BSc in Computer Science

School of Computing, Science, and Engineering

Blood Grouping

An interactive game for IOS

Author: Raman Suliman

Module: Mobile Development

2022-2023

Contents

Introduction 3

Game Story 3

Navigation Controller 3

UICollectionViewController 4

UITableView 4

Shared Model Data 4

Design Pattern (MVC) 5

Event Handlers 5

TouchesMoved 5

TouchesEnded 5

DidBegin 5

Accelerometer Sensor 6

Particles 6

Sounds 6

Game User Interface 7

Unified Modelling Language 8

Introduction

Throughout this document, a discussion over technical and design aspects of developing the Blood Grouping IOS-based game using UIKit and SpriteKit APIs will be encountered. The debate commences with an analysis of why choosing a particular layout to present a group of related views is an optimal solution in terms of performance and efficiency. Alongside the discussion, a comparison between IOS and Android APIs is given. Afterwards, heightens various kinds of event listeners and the objective of utilizing them to achieve desired interaction in terms of touching, dragging, and sensor feeds. Moreover, aspects such as sounds, animation and particle effects will be covered by defining the purposes of utilizing optimal APIs providing further hearable and visual feedback on interaction. Eventually, demonstrating the structural behaviour of developed objects using a UML class diagram and UI design of different areas in the game.

Game Story

The game is simple and yet significantly important to play and learn from. Humans' blood is divided into 8 categories, each human must inherit one type. Determine what blood type to use for the purposes of donation and blood supplements used during surgeries must share biological compatibility. A patient can only be given the blood if it is the volunteer blood type sharing is convenient. Therefore, this game is designed to target the IOS audience to help in spreading the understanding of the blood groups concept in an enjoyable 2D interactable environment.

Navigation Controller

A typical application normally consists of at least two activity windows which can easily be linked without utilizing a special object to handle the transitions. However, this is not an ideal approach since a system must follow best development practices and prepare for future upgrade measures to take place. Thus, an instance of navigation controller is recommended to handle and manage the available hierarchical activities content.

The mechanism of supervising view controllers in IOS applications is achieved via a navigation stack with the responsibility of allowing transitions between screens. At the bottom of the stack list comes the root view controller presenting the initial screen on application launches. Screens are linked with segues to navigate among with a similar approach implemented in Android with NavGraph having fragments linked via action relationships.

The Blood Grouping game utilized NavigationController to design a hierarchy of two views on its stack list. The main menu is set as the root view controller making it the opening window presented with a segue connection assigned to the play button to dismiss the current view and launch the game scene area on user interaction.

UICollectionViewController

A large portion of the game relays on displaying the current state of player interaction with the blood type objects shown on the scene. Since blood types have distinct groups of compatible elements then a list of collections is required for each kind. Therefore, the UICollectionViewController is used to create cells presenting the wanted lists. An instance of this class provides all necessary functionalities with the ability to have a custom cell of type nib file much like the RecycleView in Kotlin with XML file. The class extends UICollectionViewDelegateFlowLayout class because the game screen layout is set to the right landscape mode with UIInterfaceOrientation class and cells must be presented beside each other in the same horizontal line.

UITableView

On firing event towards the target blood type, the onboard current active type must be inserted into the target list on collision. To implement a list in each ViewCollection cell, an instance of UITableView is created to handle the insertion of a new element and appropriately display it on the user screen. To achieve this, the cell is extending UITableViewDataSournce to manage the data inserted into the table and UITableViewDelegate handling behaviours towards its content.

The number of table rows is statically defined by 4 in the numberOfRowsInSection() method since there are 8 blood types and each row would hold 2 UIImageView controls. The height is equally assigned to each row by dividing the height of the table by 4. To overcome the extra empty rows generated by default, the tableFooterView property has been set to an empty UIView object.

Shared Model Data

Sharing information between screens by keeping synchronization in mind is an essential and common task in most applications. The traditional approach is overriding the prepareForSeque() method to pass data to the intended view prior to transition. However, to keep the data isolated and globally shared the AppDelegate class is used. An object of this class is created by default on game execution with the purpose of providing shared behaviours and resources in conjunction with UIApplication to manage interactions. It somehow acts like Kotlin's ViewModel class in terms of propagating data across the entire application entities.

The game requires several BloodType instances that are retrieved and modified via multiple objects throughout the application runtime. Therefore, a model class is introduced to instantiate and store the required instances in a list. To make this list reachable, an instance of the model class is created in the AppDelegate for any game component to access.

Design Pattern (MVC)

By default, the structure of UIKit apps is based on the Model-View-Controller separating the data and application view to only be connected with a controller acting as an adapter. To extend this practice, I have applied the MVC pattern to create an embedded version within the existing pattern.

The GameViewController class is playing a stater point to create instances of required views and a controller object handling the logic to manage user interaction and application events. On the creation of GameViewSetUp object, the game view controller screen is split into three view sections: a bar area holding UI elements such as options menu, UICollectionViewController, and a view of type SKView. The MainGameController class is responsible for processing events and updating the three-section view elements.

Event Handlers

TouchesMoved

The game is supported by two different input methods the player can utilize to control the rotation of the jet main character. The default option is to touch and drag on the x-axis having motion towards the right side rotating to the right and dragging towards the left performing left movement. The touchesMoved() method is overridden to implement this feature, it captures the initial location of the first touch and waits until the user's finger is released from the screen to obtain the destination touch location. Therefore, subtracting the values given by .previousLocation and .loaction properties defines the direction of the drag. If the computed result provides a negative number, then dragged from right to left and if positive it's vice versa action.

TouchesEnded

Unlike the UI controllers provided by UIKit, the SKScene does not support direct action listener functionalities. The fire button responsible for shooting rockets and designed as SKSpriteNode to form the detection area with an image and supported with a SKLableNode as the first child with zPosition 2 bringing the text to the front. To detect user interaction with this button, the touchesEnded() method is overridden to capture the A location of the touch and compare it to the B button coordinates if the A falls within the boundaries of B then a rocket is launched.

DidBegin

Having rockets fired towards desired blood type nodes is useless without knowing if there was a hit. To obtain collision detection between objects they must be assigned with physic abilities. Each SpriteKit node comes with a .physicsBody property to assign an instance of SKPhysicsBody. However, each node must be a member of a category using .categoryBitMask and set with .contactTestBitMask category of nodes permitted to collide with.

Completing the physics setup requires a method to listen and deal with collisions during gameplay. Overriding the didBegin() helps in providing the nodes that have collided, accordingly various actions can be implemented. In this game, the only interaction is caused between blood type and rocket nodes. On collision, the launched type is added to the list of the target blood groups and an explosion particle is executed for a duration of 2 seconds.

Accelerometer Sensor

As previously mentioned, there are two input mechanisms players can use touch and drag the default option. The second way is controlling the rotation of jet character based on the physical device movement on the x-axis using the accelerometer hardware component feeds. By having an instance of CMMotionManger class of CoreMotion API, the game gets the ability to check for accelerometer hardware availability with isAccelerometerAvaliable, define the interval duration to receive feeds based on, starting, and stopping the feature. Afterwards, the RunLoop active object is used to process the timer event to handle the data received from the sensor. In the game case, only the x-axis double data is required to be passed into the updateJet_Rotation() method to process the values and update the jet rotation based on.

Particles

Bringing the game to life is a curtail task requiring an extensive set of animations and visual effects in order to optimize the user experience. Excluding the purpose of providing enjoyable movements on screen, having effects helps makes system actions clearer to the end-user and facilitate engagement. However, animation tends to pour the application with forming complexity and challenging task of modification which is why particles become the optimal solution to afford simulations. This game consists of explosions and rocket firing effects.

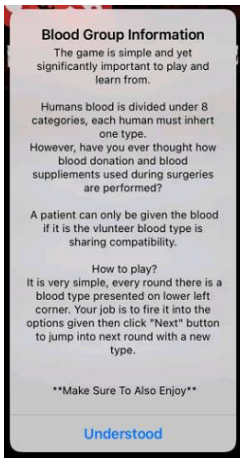
To achieve desired particles, unlike Android Studio requires a hardcoded approach the Particle Editor provided by XCode is used to create .sks files representing an effect for each. This tool offers a variety of properties to design advanced motions with a preview window giving an instant live version of the effect and provides templates of existing designs to use.

Sounds

Having high-quality graphics and a responsive experience might be enough to create a gaming environment but prompting sounds across the game brings it to life through players' ears. Sound effects play a significant role in increasing entertainment and informing players on the state of interaction by getting a reactive sound back. Therefore, the game is filled with various types of sound targeting different aspects in terms of sound effects and background music handled by SKAction API.

Game User Interface

(Figure 1) Navigation Controller

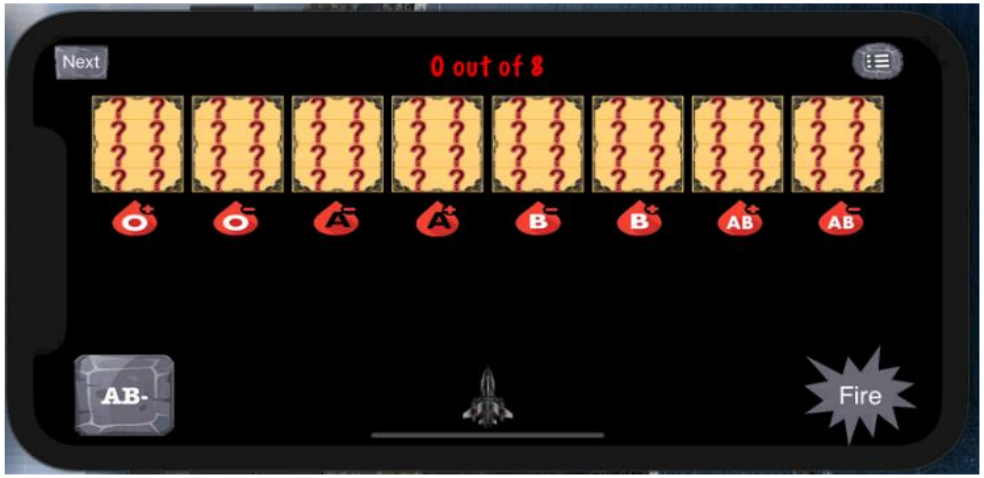


(Figure 2) About Window

(Figure 3) Splash Screen

(Figure 4) Main Menu

(Figure 5) Option Men



(Figure 6) Game Main Interface

Unified Modelling Language