Projektarbejde, Snake

Anna Ølgaard Nielsen
s144437

Christian Søholm Andersen
s103080

Mathias Enggrob Boon
s144484

Van Anh Tri Trinh
s1444449

Introduktion til Softawareteknologi, 02121
Technical University of Denmark

September-December 2014

0.1 Introduktion

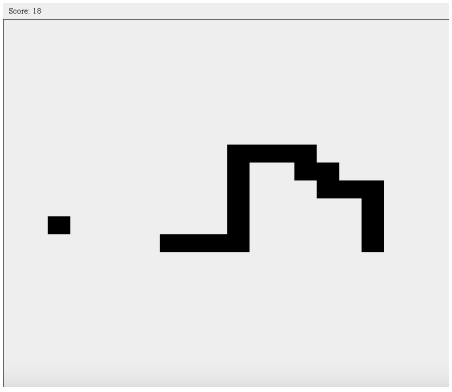
I denne rapport dokumentere vi designet, implementation og evaluering af et snake spil. Snake spillet er det velkendte spil fra for exemple Nokia 3310 [1].

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**.

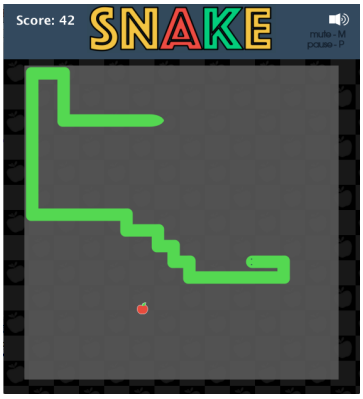
Simpel Snake er en primitiv version af spillet, hvor styring og bevægelse kun foregår vha. input fra spilleren (se figur 1a).

I Avanceret Snake er der tilføjet forskellige funktioner, som enten forbedrer brugerfladen - f.eks. tilføjelse af hovedmenu - eller ændrer spillets funktioner - f.eks. automatisk bevægelse af slangen (se figur 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, i situationer med flere forskellige muligheder.



(a) Simpel Snake



(b) Advanced Snake

Figure 1

Chapter 1

Grundlæggende

1.1 Afgrænsning

Simpel Snake er opbygget af en torrus, som er selve banen, en slange, der styres af piletasterne, og noget mad, der kan gøre slangen længere. Selve afgrænsning af denne del er defineret af projektoplægget, så det vil være bedst at læse dette for en uddybbende afgrænsning. Vi laver et resume over oplægget.

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

Slangen Slangen består af et antal sammenbundne punkter. Den har et hoved og en hale. Slangens hoved har en bevægelsesretning *op*, *ned*, *h∅jre* eller *venstre*. Gennem input fra tastetured skal det være muligt at bevæge slangen. På skærmen skal slangen visualiseres via nogle firkanter som representere slangens punkter.

Mad På banen skal der være et felt med mad, som slangen kan spise. Når dette sker bliver slangen et punkt større. Efter at slangen har spist maden, skal der sættes et nyt mad punkt på banen. Dette punkt skal blive fordelt uniformt tilfældigt på banen.

Kollision Spillet slutter når, slangen bevæger sig ind i et felt udfyldt af slangen selv. I så fald, har spilleren tabt. Man har vundet spillet, når spilleren har fyldt hele banen ud med slangens krop. Bemærk at slangen godt kan bevæge sig ind i sit hale felt, da halen også flytter sig i det samme træk.

1.2 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 *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 spillets funktioner ligger i klassen *Game* i model-pakken. *Game* får da spillets objekter fra andre klasser i *model*-pakken, f.eks. Level (banen) og Snake (slangen). Spillets tilstand ændres, når der modtages input fra control-klasserne, som bestemmer hvornår og hvordan slangen bevæger sig. *Control*-klassen opretter en keylistener, der gennem piletasterne, styrer slangen gennem banen, og derved gennem torussen. Vi har også valgt at lave to enum-klasser, *Direction*-klassen og *Action*-klassen, til at holde styr på nogle forhold gennem spillet. Det er nemmere at tilkoble en masse kommandoer og statements til hver retning og action-state frem for en integer, da man nok kommer til at kalde enum-klasserne i flere forskellige klasser, og tilkoble virkelig mange kommandoer, der skal bruges i hvert state. Så for ordnens skyld, så er det mere overskueligt med enum-klasser.

I *BoardPanel*-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 tegnes. *BoardPanel* gentegner hele spillet hver gang spillet opdateres. Først tegnes selve banen, derefter slangen og til sidst æblet. Scoren holdes opdateret i en *Score*-klasse i *model*-pakken, der observeres af *ScorePanel*. *ScorePanel* ligger som et panel øverst over selve spilletpladen, og har kun til formål at holde styr på scoren.

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

Immutable klasser - Food, Field og Board

Klasserne Food, Field og Board er meget små. De har alle samme kun to værdier i sig (enten en coloum og row eller en dimension). Vi har derfor designet klasserne sådan at de ikke kan ændres efter at de er blevet oprettet. Dette sikre os at vi kan returnere vores game board i en getter og være 100% sikre på at undefra kommende klasse ikke ændre det. Java har ikke et constant begreb. Hvis du har en reference til et object og objected exposer sine interne variable, kan man ændre på klassen. Nogengange kan det godt være svært at resonere omkring koden, hvis alle objecter vilkårligt kan ændre hinanden. Immutable klasser sikre at man ikke kan ændre klassers start værdier.

1.3 Implementering

1.3.1 Control (Styring)

I Simpel Snake er der ikke brug for nogen menu, og det er derfor muligt at holde styringen af spillet meget simpelt. Ved at definere de fire piletaster i *Control*-klassen, en funktion til at sikre at der ikke bevæges i modsat retning, samt en funktion der gør det muligt at bevæge sig som en torrus og "gå igennem vægge og om på den anden side af verden", er alle styringkrav til Simpel Snake opfyldt. Resten af spillet håndteres da i model-koden.

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. Enumklassen *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.

1.3.2 Model

Spillets model består af en række klasser, der tilsammen udgør selve spillets funktioner og objekter. Klassen *Field* bruges til at definere et et punkt, som repræsenterer et felt i banen. Den fungerer på samme måde som *Point*-objekt-klassen, hvor koordinatsystemet starter oppe i venstre hjørne. Forskellen 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, da to-dimensionelle arrays også er opdelt i rækker og kolonner, hvor den første værdi repræsenterer rækker, mens den anden værdi repræsenterer kolonner.

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.

Vi har valgt, at oprette spillepladen i *Game*-klassen og ikke have en klasse, der kun beskriver banen for sig selv. Derved er de forskellige spilelementer defineret i deres egne klasser, så f.eks. *Food* er en klasse for sig selv, *Snake* er en klasse for sig selv i stedet for at være objekter i *Game*. Hver klasse har de funktioner, der er relevante for dem, f.eks. *getPosition* for at give deres 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, det 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 hoved og hale oprettet, hvor hovedet bliver placeret i banens centrum. Halen placeres i kolonnen til højre for på samme række.

I *move*-metoden benyttes Action-klassen, der er oprettet som en enum, og betegner slangens handlinger: om slangen spiser et æble, er død eller bevæger sig (hhv. *EAT, KILL, MOVE*). I *move*-metoden undersøges der, om der er mad på samme felt som hovedets nye placering. Er dette tilfældet, bliver Action-statementet til *EAT*. Hvis ikke, undersøges der om den nye placering for hovedet allerede indeholder slangens krop. Gør der det, 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 til sidst, 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 ved at slangen ikke længere kan bevæge sig. 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 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 indlejret for-løkke, der løber gennem alle række og kolonner, hvorefter et tilfældigt element i listen vælges som æblets position.

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

1.3.3 View (Brugergrænseflade)

Brugerfladen er samlet i klassen *GameView* der forlænger *JFrame*. I *GameView* findes kun en constructor, hvor der oprettes et *ScorePanel*-objekt, der placeres øverst i vinduet, og et *BoardPanel* objekt, 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 implementerer *Observer*. Klassen har en *update*-metode, der gentegner, hver gang noget ændrer sig i de klasser i model-pakken, der forlænger *Observable*. *paintComponent*-metoden benyttes til at tegne score-teksten. Selve panelet oprettes i constructoren.

BoardPanel forlænger også *JPanel* og implementerer *Observer*, for at den også kan registerere ændringer i model-klasserne. *BoardPanel* består af en række draw-metoder, samt en *paintComponent*-metode, der kalder på draw-metoderne for at tegne alle spillets komponenter på spillets bane og spillets bane selv. I *drawSnake*-metoden bruges positionerne for slangens felter til at tegne slangen. Størrelsen for et felt udregnes i *getWindowRectangle*-metoden, og afhænger af vinduets størrelse og antallet af banens felter. Når feltets størrelse er udregnet, tegnes en rektangel med feltets størrelse på alle felterne i slangens positions-ArrayList.

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 figur **??** 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.

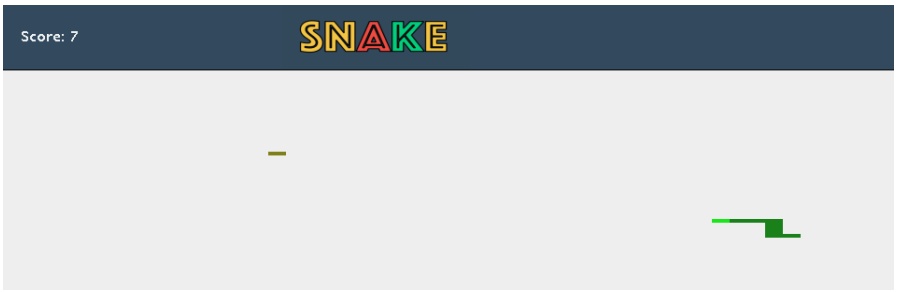


Figure 1.1: Exemple på udstrækte felter.

1.4 Evaluering

Arbejdsproces

Formålet med projektet var at starte med 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 et, til at starte med, 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. Gruppemedlemmet afgør selv, hvordan en funktion skal designes og implementeres, men sikrer at implementationen 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 dobbeltarray. F.eks. et array [10][5] vil 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.

I sidste ende valgte vi denne metode fra på grund af den ulempe, at den skal sende et todimensionelt array med 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. Vores anvendte 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.

1.4.1 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 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. Det blev bestemt at holde scoren som klasse for sig selv, så den kun blev tegnet om, når der skete ændringer.

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. *GenereateFood*-metoden og hvordan banen skulle opstilles. Vi satte ikke *move*-metoden ind i *Game*-klassen frem for *Snake*-klassen, da den ikke som sådan skal bruge nogle informationer fra andre klasser. Den skal bare bevæge sig. Grunden til vi valgte, at lægge andre klassers metoder ind i *Game*, er både fordi, det er smart at have det meste samlet et sted, og ikke behøve at hente alt for mange get-metoder fra de andre klasser. I den avancerede del af snake, ender vi med at gøre det modsatte. Vi simplificerer vores *Game* klasse fuldstændigt, og gør den i stedet til en super-klasse med abstrakte metoder. Derfra nedarver to andre klasser fra den, og på den måde er metoderne bare uddeligeret til de to klasser. Man kan også sige, at *Game* stadig består af mange funktioner, siden det er en superklasse til to store under-klasser.

Chapter 2

Advanced

2.1 Afgrænsning

Den advancerede 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 stærrelse 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 advancerede 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.

2.2 Design

2.2.1 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 *ViewFrame*s constructor, hvorefter den som KeyListener tilføjes til alle panelerne. Denne klasses funktioner er globale, da den står får '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*.

2.2.2 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 eksemple 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 eksemple 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 resultere 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 seperate single og multiplayer game klasser som arver fra en abstract game klasse. Fordelen ved dette er at vi har en seperation 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!!!!!

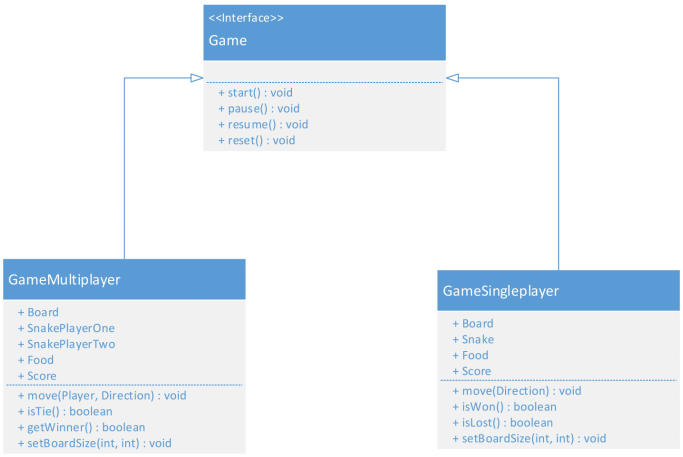


Figure 2.1: Overview of our multiplayer design.

2.3 Implementering

2.3.1 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 også dens [.....]

Multiplayer

2.3.2 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 spilscenen 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 adang 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 skaleres. Koordinaterne beregnes derved pånyhver gang vinduet skaleres, 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 kvadrate 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 vinduesstørelsen ændres, kan spilleren udvide vinduet for

at se spillet i en større version, der uafhængigt af vinduesskaleringen, skaleres 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 ArrayListen 'snakeColors' i metoden 'addSnakeColor'. Denne metode farver også alle billederne, som viser slangens dele og tilføjer den til delens egen ArrayListe. F.eks. er 'snakeHeadUp' en ArrayListe, 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 BufferedImages '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, skaleres 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 2.2a ligger de tilgrænsende felter lige under og til højre for hjørnet, men dette gælder f.eks. ikke for figur 2.2c 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

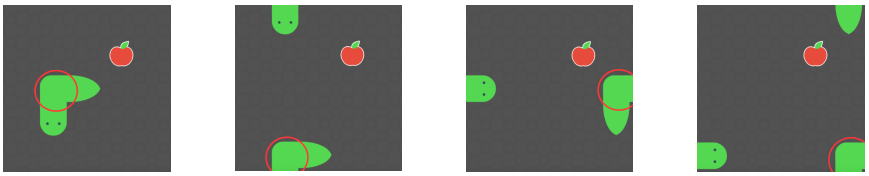
bytte plads, findes der altså otte situationer for et enkelt hjørnestykke. I alt tjekkes der derfor - for kroppen alene - 34 mulige forhold mellem et stykke og dens to tilgrænsende stykker.

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 2.2a), 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 lijner kode i *isCorner* beskriver, slangens placerering på den sidst række (altså i bunden af banen) og går gennem torussen, så noget af den ender øverst i banen (se figur 2.2b). De to næste linjer kode er på samme måde, slangens placering yderst til højre og går gennem torussen (se figur 2.2c) 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 2.2d). Derved er alle hjørne-situationer gennemgået. Man kan se et mere detaljeret billede af snake-kroppen i Appendix (Bilag A).

Figure 2.2

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



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.

2.3.3 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 gen-

vejene: (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, eftersom 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 lydikonet. [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 tastaturtasterne, som styrer slangen. Klasserne indeholder også KeyEvents'ene (P), der fryser eller "afryser" 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 helholdsvis 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 farveknap, 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 sidsts startes spillet med metoden 'start' og *ViewFrame*-klassens 'showGame' metode kaldes for at skifte options-panelet ud med spil-panelet.

2.4 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 - 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? Mac-problems Jbutton controls lack

Chapter 3

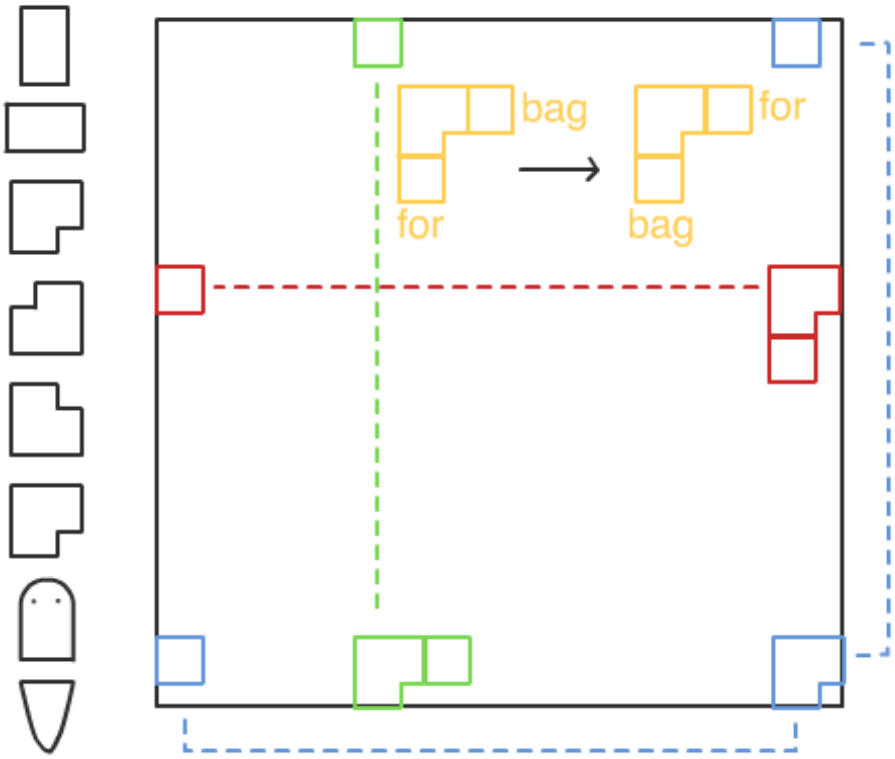
Konklusion

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].