DISMATH Final Project

DATMATH Quest II

Deticio, Ramiel, Keh, Jefferson, & Ko, Dayoung
Gokongwei College of Engineering
De La Salle University
Metro Manila, Philippines
ramiel_deticio@dlsu.edu.ph

Abstract— DATMATH Quest II is an app made in MIT App Inventor that incorporates concepts learned in our discrete mathematics class such as algorithmic thinking and set theory. The app is an adventure game that also functions as a DISMATH reviewer/learning tool for students who want to learn lessons in logic and dismath concepts.

I. Introduction

DATMATH Quest II is an adventure story game that also contains educational and arcade elements. The game itself also features branching dialogue, a persistent (works when app is closed) saving system, minigames, and educational discrete mathematics content all tied up in a linear story. The app itself mostly contains sprites, music, backgrounds and other assets that were independently made and then incorporated and programmed into MIT App Inventor.

II. Contributing Factors

A. DISMATH

Programming requires plenty of algorithmic thinking and set theory. It also requires logic and mathematical thinking. All these skills were taught and refined in DISMATH.

III. Process/Method of Implementation

A. Main Menu

This is also the start menu of app. This is the simplest one out of all the screen we made as this only needed to contain few functionalities. These include linking the "start" button to the dialogue screen, which is where most of the plot takes place. The code goes like this.

```
when "start button" is clicked, do
  open "dialogue screen"
```

The other component of this screen is the "continue" button. This would load a state of the application that was saved through another component of the application. This can be done using the "tinydB" component of the MIT App Inventor, which we used to save a list that would indicate the player's current progression.

```
when "continue button" is clicked, do
  open "dialogue screen"

<in the dialogue screen>
when "dialogue screen" initialize, do
  call tinydB list
    when tinydB list length = n
        initialize dialogue at index x
```

B. Dialogue

The dialogue screen is the screen where all of the story happens. All the dialogues are stored in this screen and this screen is linked to the mini-games through the dialogue index. The dialogue index changes as the player clicks the dialogue box and each index corresponds to specific actions (set as functions) and text.

```
dialogue index <- 1
dialogue_all <- {text1, text2, ..., textn}
when "dialogue box" is pressed, do
  for all index <- { 1 to n }
        Set "dialogue box" text to item in list w/
        index "dialogue index"
  for all index {1 to n}, do
  Action (assigned to each n)
```

C. Thought Tapper

The thought tapper made use of basic looping algorithms to randomize the "thought"'s position. The second part of this code is the algorithm is to increase the points whenever a "thought"is tapped.

```
Initialize index <- 1
While time = > 20, do
        Set image1 to random x and y
        Index = index mod 4  +1
While time => 10, do
        For index = 1,3 do,
                Set image1 to random x and y
                Index = index mod 4  +1
        For index = 2, 4 do,
                Set image2 to random x and y
                Index = index mod 4  +1
While time => 0, do
        For index= 1, do
                Set image1 to random x and y
                Index = index mod 4  +1
        For index=2, do
                Set image2 to random x and y
                Index = index mod 4 +1
        For  index=3, do
                Set image3 to random x and y
                Index = index mod 4  +1
        For index=4, do
                Set image4 to random x and y
                Index = index mod 4  +1

Initialize score = 0
When image1, image2, image3, or image4 is pressed, do
        Set score <- score+1
```

D. Memory Mania

a. Main Pseudocode

```
initialize
compare = empty set
while time > 0 or score < 6
when button is pressed
        put button in compare
        if size of compare = 2
                do compare
                if match = true
                        score ß score+1
                set isVisible for compare to false
                if match = false
                        do reverse
```

b. Initialization

The program's initialization is composed of changing the image of each card to the card back state, setting the timer intervals to 0 and enabling and disabling the proper timers. The timers used in this part of the program are the main game timer, and the two timers used for the animation of the cards.

Assignment

Part of the program initialization is the assignment of the values to each card in the game. This involves two sets of lists, with each list corresponds to a list in the same position in their respective sets. There is another set that corresponds to each card in the set. The assignment algorithm applies a brute-force approach in which it gets each value in the two sets of lists and replaces the values in the set that are assigned to each card. This involves the manipulation of the given sets and the indexes (position in the set) of the items in each set. It does this by continuously grabbing a value from each list and then checking if it's already been assigned. It will continue to grab a value until it finds one that is not assigned then it replaces the value of a card by with the corresponding list. These lists are actually the names of the images that are used for the animation of the card.

c. Card Animation

When a button (card) is clicked, the animation timer is started and this works by each card having a list of their own corresponding frames. Each card would have the same first few frames as it is being opened, but then would be continued by its assigned list of images that corresponds to the card's value. The timer will then add the value of the card to the comparison set, where it will be compared to a value already on the list or will be on the list.

Each card animation has a total of 8 frames, four of which are similar to all cards. The other animation, when a card is flipped back to be face down, works the same way but in reverse, and it will apply for both cards in the comparison set. At the end of that animation, however, it will empty the comparison set for the next pair of cards. To disable interruptions, all

cards are disabled for clicking and enables once animations are finished. The clicked button is also disabled until the corresponding animations happen so that the player can't press the same card twice and start the comparison procedure with the same card (thus making that one card disappear since you're comparing that card to itself).

d. Comparison

The comparison works by checking if both lists assigned for the cards that were clicked have the same index. If true, the buttons will then disappear, be replaced by invisible objects that will maintain the alignment of the cards, update the score variable, then cause the phone to vibrate to provide positive feedback. If not, it starts the reverse flip animation timer as described previously. The comparison is also responsible of checking whether a card was the first card of two to be selected. This is done by checking the length of the comparison set. The first card will also be stored in a temporary variable for future use (mainly in the animations).

e. Conclusion

When the timer runs out or the maximum score is achieved, the program stores the score and returns the player back to the main dialogue screen.

E. DATMATH Test

a. Main Pseudocode

```
initialize
while timer enabled = true
        when answer button is clicked
                do check
        when time = 0
                disable all buttons
        when time = -5
                do randomize
                if QuestionPool length < 5
                                do getQuestion
```

b. Initialization

The initialization of the quiz program involves the setting and the starting of the main game timer, the randomization of the next question and the order of the answer choices, and the assignment of all those elements in the corresponding buttons and labels.

c. Assignment

The randomization of the order of the answer choices is a given algorithm because the main code sets it so that the original answer list would show the first item in the list to be the correct answer. A new set is to be made that uses a brute-force approach to assign 1 to 4 in that set by continuously checking if they've already been assigned to that set.

The next phase of the assignment algorithm involves picking a random question from the question list and then getting the index of that question to get the corresponding list of answer choices from the answer list. This question is then asked to the QuestionPool, an initially empty set. The index of each question is equal to the index of its corresponding answer choice list in the master answer list. The answer list is basically a set of lists of answer choices. This way, it is very unlikely for each playthrough to involve the same questions in the same exact order. In fact, there are about 30240 different possible permutations that can be made with the same list of questions. Each button would then get an index assigned to them depending on the order the numbers 1-4 were placed in the set that was made in the randomization phase. This would then cause the order of the answers for each question to be randomized every time. This would add more randomness to the question and answer choices.

d. Checking

Each answer choice list was made so that the correct answer would be the first item in that list. The checking algorithm simply gets the text of the button that was clicked, finds it in the original assigned answer choice list and checks it if its index is one. If so, the button animation timer would start depending on the Boolean value. The button will turn green or red depending if it's correct and then restore to its default appearance once the timer is up. It will also then randomize the answer choice order again and do the assignment of a new question if the questions asked (QuestionPool) still hasn't reached a quantity of 5.

F. Gotta Go Fast!

Gotta go fast makes use of loops to simulate animations. The running animation is a loop of images of the character in different running positions. The moving clouds and obstacles are just looped to move left whenever the timer is enabled. Different timers are set to different elements to simulate differing speeds. An example of the movement is as follows.

Initialize obstacle x <- 320px, obstacle timer enabled
When obstacle timer interval passes, do
    Move obstacle x <- obstacle x - 10
When obstacle touches screen edge, do
    Move obstacle x <- 320px

IV. Conclusion

The MIT App Inventor is a useful program to learn to create apps. It simplifies code into blocks and helps you understand how code works. While this is not as versatile and powerful as C, this is a great way to learn programming. This program incorporates programming ideas that will prove to be helpful even when using other languages. Some examples of these ideas are algorithms, set theory, logical equivalences, and others. Examples of how these are used are mentioned above.

References:

www.appinventor.org/book2
https://clipartfest.com/download/d048d33c0c01846b0ee47d97de7315a270b1291f.html

Appendix:
Member contributions:
Dayoung Ko: composer, artist, designer, DISMATH information contributor
Ramiel Deticio: programmer, layout design, scriptwriter, DISMATH information contributor
Jefferson Keh: programmer, layout design, playtester, DISMATH information contributor

Code Pictures: https://drive.google.com/open?id=0BxSAEGG--befV1FqeC1JX25GVE0