KF04 projekt rapport

Gruppe 12:
Jakob Melnyk
Niklas Hansen
Emil Juul Jacobsen
Filip Hjermind Jensen
Jens Dahl Møllerhøj

April 5, 2011

Indhold

1 Features

Dette kapitel beskriver de features, vi har valgt at implementere i vores Map Of Denmark projekt i foråret 2011, og hvad vi har valgt ikke at implementere.

- Mulighed for zoom med musen
- Navigationsknapper til kortet på GUI
- Zoom in og ud funktioner på GUI
- Vejnavne vises, når man kører musen over dem
- Gør kortet større, når man hiver i kanterne
- Gå til originalt zoom, når man trykker Escape
- Viser færre veje, når man er zoomet langt ud

1.1 Mousezoom

Det er muligt at zoome ind på en specifik del af kortet ved at klikke med musen, trække en "kasse" hen over det, man vil zoome ind på, og slippe museknappen.

Billedet vil altid justere sig selv i forhold til applikationsvinduets størrelse. Dette betyder, at kortet alid vil vise det, som blev valgt, men nogle gange vil det vise mere i enten højden eller i bredden.

1.2 Navigationsknapper

Det er muligt at navigere rundt på kortet ved hjælp af knapperne på GUI. Når der klikkes på knapperne, bevæger vinduet sig i den retning vist af pilen på knappen.

1.3 Zoom in/out funktioner

Det er muligt at zoome ind og ud when hjælp af + og - knapperne på GUI. Kortet vil zoome i midten af det, som ses i øjeblikket.

1.4 Vejnavne vises, når man kører musen over dem

Når man kører musen hen over en vej, vises navnet på vejen i bunden af applikationen.

1.5 Kortets kanter kan trækkes

Når man trækker i kanten af kortet, bliver kortet mindre eller større, an på hvilken vej vinduet blev trukket.

1.6 Gå tilbage til original zoom

Det er muligt at sætte zoomniveauet tilbage til det originale. Dette gøres ved, at man trykker på Escape-tasten. Dette gør det nemmere at navigere på kortet, hvis man har zoomet langt ind.

1.7 Vis færre veje, når man er zoomet ud

Jo længere brugeren har zoomet ind på kortet, jo flere veje vises. En lav detaljegrad er ikke noget problem, når man er zoomet langt ud, og dette hjælper med at gåre kortet hurtigere.

1.8 Features, som ikke er med

Den eneste måde, at bevæge sig rundt på kortet, er ved at anvende knapperne på GUI. Vi overvejede at implementere både piltasterne og musen som navigationsmulighed. Vi har gemt det væk for nu, da vi følte, at det var en "nice-to-have" feature mere end en nødvendighed.

Vi har valgt ikke at gøre det muligt at specificere, hvilke typer veje, man vil have vist. Vi føler ikke, det er nødvendigt, da vi kun viser de store veje, når vi er zoomet ud.

2 Implementation

Dette kapitel beskriver de valg, vi har foretaget, mens vi implementerede vores Map Of Denmark projekt i foråret 2011.

2.1 Model-View-Control

For at få en ordentlig struktur på sin kode, er det vigtigt at opdele sin kode i flere dele, som arbejder sammen for at få programmet til at køre. En måde, man kan gøre dette på, er ved at lave en klasse til sin brugergrænseflade, og en klasse til resten.

En af ulemperne ved at gøre det på denne måde er, at det kan blive tvetydigt, hvor dele af ansvaret for kommunikationen mellem disse skal ligge.

Vi har valgt at lægge os op af Model-View-Controller (også kaldet MVC) arkitekturen, som er en anden måde at gøre det på. Her deler vi koden op i tre dele, for at få en fornuftig opdeling af data, logik og brugergrænseflade.

Illustation af MVC kan ses på side 9.

Ligesom ovenstående billede viser, så har et grafisk vindue sin egen klasse. Sådanna klasser bliver kaldt "views". Hvor man førhen havde en klasse til både at kommunikere med de grafiske vinduer og datakilden (fx en database, eller en CSV fil i vores tilfælde), så splitter man nu denne del op i to. Den ene del kalder vi for "models" - eller på dansk: modeller.

Disse modeller står for kommunikationen med datakilden, og hver model repræsenterer en enkelt datakilde. Hvis man brugte en relationsdatabase, ville man have en model til hver tabel i databasen. Denne model-klasse står så for al kommunikation med lige præcis den ene tabel, som den repræsenterer.

Så mangler vi bare en måde at forbinde den grafiske brugergrænseflade ("views") med vores data ("models"), og det er her "controller-delen kommer ind i billedet. Controlleren står som et mellemled, og henter data fra modellen. Disse data giver den så videre til det grafiske vindue. Controlleren har desuden nogle "lyttere", så den lytter til, om brugere trykker på en knap eller at lignende begivenheder finder sted. Hvis der fx bliver opdateret noget data i det grafiske vindue, står controlleren for at sende den nye data videre til modellen.

En af fordelene ved at dele koden op på denne måde er, at man har en fast struktur, som gør det nemt at overskue koden, og man deler ansvaret ud på forskellige klasser, så man dermed kan udnytte fordelene ved abstraktion og modularisering.

2.2 Map

Map klassen står for den del af kortet, der vises på skærmen. Den har feltet *bounds*, der er det rektangel af kortet, der vises i øjeblikket. Når **Map**-klassen skabes, beregner den *bounds* til det mindste rektangel, der viser hele kortet.

Når **View** skal tegne kortet, skaber **Map** en række objekter af klassen **Line** ud fra de **KrakEdges**, som skal vises. **Map** sørger for, at objekter af klassen **Line** har den rigtige farve og tykkelse.

2.3 QuadTree og valg af veje

Vi har valgt at anvende et **Quadtree** i vores kort. Man kan give et **QuadTree** et rektangel, hvorefter den så returnerer et **Set** af de veje, som ligger indenfor det rektangel. Dette betyder, at vi undgår at løbe alle veje igennem, når vi er zoomet langt ind.

Det er ikke interessant at få vist alle veje, når man kigger på hele Danmark. Derfor har vi valgt ikke at tegne de små veje, når man er zoomet langt ud. Dette gør kortet meget hurtigere at navigere rundt på.

Implementationen af denne optimering hænger i høj grad sammen med implementationen af **QuadTree**. Vi overvejede to måder at forbedre vores oprindelige struktur på.

Den ene måde gik ud på at dele **QuadTree** op i enten flere rødder (med hver deres vejtype) eller lave flere **QuadTree**'er. Denne løsning ville betyde, at man skal søge færre objekter igennem, når man skal tegne kortet.

Den anden måde indebar at fordele vejtyperne på forskellige dybder i træet. **QuadTree** skal så vide hvilket zoomniveau, der skal vises. Det ville så kun returnere de veje, som lå ned til den højde af træet, som zoomniveauet havde bestemt.

De to måder har hver deres fordele og ulemper.

Den første vil kræve flere **QuadTreeNode** objekter, hvilket vil øge RAM forbruget. Til gengæld vil alle vejene i denne metode blive lagt i træets blade, hvilket er en fordel, når der skal søges i træet.

Den anden er svær at implementere korrekt, da der skal defineres nogle meget specifikke mængder af veje, hver **QuadTreeNode** kan indeholde.

Vi har besluttet at vise en stor del af vejene, selv når vi er i yderste zoomniveau. Det har vi besluttet, fordi vi ikke har nogen features, som kræver, at tingene sker med det samme. Dette betyder også, at der er bedre mulighed for at genkende området, man ser på.

2.4 Control

Control står for det meste af logikken i programmet. **Control** står for at bede **Map** om at ændre sig, når der sker noget i de listeners, som **Control** har i **View**. Den står også for at konvertere Pixel til UTM og lave rektanglerne med de rigtige værdier til **Map**. Den har også styr på værdierne for, hvor meget der skal zoomes og bevæges, når man anvender knapperne i GUI'et.

2.5 PixelToUTM

Vi har lavet en metode i Control, som tager et **Point** objekt med pixelkoordinater og returnerer et **Point2D.Double** objekt med UTM-koordinater. Først vendes y-koordinaten "på hovedet", så den passer ind i UTM-koordinatsystemet. Derefter regner metoden ud, hvor langt inde på skærmen, der er klikket, finder ud af, hvor langt dette er i UTM-koordinater, og lægger det oveni feltet *bounds* koordinater.

Vores formel er følgende:

$\mathrm{UTM}_x = bounds_x + (\mathrm{a}_x \ / \ canvas_{width}) * bounds_{width}$

$\mathrm{UTM}_y = bounds_y + (\mathrm{a}_y \ / \ canvas_{height}) * bounds_{height}$

Se Figure 1 på side 8 for en illustration af konverteringen. ¡¡¡¡¡¡¡ HEAD

2.6 Line

Line klassen benyttes i forbindelse med at KrakEdge objekterne skal have et start- og et slutpunkt som er relative til skærmen. Derudover indeholder Line klassen information omkring tykkelse og farve af den pågældende vej. Tykkelsen og farven kunne vi vælge at specificere i enten Map klassen eller i selve Line klassen. Vi valgte at gøre det i Map for at spare RAM. =======

2.7 Lines

Line klassen benyttes i forbindelse med, at **KrakEdge** objekterne skal have et start- og et slutpunkt, som er relative til skærmen. Derudover indeholder **Line** klassen information omkring tykkelse og farve på den pågældende vej.

Tykkelsen og farven kunne vi vælge at specificere i enten Map **klassen** eller i selve **Line** klassen. Vi valgte at gøre det i **Map** for at nedsætte RAM-forbruget. ¿¿¿¿¿¿¿ 75244c6f8c67c43038cc86d1b07cb9a61f59fe82

2.8 View

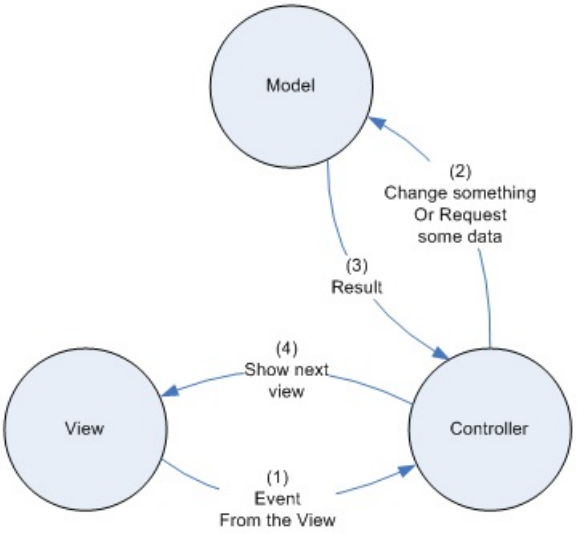
View er den klasse, som står for den grafiske repræsentation af vores data. Det er denne klasse, som sørger for opbygningen af vinduet med alt, hvad det indebærer af knapper til navigering på kortet, tekst-bar til visning af den vej, musen pejer på, samt selve kortet.

Selve kortet (**Canvas**) er implementeret som en privat indre klasse, der udvider Javas **Component**-klasse. **Canvas** har mulighed for at tegne vores **Line** objekter, og kan derved tegne kortet uden nogen viden om den bagvedliggende datastruktur.

Det er muligt for brugeren at trække et rektangel i vinduet med musen for at zoome ind på en bestemt del af kortet. For at undgå at skulle gentegne hele kortet, hver gang brugeren trækker rektanglet lidt større, tegner **Canvas** kortet på et off-screen **BufferedImage**, og bruger derefter dette billede til at tegne sammen med rektanglet. På den måde kan vinduet gentegnes på konstant tid i denne situation.

Eftersom **View** ikke har til ansvar at fortolke brugerens input, er der en lang række metoder til at tilføje Listeners til de forskellige events, der kommer, når brugeren anvender musen, ændrer på vinduet's størrelse eller trykker på tastaturet. Desuden er der naturligvis mulighed for at sende nye **Line**s ind, som derefter opdaterer **Canvas**.

Figur 1: Illustration af konverteringen fra Pixel til UTM. Pixel koordinater vender på hovedet i forhold til UTM. Derfor skal de vendes om.



Figur 2: Illustration af MVC-designpattern