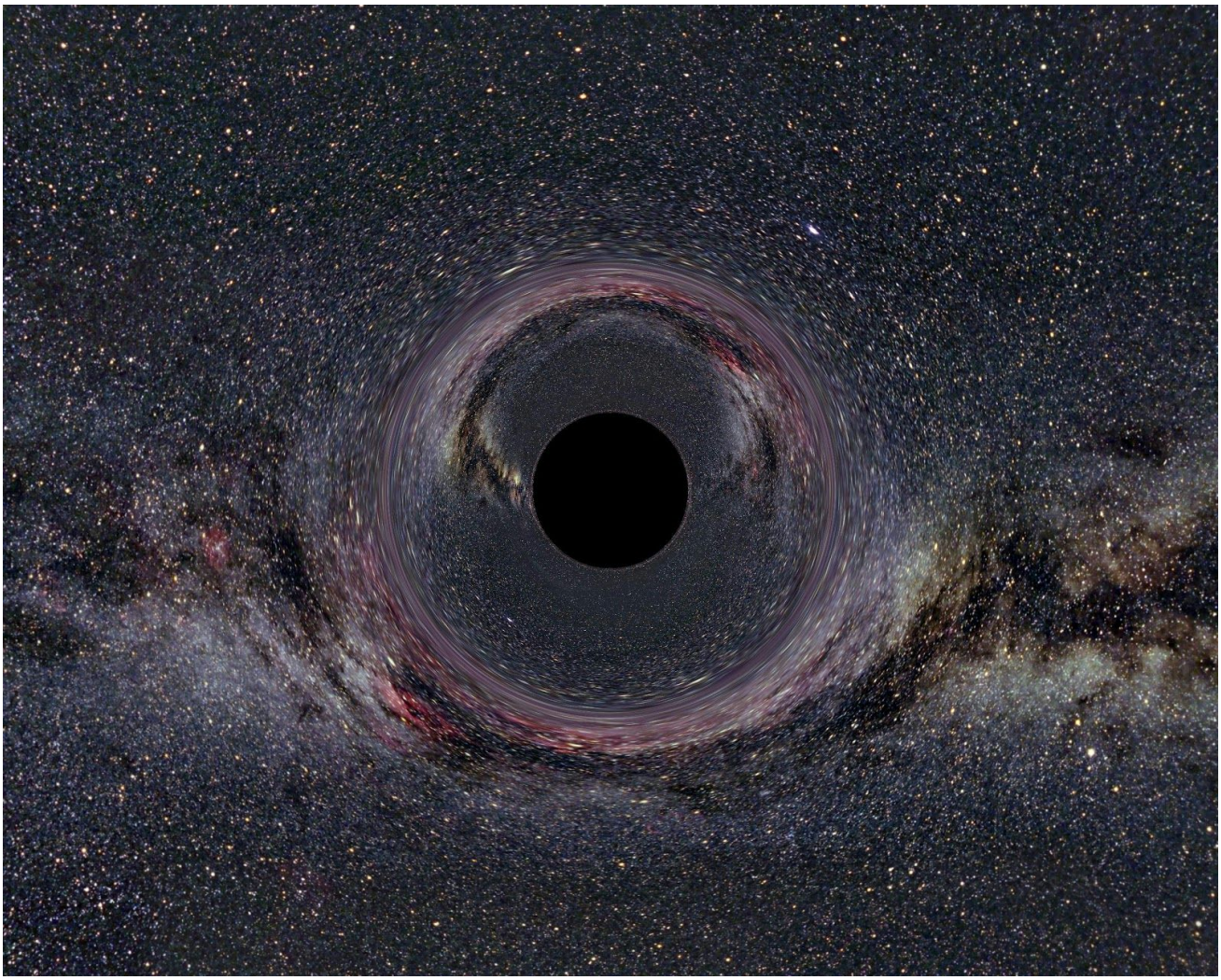
Singularity Chess: waar het allemaal om draait

Mèh me läppke - 26 juni 2015



Koen Timmermans (s4461762)
Marnix Suilen (s4443772)
Coen Borghans (s4473116)
Thijs Heijligenberg (s4451414)

Voorwoord

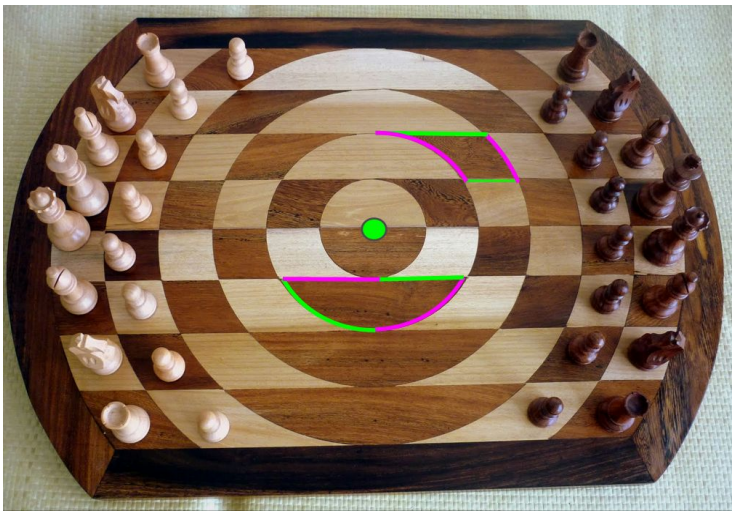
In het vierde kwartaal van het collegejaar 2014-2015 hebben wij het vak Research & Development Project gevolgd. Het doel van deze cursus was voornamelijk om uiteindelijk een werkende android-applicatie te ontwerpen en te maken. Wij hebben gekozen om *Singularity Chess*, een variant van het schaakspel, als android-app te implementeren. U leest nu het eindverslag van dit project.

Dit document is conform de opdracht ingedeeld in drie hoofdmoten: de beschrijving, het ontwerp, en de reflectie. De beschrijving bevat details en uitleg over wat het idee achter onze app is, een productverantwoording en een aantal specificaties waaraan de applicatie voldoet. Omdat we een lange, saaie uitleg van de precieze regels van Singularity Chess in de beschrijving willen voorkomen, gaan we er in dit verslag van uit dat de lezer enige kennis heeft van het normale schaakspel. In het deel *ontwerp* staat uitgelegd hoe alles werkt en waarom voor sommige opties gekozen is. De reflectie bevat zoals de naam doet vermoeden een terugblik op het project, met al zijn positieve en negatieve kanten.

Inhoudsopgave

Beschrijving

Met onze app, *Singularity Chess,* kan je exact doen wat de naam doet vermoeden: je kan er singularity chess mee spelen. Dit is een vorm van schaken waarbij het bord er anders uitziet dan de meeste mensen gewend zijn (er is namelijk een soort ‘singulariteit’) en waarbij de stukken zich ook anders lijken te bewegen. De pionnen zetten gewoon recht vooruit, maar de andere stukken hebben op het eerste gezicht een compleet ander gedrag.
Dat dit slechts schijn is, komt geheel door de vorm van het bord. De stukken houden zich aan de regels van normaal schaken, dus lopers bewegen schuin, torens recht, dames kunnen alle richtingen op, en zo verder en zo voorts. De velden van het bord zitten echter niet aan elkaar als bij gewoon schaken.
Door deze aparte samenstelling is het nodig om zetten algemener te definiëren. Om deze definities goed te kunnen formuleren, moeten we eerst kijken naar een aantal eigenschappen van de velden. Waar we vooral op moeten letten, zijn de hoekpunten en de zijdes van de velden. Voor de meeste, vierkante velden is het duidelijk wat de hoekpunten en de zijdes zijn en welke hoeken en zijdes tegenover elkaar staan. Voor een aantal velden kan het echter wat lastiger zijn om dit meteen in te zien. Afbeelding 1 toont een bord van singularity chess, waarop van een aantal velden is aangegeven wat de vier zijdes hiervan zijn. De grote stip in het midden is de singulariteit, het punt waaraan het spel zijn naam heeft te danken. Op de eigenschappen van deze singulariteit komen we later terug.

 [1]

Afbeelding 1

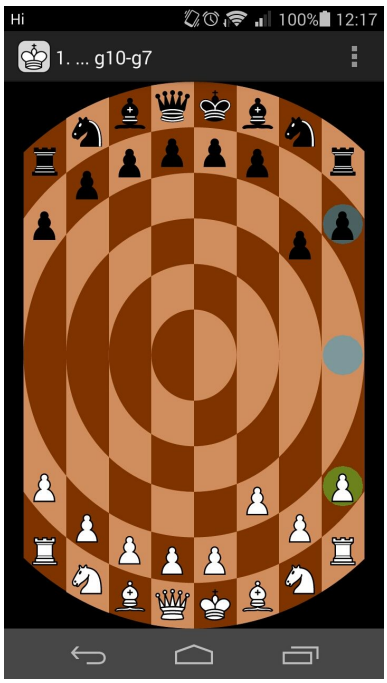
Nu definiëren we een rechte zet als het verplaatsen van een stuk naar een veld dat twee hoekpunten gemeen heeft met het oorspronkelijke veld. Op dezelfde manier definiëren we een schuine zet als het verplaatsten van een stuk van een veld naar een ander veld dat aan het eerste veld grenst met één hoekpunt.
Een lange rechte zet is vervolgens een aantal rechte zetten achter elkaar, waarbij natuurlijk het volgende veld aan de overstaande zijde moet grenzen als waaraan het vorige veld grenst. Een lange schuine zet is analoog gedefinieerd.

[1] Khoa Pham - http://abstractstrategygames.blogspot.nl/2010/10/singularity-chess.html

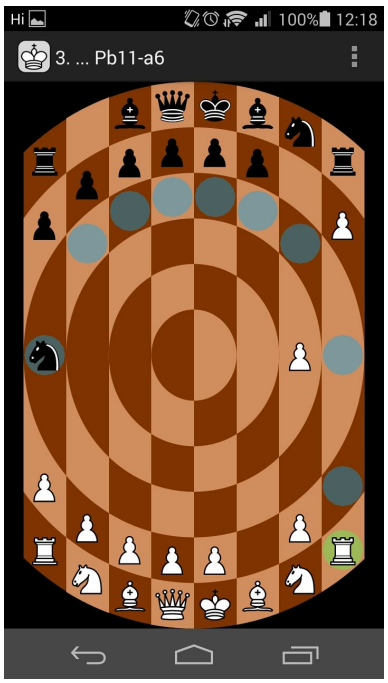
Merk op dat deze definitie bij het normale schaakspel equivalent is aan de intuïtieve definitie voor een rechte en een schuine zet.
Alle stukken bewegen volgens de normale regels van schaken conform deze definities, behalve de pionnen. Deze zullen ‘gewoon’ recht vooruit bewegen.

Ter verduidelijking volgt een aantal voorbeelden van mogelijke zetten in het spel *Singularity Chess* aan de hand van screenshots. Een groende cirkel om een stuk geeft aan dat dit stuk geselecteerd is. De blauwe cirkels geven alle mogelijke zetten voor dit stuk aan.

Afbeelding 2 toont een beginzet met een pion. De mogelijkheid om bij de eerste zet twee stappen te doen zou de pionnen te veel macht geven, dus zo’n dubbele zet is bij Singularity Chess niet toegestaan. Toch is het hier op het oog mogelijk om twee stappen vooruit te doen. Het veld waar de witte pion staat en het veld met de blauwe cirkel waar de zwarte pion in staat delen een hoekpunt, waardoor deze velden in feite schuin naast elkaar liggen. De zet ‘twee plaatsen naar voren’ is eigenlijk schuin rechts vooruit. Pionnen op een normaal bord kunnen schuin slaan en dat kunnen ze hier ook.
Afbeelding 3 maakt duidelijk hoezeer het bord een singulariteit heeft waar de stukken omheen draaien. De toren rechtsonder kan in één ‘rechte’ lijn het paard links slaan.
In afbeelding 4 zijn alle zetten van de zwarte koningin getoond, met voor de duidelijkheid de vier verschillende richtingen in vier verschillende kleuren. Zwart en rood zijn de ‘rechte’ zetten, de opeenvolgende velden hebben telkens twee hoekpunten gemeen, en blauw en groen geven de ‘schuine’ zetten aan (velden hebben één hoekpunt gemeen). Merk op dat de middelste twee velden met drie hoekpunten aan elkaar zitten, ze zitten dus zowel recht als schuin naast elkaar. Dit is de enige mogelijkheid op het bord voor lopers om van kleur te wisselen. Bij rechte schaakborden is dat niet mogelijk.



Afbeelding 2



Afbeelding 3

Afbeelding 4

Productverantwoording

In de Google's *Play Store* is geen app te vinden die zich volledig richt op singularity chess, terwijl deze juist uitpuilt van de schaakapps voor normale borden. Ook een zoekactie op het internet leverde geen resultaten over een applicatie. Wel is er in de app Happening[2] een mogelijkheid om singularity chess te spelen. Dit is echter niet het doel van die app en de implementatie laat op een aantal fronten nog wat te wensen over, dus in dat opzicht is onze app is een unicum. Qua vergelijkbare apps is het natuurlijk duidelijk dat hier alle schaakapps onder vallen, maar aangezien singularity chess een geheel andere speelervaring biedt, blijft onze app voorlopig marktleider in deze niche van bordspelapps.

Specificaties

De belangrijkste eigenschap van de app is uiteraard dat er singularity chess mee gespeeld kan worden. De gebruikers van deze app zullen vooral als doel hebben een potje te kunnen spelen en zullen verder niet veel functionaliteit verwachten. Om te garanderen dat dit werkt en goed aanvoelt, moet het model goed in elkaar zitten en moeten het model en de user-interface goed met elkaar kunnen communiceren. Ook is het van belang dat de stukken en het bord correct op het scherm worden weergegeven aan de hand van de gegevens in het model. Verder moet steeds maar één speler per keer kunnen zetten. Er zijn ook een aantal andere zaken die hier niet direct mee te maken hebben, maar de app wel veel handelbaarder en speelbaarder maken, zoals een functie om een spel op te slaan en te laden en het design in het algemeen. Op deze manier is alles op te delen in specificaties en onderspecificaties. De specificaties die **dikgedrukt** zijn, zijn de functionele en/of minimaal vereiste specificaties

- ontwerp
    - beginscherm
    - **speelscherm** (*GameActivity*)
        - **bord tekenen** (*ChessView*)
        - **stukken op juiste plek tekenen**
        - mogelijke zetten kunnen tekenen
        - tonen wie aan de beurt is
    - instellingenscherm
- spel
    - **werkend model**
        - **correcte zetten kunnen doen**
        - **bijhouden wie aan de beurt is**
        - (schaak)mat bijhouden
        - pionnen promoveren
    - **met *GameActivity* communiceren over gedane zetten**
    - **tekenen in *ChessView* van mogelijke zetten**
- ondersteunende functies
    - opslaan en laden
    - opties bijhouden
    - spelen via bluetooth[3]

[2] https://play.google.com/store/apps/details?id=im.happening.android

[3] Dit onderdeel is niet geïmplementeerd in de uiteindelijk ingeleverde versie. Het is ook niet volledig werkend gekregen en had dus ook niet veel toegevoegd. Omdat Thijs hier wel vrij veel tijd in heeft

Ontwerp

Globaal ontwerp

De app is in verschillende delen opgesplitst. We hebben hiervoor geprobeerd om het MVC-model zo goed mogelijk aan te houden. Het model bestaat uit verschillende klassen, die samen een potje singularity chess kunnen simuleren. Alle schaakstukken hebben een eigen klasse, die de abstracte klasse Piece uitbreiden. Elk schaakstuk heeft eigen regels voor het doen van een volgende zet, dus de methode die een lijst met volgende posities teruggeeft is bij ieder schaakstuk anders. Ook is er een klasse Position, die een positie op het bord representeert. De twee spelers worden gerepresenteerd door twee objecten van de klasse Player.

De klasse Game houdt het hele model bij elkaar en zorgt dat het speelbaar is. Bij het creëren van een object uit de Game-klasse wordt een array aangemaakt die aangeeft welke velden er (schuin) naast elkaar liggen en worden de schaakstukken in beginpositie op het bord gezet. De klasse Game kan ook zetten op het bord uitvoeren. Game breidt de klasse Observable uit, zodat hij de andere klassen van de app kan laten weten als er iets verandert. De gebruiker moet natuurlijk ook het spel kunnen zien, hier zorgt de ChessView voor. De ChessView implementeert de klasse Observer en kan daardoor het scherm verversen als er iets in het model verandert.
De klasse Controller registreert alle aanrakingen van de gebruiker. Als er een schaakstuk wordt ge(de)selecteerd, dan wordt dat aan de ChessView doorgegeven, die dat weer duidelijk maakt aan de gebruiker. Als er een stuk verplaatst wordt, past de Controller het model aan.

Naast de klassen die deel uitmaken van het MVC-model voor het grootste deel van de app, zijn er een aantal klassen die zorgen dat alles er uit ziet zoals andere apps voor android. Het startscherm dat de gebruiker als eerste ziet als hij de app opent, wordt gerealiseerd door de klasse MainActivity. Deze klasse breidt Activity van android uit en beschrijft wat de knoppen op het startscherm doen.
De klasse GameActivity breidt ook de klasse Activity uit, en is het scherm dat de ChessView bevat. Deze klasse bevat ook de Game en de Controller. Ook kan deze klasse het spel opslaan en laden.
Het derde scherm dat de app rijk is, is het instellingenscherm, dat wordt gerepresenteerd door PreferencesActivity, die de klasse PreferenceActivity van android uitbreidt. Omdat de lay-out van dit scherm wordt vastgelegd in een apart XML-bestand en android er zelf voor zorgt dat de instellingen worden opgeslagen, bevat deze klasse weinig code.

Dit zijn alle klassen die in de ingeleverde app aanwezig zijn. Voor de bluetoothimplementatie zijn er nog een aantal extra klassen, deze zijn beschreven in de volgende sectie, onder *Bluetooth*.

---

gestoken en omdat het correct implementeren van bluetooth in onze ogen erg interessant is, is er toch een stukje over te vinden in de ontwerp-sectie van dit verslag.

Detailontwerp

Model

We hebben het model ontworpen als een java-programma dat onafhankelijk van de app en van android kan functioneren. Op deze manier kan de code nog nuttig zijn voor een pc- of iOS-implementatie. Hiernaast was het ook makkelijker om de het model te ontwikkelen en te testen voor de rest van de app af was.
Het model is onderverdeeld in een aantal klassen. Zo is er een klasse om stukken te representeren, een klasse om gegevens van spelers bij te houden en een klasse om het spel goed te laten verlopen en het bord bij te werken. Hieronder staat voor de belangrijkste klassen van ons model beschreven wat de basisfunctionaliteit is en hoe deze met andere klassen van het model in verband staat.

Directions

Aan de basis van het model staat de driedimensionale array *directions,* een attribuut van de klasse *Game*, die later beschreven staat. Deze array geeft op een intuïtieve en efficiënte manier aan welke velden waar naast welke andere velden liggen. Zo staat er in *directions*[3][5] wat de acht buren zijn van het veld C5 en in *directions*[1][1] wat de drie buren zijn van het veld A1. De manier waarop deze buren staan opgeslagen is zo gekozen dat twee velden die (schuin)tegenover elkaar liggen, op een afstand van 4 (modulo 8) van elkaar staan in hun array. De volgorde van *directions* wordt voornamelijk gebruikt om zetten goed uit te kunnen rekenen.

Piece

De schaakstukken worden gerepresenteerd door de abstracte klasse *Piece*. Deze bevat attributen die bijhouden of het stuk van zwart of van wit is, op welke plek het staat en wat voor soort stuk het is (dame, pion, loper, etc.). Uitbreidingen van de klasse *Piece* zijn alle typen schaakstukken (de klasse *Queen*, *Rook*, *King*, etc.) De belangrijkste methode van *Piece* is de abstracte methode *moveOptions*. De implementatie hiervan in de uitbreidingen van *Piece* levert een lijst met alle velden waar het stuk deze beurt heen zou kunnen bewegen volgens de regels van *Singularity Chess*. Omdat deze regels per type stuk verschillen, is de implementatie van *moveOptions* in al deze klassen net wat anders. Het basale idee achter de methode blijft echter wel behouden, dus een uitleg van de werking voor de klasse *Rook* zal het functioneren van *moveOptions* ook duidelijk maken voor de andere stukken.
Voor elke richting die de toren op kan (rechts, voor, links, achter) wordt aan de hand van gegevens uit *directions* steeds het volgende vak in die richting toegevoegd aan de lijst, totdat de toren niet naar deze positie kan bewegen, omdat er bijvoorbeeld een ander stuk in de weg staat. De *Position*s van deze vakken worden in een *ArrayList* gestopt. Vervolgens wordt de hele *ArrayList* afgegaan om vakken te zoeken waar de speler schaak zou staan, als hij hier heen zou gaan. Dit wordt als volgt gedaan: voor elke *Position* in de *ArrayList* wordt er een kopie van de lijst met de 32 stukken en hun posities op het bord gemaakt, en op die kopie wordt een zet gedaan. Vervolgens wordt gekeken of de speler in die kopie schaak zou staan. Als dat zo is, dan was de zet dus ongeldig (het is bij schaken verboden om jezelf schaak te zetten, dus bij singularity chess ook) en wordt hij uit de *ArrayList*

gehaald. Als alle mogelijke zetten zijn gecontroleerd, dan zal *moveOptions* de *ArrayList* teruggeven.

Player

De klasse *Player* dient om gegevens van de spelers op te slaan en te verwerken. Een attribuut van *Player* is dan ook de kleur (zwart/wit) van de speler. De belangrijkste methode van *Player* is *isChecked*. Deze controleert of de speler op dat moment schaak staat volgens de regels van *Singularity Chess*. Dit wordt gedaan door voor alle stukken van de tegenstander na te gaan of deze de koning van de speler zou kunnen slaan.

Game

Centraal in het model staat de klasse *Game*. Deze representeert het spel en zorgt ervoor dat de stukken gezet kunnen worden en dat het bord wordt bijgehouden in een array. Ook staat de structuur van het bord in *Game* opgeslagen in een array genaamd *directions*, die hierboven beschreven is. De methode *fillBoard* vult een andere array, waarin het bord staat opgeslagen, met 32 uitbreidingen van *Piece* (één *King*, één *Queen*, twee *Bishop*s, *Knights* en *Rooks* en acht *Pawns* voor zowel wit als zwart) en geeft deze allemaal de juiste startpositie volgens de regels van *Singularity Chess*, door aan de constructor een goede variant van de klasse *Position* mee te geven.
Verder bevat *Game* de methode *move* om stukken op het bord te verplaatsen. Hierin wordt aan de had van gegevens van het bord gekeken of de gewenste positie wel in de moveOptions van het stuk staat en als dat het geval is, wordt de positie van het stuk in kwestie aangepast. In het geval dat er een ander stuk wordt geslagen bij zo'n zet, worden de gegevens van dat stuk ook veranderd.
Hiernaast kan in *Game* de staat van het spel worden opgevraagd met *checkMate*. Deze methode levert aan de hand van gegevens van de twee *Players* op of er een speler schaakmat of pat staat. Op deze manier gecontroleerd worden of het spel beëindigd moet worden.

Schermen

Speelscherm

Het speelscherm wordt gerepresenteerd door *GameActivity*, een klasse die *Activity* van android uitbreidt. Er bestaat maar één object van *Game*, één van *ChessView* en één van *Controller* tegelijk, en die bestaan alleen als het speelscherm te zien is, dus het is vrij logisch om deze in *GameActivity* als attributen te hebben. Deze klasse regelt ook het opslaan en laden van een spel. Opslaan gaat in z'n werk door het *Game*-object toe te voegen aan een *ObjectOutputStream*, en die aan een *FileOutputStream*, die het object met al z'n attributen opslaat in een bestand op het bestandssysteem. Via een *ObjectInputStream* en een *FileInputStream* kan een bestand, mits het bestaat, weer omgezet worden in een *Game*-object, wat weer nodig is bij het laden van een spel. Telkens als het speelscherm wordt afgesloten (als de gebruiker naar een ander scherm gaat of een andere app naar de voorgrond brengt) wordt het spel opgeslagen, en telkens als het speelscherm weer geopend wordt, wordt het spel weer geladen. Op deze manier hoeft de gebruiker er niet zelf aan te denken om het spel op te slaan. Daarnaast is er natuurlijk een methode om een nieuw spel te starten, mocht er iets mis gaan met het laden.

In het speelscherm groot het bord met de stukken te zien, en daarboven een balk die android de *ActionBar* noemt. In de *ActionBar* zijn een aantal dingen te zien. Links is een plaatje van de koning, in de kleur van de speler die aan de beurt is met zetten. Daarnaast staat als het spel net begonnen is de tekst *Wit begint*, en anders staat er de vorige zet in lange schaaknotatie[4], met de coördinaten die ook in het model gebruikt zijn. Helemaal rechts is een menuutje te zien, met daarin opties om een nieuw spel te starten, terug te gaan naar het startscherm, naar het instellingenscherm te gaan en om remise aan te vragen. Deze laatste optie laat een pop-up zien met de vraag of de gebruiker zeker is of hij een remise aan wil vragen.
Ten slotte heeft de klasse *GameActivity* nog een methode die, als een speler schaakmat staat, middels een *AlertDialog* laat zien wie er gewonnen heeft.

Startscherm

Het startscherm wordt in de app gerepresenteerd door de klasse *MainActivity*, die een uitbreiding is van de klasse *Activity* van android. De lay-out van dit scherm wordt in een apart XML-bestand vastgelegd.
Het scherm bevat vijf knoppen, met de opschriften *Nieuw spel*, *Bluetooth*, *Spel laden*, *Informatie* en *Instellingen*[5].

- De knop *Instellingen* opent het instellingenscherm.
- De knop *Informatie* laat een pop-up zien waarin in een paar zinnen staat uitgelegd wat het verschil tussen singularity chess en gewoon schaken.
- De knop *Bluetooth* laat in de ingeleverde versie het bericht ‘Nog niet geïmplementeerd’ zien, in de versie met bluetooth opent hij naar het bluetoothscherm.
- De knop *Spel laden* start een *GameActivity*, en zoals in het stuk hierboven aangegeven laadt deze altijd een opgeslagen spel als hij geopend wordt.
- De knop *Nieuw spel* verwijdert het bestand waarin een eerder spel is opgeslagen en start dan een nieuwe *GameActivity*. De *GameActivity* ziet dan dat er geen opgeslagen spel is en start een nieuwe.

Instellingenscherm

De klasse *PreferencesActivity* is een uitbreiding van de klasse *PreferenceActivity* van android, en stelt het instellingenscherm voor. De lay-out van dit scherm is in een apart XML-bestand opgeslagen en android zorgt er zelf voor dat het instellingenscherm er uit ziet en werkt zoals elk ander instellingenscherm in android, en slaat zelfs de instellingen automatisch op. Daardoor is het enige dat in *PreferencesActivity* beschreven is de methode die er voor zorgt dat het pijltje linksboven hetzelfde doet als de terugknop (namelijk naar het scherm gaan vanuit waar het instellingenscherm is geopend).
De instellingen zelf zijn *Bevestiging voor zetten* en *Scherm draaien*. *Bevestiging voor zetten* zorgt ervoor dat de gebruiker een pop-up te zien krijgt als hij een zet doet. Hiermee kan de gebruiker nog annuleren als hij zich bedenkt. Deze optie is vooral handig voor beginnende gebruikers.
Met de optie *Scherm draaien* kan de gebruiker instellen wat er moet gebeuren als de zwarte speler aan de beurt is, hij kan hier uit drie dingen kiezen:

---

[4] https://nl.wikipedia.org/wiki/Schaaknotatie#De_lange_notatie
[5] Dit zijn de teksten als de taal van het apparaat ingesteld is op Nederlands, in de andere gevallen zijn de opschriften in het Engels.

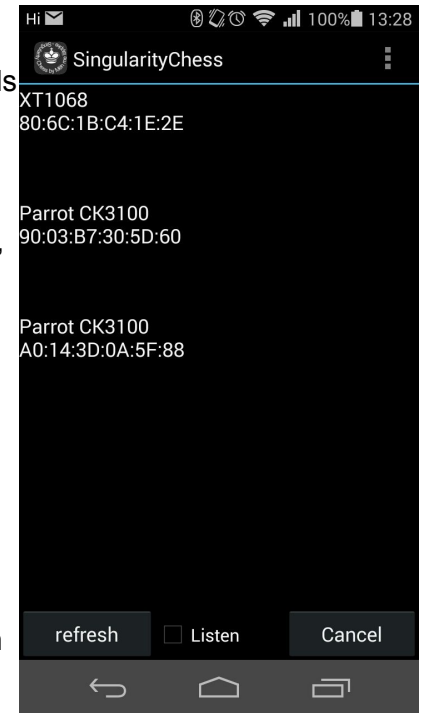
- *Draai bord* draait het bord om (en zet de stukken weer recht) zodat de zwarte stukken onder zijn als de zwarte speler aan de beurt is. Dit is vooral handig als de twee spelers die tegen elkaar spelen naast elkaar zien, en het apparaat vanaf dezelfde kant bekijken.
- *Draai stukken* zet alle de stukken ondersteboven als de zwarte speler aan de beurt is, maar laat ze wel op dezelfde plek staan. Dit is handig als de twee speler tegenover elkaar zitten te spelen, met het apparaat er tussen.
- *Niet draaien* verandert niets als de zwarte speler aan de beurt is. Deze optie is handig als de gebruikers gedesoriënteerd raken van het draaien van het bord of de stukken.

Bluetooth

Voor de duidelijkheid: alle functies en klassen hieronder beschreven zijn niet aanwezig in de ingeleverde versie, maar aangezien het toch een hoop (interessante) code is, is het toch hier opgenomen. Voor eventuele volledige code kunt u contact opnemen met de auteurs.

In de versie van de applicatie met bluetoothfuncties zitten een aantal extra klassen. Kort beschreven doen deze dit:

- *BTActivity*: activity waarin je kunt kiezen om te luisteren naar inkomende connecties, of om een reeds gekoppeld apparaat te proberen te koppelen.
- *AcceptThread*: thread die luistert naar inkomende connecties en probeert verbinding te maken. Bij geslaagde poging maakt deze een *ConnectorThread*, en geeft deze aan de *BTActivity* door dat dit gebeurd is.
- *ConnectThread*: thread die een connectie probeert aan te gaan met een *BluetoothDevice*. Bij geslaagde poging maakt deze een *ConnectorThread*, en geeft deze aan de *BTActivity* door dat dit gebeurd is.
- *ConnectorThread*: thread die van een *Accept-* of *ConnectThread* een *BluetoothSocket* meekrijgt met daarop een connectie. Deze thread maakt een *FileStream* (in en out) aan op deze connectie, en een *Objectstream* (in en out) op de *FileStream*. Deze thread blijft draaien zolang deze in stand blijven.
- *MyProperties*: een klasse met enkel wat public static variabelen die door de app heen gebruikt worden. Uitleg volgt hieronder.
- *Move*: dit is een aanpassing aan het model die ervoor zorgt dat je een beweging niet uitvoert met twee posities, maar met één *Move*, zodat deze geserializeerd en verstuurd kan worden.



Afbeelding 5

Een connectie opzetten gaat als volgt te werk: men gaat naar het bluetoothscherm (afbeelding 5) en zet bluetooth aan. Dan kan gedaan worden door ofwel op een apparaat drukken en wachten op een antwoord, ofwel op *listen* drukken en op iemand wachten die

probeert te verbinden. Het is dus nodig om al met het andere apparaat verbonden te zijn via het bluetoothmenu van android.
De *ConnectThread* roept met de meegegeven device *device.createRfcommSocketTo ServiceRecord(UUID ...)* aan, die een BluetoothSocket returnt. De *UUID* is statisch en willekeurig gekozen voor deze app, aangezien de *UUID* van de device zelf geen resultaat gaf. Vervolgens roept hij *mmSocket.connect* aan, een methode die wacht tot er een connectie is. Deze methode kan bijzonder veel bijzonder onduidelijke exceptions geven die niet gedocumenteerd zijn, waaronder IOException: *connection might failed, read ret: -1.*
De *AcceptThread* roept *mBluetoothAdapter.listenUsing RfcommWithServiceRecord (name, UUID ...)* aan met dezelfde UUID als de *ConnectThread*. Deze returnt een BluetoothServerSocket, waarmee je naar connecties kunt luisteren. Hierna roept hij *mmServerSocket.accept* aan totdat dat een keer lukt.
De *ConnectorThread* wordt aangeroepen met een *BluetoothSocket* met connectie, en maakt hierop *FileStreams* en *ObjectStreams* aan. De klasse bevat een functie *writeMove(Move m)* die een move verstuurt, en de *run* wacht zolang de stream open is op *Moves* en stuurt deze naar de *Controller*.

Voor zover werkt de bluetooth: je kunt moves oversturen. Hier ook daadwerkelijk wat mee doen vergde zoveel aanpassing van hoe beurtenafwisseling werkt dat dat op de korte termijn waarop dit alles lukte niet meer ging. Hierbij speelde een grote rol dat er niet vanaf het begin rekening mee was gehouden dat
dit überhaupt zou lukken, en dat Thijs, die dit alles heeft gemaakt, niet het model en *Controller* in elkaar had gezet en hiervan de werking te oppervlakkig kende. Ook is het hele bluetooth-gebeuren voor android vrij slecht gedocumenteerd, en geeft het een heleboel errors. De IOException '*try again*' is een voorbeeld van hoe onduidelijk dit alles in zijn werk gaat.

Chessview

De klasse *ChessView* zet het abstracte model om in een concreet schaakspel dat op het scherm wordt weergegeven. De *ChessView* werkt samen met de controller, die invoer detecteert en daar de juiste handelingen aan verbindt.

De *ChessView* is verantwoordelijk voor het tekenen van de alle afbeeldingen. Om dat te kunnen doen, worden alle afbeeldingen bij het aanmaken van de *ChessView* ingeladen. Alle afbeeldingen zijn van het SVG-formaat, en zijn dus vectorafbeeldingen in plaats van rasterafbeeldingen zoals PNG-afbeeldingen of bitmaps. Om deze afbeeldingen te kunnen lezen en om te zetten naar een voor android bruikbaar formaat, wordt gebruik gemaakt van de AndroidSVG[6] library.
Middels deze library kan een SVG-bestand worden gelezen en direct op de canvas van een view in android worden geplaatst, wat hier gebruikt wordt voor het tekenen van het bord. Daarnaast kan een SVG-bestand ook worden omgezet naar een *Picture*-object, wat een standaard object binnen android is, dat vervolgens op een *Canvas* getekend kan worden in de *onDraw* methode van de view. Dit laatste wordt gebruikt voor het tekenen van de schaakstukken, aangezien deze telkens van positie moeten veranderen.

[6] https://code.google.com/p/androidsvg/

Van elk schaakstuk wordt de SVG-afbeelding eenmalig aan het begin van een nieuw spel ingelezen en omgezet naar een *Picture*. Schaakstukken die vaker voorkomen (pionnen en dergelijke) worden uiteraard ook maar een keer ingeladen. In de *onDraw* methode worden vervolgens de schaakstukken getekend. Voor de leesbaarheid van de code is dit uitgewerkt in een aparte methode *drawPieces*. Hierin worden alle schaakstukken opgevraagd aan de game en met behulp van de methoden uit *Piece* en *Position* wordt de bijbehorende *Picture* van het desbetreffende schaakstuk op de juiste coördinaten getekend.

Naast het tekenen van de schaakstukken moet de *ChessView* ook aangeven of de gebruiker een stuk heeft geselecteerd, en waar een geselecteerd stuk naartoe kan worden verplaatst. Voor beide gevallen wordt de standaardmethode *drawCircle* van het *Canvas* object gebruikt. Hierin worden de coördinaten van de positie en de straal van de cirkel meegegeven, samen met een *Paint*-object dat bijhoudt hoe de cirkel moet worden getekend. Hier worden twee *Paint*-objecten gebruikt: een voor een groene cirkel (het geselecteerde stuk) en een voor een blauwe cirkels (de mogelijke zetten). Voor de leesbaarheid zijn ook deze stukjes code in een eigen methode gestopt, *drawSelectedPiece* en *drawMoves* respectievelijk. Om ervoor te zorgen dat de cirkels onder de schaakstukken worden getekend worden deze twee methoden vóór *drawPieces* uitgevoerd in de *onDraw*-methode.

Ontwerpverantwoording

Onze implementatie van *Singularity Chess* berust op het MCV-model, wat de code overzichtelijker en efficiënter maakt. Op deze manier was los van elkaar aan één aspect van de app werken erg prettig. Het model werkt goed los van android, wat zowel bij het creëren als bij het eventueel verder ontwikkelen erg fijn werkt. Omdat we gebruik maken van abstracte klassen, wordt het model een stuk simpeler en beter algemeen inzetbaar. Verder zijn er een aantal keuzes gemaakt in het ontwerpproces waarvan het voor een buitenstaander misschien niet direct duidelijk is waarom deze het meest effectief is. De twee meest opvallende keuzes worden hieronder verder uitgewerkt en besproken.

Keuze voor vectorafbeeldingen in plaats van rasterafbeeldingen

Android werkt standaard alleen met rasterafbeeldingen. Dit zijn afbeeldingen waarbij aan elke pixel een kleur wordt toegekend en die afhankelijk van het bestandstype met eventuele compressie worden opgeslagen. Vectorafbeeldingen zijn bestanden waarin is opgeslagen hoe de afbeelding gemaakt is. Denk hierbij aan beschrijvingen als *teken een lijn van* (x_0, y_0) *naar* (x_0, y_0), maar dan in xml of een andere formaat dat wat exacter is dan natuurlijke taal. Zo kunnen naast lijnen ook curves en andere vormen of stukjes omschreven worden. Aan zo'n vorm kan dan een kleur(overgang) of rand worden toegekend. Met vectorafbeeldingen wordt dus niet elke pixel beschreven, maar wordt bij het tekenen van de afbeelding uitgerekend welke kleur elke pixel moet krijgen.

Dit heeft een aantal voordelen. Zo is een vectorafbeelding schaalbaar zonder verlies van kwaliteit. Bij het schalen van een rasterafbeelding worden de pixels meegeschaald: er is geen informatie over wat er gedaan moet worden als een pixel op het scherm tussen twee pixels van de afbeelding inzit. Er zijn dan twee mogelijkheden. De eerste is dat de pixel een kleur aanneemt die kan worden gezien als het gemiddelde tussen beide, of hij kiest voor een van beide buurpixels. De afbeelding wordt dan wazig of blokkerig. Bij het schalen van een

vectorafbeelding worden de coördinaten die in het bestand beschreven zijn geschaald. Vervolgens wordt er weer uitgerekend welke kleur elke pixel moet krijgen. Dit zorgt ervoor dat een geschaalde vectorafbeelding haarscherp blijft.

Android probeert dit schalingsprobleem van rasterafbeeldingen te voorkomen door ontwikkelaars van een afbeelding meerdere versies in verschillende resoluties te laten maken. Android zal dan zelf op basis van de schermdichtheid[7] beslissen welke versie van de afbeelding wordt gebruikt.
Dat is een prima oplossing, maar hier zijn nog enkele problemen. De grootste afbeelding zal twee keer de grootte van het origineel zijn, voor de meeste telefoons en tablets is dat nu meer dan genoeg, maar in de nabije toekomst zal dit zeker veranderen. Daarnaast houdt het geen rekening met de eventuele wens van de ontwikkelaar om afbeeldingen om andere redenen te schalen. In deze app komt deze wens concreet naar voren: niet alle telefoons hebben eenzelfde breedte-hoogte-verhouding, desondanks is het wel van belang dat het schaakbord zo groot mogelijk wordt weergegeven, waarbij breedte en hoogte met gelijke factor geschaald worden. De schaakstukken moeten hierin meegeschaald worden. Het kan echter zo zijn dat de gewenste schalingsfactor niet overeenkomt met de door Android op basis van screendensity bepaalde schalingsfactor.
Daarom heeft de keuze voor vectorafbeeldingen in onze app daadwerkelijk toegevoegde waarde. De standaardoplossing is vooral bedoeld om een vaste afbeelding door Android zelf te laten schalen. Vectorafbeeldingen kunnen veel uitgebreider geschaald worden. Zodoende zijn vectorafbeeldingen een betere keuze dan de standaard Android oplossing als een ontwikkelaar volledige controle wilt houden over hoe afbeeldingen geschaald worden, ook als het juist niet om verschillende schermresoluties maar andere redenen te doen is.

Dit betekent echter niet dat er geen nadelen aan deze keuze kleven. Zo worden vectorafbeeldingen standaard niet ondersteund door Android. Voor de meest gangbare implementatie van deze afbeeldingen, SVG-afbeeldingen, bestaan echter enkele libraries. De voor deze app gekozen library, *AndroidSVG*[8], werkt met alle gangbare SVG-elementen en is voldoende gedocumenteerd om mee te werken. Daarmee is een ontwikkelaar wel afhankelijk van een derde partij (de maker van de library) of op zichzelf aangewezen als hij ervoor kiest zelf een implementatie te ontwikkelen.

Bijkomend nadeel van het gebruik van de hiervoor gekozen library is ook dat de uit de SVG gemaakte Picture objecten niet op alle telefoons goed getekend worden. Dit komt doordat de drawPicture methode van een Canvas-object niet goed werkt wanneer hardwareversnelling aanstaat. Het is ook zeer onwaarschijnlijk dat Google hier nog wat aan gaat doen: in de documentatie[9] staat dat voor hardwareversnelde Canvas-objecten de methode drawPicture niet ondersteund wordt.
Daarom moet voor het gebruik van deze library de hardwareversnelling van de view waarin getekend wordt worden uitgeschakeld. Dit bekent ook dat daarmee geavanceerde grafische functionaliteiten van android niet langer bruikbaar zullen zijn, of veel trager zullen werken.

[7] http://developer.android.com/training/multiscreen/screendensities.html
[8] https://code.google.com/p/androidsvg/
[9] http://developer.android.com/guide/topics/graphics/hardware-accel.html#unsupported

Dit levert in deze app echter geen problemen op omdat al het tekenwerk hier beperkt blijft tot simpele tweedimensionale afbeeldingen.

De keuze voor het opslaan van de bordstructuur in een array.

Zoals al eerder beschreven hebben we de structuur van het bord, dat wil zeggen welke velden liggen naast welke velden, opgeslagen in een driedimensionale array. Dit is natuurlijk niet een heel charmante, mooie oplossing, maar het bleek na enige tijd toch de meest efficiënte te zijn.
Bij normaal schaken zou een bord gerepresenteerd kunnen worden door een tweedimensionale acht bij acht array. De structuur is dan snel aangebracht door voor een rechte zet naar rechts, boven, links of onder respectievelijk één op te tellen bij de kolom, één op te tellen bij de rij, één af te trekken van de rij of één af te trekken van de kolom. Ook schuine zetten kunnen op eenzelfde, simpele manier worden beschreven.
Het leek ons mooi om voor deze app voor het ronde bord een soortgelijke formule te maken die rechte en schuine zetten overzichtelijk en bruikbaar kon beschrijven. Dit bleek echter na flink puzzelen met de hulp van wat basale lineaire algebra niet mogelijk te zijn. De formules die we (gedeeltelijk) hebben gevonden, waren dusdanig lang en ingewikkeld, dat het korter en mooier zou zijn om de structuur hardcoded in een array te zetten.

Reflectie

De basis van de app werkt volledig naar behoren: spelers kunnen op een telefoon tegen elkaar singularity chess spelen, zonder enige technische problemen. Het is gelukt om de complexe en soms structuurloze zetten die mogelijk zijn te verwerken tot uitvoerbare code zonder dat er fouten worden gemaakt. Daarnaast is de app ook goed afgewerkt met een hoofdmenu, instellingen, bericht wanneer het spel is afgelopen en een simpele maar nette visuele stijl.

Dat ging niet altijd zonder horten of stoten. Vooral het (theoretisch) ontwerpen van een goed werkend model bleek een grote uitdaging, waarbij uiteindelijk gekozen is voor de hierboven beschreven ‘hardcoded’ oplossing. Dat betekent echter niet dat we ontevreden zijn over de huidige oplossing. Het werkt effectief en kon ook nog geoptimaliseerd worden door het spiegelen van een deel van het bord waardoor een groot deel van de data niet handmatig uitgezocht en ingevoerd hoefde te worden. Bovenal bleek dit een goede les in het analyseren van een vrij complex systeem waar je eigenlijk nooit eerder over hebt nagedacht.

Ook het tekenen van verschillende afbeeldingen bleek een grotere uitdaging dan gedacht. De keuze maken om SVG afbeeldingen te gebruiken is makkelijker dan de implementatie van die keuze. Een library vinden was niet zo moeilijk, maar ook dan zijn er nog verschillende problemen, bijvoorbeeld het boven beschreven probleem met de hardwareversnelling. Verder zijn er dan altijd kleine problemen die in eerste instantie best lastig op te lossen waren, maar waarvan de oplossing achteraf vrij logisch is.

Het grootste gemis is natuurlijk de bluetooth functionaliteiten. Ondanks dat we hier een flink eind mee zijn gekomen bleek het te lastig voor een goed functionerende implementatie in de

uiteindelijke app. Toch weten we nu wel al hoe we op z'n minst twee telefoons met elkaar kunnen laten communiceren via bluetooth. Deze kennis en de (junk)code die nog in oude versies van onze app staat, zullen zeker van pas komen in eventuele toekomstige projecten.

De problemen die zich voordeden hebben het geheel echter nauwelijks vertraagd doordat het werk goed opgedeeld was in onafhankelijke stukken. Wanneer een van ons niet meteen verder kwam, kon de rest wel door, waardoor de app geen vertraging heeft opgelopen. Dat had wel als nadeel dat er ook een moment kwam waarop alle losse onderdelen aan elkaar geplakt moesten worden. Dan blijkt goede communicatie en documentatie cruciaal, aangezien iedereen perfect weet hoe zijn eigen deel werkt, maar nauwelijks iets weet van de andere delen. Daaruit kan ook worden geleerd dat het hebben van een goed plan voordat je begint een hoop ellende kan besparen. Door de concrete opdeling in verschillende delen was meteen duidelijk welk wie waarvoor verantwoordelijk was en dus ook in welke klasse een bepaalde methode staat. Deze werkwijze is ons zeker bevallen en we zullen dit ook proberen te handhaven bij komende projecten.

Een van onze doelen voor deze app was om het prettig speelbaar te maken. Dit is met het toevoegen van een aantal instellingen naar onze mening gelukt. Ook het duidelijk aangeven van de mogelijke zetten ziet er in onze ogen keurig uit. Op dit aspect zijn we erg tevreden met het uiteindelijke resultaat.

De app werkt naar behoren zonder enige problemen. De ontwikkeling is voltooid binnen de gegeven tijd en kan daarmee als succesvol worden gezien. Een goede planning en verdeling van het werk hebben daar sterk aan bijgedragen en zullen nog zeer handig blijken bij andere projecten. De enige domper op dit succesverhaal is het niet geslaagd implementeren van bluetooth. Daar ligt dan ook een mooie kans voor verdere ontwikkeling. Het spelen op één telefoon is leuk, maar idealiter zou men met twee verschillende telefoons asynchroon willen spelen via internet.