Universiteit Twente

Ontwerpproject

Canvas.hs

Auteurs:
Joost van Doorn
Lennart Buit
Pim Jager
Thijs Scheepers
Martijn Roo

Begeleider:
Robert de Groote

9 december 2013

Inhoudsopgave

1 Inleiding **2**

2 Ontwerp **3**
2.1 Requirements . . . 3
2.2 Globaal ontwerp . . . 4
2.3 Detail ontwerp . . . 4
2.3.1 Architectuur . . . 4
2.3.2 Externe libraries . . . 6
2.3.3 Grafische bibliotheek . . . 6
2.4 Testplan . . . 6
2.5 Testresultaten . . . 6

3 Conclusie **7**
3.1 Sectie . . . 7
3.1.1 Subsectie . . . 7

4 Evaluatie **8**
4.1 Sectie . . . 8
4.1.1 Subsectie . . . 8

A Gebruikerhandleiding **9**
A.1 Inleiding . . . 9
A.2 Primitieven . . . 10
A.2.1 Shapes . . . 10
A.2.2 Actions . . . 14
A.2.3 Events . . . 14

Hoofdstuk 1

Inleiding

Bij het informatica-keuzevak Functioneel Programmeren gebruiken studenten Haskell om fundamentele concepten van functioneel programmeren te bestuderen. Hierbij wordt door de studenten veel gebruik gemaakt van grafische weergaven om de werking van hun code inzichtelijk te maken. Het gebruik van Haskell voor het maken van grafische weergaven blijkt vaak redelijk gecompliceerd en limiteert studenten doordat zij zich bezig moeten houden met minder intuïtieve en minder essentiële aspecten van Haskell.

Om de focus binnen Functioneel Programmeren op de essentie te houden, is een simpele manier nodig om de Haskell code van de student om te zetten naar een grafische representatie. De interface tussen de code van de student en de grafische interface moet daarom eenvoudig en bruikbaar zijn zonder dat de ambitieuze student hierbij onnodig beperkt wordt.

In dit ontwerpproject is *Canvas.hs* ontwikkeld; een omgeving die Haskell-gebruikers in staat stelt op eenvoudige wijze grafische elementen op een HTML5 canvas te presenteren. Canvas.hs is ontwikkeld met het oog op gebruiksgemak en eenvoud zonder de mogelijkheid tot uitbreiding en het toevoegen van geavanceerde functionaliteit onnodig te beperken.

Verslagstructuur

In dit verslag wordt in Hoofdstuk 2 het ontwerp van Canvas.hs beschreven. Dit hoofdstuk is onderverdeeld in de secties: 2.1 Requirements, 2.2 Globaal ontwerp, 2.3 Detail ontwerp, 2.4 Testplan en 2.5 Testresultaten. Hoofdstuk 3 beschrijft de conclusies van dit project en in Hoofdstuk 4 wordt dit project met zijn uitkomsten geëvalueerd. Vervolgens wordt in Appendix A Gebruikerhandleiding een handleiding gepresenteerd die gebruikers van Canvas.hs aanwijzingen geeft over hoe de omgeving gebruikt en uitgebreid kan worden.

Hoofdstuk 2

Ontwerp

2.1 Requirements

R1 Het systeem dient events vanuit javascript door te geven aan het Haskell-programma van de student.

R2 In de javascript-omgeving zou een simpele debug-console moeten zijn.

R3 Toetsaanslagen vanuit het canvas zouden doorgegeven moeten worden aan de haskell-code van de student.

R4 Het systeem kan eventueel tekstinvoer vragen met een pop-up.

R5 Grafische primitieven zoals cirkels, vierkanten, lijnen, Bézier curves, n-hoeken en tekst moeten op een simpele manier getekend kunnen worden vanuit een Haskell-programma.

R6 Het moet mogelijk zijn de vul- en lijnkleur van grafische componenten in te stellen.

R7 Een gebruiker moet grafische componenten aan kunnen passen zonder zijn programma te hoeven hercompileren.

R8 Het systeem kan eventueel plaatjes inladen op het canvas.

R9 Het systeem kan eventueel gradients als vulkleur gebruiken.

R10 Er kan eventueel gezoomd en geschoven worden op de canvas.

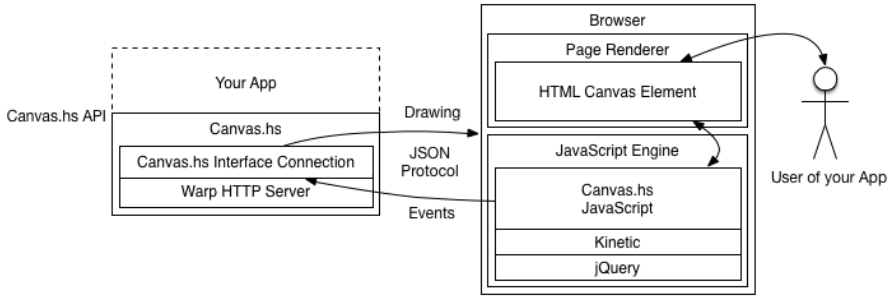
R11 Het systeem kan eventueel stapsgewijze aanpassingen geanimeerd weergeven.

R12 Het systeem dient duidelijke errors te genereren.

R13 Het systeem dient gemakkelijk en snel te gebruiken te zijn.

R14 De FPPrac-module zou automatisch een browser moeten kunnen starten.

R15 Het systeem zou de canvas opnieuw moeten laden als de Haskellcode opnieuw gecompileerd wordt.



Figuur 2.1: Overzicht van de architectuur van Canvas.hs

R16 Als de verbinding tussen javascript en Haskell verbroken wordt zou hiervan van melding gemaakt moeten kunnen worden aan de gebruiker.

R17 70% code coverage (exclusief frameworks) zou behaald moeten worden.

2.2 Globaal ontwerp

2.3 Detail ontwerp

2.3.1 Architectuur

Canvas.hs heeft een redelijk gecompliceerde architectuur. Dit komt vooral door de verschillende technologieën die nodig zijn om het HTML5 Canvas te verbinden met de te ontwikkelen Haskell API voor het bouwen van interfaces. In Figuur 2.1 is een schematische weergave van de architectuur weergegeven.

Haskell

Canvas.hs is een library die de programmeur kan importeren in zijn programma om er daarna met de API die Canvas.hs aanbiedt gemakkelijk een uitgebreide user interface mee te bouwen. Canvas.hs zal zich focussen op elementaire input en geen ondersteuning hebben voor high level interface elementen zoals buttons en textarea's. Deze elementen zouden met behulp van Canvas.hs wel eenvoudig te implementeren moeten zijn.

API De programmeur die gebruik maakt van Canvas.hs zal op een eenvoudige wijze gebruik kunnen maken van onze library. Daarom is het belangrijk dat er een API aangeboden wordt die eenvoudig te begrijpen is en niet teveel features bevat maar daarnaast wel de flexibiliteit biedt om een zeer complexe interface mee te bouwen.

Verbinding met de interface De verbinding met het canvas wordt bewerkstelligd met een eenvoudige HTTP server. Deze server biedt de gebruiker de mogelijkheid om via een webbrowser het Haskell programma te benaderen. De

HTTP server biedt pagina's aan waarin JavaScript en HTML samenwerken om op het canvas te tekenen.

Als via de API van Canvas.hs begonnen wordt met tekenen zal de HTTP server automatisch gestart worden. Het besturingssysteem wordt aangeroepen voor het openen van de standaard browser –verwijzende naar het adres van de lokaal draaiende HTTP server.

Protocol

De gegevens die verstuurd worden volgens het protocol zullen op een primaire gegevensstructuur moeten aanhouden. Hiervoor kunnen twee voor de hand liggende conventies gekozen worden: XML en JSON. Ons protocol zal gegevens coderen in JSON (JavaScript Object Notation). De voordelen van JSON zijn onder andere dat de data met weinig moeite direct gebruikt kan worden in JavaScript, lichtgewicht is en makkelijk te lezen is.

Websockets Gegevensoverdracht tussen de HTTP server en de browser moet snel gebeuren zonder een al te groote vertraging. Er zijn in de communicatie tussen een HTTP server en client verschillende mogelijkheden waarbij websockets de meestvoordehand liggende is. Websockets is een relatief nieuwe feature van HTTP en wordt op dit moment alleen ondersteund in de nieuwe browsers. Het gebruik van websockets zou als gevolg hebben dat Canvas.hs geen ondersteuning heeft voor oudere browsers.

Restful vs RPC Naast de manier waarop gegevens worden gecodeerd is het ook belangrijk via welke wegen deze aan de server worden aangeboden—alsmede hoe deze door de server worden verstrekt.

RESTful HTTP webservices hebben als voordeel dat deze de basis en de functionaliteit van het HTTP protocol optimaal benutten voor het verder defineren van een eigen protocol. RPC geeft de vrijheid om een volledig, zelf in te vullen, protocol te bouwen op het HTTP protocol. Gezien de werkbaarheid van WebSockets nog onderzocht moeten worden is het de vraag welke aanpak er uiteindelijk gebruikt zal gaan worden.

Voor AJAX requests middels HTTP fetching zou een RESTful webservice een goede optie zijn. Echter is bij WebSockets het HTTP protocol niet meer beschikbaar en zal er gebruik gemaakt moeten worden van een vorm van RPC.

Interface data De gegevens die door de HTTP server naar de browser gestuurd worden zullen voornamelijk bestaan uit interface data. Hoe, is de interface opgebouwd, elke elementen staan waar, welke attributen hebben deze elementen en zijn deze elementen in staat input van de gebruiker te accepteren. Hierbij zal er in het protocol rekening gehouden worden met de weergave van de elementen. De elementen die Kinetic.js ondersteund bieden hier een goede basis voor.

Events Events zijn de gegevens die de browser terug stuurt naar de HTTP server. Deze gegevens zullen vooral informatie bevatten over interface acties die de gebruiker uitvoerd. Het gaat hierbij om bijvoorbeeld muisbewegingen, muisklikken en toetsaanslagen. Het is belangrijk dat deze events snel door de

server worden ontvangen en verwerkt naar nieuwe output zodat de gebruiker geen zichtbare vertraging ziet in de werking van het programma.

Javascript

Voor het tekenen van de interface op het HTML Canvas alsmede de communicatie vanuit de browser met de server, zal er gebruik worden gemaakt van JavaScript. jQuery biedt een prima uitbreiding op JavaScript waarbij de mogelijkheid wordt geboden verschillende zogenaamde jQuery plugins te gebruiken. In het geval van Canvas.hs zijn twee plugins belangrijk: Kinetic.js en jQuery-websockets.

Kinetic.js zal gebruikt worden voor het tekenen van de verschillende elementen op het canvas. Daarbij heeft Kinetic.js in compinatie met jQuery ook uitstekende ondersteuning voor het doorgeven van input events, die door bijvoorbeeld muisbeweginen aangeroepen worden.

jQuery-websockets is een plugin die het gemakkelijk maakt gegevensoverdracht tussen websockets af te vangen en daar snel actie op te ondernemen. Deze lichtgewicht library maakt het mogelijk snel, zonder veel overhead opdrachten door te sturen naar Kinetic.js zodat deze op het canvas getekend worden.

Debug Console Voor de programmeur die gebruik gaat maken van de Canvas.hs library is het belangrijk dat zijn interface er zo uit ziet zoals hij dit wil. Ongetwijfeld zal een programmeur tegen problemen aanlopen bij het bouwen van de interface die hij niet had voorzien bij het schrijven van zijn code. Daarom vinden wij het belangrijk dat de interface van Canvas.hs een zogenaamde dubug console bevat waar het aanroepen van teken functies en de invloed van deze API aanroepen goed visueel en tekstueel inzichtelijk worden.

2.3.2 Externe libraries

2.3.3 Grafische bibliotheek

2.4 Testplan

2.5 Testresultaten

Hoofdstuk 3

Conclusie

3.1 Sectie

3.1.1 Subsectie

Tekst in subsectie.

Hoofdstuk 4

Evaluatie

4.1 Sectie

4.1.1 Subsectie

Tekst in subsectie.

Bijlage A

Gebruikerhandleiding

A.1 Inleiding

Canvas.hs is een haskell bibliotheek om op een simpele manier grafische elementen vanuit een haskell programma weer te geven en te reageren op input vanuit de gebruiker. Met behulp van een webbrowser, met javascript en canvaselement worden de primitieven getekend, en wordt op gebruikersinput gereageerd.

Canvas.hs maakt het mogelijk om interactie met de gebruiker, het bestandssysteem en andere elementen uit de zogenaamde ëchte wereld"zonder gebruik te hoeven maken van de IO Monad. Op deze manier is het voor beginnende haskell-programmeurs mogelijk om grafische programmas te schrijven zonder begrip te hebben van monads en monadische computaties.

Om dit alles te bereiken wordt gebruik gemaakt van event-driven-IO. De programmeur schrijft een handler-functie die elk event (bijvoorbeeld een klik van de gebruiker) afhandelt en daaruit een nieuwe output (bestaande uit een aantal te tekenen objecten en uit te voeren acties) oplevert. De bibliotheek maakt het daarnaast mogelijk voor de handler om een state bij te houden en deze bij elk event te lezen en aan te passen.

Kort gezegd maakt canvas.hs het volgende mogelijk:

- Grafische programmas schrijven in haskell
- Reageren op input, zoals muisklikken en toetsaanslagen
- Interactie met het bestandssysteem, door bijvoorbeeld bestanden te lezen of op te slaan.
- Meer, zoals het gebruik van klokken en het sturen en ontvangen van bestanden uit de browser.

Dit alles zonder gebruik te hoeven maken van monadisch programmeren.

In deze gebruikershandleiding zal worden toegelicht hoe met behulp van Canvas.hs een grafisch haskellprogramma geschreven kan worden. Er zullen een aantal simpele voorbeelden gegeven worden, vervolgens zullen alle mogelijke te tekenen vormen (`Shapes`'s), uit te voeren acties (`Action`'s) en te verwachten events (`Event`'s) uitgebreid worden toegelicht.

A.2 Primitieven

Nu we hebben gezien hoe we op verschillende `Event`'skunnen reageren, en hoe we op basis daarvan `Output` kunnen genereren in de vorm van `Shapes`'sen `Action` 'sis het tijd om een volledig overzicht te geven van alle mogelijke `Event`'s, `Shapes` 's, en `Action`'s.

Vergeet hierbij niet, zoals in het vorige hoofdstuk besproken, de types die belangrijk zijn voor canvas.hs:

```
1  -- Je functie die inkomende events afhandelt, State is
       hierbij een zelf te definiëren (data)type.
2  handler :: State -> Event -> (State, Output)
3
4  -- zie voor de verschillende Events hieronder
5  data Event = ...
6
7  data Output = Block BlockingAction | Out RegularOutput
8
9  --zie voor de verschillende BlockingActions hieronder
10 data BlockingAction = ...
11
12 -- vergeet niet dat (Nothing, _) er voor zal zorgen dat
       er niets wordt getekend
13 -- en (_, []) dat er geen acties zullen worden uitgevoerd
14 data RegularOutput = (Maybe Shape, [Action])
15
16 -- zie voor de verschillende Shapes hieronder
17 data Shape = ...
18
19 --zie voor de verschillende Actions hieronder
20 data Action = ...
```

A.2.1 Shapes

Zoals in het vorige hoofdstuk aangegeven worden shapes opgebouwd door te beginnen met een basis-shape en hier dan translaties, rotaties, randen en andere translaties op toe te passen, daarnaast kan ook nog worden aangegeven dat een shape luisterd naar Events.

Primitieven

Voor de primitieve shapes worden een tweetal types veel gebruikt. Dit zijn `Point` en `Path`.

Point Een `Point` definieert een punt op het canvas als een tuple van een x en een y coördinaat. `type Point = (Int, Int)`

Path Een `Path` definieert een pad, als een lijst van punten. `type Path = [Point]`.

Rect
`Rect` staat voor Rectangle en definieert een rechthoek.

```
1 data Shape = ..
2              | Rect Point Int Int
```

- `Point`, de linkerbovenhoek van de rechthoek
- `Int`, de breedte van de rechthoek
- `Int`, de hoogte van de rechthoek

Circle
`Circle` definieert een cirkel met een bepaald middelpunt en straal

```
1 data Shape = ...
2              | Cicle Point Int
```

- `Point`,het middenpunt van de cirkel
- `Int`, de straal van de cirkel

Line
`Line` definieert een lijn over een `Path`, dit pad wordt niet gesloten.

```
1 data Shape = ...
2              | Line Path
```

- `Path`, het pad waarover de lijn getrokken moet worden

Polygon
`Polygon` definieert een polygoon over een `Path`, het eindpunt zal aan het beginpunt worden gekoppeld waardoor een gesloten figuur ontstaat.

```
1 data Shape = ...
2              | Polygon Path
```

- `Path`, het pad waarlangs de randen van de polygoon lopen.

Text

Het is ook mogelijk om tekst weer te geven met de `Text` Shape.

```
1 data Shape = ...
2              | Text Point String TextData
```

- `Point`, het punt waar rond de text getekend zal worden, m.b.v. `TextData`, kan dit verandert worden van de linkerbovenhoek, gecentreerd of de rechter bovenhoek.
- `String`, de te tekenen tekst
- `TextData`, een aantal opties om tekst anders weer te geven, zoals lettertype en lettergrootte, zie hieronder.

TextData
`TextData` is een record om een aantal opties mee te kunnen geven bij het tekenen van `Text`. Het is een instantie van `Defaults`.

```
1  type FontSize = Int
2
3  data Alignment = Start | End | Center
4
5  data TextData = TextData {
6      font :: String,
7      size :: FontSize,
8      bold :: Bool,
9      italic :: Bool,
10     underline :: Bool,
11     alignment :: Alignment
12 } deriving (Eq, Show)
13
14 instance Defaults TextData where
15     defaults = TextData "Arial" 12 False False False
       Center
```

- font, het font van de te tekenen tekst. Dit font moet door de browser ondersteunt worden, als dit niet zo is zal de browser terugvallen op het standaard font
- size, de grootte van de te tekenen tekst
- bold, of de tekst dikgedrukt getekend moet worden
- italic, of de tekst schuingedrukt getekend moet worden
- underline, of de tekst onderstreept getekent moet worden
- alignment, de uitlijning van de te tekenen tekst.
  - `Left`, de linkerbovenhoek van de tekst wordt op het punt uitgelijnd.
  - `Center`, het middenpunt van de tekst wordt op het punt uigelijnd.
  - `Right`, de rechterbovenhoek van de tekst wordt op het punt uitegelijnd.

Translaties

De getekende primitieven (waaronder `Text`) kunnen d.m.v. translaties aangepast worden. Zo kunnen ze bijvoorbeeld gekleurd, van een rand voorzien, of geroteerd worden. Deze translaties zijn zelf ook `Shapes`'s, hierdoor is het mogelijk om meerdere translaties op elkaar uit te voeren.

Color Color definieert een kleur. De kleur wordt gedefinieert door een rood-, groen- en blauwwaarde varrieërend van 0 tot 255, daarnaast is er een alphawaarde varieërend van 0 tot 1.0. `type Color = (Int, Int, Int, Float )`

Fill
`Fill` definieert dat een `Shape`een fill van een bepaalde kleur moet krijgen.

```
1 data Shape = ...
2              | Fill Color Shape
```

- `Color`, de kleur waarmee de `Shape`gevuld moet worden
- `Shape`, de te kleuren `Shape`

Stroke
`Stroke` definieert dat een `Shape`voorzien moet worden van een rand van een bepaalde kleur en dikte.

```
1 data Shape = ...
2              | Stroke Color Int Shape
```

- `Color`, de kleur van de rand
- `Int`, de dikte van de rand
- `Shape`, de `Shape`die van een rand moet worden voorzien

Rotate
`Rotate` definieert dat een `Shape`een aantal graden tegen de klok in geroteerd moet worden rond zijn linkerbovenhoek. Van niet rechthoekige `Shapes`'swordt de rechterbovenhoek van de rechthoek die de `Shape`insluit gekozen. M.b.v. `Offset` kan dit rotatiepunt verandert worden, zie hieronder.

```
1 data Shape = ...
2              | Rotate Int Shape
```

- `Int`, de rotatie in graden (tegen de klok in)
- `Shape`, de `Shape`om te roteren

Scale
`Scale` definieert dat een `Shape`in de x en y richting geschaald moet worden. M.b.v. `Offset` kan het referentiepunt voor dit schalen verandert worden (zie hieronder).

```
1 data Shape = ...
2              | Scale Float Float Shape
```

- `Float`, de schaal in de x-richting
- `Float`, de schaal in de y-richting
- `Shape`, de te schalen `Shape`

Offset
`Offset` definieert een ander referentiepunt voor `Rotate` en `Scale`.

```
1 data Shape = ...
2              | Offset Point Shape
```

- `Point`, het punt wat als referentiepunt zal dienen
- `Shape`, de `Shape`om het referentiepunt van te veranderen

Translate
`Translate` definieert dat een `Shape`in de x en y richting verplaatst moet worden.

```
1 data Shape = ...
2              | Translate Int Int Shape
```

- `Int`, de verplaatsing in de x-richting
- `Int`, de verplaatsing in de y-richting,
- `Shape`, de `Shape`om te verplaatsen.

Luisteren naar Events

Containers

`Shapes`'skunnen worden samengebracht in `Container`'s. Deze zijn zelf ook weer een `Shape`zodat ze op hun beurt weer kunnen worden samengebracht, er translaties op kunnen worden uitgevoerd en kan worden aangegeven dat ze geïnteresserd zijn in `Event`'s.

```
1 data Shape = ...
2              | Container Int Int [Shape]
```

- `Int`, de breedte van de `Contrainer`
- `Int`, de hoogte van de `Contrainer`
- `[Shape]`, de `Shapes`'sdie in deze container zitten

Translaties In het geval van translaties worden deze altijd op de hele `Container` toegepast. Dit betekent het volgende:

- `Fill`, alle `Shapes`'sin de `Container` worden gekleurd
- `Stroke`, alle `Shapes`'sin de `Container` worden van een rand voorzien
- `Rotate`, de `Container` wordt in z'n geheel gedraaid
- `Scale`, de `Container` wordt in z'n geheel geschaald
- `Translate`, de `Container` wordt in z'n geheel verplaatst

A.2.2 Actions

A.2.3 Events