Relazione progetto LAM 2018-2019

Matteo Conti

******INDICE**

INFORMAZIONI GENERALI

- Studente: *Matteo Conti*
- Matricola: *0000802576*
- Corso: *laboratorio di applicazioni mobile*
- Anno Accademico: *2018/2019*
- Data: *1 Febbraio 2020*

Introduzione

Il progetto consiste nell'applicazione pratica delle conoscenze acquisite durante lo svolgimento del corso "Laboratorio di Applicazioni Mobile" presso l'Università di Bologna.

Lo scopo dell'applicazione è la riproduzione del funzionamento del gioco di carte italiano, chiamato Briscola, su dispositivi aventi sistema operativo iOS (piattaforma Apple). La versione attuale del gioco prevede la presenza di due giocatori e di due modalità di gioco. Nella prima modalità (single-player) un giocatore viene controllato da un emulatore basato su un semplice algoritmo a stati che determina la carta da giocare in base allo stato attuale del gioco. Sono state pienamente implementate tutte le regole e caratteristiche della Briscola, e quindi il comportamento dell'emulatore seguirà una strategia di gioco che rispecchia la logica dello stesso gioco. Inizieremo discutendo le caratteristiche e l'architettura generale del progetto, illustrando le funzionalità e l'implementazione alla base di ciascuna di esse. Proseguiremo illustrando le scelte attuate durante la fase di progettazione, i problemi e le difficoltà incontrate, evidenzieremo alcuni dettagli che hanno inciso sulle tempistiche di consegna e mostreremo i flussi di dati presenti all'interno dell'applicazione. Infine termineremo proponendo alcune possibili implementazioni, aggiungendo alcuni commenti relativi alla curva di apprendimento e le tecnologie a disposizione.

1. Scopo dell'Applicazione

Lo scopo era quello di creare due ambienti di gioco: il primo (con la presenza di un giocatore simulato) serviva per simulare una sorta di fase iniziale di "training" (anche se non c'è un vero trainer che corregge le mosse fatte dall'utilizzatore dell'applicazione), mentre il secondo serviva per testare i propri progressi confrontandosi con altri giocatori.

Architettura generale

Il pattern utilizzato nella realizzazione di questa applicazione è il Model-View-Controller (MVC). L'importanza di questo pattern consiste nella separazione della logica di presentazione dei dati rispetto alla logica di gestione di questi. Per questo progetto infatti, sono stati creati diversi modelli di dati corrispondenti alle varie entità in gioco (Carte, Giocatori, ..), tutti gestiti da diverse librerie di "gestione" (Handlers), le quali astraggono completamente il flusso di creazione/modifica dei dati rispetto alla loro presentazione (gestita all'interno dei Controllers).

2. MVC

All'interno dell'applicazione esiste un `NavigationController` attraverso il quale è possibile muoversi attraverso la varie view che compongono l'applicazione. Esistono infatti 4 schermate:

1. **Menù**: da qui è possibile impostare le varie opzioni di gioco e passare sia alla schermata di gioco, sia a quella "Social".
2. **Gioco**: in questa schermata è possibile avviare il gioco e passare alla schermata dei "Risultati"
3. **Risultati**: qui è possibile vedere il risultato della partita una volta terminata e decidere di salvarla.
4. **Social**: da qui è possible vedere la lista di tutte le partite salvate e, per ognuna, decidere se condividerla su Facebook.

Di seguito verranno riportati i principali modelli di dati utilizzati e le librerie a supporto di questi.

2.1. Carta (Modello Dati) (TODO)

.. .. ..

Gli attributi principali sono il tipo e il numero.

.. .. ..

2.2. Giocatore (Modello Dati) (TODO)

.. .. ..

contiene una variabile che identifica le carte che ha in mano e altre informazioni come il tipo (locale, emulatore), il nome e le carte conquistate presenti nel suo mazzo.

.. .. ..

2.3. Gestore Gioco (TODO)

.. .. ..

2.4. Gestore Database (CoreData) (TODO)

.. .. ..

2.5. Gestore Sessione (MultiPeer) (TODO)

.. .. ..

2.6. Gestore Social (Facebook) (TODO)

.. .. ..

3. Emulatore

Questa classe si occupa di simulare la presenza (in modalità single-player) di un giocatore reale con cui giocare. È stata realizzata, infatti, come una macchina a stati che in base alla configurazione delle carte in tavola e quelle in mano (dell'emulatore) elabora uno stato finale, che si traduce nella scelta di una carta da giocare (chiaramente tra quelle che ha in mano).

Sono stati individuati due macro stati: il primo è quello nel quale l'emulatore si trova a dover giocare per primo, mentre il secondo è quando si trova a giocare per secondo (e abbiamo quindi già una carta in tavola).

Di seguito riportiamo la sequenza di scelte dalla macchina a stati, dove quest'ultima si bloccherà al primo stato di match trovato nella sequenza di scelte disponibili.

3.1. Nessuna carta in tavola

L'emulatore è il primo a giocare.

1. gioco il liscio più basso che ho
2. gioca il carico più basso che ho.
3. gioca la briscola più bassa che ho.
4. gioca il carico più basso che ho.
5. gioco la briscola più bassa che ho

3.2. C'è già una carta in tavola

L'emulatore è il secondo a giocatore. La sequenza di scelte varia in base al tipo della carta giocata:

3.2.1. Briscola

1. gioca il liscio più alto che ho.
2. gioco il carico più basso che ho sotto i 10 punti.
3. gioco la briscola più bassa che ho (solo se non vale dei punti).
4. gioco la briscola più bassa che ho (solo se la carta in tavola vale dei punti e la mia supera quella in tavola).
5. gioco il carico più basso che ho.
6. gioco la briscola più bassa che ho.

3.2.2. Carico

1. gioco il carico più alto che ho di questo tipo ma solo se supera la carta in tavola.
2. gioco la briscola più bassa che ho.
3. gioco il liscio più alto che ho
4. gioco il carico più basso che ho.

3.2.3. Liscio

1. gioco il carico più alto che ho di questo tipo (solo se è un re, un tre on un asso).
2. gioco il liscio più alto che ho non di questo tipo.
3. gioco il liscio più basso che ho di questo tipo, a patto di non superare la carta in tavola.
4. gioco il carico più alto che ho di questo tipo.
5. gioco il liscio più basso che ho.
6. gioca il carico più basso che ho (fante o cavallo o re)
7. gioca la briscola più bassa che ho senza punti.
8. gioco il carico più basso che ho.
9. gioco la briscola più bassa che ho.

4. Funzionalità

Avendo illustrato nel capitolo precedente le 4 view principali all'interno dell'applicazione, è facile dedurre che queste corrispondo esattamente alle 4 funzionalità principali:

1. Configurazione del Gioco (tramite il menù e le varie opzioni)
2. Gioco (in modalità single/multi-player)
3. Salvataggio dei risultati
4. Condivisione dei risultati

4.1. Interfaccia Grafica (TODO)

.. .. ..

La Figura sottostante mostra ciò che l'utente vede nel momento in cui viene avviata la partita. Notiamo come le nostre carte sono scoperte (al contrario di quelle dell'avversario) e per poter giocare una carta è necessario toccarla. Una volta che anche il nostro avversario (emulatore/giocatore remoto) avrà giocato una carta, il vincente della mano si aggiudicherà le carte, ed esse verranno aggiunte al suo mazzo (non visibile), e automaticamente verrà distribuita una nuova carta (ad ogni giocatore) e la mano sarà pronta per iniziare (partirà il giocatore che ha vinto quella precedente)

FLUSSO IMMAGINI: gioco da iniziare, gioco iniziato, una carta giocata, due carte giocate.

.. .. ..

4.2. Flusso Dati (Database)

Il flusso dei dati persistiti è piuttosto semplice:

1. Appena avviata l'applicazione (nell'apposita schermata) è possibile visualizzare l'elenco delle partite salvate sul database interno.
2. Al termine di ogni partita è possibile salvare il risultato in modo facile e veloce.
   Nessun dato viene salvato automaticamente e non è possibile salvare nessuna informazione sulla partita in corso finché questa non è terminata.

4.3. Tecnologie Utilizzate

1. *MultiPeer Connectivity* [1]: questa libreria supporta la scoperta di servizi forniti da dispositivi vicini e supporta la comunicazione con tali servizi attraverso dati basati su messaggi. In iOS, il framework utilizza reti Wi-Fi infrastrutturali, Wi-Fi peer-to-peer e reti personali Bluetooth per il trasporto sottostante.
2. *CoreData* [2]: questa libreria consente di salvare i dati permanenti di un'applicazione per l'utilizzo offline e di memorizzare nella cache i dati temporanei (database in locale).
3. *Facebook SDK* [3]: questa libreria consente di utilizzare tutte le principali SDK di Facebook per applicazioni iOS. In particolare in questo progetto sono state utilizzare le API per la condivisione di link e testo.

5. Requisiti Tecnici

- Piattaforma: *Apple*
- Sistema *Operativo: iOS (versione 13.2)*
- Editor: *Xcode (versione 11.2.1)*
- Linguaggio di Programmazione: *Swift (versione 5.0)*

Progettazione

SCHERMATA GIOCO

1. Il *controller* viene creato (con all'interno l'istanza della configurazione di gioco scelta, passata dalla *view* precedente), crea un'istanza di tutte le librerie che utilizza (gestore del gioco).
2. L'*Handler* crea tutti i modelli, inizializza il gioco (in base alla modalità) e salva il loro stato al suo interno.
3. Il *controller* prepara tutti gli *assets* (stato iniziale di label, bottoni, ..), inizializza la sessione condivisa (se la modalità è multi-player) e chiama la funzione `render()` che si occupa di aggiornare la grafica in base allo stato del gioco.
4. Il *controller* riceve un *outlet* dalla view (quando ad esempio un giocatore clicca su una carta) e viene chiamato di conseguenza il metodo corrispondete all'interno dell'*handler*. Una volta terminato questo, il controller richiamerà `render()` per aggiornare la grafica.

In sostanza, il controller non interagisce mai direttamente con i modelli, ma sfrutta sempre la definizione dei metodi degli *handlers* per applicare modifiche allo stato del sistema (in questo modo tutta la logica del gioco è astratta rispetto al controller). Per esempio, per giocare a scala 40 (una altro gioco di carte simile come impostazione alla briscola) basterebbe cambiare il gestore del gioco, la grafica, ma non il controller. (Se non per i riferimenti alla nuova grafica)

FLUSSO DATI ESTERNO (sessione):

1. Un giocatore inizia il gioco avviando una sessione e rimane in attesa del numero di partecipanti richiesto (in questo caso due).
2. Un secondo giocatore inizia il gioco avviando anche lui una sessione e, trovando già un dispositivo connesso, richiederà al connessione a questo.
3. Una volta accettata la connessione, il gioco inizierà nel momento un cui uno dei due giocatori cliccherà sul pulsante d'inizio.

4. Una volta iniziato il gioco, il giocatore che ha cliccato sul pulsante d'inizio condividerà con l'altro lo stato iniziale del mazzo, oltre ad alcune informazioni personali (tra cui anche il suo indice all'interno dell'array di giocatori) e in base a queste, entrambi i giocatori creeranno due istanze di gioco identiche.
5. Ogni giocatore a turno giocherà una carta e rimarrà in attesa che l'altro faccia lo stesso.
6. Una volta terminato il gioco la sessione viene disconnessa.

L'unica differenza tra le due modalità è la gestione dell'inizializzazione del gioco e la funzione che automaticamente chiede all'emulatore di giocare una carta solo se la modalità è single-player (in caso contrario rimarrà in attesa).

6. Difficoltà e Soluzioni

La difficoltà principale (oltre alla scelta della struttura dati portante, discussa nel prossimo capitolo) nello sviluppo era la gestione del flusso di dati tra le entità principali: *Controllers*, *Models* e *Handlers*.

Ad esempio, dove salvare l'istanza dei modelli di dati utilizzati? Negli *handlers* o nei *controllers*? In questo progetto ho deciso di salvare tutte le informazioni all'interno degli *handlers*, rendendoli indipendenti dal *controller* in cui venivano usati. In questo modo, infatti, è stato possibile utilizzare il `DatabaseHandler` in due *controllers* diversi. Chiaramente gli *handlers* non nascondono affatto i dati al loro interno e gli stessi controller conoscono la struttura e la rappresentazione di queste informazioni, ma rimangono sempre indipendenti rispetto al loro ciclo di modifica. Ad esempio quando un giocatore gioca una carta, il *controller* chiama un metodo del `GameHandler`, e successivamente riporta nella *view* lo stato del gioco aggiornato (senza conoscere il flusso di modifica dello stato del gioco).

In sostanza ogni *controller*, oltre alla gestione della visualizzazione dei dati, si occupa di gestire gli eventi (*gestures*, ..) e di implementare i vari protocolli richiesti dalla librerie (comunemente chiamati *Delegates*).

7. Aspetti Rilevanti per lo Sviluppo

Inizialmente, nello sviluppo, mi sono concentrato nel cercare di costruire una base solida e facilmente estendibile. In quest'ottica ho individuato nella struttura dati `Array` la chiave della struttura di tutto il codice. Tutte le principali strutture dati (i giocatori, le carte in mano ad un giocatore, le carte in tavola, ..) sono rappresentate con questa struttura dati. La caratteristica di questa struttura è il fatto che può contenere un numero variabile di elementi (dello stesso tipo) e questo chiaramente si sposava a pieno con l'idea, un giorno, di proseguire lo sviluppo aggiungendo anche le modalità a 3, 4 e 5 giocatori. Infatti nella modalità a 2/3/4 giocatori, ognuno ha in mano (a turno iniziato) 3 carte, mentre nella modalità a 5 ognuno possiede esattamente 8 carte. In quest'ottica avrei potuto usare, ad esempio, la struttura dati `Tuple` fissando il numero di elementi di ciascuna tupla (ovvero il numero di carte in mano), ma, usando la struttura descritta prima, si potrebbe utilizzare lo stesso gestore gioco a prescindere del numero di carte in mano a ciascun giocatore. Questo è il punto fondamentale con cui ho sviluppato l'applicazione: il gestore del gioco opera a prescindere del numero di giocatori e del numero di carte in mano ad ognuno di essi.

In questo modo ogni possibile estensione (del tipo descritto) può essere realizzata con poco sforzo (rimane comunque da analizzare la gestione grafica, che nonostante sia già basata su quest'ottica, richiede una discussione giù approfondita).

Inoltre un altro aspetto fondamentale è stato l'utilizzo di `struct` per l'interscambio dei dati all'interno di una sessione condivisa (modalità multi-player). Queste infatti, sono una semplificazione dei modelli di dati usati all'interno dell'app, dai quali sono state rimosse le informazioni non necessarie nella condivisione. Inoltre ad ogni struttura, sono stati aggiunti dei metodi per poterla facilmente codificare/decodificare all'interno della sessione.

8. Possibili Estensioni e Miglioramenti

Lista di alcuni possibili miglioramenti:

1. Rendere ancora più intelligente l’emulatore.
2. Aggiungere la possibilità di scegliere il dispositivo con cui giocare (all’interno della sessione).
3. Aggiungere diversi livelli di difficoltà e rendere questa opzione configurabile (ad esempio nel menù iniziale).
4. Migliorare la grafica e aggiungere le animazioni.
5. Aggiungere i constraint per tablet e landscape mode.

Lista di alcune possibili estensioni:

1. Disegnare le carte in AR (tramite ARKit).
2. Aggiungere anche le modalità con 3, 4 e 5 giocatori.
3. Sincronizzare il database in locale con uno remoto in modo da non perdere i progressi (es: Realm, ..)

Conclusione e Commenti Finali (TODO)

.. .. ..

Questo progetto ha dato l'opportunità di approfondire meglio le tecniche di sviluppo nel campo di iOS e le strategie di programmazione di questo campo, in oltre la scelta di progettare un gioco ha permesso una trattazione di alcuni aspetti molto importanti dal punto di vista grafico e di come integrare scenari animati all'interno di una applicazione. Un'altro aspetto molto importante e intrigante è stato quello di creare un avversario che possa comportarsi autonomamente, una specie di intelligenza artificiale. La realizzazione di un ente intelligente di questo genere è un procedimento che cerca di simulare un comportamento razionale in grado di percepire ed arricchire le proprie conoscenze e di agire di conseguenza con lo scopo di raggiungere il suo obbiettivo, ovvero vincere la partita di briscola. Affinché esso possa raggiungere il suo scopo è stata definita una strategia di gioco che esso seguirà per raggiungere il suo scopo .

.. .. ..

---

1. The MultiPeer Connectivity framework supports the discovery of services provided by nearby devices and supports communicating with those services through message-based data, streaming data, and resources (such as files). In iOS, the framework uses infrastructure Wi-Fi networks, peer-to-peer Wi-Fi, and Bluetooth personal area networks for the underlying transport. In macOS and tvOS, it uses infrastructure Wi-Fi, peer-to-peer Wi-Fi, and Ethernet.
2. Use Core Data to save your application's permanent data for offline use, to cache temporary data, and to add undo functionality to your app on a single device.Through Core Data's Data Model editor, you define your data's types and relationships, and generate respective class definitions. Core Data can then manage object instances at runtime to provide the following features.
3. The Facebook SDK for Swift allows you to use all of the core Facebook SDK for iOS APIs which includes: Facebook Login, Share and Send App Events and Graph API.