Neil Tzu-Yang Ni

Summary

The GrandMa App aims to provide a photo-sharing platform for family members, with one main goal to simplify the sign-in/login process for technically difficult members. This report discusses the logical thought procedure from the origin of the problem to the most recent stage of the solution, heavily stressing on the technical resolution on the iOS front-end side.

Problem 1:

How do we allow grandma to login without asking her to sign up?

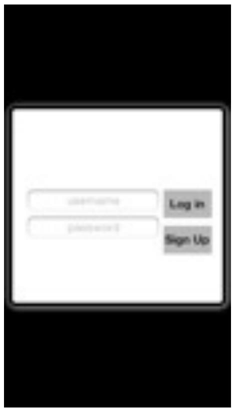

Image 1. Login View 1

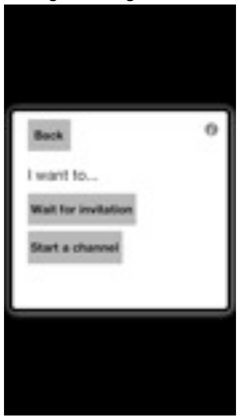

Image 2. Login View 2

Upon opening a newly installed app, the app requests a private ID from the server, and store it permanently within the app. The private ID serves as a key for server to associate the app with the device, so for example, when user choose to login, then the server knows how many different devices a particular account holds.

When it comes to grandma's case, grandma needs to click on "wait for invitation" as shown in Image 2, and the app will request an invitation code from the server. With both private ID and invitation code, the server already has enough information to identify grandma since one invitation code only associate with one private ID. This verification procedure starts from the very beginning of the app's life cycle, and at the end of each verification, whether the user is getting invited by another user, or login with their old account, the app retains the basic information such as the user's account name, display name, family ID and invitation code.

The following is the main pseudo code inside ViewDidLoad():

```
if username: //having a username means it is not the first time using.
    Start fetching the images from server and set up images.
else:
    if invitation_code: // this means the user has not been invited
        Setup the invitation code view which displays only the invitation code. Start a
        timer that fetches the invitation status in scheduled interval.
        If server responds with success status, grandma would be given a system
        generated username based on her relationship inside the family, and then the
        invitation code view will slide down, and the app will start fetching the images
        from server and set up images
    else: // this means the user is first time using
        Setup the login view as in Image 1. If the user already has an account, then it will
        start fetching images automatically. Else if the user choose to wait for an
        invitation code it will transition to the invitation code view.
```

The above storing mechanism is done by using NSUserDefaults. All the accessors method for storing and retrieving are inside the class UserInfoManager.

Problem 2: How to display the images?

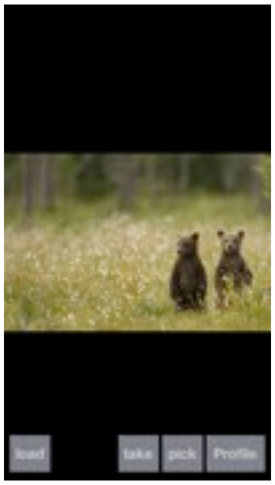

Image 3. Scroll View 1

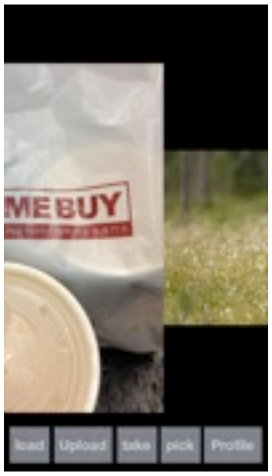

Image 4. Scroll View 2

When a user is at this step, the app has already retrieved all the user’s information from server. And for fetching new images, the app needs to request the server with user’s family ID, and in response, the server sends back image urls for the app to download.

The image display is accomplished with Apple’s official UIPageControl and UIScrollView. Each page of the PageController owns a ScrollView. The ScrollView allows the image to be zoomed in and zoomed out, and PageController allow the images to transition continuously like shown in Image 4, where the image is swiped to the right. The right image in Image 4 is actually just chosen from the photo library. User can in fact choose to either pick an image from the photo library or take a picture right away. And once the images are chosen or taken, user will click “Upload” to send the image to upload the image to the server. If the user slides back to the images that already exist on the server, the Upload button will not appear like in the case of Image 3.

There are two important PageController delegate methods that keeps track of the current view, previous view, and next view:

```
- (UIViewController *)pageViewController:(UIPageViewController *)pvc viewControllerBeforeViewController:
(PhotoViewController *)vc;
- (UIViewController *)pageViewController:(UIPageViewController *)pvc viewControllerAfterViewController:
(PhotoViewController *)vc;
```

PageController always keep up with these three views, and throws away the previously allocated views outside of these three indices. Therefore, the decision such as determining whether or not the upload button should be shown can be executed inside these two methods, since both of them carry the information about the next view.

When images are fetched, they are mapped into CoreData objects, and stored into a temporary array that act as the input for UIPageControl. The index of the array is therefore corresponding to the index of the page. Selecting an image would push the image information dictionary at the front of the array. However, the representation of dictionary object is much different than the CoreData object, so when the array is being passed around between different classes, there has to be a general way to extract the path reference among different objects. Therefore, when the images are being rendered inside the class ImageScrollView, there are three specific ways to find out the path references.

One thing to notice here is that since both UIPageControl, and ScrollView are official classes given by Apple, the multitouch interactions such as swiping can be tricky to detect. In general, it would be really helpful to respond to user even when the user slightly touch the view. However, since ScrollView already inherit pinch gesture for zooming, and UIPageControl already inherit swipe gesture for sliding ScrollViews, it is tricky to find out at what view is the gesture is actually happening since ScrollView is nested inside a UIPageControl.

Problem 3: Response to user interaction with third party library

Two third party library are used inside the app. One is MBProgressHUD, and the other one is RestKit. The three images below are the implementation for MBProgressHUD, which is

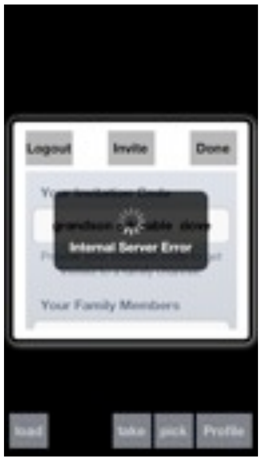

Image 5. HUD 1

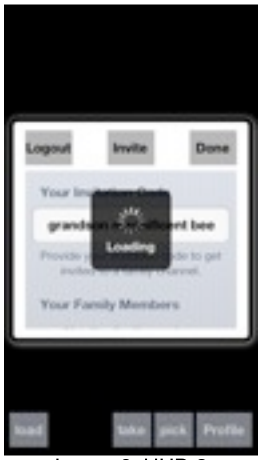

Image 6. HUD 2

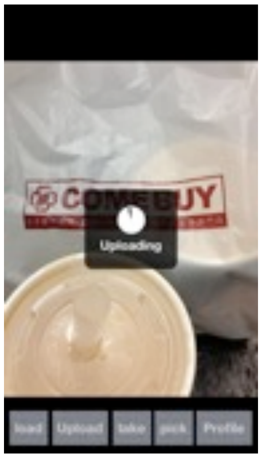

Image 7. HUD 3

tightly combined with the responses from RestKit's delegate method.

The RestKit has many delegate methods that determine whether the request is sent, received, time-out, or failed. One example using RestKit is in Image 7, the percentage of the pie is updated precisely with the total data sent divided by the image size. Another example for using RestKit is when the app start fetching the images, RestKit's RKObjectLoader can directly map the json response into CoreData objects.

The other important response to user interaction is the rotation handling. Fortunately iOS 6 provides Auto Layout, which is an easier way to manipulate the position of the rotated frame. The position for every element can be set relatively to the other, so even when the device is rotated, the element would remain relatively from each other.

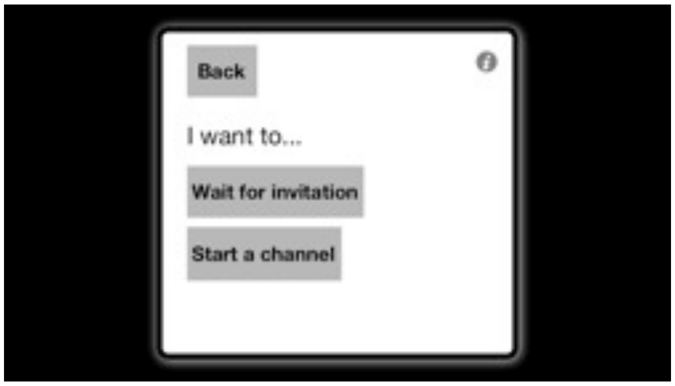

Image 8. Login View 3

Image 8 is the new display after Image 2 is rotated on sideway.

Current Stage versus Future Direction

Currently the app is not functioning completely as we expected, specifically we have not written code that allows the app to automatically update images to all the devices in the same family. The next implementation we were planing on doing is the timing for image update. For example, when the image is scrolled all the way to the left, the action should trigger a request for updating the most recent photo, and if the image is scrolled all the way to the right then it means user wants to see older photos, and it needs to request older photos before that point. The code on the server side is written, however, it is really tricky to sense the gestures at the end of the image array since the two official controllers have priority to respond to all the touches, and it is not easy to overwrite the official methods.

Going along with the images update is the storing decision for CoreData. Our next step is the management of the cache size. CoreData has to preserve a certain amount of images so that user does not have to fetch all images again when they start the app next time. However, there should not have too many images kept in the system either. So at the end of the app cycle, the app needs to delete excessive photos that it fetches, keeps the pointer to the most recently fetched photo, and save the current page index for next time to use.

Currently the login view contains six view controllers, which can essentially be minimized into one. It was easy to separate out all the views by their individual functions. However, the code gets a little clumsy at the end, and becomes a little hard to keep track of what view controller is currently using. One problem on top of having multiple view controllers instead of one is that the app needs to allocate six view controllers, which is a little overkill. In general, the code can eventually be benefited from memory checking on every stage of the app cycle. One more thing to improve is the compatibility. Since the app has been simulated inside iOS 6 on iPhone 5 simulator, the app can be further extended to iOS 5 and devices in other dimensions.

Discussion on the functionalities

At the beginning of the semester, there was only a few app examples that we can find online, but right now the option has become much more, and importantly the services are provided big companies like Facebook, Line and Twitter.

Facebook has launched its own version of photo-sharing app named “Camera” that displays only the timeline for images, with images’ messages, comments and likes. In some sense it is what we are trying to go for, but “Camera” makes uploading images really easy and really useful with its own customized upload controller that does the basic image modifications. More amazingly, recently Facebook is asking if user would like to sync all their photos up to their Facebook account, with an option not to show them publicly, but only act as a sort of cloud service.

Line on the other hand handles very well on the first login problem that we tried to come up with. Line simply associate each phone with its phone number, or an account if the person does not use iPhone. In all cases, when the person make a photo call, or say adding a new contact and communicate with their new friends, Line uses that new phone number to find out the friends' Line account and add it into Line automatically. It does not require user to login after the first time, or if the user has a phone number, it does not ask for the user to log in at all.

The code is at:
http://faceserv.cs.columbia.edu/svn/projects/grandma/trunk/frontend/Demo/