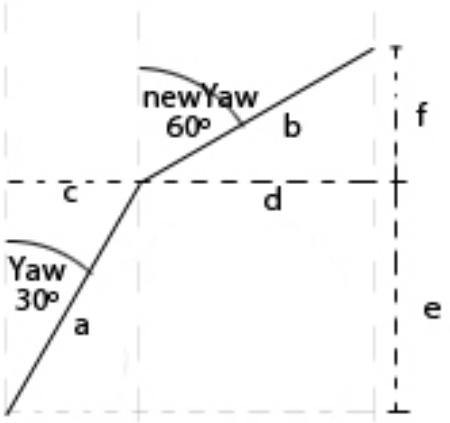
linkerkant, als men een *Yaw* van 0 heeft en men draait voordat men een afstand verplaatst, zal als de *Yaw* PI is geworden er een hoogteverschil

van d is ontstaan, terwijl we bij een halve bocht op dezelfde hoogte uit willen komen. Dit is ongewenst als we later gaan inparkeren omdat we deze afstanden nodig hebben. Aan de rechterkant wordt eerst een halve afstand in de huidige richting verplaatst alvorens te draaien en nog een halve afstand te verplaatsen, hierdoor ontstaat geen hoogteverschil.

Een bocht naar links houdt in dat de *Yaw* met de *rotationSpeed* verhoogt wordt. Om het model wat realistischer aan te laten voelen zijn een bocht links vooruit en een bocht links achteruit beide met dezelfde knop voor links te maken, zoals ook in een auto beide bochten te maken zijn door het stuur naar links te draaien. Dit is echter niet vanzelfsprekend. Als men in de bovenbeschreven implementatie achteruit beweegt en men drukt de knop in voor linksaf zal *Yaw* in het oude model opgehoogd worden, wat zal leiden tot een bocht rechts achteruit. Om deze reden wordt *Yaw* bij achteruitrijden verhoogt als de auto naar rechts gaat en verlaagt als deze naar links gaat.

Om de realiteit verder trouw te blijven dient er een limiet te zijn aan de kleinste draaicirkel die een auto kan maken. Bij een auto van 4 meter hoort een draaicirkel van ongeveer 5 meter [4].

Het uitrekenen diameter van een cirkel met behulp van *rotationSpeed* en *velocity* gaat als volgt: Het verschil tussen een keer veranderen van een



Figuur 2.2: Diameter

hoek, in dit geval tussen $Yaw = 1/6$ Pi rad en $newYaw = 1/3$ Pi rad met een bepaalde *velocity*, levert het volgende op:
Voor a en b geldt:

$$a = b = \frac{velocity}{2} \tag{2.1}$$

Voor x geldt:

$$x = c + d \tag{2.2}$$

$$c = sin(Yaw) \times a = sin(Yaw) \times \frac{velocity}{2} \tag{2.3}$$

$$d = sin(newYaw) \times b = sin(newYaw) \times \frac{velocity}{2} \tag{2.4}$$

Dus x wordt:

$$x = sin(Yaw) \times \frac{velocity}{2} + sin(newYaw) \times \frac{velocity}{2} \tag{2.5}$$

Voor z geldt:

$$z = e + f \tag{2.6}$$

$$e = cos(Yaw) \times a = cos(Yaw) \times \frac{velocity}{2} \tag{2.7}$$

$$f = cos(newYaw) \times b = cos(newYaw) \times \frac{velocity}{2} \tag{2.8}$$

Dus z wordt:

$$z = cos(Yaw) \times \frac{velocity}{2} + cos(newYaw) \times \frac{velocity}{2} \tag{2.9}$$

Om de diameter te berekenen begint men met een *Yaw* van 0 en doet men dit net zolang tot *Yaw* Pi rad is.

Er is hier gekozen voor een *rotationSpeed* van Pi/80 radialen en een *velocity* van 1/10,185. Ook hier geldt weer alle functies gelden voor welke draaicirkel men ook maar wil gebruiken.

Deze kleinste draaicirkel wordt gerealiseerd door een limiet te zetten op de verhouding tussen de hoek waarmee *Yaw* kan worden verandert en *velocity*.

$$Velocity/rotationSpeed >= 2,5002 \tag{2.10}$$

Wordt *Yaw* immers met een grotere *rotationSpeed* verandert terwijl *velocity* hetzelfde blijft, dan wordt de cirkel kleiner en de bocht dus scherper. Wordt *velocity* groter terwijl *rotationSpeed* hetzelfde blijft dan wordt ook de cirkel groter en dus de bocht minder scherp.

Waar ook rekening mee gehouden moet worden is de as waar de auto omheen draait. In plaats van in het midden ligt de as waar omheen gedraaid wordt (*pivotPoint*) van een auto tussen de achterwielen in. Dit probleem wordt opgelost door *Yaw* te veranderen en, in plaats van nu meteen de auto voort te bewegen, eerst de positie van *avatar* te 'verschuiven' zodat zijn achteras op dezelfde plaats ligt als waar deze was voordat *avatar* draaide. Op deze manier draait de auto als het ware om zijn achteras.

Als laatste is het natuurlijk niet de bedoeling dat de auto op zijn plaats rond kan draaien. Om dit te voorkomen is er een restrictie geplaatst op het draaien. *Yaw* kan niet twee keer met dezelfde *rotationSpeed* veranderen zonder dat er tussendoor minstens een keer voor- of achteruit is gereden.

2.3.3.3 Collision Detection

Collision Detection bleek een van de wat lastigere problemen van het onderzoek. Er bleek na testen dat de standaard *BoundingBox*-class zelf niet voldoende geschikt was, omdat deze als het object draait, niet zelf meedraait. De *BoundingBox* blijft een rechthoekige kubus loodrecht op de x-as, gecreëerd uit de *MAX* en *MIN* vectoren(dit zijn de punten met de grootste, respectievelijk kleinste x, y en z waarden) en de bijgeleverde functies zijn dus maar deels geschikt om te bepalen of twee objecten botsen. Om dit op te lossen is er een functie geschreven genaamd *ComputeOwnBoundingBox*. Deze berekent 11 punten van de auto, die in de rij *OBBBasic*(OwnBoundingBoxBasic) worden gezet. Deze 11 punten bestaan uit de 8 hoeken, het midden van de auto en het midden van de voor- en achterkant van de auto, zoals deze zouden zijn als de auto *position*=(0,0,0) had en *Yaw* =0. In eerste instantie was gekozen om alleen de hoeken op te slaan. Dit is genoeg als men wil bepalen of *WorldObject*s tegen elkaar gaan botsen. Er is echter gekozen voor 3 extra punten om ook bij *WorldObject*s die om een of andere reden overlap mogen hebben, te kunnen herkennen of ze overlappen. Dit wordt gebruikt bij de *avatarParkingBox*, waar later meer over verteld zal worden. Deze kan door objecten heen bewegen en het kan dus voorkomen dat op een gegeven moment geen enkele hoek overlap heeft met een ander object, terwijl de objecten zelf wel degelijk overlappen. Door de 3 extra punten zo te kiezen als gedaan is zal ook in deze instanties herkend worden of de objecten overlap vertonen. De *BoundingBox*-class heeft een functie genaamd *Contains*, welke kan bepalen of een vector in, of op de grens van, een *BoundingBox* zit. Om deze functie te kunnen gebruiken is voor de volgende twee methodes gekozen om collisions waar te nemen:

Methode 1:

Bereken het dichtstbijzijnde punt tot het eigen *WorldObject* en kijk of deze elkaar snijden/overlappen. Om het dichtsbijzijnde punt te berekenen moest eerst het dichtstbijzijnde andere *WorldObject* berekend worden. Dit is gedaan met behulp van de Euclidische afstandsfunctie, zoals te zien in Appendix B: Code.

Het bleek echter niet voldoende om alleen het dichtstbijzijnde *WorldObject* te testen op botsen. Ten eerste is het lastig om 'dichtstbijzijnde' te definiren in het model. Neemt men bijvoorbeeld het middelpunt, dan kunnen twee rechthoekige objecten waarvan de middelpunten het dichts bij elkaar liggen toch nog botsen met een derde object aan hun uiteinden. Ten tweede zou het voor kunnen komen dat twee of meerdere objecten evenver van elkaar afliggen. Omdat je in het model niet precies weet met welk ander object je botst moet voor alle *WorldObject*s (of alle objecten binnen een bepaalde

afstand) bekeken worden. Het is hierom handig om alle *WorldObject*s in een rij bij te houden en deze door te lopen op zoek naar botsingen.

Methode 2:

De bedoeling is om een object *WO* te transformeren naar zijn originele *BoundingBox*, en dezelfde transformatie toe te passen op een vector waarvan we willen weten of het zich bevind in *WO*, zodat deze op dezelfde positie blijft liggen ten opzichte van *WO*.

Er is het object van de eigen auto en een ander object, in dit model ook een rechthoekige kubus. Dit andere object noemen we voor het gemak even *currentWO*. Als eerste worden door *ComputeTransformedOwnBoundingBox* alle 11 punten van de eigen auto berekend met behulp van zijn rotatie en positie. Vervolgens wordt *BoundingBox* van *currentWO*, genaamd *CWOBB*, berekend met behulp van zijn positie maar zonder rotatie, wat in feite een transformatie oplevert ten opzichte van als deze wel met behulp van zijn rotatie was berekend. Om dezelfde transformatie toe te passen op de punten van de auto wordt voor elk punt *p* de *position* van *currentWO* afgetrokken, gedraaid met -*Yaw* van *currentWO* en als laatste de *position* van *currentWO* er weer bij opgeteld, zodat men op het nieuwe punt *currentP* komt, dat dezelfde transformatie heeft ondergaan als *CWOBB*. Zie ook Transformatie figuur.

Vervolgens wordt voor elk punt *currentP* de *Contains*-functie aangeroepen op *CWOBB* om te zien of een punt van de auto zich bevind in *currentWO*.

Dit is nog niet alles, omdat ook voor alle punten van *currentWO* moet worden gekeken of deze zich bevinden in het auto-object. Dit gaat echter op eenzelfde manier, alleen worden de functies die eerst met auto werden aangeroepen nu met *currentWO* aangeroepen en andersom.

Om botsingen ook daadwerkelijk te voorkomen, wordt voordat de auto zich verplaatst berekend of deze zal botsen met de betreffende verplaatsing. Als dat zo is wordt de desbetreffende verplaatsing niet uitgevoerd.

2.3.3.4 Automatisch parkeren - Achteruit

Automatisch inparkeren kan op vele manieren. Er is hier gekozen om het zo rechttoe rechtaan mogelijk te houden. Om in te parkeren is kennis nodig van de parkeerplaats, namelijk de grootte en de hoek waarin deze staat ten opzichte van de auto. Echter, men hoeft niet precies te weten hoe groot de parkeerplaats is, zolang men maar weet dat de auto hierin past. Om deze gegevens te achterhalen werken we in het model met een *avatarParkingBox* afgekort *APB*.

APB is een *WO* met een iets grotere *BoundingBox*(velocity/2 aan beide kanten) dan *avatar*. Waarom dit is gedaan zal later duidelijk worden. *APB* is ook manoeuvreerbaar, zodat deze in het model (en op eenzelfde manier op een scherm in het dashboard van de auto) op de plek kan worden geplaatst waar de gebruiker wil parkeren. *APB* is zo geprogrammeerd dat deze groen wordt als de parkeerplaats groot genoeg is en rood wordt als dat niet zo is.

Om *avatar* nu vanzelf in te laten parkeren laten we de *avatar* zover mogelijk in een rechte lijn bewegen totdat deze, door de scherpste bocht mogelijk te maken, op een lijn komt te staan met *APB* en dezelfde richtingsvector heeft, zodat *avatar* alleen nog maar in een rechte lijn hoeft te bewegen om op het gewenste punt met dezelfde richtingsvector te komen.

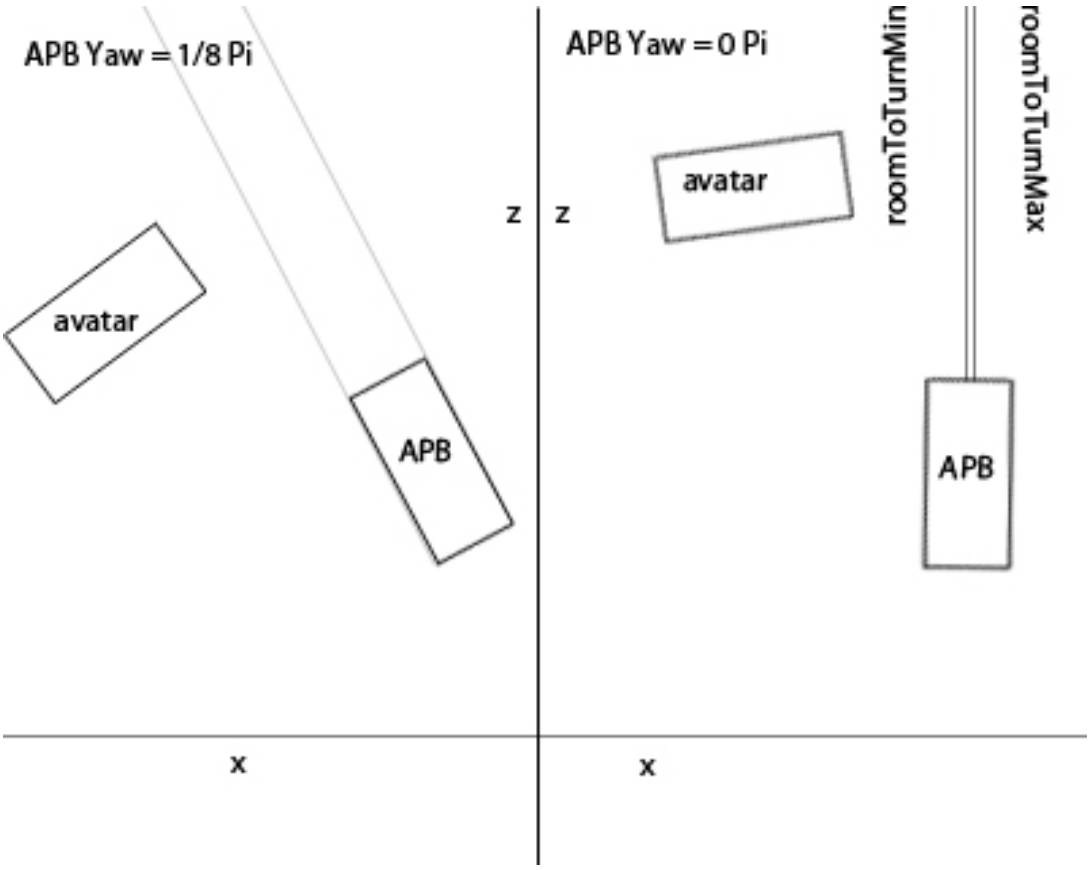
Om er voor te zorgen dat *avatar* niet tijdens het inparkeren erachter komt dat deze ergens tegen aan zal rijden en dus niet kan inparkeren, worden vantevoren alle manoeuvres berekend door middel van een niet fysieke *avatarTest*, die dezelfde waardes heeft voor alle eigenschappen van *avatar*. Pas als bepaald is dat er niet gebotst wordt, worden de manoeuvres door *avatar* uitgevoerd.

Dit alles is echter makkelijker dan het lijkt. Om ervoor te zorgen dat van alle mogelijke kant kan worden ingeparkeerd is het noodzakelijk om een algoritme te schrijven dat zoveel mogelijk abstraheert van de richting waar men vandaan komt/naartoe gaat. Ook is het belangrijk om in te zien dat als de *Yaw* van *avatar* en *APB* 1.5 Pi rad van elkaar verschillen,*avatar* soms maar 0.5 Pi hoeft te roteren en niet 1.5 Pi. Later in het verslag zal blijken waarom dit zo belangrijk is.

Ten eerste wordt er voor gezorgd dat de *Yaw* van *avatar*(*currentYaw*) en de *Yaw* van *APB*(*goalYaw*) allebei tussen de 0 en 2 Pi rad liggen. Dit zodat we niet met negatieve hoeken hoeven te gaan rekenen.

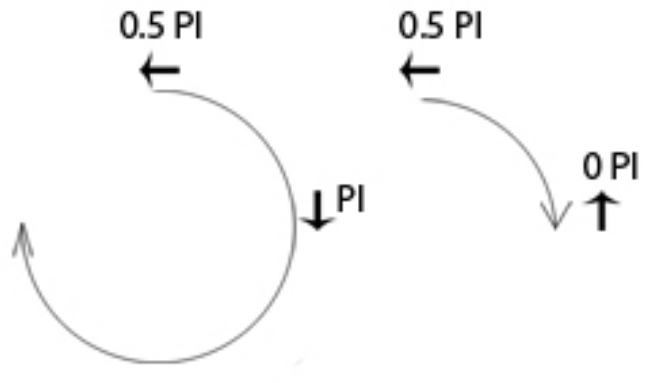
Wat hierna wordt bepaalt is of de auto een bocht naar links of rechts moet maken. Dit wordt gedaan door een transformatie te berekenen die toegepast moet worden op de avatar, zodat deze een *Yaw* van 0 krijgt en een nieuwe vector *APBTemp* te creeëren die ontstaan door diezelfde transformatie toe te passen op *APB*. Door nu te kijken of de x-waarde van de *position* van *APBTemp* groter of kleiner is dan die van *avatar* weet men welke richting men moet roteren om de juiste bocht te maken.

Vervolgens wordt berekend wat de minimale*roomToTurnMin*) en maximale(*roomToTurnMax*) x-waarde zijn waartussen de *avatar* mag uitkomen na rotatie, tenopzichte van de *APB* als deze een *Yaw* van 0 zou hebben. In de figuur is er een speling van velocity/2 links en rechts van het middelpunt van *APB*, tussen *roomToTurnMin* en *roomToTurnMax*, dus de afstand die *avatar* aflegt in een tijdseenheid. Dit komt omdat *avatar* niet altijd op de precieze x-waarde van *APB* uit kan komen. Er moet dus een speling zijn van de afstand die in een tijdseenheid wordt afgelegd oftewel *velocity*.



Figuur 2.3: Transformatie

Om nu te kijken op welke punt *avatar* moet beginnen met roteren om tussen de minimale en maximale x-waarde uit te komen moet worden berekend hoe groot de x-afstand en z-afstand is die wordt afgelegd in de bocht vanuit de *currentYaw* naar *goalYaw*. Op dit moment is het van belang om te zorgen dat *goalYaw* nooit meer dan Pi afligt van *currentYaw* in de richting waarin deze zal ga-an roteren. Is dit echter wel het geval dan zal *avatar* een grotere rotatie maken dan nodig is. Verschilt *Yaw* bijvoorbeeld 1.5 Pi rad dan legt *avatar* een rotatie van (en daarbij een x- en z-afstand die behoort aan) 1.5 Pi af terwijl deze maar een rotatie van 0.5 Pi hoeft af te leggen om evenwijdig te komen liggen met *APB*. Het is deze kleinste rotatie waarvan we de x- en z-afstand nodig hebben om te berekenen op welk punt *avatar* moet gaan roteren. Is het verschil precies Pi dan hoeft er of niet te worden geroteerd als *avatar* en *APB* al op een lijn liggen, of er moet een speciale bocht gemaakt worden bestaande uit een rotatie heen en eenzelfde rotatie terug.



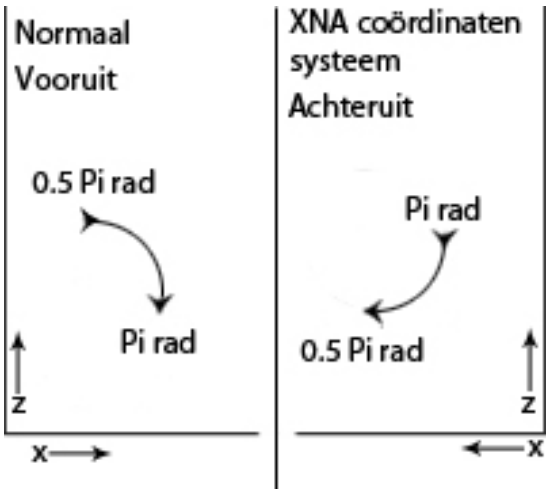
Figuur 2.4: Roteren

De x- en z-afstand wordt bepaald door *currentAngle* met *rotationSpeed* te verhogen als de *goalAngle* groter is, of te verlagen als *goalAngle* kleiner is, totdat *goalAngle* bereikt wordt. Men moet echter ook nog weten of de 0 rad gepasseerd wordt. Is dit immers het geval dan, bijvoorbeeld bij *currentAngle*=1/6 Pi en *goalAngle*=1 5/6Pi, dan is het duidelijk dat, ook al is *goalAngle* groter, deze eerder wordt bereikt door *rotationSpeed* van *currentAngle* af te trekken. In dit geval, dus van links naar rechts over de 0 rad, wordt er dan ook 2Pi van *goalAngle* afgetrokken. *goalAngle* wordt dan -1/6 Pi, wat dezelfde hoek oplevert die toch op de juiste manier eerder wordt bereikt. Wordt de 0 rad van rechts naar links gepasseerd dan wordt dit opgelost door 2Pi van *currentAngle* af te trekken.

De x- en y-afstand worden van *pivotPoint* van *avatar* afgetrokken(omdat achteruit wordt ingeparkeerd, zie figuur) en deze vector *tempGoal* wordt getransformeerd zodat deze op dezelfde positie ligt ten opzicht van *APB* als men deze zou roteren naar een *Yaw* van 0 rad.

De Normaal Vooruit-kant van de figuur laat zien hoe de gebruikte *computeTurningCircle* functie de x- en z- afstand berekent als *avatar* van 0.5 Pi rad naar Pi rad roteert. Men kan zien dat hier een positieve x- en een negatieve z-waarde uitkomt. Omdat er echter achteruit wordt gereden zoals aan de Achteruit-kant van het figuur, wisselen positieve en negatieve waarden van teken. In plaats van de x- en z- waarde bij *pivotPoint* op te tellen(zoals bij vooruit rijden), worden ze er dus vanaf getrokken.

Vervolgens wordt gekeken of *tempGoal* tussen *roomToTurnMin* en *roomToTurnMax* in ligt. Ligt *tempGoal* tussen deze het dichtstbijzijnde punt en *avatar* in, dan wordt er achteruit verplaatst totdat *tempGoal* ertussen ligt. Ligt *tempGoal* verder dan deze punten dan is *avatar* te ver verplaatst(wat als het goed is niet mogelijk is) of de auto staat op een positie ten opzichte van



Figuur 2.5: Achteruit

APB waar hij niet kan inparkeren. Ligt *tempGoal* wel tussen deze punten, dan zal *avatar* de scherpste bocht gaan maken die deze kan tot *currentYaw* gelijk is aan *goalYaw*. Op dit moment zitten ze op een lijn. Wanneer dit het geval is, zal *avatar* zich achteruit verplaatsen totdat deze binnen een afstand van *velocity* zit ten op zichte van *APB*.

Om vooruit in te parkeren zijn er slecht een klein aantal aanpassingen aan dit algoritme. Als eerste moet *goalYaw* PI/2 groter zijn, maar moet er wel getransformeerd worden met de huidig uitgerekende *goalYaw*. Als tweede moet zoals al vermeld de uitgerekende x- en z-waarde die de afstand van de scherpste bocht tot een bepaalde *goalYaw* voorstellen bij *pivotPoint* worden opgeteld om tot de juiste *tempGoal* te komen.

2.3.4 Overige

Omdat er bij het model geen handleiding zit is het handig om tekst weer te geven. Dit is overigens geen noodzaak voor het slagen van het model.

2.4 Realiteit versus Model

Er zijn een aantal punten waarop de realiteit verschilt met het model en er nog verbetering mogelijk is. Zo beweegt de auto in het model altijd met een constante snelheid. Er wordt in de berekeningen dus geen rekening gehouden met versnellen(als de inparkeermanoeuvre begint) en afremmen(als de auto in de buurt van andere auto's is en gaat botsen of eindigt met de inparkeermanoeuvre). Het probleem met versnellen is op te lossen door bij het wegrijden kleinere snelheden te gebruiken en op te bouwen tot de maximale

die men wenst bij het inparkeren, bij remmen zou men dan afbouwen naar kleinere snelheden. Om niet de botsen zou dan simpelweg vooruit gekeken moeten worden met een aftand die afhangt van de remweg bij de huidige variabele snelheid. Het automatisch inparkeren zou dan aangepast moeten worden door met draaicirkels te werken die bij de variabele snelheid horen in plaats van bij de constante snelheid waarmee het model nu werkt.

Om een computermodel als dit ook daadwerkelijk, als deel van een computersysteem, bij inparkeren te gebruiken zijn er een aantal kanttekeningen die gemaakt moeten worden:

- Gegevens verkrijgen
- Collision Detection
- Overload
- Manoeuvreren

2.4.1 Gegevens verkrijgen

Het is duidelijk dat de gegevens in het model niet allemaal vanzelfsprekend te verkrijgen zijn in de praktijk. De meeste gegevens moeten worden verzameld door de camera's en sensors die op de auto zijn aangebracht en via een onboard computer in verbinding staan met het model. Door een combinatie van camera's en sensors kan een 3D-beeld berekend worden, wat doorgegeven kan worden aan het model. Natuurlijk heeft deze vorm van gegevens verkrijgen een aantal nadelen.

Zo zal het aan de apparatuur liggen hoe nauwkeurig de gegevens zijn die verkregen worden. Hoe onnauwkeuriger gegevens worden verkregen en gemeten in de praktijk, hoe meer speling er moet zijn bij de berekeningen in het model.

Ook kunnen camera's en sensors niet door objecten heen kijken. Als een deel van de wereld dat nog niet door de auto 'gezien' is, is het ook moeilijker om vóór het starten van de inparkeerprocedure te kunnen zeggen of hier geparkeerd kan worden.

2.4.2 Collision Detection

Doordat gegevens anders verzameld worden zal de onboard computer van de auto zelf hoekpunten van objecten moeten identificeren om te kunnen gebruiken bij Collision Detection. Een manier waarop dit zou kunnen gebeuren is door het systeem figuren als rechthoeken, ovalen en driehoeken te laten identificeren en opslaan als objecten in het model. Het wordt echter lastiger als er objecten met aparte vormen in de weg staan, dit is ook nog steeds een probleem voor de huidige inparkeersystemen. Hierin is [5],[6] en wordt nog steeds veel onderzoek gedaan.

2.4.3 Information Overload

Omdat er een limiet zit aan de informatie die een computersysteem kan opslaan zal er een grens moeten zitten aan het deel van de omgeving dat het model opslaan. Omdat het niet logisch is dat iemand 100 meter van een parkeerplaats zijn auto zet voordat hij achteruit gaat inparkeren is dit echter niet rampzalig. Er zal echter wel rekening gehouden moeten worden met het feit dat *avatarTest* uit het zichtsveld van de camera's zou kunnen rijden tijdens het inparkeren. Dit zal moeten worden opgevangen door bijvoorbeeld te vereisen dat de positie van *avatarTest* nooit verder dan 100 meter van de beginpositie af mag liggen.

2.4.4 Manoeuvreren

In de praktijk is er nog een extra obstakel en dat zijn hoogteverschillen. Hoewel het in eerste instantie in dit onderzoek wel de bedoeling was om ook met hoogteverschillen te werken, is hier op een gevorderd tijdstip toch van afgezien omdat dit een hele orde aan nieuwe problemen met zich meebrengt. Zo is een lage drempel geen obstakel waarop Collision Detection plaats moet vinden, maar een liggende fiets(of wellicht persoon!) bijvoorbeeld wel. Ook kan men denken aan de problemen die het met zich meebrengt als iemand *avatarParkingBox* in een meer manoeuvreert. Het is wel duidelijk dat menselijke interactie nog steeds erg belangrijk is bij het inparkeren.

Om de avatarParkingBox te positioneren zou gebruik gemaakt kunnen worden van een touchpad scherm of knoppen op het dashboard, die de avatarParkingBox op een scherm in het dashboard doen bewegen.

Hoofdstuk 3

Leerproces

In de beginfase van de ontwikkeling van het model is gebruik gemaakt van een vierkant gratis verkrijgbaar *Model*. Dit *Model* bleek na een aantal series testen, debuggen en uiteindelijk het gebruik van de functie *BoundingSphereRenderer*, die de *BoundingSphere* van een object tekent, echter een niet gecentreerd middelpunt te hebben. Hierna is besloten om zelf een Model te maken met Blender, wat uiteindelijk relatief eenvoudig bleek en vooral ook tijdbesparend had kunnen zijn wanneer dit aan het begin was gebeurd.

Om tot de huidge methode van inparkeren te komen zijn veel methodes aan vooraf gegaan. Het was best lastig om te bedenken dat de werkende methode zo onafhankelijk mogelijk moest zijn van de plek waarvan de auto kwam aanrijden en de richting die hij op moest gaan, wat uiteindelijk neerkwam op het transformeren naar dezelfde uitgangspositie zodat de *avatarParkingBox* altijd een *Yaw* van nul had. Er zijn zeker drie zeer verschillende algoritmes geprobeerd. Een van de eerste was het werken met grotere bochten als er meer ruimte was dan de kleinste bocht vereiste. Dit berekende echter wel de juiste draaicirkel maar hield niet genoeg rekening met de *Yaw*.

Hierna is het idee van de kleinste draaicirkel gekomen. Als eerste is dit uitgeprogrammeerd voor een eenvoudig geval: *avatarParkingBox* met een *Yaw* van nul. Dit werkte verassend goed, maar helaas bleek dat dit eenvoudige geval wel erg eenvoudig was in vergelijking met de andere gevallen. Het volgende probleem was dat de volgende series algoritmes te veel leken op deze eerste. Er zijn pogingen gedaan om de ruimte te vinden waartussen *avatar* moest uitkomen als *avatarParkingBox* een bepaalde draaiing had, maar dit liep door afrondingsfouten en de oneindigheid van de tangens functie op bepaalde intervallen ook op niet veel uit.

Uiteindelijk was het het algoritme dat gebruikt was voor het Collision Detection probleem hetgene dat de oplossing bracht. Toen eenmaal ingezien

was dat dit eenzelfde soort probleem was, was het algoritme voor het inparkeren een stuk makkelijker te bedenken.

Op een bepaald punt tijdens het code van de *BoundingBox* kwam er een bug in XNA aan het licht. Het duurde even voordat duidelijk werd dat het een bug in XNA was en niet een programmeerfout. Na lang zoeken is er een workaround gevonden voor die bug, maar uiteindelijk is de functie die de bug veroorzaakte niet gebruikt.

Hoofdstuk 4

Dank

Ik wil graag Theo van Schouten bedanken voor het begeleiden van mijn scriptie, maar vooral ook voor zijn inspirerende idee(̈e)n die mij uiteindelijk deden leiden tot de huidige onderzoeksvraag.

Verder wil ik nog iedereen bedanken die me heeft aangespoord om mijn scriptie tot een goed einde te brengen, jullie weten wie jullie zijn. Ook Adam wil ik bedanken, de enige die me tijdens al die uren van nachtelijk werk heeft aangemoedigd. Als laatste wil ik nog mijn broertje bedanken, omdat ik op zijn website mijn werk kon publiceren.

Hoofdstuk 5

Appendix

5.1 Appendix A: Termen en afkortingen

- **APB** - Zie avatarParkingBox
- **avatarParkingBox** - Het object dat de plaats aanduidt waar de auto geparkeerd gaat worden
- **frustum** - Een meetkundig lichaam dat ontstaat door een piramide tussen twee vlakken te snijden.
- **model** - De wereld zoals die gemodelleerd is
- **Model** - Een model van een object waarin informatie is opgeslagen van het object
- **pivotPoint** - De vector die de positie van de draai-as representeert
- **Position** - De vector die een positie representeert
- **rotationSpeed** - De waarde die een verschil in richtingsvector representeert
- **Velocity** - De waarde die de snelheid representeert
- **WO** - Zie WorldObject
- **WorldObject** - Een object dat aanwezig is in het model
- **Yaw** - De waarde die een rotatie representeert in Pi radialen

5.2 Appendix B: Code

De code werd te onoverzichtelijk om in dit bestand te plaatsen. De volledige code is te vinden op: $http://www.vaizard-art.nl/Scriptie/BachelorScriptieKevinVriens.txt$
Voor het volledige project: $http://www.vaizard-art.nl/Scriptie/BachelorScriptieKevinVriens.rar$

Bibliografie

[1] $http : //www.ivsource.net/modules.php?name = News\&file = article\&sid = 12$

[2] Autonomous maneuvers of a nonholonomic vehicle
Paromtchik IE, Garnier P, Laugier C

[3] Multi-objective dynamic optimization with genetic algorithms for automatic parking
Daro Maravall, Javier de Lope

[4] $http : //www.arh.ukim.edu.mk/afwebnova/urban/NEUFERT_izvadoci/neufert_UD_CAR-PARKING.pdf$

[5] A survey of video processing techniques for traffic applications
Kastrinaki V, Zervakis M, Kalaitzakis K

[6] Segmentation free shared weight networks for automatic vehicle detection
Paul D. Gadera, Corresponding Author Contact Information, E-mail The Corresponding Author, Joseph R. Miramontia, Yonggwan Wona and Patrick Coffieldb

[7] Microsoft XNA Unleashed
Chad Carter

[8] McGraw-Hill: Microsoft XNA Game Studio Creator's Guide
Stephen Cawood & Pat McGee

[9] MSDN Libraby $http : //msdn.microsoft.com/en-us/library/aa139594.aspx$