

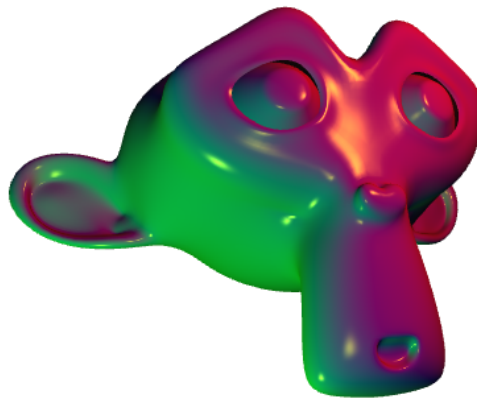
Projektarbejde, Snake

Anna Ølgaard Nielsen
s144437

Christian Søholm Andersen
s103080

Mathias Enggrob Boon
s144484

Van Anh Tri Trinh
s1444449



Introduktion til Softwareteknologi, 02121
Technical University of Denmark
September-December 2014

Chapter 1

Introduktion

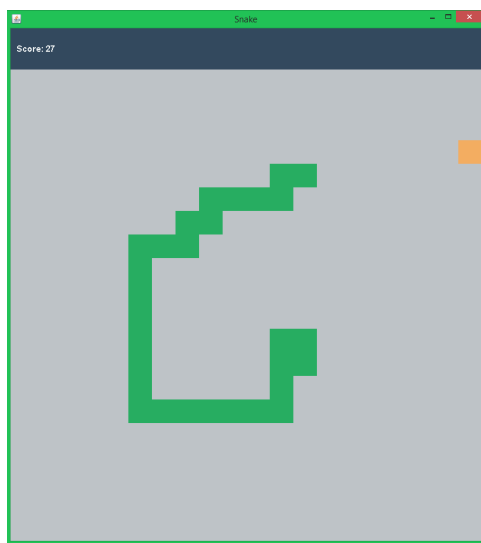
I denne rapport dokumentere vi designet, implementation og evaluering af et snake spil. Snake spillet er det velkendte spil fra for eksempel Nokia 3310 [1].

Formålet med projektopgaven er at genskabe det klassiske spil Snake, samt at dokumentere hvordan vi opbygger spillet. Spillet laves i to versioner: Simpel Snake og Avanceret Snake. Det skrives i Java.

Simpel Snake er en primitiv version af spillet, se Fig. 1.1a. Styring og bevægelse foregår kun vha. input fra spilleren. Det bruges som udgangspunkt for den avancerede udgave.

I Avanceret Snake er der tilføjet forskellige funktioner, som enten forbedrer brugerfladen eller ændre på hvordan spillet spilles. For eksempel tilføjelse af hovedmenu og automatisk bevægelse af slangen, se Fig. 1.1b. Vi har udvidet det så to personer kan spille mod hinanden.

I rapporten vil designet og implementeringen af begge spillets versioner blive forklaret. I evalueringens kapitlerne redegør vi for tanker, valg og komplikationen som vi stødte på under projektet.



(a) Sempel Snake



(b) Avanceret Snake

Figure 1.1: Billede af Sempel og Avanceret Snake.

Chapter 2

Simple Snake

2.1 Afgrænsning

Afgrænsning af Simpel Snake er defineret af projektoplægget. Det vil være bedst at læse dette for en uddybende afgrænsning. Vi laver et resume over oplæggets relevante punkter. Simpel Snake er opbygget af en bane, en slange og noget mad. Slangen kan bevæge sig med for eksempel piletasterne. Når slangen spiser mad bliver den længere.

Banen Banen er opbygget som en todimensional $n \times m$ torus. Hvis man bevæger sig ud over banens kanter, fortsætter man på den modsatte side. Banen er opdelt i todimensionale felter (x, y) .

Slangen Slangen består af et antal sammenbundne felter. Den har et hoved og en hale. Slangens hoved har en bevægelsesretning *op*, *ned*, *højre* eller *venstre*. Gennem input fra tastaturet skal det være muligt at bevæge slangen. På skærmen skal slangen visualiseres via nogle firkanter som repræsentere slangens felter.

Mad På banen skal der være et felt med mad, som slangen kan spise. Når slangen rammer dette felt skal den blive et felt større og der skal sættes et nyt mad felt på banen. Det nye felt skal være uniformt tilfældigt fordelt.

Kollision Spillet slutter når slangen bevæger sig ind i et felt som er udfyldt af slangen selv. I så fald, har spilleren tabt. Spilleren har vundet slangen fylder hele banen. Bemærk at slangen godt kan bevæge sig ind i sit hale felt, da halen også flytter sig i det samme træk.

2.2 Design

Snake-spillet er lavet efter et Model-View-Control design (MVC). Styringen, spil logikken og den visuelle repræsentation holdes adskilt i tre dele. Med dette bygges et meget modulært program. Spil logikken er Model, styringen er Control og den visuelle repræsentation er View. Disse tre dele skal kode mæssigt holdes adskilt. Spil logikken må ikke kende til styringen eller repræsentationen. Repræsentation må ikke kende til styringen. Styringen må godt kende til spil logikken og repræsentationen, se Fig. 2.1 for en illustration.

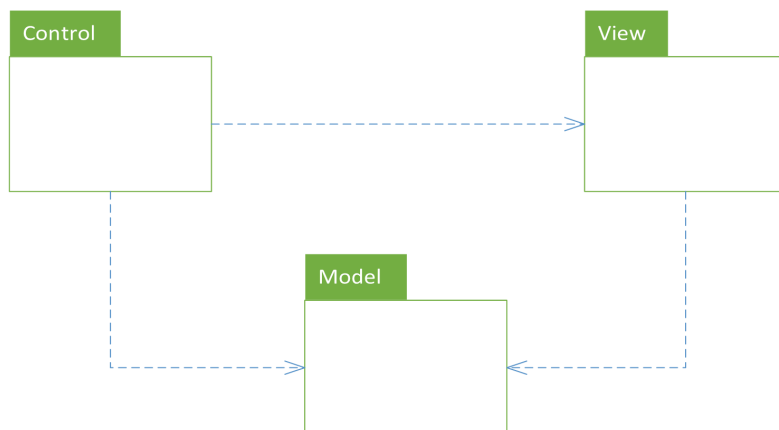


Figure 2.1: Oversigt over Model-View-Control design.

Brugeren interagerer med den del der er dedikeret til styring (Control). Control manipulerer spil logikken (Model) og spillet visualiseres i View-koden. Funktioner, der påvirker programmets tilstand, holdes i Model-koden. Control modtager input fra brugeren og kalder relevante metode i View og Model. View må ikke ændre på spillets tilstand, det skal kun vise hvad der er i Model.

Man kan tænke på det sådan at View observerer Model og viser hvad den ser til brugeren. Control kommunikerer med brugeren og sender relevant kommunikation videre til View eller Model.

Simpel Snake er designet, så spillets logik ligger i *Game* klassen i Model-pakken. *Game* indeholder relevante klasser, som for eksempel *Food* og *Snake*. Se Fig. 2.2. Vi har valgt at lave en enumerations klasse *Direction*. Den forsimples kommunikationen til *Game*. Det er nemmere at tilkoble kommandoer og statements til hver retning, hvis der eksplicit er en *Direction* klasse som begrænser retnings mulighederne. Vi har valgt at gøre *Game* til en *Observable*. Med dette kan vores View klasser observere spil logikken og opdatere det som brugeren ser, så snart spillets tilstand ændrer sig. *Snake* klassen afhænger af *Game* klassen fordi at *move* metoden i *Snake* er nødt til at signalere til *Game* at slangen har bevæget sig. Dette gør den ved at kalde *snakeHasMoved*.

I *View* klassen har vi vores vindue som brugeren ser spillet igennem. Det har en *BoardPanel* klasse som viser den nuværende spil tilstand. *BoardPanel* nedarver fra *Observer* og opdatere panelet hver gang spillets tilstand ændres. Når dette sker tegnes den nuværende bane, slange og mad som er i *Game* objektet. For eksempel bruges *getSnake()*, til at bestemme slangens position. Scoren holdes i *Game* objektet. *ScorePanel* ligger som et panel over *BoardPanel* i vinduet. Den observerer *Game* og opdatere sig selv når scoren ændre sig. Se Fig. 2.3.

Control klassen kalder relevante *Game* og *Snake* metoder når brugeren trykker på dens keyboard. Den bestemmer hvornår og hvorhen slangen skal bevæge sig. *Control*-klassen nedarver fra en *KeyListener*. Se Fig. 2.4.

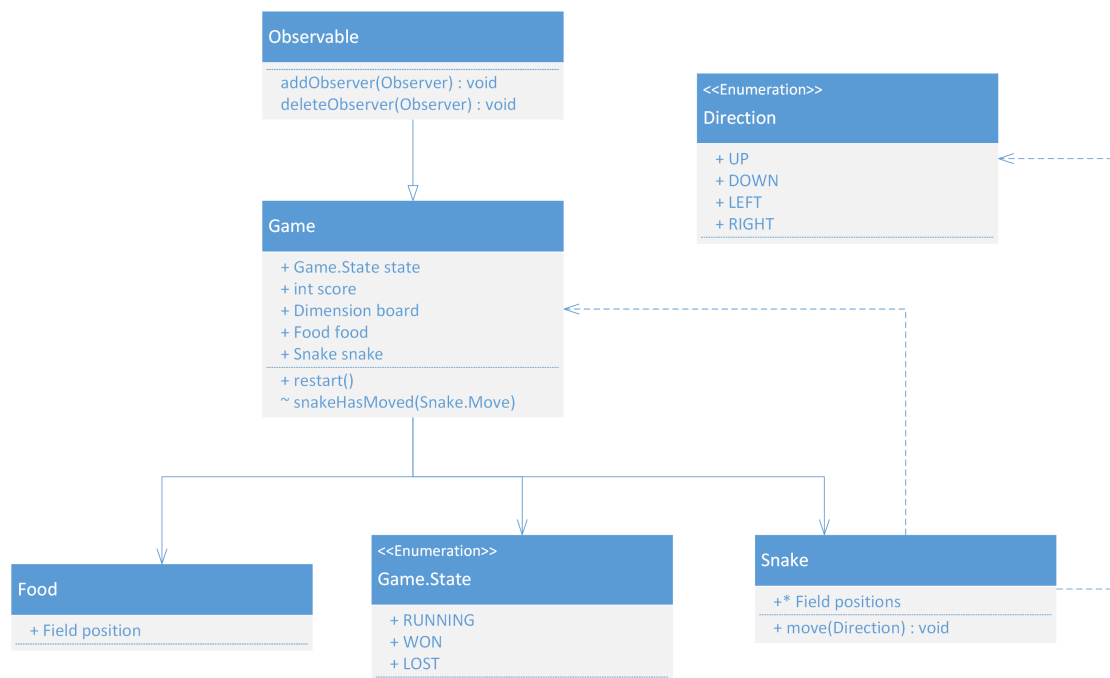


Figure 2.2: Oversigt over vores Model design.

For at starte spillet bruges klassen *Driver*, som opretter et *Game*-, *View* og *Control*-objekt.

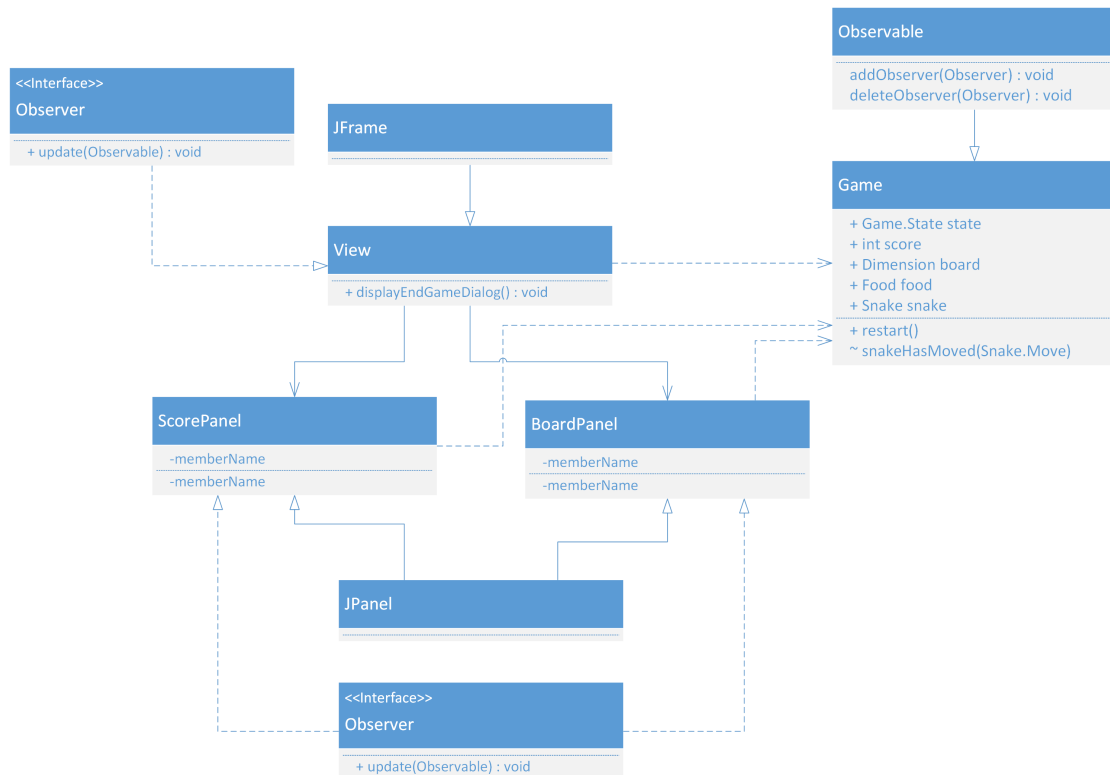


Figure 2.3: Oversight over vores View design.

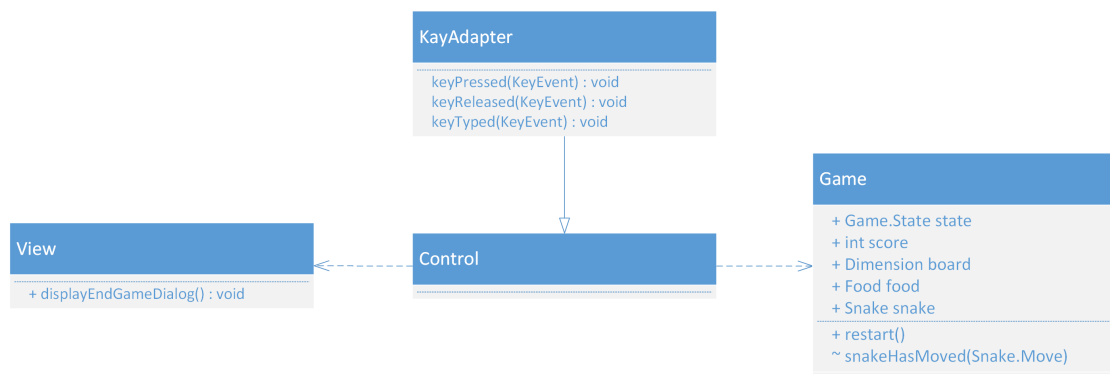


Figure 2.4: Oversight over vores Control design.

2.3 Implementering

2.3.1 Model

Spillets Model består af en række klasser, der tilsammen udgør selve spillets funktioner og objekter. Klassen *Game* er selve spillets logik.

Klassen *Field* bruges til at repræsentere et felt på banen. Den fungerer på samme måde som Javas standard *Point*-klasse. Forskellen mellem *Point* og *Field*, er at vi bruger row og column i stedet for x og y. Spillepladen er delt op i rækker og kolonner, frem for en x- og y-akse. *Field* er lidt mere selvdokumenterende og selve klasse navnet *Point* giver ikke rigtig nogen indikation om hvad det point bruges til. Vores *Field* har også den fordel at man ikke dynamisk kan ændre dens række og kolonne. De er låst fast til den samme værdi og kan kun ændres hvis man opretter et nyt *Field* objekt.

Food klassen som repræsentere et stykke mad på banen. *Food* indeholder et *Field* med madens position.

Food og *Field* har kun to værdier i sig (en række og søjle). Vi har designet klasserne sådan at de ikke kan ændres efter at de er blevet oprettet. Dette gør at vi kan returnere vores dem i getters og være 100% sikre på at udefrakommende klasser ikke ændrer dem. Nogen gange kan det godt være svært at resonere omkring kode, hvis alle objekter vilkårligt kan ændre hinanden. Immutable klasser sikre at vi altid kan være sikker på at objekter ikke har ændret sig. Det gør at det er umuligt for Control-pakken at snyde spil logikken.

Slangen er defineret i *Snake*. Slangens krop består af en række felter, hvoraf det første er hovedet, og de resterende er kroppen inklusiv halen som er til sidst. Koordinaterne for disse felter er gemt i en *ArrayList*. Det første element er slangens første led, hovedet, det andet element er slangens andet led osv. Når slangen bevæger sig tilføjer vi et nyt hoved i listen og sletter halen. Hvis slangen bevæger sig og samtidig spiser noget *Food*, så lader vi være med at slette halen. Slangen vil altid være forholdsvis lille, så om vi gemmer dens elementer i for eksempel en *ArrayList* eller *LinkedList* er stort set lige meget. I *Snake* konstruktøren bliver slangens hoved og hale oprettet. Hovedet bliver placeret i banens centrum. Halen placeres i kolonnen til højre for på samme hovedets række.

Vi har valgt at oprette spillepladen i *Game*-klassen som et *Dimension* objekt. Vi bruger ikke en *Board*-klasse, der beskriver banen for sig selv. I bagklogskabens øje burde vi nok have brugt en *Board* klasse. De andre spilelementer er defineret i deres egne klasser for eksempel *Food* og *Snake*.

Slangens flyttes ved at kalde dens *move* metode. Metoden signalere internt til moder *Game* objektet at slangen har bevæget sig. Det gøres med package private metoden *snakeHasMoved(Move)*. *Move* er et enum som betegner slangens handling. Om slangen ikke kan bevæge sig, spiser et æble, er død eller bevæger sig er henholdsvis *EAT_NECK*, *EAT_FOOD*, *EAT_BODY* og *NORMAL*. Først tjekkes at slangen ikke bevæger sig ind i dens hals. Hvis det sker signaleres *EAT_NECK* og der returneres. Det undersøges derefter om der er mad på samme felt som slange hovedets nye placering. Er dette tilfældet, signaleres *EAT_FOOD*, slangens hoved flyttes og der returneres. Hvis ikke, undersøges der om den nye placering indeholder slangens krop. Gør der det signaleres *EAT_BODY* og returneres. Hvis ikke, flyttes slangens hoved til den nye position og halen i slangen fjernes. Resten af slangens krop følger med, da slangens *ArrayList* automatisk skifter hvert index, når den ny hoved placering indsættes som det første element.

I *Game* klassen findes *snakeHasMoved(Action)* metoden, som *move* bruger til at signalere med.

Den sætter vundet eller tabt tilstanden i spillet og tjekker om slangen har spist. Hvis den har spist oprettet et nyt mad objekt. Den inkrementerer også scoren.

Herudover findes *generateFood* metoden, som sikrer generere et nyt stykke mad, på et gyldigt felt, dvs. et felt der ikke er udfyldt af slangen. For at sikre dette, placeres æblet på et tilfældigt felt inden for banens rammer, hvorefter der undersøges om et af slangens led har samme koordinater som æblets felt. 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, da der er stor sandsynlighed for at ramme et felt, der er optaget af slangen. 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 for-løkke, der løber gennem alle rækker og kolonner, hvorefter et tilfældigt element i listen vælges som æblets position.

Klasser *Game* nedarver fra klassen *Observable*, så det er muligt for View at modtage ændringerne i Model, uden at Control skal signalere til View at den har ændret Model.

Immutable Model klasser

Klasserne *Food* og *Field* er meget små. De har kun to værdier i sig (en række og søjle). Vi har designet klasserne sådan at de ikke kan ændres efter at de er blevet oprettet. Dette sikrer os at vi kan returnere vores dem i getters og være 100% sikre på at udefrakommende klasser ikke ændrer dem. Nogen gange kan det godt være svært at resonere omkring kode, hvis alle objekter vilkårligt kan ændre hinanden. Immutable klasser sikrer at man altid kan være sikker på at objekter ikke har ændret sig. Det gør at det er umuligt for Control-pakken at snyde spil logikken.

2.3.2 View (Brugergrænsefladen)

Brugerfladen er samlet i klassen *View* der forlænger *JFrame*. I *View* findes en konstruktør, hvor der oprettes et *ScorePanel* objekt og et *BoardPanel* objekt. Der begge er nedarvet fra *JPanel*. *ScorePanel* fylder det øverste af vinduet og *BoardPanel* resten. Boardpanelet viser spil banen med slangen og maden. Slangen og maden er begge firkanter i forskellige farver.

ScorePanel klassen forlænger *JPanel* og implementerer *Observer*. Klassen har en *update*-metode, som signalere at panelet skal optegnes på ny. Hver gang noget i spillet ændre kaldes denne function. Metoden *paintComponent* benyttes til at tegne score teksten.

BoardPanel forlænger *JPanel* og implementerer *Observer*. *BoardPanel* består af en række draw-metoder, samt en overridet *paintComponent* metode, der kalder draw metoderne for at tegne alle spillets komponenter. I *drawSnake*-metoden bruges positionerne for slangens felter til at tegne slangen. Størrelsen for et felt udregnes i *getWindowRectangle*, og afhænger af vinduets størrelse og antallet af banens felter. Når feltets størrelse er udregnet, tegnes et rektangel i vinduets grafik kontekst.

Da spillepladen skal være mellem 5x5 og 100x100 felter, kan det skabe problemer, hvis man ikke kan justere størrelsen på vinduet. Vinduet kan være for stort til at passe på en gennemsnitlig computerskærm. 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. Ulempen ved metoden er, at store baner kan blive for store til at være på en normal skærm. Derfor har vi valgt at lave en fleksibel størrelse for felterne, så de følger forholdet mellem vinduestørrelse og antal felter. I Fig. 2.5 kan man se, at hvis vinduet skal være justerbart, så kan man risikere at selve banen bliver aflang og ikke særlig pæn at spille på.



Figure 2.5: Eksempel på udstukne felter.

2.3.3 Control (Styring)

I Simpel Snake er der ikke brug for nogen menu, og det er derfor muligt at holde styringen af spillet meget simpel. Vi lader *Control* klassen nedarve fra Javas *KeyAdapter*. I dens konstruktør sætter vi den til at lytte efter key events på vores *View* klasse.

2.4 Evaluering

Arbejdsproces

Formålet med projektet var at lave en simpel udgave af snake, og derefter tilføje flere funktioner for at lave en mere avanceret version. Dette gør det ideelt at tilføje én funktion ad gangen, frem for at planlægge alle funktioner på en gang, og tilføje dem samtidig. Resultatet bliver til at starte med et simpelt men fungerende program, hvorefter yderligere funktioner kan tilføjes. Programmet er altså udviklet iterativt, hvorved 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, der skal tilføjes til programmet. Derefter fordeles opgaverne blandt gruppens medlemmer. Gruppe medlemmet afgør selv, hvordan en funktion skal designes og implementeres, men sikrer at implementeringen 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.

Opbygning af banen

Vi overvejede at designe banen som et todimensionalt array. F.eks. et array [10][5] vil give en bane med længden 10 og bredden 5. Hvert element i arrayet bestemmer da, hvad der befinder 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.

Vi valgte ikke denne metode på grund af at den skal sende et todimensionalt array med banens størrelse, hver gang banen skal tegnes. Ved en bane på f.eks. 100x100 betyder dette, at der sendes i alt 10000 elementer til View-koden, hver gang banen skal tegnes. Vores anvendte metode sender derimod kun de relevante elementer, dvs. æblets og slangens position, til View-koden. En anden ulempe er at det er en smule mere kompliceret at bruge dette array til noget. Hvis man vil vide hvor at slangen er skal man loope igennem hele arrayet.

Snake

Da slangen i snake-spillet består af en række felter, som alle har netop et koordinat i forbindelse med andre led, er en effektiv måde at bestemme slangens position på en ArrayList, idet denne datastruktur 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. Vi fravalgte at bruge LinkedList da denne optimering ikke kan måles så længe slangen har under 10.000 elementer.

Score

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 fordelene, 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 separat, så scorepanelet kun gentegnes, når scoren ændrer sig. Hvis scoren derimod er et datafelt, tegnes scoren efter hver tur, også selvom scoren er uændret. Vi implementerede først scoren som en klasse, men slettede den senere. Det var simpelthen for meget at operette en klasse, til at holde øje med en enkelt integer.

Tegning af spillet

For at visualisere spillet er det nødvendigt at kunne optegne banen, slangen og æblet. Dette gøres i klassen BoardPanel, som har metoderne drawLevel, 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.

Tegning af vinduet

En alternativ metode til at gøre vinduet justerbart, er at give hvert felt en absolut størrelse. Så ser spillepladen altid pæn ud, og kommer ikke til at afhænge af alle mulige vinduesforhold. Ulempen er så bare, at alt efter hvor stor banen er, så kan vinduet blive for stort til computerskærmen, og hvis den er for lille, kan man ikke forstørre banen op. På denne måde er det en "for let" løsning bare at sætte absolute værdier ind, for ikke at få en "grim" bane. Problemet bliver løst i avanceret snake ved at lave en baggrund og låse banens forhold, hvormed banen altid fylder mest muligt af vinduet ud, mens den beholder sin form, og den resterende plads bliver udfyldt af baggrunden.

Brug af *Game* klassen

Efter vi havde delt de fleste funktioner ud i klasserne, besluttede vi os for at samle nogle metoder i *Game* klassen frem for deres oprindelige klasse, f.eks. *GenerateFood*-metoden og hvordan banen skulle opstilles. Vi satte ikke *move* metoden ind i *Game* klassen, da den ikke som sådan skal bruge nogle informationer fra andre klasser. Logisk giver det også mening at slangen bevæger sig som en uafhængig enhed og at det ikke er *Game* som flytter slangen. Det giver os den ekstra afhængighed at snake er nød til at kende til game klassen, så den ved hvor at maden ligger og hvor stort boardet er. I den avancerede del går vi ind faktisk den anden vej og flytter *move* ind i *Game*. Slangen bliver så bare et *glorificeret* array, som Game flytter rundt på. Vi er ikke helt sikre på hvad der er det bedste. I den avancerede del kender *Snake* ikke til *Game*, så det er nemmere at overskue programmet. Til gengæld burde slangen måske kende til *Game* hvis

vi ville implementere noget AI, så den kunne bevæge sig som en uafhængig enhed, der tager beslutninger om sine omgivelser.

Chapter 3

Avanceret Snake

3.1 Afgrænsning

Den avancerede del er en udvidelse af det grundlæggende spil. Vi skal selv vælge hvad vi vil arbejde på. I dette afsnit redegør vi for det som vi har valgt.

Gameplay

For at kunne kalde spillet snake, så skal slangen næsten kunne bevæge sig af sig selv. Den skal kunne bevæge sig i periodiske intervaller. Ved input fra keyboardet skal slange kunne skifte retning. Til at implementere sværheds graden kan slangens hurtighed ændres. Det skal ligne snake spillet som på Nokia's 3310. Her begynder spillet at gå hurtigere jo mere slangen har spist.

Menuer

Vi vil implementere menuer. Det skal være en main menu som popper op når vi starter spillet. I den skal der mindst være en single player knap og en forlad spillet knap. Inden at man gå ind i spillet skal der være en menu hvor at man kan vælge størrelse på spillet. Eventuelt også sværheds grad og slange farve. En menu som viser keyboard tasterne til at styre spillet med vil være godt at have.

Grafik

Vi laver custom grafik til spillet. I menuerne kan vi implementere nogle gode overskrifter. Vi vil gerne have en header i vinduet med et custom snake logo. Slanges krop skal forbedres, sådan at den ikke bare er nogle firkanter. Den skal ligne en aflang slange krop.

Når vi resizer vinduet med spillet skal brugergrænsefladen tilpasse sig den nye størrelse. Skærmens størrelse skal udnyttes optimalt.

Lyd

Vi vil gerne integrere lyd! Når slangen spiser et eller andet så kan der måske være en nomnom lyd. Eller et skrig når man taber. Det er vigtig at audio er en del af view delen i model-view-controller konceptet. Det vil ikke give mening at integere det i modellen. Dette vil måske også kræve et event system, men det er en implementering detalje.

Multiplayer

Vi vil lave lokal multiplayer. To personer skal kunne spille mod hianden ved at bruge forskellige taster på samme keyboard. Multiplayer gameplayet er ikke så vigtigt, men der skal være en måde hvor at spillerne kan vinde på. Man kunne for eksempel lade den ene spiller vinde hvis han spiser den andens slange krop.

Kode

Koden til den avancerede del skal være af høj kvalitet. Vi vil prøve at skrive den så godt vi kan, med fokus på at koden skal være let læselig.

3.2 Design

3.2.1 Klasserelationer

Den avancerede versions design er også baseret på Model-View-Controller-conceptet, og indeholder derfor pakkerne, *model*, *view* og *control*. Klasserne i control-pakken er uafhængige af klasserne i de andre pakker, men oprettes og tildeles i view-klasser, der giver kontrol over spillets progression og brugergrænsefladens udseende.

I *Driver*-klassen, hvori main-metoden for programmet ligger, oprettes kun et *ViewFrame*-objekt, der nedarver fra *JFrame*, og viser spillets vindue. I denne klasses constructor oprettes alle andre relevante klasser, som findes i view-pakken [REFERENCE TIL KLASSEDIAGRAM]. Disse nedarver fra *JPanel*, og tilføjes og fjernes fra *JFramen* afhængig af hvor i programmet spilleren befinder sig. Er spilleren f.eks. i hovedmenuen, bruges *MenuPanel*-panelet, mens *HeaderMultiplayerPanel* og *BoardMultiplayerPanel* tilføjes, hvis spilleren er i multiplayer-delen af spillet. På denne måde har hver scene i spillet en eller to tilhørende klasser der nedarver fra *JPanel*.

Spilleren navigerer vha. *JButtons* eller tastatur-input der modtages i control-klasserne. Control-klassen *ViewFrameListener* oprettes ligeledes i *ViewFrames* constructor, hvorefter den som *KeyListener* tilføjes til alle panelerne. Denne klasses funktioner er globale, da den står for 'mute'- og 'return to menu'-funktionerne, der konstant skal være tilgængelige for spilleren, uafhængig af de viste paneler.

Hvert panel opretter også i deres egne constructors deres tilhørende *Listener* fra control-pakken. Disse *Listeners* står for kontrollen, der er unik for deres panel. Alle panel-klasserne oprettes med *ViewFrame* som parameter, der derefter igen bruges som parameter, når panel-klassens control-klasse oprettes. Control-klassen kan derefter bruge *ViewFrame*, til at skifte paneler ud, når knapper eller taster trykkes. Specielt for *MenuListener*, der hører til hovedmenu-panelet, oprettes i constructoren også et *GameSingleplayer*- og et *GameMultiplayer*-objekt, da disse skal bruges, når der klikkes på 'Singleplayer'- eller 'Multiplayer'-knapperne.

GameSingleplayer og *GameMultiplayer* er underklasser til *Game*-klassen, og står for spil-delens oprettelse ved at bruge de andre objekt-klasser i model-pakken: *Board*, *Food* og *Snake*. Derudover findes der i model-pakken hjælpeklasserne: *Field*, *Direction*, *Event* og *Player*. *Game* extends *Observable*-klassen, der gør det muligt for view-klasserne at blive notificeret, når der foretages ændringer i spillet, og følgelig tilpasse sig ændringerne i brugergrænsefladen.

Al grafik, som ikke er fra *Swing*-biblioteket, er importeret i klassen *Images*, giver enhver view-klasse adgang til at få fat i alle billeder. På samme måde er selv-definerede farver oprettet i klassen *Colors*.

3.2.2 Multiplayer

Multiplayer ligner singleplayer rigtig meget. De forgår begge med slanger på samme board. Problemet med at integrere multiplayer ind i vores singleplayer kodebase er at de steder hvor at multiplayer ikke ligner singleplayer er meget spredt.

Man kunne sige at singleplayer og multiplayer kunne dele 90% af koden, men de resterende 10% er flettet ind i de 90%. For eksempel har multiplayer et lidt anderledes score system, så single og multiplayer headeren som viser score kræver lidt anderledes tekst. Det samme gælder for slange farve valg, hvor at der nu er to slanger. Multiplayer delen skal nok også have lidt anderledes

gameplay. Det kan også betyde at den måske har noget ekstra slange mad eller en lidt anderledes slange.

Vores mål for multiplayer designet var forståelighed og at genbruge mest muligt kode mellem single og multiplayer. Vi overvejede at bruge en kombineret single og multiplayer klasse som kunne holde et array af slanger. Så hvis der var to slange ville vi bruge multiplayer gameplay og singleplayer gameplay når der var en. Fordelen ved dette ville være at view delen i MVC, bare kunne loope over nogle array lige meget om det var single eller multiplayer. Vi ville også nemt kunne tjekke om vi var single eller multiplayer. Det ville bare være en if statement om der er en eller to slanger. Det vil også være nemt at tilføje for eksempel 10 slanger/spillere. Ulemperne ved dette er at forskellen på single og multiplayer er meget lille. Vi kan genbruge en hel masse kode mellem dem, men er det det bedste hvis det resulterer i en masse if singleplayer then ... else statements? Et mere forståeligt design ville have en adskillelse af single og multiplayer.

Dette ledte til designet i Fig. ?? . Vi har lavet to separate single og multiplayer game klasser som arver fra en abstract game klasse. Fordelen ved dette er at vi har en separation mellem multi og singleplayer. Det som forgår i singleplayer klasse har ingen indflydelse på multiplayer klassen. Med dette design løber vi ind i problemer når vi skal implementere view og controls i MVC. Det vi har valgt at gøre er at lave en base udgave af hvert komponent. Basen har alt det som er fælles for single og multiplayer. Vi laver så to nye klasser BaseSingleplayer og BaseMultiplayer som nedarver fra base klassen. De to klasse kan så frit override/udvide implementationerne i base klasserne.

FIG VISIO!!!!

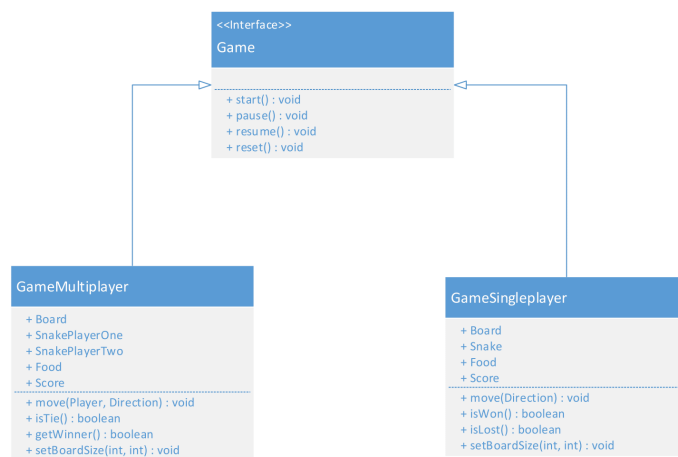


Figure 3.1: Overview of our multiplayer design.

3.3 Implementering

3.3.1 Model

Modellen er nu lavet så det er muligt at vælge mere end én spiller. I både *GameSingleplayer* og *GameMultiplayer* findes en timer hvor det dog kun er i Singleplayer-spillet at timerens opdateringsinterval formindskes efter hvert femte samlet æble. Slangen er stadig defineret som en *ArrayList* i klassen *Snake*, hvor bl.a. dens retning defineret ved en 'Direction'-enum er og en 'move'-metode.

I spil-klasserne oprettes en eller to *Snake*-objekter og et *Board*-objekt med en given højde og bredde. Der oprettes også et *Timer*-objekt, der opdaterer slangens bevægelse med et vis tidsinterval imellem hver opdatering.

Game-klasserne har også en 'move'-metode hver, som kaldes med en 'Direction' som parameter af klassernes tilhørende control-klasser, *BoardSingleplayerListener* eller *BoardMultiplayerListener* - når en a-piletasterne (player 1) eller (A, S, W, D) (player 2) trykkes ned. I denne metode tjekkes der, om slangen spiser sig selv igennem dens metode 'isNeckDirection'. Denne metode tjekker om den givne retning 'direction' er lig med 'getOppositeOf' den nuværende retning, 'headDirection'. 'getOppositeOf' er *Direction*-klassens metode, der tager en *Direction* som parameter, og returnerer den modsatte *Direction*. Returnerer 'isNeckDirection' true, gås der ud af 'move'-metoden igen, og spillet venter på det næste input. Returnerer den false fortsætter 'move'-metoden med metoden 'isBoardFull', der tjekker om det er det er den sidste frie plads i banen ved at sammenligne banens størrelse og slange-ArrayListens størrelse. I dette tilfælde har spilleren vundet.

Der oprettes også et felt, 'newHeadPosition', der oprettes ved at kalde på 'Snake'-klassens metode 'getNextHeadPosition', som tager 'direction' og 'board' som parametre. Her beregnes den nye række og kolonne for hovedet, hvis den bevæger sig i den givne retning. 'board' kalder på sin 'wrap'-metode, der tager den nye positions række og kolonne som parametre. *Board*-objektet repræsenterer banen ved at have en længde og en højde målt ved rækker og kolonner. Denne har metoden 'wrap', der tjekker om den givne række og kolonne ligger inden for banens størrelse. Hvis ikke, returnerer den et nyt *Field* med en række eller kolonne, der ligger i den modsatte del af banen. Ligger feltet inde på banen, returnerer den blot et nyt felt med de samme koordinater.

Der undersøges nu med det nye felt om det har samme koordinater som et oprettet 'Food'-objekt, der oprettes på samme måde som i den simple version af spillet. Denne boolean bruges i *Snake*-klassens egen 'move'-metode. I denne metode tjekkes først om den ny-beregnete hovedposition har samme koordinater som en del af kroppen. I dette tilfælde returnerer metoden true, spillet stopper og spilleren har tabt. Ellers fjernes det sidste element i slangens *ArrayList* (halen), hvis slangen *ikke* spiser *Food*-objektet, mens den tilføjer et nyt hoved på index 0 uanset om den spiser *Food*-objektet eller ej.

Specielt for multiplayer-spillet tjekkes der også om den en slange

Multiplayer

3.3.2 View (Brugergrænseflade)

Ændret siden simpel: Tilføjede scener. Brug af importerede billeder. Optegning af slange efter bevægelse.

Når *View*-klassen konstrueres, opretter den som tidligere nævnt panel-objekterne fra samme pakke. Derudover opretter den også view-klassen *Audio*, der står for spillets lyde, som spilles efter slangens handlinger i spils scenen hvis klassens Boolean felt 'muted' er falsk. *ViewFrame* kalder først på metoden 'showMenu', der kalder på metoden 'setFrameComponents', som bruges hver gang vinduet skal skifte paneler ud. Metoden fjerner først alle sine komponenter, tager to nye komponenter som parameter, og placerer den førstnævnte øverst i sit *BorderLayout*, og det andet komponent i midten - dvs. nedenunder. Når 'showMenu' kaldes, sættes *HeaderBasePanel*-objektet som det øverste panel, mens *MenuPanel* er panelet under. *HeaderBasePanel* viser en topbar, som indeholder spillets titel, tastatur-genveje og en *JButton*, som viser om lyden er slået til eller fra og som også kan bruges til at slå lyden til eller fra. *ViewMenu*-panelet indeholder *JButtons* som giver spilleren adgang til spillets scener: *Singleplayer*, *Multiplayer* og *Controls* - der fører til hjælpescenen *ControlsPanel*. Desuden har den også en *Quit*-knap, som fungerer som vinduets kryds i hjørnet.

Da *ViewFrame* bruger *BorderLayout*, er det simpelt at placere et header-panel 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 *setLocation* der definerer komponentets præcise placering ved x- og y-koordinater. Når spilleren ændrer vinduets størrelse, er deres koordinater dog ikke altid bevaret hvis komponenterne f.eks. altid skal ligge i midten af vinduet, eftersom punktet (0,0) ændres, idet den følger vinduets venstre hjørne. Disse kald på *setLocation*-metoden er derfor placeret i panelernes *paintComponent*-metode da, denne køres hver gang vinduet skales. Koordinaterne beregnes derved påny hver gang vinduet skales, og komponenterne kan derfor altid få de rigtige koordinater. Denne metode er brugt til alle billeder, *JComponents* og al tekst der oprettes eller tegnes.

I klassen *OptionsBasePanel* tegnes grafikken til spil-delen. Her er lagt specielt vægt på, at felternes størrelse passer til spillerens ønskede bane-størrelse og spillerens ønskede vinduesstørrelse samtidig med at bevare dens kvadratiske form, der jo ikke var tilfældet i den simple version. 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 vinduesstørrelsen ændres, kan spilleren udvide vinduet for at se spillet i en større version, der uafhængigt af vinduesskaleringen, skales lige meget på begge led.

Tegning af slangen

Udover bane-størrelse er det også muligt at vælge slangens farve. Når en farve er valgt tjekkes der i 'drawSnake'-metoden i *OptionsBasePanel*-klassen om farven har været valgt før. Hvis ikke tilføjes den nye farve til *ArrayList*en 'snakeColors' i metoden 'addSnakeColor'. Denne metode farver også alle billederne, som viser slangens dele og tilføjer den til delens egen *ArrayList*e. F.eks. er 'snakeHeadUp' en *ArrayList*e, som indeholder alle de brugte farveversioner af

'snakeHeadUp'-billedet. Farvningen af billederne sker igennem 'colorSnakeImage'-metoden, der tager et `BufferedImage` og et `Color`-objekt som parametre. Her oprettes der et `WritableRaster`-objekt, der gør det muligt at manipulere med et billedets pixel. Dette gøres ved at kopiere billedet med `BufferedImage`'s 'copyData'-metode, og parameteren null, hvilket opretter en kopi af billedets areal som en rektangel og ind i et passende `WritableRaster`-objekt. I for-løkken undersøges om den valgte pixel har farven på øjnene, da den så ikke skal farves. Dette gøres med billedets 'getRGB'-metode i et if-statement og den fundne farvekode for øjenene. Farvekoden er bestemt ved at printe alle farvekoderne for et billede af slangens hoved, og derefter finde den farvekode, som forekommer mest sjældent. I if-statementet farves slangen ved at der oprettes et array af heltal, der skal holde på R-, G- og B-værdierne for den valgte pixel. Denne defineres ved at bruge 'getPixel'-metoden på `WritableRaster`-objektet, der giver et heltals-array med størrelsen tre, der kan tildeles den nye farves RGB-værdier og derefter bruges til at farve den valgte pixel, med metoden 'setPixel', der tager imod koordinaterne og farvearrayet. Endeligt, returner 'colorSnakeImage'-metoden et nyt `BufferedImage`, der nu er farvet og kan tilføjes til kropsdelens array.

Har den valgte farve været valgt tidligere, bestemmes dens index i 'snakeColors'-ArrayListen, som kan bruges til at hente alle de rigtige farveversioner af slangens dele. På denne måde behøver billederne ikke at blive farvet en bestemt farve mere end én gang. Efter at have fundet billederne for slangen i den rigtige farve, skales alle billederne, så de passer til bane-størrelsen.

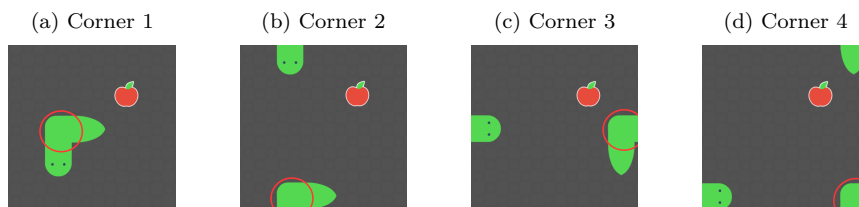
I Simpel Snake kan spilleren ud fra slangens krop se hvor han har været, men ikke hvilken vej han har bevæget sig, idet de udfyldte felter er fyldt helt ud til kanten. Slangens udseende forbedres derfor ved at vise slangens bevægelsesretning og retningsskift i hver enkel del af dens krop og generelt erstatte alle firkanterne, med mere beskrivende billeder. Dette giver seks mulige udseender for hver enkel del af slangens krop. Hovedet og enden af halen, findes hver 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 2.2). I figur 3.2a ligger de tilgrænsende felter lige under og til højre for hjørnet, men dette gælder f.eks. ikke for figur 3.2c 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 for kroppen ligger i en Array-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 *BoardBasePanel*-klassen blev der undersøgt alle de mulige situationer hvorpå slangen kan se ud. Passede den valgte kropsdels forhold til delen foran og bagved med et if-statement, blev kropsdelens billede skiftet ud med det passende billede og derefter tegnet for den del. Det har dog været meget langt, da et if-statement for ét billede af et hjørne alene tog flere linjer, men koden kunne simplificeres ved at finde et mønster i de fire if-statements. 'isSnakeCorner'-metoden returnerer en boolean 'isCorner', som fortæller om tre *Field*-objekter udgør et specifikt hjørne. Et if-statement før indeholdte alle situationer opdelt med ||-symbolet, men er nu erstattet af fire korte

if/else statements med 12 forskellige variable, der gør det muligt kun at kalde én boolean. Programmet kører 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*-metoden, bruges når slangen er placeret midt på banen (se figur 3.2a), og derved grænser op til sin foran- (front) og bagvedliggende del (front og back). Disse dele findes enten ved getColumn()+1 eller getColumn()-1 og på samme måde getRow()+1 eller getRow()-1. De to næste linjer kode i *isCorner* beskriver, slangens placering på den sidst række (altså i bunden af banen) og går gennem torussen, så noget af den ender øverst i banen (se figur 3.2b). De to næste linjer kode er på samme måde, slangens placering yderst til højre og går gennem torussen (se figur 3.2c) hvorefter noget af den ender på den modsatte side. De to sidste linjer kode er til, slangens bevægelse igennem 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 3.2d). Derved er alle hjørne-situationer gennemgået. Man kan se et mere detaljeret billede af snake-kroppen i Appendix (Bilag A).

Figure 3.2



Når spilleren færdiggør spillet enten ved at tabe eller vinde, tegnes Game Over-skærmen ved en gennemsigtig rektangel, tekst og JButtons, der giver mulighed for at gå tilbage til menuen eller spille igen.

3.3.3 Control (Styring)

Ændret siden simpel: Mute. Pause. Return. Return to Menu. Start/Play Again.

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 genvejene: (M) - mute, (P) - pause, (Esc) return to menu, (Backspace) - return og (Enter/Space) - start / play again. Control-klasserne nedarver fra KeyAdapter-klassen, der registrerer tastatur-input. Samtidig implementerer den ActionListener-klassen, der registrerer tryk på JButtons. Disse klasser tilføjer KeyListeners til *ViewFrame*-klassen, og ActionListeners til hver enkel knap, der har funktioner i ActionListenerens tilhørende control-klasse. Individuel kontrol over de forskellige knapper fås ved at sætte en unik ActionCommand til hver enkel knap, der derefter kan tjekkes for i ActionListeneren abstrakte metode 'actionPerformed'. Det er her også nødvendigt efter knappetryk at bruge 'requestFocus'-metoden på *ViewFrame*-klassen, da spillet efter knappetryk, får et nyt fokus, hvilket betyder, at tastatur-inputtet ikke opfattes af spillet. Især når *ViewFrame* oprettes, bruges 'setFocusable'- og 'requestFocus'-metoderne, efter-

som tastatur-input allerede skal være brugbart når spillet åbnes på grund af 'mute'-funktionen.

ViewFrameListener-klassen er som tidligere nævnt en global klasse, der står for kontrollen overalt i spillet. Dvs. muligheden for at slå lyden til eller fra eller vende tilbage til menuen. Når enten (M) eller lydknappen trykkes på, sættes *Audio*-klassens boolean 'mute' til det modsatte af det den i forvejen var, mens *HeaderBasePanel* (eller dens underklasser) notificeres for at opdatere lyd-ikonet. [INSERT PIC]

BoardSingleplayerListener og *BoardMultiplayerListener* indeholder metoden 'actionPerformed', der genstarter eller går ud af spillet når spilleren trykker på 'Play Again'-knappen eller 'Menu'-knappen. Klasserne indeholder også KeyEvents, som kalder på en 'move'-metode i *GameSingleplayer* eller *GameMultiplayer*, der får slangen til at bevæge sig, når spilleren trykker på tastatur-tasterne, som styrer slangen. Klasserne indeholder også KeyEvents'ene (P), der fryser eller "af-fryser" spillet og (ENTER)/(SPACE), der fungerer som 'Play Again' knappen.

OptionsListener, der er en abstrakt klasse implementeret af *OptionsSingleplayerListener* og *OptionsMultiplayerListener*, styrer knapperne i indstillings-scenerne. Før spillets start kan spilleren som tidligere nævnt - udover at vælge farve og sværhedsgrad - vælge banestørrelsen defineret med 'width' og 'height', der beskriver henholdsvis antal felter hen ad x-aksen og antal felter op ad y-aksen. 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. Da tekstfelterenes Formatter får deres caret til at sætte sig i starten af tekstfeltet selvom spilleren trykker et andet sted i feltet, implementerer *OptionsBasePanel*-klassen en FocusListener, der har de abstrakte metoder 'focusGained' og 'focusLost'. I 'focusGained' bruges SwingUtilities 'invokeLater'-metode, der sørger for at opgaver ikke udføres samtidig så brugergrænsefladen kan opdateres korrekt.

Trykkes på en af farve-knapperne eller sværhedsgrads-knapperne, optegnes den vha. JButton-metoden 'setBorder' med en tyk kant, for at vise at den er aktiv, mens kanterne på de andre knapper fjernes med 'setBorderPainted(false)'. Trykkes på en farveknop, sendes farven til *BoardSingleplayerPanels* eller *BoardMultiplayerPanels* 'setSnakeColor'-metode, der derefter bruger farven som beskrevet tidligere. Trykkes på en sværhedsgradsknap, sættes en 'Difficulty' enum i superklassen til den valgte sværhedsgrad, hvorefter denne bruges når 'Play'-knappen i underklasserne trykkes eller der trykkes (ENTER). 'Play'-knappen kalder på den implementerede abstrakte metode 'playAgain', som først tjekker om tekst-felterne for bane-størrelsen er tomme, da der så printes en fejlmelding. For ikke at gøre det for svært for spilleren at indtaste en gyldig værdi, ses der bort fra eventuelle mellemrum før, i eller efter tallene. Dette gøres med metoden 'getInput' i superklassen, der fjerner alle mellemrum. Der undersøges derefter om tallene ligger mellem 5 og 100. Hvis dette er tilfældet, sættes bane-størrelsen vha. game-klassernes 'setBoardSize'-metode, sværhedsgraden sættes med 'setTimedMovementSpeed'-metoden med et heltals-parameter afhængig af Difficulty enum-staten. Til sidst startes spillet med metoden 'start' og *ViewFrame*-klassens 'showGame' metode kaldes for at skifte options-panelet ud med spil-panelet.

3.4 Evaluering

3.4.1 Forskellige Operativsystemer

For at sikre stabilitet af brugergrænsefladen er spillet afprøvet i forskellige størrelser og på forskellige operativsystemer. Placeringen af komponenterne er ens i Microsoft Windows 8.1 og dens ældre versioner, [INSERT———]. Derimod har JButtons et lidt andet udseende på [INSEEEEEERT], 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 konstruktøren, men stadig med *setBounds*-metoden i *paintComponent*-metoden. En anden forskel i spillet, når det kører på forskellige operativsystemer, er at på Mac reagere tasterne anderledes i multiplayer. Player1 (den der styrer med piletasterne) kan godt holde en af piletasterne nede og kører hurtigere, dog hver gang dette gøres, kommer player2 til at stå stille. Hvis player2 holder en af dens control-taster nede, så fortsætter begge slanger bare i samme tempo som før. Dette er dog ikke tilfældet på Windows eller Linux. Vi havde også nogle problemer med lyden på Linux, da spillet ikke ville spille hele sekvensen fra lydfileerne, når spillet blev startet. Dette er dog kun småfejl, som er et hardware-problem.

3.4.2 Køretid

Efter der blev tilføjet en masse grafik og udvidede funktioner til Simpel Snake, kunne man på nogle computere se at spillet ikke reagerede så hurtigt, og køretiden var for stor. Dette sås også på vores endelige Avanceret Snake, hvis man satte størrelsen på banen til 100x100. Man kunne ikke holde piletasterne ned, og få slangen til at køre hurtigere. Det løste vi, ved først at lave vores slange om fra billeder til filledRectangles. Så kørte spillet normalt hurtigt igen. For at få billederne til også at fungere ved normal hastighed, ændrede vi i stedet strukturen i hvordan billeder kaldte *getScaledInstance*-metoden. Normalt kaldte hvert billede denne metode, og derefter blev sendt gennem if-statements og for-loops. Nu kalder billederne kun denne metode, når de faktisk bliver brugt, og ikke fra starten af. Dette løste problemet så køretiden var normal igen.

3.4.3 Slangens farver

Nederst i *BoardBasePanel*-klassen ligger metoden *colorSnakeImage*. Denne metode farver hver pixel i slange-billederne, så de passer med valgene fra menuen. Inden denne metode blev lavet, overvejede vi at uploade 3 gange så mange flere snake-billeder, der havde de tre sidste farver til hele slangen. Da det i forvejen er mange billeder, der bruges til slange-kroppen, ville dette være for mange billeder at uploade igen, hvis man nu kunne finde en metode til farvningen. På denne måde er farve-valg i menu'en blevet optimeret en del.

-ingen brug af jButton -> jButton -Se alle slangens led - gør den lidt forsinket, når den er lang - Tjek for gentagelser - Hard coded graphics + image + button placements vs using javas layout managers. - git til prototype hindre 4 personer i at lave noget hele tiden dropbox bedre? JButton

Chapter 4

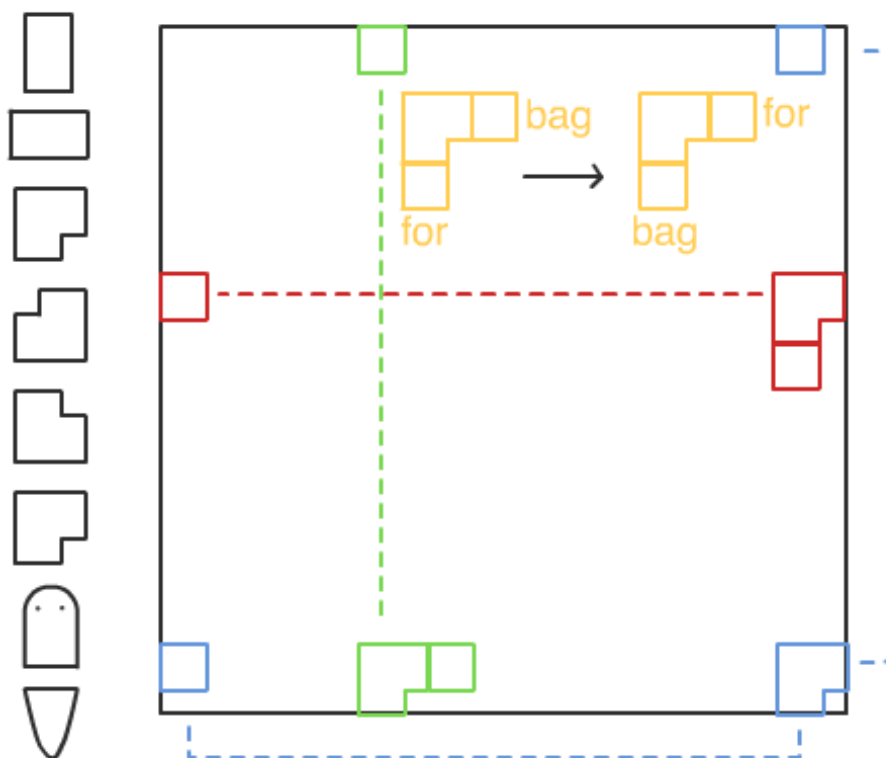
Konklusion

HELLO THIS IS CONCLUSIOSN

Appendix A

Slangens opbygning

A.1 Detaljeret oversigt af slangens opbygning



References

- [1] Wikipedia. Snake (video game), 2014. URL https://en.wikipedia.org/wiki/Snake_%28video_game%29. [Online; Tilgået 8-Januar-2015].