Programmeerproject 1: Verslag “Space Invaders” (Fase 1)

Maxim Brabants

0576581

Maxim.Lino.Brabants@vub.be

Academiejaar 2020-2021

Inhoudsopgave

1 Functionaliteit

We starten met het ADT te programmeren dat de posities representeert van objecten in het spel. Op die manier heeft elk spelelement een positie die bijgehouden wordt zonder dat we dat telkens apart moeten implementeren in elk ADT. De plaatsbepaling gebeurt op basis van een x- en y-coördinaat maar we moeten er later mee rekening houden dat deze moeten omgezet worden naar een pixel gebaseerde positie op het scherm.

Afgezien van de posities moeten we twee belangrijke elementen implementeren :

- We moeten er zien voor te zorgen dat er een raketje op het scherm te zien is en meer specifiek onderaan het scherm. Als de raket zichtbaar is op het scherm dan is de volgende stap om hem te laten bewegen, maar slechts van links naar rechts en omgekeerd. Ook moet deze een kogel kunnen afvuren richting de alienschepen. Om het dan volledig af te werken moeten we de beweging van de raket beperken tot de randen van het spelscherm. Onze raket mag natuurlijk niet buiten de spelzone bewegen.

- Een tweede cruciaal element is het Alienvloot. Deze moeten we aan het begin van het spel laten tekenen bovenaan het scherm, tegenover de raket en deze moet voorgesteld worden in de vorm van een rechthoek. Naast de raket moet het vloot ook kunnen bewegen van links naar rechts en omgekeerd en ook naar beneden in het geval dat deze één van de spelranden raakt.

1.1 Beweging van raket

Ik opteer ervoor om te starten met de implementatie van de raket. Om goed te zijn moeten we onze raket aan het begin van elk spel onderaan het scherm laten tevoorschijn komen omdat deze natuurlijk naar boven, richting de alienschepen moet schieten. Vooraleer we de operaties beginnen programmeren moeten we een plek voorzien waar we deze kunnen definiëren. Hiervoor voorzien we een raket-ADT. Daarin kunnen we alle handelingen programmeren die de raket moet kunnen verrichten.

De constructor van de raket verwacht een positie zodat het spel op elk moment weet waar de raket zich bevindt. Ik voorzie een beweeg -en schietoperatie en ten slotte ook de dispatch-functie. De beweegfunctie van de raket heb ik als volgt geïmplementeerd: Herinner dat onze raket een positie heeft. Als we als argument een richting nemen en we geven deze dan vervolgens mee met de beweegoperatie van het positie-object, dan wordt de beweging in het positie-ADT afgehandeld en moeten we ons voor de rest geen zorgen maken. We moeten dan wel zien dat de implementatie van beweeg in het positie-ADT correct is.

```
(define (beweeg! richting)
  ((positie 'beweeg!) richting))
```

De schietoperatie gaan we later bekijken. Eerst gaan we trachten om ook een groep van alienschepen op het scherm te krijgen in de vorm van een vloot. Ik veronderstel dat dit wat meer tijd in beslag zal nemen en dat we even over de aanpak zullen moeten nadenken aangezien we alle alienschepen zowel apart als één geheel moeten kunnen manipuleren. (alienschepen moeten individueel een kogel kunnen afvuren richting de raket maar deze moeten samen als een geheel bewegen over het scherm.)

1.2 Matrix

Als we kijken naar hoe een alienvloot is opgebouwd, dan zien we dat deze bestaat uit een aantal rijen en kolommen met op elke positie een alienschip. We merken op dat dit te vergelijken is met een matrixvorm. Het proces om een alienvloot te implementeren zou dan ook een stuk gemakkelijker worden als we over een datastructuur zouden beschikken die de vorm aanneemt van zo'n matrix. We weten dat dit niet standaard aanwezig is in Scheme maar dat is helemaal geen probleem want we kunnen dit proberen zelf te programmeren.

Ik heb ervoor gekozen om eerst een matrix-ADT'tje / datastructuur te gaan programmeren zodat het achteraf eenvoudiger wordt om de alienschepen te gaan manipuleren. Ik doe dit op volgende wijze : Ik maak eerst een centrale vector aan waarbij elke locatie een rij-index voorstelt en daarna vul ik op zijn beurt elke locatie van de centrale vector met vectoren van dezelfde grootte. Op die manier verkrijg je een datastructuur die een matrixvorm aanneemt.

We kunnen in principe de vector vullen door zoveel keer vector-set! te doen als we kolommen nodig hebben, maar dan zou het achteraf moeilijker worden om het aantal rijen en kolommen van het alienvloot te wijzigen. Een mogelijke oplossing hiervoor zou zijn om dit in een functie onder te brengen die een index bijhoudt om zo maar één keer vector-set! te moeten neerschrijven in code. We hebben dus twee variabelen nodig die het aantal rijen en aliens per rij bijhouden zodat we door aanpassing van deze twee ons geheel vloot kunnen configureren naar gewenste grootte. Het bijhouden van deze variabelen doen we in een 'constanten.rkt' bestand.

```
(define (vul-vector)
  (let loop
    ((huidige-idx 0))
    (if (< huidige-idx aantal-rijen-aliens)
        (begin
          (vector-set! vector huidige-idx (make-vector aantal-aliens-per-rij))
          (loop (+ huidige-idx 1))))))
```

1.3 Implementatie alienvloot

Met behulp van onze matrix-datastructuur kunnen we nu eenvoudig beginnen met de implementatie van ons alienvloot. Natuurlijk hebben we ook iets nodig dat een individueel alienschip voorstelt, een object van het type Alienschip dat we dan in meerdere malen in onze matrix kunnen steken.

Als we eens nadenken over de operaties die het Alienschip-type moet bevatten dan komen deze eigenlijk zo goed als overeen met diegene die we in het raket-ADT hebben geïmplementeerd. Een alienschip moet namelijk ook tweedimensionaal kunnen bewegen (ook naar beneden maar veel verandert dat er niet aan) en kunnen schieten, wat dus ook handelingen zijn die de raket implementeert.

We werken een alienvloot-ADT uit en daarin houden we dus een matrix bij die dan het geheel van de alienschepen voorstelt. In een let*-expressie houden we dan een variabele 'schepen' bij en die initialiseren we met het resultaat van de expressie : '(maak-matrix)'. Deze expressie zal dan de matrix teruggeven die een vector van vectoren voorstelt. Daarnaast is het ook handig om een richting bij te houden alsook de size van de matrix (aantal rijen). Uiteraard is de matrix initieel nog niet gevuld met alienschepen. Ik kies ervoor om dit handmatig te doen in het ADT zelf. Ik maak een procedure 'vul-vloot' die elke locatie in de vector zal afgaan om zo de matrix volledig te vullen met alienschipobjectjes.

1.4 Beweging van het vloot

Nu is het eigenlijk de bedoeling dat als het alienvloot beweegt, dat dan elk alienschip na elkaar/tegelijk één eenheid moet opschuiven. Er zijn een tal van manieren waarmee we dit kunnen verwezenlijken maar het zou qua uitbreiding en elegantie gemakkelijker zijn indien we een hogere orde procedure zouden voorzien die op elk alienschip een functie zou loslaten. Deze werkwijze laat ons toe om eventueel in de toekomst uitbreidingen te vinden waarbij er 'iets' moet gebeuren met ELK alienschip in de matrix/vloot, want we kunnen dan eender welke functie meegeven aan de hogere orde procedure en deze zal dan het werk voor ons doen en die meegegeven functie toepassen op elk alienschip. Denk maar aan het tekenen van ALLE alienschepen → voor-alle-schepen : teken-alienschip! of het laten bewegen van ALLE alienschepen in het vloot → voor-alle-schepen : beweeg!.

Als gevolg hiervan kunnen we de procedure die de beweging van het vloot gaat afhandelen implementeren aan de hand van deze hogere orde procedure. Als de beweegfunctie van het alienschip-ADT correct is geïmplementeerd en we schrijven in het alienvloot-ADT nog een kleine procedure die deze beweegfunctie met een bepaalde richting zal oproepen voor een gegeven schip dan zijn we eigenlijk klaar. Hier komt de richtingsvariabele van pas die we in de let* hadden gedefinieerd, want deze moeten we telkens meegeven als richting waarnaar elk alienschip zal bewegen.

We zitten nu met een klein probleempje want op dit moment beweegt ons alienvloot volgens een bepaalde richting (links of rechts), maar op een bepaald moment gaat ons vloot de rand van ons scherm raken en zelfs erover gaan als we geen rekening houden met de grenzen van ons speelveld. Indien één van onze alienschepen de rand van het scherm raakt, dient het gehele vloot een eenheid naar onder te schuiven om dan vervolgens de tegengestelde richting uit te gaan. Voor het alterneren van de richting kunnen we een simpele functie schrijven die destructief de richting van het alienvloot zal veranderen naar de tegengestelde richting (links wordt rechts en rechts wordt links).

```
(define (switch!)
  (if (eq? richting 'rechts)
      (set! richting 'links)
      (set! richting 'rechts)))
```

1.5 Kogels afvuren

We kunnen kogels zien als objecten die door de spelwereld vliegen en objecten hebben bepaalde eigenschappen en gedrag die ze kunnen vertonen. Zo kunnen we bij een kogel denken aan de snelheid waarmee een kogel vliegt, de startpositie, etc. Daarom was mijn eerste reactie om daarvoor een ADT te ontwerpen. Als we nu eens codegewijs gaan kijken naar hoe we dit zouden kunnen programmeren dan zijn er eigenlijk toch wel twee voor de hand liggende opties. Ik ga natuurlijk één van de twee hanteren maar ik bespreek ze even kort.

Als we willen dat onze raket een kogel afschiet dan moet deze vertrekken vanuit de raket zelf. Het eerste dat we hier opmerken is dus dat de startpositie van de kogel gelijk moet zijn aan de huidige positie van de raket aangezien de kogel hier start met zijn beweging naar boven. We moeten natuurlijk in staat zijn om zijn positie constant te kennen. Nu, dat op zich is geen probleem, maar het wordt wel moeilijk in het geval dat er meerdere kogels tegelijk in het speelveld zouden aanwezig zijn.

Hier kunnen we twee gevallen onderscheiden: Er kan voorzien worden dat er per keer één kogel kan afgeschoten worden, waarmee wordt bedoeld dat pas wanneer een bepaalde kogel het speelveld

verlaat of wanneer er een alien werd geraakt, er dan pas een nieuwe kogel kan geschoten worden. Het zou in dit geval volstaan om dan een variabele : 'kogel-ADT' bij te houden in ons level-ADT en deze standaard te initialiseren op false. Pas wanneer dan een kogel wordt afgevuurd kunnen we deze false dan vervangen door een instantie van het kogel-ADT en dit is ook wat ik als eerste optie in gedachte had qua implementatie.

Het zou dan echter niet mogelijk zijn om meerdere kogels tegelijk te manipuleren, want meerdere kogels wilt zeggen dat we ergens in ons programma ook meerdere instanties moeten bijhouden van het kogel-adt (meerdere objectjes). Dit is waar onze tweede optie bij komt kijken want de tweede optie houdt in dat we een lijst zouden kunnen bij houden in plaats van één enkele variabele zodat we zo meerdere kogel-objectjes kunnen opslaan en manipuleren.

Ik ga hier analoog te werk als voor mijn aliens en alienvloot. We houden in ons Level-ADT een lijst bij van kogel-tiles. Deze lijst bestaat uit cons-cellen met in de car een object van het type kogel-adt en in de cdr de overeenkomstige tile. Zo kunnen we de assoc- functie telkens toepassen indien we een tile van een bepaalde kogel zouden nodig hebben.

Het is ook belangrijk dat we een Kogels-ADT hebben en daarin staat onze lijst als lokale toestand die we zonet besproken hebben. Ook in dit ADT definieer ik een hogere orde functie 'voor-alle-kogels (idem voor-alle-schepen) zodat we gemakkelijk een bepaalde operatie op al onze kogels kunnen toepassen. Als één van de kogels uit het scherm verdwijnt of deze heeft één van de alienschepen geraakt, dan moeten we deze kogel in kwestie best verwijderen omdat ons spel anders naargelang het aantal afgevuurde kogels trager wordt. Een kogel kunnen we gemakkelijk toevoegen aan onze lijst door onze kogels-lijst destructief aan te passen.

```
(define (voeg-kogel-toe! kogel-adt)
  (set! kogels-lijst (cons kogel-adt kogels-lijst)))
```

2 Beschrijving ADTs

2.1.1 Positie

Een belangrijk onderdeel waarmee rekening moet gehouden worden in bijna elk computerspel is de plaatsbepaling van bepaalde objecten in de spelwereld. Die moet namelijk bijgehouden worden doorheen het programma om te weten waar in de spelwereld deze objecten zich bevinden. (een voorbeeld vanuit het spelletje zelf : Als je bijvoorbeeld wilt weten of een afgevuurde kogel een alienschip heeft geraakt dan moet je zowel de positie van de kogel als die van het doelwit bijhouden zodat je weet wanneer het schip geraakt is). Het is daarom een geschikte methode om dit onder te brengen in een abstract datatype want als je bijvoorbeeld in je code werkt met x- en y-coördinaten en je moet dit op elke plaats in je programma telkens opnieuw gaan implementeren waar je dit nodig hebt dan kan je code erg lang en onoverzichtelijk worden. Ook zorgt dit voor codeduplicatie wat in alle gevallen moet vermeden worden. Het Positie ADT maakt een abstractie van een positie in een tweedimensionaal veld.

De volgende operaties horen bij dit type :

Naam	Signatuur
maak-positie	(number number → Positie)
x	(Ø → number)
y	(Ø → number)
x!	(number → Ø)
y!	(number → Ø)
gelijk?	(Positie → boolean)
Rand?	(Ø → boolean)

- De **maak-positie** operatie zal een specifiek Positie objectje aanmaken en dit stelt dan een specifieke positie voor in de spelwereld. Deze procedure verwacht twee numbers : een x- en y-coördinaat aangezien het spel gespeeld wordt in een tweedimensionale wereld. Dit zal evalueren in een Positie object waarop je dan de rest van de operaties kan toepassen.

- Als je de waardes van de x- en y-coördinaat wilt opvragen om daarmee bijvoorbeeld berekeningen te gaan doen dan kan dit door de accessormethodes **x** en **y** op te roepen die respectievelijk de x- en y-coördinaat zullen teruggeven van een bepaald object.

- Ook is het mogelijk om deze data te wijzigen. Dit kan door de mutatormethodes **x!** en **y!** aan te roepen. Je geeft een nieuwe waarde mee in de vorm van een number aan de methode van de coördinaat die je wilt wijzigen en de desbetreffende waarde zal aangepast worden.

- De **gelijk!** operator kan gebruikt worden om te kijken of twee posities identiek zijn. Dit predicaat geeft een boolean terug indien ze gelijk zijn.

- Het nut van het **rand?** predicaat is puur om te vermijden dat de waarde van een x-coördinaat de grenzen van het venster overschrijden.

2.1.2 Raket

Tijdens het spel is het natuurlijk de bedoeling dat je met de raket onderaan het scherm alle alienschepen van het vloot bovenaan probeert te vernietigen. De raket is een belangrijk element in het spel en kan dus best ook in een ADT verpakt worden met bijhorende operaties.

De volgende operaties horen bij dit type :

Naam	Signatuur
maak-raket	(Positie → Raket)
beweeg!	(symbol → ∅)
rand-geraakt?	(∅ → ∅)
Schiet!	(∅ → ∅)

- Bij de opstart van een nieuw spel of wanneer de raket nog levens over heeft dan kan de **maak-raket** methode opgeroepen worden. Het Teken ADT zal er dan voor zorgen dat er telkens in die gevallen een nieuwe raket wordt getekend op een vaste positie in de wereld.
- Voor de **beweeg!** operatie kan er een argument worden meegegeven in de vorm van een number die dan een soort van richtingsargument voorstelt. De procedure kan dan op basis van dat argument bepalen of de raket naar links of naar rechts moet verschoven worden.
- De **rand-geraakt?** functie zal de rand? -functie van de positie oproepen die dan op zijn beurt zal teruggeven of de raket al dan niet aan de rand zit.
- De **schiet!** operatie laat een kogel afvuren vanaf de positie van de raket maar die is gekend dus ook hier dient niets meegegeven te worden.

2.1.3 Kogel

Een kogel die door de spelwereld wordt geschoten kan gezien worden als een bewegend objectje in de ruimte. Het is een spelelement en kan dus rechtstreeks herleidt worden tot een ADT. Een kogel kan afgeschoten worden ofwel door een van de alienschepen ofwel door de raket.

De volgende operaties horen bij dit type :

Naam	Signatuur
maak-kogel	(Positie → Kogel)
positie	(∅ → Positie)
positie!	(number number → ∅)
geraakt?	(Positie → boolean)
verdwijn	(∅ → ∅)

- De **maak-kogel** operatie verwacht als argument de positie vanwaar hij wordt afgevuurd. Dit kan zoals gezegd de positie van een van de alienschepen zijn maar ook de positie van de raket.

- Je moet de **positie** van de kogel kunnen opvragen om te checken of de kogel iets heeft geraakt of niet.
- De positie moet ook kunnen aangepast worden met **positie!** want eens dat hij is afgevuurd beweegt hij zich in een rechte lijn naar boven dus moet deze positie constant worden aangepast doorheen zijn beweging.
- **geraakt?** bepaalt op basis van een positie of deze een ander object in de ruimte heeft geraakt.
- Na het botsen van een kogel tegen een ander object in de spelwereld moet de kogel natuurlijk verdwijnen zodat deze niet verder blijft bewegen.

2.1.4 Kogels

Dit stelt het ADT voor dat de lijst van alle actieve kogels beheert.

Naam	Signatuur
maak-kogels-ADT	(∅ → ∅)
voeg-kogel-toe!	(Kogel → ∅)
voor-alle-kogels	(<procedure> → ∅)
verwijder-kogel!	(Kogel → ∅)

- **maak-kogel-ADT** houdt simpel een lijst bij waarop je een aantal operaties kan toepassen.
- **voeg-kogel-toe!** Verwacht een argument van het type kogel en die kogel wordt ook effectief toegevoegd aan de kogels-lijst.
- **voor-alle-kogels** past net zoals voor-alle-schepen in het Alienvloot-ADT een functie toe op alle kogels in de kogels-lijst.
- **verwijder-kogel!** gaat de kogels-lijst af om de kogel die wordt meegegeven te verwijderen.

2.1.5 Alienschip

Elk alienschip apart moet de mogelijkheid kunnen hebben om zelfstandig een kogel af te vuren en uiteraard om te ontploffen wanneer deze geraakt wordt door een kogel van de raket. Een abstractie hiervan is dus zeker niet overbodig.

De volgende operaties horen bij dit type :

Naam	Signatuur
maak-alienschip	(Positie → Alienschip)
Beweeg!	(symbol → ∅)

rand-geraakt?	(∅ → ∅)
schiet!	(∅ → ∅)

- Elk gegenereerd alienschip heeft naast een eigen positie in de vector ook een unieke positie in de spelwereld op basis van een x- en y-coördinaat. Voor **maak-alienschip** dient er dan een positie meegegeven te worden wat dan evalueert naar een alienschip op die specifieke positie in de spelwereld.

- Voor de **beweeg!** operatie kan er een argument worden meegegeven in de vorm van een number die dan een soort van richtingsargument voorstelt. De procedure kan dan op basis van dat argument bepalen of het alienschip naar links of naar rechts moet bewegen.

- De **rand-geraakt?** functie zal de rand? -functie van de positie oproepen die dan op zijn beurt zal teruggeven of de raket al dan niet aan de rand zit.

- **schiet!** verwacht geen argumenten aangezien het Alienschip ADT toegang heeft tot zijn positie en dus ook vanop die positie een kogel kan laten afvuren.

2.1.6 Matrix

Dit bestand bevat de implementatie van de datastructuur die we gebruiken om ons alienvloot te construeren. Het maakt een vectormatrix en geeft deze dan onmiddellijk terug.

Naam	Signatuur
maak-matrix	(∅ → ∅)
vul-vector	(∅ → ∅)

- **maak-matrix** verwacht niets omdat deze een nieuwe vector gaat aanmaken met een size die afhankelijk is van de variabele in het ‘constanten.rkt’ bestand.

- De **vul-vector** gaat de vector in de let-expressie vullen met vectoren van dezelfde grootte dus alsook deze heeft geen argumenten nodig.

2.1.7 Alienvloot

Het geheel van alienschepen dat bovenaan het scherm beweegt abstraheren we best in een ADT zodat we het gehele vloot als één geheel kunnen manipuleren. Dit gaat het misschien makkelijker maken in het geval dat we al onze alienschepen moeten laten bewegen en dit is ook het geval in ons spel.

De volgende operaties horen bij dit type :

Naam	Signatuur
maak-alienvloot	(∅ → ∅)
beweeg!	(∅ → ∅)

Voor-alle-schepen	(<procedure> → ∅)

- Om een Alienvloot te kunnen maken zal er een bepaalde datastructuur nodig zijn die een vector van aparte Alienschip-objecten voorstelt. Een Alienvloot kun je dan best zien als een geheel van Alienschepen die over het scherm bewegen. Dit wordt teruggegeven door **maak-vloot**. Ik opteer ervoor om het alienvloot te laten voorstellen als een matrix door een vector aan te maken van meerdere vectoren. Dat zal het vinden van een specifiek alienschip misschien makkelijker maken aangezien het zoeken in O(1) gebeurt. Er hoeft geen positie meegegeven te worden omdat de aliens zelf de positie zullen bepalen van het vloot.

- Een alienvloot kan afwisselend van links naar rechts of omgekeerd over het scherm bewegen tot aan de rand. We hoeven geen argumenten mee te geven aan **beweeg!**, aangezien het vloot zelfstandig en automatisch over het scherm zal bewegen. Het ADT zal zelf bepalen in welke richting het vloot uit moet gaan.

- De **voor-alle-schepen** procedure is de hogere orde procedure die we eerder besproken hebben en die op basis van een functie het gewenste effect zal toepassen op elk alienschip.

2.1.8 Teken

Dit speciale ADT zal zijn bronnen moeten halen van een voorgedefinieerde grafische bibliotheek want anders zou het voor geen enkel type mogelijk zijn om iets te kunnen laten tekenen door het Teken ADT.

De volgende operaties horen bij dit type :

Naam	Signatuur
maak-teken-adt	(number number → Teken-ADT)
teken-spel!	(Spel → ∅)
teken-level!	(Level → ∅)
teken-Raket!	(Positie → ∅)
teken-Alienschip!	(Positie → ∅)
teken-vloot!	(Alienvloot → ∅)
teken-kogel	(Positie → ∅)

- **maak-teken-ADT** verwacht twee waarden in de vorm van numbers. Deze stellen het aantal pixels horizontaal voor en het aantal pixels verticaal.

- De **teken-spel!**-operatie gaat de teken-level! operatie oproepen.

- De **teken-level!**-operatie gaat de spelsituatie tekenen door de raket te tekenen en het vloot te tekenen.

- **teken-raket!** spreekt voor zich en gaat aan de hand van een raketobject en een overeenkomstige tile de raket mooi op het scherm tekenen.
- **teken-alienschip!** spreekt voor zich en gaat aan de hand van een alienschipobject en een overeenkomstige tile het schip mooi op het scherm tekenen.
- **teken-vloot!** is een beetje complexer. We roepen de voor-alle-schepen operatie van het alienvloot op met de teken-alienschip!-operatie.

2.1.9 Spel-ADT

Dit ADT bevat de procedure om het spel te starten door de spellusfunctie en de toetsfunctie in te stellen en door te geven aan het Teken-ADT. In het ADT worden instanties gemaakt van het Teken-ADT en het Level-ADT.

Naam	Signatuur
start	(∅ → ∅)

- In het ADT schrijven we twee procedures. Eentje die fungeert als spellusfunctie en eentje die de gebruikersinvoer gaat afhandelen. Die twee procedures moeten we meegeven aan de procedures in het Teken-ADT die via de grafische bibliotheek de callbacks gaat instellen.

2.1.10 Spel

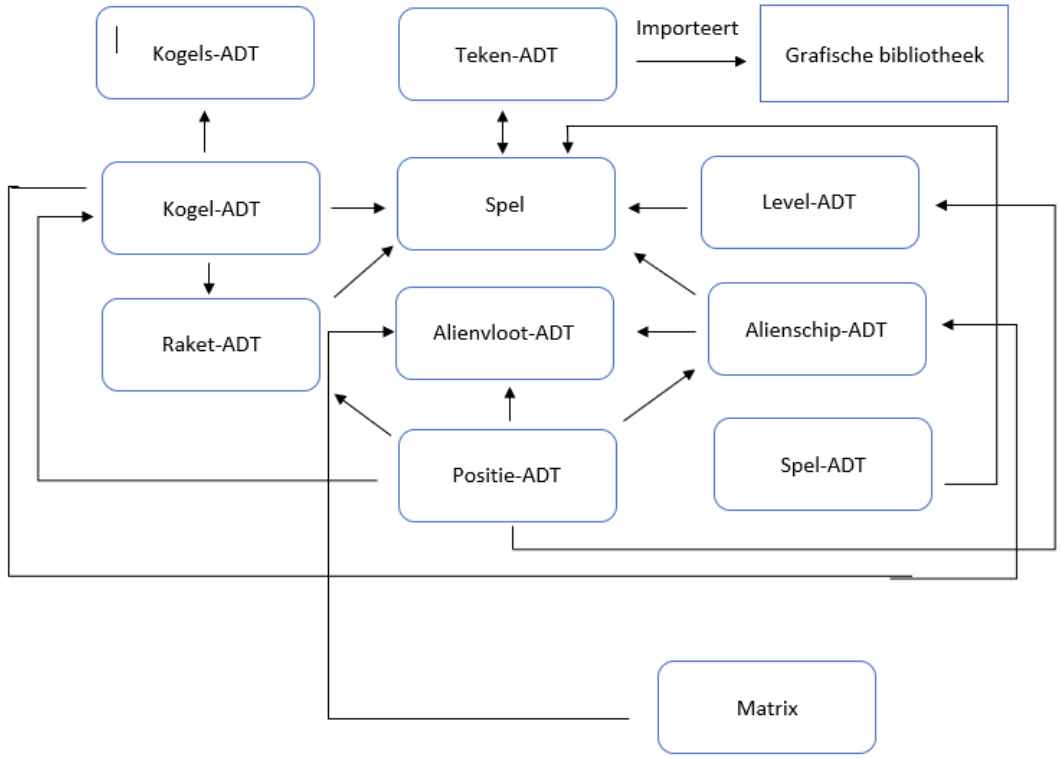
Dit stelt het centrale bestand voor dat het spel zal laten opstarten en alle benodigde ADTs zal oproepen in een logische volgorde. Deze maakt een instantie van het spel-ADT om vervolgens van dat ADT de ‘start’ procedure aan te roepen zodat het spel in gang wordt gezet.

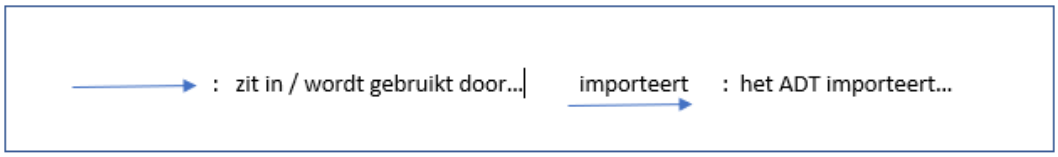
- Dit bestand laadt simpelweg de ADT’s in en maakt dan daarna een instantie van het Spel-ADT. Het is dus eigenlijk het bestand dat je moet runnen om het spel te kunnen gaan spelen.

3 Afhankelijkheidsdiagram

Er zijn bepaalde types die niet correct gaan kunnen werken zonder het bestaan van andere. Er is dus een bepaalde afhankelijkheid die dient gerespecteerd te worden om een goede werking van het gehele programma te kunnen realiseren.

Voor mijn ontwerp ziet dat er als volgt uit :





Op het schema is er te zien dat het Positie ADT toch wel een belangrijk en handig type is bij het ontwerp van het spel. Dit komt natuurlijk omdat al de objecten in de game (Kogel, Raket, Alienvloot, Alienschip) allemaal een voor een afhankelijk zijn van hun positie. Het Positie ADT is nodig in elk ADT van een spelelement.

Alsook staat Alienvloot in verbinding met Alienschip omdat Alienvloot een structuur is van meerdere Alienschepen. Als er geen Alienschepen zijn kan er ook geen Alienvloot bestaan. Door het Teken ADT onderscheiden we de spellogica en de tekenlogica van elkaar. Enerzijds heeft het Spel ADT het Teken ADT nodig om specifieke dingen te tekenen op het scherm en anderzijds heeft het Teken ADT ook het Spel ADT nodig om te weten wat er juist getekend moet worden (dubbele pijl). Daarvoor worden namelijk de specifieke teken-procedures aangeroepen.

Het vierkante vak staat niet voor een zelf gedefinieerd type maar het is een bestand dat het Teken ADT in staat stelt om grafische elementen van te halen en vervolgens te tekenen in ons spel. Het teken ADT zal weliswaar een tekenmethode bevatten voor elk spelelement.

4 Werktijd

Dit zijn de reeds uitgevoerde taken per week.

Week	**Taak**
Week 10 -11	Positie ADT geïmplementeerd en begonnen met implementatie van Raket ADT.
Week 12	Raket laten bewegen onderaan het scherm.
Week 13	Implementatie van Matrix ADT.
Week 14	Probleem met window-sizing dat veroorzaakt wordt door de grafische bibliotheek en besturingssysteem. (mail gestuurd naar Bjarno)
Week 15	Alienvloot laten bewegen over het scherm.
Week 15 - 16	Proberen beweging van objecten te beperken tot de randen van het speelveld.
Week 16 - 17	Kogel-ADT implementeren + raket kogels laten afvuren.
Week 18	Aliens laten raken door de afgevuurde kogels.
Week 19	Taak van week 18 opvangen in fase 2
Week 20	Verslag afschrijven