1. **Temat projektu:**

Gra MemoGame z dodatkiem reklam.

2. **Charakter merytoryczny:**

Gra MemoGame jest grą pamięciową. Użytkownik w określonym z góry czasie odkrywa kart w celu odnalezienie dwóch tych samych. Za każdą odkrytą kartę użytkownik dostaje punkty. Jego głównym zadaniem jest znalezienie jak największej ilości par, jak najmniejszą ilością ruchów. Głównym utrudnieniem dla użytkownika jest fakt, że runda trwa 2 min, a co 30 sekund plansza jest resetowana (wraz z resetowaniem zmienia się rozmieszczenie kart). W przypadku gdy użytkownik odkryje wszystkie karty, plansza również jest resetowana. Aplikacja została wzbogacona o moduł reklamowy, który pozwoli uzyskać użytkownikowi dodatkowe profity. Gracz ma możliwość otrzymania nagrody po obejrzeniu reklamy. Do nagród należy:

- "*znajdź mnie!*" - automat wyszukuje i odkrywa parę do bieżącej odkrytej karty
- "*dodaj czas*" – czas rundy zostaje wydłużony o 1 minutę (limit górny w danym momencie wynosi 5 min).
- *"resetuj"* – cykl losowania planszy zostaje zresetowany. Odliczanie ponownie zaczyna się od 30 sek.

Aplikacja została wyposażona w 3 moduły reklamowe:

- *AdMobBaner* - stale wyświetlający się baner reklamowy w dolnej części aplikacji.
- *AdMobIntestitial* - reklama pełnoekranowa wyświetlająca się w określonych cyklach czasowych (w tym przypadku po zakończeniu *rozgrywki*).
- *AdMobRewarded* - reklama wyświetlająca się na żądanie. Po jej obejrzeniu użytkownik dostaje nagrodę.

3. **Krótkie omówienie rozpoznanych zagadnień (w tym narzędzi wykorzystanych do utworzenia aplikacji)**

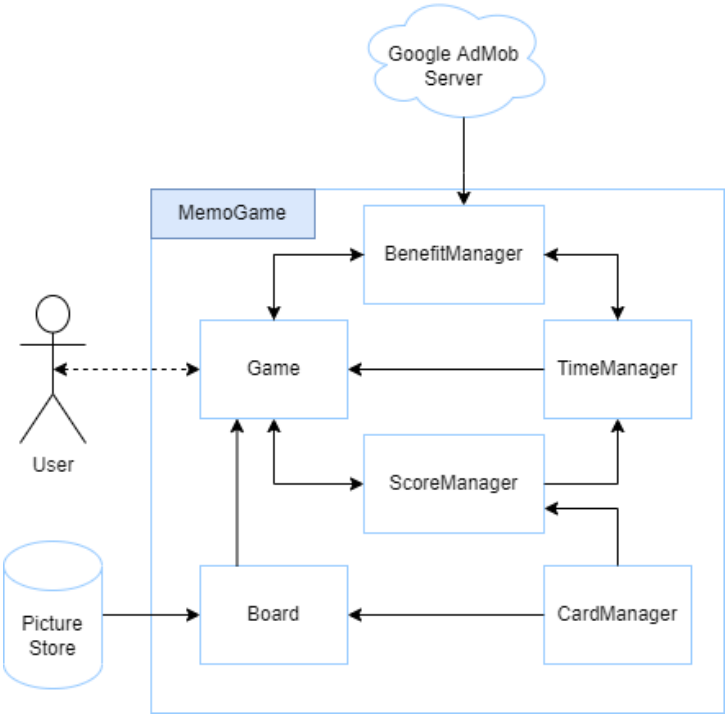
- *React Native* - otwarto źródłowy zestaw narzędzi dla programistów przeznaczony do tworzenia natywnych, wieloplatformowych aplikacji mobilnych, komputerowych, internetowych oraz aplikacji telewizorowych systemów operacyjnych. Sam w sobie łączy najlepsze elementy programowania natywnego z Reactem. Zapewnia najlepszą w swojej klasie bibliotekę JavaScript'ową do budowania interfejsów użytkownika. Hasłem przewodnim owej technologii jest "*Many platforms, one React",* co oznacza, że jedna baza kodu może zostać udostępniona na różnych platformach, dzięki czemu jeden zespół jest w stanie utrzymać dwie platformy i korzystać z jednej technologii.
- *AdMob* - bibliotek oferowana przez expo umożliwiająca obsługę pakietu Google AdMob SDK. Biblioteka udostępnia gotowe komponenty i niezbędny interfejs API do banerów reklamowych, reklam pełnoekranowych i reklam wideo z nagrodami, dzięki czemu zastosowanie reklam w aplikacji jest dość proste i intuicyjne.
- *expo* - framework i platforma dla uniwersalnych aplikacji React. Inaczej mówiąc jest zestawem narzędzi i usług zbudowanych na platformach React Native i natywnych, które wspomagają tworzenie, budowanie, wdrażanie i szybkie iterowanie aplikacji zarówno na Androidzie jaki i IOS. Umożliwia również tworzenie aplikacji internetowych bazujących na tej samej bazie kodu.

**a) wykorzystane źródła wiedzy**

- *React Native* - https://reactnative.dev/
- *AdMob* - https://docs.expo.dev/versions/latest/sdk/admob/#admobinterstitials
- Expo - https://docs.expo.dev/
- *Enabling test ads* - https://developers.google.com/admob/android/test-ads

- *React* - https://pl.reactjs.org/

## 4. Rysunek poglądowy aplikacji wykonywanej przez zespół projektowy




## 5. Implementacja

### a) Menadżer główny - Game

Menadżer główny - jest komponentem zarządzającym *rozrywką*. Przechowuje w sobie wszystkie najważniejsze stany, obiekty oraz informację z nimi związane. Komponent ten odpowiada za akcje takie jak: rozmieszczenie kart na planszy, odkrywanie kart, odpowiednie parowanie, zliczanie znalezionych par oraz kliknięć. Nadzoruje również czas i decyduje czy rozgrywka trwa czy też się zakończyła. W przypadku gdy rozgrywka dobiegła końca blokuje planszę oraz wyświetla reklamę typu *Interstitial* (reklama pełnoekranowa). Poniżej kilka najważniejszych metod.

- *setFlipped* - metoda odpowiedzialna za odwracanie kart. Najważniejszy mechanizm gry Memo.

setFlipped przyjmuje index danej karty. Karty są numerowane od *0* do 2*n-1*, gdzie *n* to liczba wszystkich kart. Można zauważyć, że liczba ta zawsze będzie parzysta, ze względu na logikę gry. Gra polega na znalezieniu pary tych samych kart, więc mamy dwa scenariusze: parzysta lub nieparzysta liczba zdjęć na wejściu. Każda z grafik jest traktowana jako unikatowy byt, który jest wykorzystywany dwukrotnie - w taki sposób powstają pary. Posiadając index, metoda przeszukuję tablicę z kartami i wykonuje na ich odpowiednie akcje, oparte na scenariuszach. Wyróżniamy 3 scenariusze:

- w rundzie bieżącej jest odkrytych mniej kart niż 2 - odkryj kartę,
- w rundzie bieżącej została kliknięta karta, ale są już dwie odkryte:
  - Jeżeli 2 bieżące odkryte karty są różne - zakryj je i odkryj kliknięta,
  - Jeżeli 2 bieżące karty są takie same - zostaw je i odkryj kliknięta,
- została kliknięta odkryta karta, a liczba odkrytych kart wynosiła 1 - zakryj ją.

Sam menadżer bazuje na stanie *CurrentRound* (runda bieżąca), w którym są przechowywane, do dwóch ostatnich odkrytych kart. Zakrycie i sparowanie karty równa się z usunięciem ich z *CurrentRound*. Dzięki owemu stanowi menedżer nie musi śledzić całej planszy, tym samym sparowane karty nie wadzą w procesie dalszego przetwarzania. Metoda *setFlipped* ma możliwość operowania tylko i wyłącznie na kartach, które:

- posiadają flagę *flipped* ustawioną na *false*,
- posiadają flagę *flipped* ustawioną na *true* i ich index znajduje się w stanie *CurrentRound*

Dzięki takiemu podejściu użytkownik na danym *poziomie* może sparować tylko raz daną parę.

```
const setFlipped = (index: number) => {
    if (!gameOver) {
      //Jeżeli: karta jest zakryta i jest mniej odkrytych niż 2
      if (!card[index].flipped && currentRound.length < 2) {
        updateCard(index);
        incrementClicker();
        update(index);
      }
      //Jeżeli: karta jest zakryta, ale są już dwie odkryte
      else if (!card[index].flipped && currentRound.length == 2) {
        //Dwie odkryte karty są różne
        if (currentRound[0].pictureId !== currentRound[1].pictureId) {
          currentRound.forEach((item) => {
            updateCard(item.index);
          });
          setCurrentRound([]);
          updateCard(index);
          incrementClicker();
          update(index);
        }
        //Jeżeli została kliknięta któraś z klikniętych kart
      } else if (card[index].flipped && currentRound.length == 2) {
        const indexToSave = currentRound.findIndex((i) => i.index === index);
        let indexToRemove;
        if (indexToSave === 0) {
          indexToRemove = 1;
        } else {
          index Remove = 0;
        }
        updateCard(currentRound[indexToRemove].index);
        setCurrentRound([{ ...currentRound[indexToSave] }]);
        incrementClicker();
      }
    }
  };
```

- *updateCard* - metoda odpowiedzialna za aktualizację obiektu *Card*

Metoda *updateCard* posiadając index karty, dzieli kolekcję na 3 części: [0,n-1],[n].[n+1, m-1], gdzie *n* to index poszukiwanej karty, a *m* to liczba wszystkich kart. Takie podejście pozwala, bezpośrednio wyciągnąć poszukiwany obiekt, edytować go bez ingerencji w inne obiekty, a następnie z 3 części stworzyć jedną kolekcję [0, m-1]. Podejście to również zapewnia złożoność na poziomie 1.

```
  const updateCard = (index: number) => {
    setCard((card) => [
      ...card.slice(0, index),
      { ...card[index], flipped: !card[index].flipped },
      ...card.slice(index + 1),
    ]);
  };
```

- *useEffect()* - efekt odpowiedzialny za wyświetlenie reklamy pełnoekranowej

Efekty są pewnymi "wyzwalaczami" w *React*. Działanie efektu: jeżeli zmieni się stan *gameOver*, a jego wartość będzie *true* wyświetl reklamę pełnoekranową.

```
useEffect(() => {
  if (gameOver) {
    const showInterstitialAds = async () => {
      try {
        AdMobInterstitial.setAdUnitID(androidInterId);
        await AdMobInterstitial.requestAdAsync({
          servePersonalizedAds: false,
        });
        await AdMobInterstitial.showAdAsync();
      } catch {
        (e: any) => console.log(e);
      }
    };

    showInterstitialAds();
  }
}, [gameOver]);
```

### b) Pomodoro - metoda zarządzania czasem

Pomodoro jest metodą zarządzania czasem. Czas odgrywa w aplikacji dwojaką rolę i wprowadza pewne utrudnienie. Pierwszym utrudnieniem jest maksymalny czas rozrywki. Użytkownik w danym czasie musi zdobyć jak największą ilość punktów, przy możliwie jak najmniejszej liczbie kliknięć. Po upływie czasu, *rozgrywka* dla użytkownika kończy się. Drugim utrudnieniem jest czas bieżącego *poziomu*. Czas bieżącego poziomu jest pewną częścią czasu rozgrywki (np. ¼, przy *rozgrywce* 2 min., mamy około 4 *rundy* po 30 sek.). Po upływie czasu *poziomu* plansza zostaje zresetowana, karty zakryte, a ich rozmieszczenie przetasowane. Dzięki takiemu rozwiązaniu został zwiększony poziom trudności gry. W przypadku gdy użytkownik odkryje wszystkie karty przed upływem czasu *poziomu*, czas *poziomu* zostaje zresetowany tym samym zaczyna się kolejny *poziom*. W rozgrywce może wystąpić więcej *poziomów* niż zostało założone. Może być ich również mniej.

- *resetBoard* - metoda odpowiedzialna za resetowanie planszy po określonym czasie.

```
const resetBoard = () => {
  setCard(CardStoreConstructor());
  setCurrentRound([]);
  setCurentPair(0);
};
```

- *useEffect* - efekt odpowiedzialny na odliczanie czasu. Uruchamiany cyklicznie, co 1 sek. W przypadku gdy czas osiągnie 0, wywołuję funkcję *quitGame* udostępnioną przez *Menadżera głównego.* Funkcja ta zmienia stan *gameOver* na *true*, co uniemożliwia dalsze odliczanie czasu.

```
useEffect(() => {
  if (!gameOver) {
    let interval = setInterval(() => {
      //Czas do końca
      if (secondToGo === 0) {
        if (minutesToGo !== 0) {
          setSecondToGo(59);
          setMinutesToGo((m) => m - 1);
        } else {
          quitGame();
        }
      } else {
        setSecondToGo((s) => s - 1);
      }

      //Czas do losowania
      if (secondToDraw > 0) {
        setSecondToDraw((s) => s - 1);
      } else {
        if (timeToEnd()) {
          setSecondToDraw(SecondToDrawConst - 1);
          resetBoard();
        }
      }
    }, 1000);
    return () => clearInterval(interval);
  }
}, [secondToGo, secondToDraw]);
```

### c) *Benefit Manager*

Benefit Manager jest komponentem skierowanym dla użytkowników. W swojej idei jest sprzymierzeńcem gracza. Pozwala niewielkim kosztem, którym jest obejrzenie reklamy, otrzymać pewne profity. Bonusy które oferuje to:

- *znajdź mnie* - pomaga znaleźć parę do danej karty

```
const findMe = () => {
  if (currentRound.length === 1) {
    const findIndex = card.findIndex(
      (card, i) =>
        card.pictureId === currentRound[0].pictureId &&
        currentRound[0].index !== i
    );
    updateCard(findIndex);
    incrementFoundPairs();
    incrementClicker();
    setCurrentRound([]);
  }
};
```

- *zatrzymaj losowanie*- resetuje czas bieżącego *poziomu*, tym samym zmniejszając jego poziom trudności

```
const resetTimeToDraw = () => {
  if (timeToEnd()) {
    setSecondToDraw(SecondToDrawConst);
  }
};
```

- *dodaj czas* - pozwala dodać czas do bieżącej *rozgrywki*

```
const addTime = () => {
  if (!gameOver) {
    const time = minutesToGo * 60 + secondToGo + SecondToDrawConst;
    const maxTime = MaximumGameTime * 60;
    if (time < maxTime) {
      const sec = 60 - secondToGo;
      if (sec > SecondToDrawConst) {
        setSecondToGo((s) => s + SecondToDrawConst);
      } else {
        setMinutesToGo((m) => m + 1);
        const time = 60 - secondToGo;
        const sec = SecondToDrawConst - time;
        setSecondToGo(sec);
      }
    }
  }
};
```

Zastosowanie owego menadżera pozwoliło, w sposób sensowny wykorzystać moduł reklamowy. Sam menadżer może działać tylko w przypadku gdy *rozgrywka* trwa. Reklamy czasowe blokują zarówno czas *poziomu*, jak i *rozrywki*.

### d) Widok główny

Widok główny składa się z kilku podstawowych części:

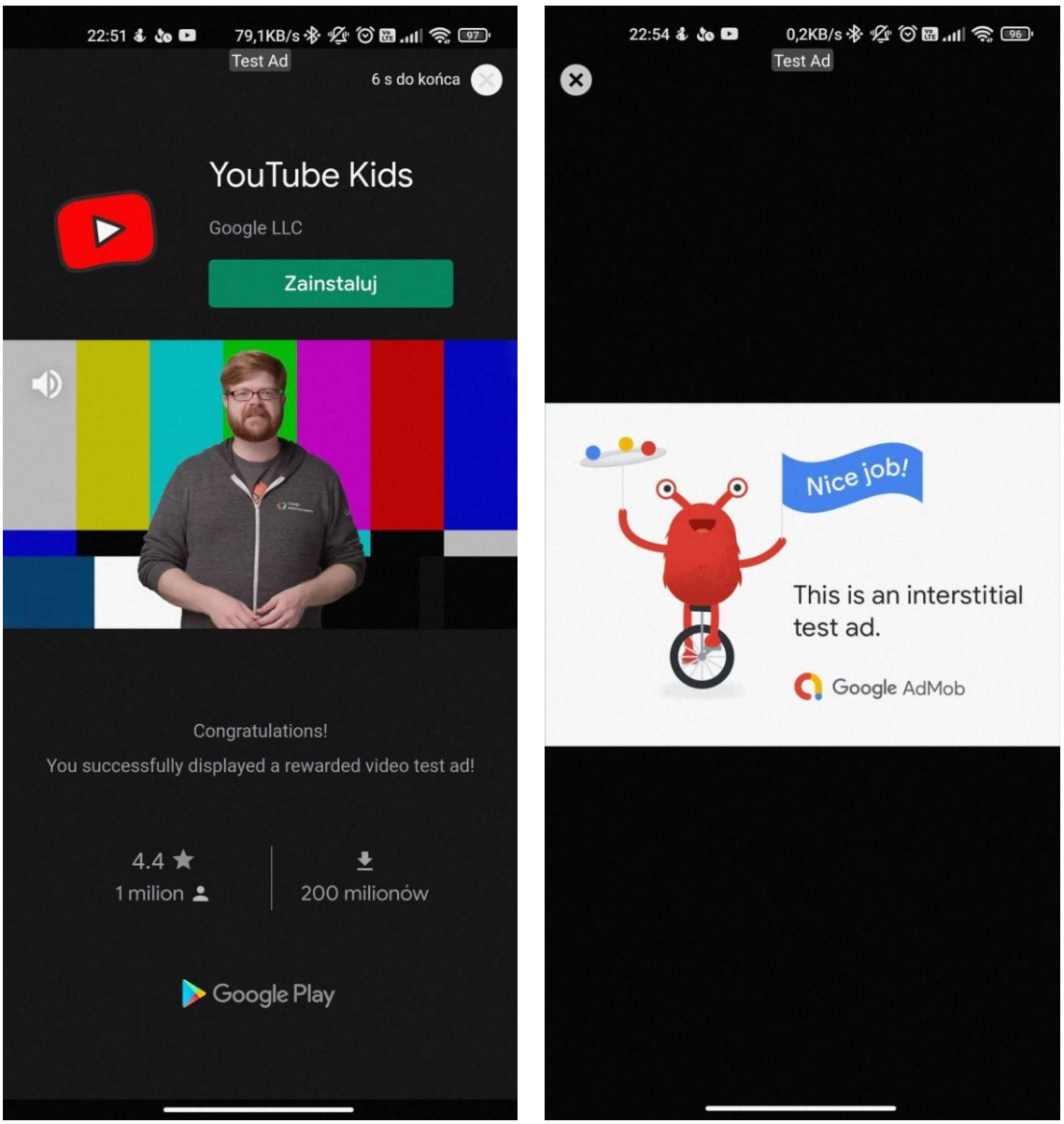
- nazwy aplikacji,
- planszy zawierającej określoną liczbę kart,
- tabeli wyników, na której znajdują się pozycje takie jak: liczba znalezionych par, liczba kliknięć, czas do końca *poziomu* (losowanie), czas do końca *rozgrywki*,
- 3 przycisków *Benefit Managera*: znajdź mnie!, resetuj losowanie, dodaj czas,
- baneru reklamowego.

Poniżej zaprezentowano implementację widoku:

```
export const MainView: FC<Props> = ({ text }) => {
  return (
    <View style={container}>
      <View style={phoneBar} />
      <View style={[titleBar, constCenter]}>
        <Text style={titleText}>{text}</Text>
      </View>
      <View style={board}>
        <Cards />
      </View>
      <View style={[constRow, scoreBar]}>
        <Score />
      </View>
      <View style={[constRow, bottomBar]}>
        {buttons.map((btn) => {
          return <ButtonBasic {...btn} key={btn.id} />;
        })}
      </View>
      <AdMobBanner
        bannerSize="banner"
        adUnitID="ca-app-pub-3940256099942544/6300978111"
        servePersonalizedAds={false}
        style={[adsBar, constCenter]}
      />
    </View>
  );
};
```

## 6. Prezentacja aplikacji


Powyżej została zaprezentowana aplikacja *MobileMemo*. Po stronie lewej mamy początkowy stan gry, po prawej w trakcie rozgrywki. Aplikacja została zaprojektowana w sposób minimalistyczny, funkcjonalny, zachowując przy tym nowoczesny design. Główną uwagę użytkownika przykuwa plansza, dzięki czemu większość uwagi użytkownik skieruje właśnie na rozrywkę. Odpowiednie proporcję pomogą mu cały czas kątem oka monitorować czas, a duże przyciski, w dolnej części ekranu, pozwolą mu szybko skorzystać z benefitów. Baner umieszczony na samym spodzie jest delikatnym, mało irytującym dodatkiem.




Wyżej zostały zaprezentowane dwa typy reklam: po lewej *reward* (reklama, za którą otrzymujemy nagrodę), po prawej *interstitial* (reklama pełnoekranowa uruchamiana po zakończeniu *rozgrywki*).

## 7. Podsumowanie

Implementacja gry *MemoGame* była swego rodzaju wyzwaniem. Przede wszystkim spotkanie z nowym środowiskiem, technologią i zgłębienie ich tajników. Następnie przeniesienie zasad i logi gry do programu, gdzie z pozoru proste rzeczy okazały się dość wymagające. Zabezpieczenie logiki przed przeróżnymi błędami użytkownika, implementacja modułu reklamowego i znowu zgłębianie nowych rzeczy. A na końcu testowanie samej aplikacji.

Jedną z trudniejszych rzeczy do zaimplementowania była plansza, umieszczone na niej karty, mechanizm odwracania, a w tym wszystkim utrzymanie stanów. Na wideo poniżej można zobaczyć jeden z najtrudniejszych napotkanych problemów:

https://photos.app.goo.gl/kNtFaABAiaB1MGqA9

Możemy zauważyć niekontrolowane i wielokrotne obracanie kart.

Problematyczny również okazał się system mierzenia czasu.

## 8. Spis literatury

Dokumentacja Expo: https://docs.expo.dev/
Dokumentacja React-Native: https://reactnative.dev/docs/getting-started
Testowe ID systemu AdMob: https://developers.google.com/admob/android/test-ads
Tutorial 1: https://akhromieiev.com/tutorials/adding-admob-expo-project-step-step/
Tutorial 2: https://medium.com/swlh/using-admob-rewarded-videos-in-your-expo-project-2d4231cf63a6