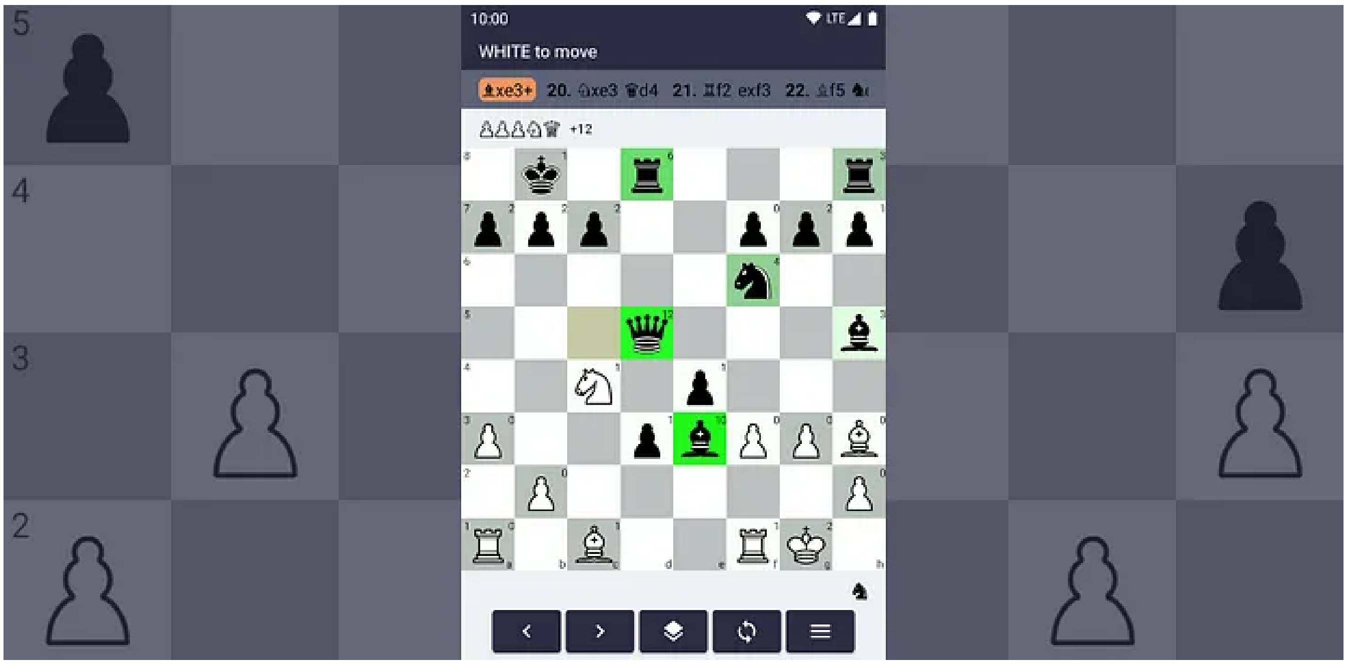


Checkmate on Compose — Part II

Zsolt Kocsi · Follow
Published in Bumble Tech
10 min read · Sep 2, 2021

Listen Share

Lessons learned from a Jetpack Compose-based chess app

A sneak peek at what this two-part article series is about

Chesso is a Jetpack Compose-based chess app with animated visualisation layers on top of the board: https://github.com/zsoltk/chesso

In this two-part article we explore the principles on which it was built:

- **Part I** (previous article): fundamentals of the game and UI
- **Part II** (this article): animations and visualisations

Piece animations — running into troubles

We left off at the end of the first article with having finished the core elements of the app. Now it's time to start adding some fancy bits!

One of the visually appealing features of chess apps is when piece moves are animated — but in the beginning, it wasn't obvious how to make it work. Rather than jumping straight to the end result, let me walk you through my thought process in tackling the challenge.

The main issue is that the piece being moved ends up in different positions in the hierarchy of composables, and

> There is no shared element transition available in Jetpack Compose at the time of writing this article.

But then — I thought — maybe I don't need it after all. Couldn't I just animate each piece manually? It couldn't be that complicated, could it?

The good news is that Compose has an *offset* modifier that could be used to… well, offset the position at which a composable is rendered. Halfway there!

Also on the plus side, Compose does not clip when drawing outside of the "default" bounds (as would be the case with old-school, XML-based views), so there's no need to worry about that either.

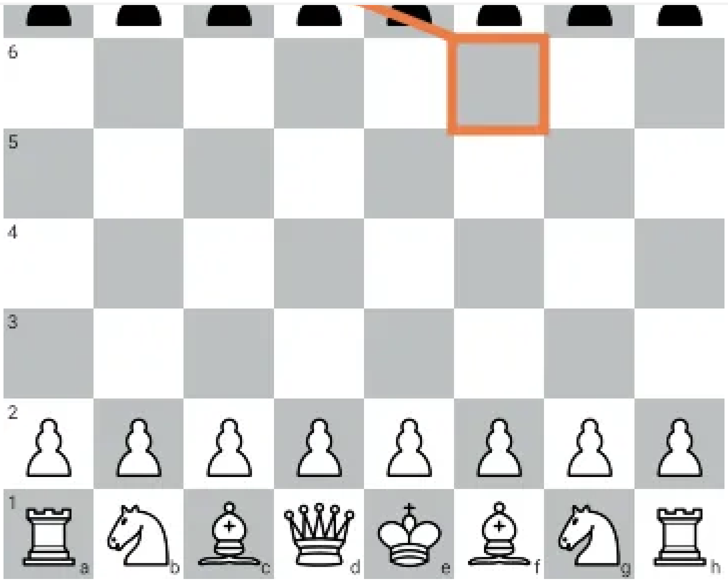
One look at the official docs on animation, and you'll see that it's super simple to animate a single value. So I figured I could certainly animate a piece by animating the value passed to the offset modifier — if I only knew *where to animate the value to.* So how could I calculate that?

Refactoring the board rendering

At this point, my initial approach on rendering the board was becoming a limitation. (If you recall, I used equal weights to render an **8x8** board filling up the screen space — consequently, I delegated the calculation to Compose and I couldn't know which position any given square was rendered at.)

So I decided to take the whole thing apart and start from the other end:

1. Measure the available horizontal space
2. Divide it by **8** to get the width of a single square
3. Instead of a *Column* / *Row* hierarchy, draw all the **64** squares in the top left corner as a default
4. Modify this rendering position by an offset
5. Calculate the offset based on two elements: the board position and the size of the square calculated in **step 2.**



f6

For example:

- Let’s take the square at the **f6** position on the board
- That’s the 6th file (= 6th column from the left) at the 6th rank (= 3rd row from the top)
- The coordinate relative to the top left corner is then: (6–1, 3–1)
- The offset(x, y) relative to the top left corner is then: **(5 * square size, 2 * square size)**

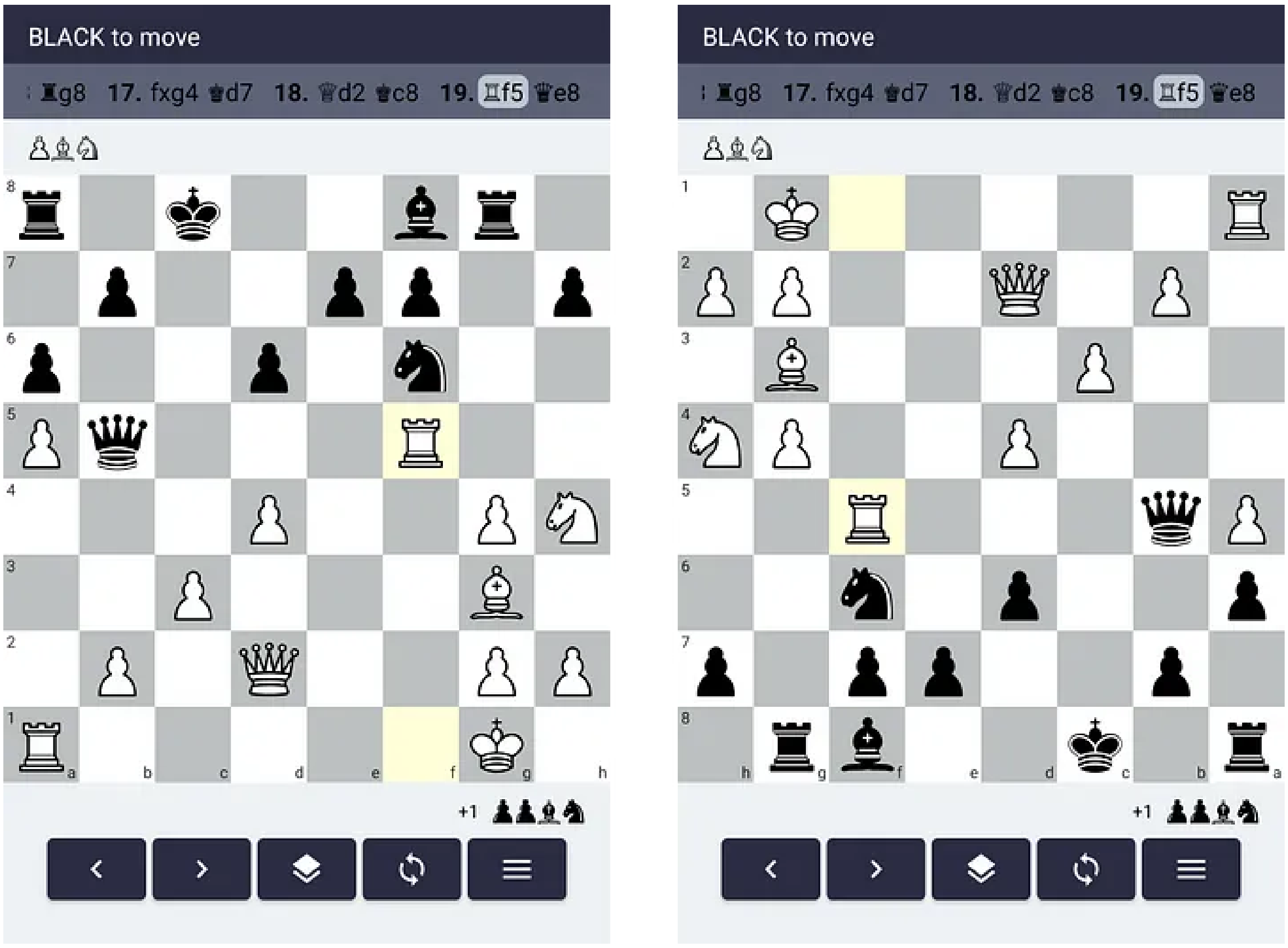
But how do we get the measured size at **step 1**? *BoxWithConstraints* to the rescue, which exposes the *maxWidth* and *maxHeight* properties:

```kotlin
BoxWithConstraints(
    modifier = Modifier
        .fillMaxWidth()
        .aspectRatio(1f)
) {
    val squareSize = maxWidth / 8
}
```

chess-box-with-constraints.kt hosted with ❤ by GitHub view raw

Flipping the board

This offset-based approach also made flipping the board extremely easy: we can calculate the offsets relative to either the top left or the bottom right corner based on a boolean value — and that's it!


BLACK to move
♜g8 17. fxg4 ♚d7 18. ♕d2 ♚c8 19. ♖f5 ♛e8
+1
BLACK to move
♜g8 17. fxg4 ♚d7 18. ♕d2 ♚c8 19. ♖f5 ♛e8
+1

The board and its flipped version

Piece animations — the actual animations

Following our method of calculating offsets, we can obtain the offsets both for the previous and the current game states. (Bear in mind that there might not be a previous one in some cases, e.g. at the beginning of the game!)

We can then create the animation. Handling the case when there’s no previous state, we can immediately use the target value as the starting point, and we have a graceful fallback with no animation:

The *LaunchedEffect* starts the animation to the *targetOffset* value (and relaunches it every time *targetOffset* changes).

The animated offset value then gets passed as a modifier to the *Piece* composable.

Animating all the pieces

A fun consequence of all this is that once we can animate a single piece, we can animate all the pieces between *any two arbitrary states* of the game:

However, flipping the board also changes the offsets of the pieces, which is then automatically animated — this means that we'll see all the pieces flying across the board each time, which isn't quite what I wanted:

To avoid this we can simply add a second *LaunchedEffect* to run every time the value of *isFlipped* changes, and immediately snap the animation to its end value:

Extensible rendering layers

Up until this point all the Composables I wrote were embedded into each other the usual way.

However, I wanted to introduce some flexibility here so that visualisations (and really, any kind of generic decoration) could be easily added via separate classes.

I came up with a few classes. First, I captured all the information that rendering would need from UI state on individual squares to game state regarding the whole board:

Next, I created interfaces that can emit UI using the above data structures:

For example, the responsibility of a single **SquareDecoration** can be to paint a square's background; or add a label to it, or add some highlight (topics we covered in the first article) —but not more than one at a time to keep them nicely separated.

A single **BoardDecoration** can be to paint squares (decorated, using **SquareDecorations)**, or to add pieces.

One decoration each would surely not be enough — we're going to need many. So renderer interfaces are introduced, which are basically glorified lists of decorations:

The default implementations are for example:

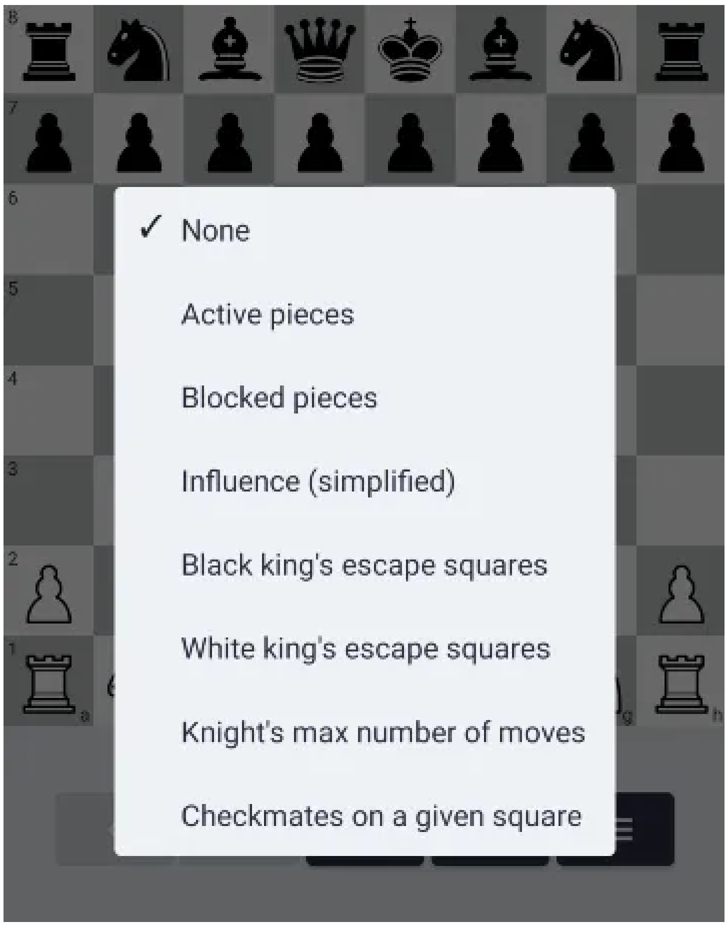
And finally, the client code — the refactored **Board** composable, using the above in practice:

Benefits of using this pattern:

- The board assumes nothing about rendering and delegates the task
- Were there multiple possible implementations we could inject them via either a CompositionLocal or DI
- The decorations themselves are nicely decoupled
- We have an ease of extensibility: just add a new decoration to a list, or replace the whole list altogether

Dataset visualisations

Now that we have a fully functional chess app with the rendering layers in place, we can add the visualisations that were the motivation for creating the app in the first place.


✓ None
Active pieces
Blocked pieces
Influence (simplified)
Black king's escape squares
White king's escape squares
Knight's max number of moves
Checkmates on a given square

Some visualisations that can be added to the app

Goals

- To have a dialog to pick an active visualisation (or none)
- Visualisations to be possibly (but not necessarily) based on the current game state
- For those based on the current game state: to animate nicely along with the game itself

Rough plan

All visualisations will be based on a numerical dataset:

1. We'll associate some kind of numeric value to each of the squares (this can be anything based on what we want to visualise) — this will be a data point

2. We'll look at where this value fits on an expected min/max range for the whole of the board
3. Based on this, we'll calculate an interpolated colour on a colour range (corresponding to the previous min/max values)
4. We'll overlay this colour on each square on the board

Implementation

Let's consider this interface:

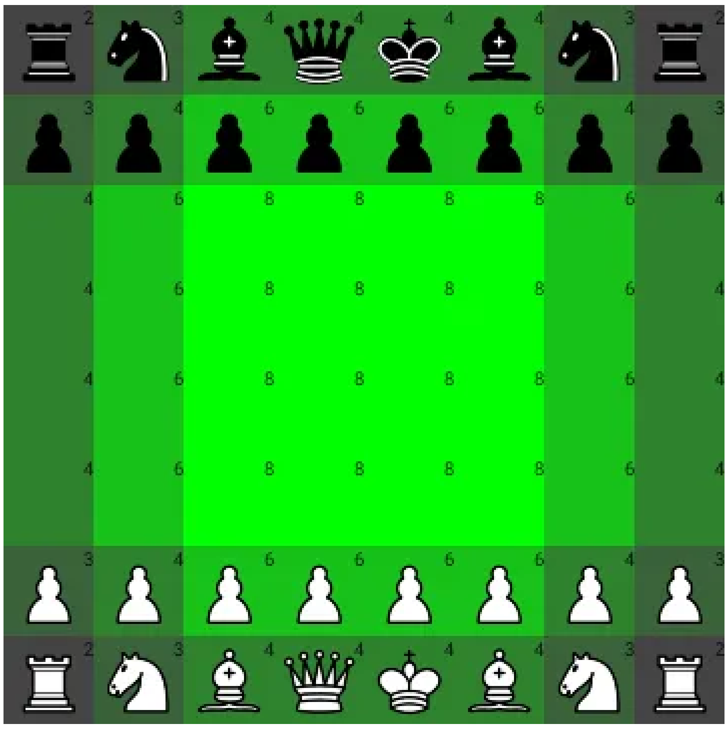
We can create a new **SquareDecoration** using these interfaces.

It uses an *ActiveDatasetVisualisation* CompositionLocal to fetch the active visualisation which the user sets in the dialog:

Static visualisations

We can then leverage this API quite easily. Consider these:

Knight’s move count



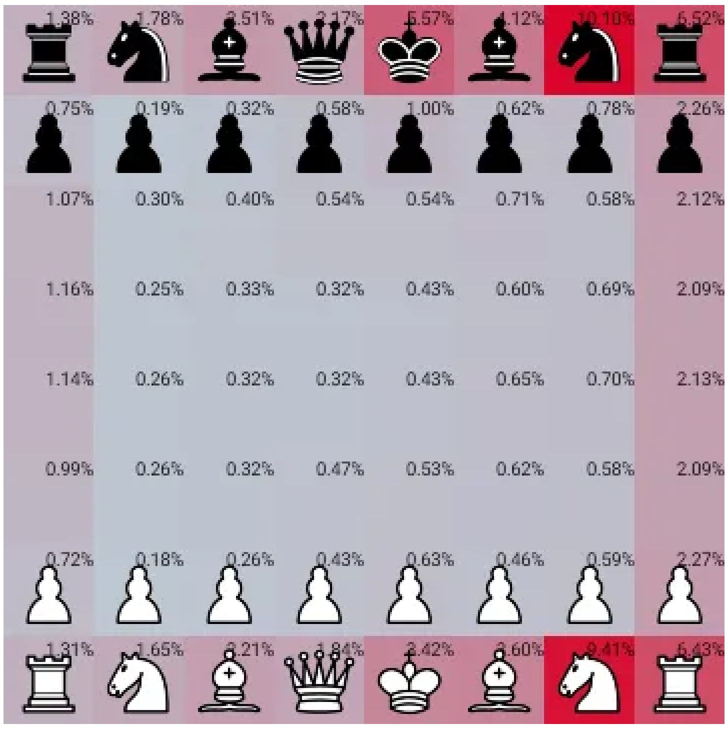
A simple visualisation for beginners that shows in which squares the knight is most useful.

This is the code:

And this is how it works:

- Since this visualisation is static (not dependent on the game state), we can provide datapoint values as int literals.
- A single datapoint describes the maximum number of legal moves the knight can make from any given position.
- The minimum value is 2 (when the knight is in a corner), the maximum is 8 (when the knight is in the central area).
- The 2–8 range is then used to interpolate the colour of any given square based on its own value between a dark grey and a shade of green.

Checkmate statistics



This one's based on a post I ran into on Reddit, and shows the percentage of checkmates that occurred on any given square, based on a million actual games from Lichess. Credit goes to /u/atlas_scrubbed!

The implementation follows the same logic as the Knight's move count, so I will skip the code here.

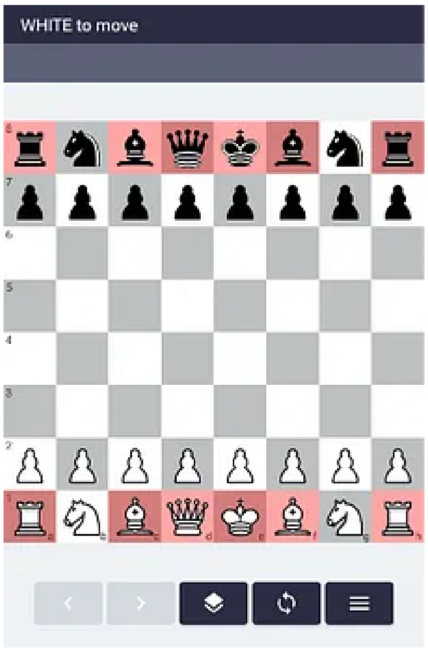
Dynamic visualisations

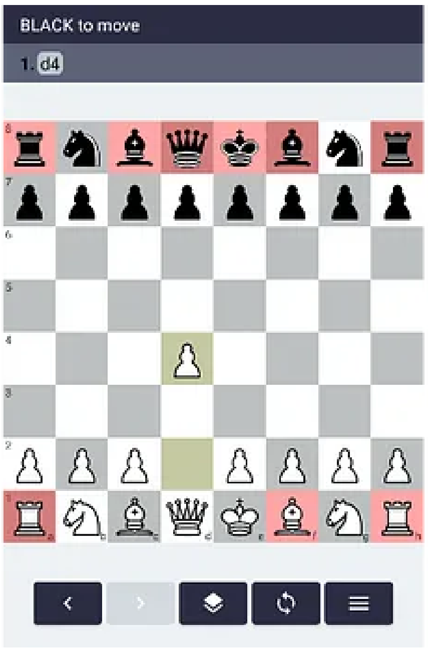
These depend on the current state of the game and will animate along with moves.

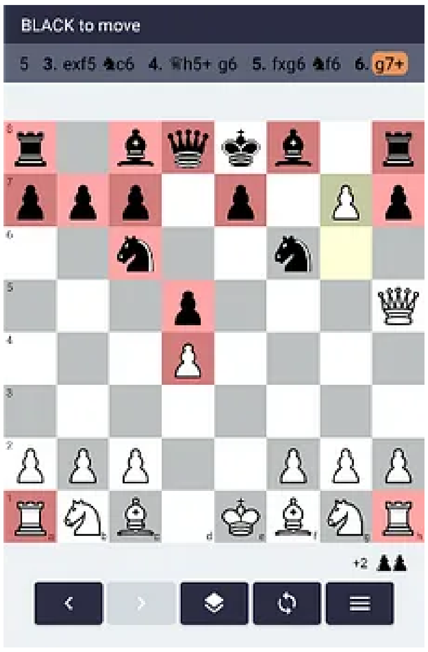
Blocked pieces

Highlights those pieces which can't make a move in the current game state.

For example:







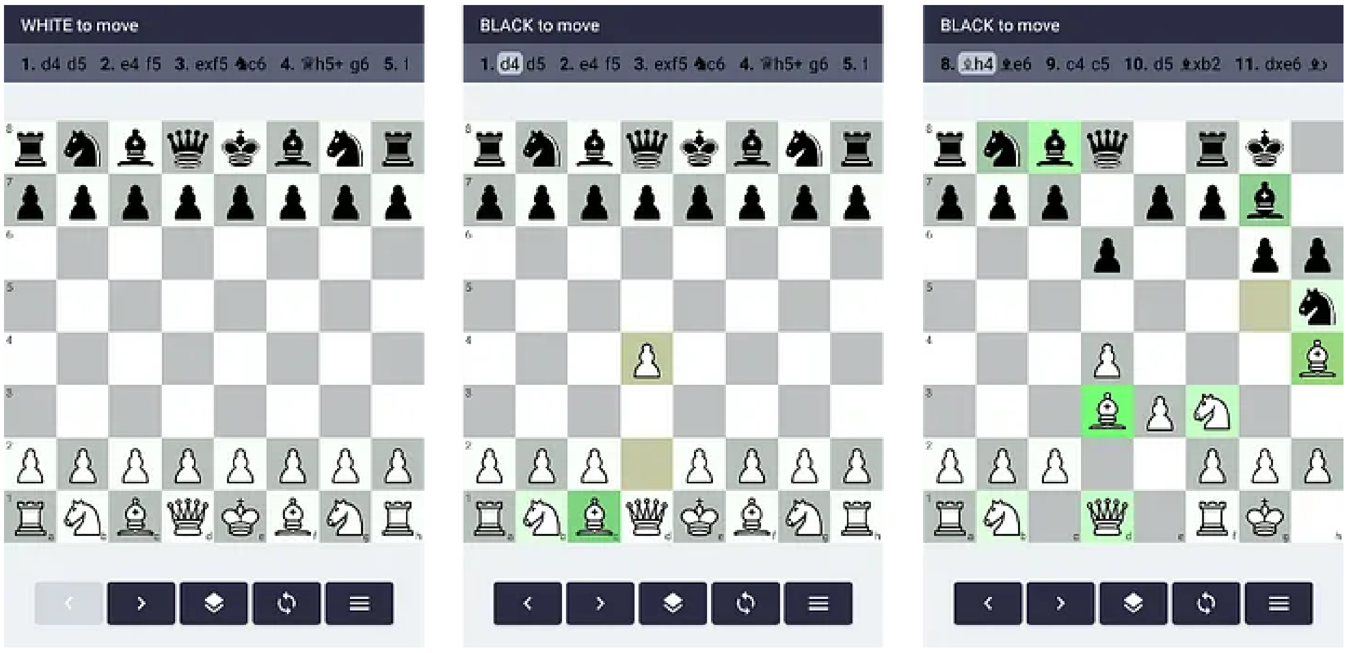
1. First image: At the beginning of the game, only pawns and the knights can move, all other pieces are blocked by some other piece.
2. Second image: After **d4**, white's bishop, king and queen have gained some squares where they can move to, so they're no longer marked with red
3. Last image: black's king is in check by white's queen on **h5**. The only pieces not marked with red are: a) black's king (can escape to **d7**), and b) black's knight on **f6** (can take white's queen). No other black piece is allowed to make a move since they wouldn't remove the check.

How does this work?

This one was actually quite easy to implement based on the already implemented game mechanics (explained in the first article)! All we need to do is:

1. Take the number of legal moves for each piece (reminder: legal move = pseudo-legal moves + check constraints applied)
2. If this number is 0, we'll consider the piece blocked

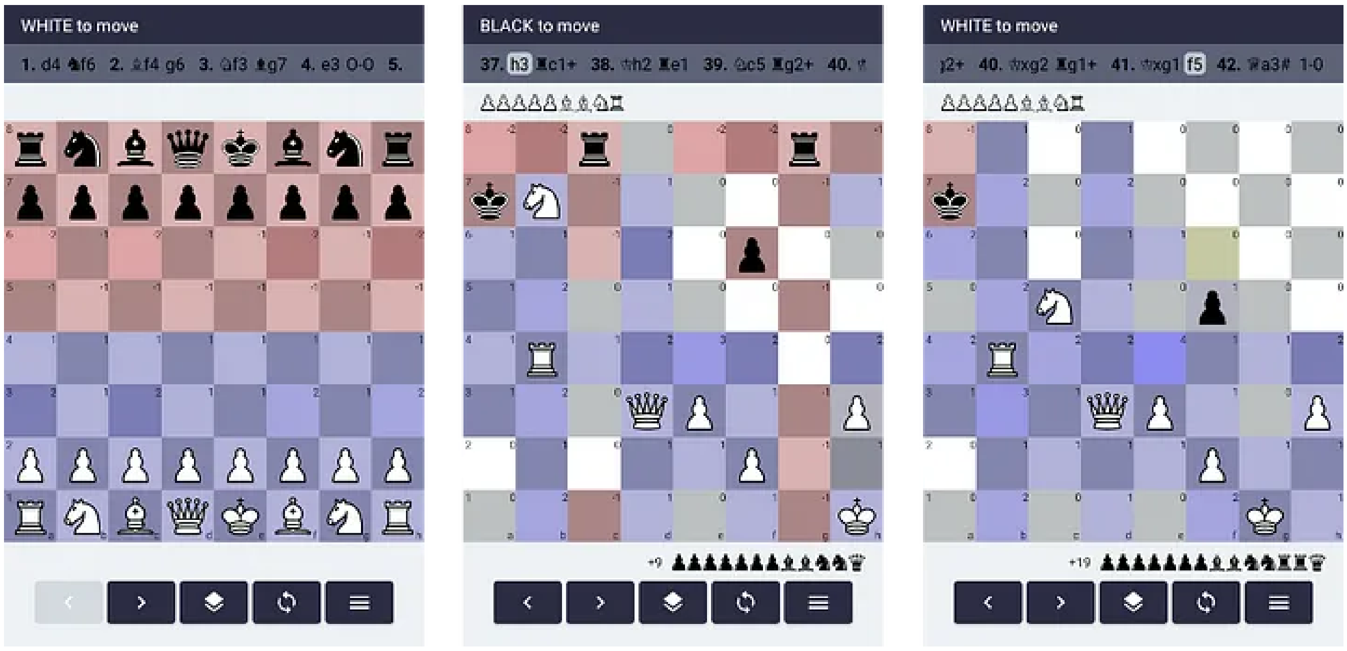
Active pieces


WHITE to move
1. d4 d5 2. e4 f5 3. exf5 ♞c6 4. ♕h5+ g6 5.
BLACK to move
1. d4 d5 2. e4 f5 3. exf5 ♞c6 4. ♕h5+ g6 5.
BLACK to move
8. ♗h4 ♝e6 9. c4 c5 10. d5 ♝xb2 11. dxe6

This one is almost identical in implementation to blocked pieces — except now we visualise those pieces which have legal moves instead of the ones that don't. The more they have, the stronger the green colour we apply.

It's a great way to demonstrate to beginner players which pieces aren't really doing much on the board.

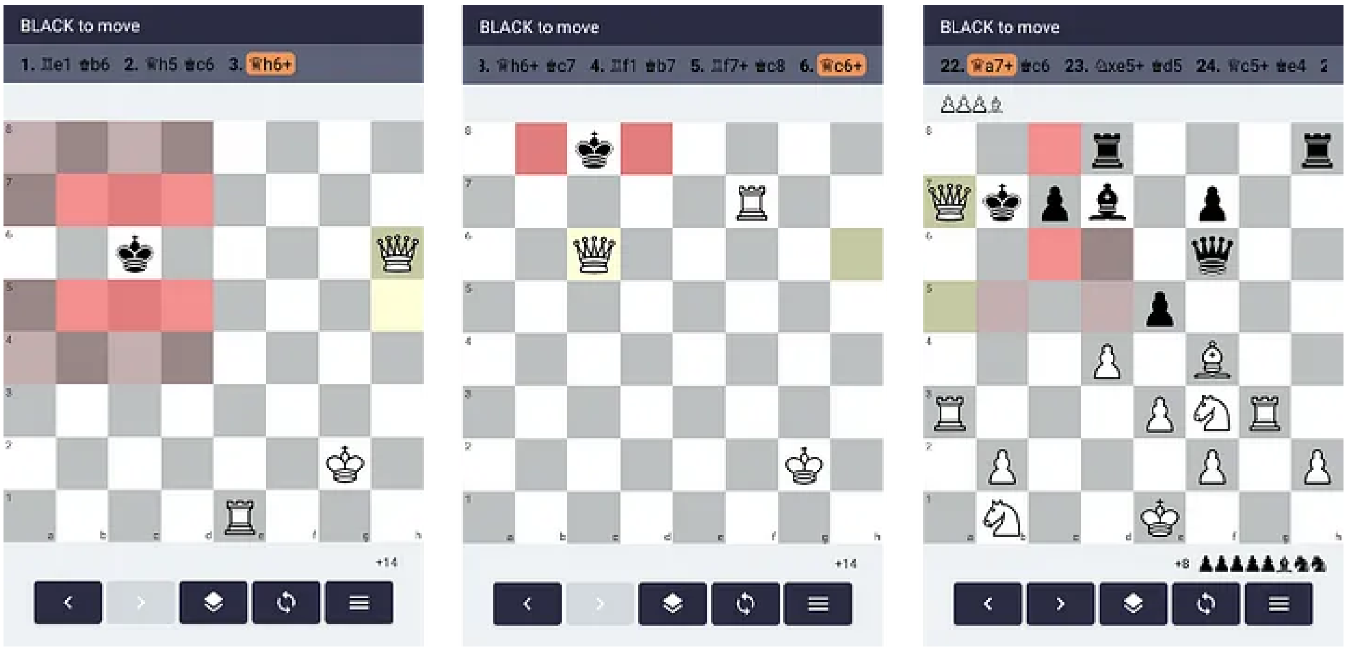
Influence (simplified)


WHITE to move
1. d4 ♞f6 2. ♗f4 g6 3. ♘f3 ♝g7 4. e3 O-O 5.
BLACK to move
37. h3 ♜c1+ 38. ♔h2 ♜e1 39. ♘c5 ♜g2+ 40.
+9
WHITE to move
g2+ 40. ♔xg2 ♜g1+ 41. ♔xg1 f5 42. ♕a3# 1-0
+19

- Calculates how many pieces can move to a square for each of the players
- Uses a different colour scale per dominating side

- Does not consider defenders of a square or more complex scenarios (e.g. considering relative values), hence it's dubbed "simplified". It's great for showing available space though.

King's escape squares


BLACK to move
1. ♖e1 ♚b6 2. ♕h5 ♚c6 3. ♕h6+
+14
BLACK to move
3. ♕h6+ ♚c7 4. ♖f1 ♚b7 5. ♖f7+ ♚c8 6. ♕c6+
+14
BLACK to move
22. ♕a7+ ♚c6 23. ♘xe5+ ♚d5 24. ♕c5+ ♚e4
+8

A useful visualisation when trying to corner the king. Brighter coloured squares show the king's immediate moves, paler ones show possible moves from those squares.

The power of Compose animations

Compose comes with an extremely powerful animation system that relieves the developer of an enormous amount of effort.

I'd like to highlight two of its important features in the context of this article:

1. All the animations defined happen independently of each other
2. The animation system can start seamlessly animating towards new target values at any point

To demonstrate all this, I slowed down the animations quite a bit:

Here you can see:

1. We jump to a move in the game history, but before the pieces can finish moving to their new squares we jump to yet another new move — and the pieces simply change direction towards their new targets.
2. When the selected visualisation layer is changed, squares seamlessly animate their colours to new values both based on the new visualisation layer and the new game state.

Calculations, performance, vararg keys

I accidentally introduced some unwanted performance hits by forgetting how these factors interact:

1. Datapoint calculations can be based on game state and can be resource-intensive (e.g. when calculating the number of possible moves for all the pieces).
2. When the pieces are animated, Compose will of course re-render the UI for each interpolated value in the animation

This is a problem because it fetches data points for every frame of the animation:

Instead, we need it to calculate only once per game move, and to be recalculated in two cases only:

1. When the selected active visualisation changes
2. When the game state itself changes (by making a move or traversing the moves list) → as a consequence, the passed in properties will change too.

We can express this easily. *Remember* comes in really handy with its vararg keys:

There are many different methods in Compose (*remember, LaunchedEffect,* etc.) that allow passing generic vararg keys to tell the framework when a certain block should be re-run / re-calculated — a brilliant mechanism in all its succinctness.

And by fixing the above, now — of course — there's a significant improvement in UI performance too.

That's all folks!

Thanks for reading these articles! I hope you've enjoyed them as much as I enjoyed creating this project. I found it had just the right level of challenge, and I certainly learned a lot in an entertaining way. Compose is definitely a lot of fun, a fun that we all needed in the world of Android — and this is just the beginning.

Found a bug?

Got an idea of how to do something better than I did?

Got a ground-breaking idea for a new visualisation layer?

Come over to https://github.com/zsoltk/chesso and open an issue!

Android   Android App Development   AndroidDev   Jetpack Compose   Chess

Follow

Written by Zsolt Kocsi

450 Followers · Writer for Bumble Tech

Principal Android Engineer @ Bumble

More from Zsolt Kocsi and Bumble Tech