Projektarbejde, Snake

Anna Ølgaard Nielsen - s144437
Christian Søholm Andersen - s103080
Mathias Enggrob Boon - s144484
Van Anh Tri Trinh - s144449

18. januar, 2015

Indhold

Kapitel 1

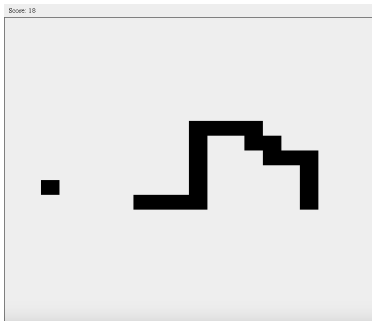
Introduktion

Formålet med projektopgaven er at genskabe det klassiske spil **Snake**, samt at dokumentere hvordan spillet er lavet. Spillet er gengivet i to versioner: **Simpel Snake** og **Avanceret Snake**.

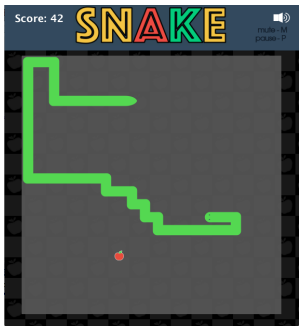
Simpel Snake er en primitiv version af spillet, hvor styring og bevægelse kun foregår vha. input fra spilleren (se figur 1.1a).

I Avanceret Snake er der tilføjet forskellige funktioner, som enten forbedrer brugerfladen - f.eks. tilføjelse af hovedmenu - eller ændrer spillet - f.eks. automatisk bevægelse af slangen (se figur 1.1b).

I rapporten vil designet af begge spillets versioner blive forklaret, samt implementationen af spillet. Kapitlet "Udviklingsproces"forklarer de tanker, der ligger bag implementationen, og de valg der er foretaget, når der er flere mulige implementationer.



(a) Simpel Snake



(b) Advanced Snake

Figur 1.1

Kapitel 2

Simpel Snake

2.1 Afgrænsning

I Simpel Snake består slangen af to felter (et hoved og en hale), der er placeret i banens centrum, rettet mod venstre. Det er muligt gennem input fra tastaturet at bevæge sig frit på banen i de fire retninger i et almindeligt koordinatsytem: op og ned ad y-aksen og hen ad x-aksen i begge retninger - dog ikke modsatrettet slangens bevægelsesretning. Bevæger man sig ud over banens kanter, skal man kunne fortsætte på den modsatte side. Det skal derudover være muligt at øge slangens længde ved at bevæge sig over et felt med et "æble". Spillet slutter, når spilleren har fyldt hele banen ud med slangens krop og dermed har vundet spillet, eller ved at spilleren bevæger sig ind i et felt udfyldt af slangen, der ikke er halestykket, som ved slangens bevægelse også rykker videre til et nyt felt. I så fald, har spilleren tabt spillet.

2.2 Design

Snake-spillet er lavet efter et *Model-View-Controller*-design (MVC), hvorved selve spillet, styringen af spillet og den visuelle repræsentation af spillet holdes adskilt i tre dele. På denne måde interagerer brugeren kun med den del af programmet, der er dedikeret til styring. Styringen manipulerer programmets tilstand i *model*-koden, som visualiseres i *view*-koden. Dette betyder derfor også, at alle funktioner, der påvirker programmets tilstand, skal holdes i *model*-koden. *Control* modtager kun input fra brugeren, og sender dette videre til *View* og/eller *Model*. *View* modtager input fra *Control* og sender ændringerne i *View* videre til *Model*. Det er derefter muligt gennem en observer at "observere"ændringerne i *Game*, som bliver opdateret i *View*.

Simpel Snake er designet, så selve spillets funktioner ligger i klassen *Game* i model-pakken. *Game* får da spillets objekter fra andre klasser i *model*-pakken, f.eks. Board (spillebrættet) og Snake (slangen). Spillets tilstand ændres, når der modtages input fra *control*-klasserne, som bestemmer hvornår og hvordan slan-

gen bevæger sig. I *ViewBoardSingleplayer*-klassen er en observer, som notificeres hver gang banens tilstand ændres. Når dette sker, eller når spillet startes, tegnes banen vha. klasserne *View* og *BoardPanel*. Disse klasser modtager information fra *model*-klasserne, f.eks. *Snake*, til at bestemme hvordan spillet gentegnes. *BoardPanel* gentegner hele spillet hver gang spillet opdateres. Først tegnes selve banen, derefter slangen og til sidst æblet. For at starte spillet bruges klassen *Driver*, som opretter et nyt *Game*-, *View* og *Control*-objekt.

2.3 Implementering

2.3.1 Control (Styring)

I Simpel Snake bruges kun fire taster til input, nemlig de fire piletaster, som derfor er defineret i *Control*.*Control*-klassen har metoden *keyPressed*, som kommer af at *Control*-klassen implementerer *KeyListener*, som kaldes hver gang der tastes på tastaturet. Hvis tasten er en af de fire piletaster, kaldes funktionen *move* i *Snake*-klassen, der flytter hovedet i samme retning som piletasten. Enum-klassen *Direction* indholder *RIGHT, LEFT, UP* og *DOWN*. Dette bruges i *Control*-klassen, da det ikke skal være muligt at bevæge sig i modsatte retning af slangens nuværende retning. I *Control*-klassens constructor er Direction som udgangspunkt sat til *LEFT*, da slangen starter mod venstre. Hver gang der tastes på piletasterne, sikrer *keyPressed*-metoden, at retningen ikke er modsat af den nuværende retning. Herefter flyttes slangen, uanset om den er midt på spillepladen eller om den bevæger sig rundt om torussen. Til sidst sættes *Direction* til den nuværende retning, så det ved næste tasteanslag igen kan undersøges, at retningen ikke er modsat.

2.3.2 Model

Spillets model består af en række klasser, der tilsammen udgør selve spillets funktioner og objekter. Klassen *Field* bruges til at definere et objekt, som opdeler banen og repræsenterer et felt i banen. Den fungerer på samme måde som *Point*-objekt-klassen, hvor koordinatsystemet starter oppe i venstre hjørne. Forskellenl mellem *Point*- og *Field*-klassen, er hvordan man kommer frem til et punkt. *Point*-klassen går først ud ad x-aksen og derefter ned ad y-aksen, hvorimod *Field*-klassen gør det i omvendt rækkefølge. Da hele spillepladen er delt op i rækker og kolonner, frem for en x- og y-akse, er *Field* lettere at bruge for at undgå forvirring med, hvordan to-dimensionelle arrays opdeler rækkerne og kolonnerne.

Funktionenen *equals* er defineret i denne klasse, og bruges til at undersøge om to objekter ligger på samme felt, f.eks. slangens hoved og æblet. Æblet bliver oprettet i klassen *Game*, hvor datafeltet *position* afgør dets nuværende position.

Slangen selv er defineret i klassen *Snake*. Slangens krop består af en række felter, hvoraf det første er hovedet, og de resterende er kroppen (inklusiv halen til sidst). Koordinaterne for disse felter er gemt som elementer i en ArrayList

kaldet *positions*. Det første element er slangens første led, hovedet, andet element er slangens andet led osv. Når et nyt led tilføjes, tilføjes et nyt element til listen. Dette sker, når slangen spiser et æble, hvor den ikke bevæger sig som ellers. I stedet tilføjes et nyt hoved på æblets felt. Hovedets retning er den samme, som slangens sidste bevægelse.

I *Snake*-constructoren bliver slangens hovede og hale oprettet, hvor hovedet bliver placeret i banens centrum. Halen placeres i næste kolonne på samme række.

I *move*-metoden benyttesi Action-klassen, der er oprettet som en enum, og betegner om slangen spiser et æble, er død eller bevæger sig (hhv. *EAT, KILL, MOVE*). I *move*-metoden undersøger den, om der er mad på samme felt som hovedets nye placering. Er dette tilfældet, bliver Action-statementet til *EAT*. Hvis ikke, undersøges det om den nye placering for hovedet allerede indeholder slangens krop. Dette gøres ved at bruge en contains-metode på *positions*, med den nye placering som argument. Returnerer funktionen "true", bliver Action-statementet til *KILL*. Hvis ikke, flyttes slangens hoved til den nye position. Resten af slangens krop følger med ved at ændre koordinaterne i *positions* fra halen og op til hovedet, dvs. at halens koordinater bliver næstsidste led, næstsidste bliver tredjesidste osv. Slangens krop flyttes først, hvorefter hovedet flyttes, for at undgå at andet led indtager samme plads som hovedet.

I *Game*-klassen findes *checkAction*-metoden, der undersøger om slangen spiser eller dør. Hvis Action-statementet er *EAT*, øges slangens længde ved at tilføje et nyt element på æblets felt, som fungere som nyt hoved. Scoren inkrementeres, der laves et nyt æble, og Action-statementet sættes til *MOVE*. Hvis Action-statementet er *KILL*, nulstilles scoren, og spillet slutter. Herudover findes *generateFood*-metoden, som sikrer, at maden altid placeres på et gyldigt felt, dvs. et felt der ikke er udfyldt af slangen. For at sikre dette, placeres æblet på et tilfældigt felt, hvorefter det sammenlignes om et af slangens led har samme koordinater som feltet. Er dette tilfældet, gives æblet et nyt felt, indtil det lander på et felt uden slangen. Er slangen tilpas stor, er dette dog ikke effektivt. Af denne årsag undersøger metoden først, om slangen fylder mere end halvdelen af banen. Er dette tilfældet, laves der i stedet en liste med alle tomme felter vha. en løkke og en indlejret løkke, der løber gennem alle række og kolonner, hvorefter et tilfældigt element i listen .

De fire klasser *Food, Score, Game* og *Snake* forlænger Observable, så det er muligt for View at modtage ændringerne i Model, uden at Model afhænger af View.

2.3.3 View (Brugergrænseflade)

Brugerfladen er samlet i klassen *GameView* der forlænger *JFrame*. I *GameView* findes kun en constructor, hvor der oprettes et score-panel, der placeres øverst i vinduet, og et board-panel, der placeres direkte under det. Board-panelet viser selve banen med slangen og æblet, der begge er vist ved farvede firkanter.

ScorePanel-klassen forlænger *JPanel* og implementer *Observer*. Klassen har en *update*-metode, der gentegner, hver gang noget ændrer sig i de klasser i

Model, der forlænger *Observable*. *paintComponent*-metoden benyttes til at lave selve score-teksten. Selve panelet oprettes i constructoren.

BoardPanel forlænger også *JPanel* og implementerer *Observer*, ligesom klassen også har en *update*-metode. *BoardPanel* består af en række draw-metoder, samt en *paintComponent*-metode, der tegner alt på selve banen. I *drawSnake*-metoden bruges positionerne for slangens felter til at tegne slangen. Størrelsen for et felt udregnes i *getWindowRectangle*-metoden, og afhænger af vinduets og banens størrelse. Når feltets størrelse er udregnet, tegnes en rektangel med feltets størrelse på alle felterne i slangens positions-ArrayList.

2.4 Udviklingsproces

2.4.1 Arbejdsproces

Formålet med projektet var at starte med at lave en simpel udgave af snake, og derefter tilføje flere funktioner til at lave en mere avanceret version. Dette gør det ideelt at tilføje en funktion af gangen, frem for at planlægge alle funktioner på en gang, og tilføje dem samtidig. Resultatet bliver et, til at starte med, simpelt men fungerende program, hvorefter yderligere funktioner kan tilføjes. Programmet er altså udviklet iterativt, hvormed der opstår flere fungerende versioner af spillet, men med forskellige funktioner. Dette gør det muligt at tilpasse programmet, hvis der opstår nye idéer eller krav undervejs.

Den iterative tilgang gør det muligt at have en "cyklus"for udviklingen af programmet. Først bestemmes det, hvad der skal tilføjes til programmet. Derefter fordeles opgaverne blandt gruppens medlemmer. Gruppemedlemmet afgør selv, hvordan en funktion skal designes og implementeres, men sikrer at implementation er kompatibel med alle nuværende funktioner, og ikke vil hindre fremtidige tilføjelser i at blive tilføjet. Eventuelle justeringer til programmet laves for at undgå fejl med nye funktioner, hvorefter "cyklussen"starter forfra ved idéfasen.

2.4.2 Control

Idet der ikke er brug for nogen menu eller lign. i simpel-snake, var det muligt at holde styringen af spillet meget simpel. Ved at definere de fire retningstaster i Control-klassen, samt en funktion til at sikre at der ikke bevæges i modsat retning, er alle styringkrav til simpel-snake opfyldt. Resten af spillet håndteres da i model-koden.

2.4.3 Model

Opbygning af banen

Til designet af selve banen, som slangen bevæger sig på, forelås to muligheder. Den ene er at lave et to-dimensionelt array af datatypen enum, hvis størrelse afgør banens endelige størrelse. Et array [10][5] vil f.eks. give en bane med

længden 10 og bredden 5. Hvert element i arrayet bestemmer da, hvad der befiner sig på netop denne plads på banen. Elementerne i arrayet kan f.eks. være et blankt felt, et æble, et led af slangen osv. Dette gør det nemt at introducere nye spilelementer i fremtiden, f.eks. bonus-point, vægge og miner, idet der blot skal tilføjes nye værdityper. Visualisering af spillet foregår ved at definere et billede for hvert spilelement, og få programmet til at tegne objektet på arrayets plads.

En anden metode er, at lade de forskellige spilelementer være defineret i deres egne klasser, så f.eks. *SnakeFood* er en klasse for sig selv, *SnakePlayer* er en klasse for sig selv osv. Hver klasse har de funktioner, der er relevante for dem, f.eks. *getPosition* for at give deres nuværende position. Programmet tegner da spillet hver gang en tur afsluttes, dvs. når alle elementer som skal ændres, er ændret. Programmet har fået defineret billedet for de forskellige elementer, og modtager deres position vha. en *getPosition* metode. Ulempen ved denne metode er, at det skal laves flere metode og klasser, hvormed programmets kompleksitet stiger. I den første metode er alle felter allerede defineret, og for at ændre dem behøves værdien for hvert felt blot at blive ændret, så f.eks. et blankt felt ændres til æble. Ønskes et spilelement ændret eller introduceret med den anden metode, skal der laves et nyt objekt. Ønsker man at introducere nye elementer, kan man med første metode blot tilføje en ny værdi-type til det todimensionelle array. Med anden metode skal der laves nye klasser, metoder osv. for at introducere nye elementer til spillet.

I sidste ende blev metode 2 valgt, idet den gjorde det nemmere at holde Model-koden adskilt fra View- og Control-koden. Herudover har første metode den ulempe, at den skal sende et todimensionelt array på banens størrelse, hver gang banen skal tegnes. Ved en bane på f.eks. 100x100 betyder dette, at at sendes i alt 10000 elementer til View-koden, hver gang banen skal tegnes. Den anden metode sender derimod kun de relevante elementer, dvs. æblets og slangens position, til View-koden.

Snake

Da slangen i snake-spillet består af en række felter, som alle har netop en koordinat i forbindelse med andre led, er en effektiv måde at bestemme slangens position på en ArrayList, idet denne datastrukturer er fleksibel i størrelse og passer til formålet. Når slangen vokser i størrelse, bliver dette dog mindre effektivt, idet slangens hoved altid sættes som element 0. Når slangen vokser, tilføjes et nyt element på plads 0, hvormed hele listen skal flyttes. En anden mulighed ville være at bruge en anden datastruktur, f.eks. LinkedList, eller lade hovedet være defineret som element positions.size()-1, hvormed nye hovedet tilføjes sidst i listen. Et problem ved at bruge LinkedList er dog, at denne datastruktur ikke tillader vilkårlig adgang af værdier, men derimod altid bevæger sig fra første eller sidste element. Dette problem kan løses ved at bruge en iterator, men blev fravalgt, idet det skabte problemer, når listen skulle bruges på tværs af klasserne.

Til implementationen af scoren blev to løsninger foreslået. Enten at lade

scoren være et datafelt i game-klassen, eller at lade det være en klasse for sig selv. Ved at lade scoren være et datafelt, bliver implementationen simplere. At lade scoren være en klasse for sig selv har derimod fordelen, at der kan tilføjes en observer til Score-klassen, som dermed kun opdateres, når scoren ændrer sig. Scorepanelet tegnes ikke i samme klasse som banen, og kan derfor holdes seperat, så scorepanelet kun gentegnes, når scoren ændrer sig. Hvis scoren derimod er et datafelt, tegnes scoren efter hver tur, også selvom scoren er uændret. I sidste ende blev det bestemt at holde scoren som datafelt, hvormed fremtidige score-relaterede funktioner bliver simplere at implementere.

2.4.4 Brugerflade og visualisering af programmet

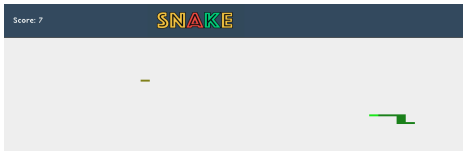
Tegning af spillet

For at visualisere spillet er det nødvendigt at kunne optegne banen, slangen og æblet. Dette gøres i klassen ViewBoard, som har metoderne drawBackground, drawBoard, drawSnake og drawFood, som hver især tegner deres tilsvarende element. Metoden paintComponent kalder de førnævnte metoder, således at baggrunden tegnes først, så banen, så æblet og sidst slangen. Alting gentegnes hver gang spillet opdateres, hvormed der "tegnes over"det gamle billede. Det kan her argumenteres for, at f.eks. baggrunden ikke bør gentegnes, når slangen bevæger sig, lige som store dele af banen forebliver uændret, og derfor ikke behøver gentegnes. Alternativt kunne man lave et system, som undersøger hvilke områder der skal gentegnes, og hvilke der er uændret, og derfor ikke skal gentegnes. Dette kan dog i sidste ende betyde, at de kræfter der spares på tegning i stedet blot bruges på at afgøre, hvad der skal tegnes.

Vinduesstørrelse

Området, som spillet foregår på, skal kunne bestemmes til at være mellem 5x5 og 100x100. Dette kan dog skabe problemer, hvis størrelsen bliver for stor, idet banen både skal være synlig, men også passe på en gennemsnitlig computerskærm. En bane på 10x10 kan sagtens passe på en opløsning af f.eks. 400x400, men øges banens størrelse til 50x50, bliver banen svær at se. Herudover varierer skærmstørrelser, og det er derfor nødvendigt at gøre spillets vinduestørrelse fleksibel. En løsning på dette problem ville være at bestemme en fast størrelse for felterne, og lade vinduet justere sin størrelse efter dette. Ønskes det f.eks. at felterne altid har størrelsen 20x20, og at banen skal være 15x25, vil vinduets størrelse blive 300x500. Fordelen ved denne metode er at det sikres, at banen altid er synlig, og at der ikke opstår problemer, fordi forholdet mellem vinduets størrelse og banen størrelse ikke passer sammen. Ulempen ved metoden er, at store baner kan blive for store til at være på en normal skærm. F.eks. vil en bane med felter af størrelsen 20x20 og banestørrelsen 75x75 fylde 1500x1500. Herudover er løsningen ikke brugervenlig, idet en bruger kan blive forvirret over hvorfor vinduets størrelse ændrer sig fra bane til bane, og måske ligefrem ikke passer på skærmen.

En anden metode er at gøre vinduet justerbart. Denne løsning er mere brugervenlig, idet vinduets størrelse frit kan justeres så det passer til den enkelte person. Denne metode introducerer dog et andet problem, nemlig at felternes størrelse skal skaleres til at passe vinduet. Nogle opløsninger af vinduet vil ikke være et multiplum af banens størrelse, hvormed elementerne i spillet vil blive aflange. Dette problem blev løst ved at lave en baggrund og låse banens forhold, hvormed banen altid fylder mest muligt af vinduet ud, og den resterende plads bliver udfyldt af baggrunden. Idet denne løsning var nemmere at bruge og forstå, og giver bedre mulighed for justering af spillet, blev den valgt i stedet for den første løsning.



Figur 2.1: windowsize

2.5 Evaluering

-To eller en Arraylist -¿ linklist ? -Point -¿ Field -Int -¿ Enum (direction) -width+height ¡-¿ kvadratisk variabel ? -Banens opbygning: double-array -¿ positions-placering -Optimering af runtime i “food”-klassen (dobbelt for-loop eller random placering) -brug af observer - får model-view-controller til at gå op -Flere opgaver er givet til “game”-klassen frem for de andre klasser

Kapitel 3

Avanceret Snake

I dette afsnit vil den endelige version af snake blive forklaret. De dele af spillet, som også er brugt i simpel-snake, og som ikke er blevet ændret, f.eks. justering af vinduesstørrelse, vil ikke blive forklaret i dette afsnit.

3.1 Afgrænsning

I den avancerede del af spillet skal slangen bevæge sig automatisk ved hjælp af en timer, og venter derfor ikke på input fra tastaturet. Input fra tastaturen bruges derimod til at skifte retning eller til at fremskynde den automatiske bevægelse. Derudover skal slangen accelerere, dvs. at tiden mellem hver automatisk bevægelse sænkes, efterhånden som spillerens score stiger, hvilket øger spillets sværhedsgrad. Brugergrænsefladen skal kunne tilpasses brugerens skærmstørrelse, ønskede vinduesstørelse og brugerens operativsystem. Det skal også være muligt at navigere både igennem input fra et tastatur eller en mus. Før spillets start skal det være muligt at vælge mellem flere starthastigheder, slangens farve og banens størrelse. Det skal være muligt at kunne spille to spillere på samme bane.

3.2 Design

I *Driver*-klassen, hvori main-metoden for programmet ligger, oprettes tre objekter fra de tre hovedklasser *Game*, *View* og *Control*, der hver især står for henholdsvis *Model*, *View* og *Control* i *Model-View-Controller*-designet.

I JFrame'n *View* er spillets forskellige komponenter opdelt i hver deres klasse. Disse View-klasser er alle JPanels som tilføjes til vinduet og skiftes ud alt efter spillets tilstand. Er spilleren f.eks. i hovedmenuen, bruges ViewMenu klassen til at tegne denne. Er spilleren i et multiplayer-spil, bruges ViewBoard, ViewBoardMultiplayer, ViewHeader og ViewHeaderMultiplayer. På denne måde har hver scene i spillet en tilhørende klasse der forlænger JPaneler. Klasserne oprettes alle med View som parameter, og kan derfor give og bruge hinandens

funktioner, idet View-klassen har getter-metoder til alle de andre View-klasser. View-klasserne kan derfor genbruge nødvendige metoder, f.eks. metoden til at tegne baggrunden op, der f.eks. skal være ens i alle paneler. Spilleren navigerer vha. JButtons eller tastatur-input der modtages i Control-klasserne.

Control-klassen er tilknyttet View, og virker derfor alle steder i spillet. Control-klassen har funktioner som *mute* og *return to menu*, som skal være fungerende uanset hvor i spillet spilleren er, og dermed hvilke paneler der er vist. I samme pakke ligger andre control-klasser, som alle hører til en bestemt spiltilstand. Dette sikrer, at der kun bliver givet de styringfunktioner, som er nødvendige i den nuværende spiltilstand.

I *Model*-pakken findes den abstrakte klasse *Game*, som klasserne *GameSingleplayer* og *GameMultiplayer* implementerer. [.....]

3.3 Implementering

3.3.1 Model

Ændret siden simpel: Multiplayer

3.3.2 View (Brugergrænseflade)

Ændret siden simpel: Brug af billeder. Optegning af slange.

Al grafik, som ikke er fra *Swing*-biblioteket, er importeret i en fælles klasse *Image*, der gør det muligt for enhver view-klasse at få fat i alle billeder. På samme måde er selv-definerede farver oprettet i klassen Colors. Når *View*-klassen konstrueres opretter den i forvejen alle andre paneler, men tilføjer først *ViewHeader*-panelet og *ViewMenu*-panelet. Disse skiftes kun ud med andre paneler gennem input fra spilleren, når spilleren bevæger sig rundt i spillet. Denne frame bruger *BorderLayout*, der gør det simpelt at placere *ViewHeader*-panelet øverst, og det ønskede scene-panel nedenunder. Alle andre view-klasser er efterladt med standard-layoutet *FlowLayout*, da deres komponenter skal placeres i specifikke koordinater. Dette gøres med *setBounds* der definerer komponentets størrelse og placering. For at komponenterne følger med, når spilleren ændrer vinduets størrelse, er deres koordinater dog ikke altid bevaret hvis komponenterne f.eks. altid skal ligge i midten af vinduet. Disse kald på *setBounds*-metoden er placeret i panelernes *paintComponent*-metode da, denne køres hver gang vinduet skaleres. Koordinaterne beregnes derved påny hver gang vinduet skaleres, og komponenterne kan derfor altid få de rigtige koordinater. Denne metode er brugt til alle billeder, komponenter og tekst der oprettes eller tegnes.

ViewMenu indeholder metoder der tegner flise-baggrunden og et gennemsigtigt område med scenens indhold. Disse metoder bruges også af alle andre paneler, da den samme baggrund skal tegnes. Som den eneste klasse opretter *ViewMenuSingleplayer* sit eget panel der svarer til det gennemsigtige område, hvilket gør det lettere at beregne koordinaterne til alle panelets komponenter

lettere, idet der ikke længere skal tages hensyn til flise-baggrunden, som også hører med til det oprindelige panel.

I klassen *ViewBoard* tegnes grafikken til spillet. Her er lagt specielt vægt på, at felternes størrelse passer til spillerens ønskede bane-størrelse og spillerens ønskede vinduesstørrelse samtidig med at bevare deres kvadrate form. Metoden *getFieldSideLength* beregner ud fra disse to størrelser den størst mulige længde og højde af et enkelt felt og returnerer derefter den mindste af de to værdier. På denne måde kan feltet være kvadratisk og passer med sikkerhed på begge led af vinduet. Denne *getFieldSideLength*-metode kan nu bruges til at bestemme størrelsen af og tegne banen, slangen og æblet, så de passer til vinduets størrelse. Da disse tegnemetoder bliver kaldt fra *paintComponent*-metoden, der køres igennem konstant under spillet og når vinduesstørelsen ændres, har spillets komponenter altid en passende størrelse. Når spilleren færdiggør spillet enten ved at tabe eller vinde, tegnes *Game Over*-skærmen ved en gennemsigtig rektangel, med *Final Score* og JButtons, der giver mulighed for at gå tilbage til menuen eller spille igen. Udover bane-størrelse er det også muligt at vælge slangens farve. Dette gøres lige før spillets start, når spilleren har valgt farve. Biillederne for slangen farves og kan derefter bruges til at tegne slangen. Slangen farves ved at køre hver enkel af et billedes pixel igennem og derefter give det en anden farve defineret ved tre heltal givet som parametre til *colourSnake*-metoden, der farver den givne pixel, hvis dens farve ikke er svarende til slangens øjenfarve.

3.3.3 Control (Styring)

Ændret siden simpel: Mute, pause, escape.

For at give spilleren valgfrihed er der til mange funktioner både implementeret en JButton og en tilhørende genvej gennem tastaturet. Fra tastaturet er der blevet implementeret følgende genveje: Mute-button (M), pause-button (P), menu-button (Esc), tilbage-button (Backspace), (re)start-button (enter/space), som (gen)starter spillet. *Control*-klasserne [extends] *KeyAdapter*, der registerer tastatur-input, og implementerer *ActionListener*, der registrerer tryk på knapper. Disse klasser tilføjer *KeyListeners* til *View*-klassen, og *ActionListener* til panelet, der indeholder den bestemme *Control*-klasses knapper. Individuel kontrol over de forskellige knapper fås ved at sætte en unik *ActionCommand* til hver enkel knap, der derefter kan tjekkes for i *ActionListener*ens abstrakte metode *actionPerformed*. Det er her også nødvendigt efter knappetryk at bruge *requestFocus*-metoden på *View*-klassen, da spillet efter knappetryk, får et nyt fokus, hvilket betyder, at tastatur-inputtet ikke opfattes af spillet. Især når *View* oprettes, bruges *setFocusable*- og *requestFocus*-metoden, eftersom tastatur-input allerede skal være brugbart her på grund af *mute*-funktionen. Før spillets start kan spilleren som tidligere nævnt - udover at vælge farve og sværhedsgrad - vælge banestørrelsen. Denne funktion er indbygget vha. to *JFormattedTextFields* der har fået en *Formatter*, som begrænser inputtet til højst et tre-cifret tal. Dette begrænser spillerens mulighed for at indtaste en ugyldig størrelse. Giver spilleren alligevel en ugyldig værdi (under 5 eller over 100) eller efterlader et felt

tomt, printes en fejlmelding og spilleren kan ikke starte spillet før der er indtastet gyldige værdier. For alligevel ikke at gøre det svært for spilleren at indtaste rigtige værdier, ses der bort fra eventuelle mellemrum før, efter eller i midten af tallene, da disse fjernes før inputtet regnes om til et heltal, som derefter sammen med den valgte sværhedsgrad gives videre som informationer til *model*-pakkens *Game*-klasse. Farvevalget gives til *view*-pakkens *ViewBoard*-klasse, der farver billederne til slangen.

(((((((Da tekstfelterenes *Formatter* får deres caret til at sætte sig i starten af tekstfeltet selvom spilleren trykker et andet sted i feltet, implementerer klassen en *FocusListener*, der har de abstrakte metoder *focusGained* og *focusLost(...)*. I *focusGained* oprettes))))))).

3.4 Udviklingsproces

3.4.1 Control

Diskussion: Opdeling af controlklasser

3.4.2 Model

Menu

Multiplayer

Singleplayer og multiplayer

Automatisk bevægelse

Timer og acceleration

3.4.3 View

Tegning af slangen

Ændret siden simpel: Undersøgelse af slangens led

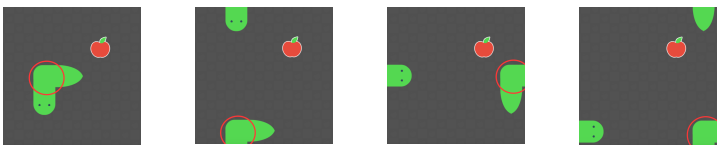
I Simpel Snake kan spilleren ud fra slangens krop se hvor han har været, men ikke hvilken vej han har bevæget sig, idet de udfyldte felter er fyldt helt ud til kanten[FIG HENVISNING]. Slangens udseende kan derfor forbedres ved at vise slangens bevægelsesretning og retningsskift i hver enkel del af dens krop og generelt erstatte alle firkanterne, med mere beskrivende billeder [FIG HENVISNING]. Dette giver seks mulige udseender for hver enkel del af slangen udover hovedet og enden af halen, der hver findes i fire versioner afhængig af retningen, som spilleren vælger for hovedet, eller retningen af kropsdelen lige før halen. Kropsstykkerne imellem er dog ikke kun afhængig af retningen af stykket lige før eller lige efter, men begge dele. Den vandrette del og den lodrette del af slangen bestemmes let ved at undersøge om stykket, der skal tegnes ligger i samme række eller kolonne som stykkerne før og efter. Slangens hjørnestykker bestemmes

på en mere indviklet måde, da stykket og dens tilgrænsende stykker aldrig ligger i samme række eller kolonne, men derimod kan ligge i fire forskellige forhold til hinanden (Se figur 3.1). I figur 3.1a ligger de tilgrænsende felter lige under og til højre for hjørnet, men dette gælder f.eks. ikke for figur 3.1c hvor det ene stykke ligger lige under, mens det andet stykke ligger til venstre for hjørnet uden at grænse op til dens venstre side, der ellers ville give hjørnestykket spejlvendt i y-aksen. Da felterne ligger i en XXXXX-liste sorteret efter slangens opnåede dele, sammenlignes et felt med feltet før og efter det i listen. Da stykket foran og bagved uden påvirkning på hjørnestykket kan bytte plads, findes der altså otte situationer for et enkelt hjørnestykke. I alt tjekkes der derfor - for kroppen alene - 34 mulige forhold mellem et stykke og dens to tilgrænsende stykker.

I *ViewBoard*-klassen er der lavet en *drawSnake*-metode, der består af en del if-statements, for hvilket figur af slangen, der skal bruges i en bestemt retning. I *snakeCorner*-metoden returnerer den en lang boolean *isCorner*, som tidligere har været kopieret 4 gange for de fire retninger, men som nu er blevet simplificeret med en masse variable til den samme boolean. Måden det blev gjort på, var at finde et mønster i de fire lange booleans før og så indsætte 12 forskellige varibale, der gjorde det muligt kun at kalde en boolean. På samme tid kører programmet ikke alle if-else-statements igennem hver gang den skal lave et hjørne, men kun det hjørne som bliver kaldt. Variablerne kan enten have værdien 0, 1, -1, lastColumn eller lastRow. Da det specifikke hjørne skal dukke op et bestemt sted alt efter hvor på banen slangen befinder sig. De første to rækker kode af *isCorner*, fortæller om når slangen er placeret midt på banen (se figur 3.1a), og drejer til den specifikke retning, så må det foranliggende (front) og bagvedliggende stykke (behind) ligge et bestemt sted, som enten getColumn()+1 eller getColumn()-1 og på samme måde getRow()+1 eller getRow()-1. I de to næste ligner kode i *isCorner* beskriver den, hvis slangen er placeret på den sidst række (altså i bunden af banen) og går gennem torussen, så den ender øverst i banen (se figur 3.1b). De to næste linjer kode er på samme måde, når slangen ligger yderst til højre eller venstre og går gennem torussen (se figur 3.1c). De to sidste linjer kode er til, når slangen går gennem torussen to steder i et af hjørnerne. Altså hvis slangen f.eks. er nede i højre hjørne, går gennem torussen ved at gå ned ad, og straks til højre gennem torussen igen (se figur 3.1d). Derved er alle hjørne-situationer gennemgået. Man kan se et mere detaljeret billede af snake-kroppen i Appendix (Bilag A).

Figur 3.1

(a) Corner 1 (b) Corner 2 (c) Corner 3 (d) Corner 4



Lyd

Som en mindre tilføjelse til spillet er lyd tilføjet i form af Audio-klassen. Funktionerne i Audio-klassen bruges til individuelt at spille

3.5 Evaluering

For at sikre stabilitet af brugergrænsefladen er spillet afprøvet i forskellige størrelser og på forskellige operativsystemer. Placeringen af komponenterne er ens i Micrsoft Windows 8.1 og dens ældre versioner, [INSERT———]. Derimod har JButtons et lidt andet udseende på [INSEEEEERT], idet deres baggrundsfarve i dette operativsystem ikke er synlig medmindre deres kant skjules. Derudover virker de her heller ikke, hvis de tilføjes til et panel i *paintComponent*-metoden. Det første problem løses ved at skjule kanten eller i stedet at bruge et ImageIcon til knappen i stedet for at give den en baggrundsfarve. Knappens funktionalitet opnås ved at tilføje knapperne i konstuktøren, men stadig med *setBounds*-metoden i *paintComponent*-metoden.

Med de nye funktioner og visuelle forbedringer kørte spillet en smulle langsommere end før, mens det dog kun er synligt på svage [¡-ERSTAT M SOMETHING NICER] computere. [INSERT: LØSNING HVIS VI NOGENSINDE FINDER EN : D]

-ingen brug af jbutton -¿ jbutton -Se alle slangens led - gør den lidt forsinket, når den er lang - Tjek for gentagelser

Kapitel 4

Konklusion

Bilag A

Slangens opbygning

A.1 Detaljeret oversigt af slangens opbygning