Vrije Universiteit Brussel

Programming assignment

A game in assembly

Othello

Authors:
Kriz De Logi
Linda de Corte

January 3, 2015

Contents

1 Inleiding

1.1 De opdracht

De opdracht was duidelijk van het begin, maak een spel in of een image encoder/decoder in assembly. De keuze ging uiteindelijk naar het maken van het spel Othello, ook wel bekend als Reversi op de oude Nokia telefoons. Het concept van het spel is duidelijk: een bord met 64 vakjes dient voor het plaatsen van steentjes die voor beide spelers resp. wit en zwart zijn ingekleurd. Bij het zetten van een witte steen worden alle zwarte stenen tussen de witte stenen en de juist geplaatste witte steen omgezet naar witte stenen. Dit wordt het 'stelen' van stenen genoemd. Hetzelfde principe geldt voor de omgekeerde situatie. De speler die aan het einde van het spel de meeste stenen op het bord heeft liggen wint het spel.

1.2 Gebruiksaanwijzing

Othello word in principe altijd gespeeld op een bord van 8 bij 8. Er zijn twee kleuren stenen: wit en zwart. En speler gebruikt zwarte stenen, de andere witte stenen. In het begin van het spel liggen er vier stenen in het midden, twee zwarte en twee witte die diagonaal van elkaar liggen. Het spel begint altijd met zwart, en de spelers maken om de beurt een zet.

Als een speler aan de beurt is plaatst hij een steen op het bord. Hij mag de stenen op een van de lege plaatsen op het bord zetten indien deze grenst aan een witte steen (ook diagonaal) waarbij in dezelfde richting nog een zwarte steen aanwezig is. M.a.w. er moet een 'diefstal' van stenen kunnen gebeuren bij het leggen van een steel. In deze versie van Othello kan je de mogelijke zetten herkennen aan de hand van vakjes die lichtgroen zijn gekleurd, terwijl de andere vakjes een donkergroene kleur hebben.
Het spel eindigt wanneer geen zetten meer mogelijk zijn: indien het speelveld volledig gevuld is met stenen of indien er voor beide partijen geen zetten meer kunnen gebeuren met 'diefstal'. Het spel is dan gewonnen door de speler met de meeste stenen op het veld.

De tussenstand tijdens het spelen is zichtbaar aan de rechterkant van het speelbord. Tevens is er aangeduid welke kleur aan zet is a.d.h. van een pijl. Het spel kan beindigt worden door te klikken op de rode knop die zich rechtsboven bevindt.

2 Het ontwerp

2.1 Design model

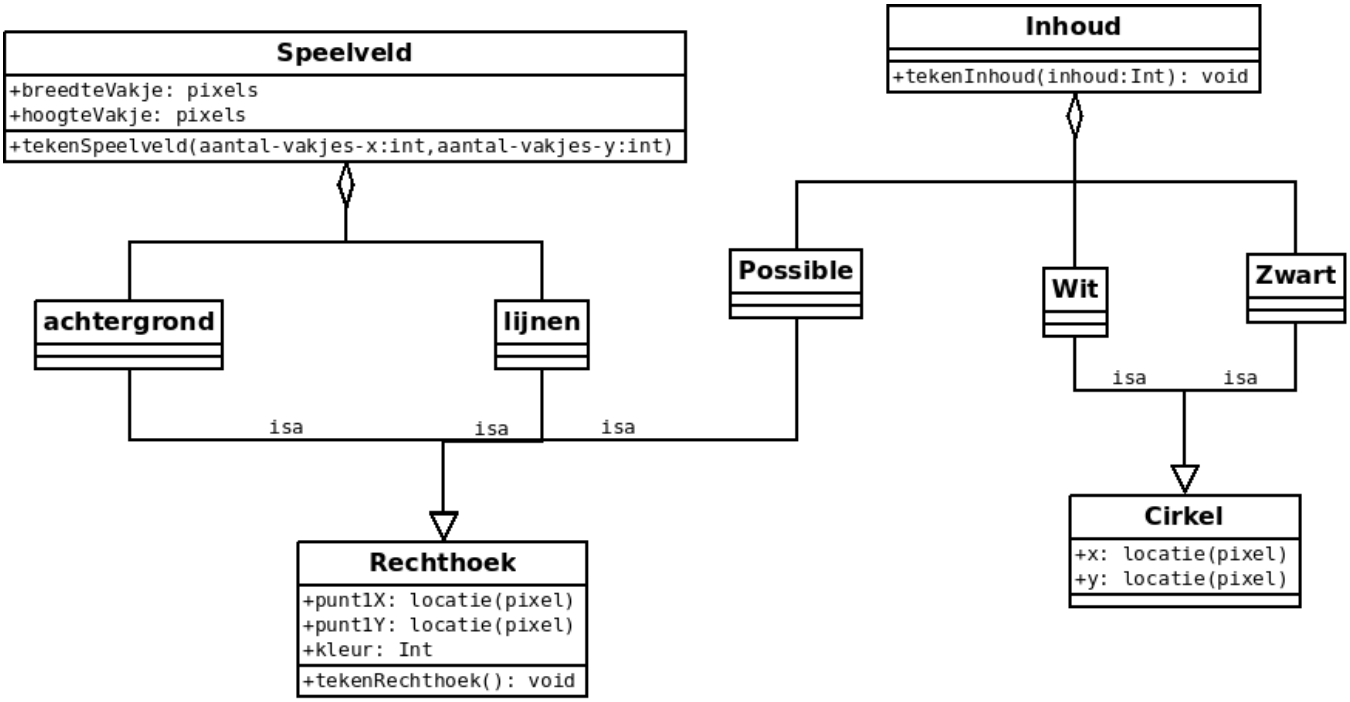

Figure 1: Diagram van de graphics

Zoals te zien is in het diagram, zijn alle tekeningen gebouwd op de rechthoek. Zodat elke tekening kan gemaakt worden zonder pixel per pixel te werken, maar dus a.d.v. combinaties van rechthoeken.

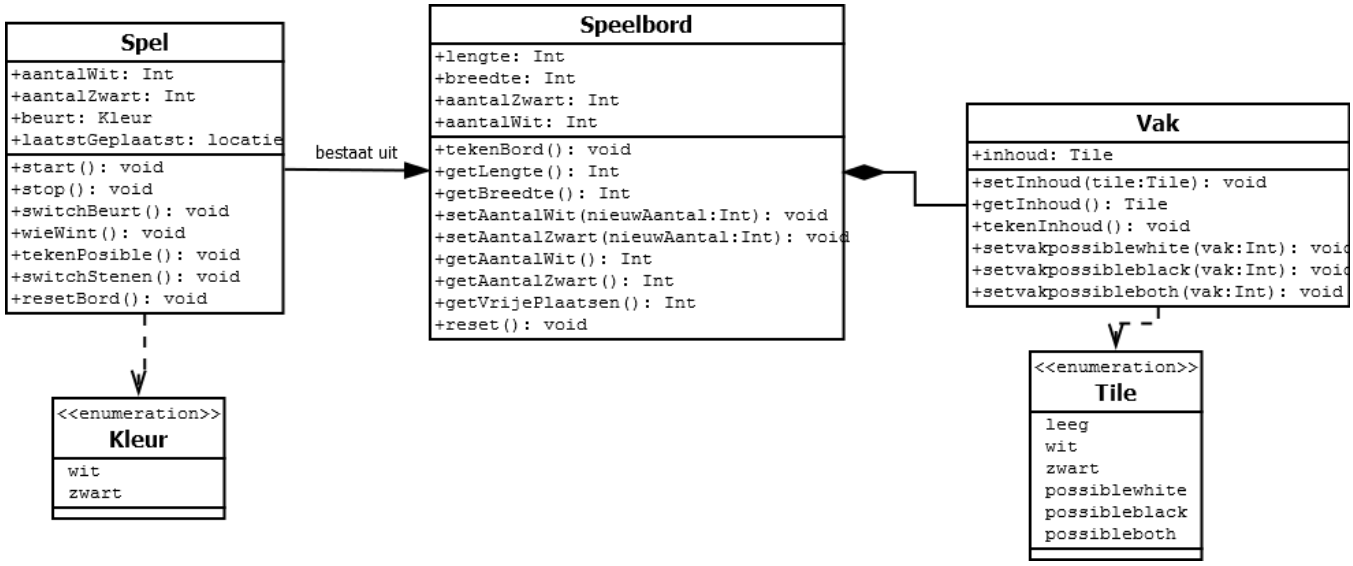

Figure 2: Diagram van het spel

Het spel bestaat uit een speelbord, die in dit project in een array van 64 bytes word voorgesteld, iedere byte bevat de informatie van de inhoud van het respectievelijke vak. Verder wordt er bijgehouden wie er aan de beurt is en hoeveel stenen ieder op het bord heeft liggen op dat moment.

3 Problemen

3.1 Conditionele jumps

Vrij snel na het schrijven van onze basis code kwamen we er achter dat een conditionele jump maar een beperkt bereik heeft. Wat vrij lastig is als er veel nodig zijn. Om dit te omzeilen hebben maken we gebruik van naamloze labels.
Voorbeeld:

```
vergelijkzwart:
        mov ax, 0
        mov al, [speelveld+bx]
        cmp al, 2
        jne @F
        jmp swapstart
@@:
        cmp ax, 3
        jne @F
        jmp start
@@:
        jmp done
```

3.2 Maximaal 3 tot 4 asm files

Na het schrijven van alle code en het splitsen in verschillende files deed er zich een probleem voor. Namelijk dat CodeView niet kan werken met een argumentenlijst die langer is dan 255 tekens. Kortere namen gebruiken was ook geen optie, de code werd onleesbaar en lang. Na een gesprek met Tim Bruylants vernamen we dat we de functies beter konden samenvoegen in maximaal 3 a 4 files en dat variabelen maar een maal gedeclareerd mogen worden. Dit deden we vaker waardoor ons programma veel te veel geheugen nodig had.
We hebben toen besloten om alle teken functies samen in een file te zetten, de logica functies in een file en een main file te maken. Zodanig dat alle functies mooi geclasseerd staan in een module.

3.3 Elementen van de stack halen

Het gebruiken van een array als manier voor het doorgeven van functiewaarden is niet optimaal. Hiervoor wordt het best gewerkt met de stack. De vooraf gepushte variabelen kunnen in een functie worden uitgelezen aan de hand van de base-pointer. Bij NEAR procedures moet er bij deze base-pointer 4 worden opgeteld, bij FAR procedures is dit 6. De laatst meegegeven waarde kan dan worden uitgelezen op locatie [0], de volgende op [2], enzovoort. (indien er bytes op de stack werden gepusht verhoogt dit getal uiteraard maar met 1, maar het gebruik van bytes op de stack word afgeraden).

3.4 Elementen uit een array halen

Het speelveld is een array van 64 bytes, de inhoud van een vakje uitlezen gebeurd a.d.h. van de mov ?l, [arraynaam+locatie] instructie.
Voorbeeld:

```
beurtblack:
        mov bx, VAK
        mov al, [speelveld+bx]
        cmp al, 4
                je vakjepossible
                jmp bothcmp
```

3.5 Het plaatsen van elementen in array

Het wegschrijven van een waarde in een array gebeurd analoog met het uitlezen. Namelijk a.d.h. van de instructie mov [arraynaam+locatie], ?l instructie.

3.6 Compare problemen

Compare is vaak een groot struikelblok geweest in het proces van het project.

```
vergelijkvak:
                mov bl, [beurt]
                cmp bl, 1
                        je @F
                        jmp vergelijkzwart
```

In dit voorbeeld word een compare uitgevoerd op twee byte waarden, waarbij in sommige gevallen het programma niet meer loopte. Het gebruik van words in plaats van bytes bleek wel te werken:

```
vergelijkvak:
                mov bx, 0
```

```
mov bl, [beurt]
cmp bx, 1
        je @F
        jmp vergelijkzwart
```

3.7 Makkelijk fouten maken

Assembly is een taal waar je heel gemakkelijk een komma of een dubbelepunt fout kunt maken. Dus ipv:

```
mov ax, 4
```

met komma na ax, schrijf je per ongeluk:

```
mov ax 4
```

Dit lijkt een kleine fout, maar betekend wel dat heel je programma niet werkt en dit meteen terug geeft. Het oplossen is niet zo moeilijk want nmake geeft je precies terug waar de fouten zitten.

3.8 Code View

We hebben veel problemen gehad met de debugger die op Dosbox zit: Code View. In het begin kregen we niet alle functie calls te zien waardoor je nog niet wist wat er nu precies fout was. Na enige tijd kwamen we er achter dat je via F8 een oproep kan doen. Ook kwamen we er vrij laat pas achter dat je ook breakpoints kunt zetten, wat wel handig is als constant een functie aan het einde moet debuggen. Tot slot zijn we er achter gekomen dat Code View een heel duidelijk overzicht heeft van alle functies in assembler, wat ons tot een paar oplossingen heeft geleid. Misschien een goed idee om volgend jaar mee te geven aan de studenten.

3.9 Overzichtelijkheid

Omdat assembly zelf een hele on-overzichtelijke taal is waar je zonder uitleg niet veel van kunt maken was het samenwerken af en toe wel lastig. Daarom hebben we snel besloten om een paar regels op stellen voor overzichtelijkheid:

1. Voor de start van een procedure moet en vak waarin de naam staat, de uitleg en wat er eventueel in welke volgorde gepusht dient te worden.
2. Aan het eind van een procedure komt een lijn.
3. Bij een loop (na een label wordt er ingesprongen)

   Voorgegaan met een lijntje met de naam van de loop.

   Na de laatste jump, in dat deel van de code, sluit dit ook af met een korte lijn.

4. Na een compare wordt er ingesprongen op de volgende manieren:

   Een conditionele jump fungeert als true tak en wordt ingesprongen.

   De insprong blijft tot de volgende jump of label, die als false tak fungeert, behalve indien

   er een nieuwe compare plaats vindt.

5. Langs alle code, aan de rechterkant. Wordt er zo volledig mogelijk commentaar geplaatst op een verticale lijn.

4 Slotwoord