AURI
An Augmented Reality Android-based Food Exploration Application

Alan Burstein, Jinghu Lei, Chih-Wei Tung, Chunxi Wang and Chuan Xing Zheng
Boston University
https://github.com/jinghul/auri

***Abstract*— In this paper we describe the motivation, technical challenges and details, and analysis of the AURI project.**

I. INTRODUCTION

After the phenomenon of Pokemon GO (an augmented reality (AR) mobile game developed by Niantic and Nintendo), its clear that AR is going to be the next big mobile trend. The gaming industry is by all accounts the pioneer in the AR mobile app development. However, there are many famous AR applications outside of gaming. For example, the IKEA Place app, Sephora mobile app, and entertainment apps like Instagram. Nowadays, AR development has great potential for many businesses. The technology can provide an interactive and intuitive user experience, user engagement, and a great way to present products and communicate with target audiences. With AURI we strive to develop an AR android-based food exploration app as a successor to traditional applications such as the Yelp mobile application.

Imagine on a Sunny day, you are strolling down an unfamiliar street filled with salons, boutiques, and fabulous dining. After two hours of shopping on this enchanting street, you hear a growling sound from your stomach. The next thing that comes to your mind is deciding on which nearby restaurant to go to. You open AURI using the AR mode on the app and quickly look around at the restaurants nearby. AURI uses the devices camera to search for restaurants near you. It scans, finds, and positions nearby dining options right on your camera view. You click on a restaurant to get relevant information. After reading the reviews, you decide to eat there. Remarkably, AURI is one stop shop for finding all relevant dining information while you are outside. Its unnecessary to switch between apps like Yelp and Foursquare to search for dining options.

A. Use Cases

While designing Auri we had several core usages in mind and focused our development process and made tradeoffs to minimize degrading the experience of the following use cases.

- A user walks down a street and wants to quickly explore nearby food options in camera view.
- A user wants to explore all nearby options positioned on a map
- A user wants informations about a specific restaurant, eg. price, reviews, and star rating.
- Remembering intriguing restaurants for next time.
- At night or in the winter, when the user cant see nearby restaurants clearly, visualize the nearby options.
- A user travels abroad and wants valuable details about unknown restaurants.

II. AURI WALKTHROUGH

Augment Reality was introduced to Android in late 2017 through the ARCore package and requires API level 27 or above. For the best experience with AURI ensure that your device has a reliable data connection (Wifi, 3G, or 4G) and is one of Googles supported devices for AR. See Figure 1 to see a walkthrough of AURIs components and how it relates to user action flow while using the application.

A. Onboarding and Authentication

Onboarding is served to first-time users in order to direct them through the apps interface. Users lose interest when an app is confusing and difficult to navigate. We use a brief onboarding phase to help acclimate first-time users to AURIs features. The onboarding screens educate users about the functions and benefits of AURI.

After onboarding, the user is redirected to the login page. If the user hasnt register an account before, then the user needs to go to the registration page to register with AURI. The login and registration pages require users to input their emails and passwords. After the user registers or logs in, they are guided to AURIs main activity page.

B. Homepage: Main Activity

The main activity or homepage is the landing page for AURI. The homepage consists of a map, a search bar, and a floating action button. The floating action button utilizes the Fab-Speed-Dial library and makes use of the menu resource in order to present a list of actionable buttons. There are a total of four buttons, which are nearby restaurants, favorite, AR mode, and setting.

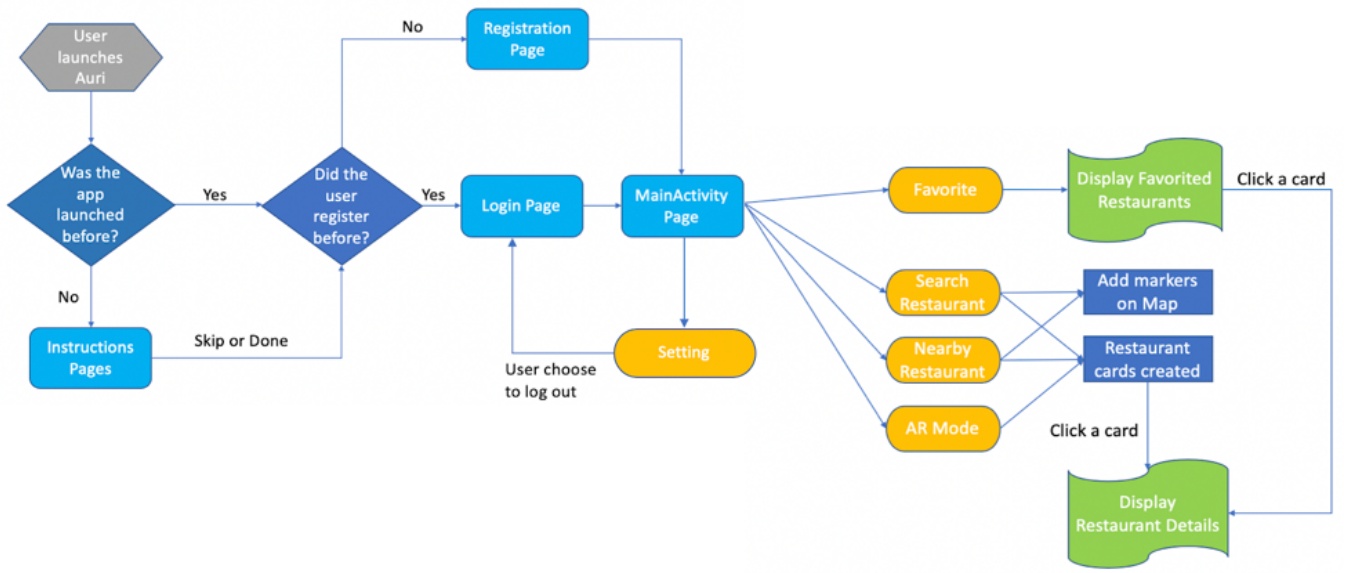

Fig. 1. The logical flow of AURI

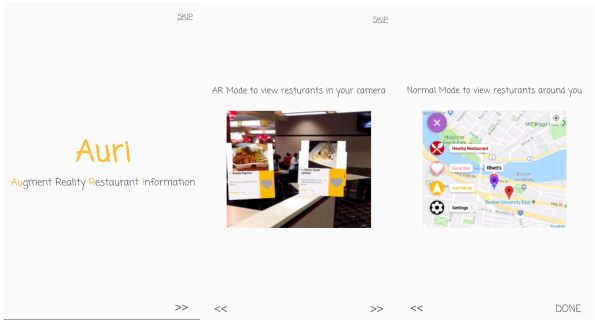

Fig. 2. Onboarding cards

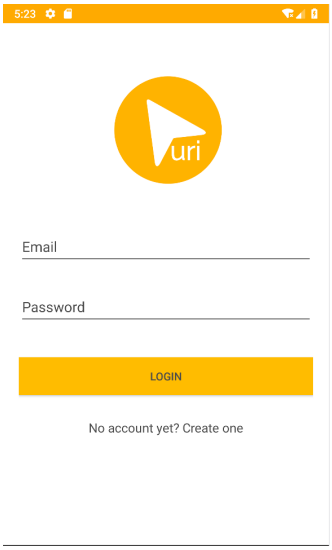

Fig. 3. Login page

C. Nearby Restaurants

The nearby restaurants action button will mark all of the nearby food options on the map in the homepage. Concurrently, cards are created for all of these nearby food options and appear on the bottom of the homepage as a infinite cycle ViewPager. Users can swipe left or right on the cards and the corresponding restaurant marker will reflect the showed restaurant by being highlighted. Also, if users click on the marker, the card will change respectively. Users can click on the favorite button on the card to save or remove the restaurant in the users favorite list. Cards are clickable too. After clicking on a card, users are taken to a detail page (Section 2.H).

D. Favorites

The favorites page is a RecyclerView for displaying restaurants that the user saved. In this page, we get the photo and the name of the restaurant from the Firebase Realtime Database and display them on an itemView which include a TextView and an ImageView. Each itemView is clickable and so that when a restaurant is clicked user will be directed to restaurants detail page (Section 2.8). Since swipe to delete is a prevailing paradigm users are accustomed on mobile platform, this function is implemented by attaching ItemTouchHelper to the RecyclerView to watch for deletion swipes and then the Firebase Realtime Database is updated.

E. AR Mode

The major motivation for creating AURI is centered around having an augmented reality interface for viewing restaurant information. This page is a fullscreen display of what your camera with restaurant cards anchored to the real life location of the restaurants. This means that right on your screen you can see restaurant cards placed on top of the restaurant

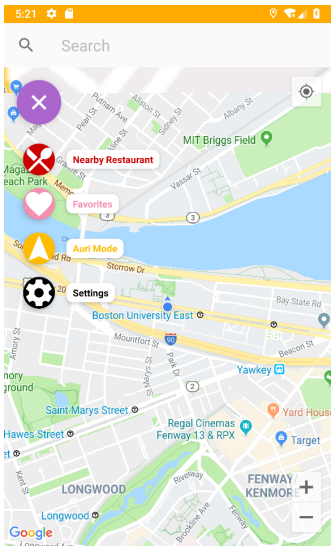

Fig. 4. Homepage

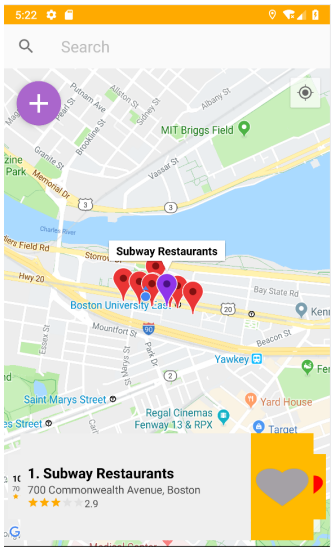

Fig. 5. Nearby Restaurants

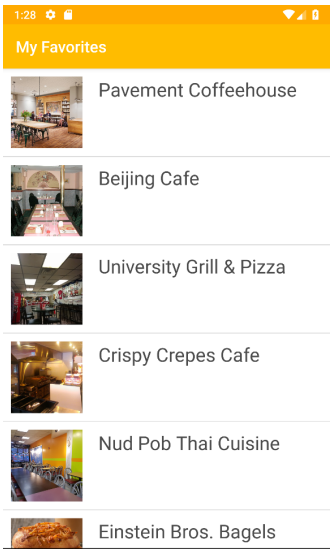

Fig. 6. Favorites Page

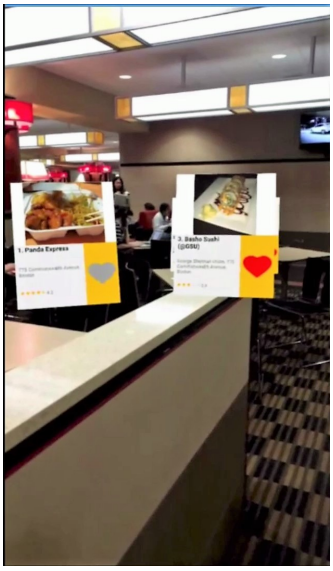

Fig. 7. AR view while inside a foodcourt

you are viewing in your camera. This functionality also works regardless of time in the day (during night it proves useful to show difficult to see places). We place overlapping restaurants into buckets so that you can slide through them in an intuitive way. Every 2 minutes it queries for nearby restaurants positions and replaces the cards when necessary.

F. Settings

The settings page is where users set their preferences. For now, there are three preferences: change a profile photo, reset password and log out. The first two preferences will connect to the Firebase Realtime Database when they are used. In order to get better performance, the data of profile photo is also stored in SharedPreferences. Extensions such as setting the radius or nearby restaurants and supporting more languages (Section 4.2) will be integrated here in the future.

G. Search Bar

The search bar is at the top of the homepage and lets users search for restaurants by keywords. It is a PlaceAutocompleteFragment from Google Place API and returns place predictions in response to user search queries. By setting a place-selected listener on it, it will get the place that user selected and get the place information with a place id. After obtaining the place information, a marker will show up on the map, and the card view for showing a brief detail of the place will be created. And the card is clickable for displaying the restaurants details (Section 2.H).

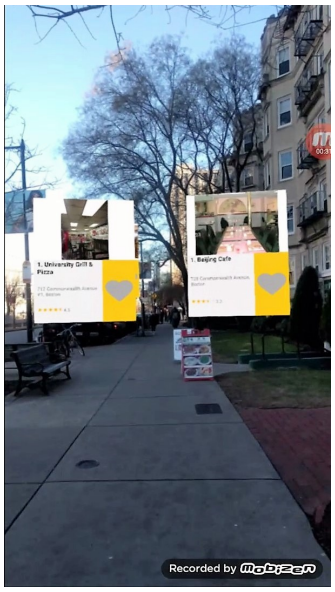

Fig. 8. AR view during the day

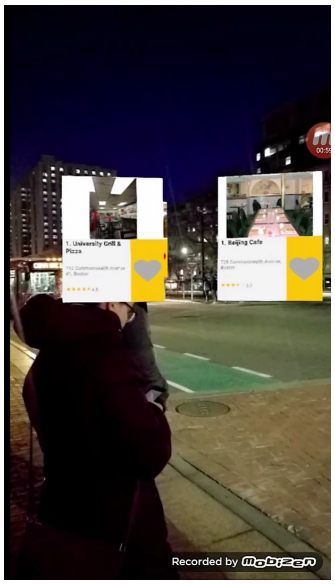

Fig. 9. AR view at night

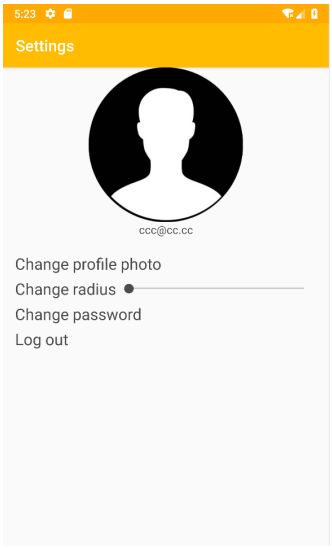

Fig. 10. Settings page

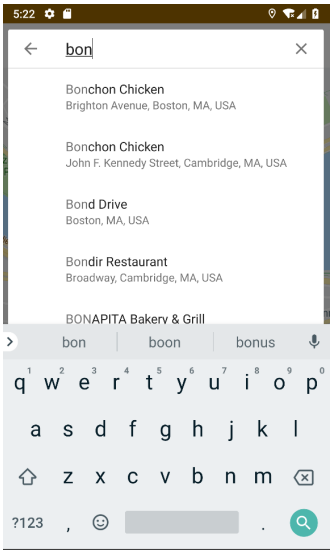

Fig. 11. Searching for Bonchon Chicken

H. Restaurants Detail Page

Each restaurant has useful information for users, and we wanted to ensure they could view all of it in one page. This page contains the restaurants name, a map, star rating, price level, reviews, and three buttons. The three buttons are for calling the restaurant, browsing the restaurants webpage, and favoriting the restaurant to the users favorite list (Section 2.D). The map is used to show the user where the restaurant is located. The star rating and reviews are valuable details about the restaurant. Reviews are shown in a ListView.

III. Implementations & Challenges

To access shared resources on the framework, Android applications request the specific permissions they require. Some of these permissions enable the app to use phone functionality, to access the network, control the camera and other hardware sensors. While developing the app, required permissions should be specified in Android manifest file. For instance, AURI requires the phones camera hardware functionality for AR mode, hence permission was specified in the Manifest file. Figure 13 shows all the permissions required for AURIs features. The components mentioned in Section 2 are developed by using various handy approaches, dependencies and/or libraries, which is conferred and pointed out below:

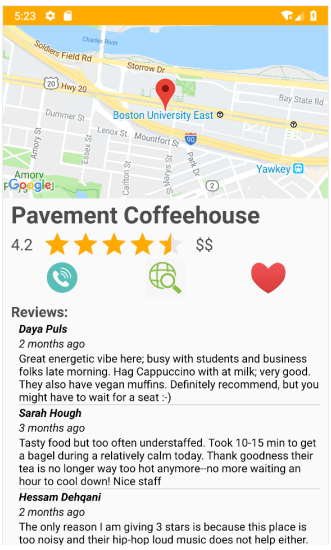

Fig. 12. Details on Pavement Coffeehouse

Fig. 13. Auri permissions in manifest

A. Firebase Authentication and Realtime Database

In order to provide a personalized experience and record user information, we need to authenticate. Firebase is Googles backend-as-a-service platform. Firebase Authentication and Realtime Database are the two backend services used in AURI.

Firebase Auth service helps the login, register, and logout process. After enabling Firebase Auth and adding the Firebase Auth dependency to the project, two activities are created for login and register, which are named LoginActivity.java and RegActivity.java. Firebase provides createUserWithEmailandPassword() method to create a new user with the email and password inputted in the textViews by the user as you can see in Figure 14. If registration succeeds, the auth state listener will be notified and logic to handle the signed in user can be handled in the listener. Then, the user is directed to the homepage of the app. If registration fails, a toast message will be displayed to the user. For login, Firebase provides signInWithEmailandPassword() method to sign in the user. Firebase Auth will do all the email and password checking. If sign in succeeds, user will be directed to the homepage. Toasts will be displayed in case sign in fails. The code for logging in is shown in Figure 15.

The Firebase Realtime Database is a cloud-hosted database. Data is stored as JSON and synchronized in realtime to every connected client. By getting an instance from firebase and referencing the location, we can set value in it. To read data from the database, we need to add a ValueEventListener to the location that we referenced. The onDataChange method in the listener will be triggered whenever the data changed.

B. GPS Location

Google Play Services is a bundle of APIs that are used for interacting with Google Map interface and related information such as the devices location. Google Play Services provide us the freedom to use the newest APIs for popular Google services without worrying about device support. For integrating the Google Map, the onMapReady() functionality is called when the map fragment is ready to be used with Google Map service and then markers can be added on top of the map. After this setMyLocationEnabled() is used to enable the location layer which will allow a user to interact with their current location. To get and update the users location, Fused Location Provider client were implemented in our getCurrentLocation() to retrieve GPS location data (longitude and latitude) in order to provide the highest accuracy data as seen in Figure 16. Moreover, onLocationChanged() is declared to simply get coordinates of current location and used to compare with the last coordinates. If they are different, then the current longitude and latitude are updated. AURI ask the user for permission before the map is loaded in the homepage. This is done by implementing getLocationPermission(), which checks if the location permission is granted as shown in Figure 17. If it is granted, Fused Location Provider client has the permission to obtain the devices location.

C. Google Places API

In our project we use both the Google Place SDK and API. Most of the restaurant details that we currently to serve are from the Place SDK except for restaurant reviews which are from the Places API. By using the Place SDK, we can use Place Autocomplete to implement the search bar (Section 2.G), get restaurant photos, get restaurant details (including name, latlng, rating, price level, phone number and website url). By using the Places API, we can request using an HTTP URL and return a JSON response which includes 5 restaurant reviews in total.

D. Compass and position

Since our application relies on having extremely precise measurements, minor deficiencies in Compass would ruin the user experience. Android hardware consistently has difficulty with compass precision and often mis-calibrates itself (especially while holding the phone vertically). The built in compass sensor is thus unable to give a stable and accurate result. We fixed this by using a public library https://github.com/hoananguyen/dsensor that uses more measurements to and is built for vertically holding the phone.

Once we have the sensor and API data, we can position the restaurants for AR. To do this we calculate the position

```java
// Create User with Firebase Auth
mAuth.createUserWithEmailAndPassword(Email, Password)
        .addOnCompleteListener( activity: this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                try {
                    //check if successful
                    if (task.isSuccessful()) {
                        Log.d(TAG,  msg: "createUserWithEmail:success");
                        //User is successfully registered and logged in
                        //Go to Main Activity Page here
                        Toast.makeText( context: RegActivity.this,  text: "Registration Successful",
                                Toast.LENGTH_SHORT).show();
                        currentUser = mAuth.getCurrentUser();
                        sp.edit().putString("account", currentUser.getUid()).apply();
                        sp.edit().putString("email", Email).apply();
                        sp.edit().putString("profile", "").apply();
                        Intent intent2 = new Intent( packageContext: RegActivity.this, MainActivity.class);
                        intent2.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
                        startActivity(intent2);
                        sp.edit().putBoolean("logged",true).apply();
                        finish();
                    }else{
                        // If sign in fails, display a message to the user.
                        Log.w(TAG,  msg: "createUserWithEmail:failure", task.getException());
                        Toast.makeText( context: RegActivity.this,  text: "Please make sure it is an Email & Password must be at least 6 characters!",
                                Toast.LENGTH_LONG).show();
                    }
                }catch (Exception e){
                    e.printStackTrace();
                }
            }
        });
```

Fig. 14. Registration code

```java
// Use Firebase Auth to do the login task
// Check if the username and password match with the ones in the Firebase
mAuth.signInWithEmailAndPassword(Email, Password)
        .addOnCompleteListener( activity: this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (task.isSuccessful()){
                    currentUser = mAuth.getCurrentUser();

                    // saved account name, so that after login can get the account name
                    sp.edit().putString("account", currentUser.getUid()).apply();
                    sp.edit().putString("email", Email).apply();

                    // if login successful, then enter the main activity page
                    Intent intent2 = new Intent( packageContext: LoginActivity.this, MainActivity.class);
                    intent2.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
                    startActivity(intent2);
                    sp.edit().putBoolean("logged",true).apply();
                    finish();

                }else {
                    // else toast that there's an error
                    Toast.makeText( context: LoginActivity.this,  text: "Couldn't Log In, Please check your info!",
                            Toast.LENGTH_LONG).show();
                }
            }
        });
```

Fig. 15. Login code

```
public static void getCurrentLocation(Context context, LocationListener locationListener) { // get the device location
    Log.d( tag: "thisClass",  msg: "getDeviceLocation: getting the devices current location");
    // use the location service
    mFusedLocationProviderClient = LocationServices.getFusedLocationProviderClient(context);

    try{
        if(mLocation){ // if allow to find device location
            Task location = mFusedLocationProviderClient.getLastLocation();
            location.addOnCompleteListener(new OnCompleteListener() { // @onLocationReturned and LocationListener
                // Perform the location listener
                @Override
                public void onComplete(@NonNull Task task) {
                    if(task.isSuccessful()){
                        try {
                            Location mLastLocation = (Location) task.getResult();

                            LatLng latLng = new LatLng(mLastLocation.getLatitude(), mLastLocation.getLongitude());
                            double latitude = mLastLocation.getLatitude();
                            double longitude = mLastLocation.getLongitude();

                            locationListener.onLocationUpdated( success: true, latitude, longitude);
                        } catch (NullPointerException e) {
                            locationListener.onLocationUpdated( success: false,  latitude: 0,  longitude: 0);
                        }

                    }else{ // Error of finding the current location
                        Log.d( tag: "errorfinding",  msg: "onComplete: current location is null");
                    }
                }
            });
        }
    } catch (SecurityException e){
        Log.e( tag: "cantgetLoc",  msg: "getDeviceLocation: SecurityException: " + e.getMessage() );
    }
}
```

Fig. 16. Location listener code

```
private void getLocationPermission() {
    Log.d(TAG,  msg: "getting location permissions");
    String[] permissions = {
            Manifest.permission.ACCESS_FINE_LOCATION,
            Manifest.permission.ACCESS_COARSE_LOCATION
    };
    /* First, Check for Fine Location */
    if (ContextCompat.checkSelfPermission(this.getApplicationContext(),
            Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
        /* Secondly, Check that Course Location also allowed */
        if (ContextCompat.checkSelfPermission(this.getApplicationContext(),
                Manifest.permission.ACCESS_COARSE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
            // All permissions granted, can start the map.
            locationPermissionGranted = true;
            initMap();
        } else {
            // Request for the Location Permissions needed to run the app.
            ActivityCompat.requestPermissions( activity: this, permissions, LOCATION_PERMISSION_REQUEST_CODE);
        }
    } else {
        // Request for the Location Permissions needed to run the app.
        ActivityCompat.requestPermissions( activity: this, permissions, LOCATION_PERMISSION_REQUEST_CODE);
    }
}
```

Fig. 17. Location permission code

vector from the user to each restaurant and the vector for the direction the user is facing. We then convert each vector to polar coordinates measuring from North. We then calculate the difference in angles. This resulting angle is used to position the restaurant in the AR view.

E. ARCore

For ARCore, we also incorporated Androids supplementary Sceneview library. Out of the box, ARCore works by using the camera to find meaningful points (features) and creating a coordinate system relative to your position. By tracking the movement of the features as well as using the sensors of the phone, it can build an understanding of your current environment. Sceneview organizes the objects on an augmented reality plane into relative nodes. There are, generally, two types of nodes: Anchors and Ordinary. Anchor nodes locks its position to a feature (point) recognized by ARCore. This means that regardless of your orientation or movement of your camera, as long as that feature can be re-recognized, the positioning of that node will stay fixed. Ordinary nodes are placed relative to each other (e.g. left by 30cm of another node), and as a result can be placed relative to an anchor node.

For our implementation, we place an anchor node on the exact position you are standing, and then position the restaurant cards relative to your orientation as shown in Figure 18.

ARCore generates objects to place in the worldspace on the fly. For us, it converts a 2D view (a layout resource file called $ar_restaurant_card.xml$) and places it in a 3D space. We first program this card in and make a readable display for the user to view in a 3D world. Then we render this in and set it to be the display for a given node. You can see our rendering logic in Figure 19.

Since it is a 2D view, we want to make sure the information stays facing you on every frame, no matter how you move. To do this we used the onUpdate method to update the rotation based on the 3D world space, shown in Figure 20.

In this method, we first get the location of the camera in the world position (what ARCore reads as our world coordinate map). Then we get the position of the card relative to the world as well as we mentioned in (Section 3.D). We change the directional vector into a quaternion (a 3D representation of a rotation, since a rotation normally represents a 2D place, a quaternion is for all three directions). Then with the quaternion, we can update the rotation of the card to keep facing us and stay readable.

F. UI Libraries

- For making use of the menu resource in order to present a list of actional buttons in Main Activity, we use io.github.yavski:fab-speed-dial:1.0.6 and com.android.support:design:28.0.0-alpha3
- For implementing swipeable cards both in Main Activity and AR mode, we use com.android.support:cardview-v7:28.0.0 and com.github.devlight:infinitecycleviewpager:1.0.2
- For implementing RecyclerView in Favorite, we use com.android.support:recyclerview-v7:28.0.0

IV. Conclusion

Augmented reality has many potential far-reaching areas of use in any industry. It is still in development and tech companies are working to improve it for their mobile apps. AURI is currently an android-based app that shows where nearby restaurants are in the real-world by using AR technology. Throughout this course and the development of this app, we learned several Android app development techniques as well as acquired our first experience in developing an AR mobile app. Even though AURI is currently functional with all the extensive features integrated, there is still a some ways to go before AURI can be released out as a final product. Section 4.A will touch upon the things we would do differently, improvements, and interesting extensions for future work.

A. Interesting Extensions

- Integrate filters so that user can set the type of places they want to show on map or camera, not limit only in restaurant, but can be museum, hotel, etc.
- Apply openCV or machine learning techniques to our app, then our AR mode can determine if the camera is facing a door or something not related to restaurants.
- In AR mode, if the user wants to walk to a restaurant, a route to the this restaurant will show on the screen, which is more intuitive than using google map.
- Integrate other APIs (eg. Yelp, TripAdvisor, FourSquare, etc) to collect more food options and reviews. This can provide our users more choices to choose.
- Have a button for users to click and share their favorite places to their friends through social media. This may increase the number of users using our app and it will be a great fun with social attributes.
- Support other languages for users cant read English fluently.
- Currently, the radius of nearby restaurants is hard-coded. In the future, this can be changed so that our users can customize the radius.
- Login and register an account with either Google or Facebook account.
- Support users to add reviews and rating to the restaurants.

ACKNOWLEDGMENT

This project is made for the course CS591: Mobile Application Development at Boston University in Fall 2018. We would like to thank Professor Shereif El-Sheikh for teaching the class and supporting our project's development

```
// 1. First set the anchor at the current location and orientation of the camera.
try {
    //Get camera pose to update where we are facing
    Pose newPose = arSceneView.getArFrame().getCamera().getPose();
    Anchor anchor = arSceneView.getSession().createAnchor(newPose);

    // Remove all existing elements
    deleteAllCards();

    // Update global anchor
    anchorNode = new AnchorNode(anchor);
    anchorNode.setParent(arSceneView.getScene());
```

Fig. 18. ARCore: creating a scene

```
public void onActivate() {
    if (getScene() == null) {
        throw new IllegalStateException("Scene is null.");
    }

    ViewRenderable.builder()
            .setView(context, restaurantBucket)
            .build()
            .thenAccept(
                    (renderable) -> {
                        this.setRenderable(renderable);
                    })
            .exceptionally(
                    (throwable) -> {
                        throw new AssertionError( message: "Could not load restaurant card.", throwable);
                    });
}
```

Fig. 19. ARCore: Making the renderable

```java
@Override
public void onUpdate(FrameTime frameTime) {
    // If onUpdate is called explicitly or if the node is removed from the scene on a
    // different thread during onUpdate, then getScene may be null.
    if (getScene() == null) {
        return;
    }

    Vector3 cameraPosition = getScene().getCamera().getWorldPosition();
    Vector3 cardPosition = getWorldPosition();
    Vector3 direction = Vector3.subtract(cameraPosition, cardPosition);
    Quaternion lookRotation = Quaternion.lookRotation(direction, Vector3.up());
    setWorldRotation(lookRotation);
}
```

Fig. 20. ARCore: Making the renderable