Programmeerproject 1:

Verslag “Space Invaders”

(Fase 2)

Maxim Brabants

0576581

Maxim.Lino.Brabants@vub.be

Academiejaar 2020-2021



Inhoudsopgave

[1 Functionaliteit 3](#_Toc72097730)

[1.1 Beweging van raket 3](#_Toc72097731)

[1.2 Implementatie alienvloot 4](#_Toc72097732)

[1.3 Beweging van het vloot 5](#_Toc72097733)

[1.4 Kogels afvuren 5](#_Toc72097734)

[2 Beschrijving ADTs 6](#_Toc72097735)

[2.1.1 Positie 6](#_Toc72097736)

[2.1.2 Raket 8](#_Toc72097737)

[2.1.3 Kogel 9](#_Toc72097738)

[2.1.4 Kogels 9](#_Toc72097739)

[2.1.5 Score 10](#_Toc72097740)

[2.1.6 Power-Up 11](#_Toc72097741)

[2.1.7 Alienschip 11](#_Toc72097742)

[2.1.8 Alienvloot 12](#_Toc72097743)

[2.1.9 Teken 14](#_Toc72097744)

[2.1.10 Level-ADT 15](#_Toc72097745)

[2.1.11 Spel-ADT 18](#_Toc72097746)

[2.1.12 Spel 19](#_Toc72097747)

[3 Afhankelijkheidsdiagram 19](#_Toc72097748)

[4 Werktijd 21](#_Toc72097749)

# 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.

## 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.

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.)

## Implementatie alienvloot

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.

In het alienvloot-adt heb ik ervoor gezorgd dat bij aanmaak van zo’n vloot, er dan standaard zo’n matrixvorm wordt gemaakt 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.

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 nog andere operaties…

We werken een alienvloot-ADT uit en daarin houden we dus een matrix bij die dan het geheel van de alienschepen 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.

## 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).

## 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.

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 kies voor de tweede optie en 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.

Om dan de alienschepen te kunnen vernietigen met de kogels, voorzien we dan één procedure ‘check-kogels-geraakt’ die voor elke kogel in de kogels-lijst nakijkt of die de raket of een alienschip heeft geraakt. Om het gemakkelijk te maken houden we rekening met de functionaliteit die nog moet komen en gaan we elke kogel taggen zodat we op elk moment in ons programma weten van wie de kogel afkomstig is (ofwel raket ofwel alien). Voor elke kogel onderscheiden we hierbij de volgende gevallen:

* De kogel komt van de raket en hij raakt een alienschip dat meer dan 1 leven heeft.
* De kogel komt van de raket en hij raakt een alienschip dat slechts 1 leven heeft.
* De kogel komt van een alienschip en hij raakt de raket die meer dan 1 leven heeft.
* De kogel komt van een alienschip en hij raakt de raket die meer dan 1 leven heeft.

# Beschrijving ADTs

### 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 | → number) |
| y | → number) |
| x! | (number → |
| y! | (number → |
| gelijk? | (Positie → boolean) |
| rand-horizontaal? | 🡪 boolean) |
| rand-verticaal? | 🡪 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-horizontaal?** predicaat is puur om te vermijden dat de waarde van een x-coördinaat de linker -en rechterrand van het venster overschrijdt.
* Het nut van het **rand-horizontaal?** predicaat is puur om te vermijden dat de waarde van een y-coördinaat de boven -en onderrand van het venster overschrijdt.

### 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? | (→ boolean) |
| verminder-levens! | (→ ) |
| voeg-leven-toe! | (→ ) |
| toggle-schild! | (→ ) |
| toggle-upgrade! | (→ ) |
| reset-levens! | (→ ) |

* 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 symbol 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-horizontaal? -functie van de positie oproepen die dan op zijn beurt zal teruggeven of de raket al dan niet aan de rand zit.
* **verminder-levens!** wordt opgeroepen wanneer de raket wordt geraakt door een kogel van een alien en vermindert de levens met 1.
* **voeg-leven-toe!** deze operatie kan gebruikt worden voor een power-up en voegt een leven toe aan de raket.
* **toggle-schild!** deze operatie is ook bedoeld als power-up
* **toggle-upgrade!** deze operatie is ook bedoeld als power-up
* **reset-levens!** wordt opgeroepen als het game-over is en de levens terug moeten worden gezet.

### 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. Daarom taggen we de kogels met een symbool zodat we op elk moment weten vanwaar de kogel komt.

De volgende operaties horen bij dit type :

|  |  |
| --- | --- |
| Naam | Signatuur |
| maak-kogel | (Positie → Kogel) |
| beweeg! | (symbol 🡪 |
| positie | (→ Positie) |
| toggle-type! | ( 🡪 ) |

* 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.
* De kogel moet zijn positie kunnen doen veranderen en dat gebeurt aan de hand van de **beweeg!** operatie die een richting aanneemt in de vorm van een symbol en die dan volgens die richting zal bewegen.
* Je moet de **positie** van de kogel kunnen opvragen om te checken of de kogel iets heeft geraakt of niet.
* **toggle-type!** zal het type veranderen naar het andere type. Dit doen we als we de kogels willen laten kaatsen met de schild-power-up

### Kogels

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

|  |  |
| --- | --- |
| Naam | Signatuur |
| maak-kogels-ADT | ( → 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.

### Score

De score die we moeten bijhouden doorheen het spel bestaat uit twee onderdelen die we gemakkelijk kunnen samennemen en verpakken in een ADT met operaties die hieronder worden beschreven. Het komt erop neer dat we twee variabelen nemen als lokale toestand in het adt (huidige score en hoogste score) en die we uiteindelijk met behulp van onderstaande operaties kunnen manipuleren. Op deze manier kunnen we een objectje van dit type bijhouden in het level-adt en ons geen zorgen maken over waar we de procedures van de score moeten bewaren alsook de scores zelf.

|  |  |
| --- | --- |
| Naam | Signatuur |
| maak-score | ( → Score) |
| verhoog-score! | (number → ) |
| reset-score! | ( → ) |
| verander-hoogste! | (number→ ) |
| meer-dan-hoogste? | ( → boolean) |

* We roepen **maak-score** op in het level-adt om alle gegevens van de score in een verpakt objectje te verkrijgen om zo later deze te gaan manipuleren op elk gewenst moment.
* **verhoog-score!** incrementeert de huidige score met een gegeven aantal punten.
* **reset-score!** zet de huidige score terug op 0 zetten (in het geval van een game-over (maak-nieuw-spel))
* **verander-hoogste!** moet ook opgeroepen worden bij een game-over omdat op die moment de high-score moet bepaald worden.
* Ten slotte is **meer-dan-hoogste?** nog een handig predicaatje dat de vergelijking maakt van de huidige en hoogste score.

### Power-Up

De huidige paragraaf beschrijft een integraal onderdeel deel uitmakende van de functionaliteit die vereist is in fase 2. We maken hiervoor een abstract datatype aan dat kan fungeren als elke mogelijke power-up. Dit is te realiseren door bij de aanmaak van een power-up willekeurig een type te laten kiezen uit een vector en dan kunnen we op basis van dat type in het level-adt de juiste activatieprocedure aanroepen.

|  |  |
| --- | --- |
| Naam | Signatuur |
| maak-power-up | (Positie → ) |
| beweeg! | ( → ) |
| toggle-actief! | ( 🡪 ) |

* Net zoals de ‘maak-kogel’ constructor dienen we alsook aan **maak-power-up!** een positie mee te geven aangezien deze op een variabele positie kan aangemaakt worden .
* **beweeg!** verwacht geen extra argumenten aangezien een power-up enkel naar beneden kan bewegen richting de raket.
* **toggle-actief!** zet de huidige power-up op actief. Dit is belangrijk om te weten in het level-adt. Zo moeten we bijvoorbeeld enkel de ‘check-power-up-geraakt!’ functie laten uitvoeren op het moment dat de power-up al bestaat maar nog niet actief is.

### 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? | (→ boolean) |
| onderkant? | (→ boolean) |
| kleur! | (→ ) |
| levens! | (→ ) |
| zet-inactief! | (→ ) |

* 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 symbol 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-horizontaal? -functie van de positie oproepen die dan op zijn beurt zal teruggeven of de raket al dan niet aan de rand zit.
* **onderkant?** gaat checken of de alien in kwestie de raket bereikt heeft. Hier wordt de y-positie vergeleken.
* **kleur!** Is een destructieve operatie die de kleur/soort van een alien definieert. Het wordt aangeroepen wanneer er een nieuwe alien wordt aangemaakt en toegevoegd aan het vloot.
* **levens!** is ook een destructieve operatie die de levens van een aangemaakt alienschip zal bepalen aangezien deze afhankelijk zijn van de kleur/soort.
* **zet-inactief!** is bedoeld om een alien te verwijderen zodra deze geraakt is maar aangezien we ons vloot voorstellen als een vector van vectoren kunnen we niet echt een alien uit de vector verwijderen dus houden we een status bij die aangeeft of de alien al dan niet actief is. (inactief zetten = positie van alien buiten het speelveld zetten)

### 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 | ( → Alienvloot |
| vul-vloot! | (symbol 🡪 ) |
| beweeg! | ( → |
| voor-alle-schepen | (<procedure> 🡪 |
| verwijder-schip! | ( → |
| zet-vloot-terug! | ( → |
| verhoog-vernietigde-schepen! | ( → |
| reset-onderkant-geraakt! | ( → |
| reset-vloot-vernietigd! | ( → |
| reset-aantal-vernietigde-schepen! | ( → |

* 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. Er hoeft geen positie meegegeven te worden omdat de aliens zelf hun eigen positie zullen bepalen in het vloot.
* De **vul-vloot!** is de procedure die verantwoordelijk is voor het vullen van het vloot wanneer maak-alienvloot wordt aangeroepen. We roepen deze procedure dan ook onmiddellijk op als we een instantie hebben van het alienvloot-adt. De reden dat we dit hebben gedefinieerd als een procedure is omdat we deze later ook gebruiken om een nieuw vloot te vullen/initialiseren (bijvoorbeeld bij een game-over).
* 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. We lossen dit op door een richting bij te houden in het adt die dan destructief wordt aangepast afhankelijk van de situatie. Het ADT zal zelf bepalen in welke richting het vloot uit moet gaan. Deze procedure neemt ook het checken van de randen op zich. Dit gebeurt als volgt: er wordt over alle schepen gegaan en per alien worden de volgende mogelijkheden nagekeken: zit de alien aan de rand, heeft de alien de raket bereikt. In het algemeen wordt er oo nog gecheckt of alle aliens vernietigd zijn door vast te stellen dat ze al dan niet allemaal op ‘inactief staan.
* 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.
* **verwijder-schip!** krijgt een alien mee en zal in het vloot op zoek gaan naar die alien en deze vervolgens op inactief zetten.
* **zet-vloot-terug!** wordt aangeroepen bij de activatie van één van de vijf power-ups. Alle alienschepen worden iteratief terug op hun beginpositie gezet.
* **verhoog-vernietigde-schepen!** incrementeert een variabele die het aantal vernietigde schepen bijhoudt en is bedoeld om de regelmaat van de vrijgave van power-ups te bepalen.
* **reset-onderkant-geraakt!** past simpelweg een boolean aan en het is nodig in het level-adt. Daar wordt bepaald wat er moet gebeuren als de onderkant is geraakt. (game-over)
* **reset-vloot-vernietigd!** past simpelweg een boolean aan en het is nodig in het level-adt. Daar wordt bepaald wat er moet gebeuren als alle aliens zijn vernietigd door de raket.
* **reset-aantal-vernietigde-schepen!** past simpelweg een boolean aan en is nodig in het level-adt.

### 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-huidige-score! | (Score 🡪 |
| teken-hoogste-score! | (Score 🡪 |
| teken-levens! | (Raket 🡪 |
| teken-power-up-image! | ( → |
| toggle-raket-schild! | (Raket → |
| verwijder-power-up-image! | ( → |
| verwijder-kogel! | (Kogel → |
| verwijder-vloot! | (Alienvloot → |
| verwijder-power-up! | (Power-up 🡪 ) |

* **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 met een Spelinstantie.
* De **teken-level!-**operatie gaat de gehele spelsituatie herhaaldelijk gedurende heel het spel blijven tekenen om alles bijgewerkt te houden. Onder andere : teken-raket!, teken-vloot!, teken-kogels! en teken-power-up! worden hierin aangeroepen om de tiles van deze elementen telkens op de juiste plek te tekenen.
* **teken-huidige-score! :** huidige score visueel up-to-date houden via het Level-adt (krijgt score-object van Level-ADT)
* **teken-hoogste-score! :** hoogste score visueel up-to-date houden via het Level-adt (krijgt score-object van Level-ADT)
* **teken**-**levens!** Gaat visueel de levens up-to-date houden. (krijgt levens van level-adt)
* **teken-power-up-image!** tekent gewoon een bepaalde image in de rechterbovenhoek ter indicatie dat een power-up is opgenomen en dat deze kan geactiveerd worden.
* **toggle-raket-schild!** wordt aangeroepen wanneer de schild-power-up wordt geactiveerd. Hierdoor wordt de raket getekend maar dan met een schild zodat de speler weet dat de schild-power-up is geactiveerd. De operatie doet dit aan de hand van de schild? boolean van het raket-adt. Daarom wordt ‘Raket’ meegegeven.
* **verwijder-power-up-image!** is de tegenhanger van teken-power-up-image! en laat de image terug verdwijnen van het scherm zodra de power-up geactiveerd/gebruikt is.
* **verwijder-kogel!** neemt een kogel aan, gaat vervolgens in de kogel-tiles-lijst zoeken naar een associatie die overeenkomt met de meegegeven kogel en verwijdert de tile ten slotte van de laag.
* **verwijder-vloot!** gaat over alle alienschepen en past ‘verwijder-alien!’ toe. Deze gaat gelijkaardig te werk als ‘verwijder-kogel!’ en verwijdert de tiles van alle aliens op het scherm. Wanneer bv. een nieuw spel moet aangemaakt worden dan moet deze worden aangeroepen om eerst alle tiles te verwijderen en dan met ‘teken-vloot!’ weer alle tiles tekenen.
* Als een power-up uit het scherm verdwijnt of de raket vangt hem op dan wordt de overeenkomstige tile van het scherm gehaald door **verwijder-power-up!.**

### Level-ADT

Een zeer belangrijk adt dat instaat voor het bijhouden van de spelobjecten en de spellogica die ermee geassocieerd is. Het omvat heel wat operaties onder andere de update! -functie die fungeert als de spellusfunctie die we ingesteld hebben in het teken-adt. Enerzijds hebben we dus de ‘update!’ die herhaaldelijk opgeroepen wordt in het level-adt en anderzijds ‘teken-spel!’ die ook constant uitgevoerd wordt in het teken-adt. Dit zorgt voor een volledige scheiding tussen het tekenen van de spelelementen en het bepalen van de logica.

De volgende operaties horen bij dit type :

|  |  |
| --- | --- |
| Naam | Signatuur |
| maak-level | (number number 🡪 Level-adt) |
| beweeg-raket! | (symbol 🡪 |
| beweeg! | ( 🡪  (alle beweegoperaties) |
| schiet! | ( 🡪  (alle schietoperaties) |
| check-kogels-geraakt! | ( 🡪 |
| creëer-power-up! | (Alien 🡪 |
| activeer-power-up! | (symbol 🡪 |
| roep-power-up-op! | ( 🡪 |
| check-power-up-einde! | ( 🡪 |
| check-power-up-geraakt! | ( 🡪 |
| power-up! | ( 🡪  (alle activatieprocedures) |
| vergelijk-met-hoogste! | ( 🡪 |
| bepaal-score! | (Alien 🡪 |
| check-vloot! | ( 🡪 |
| maak-nieuw-spel! | ( 🡪 |
| update! | (number 🡪 |
| toets! | (symbol 🡪 |

* Om een instantie van het level-adt te maken kunnen we **maak-level** aanroepen met twee argumenten die het aantal cellen hoogte en breedte voorstellen en een derde additioneel argument voor het teken-adt want zoals besproken geven we dit mee met de reden het tekenen van spelobjecten te vereenvoudigen vanuit het level-adt. Anders zouden we aan elke procedure een teken-adt argument moeten meegeven als we die nodig zouden hebben.
* **beweeg-raket!** is een operatie die behoort tot onze key-callback die deel uitmaakt van de grafische bibliotheek. Je geeft een symbool mee als argument en de operatie zal via het positie-adt de positie van het raket-object wijzigen.
* Dan hebben we vervolgens nog onze **beweeg!** procedures van de resterende elementen in het spel die we in de update-callback plaatsen wat ook een onderdeel is van de grafische bibliotheek. Het zal gelijkaardig de positie bewerken via het positie-adt. (beweeg-vloot!, beweeg-kogels!, beweeg-power-up!)
* We hebben twee schietoperaties namelijk die van de raket en die van de alien(s). Die van de raket zit uiteraard in de toets-functie want die moet getriggered worden bij een toetsaanslag. In ons geval is dat de spatiebalk. In tegenstelling tot de schiet-functie van de alien die in de update-functie gevat wordt. Deze moet per tijdstip uitgevoerd worden.
* In ons level-adt houden we een lijst bij met alle kogels die afgevuurd zijn en nog in het spel zijn. De lijst omvat zowel de kogels van de raket als die van de aliens, maar dat is geen probleem want dat hebben we opgelost door ze te taggen met een symbool. In **check-geraakt!** gaan we dan over deze lijst en checken we per kogel of deze ofwel de raket of één van de aliens raakt. Als het een raketkogel is dan checken we enkel op het raken van de aliens en andersom checken we enkel op het raken van de raket.
* Globaal in het level-adt houden we een power-up variabele bij die standaard op false staat en als er een power-up moet gemaakt worden dan zal **creëer-power-up!** ervoor zorgen dat deze wordt geïnstantieerd met een power-up van het power-up-adt.
* Als je met de raket een power-up hebt opgevangen dan is deze standaard nog niet actief. Je zult daarvoor op de tab-toets moeten drukken om werkelijk het effect ervan te ondervinden. Het opvangen van die toetsaanslag gebeurt door **activeer-power-up!**. De aanroep bevindt zich in de toets-functie.
* In het adt staan de activatieprocedures voor elke power-up (1 per power-up). **roep-power-up-op!** zal uitmaken welke van die moet aangeroepen worden.
* Sommige power-ups zijn tijdsgebaseerd en dienen uitgeschakeld te worden na een termijn. De **check-power-up-einde!** gaat hiervoor zorgen.
* We gebruiken een paar booleans om een onderscheid te kunnen maken tussen het opnemen van de power-up (raken van de raket) en het effectief activeren ervan. Als we willen weten of deze geraakt is dan kunnen we **check-power-up-geraakt!** oproepen die eigenlijk doorheen het spel constant checkt of een power-up door de raket is opgenomen. (🡪 update-callback)
* Dan heb je de activatieprocedures van de power-ups die niets van argumenten verwachten maar meestal gewoon destructief dingen zullen gaan aanpassen om zo de gewenste effecten van de power-up te bekomen.
* **vergelijk-met-hoogste!** gaat na een spel (game-over) nakijken of de huidige score de hoogste score heeft overschreden. Hierin komen operaties van het score-adt te pas en wordt de high-score ook telkens visueel geüpdated on-screen.
* **bepaal-score!** wordt opgeroepen vanuit ‘check-geraakt!’. Deze neemt het updaten van de huidige score op zich. Zowel van het object als visueel op het scherm voor de speler. Na het vernietigen van 1 enkele alien wordt deze getriggered.
* **check-vloot!** hoort eveneens thuis in de update-callback! omdat deze de gevallen zal opvangen waarin het vloot de onderkant raakt en alle aliens in het vloot vernietigd zijn.
* **maak-nieuw-spel!** is verantwoordelijk voor het aanmaken van een nieuw spel (t.t.z. herinitialiseren van alle elementen) bij een game-over of bij de overgang naar een nieuw level.
* Tot slot hebben we onze **update!** en **toets!**-functies die respectievelijk per tijdsinterval de instructies in de body zal uitvoeren en de toetsaanslagen van de speler zal verwerken.

### 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. Bij de aanmaak van het level-adt geven we het teken-adt mee zodat in het geval we iets moeten tekenen, dat we dan op elk moment de nodige teken-operaties kunnen laten uitvoeren. We kunnen eigenlijk zeggen dat het spel een samenwerking is tussen het teken -en level-adt waarbij het level-adt alle objecten van de spelelementen bewaart + de spelsituatie bepaalt en het teken-adt alles tekent.

|  |  |
| --- | --- |
| 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.

### Spel

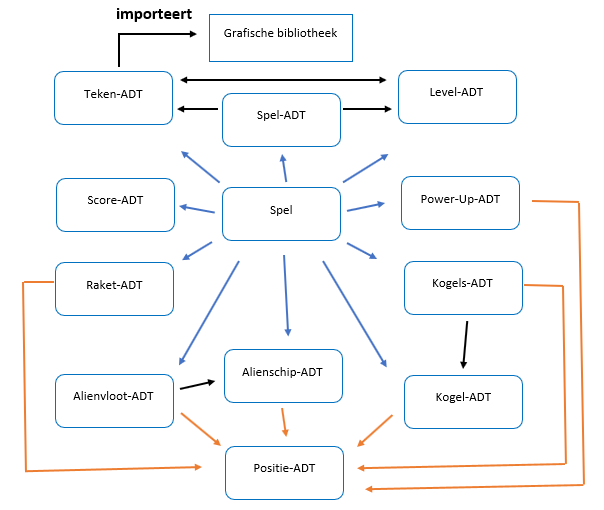
Dit stelt het centrale bestand voor dat het spel zal laten opstarten en alle benodigde ADTs zal oproepen in een logische volgorde. Deze maakt een instantie van het spel-ADT om vervolgens van dat ADT de ‘start’ procedure aan te roepen zodat het spel in gang wordt gezet.

* 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.

# 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 garanderen.

Voor mijn ontwerp ziet dat er als volgt uit :



**Legende / Conclusie :**

Op het diagram 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, Power-Up) allemaal een voor een afhankelijk zijn van hun positie. Het Positie ADT is nodig in elk ADT van een spelelement. We kunnen dus besluiten dat dynamische spelelementen afhankelijk zijn van een positie en die positie wordt bepaald door het Positie-ADT. Deze elementen bewegen namelijk over het scherm waardoor de positie zeer belangrijk wordt. Statische elementen zoals de weergave van de score hebben in mijn ontwerp niet echt nood aan een positie-type omdat hun positie nooit veranderd dus we moeten zeg maar geen operaties toepassen op zijn positie. (oranje pijlen)

Als we dan een niveau hoger gaan kijken in het diagram dan zien we dat het Alienvloot afhankelijk is van het Alienschip-type omdat Alienvloot een structuur is van meerdere Alienschepen. Als er geen Alienschepen zijn, kan er ook geen Alienvloot bestaan. (t.t.z. dan stelt een Alienvloot niet veel voor) In ons spel vullen we dan ook het Alienvloot met objectjes van het type Alienschip vlak na de aanmaak van een vloot. Het is een zeer doeltreffende abstractie die eigenlijk niet weg te denken is uit het spel door het feit dat we heel veel operaties op alle alienschepen tegelijk moeten uitvoeren alsook sommige onderdelen die moeilijk te implementeren zouden zijn zonder dit type. (zwarte enkele pijl)

We kunnen een gelijkaardige uitleg besteden aan de afhankelijkheid tussen het Kogel -en Kogels-type. Het bijhouden van meerdere kogels tegelijk doorheen het spel en het toepassen van verschillende operaties op die kogels verklaart eigenlijk de notie aan een Kogels-ADT. In mijn ontwerp doet dat zich voor als een lijstabstractie die Kogel-types kan bijhouden en daarop manipulaties kan uitvoeren. Deze afhankelijkheid komt eigenlijk voort uit min of meer dezelfde gedachtengang als het Alienvloot-type. (zwarte enkele pijl)

Onze meest belangrijke en veelzijdige dependency kunnen we centraal waarnemen in het diagram. We merken op dat vanuit deze plek (Spel) een pijl vertrekt naar elk ADT. (ook pijl naar Positie-ADT) ‘Spel.rkt’ is het bestand dat we moeten runnen als we het spel willen spelen en tijdens de implementatie en het testen van het spel op verschillende momenten ondervonden we dat er in alle andere adt’s geen syntax-errors of andere code-gerelateerde fouten mochten aanwezig zijn alvorens we echt konden beginnen met spelen. Met andere woorden kan het spel dus niet worden gespeeld als alle andere ADT’s niet correct worden geïmplementeerd. (blauwe pijlen)

Bovenaan bespreken we eerst de verbinding van Spel-ADT met Teken-ADT en Level-ADT. Deze maakt eigenlijk de verbinding tussen Teken -en Level-ADT mogelijk omdat we het Teken-ADT meegeven aan het Level-ADT. In het Spel-ADT worden de toets -en spellusfunctie gedefinieerd. De toets! -en update! worden van het Level-ADT gehaald en de teken-spel! van het Teken-ADT. Ten slotte worden deze geset! In het Teken-ADT. Zowel het Teken als Level-ADT moeten voorzien worden van de juiste procedures opdat het spel correct gaat willen werken. De wederzijdse afhankelijkheid tussen Level en Teken is te verantwoorden omdat het Level-ADT de teken-procedures moet kunnen aanroepen en het Teken-ADT de objecten op de juiste plaats moet kunnen tekenen a.d.h.v. de overeenkomstige positie. (zwarte enkele + dubbele pijlen bovenaan)

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. (uiterst bovenaan)

# 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 |