SpeedStocks

Background

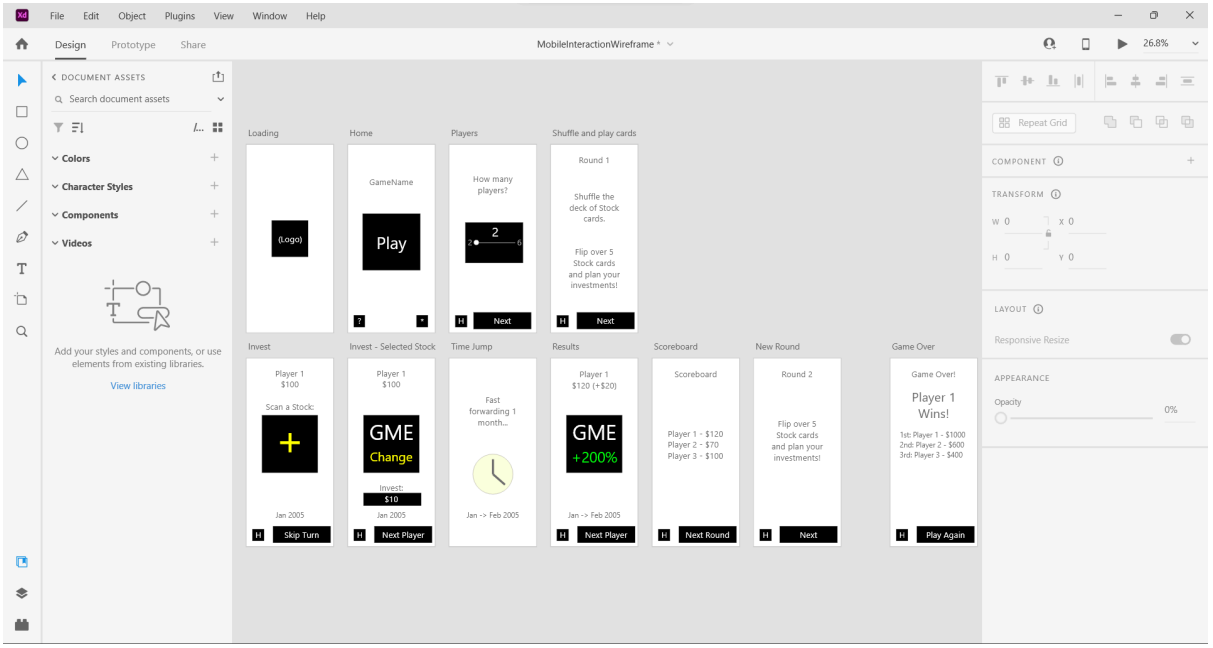
Overview

SpeedStocks is a mobile application and card game where 1-5 players invest in stocks and get returns based on recent real-world stock market data. Players scan QR codes on cards to select the stock they wish to invest in. The app provides a gamified solo or competitive experience while teaching the player about recent day-to-day stock trends.

Aims

I aimed to develop an engaging, educational game that gave the average player an accessible taste of a financial field like currency exchange or the stock market. I also wanted to implement a physical QR code mechanic to create a dynamic tabletop game, bridging the gap between a tactile, friendly experience and in-depth stock information. This mechanic would also be open-ended to allow for future additions and multiple modes of play. The app would follow a linear structure that could easily be paused and resumed. The app would also have minimal loading times and provide feedback when encountering issues.

Experience Design



Unlike other stock apps that facilitate trading and watching the market, this app attempts to simplify and gamify the stock market to introduce stock trading to new audiences. The game uses cards representing stocks to simulate a tabletop card game and garner interest from tabletop, casual, and family gamers. The game is accessible to younger audiences with large bright buttons, a playful display font, limited gameplay options per round, and the ability to scan cards instead of manually typing their stock symbols.

QR codes offer a unique tabletop gameplay opportunity: the function of a card with a QR code can change based on code. In *SpeedStocks*, cards' value varies based on the in-game date, taken from recent real-world data. This means playing the game the same way at different times will yield different results. The game initially centred around currency exchange, but I switched to stocks for simplicity, replayability, and real-world applicability.

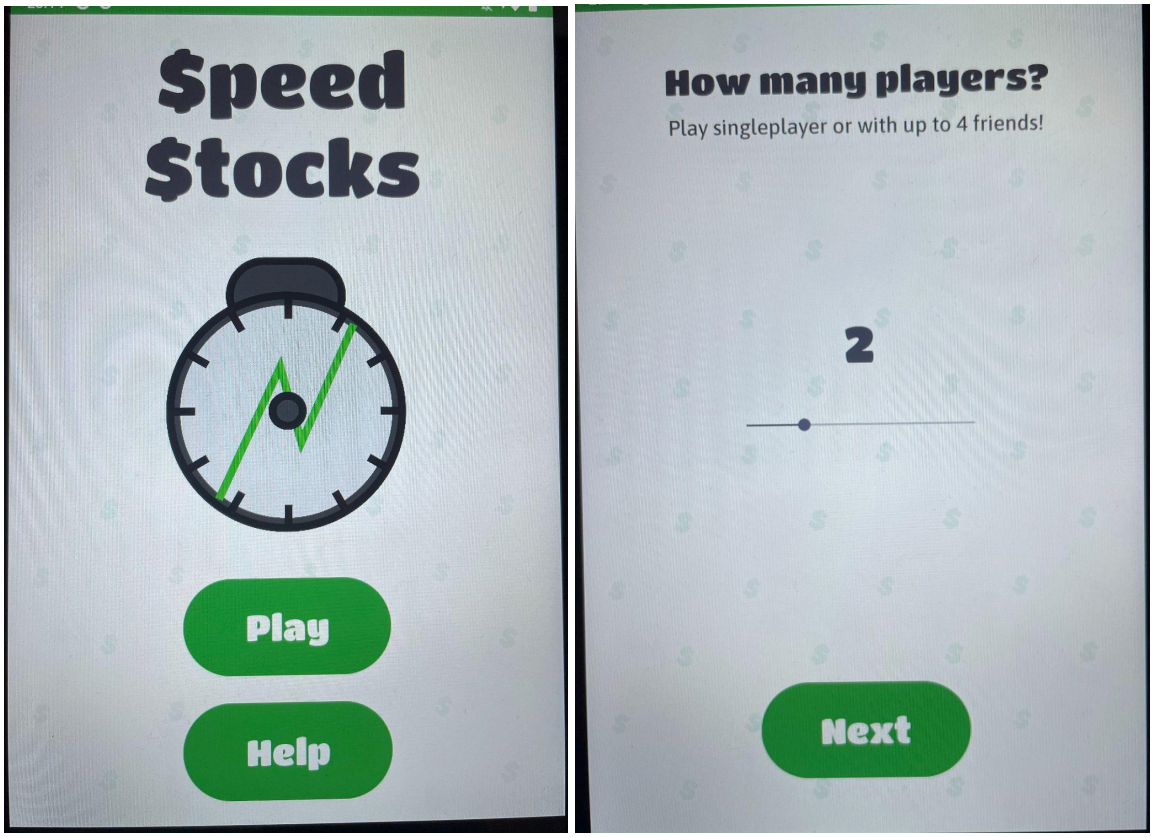
In-game investment returns will tell players which stock cards have been doing well in the past couple of months; should they become interested in the stock market after playing the game, this may give them an idea of where to start investing. The stocks featured on the cards are the most successful S&P 500 stocks from the last 20 years (Lu, 2023). While futureproofing physically-represented stocks is impossible, these stocks should remain successful in the coming years and keep the game enjoyable. The cards are designed to be very simple to keep most game logic app-side. This allows additional gamemodes with different win conditions and levels of complexity to be easily implemented in the future.

```
//amount to inflate the percentage of change between stock closes across two dates
//this inaccurately represents stock fluctuations...
//but exaggerates day-to-day changes to be more influential/fun in-game
//1 = no change, <1 = less change, >1 = more change. e.g. 1.2 = 20% more exaggerated
public val percentageMultiplier: Float = 1.2f
```

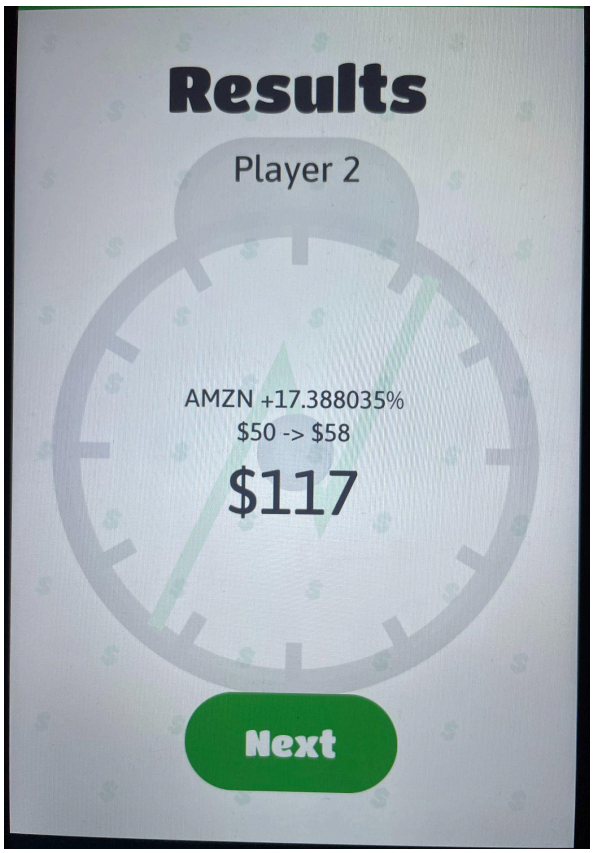
Due to day-to-day stock fluctuations being relatively insignificant most of the time, the rate of change is artificially exaggerated by +20%. While this misrepresents the accurate

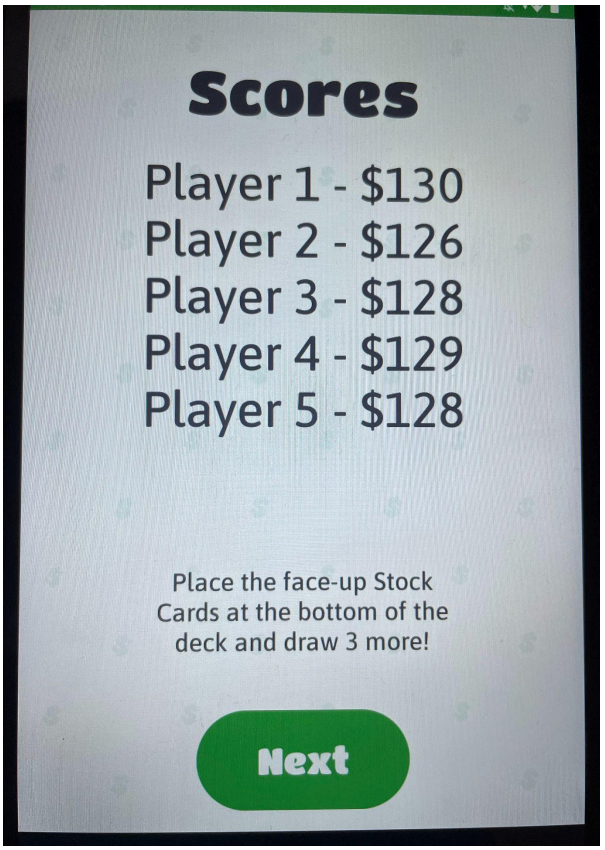
data received from *marketstack*, it more closely aligns the game with my goal of providing an accessible and engaging experience for the average player.

Gameplay



The game opens to a loading screen before opening the home screen. From here, players can open a one-off activity with gameplay rules or start the game. On pressing play, the player uses a seekbar to set the number of players and is prompted to shuffle the deck. Then, each player clicks the camera button on the investments screen to scan a card's QR code. The app will recognize the card by its QR code. The player chooses how much to invest and passes the turn to the next player.



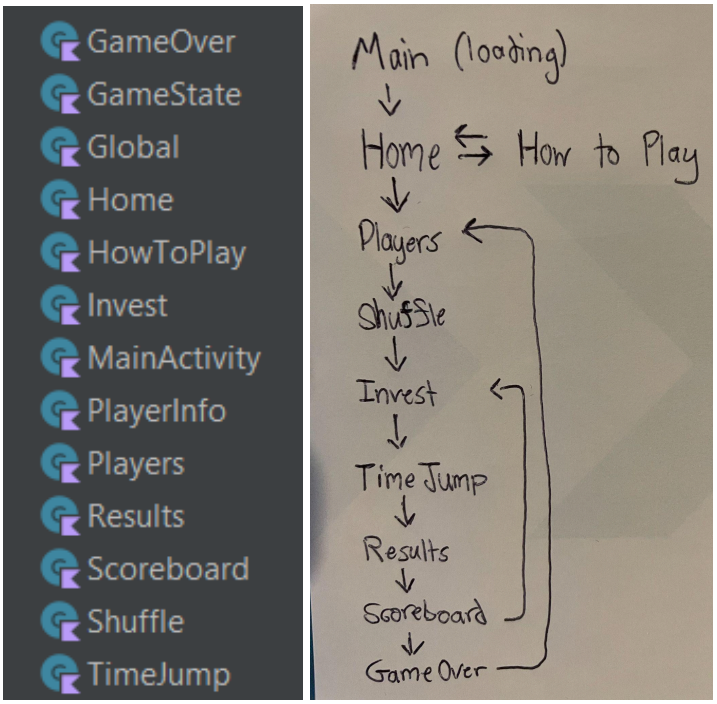


Once all players have invested, time will “jump ahead” by a day or more. After this time, each player can see their results, including their chosen stock’s percentage of change, their investment’s increase in value, and their final balance. A final screen summarizes all players’ scores before returning to the investment page for the next round. After five rounds, the game ends, and the player with the highest balance wins.

Description and Justification

Technical Overview

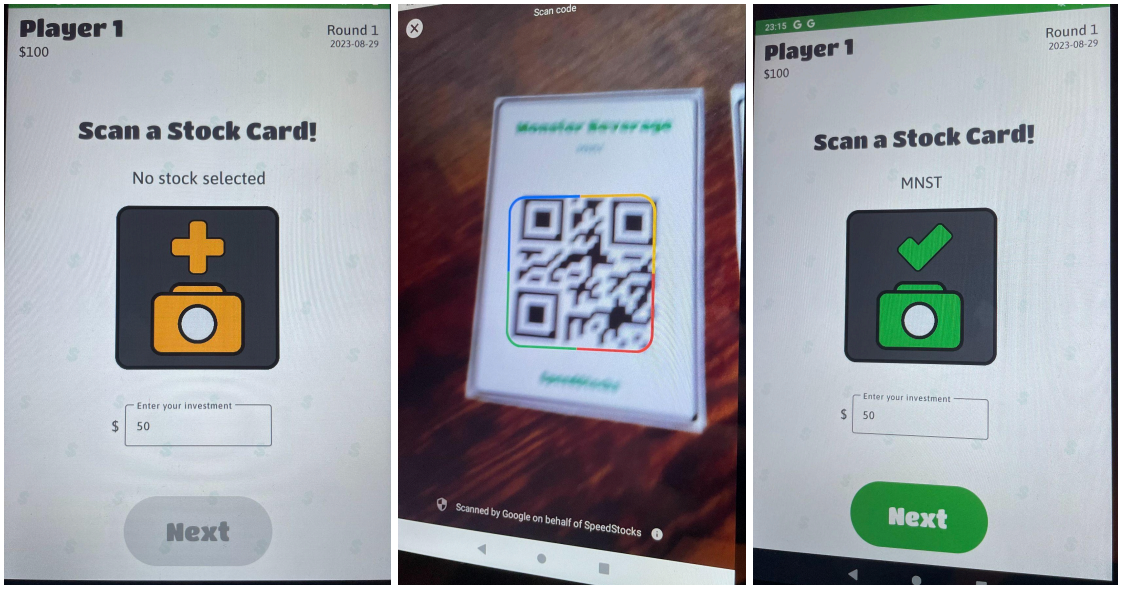
OkHttp tests internet connection and connects to two APIs: *Google code scanner* and *marketstock*. *Google code scanner* scans QR codes through Google Play Services, meaning the app requires no camera permissions. *marketstock* provides recent stock data for requested stock symbols (e.g. AAPL for Apple). Assuming the Android device has Google Play Services updated, the only permission the app requires is access to the internet, which it should have by default.



SpeedStocks operates along a linear chain of intents and activities consisting of introductory activities, a central loop, and an end-game activity. Most activities can be paused and returned to later with no issue. The only exception is when awaiting data from an API request in the TimeJump Activity, where the app will re-request data on onStart() until the data is retrieved while the app is open and visible.

The only potential ethical issue is accessing the camera without the user explicitly granting permission to *SpeedStocks*. However, the app has no direct access to the camera feed and is only given the QR code data.

Investments



In the Invest Activity, players use *Google code scanner* to read stock symbols from cards' QR codes. The scan button changes after successfully scanning to indicate the symbol has been read. When players enter their desired investment amount in the text field, it is clamped to the valid range between the minimum valid investment and the player's total balance. If the entered value is not a number, it will be replaced with the default value.

API Requests

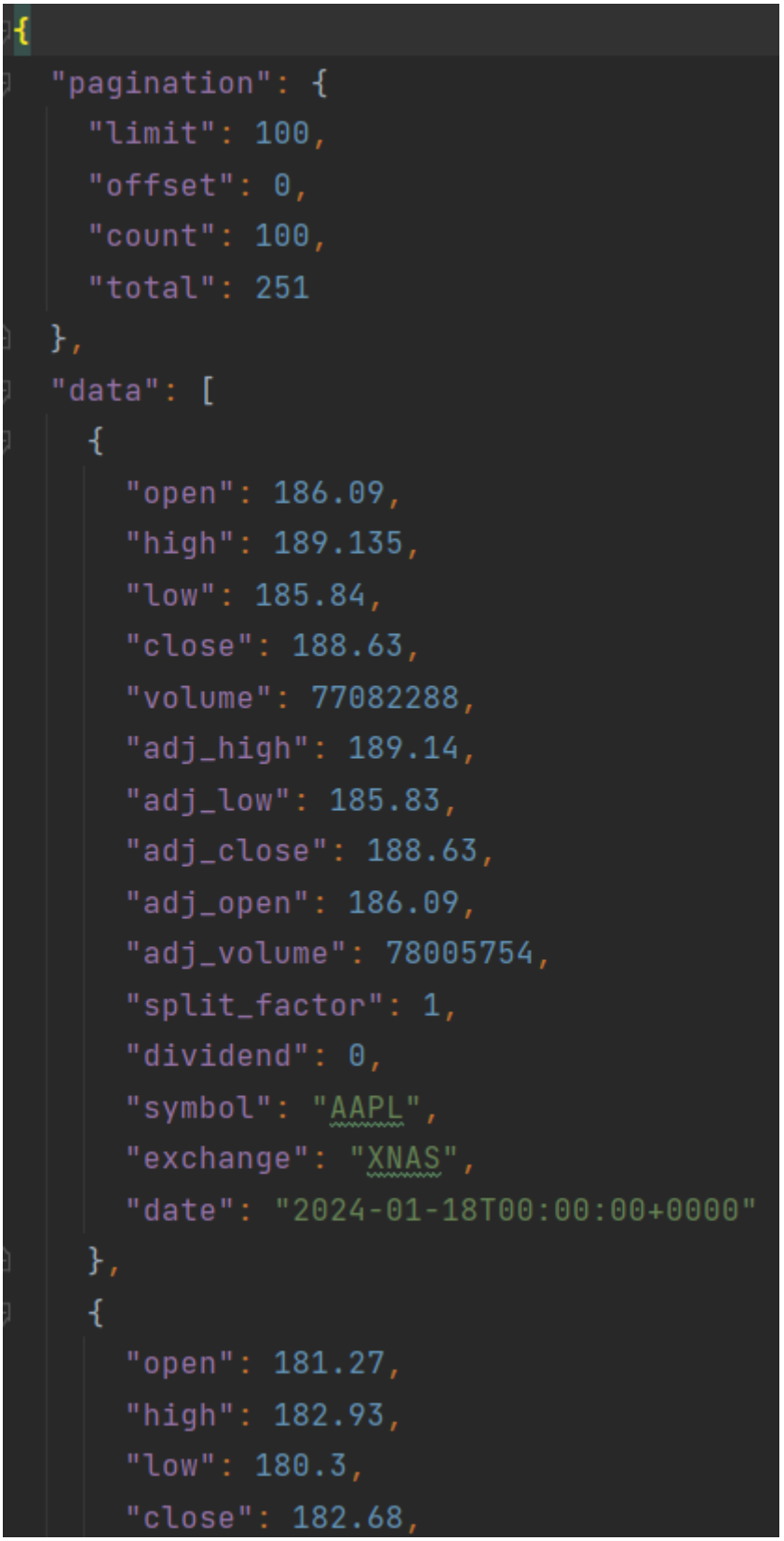
```
fun hasInternetOrUsingBackup(callback: (Boolean) -> Unit) {

    //using backup data? cool, don't need the internet for anything anyway
    if (useBackupData) {
        callback(true)
        return
    }

    //otherwise we need internet
    val client = OkHttpClient()
    val request = Request.Builder().url("http://www.google.com").build()

    //callback with whether we connected to google.com or not
    client.newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) {
            callback(false)
        }
        override fun onResponse(call: Call, response: Response) {
            callback(true)
        }
    })
}
```

Before requesting stock data, the internet connection is tested by attempting to connect to google.com. If this check fails, an alert dialog asks the user to connect to the internet. Once successfully connected, Global.kt's companion object requests the necessary symbol information, assuming the internet connection is stable. This uses callbacks to continue from the original Activity once the data is retrieved.



```
{
  "pagination": {
    "limit": 100,
    "offset": 0,
    "count": 100,
    "total": 251
  },
  "data": [
    {
      "open": 186.09,
      "high": 189.135,
      "low": 185.84,
      "close": 188.63,
      "volume": 77082288,
      "adj_high": 189.14,
      "adj_low": 185.83,
      "adj_close": 188.63,
      "adj_open": 186.09,
      "adj_volume": 78005754,
      "split_factor": 1,
      "dividend": 0,
      "symbol": "AAPL",
      "exchange": "XNAS",
      "date": "2024-01-18T00:00:00+0000"
    },
    {
      "open": 181.27,
      "high": 182.93,
      "low": 180.3,
      "close": 182.68,
```


If the internet connection is stable but *marketstack*'s data is invalid, the app resorts to using backup data from backupapidata.json. This file holds a sample symbol request from *marketstack*, which will be used for all in-game symbols necessary. Ideally, this data will

never be used, as all symbols using it will have equal values, defeating the game's purpose. However, it was invaluable during development to test other app features without requesting unnecessary API data (especially given *marketstack*'s limited free requests per month). The exclusive use of backup data is toggled with Global.useBackupData.

```
fun getInitialIndex(context: Context, gameState: GameState, callback: (Int) -> Unit) {
    requestSymbolData(context, initRequestSymbol) {
        //-2 = -1 to get in array bounds + -1 to ensure there was a previous date to compare to
        callback(getSymbolData(initRequestSymbol)!!.length() - 2)
    }
}
```

Symbol data is only requested if not currently stored in the symbolData dictionary. This way, API requests are kept to a minimum by only sending once per unloaded symbol. At the start of the game, one arbitrary symbol is loaded to determine the oldest data the API provides. This provides a “date” index from which all data can be retrieved, regardless of symbol. Using a date index optimises data searching because desired data can be directly accessed rather than needing to be located per symbol each round. It also bypasses missing date data on weekends and holidays, using the dates with provided data rather than progressing the current in-game date one day at a time.

Critical Reflection

SpeedStocks met most of its goals. QR code cards communicate stocks as physical opportunities and create a more tactile understanding of investments. The game's API processing works well to concisely communicate stock data to players who may not understand the intricacies of the stock market. The optimisation work in these areas also means the app will run quickly and with fewer errors. This could be improved by reducing the amount of JSON data stored in memory at runtime, as much of it is unnecessary.

Many technical fixes are necessary. While designing the game, I failed to consider Android's back button, so pressing back can often crash the app. No check is in place to verify a scanned QR code is a valid stock symbol, though the backup data will be used in these cases. If the internet connection fails after the internet check and during API requests, the app will not continue. An issue may arise if *marketstack* adds new date data mid-game. If accessed by the app, this would break the way data is accessed via an index, as all indices would be moved down the array by one place. While this may not be noticeable in-game, some symbol closes would be offset from the rest, leading to inaccurate data.

Multiplayer functionality could be added to speed up the game loop by allowing each player to invest and view returns on their individual device. More gamemodes would boost replayability, as the current gameplay does not include any player-to-player interactions. The mechanics were designed to be compatible with future gamemode additions, so they would not be difficult to implement. Additional settings, like a configurable number of rounds per game, would boost accessibility.

SpeedStocks successfully invites players to engage in an accessible approximation of stock trading. While its gameplay could be improved through more gamemode additions, the implemented functionality meets its goals.

Word count: 1484

References

APILayer. (2022). *marketstack* [Software]. Available at: https://marketstack.com/

Google. (2024). *Google code scanner* [Software]. Available at: https://developers.google.com/ml-kit/vision/barcode-scanning/code-scanner

Lu, M. (2023). *Visualized: The Top S&P 500 Stocks Over 20 Years* [Online]. Visual Capitalist. Last updated: 26 December 2023. Available at: https://www.visualcapitalist.com/top-sp-500-stocks-over-20-years/ [Accessed 16 January 2024].

Square. (2019). *OkHTTP* (v4.x) [Software]. Available at: https://square.github.io/okhttp/