Introduktion til Softwareteknologi - Dam

Joel A. V. Madsen (s194580)
Carl Alexander Jackson (s194585)
Victor Rodrigues Andersen (s194577)
August Lykke Thomsen (s194611)

19. januar 2020

Arbejdsfordeling.

August stod for den primære implementation af 3D.

Joel stod for den primære implementation af AI samt grundlæggende funktionalitet af Avanceret Dam.

Carl stod for den primære opbygning og design af scener.

Victor stod for den grundlæggende programmeringsdel og mindre funktionaliteter.

Indhold

1 Indledning

(Skrevet af alle)

I denne rapport beskrives designet, tankegangen og implementeringen af programmerne SimpelDam og AvanceretDam. Den basale del af opgaven stod på at udarbejde programmet SimpelDam, som er en forsimplet version af dam spillet, hvor kun to brikker benyttes, og rykkes efter nogle sær regler. Efterfølgende blev SimpelDam brugt som grundlaget for programmet AvanceretDam, som, trods dets navn, er en normal version af brætspillet dam. I denne del af opgaven var der fri fortolkning på hvilke udvidelser og modifikationer hver gruppe ville indføre. De modifikationer vi har valgt at inkorporere er en 3-dimensionel version, en kunstig intelligens med 3 sværhedsgrader, en tids begrænsende version, en menu samt lyde.

2 Afgrænsning

(Skrevet af Victor)

I dette afsnit bliver kravene for begge versioner af spillet redegjort, samt hvilke værktøjer vi har brugt til at understøtte udviklingen af programmet.

2.1 Værktøjer

Programmet er udviklet i programmeringssproget Java med biblioteket JavaFX hvori Eclipse er brugt som IDE til at redigere koden. Rapporten er skrevet skrevet i LaTeX. For visualisering af klasseopbygnignen og diagrammer er der benyttet Draw.io.

Vi har valgt at bruge en fælles Google Drive mappe, til at gemme de forskellige versioner af programmet. Vi overvejede brugen af GIT, men kom frem til at det vil tage et stykke tid at sætte det op, og vi forventede at der ville forekomme problemer mht. merge konflikter, som ville kræve unødvendig tid rette op på.

2.2 Simp-Dam

Følgende krav blev opstillet ud fra opgave beskrivelsen for Simp-Dam:

- Spillet spilles på et kvadratisk bræt med $n * n$ felter hvor $n \in \{3, \ ... \ , 100\}$.
- n skal kunne tages som kommandolinjeparameter.
- Spilles af to spillere med to forskellige brikker.
- Brikkerne skal i starten placeres på modsatte kanter af brættet.
- Spillerne laver træk på skift til et nabofelt på skråt hvis den er ledig.
- Hvis modstandernes brik står på nabofeltet skråt for eget felt, og det næste felt efter modstanderen er ledig, så bliver modstandernes brik sprunget over, og elimineret fra spillet, spilleren hvis brik er tilbage har vundet.

2.3 Avanceret Dam

Versionen Avanceret Dam bygger på Simp-Dam, men med ekstra krav til udvidelser, som vi synes bidrager til brugervenligheden og muligheder for diversitet i forhold til spillet.

Kravene til Avanceret Dam, i den prioriterede rækkefølge, er:

- Flere brikker (samme antal som almindelig dam).
- Konge brik, hvis en brik når til modstanderens ende.
- Restriktion på brikkernes bevæglelse, sådan at brikker kun rykker "fremad"afhængigt af farve, og i alle skrå retninger hvis det er en konge brik.

- Elimination af flere brikker hvis muligheden er der.
- Rykke muligheder, når man trykker på en brik bliver de mulige træk fremhævet.
- En Menu.
- Indstillinger hvor man kan ændre om man vil have lyd på, timer tid, spille mod AI eller 3D versionen af spillet.
- Muligheden for at spille mod en AI som erstatter en spiller.
- En scene der viser hvem har vundet.
- En 3D version af spillet.
- Animation af brikkerne når et træk udføres.
- En tidsbegrænset version, hvor hver spiller har en timer som tæller ned så længe det er deres tur (som blitz skak).
- Lydeffekter.

3 Design

(Skrevet af Victor)

I denne del af rapporten kommer vi ind på spillets design med hensyn til design af koden og brugergrænsefladen.

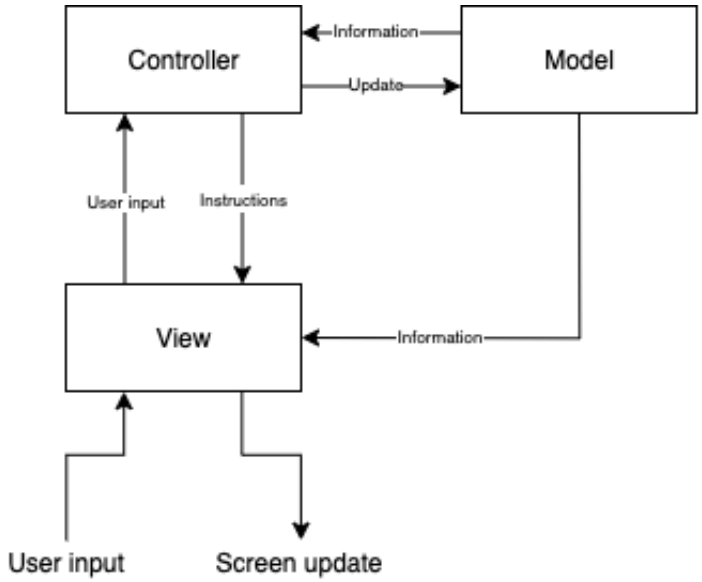
3.1 Arkitekturen

Her vil vi beskrive kodestrukturen og hvordan programmet hænger sammen.

3.1.1 Model-View-Controller (MVC)

Vi valgte at bruge MVC modellen til at bygge vores spil, fordi vi mener at det gør koden meget mere overskueligt at arbejde med og giver et bedre overblik over hvordan tingene hænger sammen og interagerer med hinanden.

Vi har sat vores MVC op sådan, at vi under Model har klasserne `Tile` og `Piece`, som indeholder egenskaber og data, der kan bliver modificeret af vores controller. Vores Controller står for at styre selve spillet, og udføre de opgaver programmet får når spilleren interagerer med spillet. Vores View står for at opdatere input fra controlleren til spillet, og står ydermere også for interaktionen mellem spiller og program, i form af eventhandlers.



Figur 1: Viser sammenhængen mellem module, view og controller.

3.2 Brugergrænseflade

(Skrevet af Carl)

3.2.1 Simp-Dam

Den grafiske brugergrænseflade i vores spil `SimpDam`, består blot af et `GridPane`, hvor hvert felt er fyldt med en såkaldt `Tile`, en klasse der forlænger klassen `StackPane`. Vi mente at det var nemmest at lave spillet på denne måde, da hvert felt dermed vil være nemmere at tilgå, mens vi stadig har gode redigeringsmuligheder for hver `Tile`. Selve brikkerne kaldes `Pieces`. Klassen `Piece` forlænger blot klassen `Circle`. På denne måde tegner vi en brik (en cirkel, i sort eller hvid) på de nødvendige felter.

3.2.2 Avanceret Dam

Selve spillet Avanceret Dam består, på samme måde som før, af et `GridPane`, hvor hvert felt er fyldt med `Tiles`, og de forskellige `Tiles` enten er tomme, eller er fyldte med hhv. `pieceWhite` eller `pieceBlack`.
Avanceret Dam har har flere scener, der skiftes imellem. Disse scener hedder `mainMenu`, `nameMenu`, `settingsMenu`, `playScene` og `victory`.

Ved opstart vises scenen `mainMenu`. Denne scene har et `BorderPane`, kaldet bp1 som root layout. I midten af dette BorderPane har vi en `play`-knap, og i bunden af `bp1` har vi en `settings`-knap. `Play`-knappen har en `eventhandler` bundet til sig, denne skifter scene til `nameMenu`. Alle vores knapper har `eventhandlers`, der har flere funktioner, deres primære funktion er at de skifter mellem de forskellige scener. Blandt disse knapper har vi `settings`-knappen, som skifter scene til `settingsMenu`.

Scenen `settingsMenu` har også et `BorderPane` (`bp5`) som root layout.
I midten af `bp5` har vi placeret en `VBox`, dette har vi gjort for at holde de forskellige `CheckBoxes` adskildt. Vores `CheckBoxes` er navngivet hhv. `cb1`, `cb2` osv.. `cb1` bestemmer om lyden er slået til eller fra. `cb2` ligger sammen med tekstfeltet `tf3` i en `HBox`, dette er så man kan til- eller fravælge en `timer`, samt skrive et antal sekunder, som man vil sætte af til hver spiller. cb3 bestemmer om 3d er slået til eller fra. cb4 bestemmer AI er slået til eller fra. Til denne hører der en "slider"med ter indstillinger, Easy, Medium og Hard. De styrer antallet af rekursioner som AI'en forudsiger, det er hhv. 2, 4 og 6 rekursioner.
I bunden af `bp5` har vi placeret en `back`-knap, ved tryk på denne, returneres brugeren til scenen `mainMenu`.

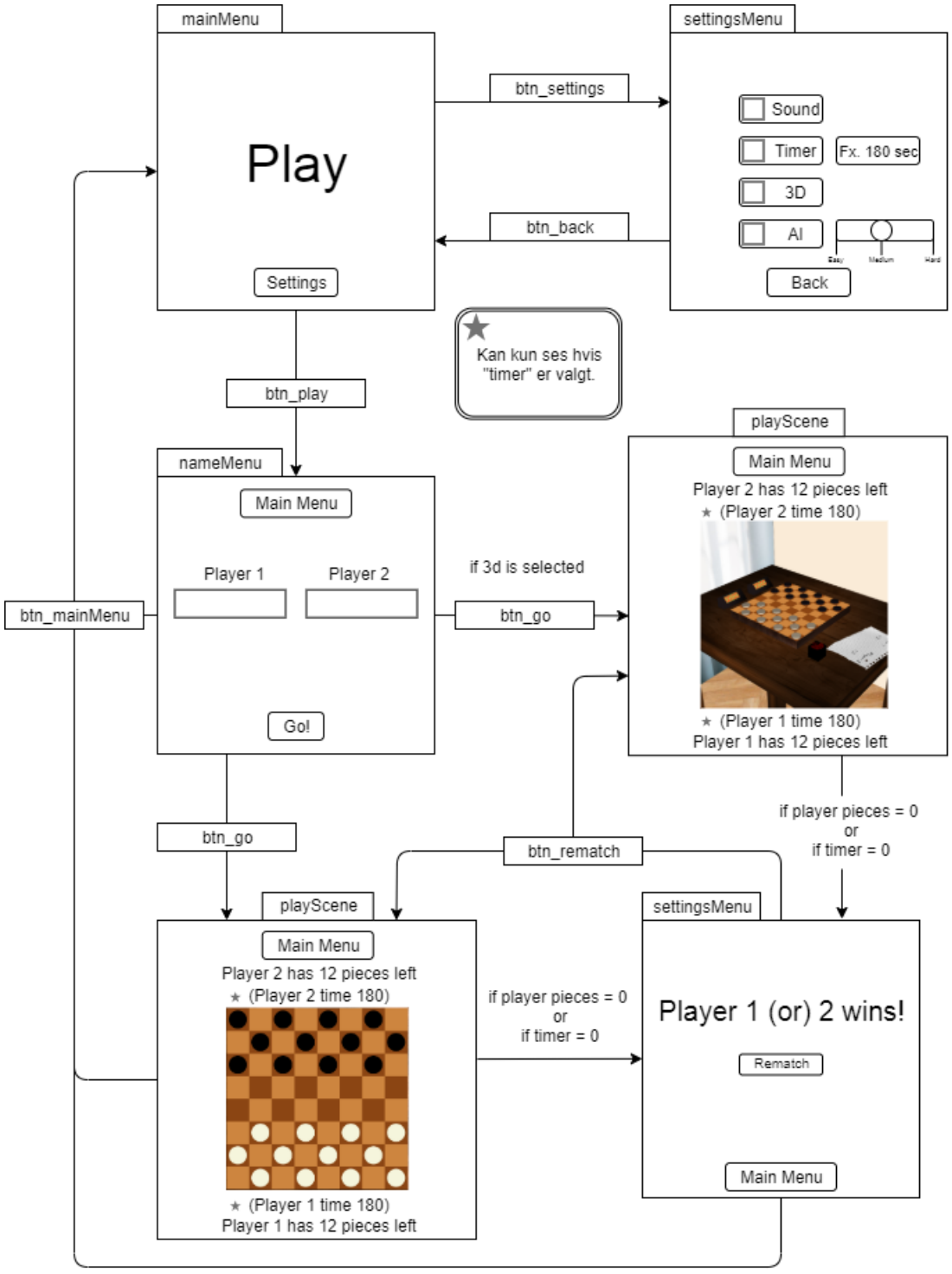
Scenen `nameMenu` har også et `BorderPane` (`bp2`) som root layout.
I toppen er der paceret en Main Menu-knap, der tager brugeren tilbage til scenen mainMenu. I både højre og venstre side er der placeret en `VBox`, i hver af disse er der både et `Label` og et tekstfelt. Disse er beregnet til at tage hhv. spiller 1 og 2's navne. I midten af bp2 er der et label med teksten "vs". I bunden af `bp2` har vi en `Go`-knap, denne har mange funktionaliteter, men fra brugerens synsvinkel, bliver man blot taget videre til spille scenen `scene`.

Scenen `playScene` har også et `BorderPane` (`bp3`) som root layout.
I midten af dette er der placeret det tidligere nævnte `GridPane`. Derudover har vi tilføjet nogle hjælpsomme `Labels` til Avanceret Dam. Dvs. at i toppen og bunden af `bp3` er der `Labels` der fortæller spilleren hvor mange brikker han/hun har tilbage, samt hvor lang tid han/hun har tilbage (hvis denne funktion er tilvalgt i `settingsScene`). Dette er gjort ved at lægge disse `Labels` ind i en `VBox`. I den `VBox` der ligger i toppen af `bp3` er der tilføjet den samme `Main Menu`-knap fra før.

Scenen `victory` har også et `BorderPane` (`bp4`) som root layout.
I midten af dette er der placeret en `VBox` der indeholder et `Label`, der opdateres alt efter hvilken spiller der har vundet, derudover er der en `rematch`-knap, der starter spillet forfra med samme startbetingelser, f.eks. tid og navne, men hver spille har skiftet side. I bunden af `bp4` er den samme `Main Menu`-knap som tidligere nævnt. Denne har samme funktion som før.

`Go`-knappen indsamler data fra tekstfelterne, og gemmer dem som Player 1 og Player 2. Derudover tjekker den hvilke `CheckBoxes` der er valgt, og ændrer værdien af de `booleans` der bestemmer om hhv. lyd, timer, 3d og AI er slået til eller fra. Til sidst kører den metoden `start(stage)`. Ved opstart vil `start(stage)` gå til `mainMenu`. Dette har vi sørget for vha. et `if-statement`, der siger at kun hvis `p1 name` og `p2 name` er `null`, skal start(stage) gå til mainMenu. Eftersom Player 1 og Player 2 som standardindstilling altid hedder Player 1 og Player 2, hvis der ikke er skrevet noget i felterne, vil `start(stage)` kun føre til `mainMenu` ved opstart af programmet. Dermed vil `start(stage)`-metoden der er bundet til `Go`-knappen føre spilleren til `mainScene`, denne fungerer derfor som en "reset"funktion.

`Main Menu`-knappen er den samme instans i de tre forskellige scener hvor den bruges. Det virker nemmere at rykke den samme knap fra scene til scene, i stedet for at duplikere knappen, og dermed have flere forskellige knapper med samme funktion.



Scene forløb

(Skrevet af August)

Vi har i vores afgrænsning lavet et krav, at lave en version af spillet i 3d. Vi har valgt at designet skulle være at man spiller, rundt om et bord, i et rum. Implementering af 3D består hovedsageligt af de 8 klasser `Table`, `Skybox`, `Sp3D`, der ligger i pakken `view`, `CamControl`, der er i pakken `controller`, `Piece3d`, `Crown`, `ButtonRotate` og `Timer` i pakken `model`.

Klassen `Table()` indholder den statiske metode `makeTable()` der returnerer en `StackPane`. Denne metode laver kassen som brættet skal ligge på, bordet, ure til at se hvor lang tid man har tilbage, en knap til at sætte kameraets stilling, og et papir der holder styr på, hvor mange brikker hver spiller har. Derefter sætter den bordet, papiret, urene, knappen og kassen, sammen i en `StackPane` med de rigtige koordinater, som den til sidst retunerer. Her bliver hjælpemetoderne `makePaper()` og `makeLeg()` brugt, som klassen også indeholder. Og metoden makeTimer() og makeButton(), fra klasserne `ButtonRotate()` og `Timer`

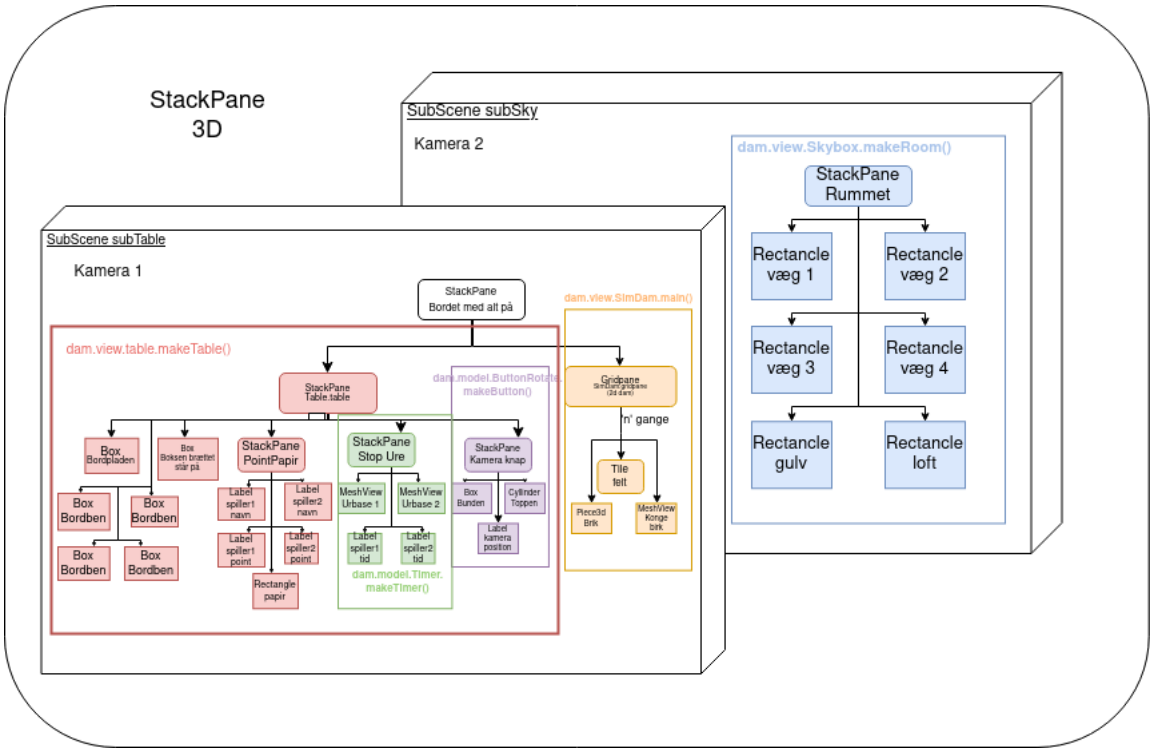
Klassen `Skybox`, laver rummet uden om bordet, den består ligesom klassen `Table` af en hovedmetode `makeSkybox()` der returnerer en StackPane, denne retunerede stackpane indeholder, 6 rektangler med den rigtige textur, som skaber et rum.
Klassen `CamControl` indeholder en void funktion `setCamera()`, der der tager 2 `SubScenes` og sætter et passende kamera på hver, i funktion bliver der også til en af scenerne givet en handler, til tastatur tryk, som styre kameraerne.
Der er også lavet en funktion som drejer brættet til, så spillerens hvis tur det er for, kommer foran sine brikker.

Klassen `Sp3D` forbinder de 3 klasser sammen. Det gør den med en funktion, der returnerer en `StackPane`, som indeholder 2 scener, den ene scene indeholder det store rum, ved brug af funktionen `Skybox.makeSkybox()`, den anden scene indeholder, bordet og det 2d `GridPane` vores spil kører på. Kamera klassen bliver brugt til at sætte kameraerne på disse scener. Den retunerede StackPane, bliver sat i center af, en BorderPane, i vore hoved spil scene, hvis 3d er slået til.

Klasserne `Piece3d` og `Crown`, indeholder 3d modellerne for brikkerne, Piece3d for de normale, og Crown laver, brikken for krone brikken. Den normale 3d prik er en udvidelse af en cyllinder, fra javfx biblioteket. Kronebrikken bliver lavet, som et `MeshView`, som bruger et mesh vi selv har lavet.

Knappen er en blanding af en `Cyllinder`, `Box` og en `label`. Knappen skal kunne skifte mellem kamera vinkler, så man kan se fra oven, fra siden, og lige på, og kamereat drejer til hvis spiller tur det er.

Urene, er lavet af et selvlavet MeshView, med et Label på, som bliver opdateret med en timer, hvis man har sat tid til.
Vi har lavet et diagram over den StackPane makeSp3D returnerer, med hvilke typer objekter der kan ses, og hvilke funktioner der skaber dem.



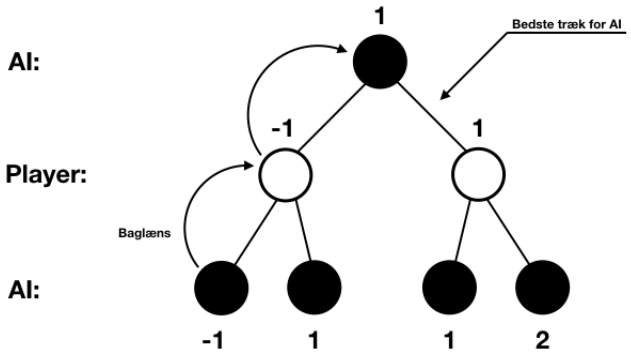
Figur 2: Diagram over noderne i 3d StackPane, farverne viser hvilken metode der er blivet brugt til at lave noden

3.3 Kunstig Intelligens

(Skrevet af Joel)

Til design af den kunstige intelligens, valgte vi at benytte en algoritme kaldet MiniMax. I korte træk forsøger denne algoritme at lave det bedste mulige træk, ved at maksimere sit eget træk, og minimere modstanderens. Algoritmen fungerer rekursivt, og virker ved at forudsige spillet X antal træk forud.

Den kunstige intelligens undersøger hvilke af dets brikker der har mulighed for at lave et træk, herefter laver den alle de mulige træk, for alle de mulige brikker. Den kunstige intelligens tjekker dernæst hvilke mulige træk modstanderen kunne foretage, til hver af de træk den selv har taget, og fortsætter derefter med at tjekke hvilke træk den selv kan foretage, i forhold til modstanderens træk. Denne cyklus fortsætter de X antal gange.
Når den kunstige intelligens har tjekket de X antal træk frem, vil den tildele den tilhørende spilplade en score, som er beregnet ud fra forskellen mellem den kunstige intelligens brikker, og modstanderens brikker. En negativ score opfattes som godt for modstanderen og dårligt for den kunstige intelligens, en positiv score opfattes modsat. Efter tildelingen af score kører algoritmen "baglæns", fra bunden og op, hvor den skiftevis tjekker det bedste mulige træk for den kunstige intelligens og dernæst for modstanderen, indtil den når til de første mulige træk. Det fungere altså som et skifte af tur, hvor



Figur 3: Viser udregning af det bedste træk for den kunstige intelligens, ved 3 rekursioner.

modstanderen går efter at få den laveste score, og den kunstige intelligens går efter den højeste score. Den kunstige intelligens vil tage det træk, med den bedste score. Hvis der er flere træk, tildelt samme score, udtages et tilfældigt af de træk. Den kunstige intelligens antager at modstanden spiller optimalt fra dens synsvinkel.

For nemmere forståelse kan algoritmen i den kunstige intelligens illustreres som et træ, der forgrener sig for hvert mulig træk. Udregningen af forudsigelse og bedste træk, for en brik, kan ses på illustrationen nedenfor. Vi antager at hver brik kun har to mulige træk, for overskueligheden af illustrationen. På illustrationen køres der tre forudsigelser. Den implementerede kunstige intelligens tager højde for eliminationer, bevægelse og kongebrikker. Combo eliminationer bliver ikke medtaget i udregningen i nuværende version.

4 Implementering

4.1 SimpDam

(Skrevet af Joel)

Programmet SimpDam er en grundstruktur for spillet dam, denne version af spillet består blot af to brikker, som i modstrid til det normale dam, kan rykke på alle diagonaler ved start af programmet. Spilpladen har størrelsen $n \cdot n$, hvor $n \in \{3, \ ... \ , 100\}$, brikkerne er placeret i hvert to af hjørnerne af spilpladen, hhv. række 0 kolonne n - 1 og række n - 1 kolonne 0.

(Skrevet af Victor)

Vores hovedklasse som indeholder `main()` metoden og som står for at køre selve spillet har vi kaldt for `SimDam`. Først har vi et heltal `n` som definerer hvad dimensionerne af brættet skal være. Værdien af `n` er så det argument brugeren taster ind i terminalen når kommandoen for at køre spillet bliver tastet ind.

For at bygge brættet valgt vi at bruge `GridPane` og udfylde den med `Tile`, som er en klasse vi har lavet for at gemme og holde styr på data fra hver felt og tilføje og fjerne brikker, der nedarver fra StackPane. Hver `Tile` har en x og en y som er dens koordinater i `GridPane`, en boolean for at se om den skal være mørk eller lyst, og en variabel `state` som kan være 0, 1, eller 2. Hvis `state == 0` så er der ingen brikker på den Tile, hvis `state == 1` så er der en sort brik på den Tile og hvis `state == 2` så er der en hvid brik på den Tile. For at danne brikker har vi lavet en klasse `Piece` som nedarver fra `Circle`, og har feltet `col` som gemmer hvilken farve den har. Både `Tile` og `Piece` er en del af Model. Med det ligger vi så en brik i hver ende af brættet og kalder Controlleren.

Det Controlleren så gør er at den starter med en `for each` løkke der kører gennem GridPanet og laver de nødvendig ændringer afhængig af det spilleren gør. Lige efter det bruger vi et `MouseEvent` som udføres når spilleres trykker på en Tile med musen. Det første tilfælde den går igennem er hvor der ikke er valg på nogle brikker i forvejen og spilleren trykker på en Tile. Når det sker så defineres der værdier for variablerne `selx` og `sely` som gemmer koordinaterne for den Tile spilleren har valgt, og gennem andre variabler gemmer de forskellige komponenter som den Tile har; farve, brikken og selve Tilen. Derefter når spilleren trykker på en anden Tile, defineres der værdier for variablerne `xx` og `yy` som gemmer koordinaterne for den anden Tile spilleren har trykket på. Med de værdier tjekker Controlleren så om forskellen mellem `selx` og `xx`, og `sely` og `yy` er 1, og om den anden Tile ikke indeholder en brik. Hvis de betingelser er opfyldt så bliver metoden `swap_tiles()` kaldt med den gamle Tile og koordinaterne til den nye hvor brikken skal rykkes til.

Det metoden gør er at den stiller den gamle Tile til den almindelige tilstand, dvs. uden brik og `state == 0`. Derefter bliver den nye Tile opdateret ved at tilføje den brik der blev gemt når spilleren valgte en brik, og give den rigtige `state` alt afhængig af hvilken tur det er. Efter metoden er så færdig med at udføre sine opgaver, bliver værdien af `turn` ændret og spillet afvænter modstanderen

Det andet tilfælde Controlleren undersøger er om forskellen mellem `selx` og `xx`, og `sely` og `yy` er 2, og om den anden Tile ikke indeholder en brik. Hvis de betingelser er opfyldt, tjekker metoden så om modstanderen har en brik liggende mellem de to Tile der blev trykket på, og hvis alle de betingelse er opfyldt så udføres det et drab af modstanderens brik. Dvs. at spillerens brik rykkes to

felter på skrå og modstanderens brik bliver fjernet fra det felt den var før og fra spillet.

Til sidst ind i Controlleren udover metoderne `Control()` og `swap_tiles()` har vi også en metode `getNodeFromGridPane()` tager et GridPane og to koordinater som parameter og returnere så den `Node` der befinder sig i de koordinater.

Til sidst i `SimDam` bruger vi `ColumnConstraints` og `RowConstraints` på GridPanet for at have at alle felter har samme størrelse og er firekantede så længe vinduet også er firekantet, og vi definerer størrelsen på GridPanet, sætter scenen og viser `stage`

4.2 Avanceret Dam

(Skrevet af Victor)

Det første vi implementeret i Avanceret Dam er at brættet bliver udfyldt med 3 rækker af brikker for hver spiller, hvor hver brik placeres på de felter som opfylder `i + j % 2 == 0` og brikkerne beholder samme farve som i SimpDam. Dette udføres ved brug af et `for` løkke som kører gennem alle de Tiles GridPanet har, og tilføjer en brik til den Tile og opdaterer dens `state`.

Efter det lavet implementeret vi så også et ekstra tilfælde som `Control()` undersøger, hvor den tager hensyn til om spilleren har fortrudt sit valg af brik. Dvs. at spilleren har trykket på en brik, men vil ikke bruge den alligevel. Så kan brugeren godt trykke på en anden brik og bruge den i stedet.

4.2.1 Trækbetingelser og kongebrik

Derefter begrænser vi de mulige træk mht. de almindelige damregler. For at begrænse hvilke træk hver spiller må udføre, blev der kun ændret på `if` betingelserne fra `SimpDam` således at de sorte brikker (`tile.state == 1`) kun må rykke til y + 1 og de hvide brikker (`tile.state == 2`) kun må rykke til y - 1 så længe det ikke er en kongebrik.

For at lave kongebrikken giver vi en `boolean` værdi `king` for hver Tile som definerer om der er en kongebrik i den Tile. En brik bliver så omdannet til en kongebrik hvis den rammer den modsatte side af brættet; række n - 1 for de sorte brikker, og række 0 for de hvide brikker. Så snart det sker bliver `king` sat til `true`, og brikken udfyldes af et billede af en krone. Træk begrænsningerne for kongebrikken er de samme som fra `SimpDam`, så de betingelser genbruges, udover at Tile skal have en kongebrik (`king == true`). Så snart en der udføres et træk med en kongebrik, sættes det nye `Tile.king = true` og den gamle til `false`.

4.2.2 Combo-drab

(Skrevet af Joel)

Vi har valgt at implementere reglen om mulighed for drab af flere af modstanderens brikker. Til implementation af dette har vi en metode `check_neighbor()` som danner en `ArrayList` af naboerne for feltet brikken lander på, efter det første drab. ArrayListen af naboer bliver derefter tjekket igennem, og hvis brikken har mulighed for at eliminere endnu en af modstanderens brikker, så vil et

klik på et muligt combo felt lede til dette, det mulige felt bliver fremhævet som gult. I vores version af avanceret dam har vi valgt at combo drab skal udføres til ende.

4.2.3 Kunstig intelligens

(Skrevet af Joel)

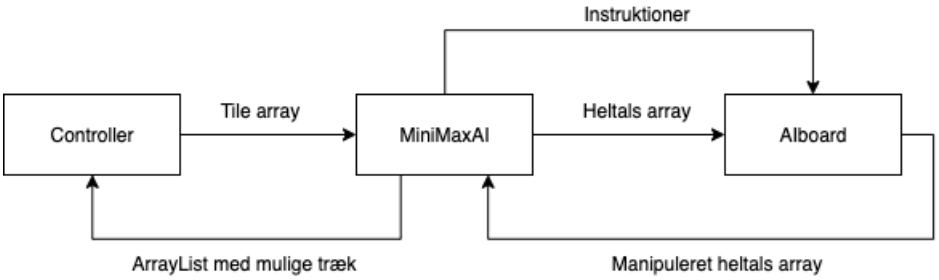
Den kunstige intelligens er implementeret som en klasse, kaldet MiniMaxAI, og har en tilhørende hjælpeklasse kaldet AIboard. Den kunstige intelligens bliver kaldt igennem vores Controller, efter hvert færdige træk den anden spiller har foretaget.

MiniMaxAI's konstruktør tager et Tile array som parameter. Det Tile Array bliver derefter omdannet til et to dimensionalt heltals array, der indeholder hvert Tiles state, hvor et frit felt er angivet som 0, sort brik som 1, hvid brik som 2, sort konge som 3 og hvid konge som 4. Heltals arrayet sendes nu til AIboard, som indeholder hjælpe metoder, til manipulation af det to dimensionelle heltals array.

AIboard indeholder følgende metoder:

1. **getBoard()** - returnerer et to dimensionalt array, som repræsenterer det manipulerede spilbræt.
2. **getState(x, y)** - returnerer en briks state, givet fra et x, y koordinat.
3. **move(x, y, x1, y1)** - rykker en brik fra x, y til x1, y1 på heltals arrayet.
4. **remove(x, y)** - fjerner en brik på en given x, y koordinat.
5. **printBoard()** - printer en visualisering af det to dimensionelle array til konsollen.
6. **eliminate** - rykker en brik fra x, y til x1, y1, og fjerne brikken imellem, hvis der skulle være en.

Efter at brættet er blevet manipuleret i AIboard sendes det tilbage til MiniMaxAI, hvor der skabes en ArrayList med mulige træk, denne ArrayList sendes til Controlleren, som udvælger et træk og udfører det på det reelle spillebræt.



Figur 4: Viser sammenhængen mellem controlleren og den kunstige intelligens.

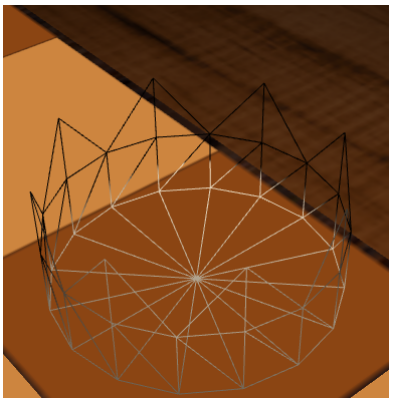
4.2.4 3D

(Skrevet af August)

Vi har valgt at vores spil skulle indeholde både en 2D og 3D version, da vi først havde lavet 2d version, skulle vi finde en måde at overføre, 2D til 3D. I Javafx virker 2d objekter også i 3d miljøet, så vi kunne bruge den samme GridPane, som vi havde lavet i forvejen, og bruge samme kontroller.

For at kunne få et 3D rum, skulle vi finde en måde at bevæge kammeraet, rundt i et 3D rum. Dette gjorde vi ved at lave, en EventHandler for tastatur input, der indeholdt, en switch-sætning, der kører funktioner alt efter hvilken knap der bliver klikket på.
Først prøvede først at man kun styrede kameraet, ved at prøve at få det til at rotere rundt om brættet, men det gav nogen unaturlige synspunkter. Så vi fandet ud af at det var meget bedere at styre kameraet i x,y,z, planen men altid have det peget nedad af z-aksen, og så rotere brættet rundt dens center punkt. På denne måde virker det også som at zoome ved at rykke kameraet, op og ned ad z-aksen. Det brugte vi til at binde til en EventHandler som tjekker for scroll på musen, så man kan zoome ved at scrolle.
Vi valgte at man styre kameret rundt med w, a, s og d, og roterede rundt om dam spillet ned i, j, l , k, og også zoome med, z og x.

Men vi kunne ikke bare have en et fladt bræt at spille på, vi ville implementere nogen rigtige 3d elementer. Første var at få brikkerne til at være 3d. Så vi lavede en ny klasse som forlængede javafx klassen Cyllinder3D, med en fast størrelse, og en konstruktør, der tager en boolean, for om den skal farves sort eller hvid.
For konge brikken ville vi gerne have en 3D krone model. Vi undersøgte om man kunne indlæse en 3D, model, men så skulle man bruge et tredje program til at lave den til et mesh som javafx kan se. Men da en krone kan lave forholdvis simpel, lavede vi vores eget TriangularMesh, som gøres ved først at lave alle punkterne, og derefter sætte flader i form af trekanter, mellem punkterne. Vi lavede bunden som en lige sidet 16 kantet polygon, punkternes kordinater kunne findes ved cos, og sin af $\pi/8,\ \pi/4,\ 3 * \pi/8$. Derefter byggede vi bunden op som en prisme, og til sidst satte punkter, som spiderne på toppen.
Her kan kronen ses som når den er tegnet med streger.



Figur 5: Mesh visning af krone brikken

Med en statisk boolean variabel (`XD`), tjekker vi om der er 2d, eller 3d, på og fylder brættet med 2d eller 3d brikker alt efter værdien. Variablens værdi bliver sat af en checkboks, i indstillingsmenuen.

Vi fik et problem med javafx, og den måde vi hånterede museklik, da vi fik elementer der gik længere op af z-aksesn, ville det løfte de fleste af 2d elementerne det var kun muligt at klikke på det højeste element. Vi startede med at fikse det ved at give alle felter en usyndlig boks, så alle altid var højere end kongen, men det gav problemer for så skulle man klikke højere, end felt for at ramme. Derfor ændrede vi EventHandleren, hvor i vores originale simple damspil, havde givet alle felterne en håndtereing for muse klik, til at hele brættet kun havde en, og derefter chekkede hvilken node man klikkede på.

Vi ville gerne udnytte vores 3d mere interaktiv, som med scoren og tiden. Vi lavede først et stort bord, som vi kunne tilføje ekstra elementer ovenpå. Det er lavet af 5 bokse, 4 til ben og en til borpladen, vi gav også brættet en boks under for at give det fylde. Og så lavede vi en tekstur der lignende træ, med et billede.

Vi valgte at alle størrelse er skaleret efter størrelsen på vinduet, så man kunne frit ændre strørrelsen på vinduet, og alle elementer ville skalerer sig selv.

Vi har valgt at bruge StackPanes, til at holde hvert object som er bygget op af flere noder. Fordi man let kan placere noderne hvor man vil da de alle starter i samme punkt.

Vi lave et papir af et 2d rektangel, med en tekstur af et billede af et papir, og puttede Labels, på som dem der er i over og under spillet. Og givet dem en font der ligner håndtegning. Vi lavede en funktion som puttede alle noderne i en StackPane.

For at lave urene lave vi igen et trekant mesh, til at skabe en prisme med en retvinklet trekant som grundflade. Og lavede Labels, som vi havde gjort med tidtagning i 2d, og vi gav dem en glow effekt så det ligner et rigtigt digital ur.

For at man ikke selv manuelt skulle sætte kamererat, ville vi have nogen faste, kameravinkler. Vi har lavet 3 vinkler, fra siden, fra toppen, og lige på, hvor man drejer rundt efter hver tur. Vi lavede animationer af knappen og kameraet, ved brug af javafx klassen Timeline, som flydende ændrer en nodes, værdi til en valgt værdi, over en tidsperiode. Vi valgte også at sætte en tekst, som bliver større når man klikker, så man ved hvilken indstilling man er på.
Vi lavedet en ny Enum, så vi kunne lave en switch sætning, da vi havde en knap, til tre værdier.

Vi ville gerne implementere et rum rundt om, så man ikke spillede i et hvidt tomt sted. Først prøvede vi at lave rummet, som store rektangler, rundt om bordet, men det bliv for tungt for computeren. Så vi lavede vores version af det man kalder en Skybox. Ved brug af SubScene, kunne vi skabe to 3d scener, som bliv vist oven på hindanen. Den bagerste scene lavede vi til vores skybox, ved at lave seks rektangler, på samme størrelse som brættet, til et rum. Og sætte et kamera i midten så det så større ud. Vi gav rummet en rotation om midten. og rotationen med brættets rotation, så de altid havde samme rotering.

4.2.5 Timer

(Skrevet af Victor)

Vi har valgt at sætte en timer ind i spillet mage til den man har i skak. Hver spiller har sin egen timer som tæller ned så længe det er deres tur. Når spilleren har udført et træk, begynder modstanderens timer at tælle ned indtil han laver sit træk.

For at implementere timeren i spillet bruger vi Java klasserne `Timer`, `TimerTask` og `Platform`. Vi har lavet en metode i `Controller` som hedder `doTime()`. Hver gang metoden bliver kaldt starter en ny timer med de værdier spilleren har valgt. Metoden bliver kaldt efter man trykker på "Go"eller på "Rematch"knappen.

Så snart metoden bliver kaldt, danner den en ny timer og giver den en `TimerTask`, som skal køre med `Platform.runLater()`. Dvs. at det er en task som skal opdatere spillets UI løbende. Den tjekker så hvis tur det er og formindsker et heltal `seconds1` hvis `turn == true`, og heltallet `seconds2` hvis `turn == false`, og opdaterer timerens `Label` i View hver sekund. Så snart en af timerne rammer 0, taber spilleren der løb tør for tid og timeren bliver aflyst.

Metoden tjekker også om timeren skal blive aflyst selvom der ikke er en spiller der har vundet. Det sker når spilleren forlader spillet uden at have en vinder ved at trykke på "Main Menu"knappen. Vi tager hensyn til det ved at have en `boolean "cancel"`, som holder styr på det og så snart `cancel == true` bliver timeren aflyst.

4.2.6 Lyd

Til implementering af lyd i spillet, har vi valgt at spilleren har mulighed for at vælge om lydeffekter, fx. når en brik bliver rykket, skal være slået til. Dette gøres ved brug af en boolean, som kan blive slået til eller fra, under indstillinger. For at implementere lydeffekter hentes lydfilerne som `Media` og der bruges `MediaPlayer`, med metoden `.play()` til at afspille lyden.

4.3 animation

Vi valgte at give brikkerne en animation, vi brugte en metode, som efter brikken var rykket til det nye felt, blev den på samme tid translateret til den gamle plads, og med en TimeLine, lavede vi en animation til den er på plads igen.

4.4 Overvejelser

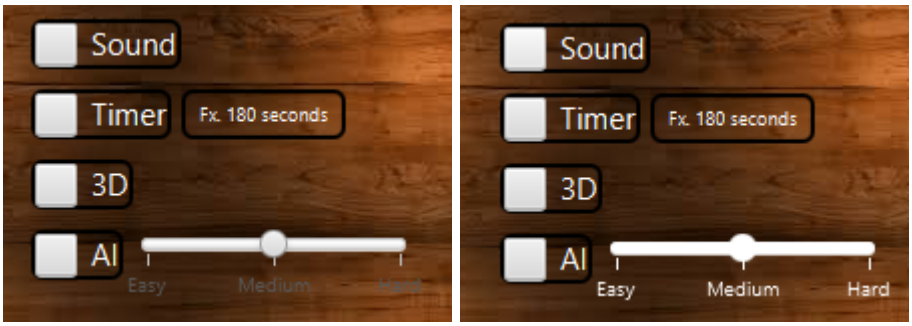
(Skrevet af Carl)

Ideén fra start var at lave selve spillet vha. JavaFX, da vores GridPane ville være en nem og effektiv måde at danne et spilbræt på. Resten af spillets scener (alle menuerne), ville vi lave i FXML vha. SceneBuilder. Da vi allerede var begyndt på at designe de forskellige menuer i JavaFX, og mange af vores `Button handlers` allerede var på plads, besluttede vi os for ikke at skifte til `FXML`, da vi derfor skulle lave det forfra.
Vi har dog stadig brugt `SceneBuilder` i sammenspil med `FXML`-dokumenter. Vi har brugt det i den forstand, at mange af vores scener er blevet "designet"i `SceneBuilder`, der fungerer som et godt

skabelon-værktøj. Mange af de funktionaliteter vi har tilføjet til knapper, labels og textfelter, har vi opdaget vha. `SceneBuilder`.

Et godt eksempel er `slider`-indstillingen, der hører til `cb4`. Vi havde mange problemer med at få farven på vores `slider-labels` til at blive hvid, af en eller anden årsag forblev den grå, selvom vi prøvede at ændre stil vha. `css-styling`. Efter at have rodet rundt i `SceneBuilder`, opdagede vi at der var mulighed for at tilføje en effekt, kaldet `ColorAdjust`. Dette inkorporerede vi i koden, og derefter var "slideren"så tydelig som vi ville have den.

Eksempel `Før- og efter-ColorAdjust`.



5 Evaluering

(Skrevet af Joel)

I dette afsnit vil vi beskrive hvilke tests vi har udført på programmet, samt hvilke overvejelser til forbedringer vi har gjort os efter at have lavet programmerne.

5.1 Tests

Til at teste vores programmer har vi udført en række forskellige tests. Undervejs i skrive processen af programmerne, har vi testet de funktioner som blev tilføjet, dette blev gjort for at sikre at funktionerne virkede som forventet. Da programmerne var færdige testede vi dem grundigt, for at sikre os at der ikke var nogle uventede fejl, og at alt fungerede optimalt. Ydermere benyttede vi test personer, som ikke havde nogen forrig relation til vores program, til at afprøve programmet imens vi observerede funktionaliteten. Dette blev tildeles også gjort for at se om brugerne følte spillet var intuitivt, og nemt at forstå, selvom en bruger aldrig havde spillet dam før.

5.2 Forbedringer

Strukturering
Hvis vi skulle lave programmet forfra, står det helt klart at der skulle have været en bedre kode struktur, og en mere sammenfattet forståelse for hvordan koden skulle skrives, variabler navngives, og hvilken mængde af kommentarer der blev forventet. En mangel på dette har senere i processen gjort små opgaver større, idet man nemt kunne fare vild i koden, hvis man arbejdede i et afsnit man ikke selv havde skrevet.

Grundlaget til Dam
I det tidlige stadie af SimpDam stod vi med to ideer til implementationen af spilbrættet, den ene ide stod på at lave et to dimensionelt helttals array, indeholdene alle felter og brikker, det visuelle spilbræt ville så opdateres ud fra dette array, den anden ide stod på at direkte tilgå vores GridPane, og igennem det have adgang til felter og brikker. Vi valgte at gå med den direkte adgang til GridPane, da det virkede som den nemmeste og mest effektive implementation. Vi genovervejer nu om dette valg var det bedste, der kan nemlig findes fordele og ulemper ved begge metoder, men den største fordel ved array metoden opstod da vi skulle implementere den kunstige intelligens. Den kunstige intelligens forudsiger træk baseret på et to dimensionelt heltals array, og ved array metoden ville den kunstige intelligens blot kunne returnere et spilbræt hvor trækkende var foretaget, hvorefter det visuelle spilbræt direkte kunne opdateres. Vores nuværende metode returnere postionen af brikken og hvor den skal rykkes til.

6 Konklusion

(Skrevet af alle)

Vi har vha. JavaFX været i stand til at opfylde opgavekravene der blev stillet. Udover den færdige version af Simp-Dam, gik vi i gang med en udvidet version, Avanceret Dam. Vi har vha. CSS-styling, udviklet et program med en intuitiv brugergrænseflade. Derudover har vi implementeret 3d-perspektiv og en modstander, styret af en kunstig intelligens. Det var dog en lidt tung opgave at implementere både 3d og kunstig intelligens, derfor nåede vi ikke at implementere "combo-kill"i vores kunstige intelligens. Vi har testet programmet for fejl og mangler, både ved selv at køre det utallige gange, men også ved at lade diverse test personer prøve det. Fejlene der er opstået undervejs har vi hurtigt fikset, og vi står tilbage med et tilsyneladende velfungerende og intuitivt program.

7 Litteraturreferencer

[1] Oracle. *Java og JavaFX Dokumentation* URL:`https://docs.oracle.com/`

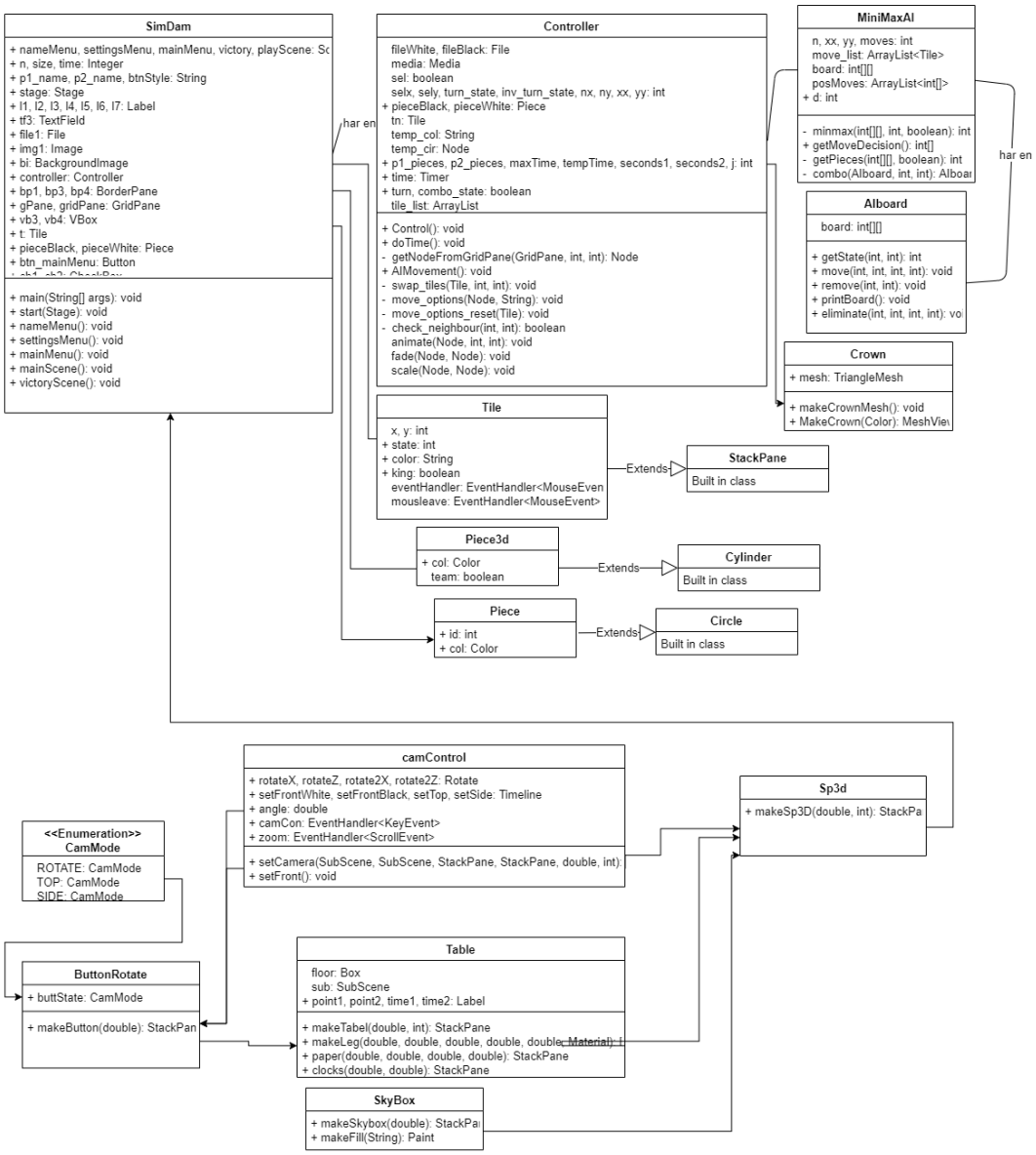
[2] Stack Overflow URL:`https://stackoverflow.com/`

[3] Eksempelrapport - Steffen Holm Cordes og Benjamin Bogø, *"02121 - Introduktion til Softwareteknologi Snake"* (2019)

[4] Wikipedia Draughts URL:`https://en.wikipedia.org/wiki/Draughts`

[5] Sebastian Lague: *Algorithms Explained - Minimax and alpha-beta pruning* URL:`https://www.youtube.com/watch?v=l-hh51ncgDI&t=296s`

8 Bilag



Figur 6: Klassediagram