Swipe A Selfie

Technical University of Denmark
DTU Compute

02128 - Softwareproject
Group 4

Anna Ølgaard Nielsen - s144437   Martin Dariush R. Hansen - s144459
Per Lange Laursen - s144456   Van Anh Thi Trinh - s144449

23 June 2015

Contents

1 Introduction

"Swipe a Selfie" is a card game application for Android that tests the speed of the player's reaction. The main purpose of the game is to match two photos shown on the screen to gain points. The application is thereby for entertainment and can be played by a person of any age. This report will briefly explain how the application is designed, implemented, and tested.

2 Gameplay

The essence of the application is to match two cards: a reference card and a card from the player's deck. The user has to swipe up to skip when the two cards do not match, and swipe down to keep two matching cards. If the bonus card or "Rush Hour" card appears, the player has to double tap on the card. The bonus card gives five extra points, and the "Rush Hour" card gives a 5 second "Rush Hour"-round.

3 Analysis

The game's main functionality consists of gestures, camera interaction, score, timer and image comparison. The specific functionalities and game features are shown below in the MoSCoW table.

Must:	Should:	Could:	Won't
Fling + tap gesture	Simple highscore	Sound effects	Online integration
Image import	Camera	Bonus points	Difficulty
Timer	Instructions	More image-themes	Orientation
Score		Graphics	
		Highscore system	

Table 2: MoSCoW table

4 Design

The application is made up of activities, retainable fragments, and different types of listeners. When starting the application, the first screen visible to the player is the main menu defined in *MainActivity*. This activity contains the three buttons: 'Play', 'Instructions' and 'Highscore'. The buttons navigate from the *MainActivity* to the buttons' corresponding activities within the application.

When the instruction button is tapped, *InstructionsActivity* appears with a slideshow of images that explain the rules of the game. The user can navigate through the images by swiping left and right.

When the play button is tapped, *PhotoActivity* opens where the user has to choose a bonus card. This can either be a default card or a personal photo taken with the device's default camera application. To be able to use this, the application uses the camera permission.

After the user has made a choice, *GameActivity* starts and the game begins. When the game is over, the user can go to the main menu or stay in *GameActivity* to play again. If the player's score is high enough to make it into the highscore list, the player can submit the score and enter *HighscoreActivity*. This activity is as previously mentioned also accessible from the main menu.

To handle a part of the game logic, we have used a retainable fragment with an AsyncTask. This fragment is called VerifyFragment. It checks the type of interaction that the player has performed and whether the two images are identical or not. Another fragment used in the game is a DialogFragment called *FinishDialogFragment*, where the options change based on the player's score.

The buttons in *PhotoActivity* are ImageViews with added OnClickListeners, while OnTouchListener is mainly used for the fling gesture in the game and the instructions menu. Within the OnTouchListener there is a custom gesture listener that makes it possible to have full control over the different types of gestures available in *GameActivity* and in *InstructionsActivity*.

5   Implementation

Activities are opened and closed by starting and finishing intents. In certain activities where the user should be able to return to the main menu and the main menu is not the previous activity on the stack, a new intent to the main menu is created with the added flag `FLAG_ACTIVITY_CLEAR_TOP`. This brings the very first activity (*MainActivity*) to the top and closes the others. Another used flag in the application is `FLAG_ACTIVITY_NO_ANIMATION`, which removes the animation during the transition between *PhotoActivity* and *GameActivity*. To remove the animation when an activity is closed, the finish() method is overridden with the added line "overridePendingTransition(0, 0)".

The only activity that starts with startActivityForResult() is the camera application. When a photo is taken, onActivityResult() is called with a request code, a result code and data. If the request code and result code are correct, a new intent to *GameActivity* is created with the data added as extra data. In *GameActivity*, the data can now be used to get the photo that the user has taken and apply it to the bonus card. The application checks whether the user has taken a photo or wants to use the default card with getStringExtra().

A String key word is used to determine the type of gesture that the user has performed in *GameActivity*. A down swipe has the key "keep", an up swipe has the key "skip", and double tap has either the key "bonus" or "rush" since double tap is used for both bonus cards and the rush hour card. The *VerifyFragment* class creates an interface that makes it possible to interact with the *GameActivity* class, in order to update the score, the current card and the reference card. These images are then sent through the AsyncTask where the photos' IDs are compared. If the images share the same ID and the key is "keep, the integer 'addPoints' is set to 1 and the boolean 'updateAllCards' is set to true. If the images are not the same and the key is "keep", 'addPoints' is set to -1 and 'updateAllCards' is set to false. After imageMatch() is completed 'addPoints' and 'updateAllCards' will be sent back to *GameActivity* and the score and cards are updated accordingly. If the key is "skip", 'addPoints' will be set to 0 and 'updateAllCards' is set to false. If the key is "bonus", 'addPoints' is set to 5 and 'updateAllCards' to true, and if the key is "rush", the "Rush Hour" mode starts. Within this game mode, 'addPoints' will be set to 1 and 'updateAllCards' to false whether the key is "keep" or "skip".

In *FinishDialogFragment* an AlertDialog.Builder is used to built a dialog. The dialog has different text fields and buttons depending on the score that the user has achieved during the game. If the score is a highscore the text is set to "Congratulations! You made your highscore" or "Congratulations! You beat your highscore" if it is the highest high score. If the score is a high score an editText box is added to the dialog where the user can enter his or her name and a submit button that will store the score and the name in a SharedPreferences when pressed. If the score is not a high score the message text is set to "You scored: XX score". *FinishDialogFragment* also uses an interface to interact with *GameActivity* if the user presses the start button.

The *Image* class extends the ImageView class, where we have overwritten the constructor so it accepts a custom ID, an image resource or bitmap, a bonus boolean and a rush hour boolean. The remaining functions in the *Image* class are getter methods for use with *GameActivity*, *GestureListener* and *VerifyFragment*. getDrawImage() returns the image resource stored within the object and getBitmap() returns the bitmap. In *GameActivity*, getBitmap() is used to check if an image has a bitmap that can be used as a resource, otherwise it will use getDrawImage(). isBonus() and isRush() are used to access the corresponding booleans from the *GestureListener*. getID() is used in *VerifyFragment* to get two images in the *Image* array's ID for the idMatch() check.

The *GestureListener* class extends `GestureDetector.SimpleOnTouchListener`, which implements two types of gestures: fling and double tap. onFling() checks what type of swipe the user has performed. If it is a topSwipe, *VerifyFragment*'s starts() method is called with the key "skip". If it is a bottomSwipe, *VerifyFragment*'s starts() method is called with the key "keep". The topSwipe and bottomSwipe check if the difference between the coordinates from the two events is greater than 120 and the velocity is greater than 200. The only difference between the two methods is the order in

which the numbers appear in the subtraction. onDoubleTap() checks if *GameActivity*'s current image is a bonus card or a "Rush Hour" card. If the 'bonus' boolean is true *VerifyFragments*'s start() method is called with the key "bonus". If the 'rush' boolean is true, the start() method is called with the key "rush".

In *HighscoreActivity* a top 5 list is shown with the five highest scores since the last time the list was reset. The highscore is kept in a SharedPreferences-class to make sure that the scores still appear even when destroying and recreating the application. The SharedPreferences keep the information as key-value pairs. These pairs have a key to both the name and the points that determine the place in the highscore. For example, the key "name1" will be the name of the first place in the highscore, and the key "point5" will be the points of the fifth place in the highscore list. These keys match the ID of each TextView shown in the `activity_highscore.xml` file. The methods in *GameActivity* involving the highscore save the highscore retrieved from *FinishDialogFragment*, and decides whether the score can make it into the highscore board or not. The method getMinKey() returns the key to the lowest score to replace, so that only the 5 highest scores are saved in the SharedPreferences. Every time *HighScoreActivity* is entered, the activity updates the names and points, so the values of the specific keys match the ranking. For this an "insertion sort"-algorithm is used to sort the points, and both sorted names and points are saved in two separate arrays. These arrays are then shown as TextViews in a GridLayout in the `activity_highscore.xml` file. The reset button clears the editor in the specific SharedPreference class and calls onResume() to update the view.

InstructionsActivity displays a background image, a menu title, an image with the instructions, a text to the corresponding image, an image number, and a back button. It contains an array of eight different images and an array with image text strings. The initial image is the first image in the array. When swiping left, the next image in the array is displayed if the last image has not been reached. Swiping right displays the previous image if the first image is not being displayed. The gesture detection is set up in gestureHandlingSetup(). The swipe is detected in the *CustomGestureDetector* class, which extends `GestureDetector.SimpleOnTouchListener`. When the detected horizontal swipe distance and speed are greater than certain values, the image number increases or decreases and the update method is called, setting the correct image, the correct image text, and image number.

The timer is implemented as a CountDownTimer with the time set to 60 seconds. The TextView showing the time is updated in the timer's OnTick() method. When the timer reaches 0, *FinishDialogFragment* appears and the game is over. However, if the "Rush Hour" card is double tapped, the boolean 'rushTime' is set to true. If the boolean 'rushTime' is true, the TextView will count down from 5 seconds while the timer is still running as before - that is, the actual time might be more or less than 5 seconds. When the changed TextView hits 0, the TextView shows the actual time from the timer with 7 seconds in addition. The TextView therefore shows 7 seconds more than the actual time that is left. This makes it look like two different timers where the first timer continues after the second timer has ended. After the actual timer terminates - which happens before the TextView reaches 0 - a new timer is created to show the remaining seconds until the TextView reaches 0. This was done since having two timers run at the same time caused the application to crash.

6   Testing

The application has been tested on the emulators *Galaxy Nexus API 18* and *Nexus 5 API 22* and a real *HTC One M7 API 21* to ensure that the applications runs fast and smoothly each time a new feature has been added.

7   Development process

Following the MoSCoW model from the analysis section, the first things implemented were the game logic, timer and score from the Must column. Since we had time left, the Should and Could functions were also added afterwards and additional game ideas such as "Rush Hour" mode. The way the CountDownTimer is implemented during "Rush Hour" is very complicated, because as mentioned earlier, the application crashed when we tried to add another timer. Therefore we had to implement the CountDownTimer in a very inefficient way, so the application could run. This could be improved a lot if we had more time.

8　Responsibilities

People	Main Responsibility	Other
Anna Ølgaard Nielsen (s144437)	HighscoreActivity	Highscore-methods (GameActivity + FinishDialogFragment) Rush Hour mode (GameActivity) Custom font
Martin Dariush R. Hansen (s144459)	InstructionsActivity	
Per Lange Laursen (s144456)	GameActivity VerifyFragment (part of GameActivity) FinishDialogFragment (part of GameActivity)	Danish Translation MainActivity Custom icon
Van Anh Thi Trinh (s144449)	PhotoActivity (camera implementation)	Bonus card (GameActivity) Rush Hour mode (GameActivity) Sound effects (GameActivity) Graphics

Table 4: Responsibilities table

9　Sources

- Android Developer: "AlertDialog" - 16-06-2015
  http://developer.android.com/reference/android/app/AlertDialog.html

- Android Developer: "Camera" - 16-06-2015
  http://developer.android.com/reference/android/hardware/Camera.html

- Android Developer: "CountDown Timer" - 16-06-2015
  http://developer.android.com/reference/android/os/CountDownTimer.html

- Android Developer: "Dialog" - 16-06-2015
  http://developer.android.com/reference/android/app/Dialog.html

- Android Developer: "DialogFragment" - 16-06-2015
  http://developer.android.com/reference/android/support/v4/app/DialogFragment.html

- Android Developer: "Dialogs" - 15-06-2015
  http://developer.android.com/guide/topics/ui/dialogs.html

- Android Developer: "Introduction to Android" - 15-06-2015
  http://developer.android.com/guide/index.html

- Android Developer: "FragmentActivity" - 15-06-2015
  http://developer.android.com/reference/android/support/v4/app/FragmentActivity.html#getSupportFragmentManager()

- Android Developer: "Taking Photos Simply" - 16-06-2015
  http://developer.android.com/training/camera/photobasics.html

- Cormen, T. H. et al. - Massachusetts Institute of Technolgy (2009) - "Introduction to Algorithms" - 3. edition - page 18 (Insertion sort)

- Flashkit - 18-06-2015
  http://www.flashkit.com/

- foobarpig "Disable Activity "slide" animation on StartActivity, Finish() and Back-Pressed" - 18-06-2015
  http://foobarpig.com/android-dev/how-to-disable-animation-on-startactivity-finish-and-backpressed.html

- Github: "aporter" - 17-06-2015
  https://github.com/aporter/coursera-android-labs

- Google Font - 19-06-2015
  https://www.google.com/fonts#

- SoundJay - 18-06-2015
  http://www.soundjay.com/

- Stack Overflow "Add bigger margin to EditText in Android AlertDialog" - 18-06-2015
  http://stackoverflow.com/questions/27774414/add-bigger-margin-to-edittext-in-android-alertdialog

- Stack Overflow "AlertDialog's setCancelable(false) method not working" - 17-06-2015
  http://stackoverflow.com/questions/8906269/alertdialogs-setcancelablefalse-method-not-working

- Stack Overflow "Android - basic gesture detection" - 19-06-2015
  http://stackoverflow.com/questions/937313/android-basic-gesture-detection

- Stack Overflow "Custom fonts and XML layouts (Android)" - 19-06-2015
  http://stack-overflow.com/questions/2376250/custom-fonts-and-xml-layouts-android

- Stack Overflow "Hot to show a dialog to confirm that the user wishes to exit an Android Activity" - 19-06-2015
http://stackoverflow.com/questions/2257963/how-to-show-a-dialog-to-confirm-that-the-user-wishes-to-exit-an-android-activity

- Stack Overflow: "Limit text length of EditText in Android" - 18-06-2015
http://stackoverflow.com/questions/3285412/limit-text-length-of-edittext-in-android